National University of Computer and Emerging Sciences

Islamabad Campus

Parallel and Distributed Computing (CS3006)

Course Instructor(s):

Dr. Muhammad Arshad Islam, Mr. Adil Rehman, Mr. Farrukh Bashir, Mr. Fahad Shafique

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

Final Examination

Total Time (Hrs): 3

Total Marks: 145

Total Questions: 9

Date: May 19, 2025

_____ _____ _____

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

Do not write below this line.

Attempt all the questions.

Part A (To be solved on Question Paper)

[CLO1: Demonstrate understanding of various concepts involved in parallel and distributed computer architectures.]

Q1 : Answer the following conceptual questions **[5+5 marks]**

a. A data analytics company is processing large batches of information stored as 2D matrices. One of the common tasks involves multiplying a matrix of size N x N with a vector of size N. A developer has written a parallel version of the matrix-vector multiplication function, as shown below. You are part of the performance optimization team and have been asked to evaluate how effectively the five parallelization strategies are applied to this problem. Clearly **name** and **apply** each of the five parallelization strategies to this problem**.** Explain how each strategy is reflected in the code**. [5 Marks]**

```
 void matvec_mul_parallel(int N, float* mat, float* vec, float*
result) {
    #pragma omp parallel for
    for (int i = 0; i < N; i++) {
    result[i] = 0;
    for (int j = 0; j < N; j++) {
        result[i] += mat[i * N + j] * vec[j];
    }
    }
}
```

1. Problem Understanding

We want to compute matrix-vector multiplication, which is common in many scientific applications. The computation for each result[i] is **independent** from others — this makes it a good candidate for parallelism.

2. Decomposition (Partitioning)

Decompose by **rows of the matrix**:

- Each row i leads to the calculation of result[i].
- So we can assign the task of computing each result[i] to a different thread or process.

3. Assignment

Assign:

- One thread/process to each chunk of rows.
- For example, in a 4-thread system and N = 100, thread 0 computes rows 0–24, thread 1 computes 25–49, and so on.

4. Orchestration

Orchestrate using threads (e.g., std::thread in C++ or OpenMP). Ensure:

- No two threads write to the same index of result[].
- Synchronization is not needed for writes, as each thread handles its own section.

5. Mapping

- If using **OpenMP**, use #pragma omp parallel for over the outer loop.
- If using **C++ threads**, launch multiple threads, each computing a portion of the matrix rows.
- If using **MPI**, distribute rows across processes, send required parts of vec[], and gather the final result[].

b. Consider a program that processes a matrix row-by-row in parallel.

I. Describe a scenario where **barrier synchronization** would be necessary.

Barrier synchronization is necessary when all parallel threads or processes must reach a certain point in the program before any can proceed.

Consider a program that performs a matrix computation in multiple phases, where each phase depends on the complete results of the previous one. For example, if each thread processes a row of a matrix in a stencil operation and the computation for time step t+1 depends on the results of all rows from time step t, then a barrier is needed after each time step. This ensures no thread moves on to the next step before all have finished the current one.
)

II. Describe a case where **lock-based synchronization** is preferable over barriers.

Lock-based synchronization is preferable when only certain parts of the data require mutual exclusion, and you want fine-grained control over access without forcing all threads to wait.

Suppose multiple threads are updating a shared hash table or global result array, but only some keys or indexes overlap. Using locks per key or per row allows safe concurrent access without stopping other threads unnecessarily, unlike a barrier which would block all.

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

Q2.1: part (a-e) Below is given the codes for OpenMP programs. Determine the output, by considering codes as error free **[5+5+5+5=20 Marks]**

a)

```
int calculate_offset(int tid) {
    return (tid % 2 == 0) ? 5 : 10;
}
int main() {
    int base = 100;
    int factor = 2;

    #pragma omp parallel num_threads(4) reduction(+:base)
firstprivate(factor)
    {
        int tid = omp_get_thread_num();
        int offset = calculate_offset(tid);
        base = (base - (tid * offset)) * factor;
    }

    printf("\nFinal reduced value: %d\n", base);
    return 0;
}
```

Output:

