National University of Computer and Emerging Sciences

Islamabad Campus

Parallel and Distributed Computing (CS3006)

Course Instructor(s):

Dr. Muhammad Arshad Islam, Muhammad Farrukh Bashir, Muhammad Aadil Ur Rehman, Fahad Shafique

Section(s): A,B,C,D,E,F,G,H,J,K

Sessional-II Exam

Total Time (Hrs): 1

Total Marks: 55

Total Questions: 4

Date: Apr 7, 2025

_____ _____ _____

Roll No **Course Section** **Student Signature**

Do not write below this line.

Attempt all the questions.

[CLO 1: Demonstrate understanding of various concepts involved in parallel and distributed computer architectures.]

Q1: Multiple choice questions. **[5x1 & 10x2 = 25 marks]**

1) Consider the machine file with the following configuration for MPI:

```
node1:4
node2:2
node3:3
```

Assume an MPI program is launched using the following command, choose the correct order of process spawning across these nodes?

```
mpirun -np 15 -machinefile machinefile myprogram.
```

a) 4 processes on node1, 2 processes on node2, 9 processes on node3.
b) 3 processes on node1, 2 processes on node2, 10 processes on node3.
c) 6 processes on node1, 2 processes on node2, 7 processes on node3.
d) 5 processes on node1, 3 processes on node2, 7 processes on node3.
e) Any random order

2) What is the main advantage of OpenCL over CUDA?
a) OpenCL supports all programming languages
b) OpenCL is more efficient for multi-threaded CPU programming
c) OpenCL supports a wide range of devices, including CPUs, GPUs, and FPGAs
d) OpenCL has a simpler programming model
e) None of above

3) Source code in an OpenMP that is not covered by a pragma is executed by:
a) All threads
b) Single thread
c) Error, it must be specified with thread id
d) None of the above

4) Which of the following loop scheduling in OpenMP has the most overhead:
a) Static
b) Dynamic
c) It does not support loop scheduling
d) #pragma omp for
e) Guided

5) In OpenCL, which of the following is a key difference between buffers and images?

a) Buffers store 1D or 2D data, while images store 2D or 3D data.

b) Buffers are accessed directly via global IDs, while images are accessed using texture coordinates.

c) Buffers support only int and float types, while images support half, float, and double.

d) All of the above are true.

e) Only a) and b) are true.

6) What will be the output of the following code:

```
int i;
int j;
int n=10;
int t=0;
omp_set_num_threads(2);

#pragma omp parallel private(j)
{
      # pragma omp for
      for (i=0; i<n; i++) {
            printf("a\n");
      }
      for(j=0; j<n; j++) {
            printf("b\n");
      }
}
```

a) 10 a's and 10 b's

b) 20 a's and 20 b's

c) 10 a's and 20 b's

d) 20 a's and 10 b's

e) Cannot be determined

7) Consider the following OpenMP code and determine the execution behavior of the loop:

```
omp_set_num_threads(10);
int i;
#pragma omp parallel for private(i)
for (i=1; i<99; i++)
      A[i+1] = A[i-1] + 20;
```

a) Serial Execution

b) Parallel Execution

c) Runtime Error

d) Master thread only

e) Syntax Error

8) What will be the output of the following OpenMP code:

```
omp_set_num_threads(4);
int n=1, x=10;
#pragma omp parallel for lastprivate(n)
For(i=1; i<x; i++)
     If(x/2 == i)
         n=i;
printf("%d", n);
```

a) 1
b) 10
c) 9
d) 5
e) None of the above

9) What will be printed by process 0, considering total 4 processes?

```
int rank, data[5] = {10,20,30,40,50};
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
if (rank == 0) data[0] = 5;
MPI_Bcast(&data, 1, MPI_INT, 2,
MPI_COMM_WORLD);
if (rank == 0) printf("Rank 0: %d\n", data);
```

a) `Rank 0: undefined`
b) `Rank 0: 10`
c) `Rank 0: 5`
d) `Rank 0: 0`
e) Segmentation Fault