Final reduced value: -100

b)

```
int main() {
    int j, num, flag, rem, res;
    #pragma omp parallel num_threads(4) private(num, res, rem)
    {
        int i = 2538;
        rem = 0, res = 0;
        i += omp_get_thread_num();
        num = i;

        while (num != 0) {
            rem = num % 10;
            res += (rem * rem * rem * rem);
            num /= 10;
        }

        if(!omp_get_thread_num())
            printf("number= %d \n", res);
    }
    return 0;
}
```

Output:

number = 4818

c)

```
int main() {
    int x = 5, y = 10, z = 15;
    int result = 0;

    #pragma omp parallel for default(none) firstprivate(x)
private(y) lastprivate(z) shared(result) num_threads(4)
    for (int i = 0; i < 4; i++) {
        y = x + i;
        z = y * 2;

        #pragma omp critical
        {
            result += z - (i % 2 ? x : y);
        }
    }
    printf("x=%d ", x);
    printf("y=%d ", y);
    printf("z=%d ", z);
    printf("result=%d ", result);    }
```

Output:

x=5
y=10
z=16
result=30

d)

```
int main() {
    int i, A[8] = {1,2,3,4,5,6,7,8};
    int sum1 = 0, sum2 = 0;

    omp_set_num_threads(4);


    #pragma omp parallel for schedule(dynamic, 2)
    for (i = 0; i < 8; i++) {
        #pragma omp critical
        sum1 += A[i] + omp_get_thread_num();
    }

    #pragma omp parallel for schedule(static, 2)
    for (i = 7; i >= 0; i--) {
        #pragma omp critical
        sum2 += A[i] - omp_get_thread_num();
    }

    printf("sum1 = %d, sum2 = %d\n", sum1, sum2);
    return 0;
```

```
}
```

Output:

sum1 = 48, sum2 = 24

Q2.2: Following is given the openMP code with the output . You have to write down the missing code by showing how each variable will behave for the given pragma **[5 marks]**

```
int main() {
    int a = 5, b = 10;
    int sum = 0;
    int tid;

    #pragma omp parallel num_threads(3)_____________________________
    {
      tid = omp_get_thread_num();
      a = a + tid + 5;
      b = b - tid;
      sum += a;

      printf("Thread %d: a=%d, b=%d, sum=%d\n", tid, a, b, sum);
    }

    printf("\nFinal values:\n");
    printf("a=%d\n", a);
    printf("b=%d\n", b);
    printf("sum=%d\n", sum);

    return 0;
}
```

Output:

```
Thread 0: a=5, b=10, sum=1
Thread 1: a=6, b=9, sum=2
Thread 2: a=7, b=8, sum=3

Final values:
a=7
b=8
sum=6
```

#pragma omp parallel num_threads(3) private(tid) lastprivate(a) shared(b) reduction(+:sum)

Q3.1: For each of the following code segments (Parts a–e), indicate whether it is suitable for parallelization. Provide a brief justification for your answer in each case **[2 +2 +2 +2 +2=10 Marks]**

a.

```
#pragma omp parallel for
for (i = 0; i < n; i++) {
    a[i] = foo(i);
```

```
    if (a[i] < b[i]) break;
}
```

Justification:

Not Suitable , because of break statement

b.

```
dotp = 0;
#pragma omp parallel for
for (i = 0; i < n; i++)
    dotp += a[i] * b[i];
```

Justification:

Not Suitable , because of shared value of dotp

c.

```
#pragma omp parallel for
for (i = 0; i < n; i++) {
    a[i] = foo(i);
    if (a[i] < b[i]) a[i] = b[i];
}
```

Justification:

Suitable

d.

```
flag = 0;
#pragma omp parallel for
for (i = 0; (i < n) && (!flag); i++) {
    a[i] = 2.3 * i;
    if (a[i] < b[i])
        flag = 1;
}
```

Justification:

Not Suitable, because value of flag can be set to 1 by any other thread

e.

```
#pragma omp parallel for
for (i = k; i < n; i++)
    a[i] = b * a[i - k];
```

Justification:

Suitable, in case if k=0. But not suitable for k>0

Q3.2. The below loop performs a sequential computation where each iteration depends on the result of the previous one. Since the value of a[i] is derived from a[i-1], there is a loop-carried dependency that prevents straightforward parallelization.**[5 marks]**

```
a[0] = 0;
for (i = 1; i < n; i++){
     a[i] = a[i-1] + i;
     printf("Iteration %d:= %d\n", i, a[i]);
}
```