10) What will be the output of the following code, considering total 4 processes?

```
int rank, send = 10, recv = 0;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
send = rank;
MPI_Reduce(&send, &recv, 1, MPI_INT, MPI_MAX,
1, MPI_COMM_WORLD);
if(rank == 0) printf("Recv: %d\n", recv);
```

a) `Recv: 10`
b) `Recv: 0`
c) `Recv: garbage`
d) `Recv: 3`
e) Segmentation Fault

11) What will be the output of the following code, considering total 4 processes?

```
int rank;
int send = 1;
int recv;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Allgather(&send, 1, MPI_INT, &recv, 1,
MPI_INT, MPI_COMM_WORLD);
if(rank == 0) printf("Recv: %d\n", recv);
```

a) `Recv: 3`
b) `Recv: 1`
c) `Recv: garbage`
d) `Compile-time error`
e) Runtime error

12) What happens in this kernel execution, considering Global size = 2. Initial A[0] = 0.?

```
__kernel void test(__global int* A) {
    int id = get_global_id(0);
    if (id == 1)
        A[0] = 10;
    else
        A[0] = 5;
}
```

a) A[0] = 10
b) A[0] = 5
c) A[0] = 0
d) A[0] = non-deterministic
e) Compile-time error

13) What will be the output of the following OpenCL kernel when executed with global size = 3 and local size = 2?

```
__kernel void check() {
    int gid = get_global_id(0);
    int lid = get_local_id(0);
    int group = get_group_id(0);

    int temp = gid * 2 + lid + group;

    if (lid == 2)
        printf("Temp: %d\n", temp);

    // This will help us confirm thread count
    // printf("GID: %d, LID: %d, Group:
%d\n", gid, lid, group);
}
```

a) Temp: 6
b) Temp: 4
c) Temp: 5
d) Temp: 10
e) No output

14) In the given MPI program what is the value of the global product printed by process 0 when the program is executed with **3 processes**?

```
int rank, size, n = 12, buffer[12], result[4];
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);

if (rank == 0) {
    for (int i = 0; i < n; i++) {
        buffer[i] = 2 * (i % 2) + 1;
    }
}
```

```
MPI_Scatter(buffer, n / size, MPI_INT, result, n / size,
MPI_INT, 0, MPI_COMM_WORLD);
    int local_product = 1;
    for (int i = 0; i < n / size; i++) {
        local_product *= result[i];
    }
int global_product;
MPI_Reduce(&local_product, &global_product, 1, MPI_INT,
MPI_PROD, 0, MPI_COMM_WORLD);
if (rank == 0) {
    printf("Global product: %d\n", global_product);
}
```

a) `Global product: 24`
b) `Global product: 32`
c) `Global product: 64`
d) `Global product: 96`

15) What will be the output of the following code snippet?

```
int N=4, s_data = 10, p_data = 0, M = 0, S = 0, L = -1;
omp_set_num_threads(5);
#pragma omp parallel num_threads(3) firstprivate(s_data)  shared(M, S)
for (int i = 0; i < N; i++) {
  int thread_id = omp_get_thread_num();
  #pragma omp master
  {
    M += s_data + i;
  }
  #pragma omp single
  {
    S++;
  }
  s_data += (thread_id + i);
  L = thread_id;
}

printf("M: , ", M);
printf("S: , ", S);
printf("s_data: %d\n", s_data);
```

a) M: 50, S: 4, S_data: 10
b) M: 50, S: 12, S_data: 10
c) M: 50, S: 4, S_data: 16
d) M: 46, S: 4, S_data: 10
e) None of above

[CLO 2: Implement different parallel and distributed programming paradigms and algorithms using Message-Passing Interface (MPI) and OpenMP]

Q2: Write output of the following program. **[10 Marks]**

```
#define NUM_THREADS 4
#define ITERATIONS 2

float *create_nums(int num_elements, int rank) {
  float *nums_array = (float *)malloc(sizeof(float) *
num_elements);

  for (int i = 0; i < num_elements; i++) {
    nums_array[i] = rank + i * 0.1;
  }
  return nums_array;
}

int main(int argc, char** argv) {

  MPI_Init(NULL, NULL);
  int rank, size;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  float *nums_array = create_nums(num_elements_per_proc, rank);

  printf("Process %d - nums_array: ", rank);
  for (int i = 0; i < num_elements_per_proc; i++) {
    printf("%f ", nums_array[i]);
  }
  printf("\n");

  float sum = 0;
  for (int i = 0; i < num_elements_per_proc; i++) {
    sum += nums_array[i];
  }
  printf("Process %d - sum: %f\n", rank, sum);

  float total_sum;
  MPI_Allreduce(&sum, &total_sum, 1, MPI_FLOAT, MPI_SUM,
MPI_COMM_WORLD);
  float mean = total_sum / (num_elements_per_proc * size);

  printf("Process %d - mean: %f\n", rank, mean);
  float sq_diff = 0;
  for (int i = 0; i < num_elements_per_proc; i++) {
    sq_diff += (nums_array[i] - mean) * (nums_array[i] - mean);
  }
  printf("Process %d - sq_diff: %f\n", rank, sq_diff);
```

```
  float total_sq_diff;
  MPI_Reduce(&sq_diff, &total_sq_diff, 1, MPI_FLOAT, MPI_SUM, 0,
MPI_COMM_WORLD);

  if (rank == 0) {
    float stddev = sqrt(total_sq_diff / (num_elements_per_proc *
size));
    printf("Final Mean = %f, Standard Deviation = %f\n", mean,
stddev);
  }

  free(nums_array);

  MPI_Barrier(MPI_COMM_WORLD);
  MPI_Finalize();
}
```

If we consider, num_elements_per_proc=3 (for other assumptions you can calculate likewise), then following will be output:

Process 0 - nums_array: 0.0 0.1 0.2
Process 1 - nums_array: 1.0 1.1 1.2
Process 2 - nums_array: 2.0 2.1 2.2
Process 3 - nums_array: 3.0 3.1 3.2

Process 0 - sum: 0.30
Process 1 - sum: 3.30
Process 2 - sum: 6.30
Process 3 - sum: 9.30

Process 0 - mean: 1.60
Process 1 - mean: 1.60
Process 2 - mean: 1.60
Process 3 - mean: 1.60

Process 0 - sq_diff: 3.48
Process 1 - sq_diff: 0.48
Process 2 - sq_diff: 0.48
Process 3 - sq_diff: 3.48

Final Mean = 1.600000, Standard Deviation = 0.836660

[CLO 1: Demonstrate understanding of various concepts involved in parallel and distributed computer architectures.]

Q3: Below is an OpenCL kernel that computes the degree of each node in an undirected graph using adjacency matrix. This kernel is invoked using following command

```
    clEnqueueNDRangeKernel(queue, kernel, 1, NULL,
&global_work_size, NULL, 0, NULL, NULL);
```

[2+3+5=10 Marks]

```
__kernel void computeDegree(__global int *adjMatrix, __global int
*degree, int numNodes) {
    int id = get_global_id(0); // Get the index of the current
node

    if (id < numNodes) {
        for (int j = 0; j < numNodes; j++) {
            degree[id] += adjMatrix[id * numNodes + j];
        }
    }
}
```

a) Identify potential performance issue in this kernel. Discuss briefly inefficiency related to memory access.

Global Memory Access Latency:
Each work-item reads data[id] from global memory, computes the square, and writes it back. Global memory is slow, and frequent accesses create bottlenecks.

No Work-Group Optimization:
All computations occur at the global memory level, missing potential optimizations with local memory.

b) Modify the kernel to utilize private memory for computing degree of each node for improving efficiency.

```
int sum = 0;
for (int j = 0; j < numNodes; j++) {
    sum += adjMatrix[id * numNodes + j];
}

degree[id] = sum;
```

c) Provide kernel code that utilizes local memory for buffering the given matrix with the aim of improving performance by reducing memory access time. The new kernel will be invoked using following command

```
clEnqueueNDRangeKernel(queue, kernel, 1, NULL, &global_work_size,
&local_work_size, 0, NULL, NULL);
```

```
__kernel void computeDegreeOptimized(
__global int *adjMatrix,
__global int *degree,
int numNodes
) {
    int gid = get_global_id(0);       // Global thread ID (node index)
    int lid = get_local_id(0);        // Local thread ID
    int group_size = get_local_size(0); // Size of the work-group

    __local int localRow[256];  // Adjust size based on hardware limits

    int sum = 0;

    // Load the row in chunks of `group_size`
    for (int j = 0; j < numNodes; j += group_size) {
```

```
        int index = gid * numNodes + (j + lid);  // Global index

        // Bounds check to avoid out-of-range access
        if ((j + lid) < numNodes) {
            localRow[lid] = adjMatrix[index];
        } else {
            localRow[lid] = 0;
        }

        // Synchronize to make sure all work-items have written to localRow
        barrier(CLK_LOCAL_MEM_FENCE);

        // Accumulate the values from the shared chunk
        for (int k = 0; k < group_size && (j + k) < numNodes; k++) {
            sum += localRow[k];
        }

        barrier(CLK_LOCAL_MEM_FENCE);
    }

    // Write the final degree to global memory
    degree[gid] = sum;
}
```