How can you modify this loop with the aim of parallelizing it using above OpenMP pragma parallel for without changing the meaning of the program? Hint= note that this loop is initializing an array.

```
#pragma omp parallel for
    for (int i = 0; i < n; i++) {
        a[i] = (i * (i + 1)) / 2;
    }
```

Question 4. You are given a large array of integers. Your task is to design an OpenCL kernel that partitions this array into two separate arrays based on a given threshold value:**[15 marks]**

- All values **less than or equal to** the threshold should be placed in the first output array.
- All values **greater than** the threshold should be placed in the second output array.
- The number of elements in each output array should be dynamically determined during execution.
- You may assume the threshold and input size are provided as kernel arguments.
- Use atomic operations to safely manage parallel writes into the output arrays.
- You may assume the output arrays are pre-allocated to the size of the input array

```
__kernel void partition_array(__global const int* input,
                              __global int* out_a,
                              __global int* out_b,
                              __global int* index_a,
                              __global int* index_b,
                              const int threshold,
                              const int size) {
    int gid = get_global_id(0);

    if (gid >= size) return;

    int val = input[gid];

    if (val <= threshold) {
        int pos = *index_a++;
        out_a[pos] = val;
    } else {
```

```
            int pos = *index_b++;
            out_b[pos] = val;
        }
    }
```

Question 5: You are given a 2D matrix of integers, initialized at process 0. The number of rows in the matrix is greater than the total number of MPI processes (world_size). Each process is assigned one row , specifically, the row with index equal to its rank**.[3+1+1+2+4+9 marks]**

Each process should:

1. Receive its corresponding row.
2. Compute the sum of the row.
3. If the row sum % world_size equals the rank, then that process is guilty and must also compute the total sum of all remaining rows of the matrix as well as penalty.

```
int main(int argc, char *argv[]) {
    int rank, world_size;
    int matrix[ROWS][COLS];

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    int row_buffer[COLS];

    if (rank == 0) {
        // Initialize matrix with some values
        for (int i = 0; i < ROWS; i++)
            for (int j = 0; j < COLS; j++)
                matrix[i][j] = i + j;
        // TODO:1 Send each row to the matching rank

 for (int i = 1; i < world_size && i < ROWS; i++)
            MPI_Send(matrix[i], COLS, MPI_INT, i, 0,
MPI_COMM_WORLD);
```

```
        // Copy row 0 for self
        for (int j = 0; j < COLS; j++)
            row_buffer[j] = matrix[0][j];
    } else {
        // TODO:2 Receive row
MPI_Recv(row_buffer, COLS, MPI_INT, 0, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);




    }
    // TODO3: Compute row sum and print it

for (int j = 0; j < COLS; j++)
        row_sum += row_buffer[j];
    printf("Process %d has row sum = %d\n", rank, row_sum);



    // TODO4: identify if the condition is meet for any process

 int guilty = (row_sum % world_size == rank) ? 1 : 0;

    // Report to rank 0 whether guilty
    MPI_Gather(&guilty, 1, MPI_INT, NULL, 0, MPI_INT, 0,
MPI_COMM_WORLD);


int all_guilties[world_size];
    if (rank == 0) {
        all_guilties[0] = guilty;
        for (int i = 1; i < world_size; i++) {
            MPI_Recv(&all_guilties[i], 1, MPI_INT, i, 1,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        }
        for (int i = 0; i < world_size; i++) {
            if (all_guilties[i]) {
                guilty_rank = i;
                break;
            }
        }
```

// TODO5: send the remaining rows to guilty process. If condition is not meet then nothing to do.

```
        if (guilty_rank != 0) {
            // Send remaining rows (except guilty's) to guilty_rank
            for (int i = 0; i < ROWS; i++) {
                if (i != guilty_rank) {
                    MPI_Send(matrix[i], COLS, MPI_INT, guilty_rank,
i, MPI_COMM_WORLD);
                }
            }
        }
    } else {
        MPI_Send(&guilty, 1, MPI_INT, 0, 1, MPI_COMM_WORLD);
    }
```