[CLO 2: Implement different parallel and distributed programming paradigms and algorithms using Message-Passing Interface (MPI) and OpenMP.]

Q4: A company requires a simple console-based utility to find the brightest pixel in a grayscale image. The grayscale image is already formatted as a 2D array of pixels, where each pixel has an intensity value ranging from 0 (black) to 255 (white). When the image is provided, the utility must output two things:

1. The brightest pixel in the entire image.

2. The brightest pixel on the border of the image (i.e., the first row, last row, and the first and last columns of the image).

The company has some multicore machines having different architectures where this utility will be executed. Utility must process the image using multithreading with OpenMP, adhering to the following guidelines:

- The number of threads should be decided by the user.
- Each thread will process a row-wise subset of the image.
- Each thread will find the brightest pixel in its assigned region and the brightest pixel among the border pixels of its assigned region.

The basic skeleton code is already provided. Your task is to understand the code and complete it by adding the necessary code to implement the functionality.

Note: Please only write the missing code on the answer sheet with proper labels as provided here. Do not write the whole code again. Assume all required libraries are included. [10 marks]

```
int main() { int IMAGE_ROWS = 5,IMAGE_COLS = 10
    int image[IMAGE_ROWS][IMAGE_COLS] = {
        {10, 20, 30, 40, 50, 60, 70, 80, 90, 100},
        {110, 120, 130, 140, 150, 160, 170, 180, 190, 200},
        {210, 220, 230, 240, 250, 235, 245, 235, 225, 215},
        {205, 195, 185, 175, 165, 155, 145, 135, 125, 115},
        {105, 95, 85, 75, 65, 55, 45, 35, 25, 15}};
```

```
    int brightest_pixel = brightest_border_pixel = local_max = 0;
    int local_border_max = 0, num_threads;
    printf("Enter the number of threads: ");
    scanf("%d", &num_threads);

    // A. Write the missing code here
_____________________________________________________________________

    // B. Write the missing clauses
    #pragma omp parallel _____________________________
    {
        // C. Write the best suited scheduling scheme
        #pragma omp for ____________________________________________
        for (int i = 0; i < IMAGE_ROWS; i++) {
            for (int j = 0; j < IMAGE_COLS; j++) {
                // Update the brightest pixel in the entire image
                if (image[i][j] > local_max) {
                    local_max = image[i][j];
                }
                // D. Update the brightest pixel on the border
                if (_____________________________________________________
                    _____________________________________________________
                    )
                {
                    if (image[i][j] > local_border_max) {
                        local_border_max = image[i][j];
                    }
                }
            }

            // E. Update the global maximum values
            ____________________________________________________________
            ____________________________________________________________
            ____________________________________________________________
            ____________________________________________________________
            ____________________________________________________________
            ____________________________________________
        }
    }
    printf("Brightest pixel: %d\n", brightest_pixel);
    printf("Brightest border pixel: %d\n", brightest_border_pixel);
    return 0;
}
```

A. `omp_set_num_threads(num_threads);`

B. `shared(brightest_pixel, brightest_border_pixel) private(local_max, local_border_max)`

C. `schedule(dynamic)`

D. `if (i == 0 || i == IMAGE_ROWS - 1 || j == 0 || j == IMAGE_COLS - 1)`

E.
```
#pragma omp critical
    {
        if (local_max > brightest_pixel) {
```

```
                brightest_pixel = local_max;
            }
            if (local_border_max > brightest_border_pixel) {
                brightest_border_pixel = local_border_max;
            }
        }
```