// TODO6: the guilty process receives the data and computes the sum of all remaining rows.

```
 if (rank == guilty_rank) {
        int total_sum = 0;
        if (rank != 0) {
            // Receive rows from rank 0
            for (int i = 0; i < ROWS; i++) {
                if (i != rank) {
                    MPI_Recv(row_buffer, COLS, MPI_INT, 0, i,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                    for (int j = 0; j < COLS; j++)
                        total_sum += row_buffer[j];
                }
            }
        } else {
            // Rank 0 computes sum locally
            for (int i = 0; i < ROWS; i++) {
                if (i != 0) {
                    for (int j = 0; j < COLS; j++)
                        total_sum += matrix[i][j];
                }
            }
        }
```

```
        printf("Guilty Process %d computed sum of remaining rows =
%d\n", rank, total_sum);


    MPI_Finalize();
    return 0;
}
```

Part B to be solved on Answer Book

[CLO3: Perform analytical modelling, dependence, and performance analysis of parallel algorithms and programs.]

Question6: Consider the following nested loop operating on a 2D array A. **[6+6+8 marks]**

a. Identify all types of data dependences present in the loop. For each dependence, specify the source and sink statements, and the variables involved

- True dependence (Read after Write)
- Anti-dependence (Write after Read)
- Output dependence (Write after Write)

```
for (i = 1; i < N; i++) {
    for (j = 1; j < N; j++) {
        A[i][j] = A[i-1][j];
        A[i][j] = A[i][j] + A[i][j-1];
        A[i][j] = A[i][j] + A[i][j];
    }
}
```

b. For each dependence identified in Part a:

I. Determine whether it is loop-carried or loop-independent
II. Indicate which loop (i or j) carries the dependence (if applicable)

c. Calculate the direction vectors and distance vectors for each dependence.

1. True Dependence (Read after Write - RAW)

Source → Sink	Variable	Explanation
S1 → S2	A[i][j]	S2 reads value written by S1
S2 → S3	A[i][j]	S3 reads value written by S2
S2 → S2	A[i][j-1]	S2 of (i,j) reads A[i][j-1] written in previous iteration (j-1)

2. Output Dependence (Write after Write - WAW)

Source → Sink	Variable	Explanation
S1 → S2	A[i][j]	Both write to same location in same iteration
S2 → S3	A[i][j]	Again, multiple writes to same memory location

Dependence	Type	Carried?	Loop
S1 → S2 (A[i][j])	WAW	**Loop-independent**	—
S2 → S3 (A[i][j])	WAW	**Loop-independent**	—
S1 → S2 (A[i][j])	RAW	**Loop-independent**	—
S2 → S3 (A[i][j])	RAW	**Loop-independent**	—
S1 → S1 (A[i-1][j])	RAW	**Loop-carried**	i
S2 → S2 (A[i][j-1])	RAW	**Loop-carried**	j

Direction and distance vectors compare source and sink iteration points:

Notation:

- Direction Vector: (i_dir, j_dir)
- Distance Vector: (i_dist, j_dist)

Source → Sink	Type	Iterations	Direction Vector	Distance Vector
S1(i,j) → S2(i,j)	RAW	Same iteration	(=, =)	(0, 0)
S2(i,j) → S3(i,j)	RAW	Same iteration	(=, =)	(0, 0)
S1(i,j) → S2(i,j)	WAW	Same iteration	(=, =)	(0, 0)
S2(i,j) → S3(i,j)	WAW	Same iteration	(=, =)	(0, 0)
S1(i−1,j) → S1(i,j)	RAW	Across i	(<, =)	(1,0)
S2(i,j-1) → S2(i,j)	RAW	Across j	(=, <)	(0, 1)

Question 7: Consider the following code **[5+5+5 Marks]**

```
for (i = 2; i <= 1000000; i += 2) {
    A[i] = B[i] + C[i];
    D[i] = A[i] * 2;
    E[i] = D[i] - C[i];
}
```

a. Show the normalize loop and explain how you adjusted the indexing of array accesses.

```
for (i1 = 0; i1 < 5; i1++) {
    int i = 2 * i1 + 2;
    A[i] = B[i] + C[i];
    D[i] = A[i] * 2;
    E[i] = D[i] - C[i];
}
```

b. Apply loop distribution to split the normalized loop into separate loops, each containing a single statement.

```
for (i1 = 0; i1 < 500000; i1++) {
    int i = 2 * i1 + 2;
    A[i] = B[i] + C[i];
}
for (i1 = 0; i1 < 500000; i1++) {
    int i = 2 * i1 + 2;
    D[i] = A[i] * 2;
}
for (i1 = 0; i1 < 500000; i1++) {
    int i = 2 * i1 + 2;
    E[i] = D[i] - C[i];
}
```

c. Explain one potential advantage and one potential disadvantage of loop distribution in this case.

Question 8.1: Given the following results:[**5 +5 marks]**

System	Clock Speed (GHz)	Cores	Efficiency (%)	Execution Time (s)
heer	3.0	4	80	10
ranjha	2.5	8	60	6

a. Calculate the speedup of system B over A.

\frac{10}{6} = 1.67 \times

(b) Per-core efficiency:

- A: 80\% \div 4 = 20\%
- B: 60\% \div 8 = 7.5\%

→ A is more efficient per core.

b. Which system is more efficient per core?

Question 8.2: A processor executes a program that consists of 2 billion instructions. The system reports the following performance metrics during execution:[5+5 marks]

- Wall-clock time (total elapsed time): 5 seconds
- CPU time (user + system time): 3.8 seconds
- System overhead time (I/O waits, context switches, etc.): 1.2 seconds
- The processor's clock rate is 2.5 GHz
- It is known that 25% of the instructions are floating-point operation

a. Calculate the MIPS rating of the processor. Clearly justify which time value you used in your calculation and why.

$$\text{MIPS} = \frac{\text{Instruction Count}}{\text{Execution Time} \times 10^6} = \frac{2 \times 10^9}{3.8 \times 10^6} = 526.31 \text{ MIPS}$$

b. Using the appropriate timing, calculate the number of FLOPS achieved.

$$\text{FLOPS} = \frac{\text{Floating Point Operations}}{\text{Time}} = \frac{5 \times 10^8}{3.8 \times 10^6} = 10^8 \text{ FLOPS} = 132.5 \text{ FLOPS}$$

Question 9 : As part of your course project, you selected one of five research papers addressing parallel solutions to graph/network-related problems. You used tools such as MPI, OpenMP, and METIS to implement and evaluate the algorithm on real-world datasets.[**3+4+3 Marks**]

a. Briefly describe the core computational challenge addressed in the research paper you implemented.
b. What were the key considerations when parallelizing the algorithm using MPI or OpenMP? Mention how data partitioning or load balancing was handled.
c. Reflecting on your experience, what was one unexpected issue you encountered during implementation or testing, and how did you resolve it?

MPI Syntax-Sheet

`int MPI_Init(int *argc, char ***argv)`	`int MPI_Comm_size(MPI_Comm comm, int *size)`
`int MPI_Finalize( )`	`int MPI_Comm_rank(MPI_Comm comm, int *rank)`
`MPI_Send(void* data, int count, MPI_Datatype type, int dest, int tag, MPI_Comm comm)`	`MPI_Recv(void* data,int count,MPI_Datatype type, int source,int tag,MPI_Comm comm,MPI_Status* status`
`int MPI_Get_count(MPI_Status* status, MPI_Datatype type, int* count)`	`int MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status *status)`
`int MPI_Isend(const void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)`	`int MPI_Irecv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request * request)`
`int MPI_Wait(MPI_Request *request, MPI_Status *status)`	`int MPI_Waitall(int count, MPI_Request array_of_requests[], MPI_Status array_of_statuses[])`
`int MPI_Waitany(int count, MPI_Request array_of_requests[], int *indx, MPI_Status * status)`	`int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)`
`int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)`	`int MPI_Scatter(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)`
`int MPI_Gather(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)`	`int MPI_Gatherv(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, const int *recvcounts, const int *displs, MPI_Datatype recvtype, int root, MPI_Comm comm)`
`int MPI_Scatterv(const void *sendbuf, const int *sendcounts, const int *displs, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)`	`int MPI_Allgather(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)`
`int MPI_Allgatherv(const void *sendbuf, int sendcount, MPI_Datatype sendtype,`	`int MPI_Alltoall(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void`