MIT-ADT UNIVERSITY
PUNE, INDIA
A leap towards World Class Education
(Established by MIT Art, Design and Technology University Act, 2015
(Maharashtra Act No. XXXIX of 2015)

MIT Art Design and Technology University

MIT School of Computing, Pune

Department of Computer Science and Engineering

Lab Report

Course- HPC Lab

Class - L.Y. (SEM-I), Core, AIEC

Name: Sourav Shailesh Toshniwal

Roll No: 2213047

A.Y. 2024 - 2025

Lab Experiment List

Sr. No.	Name of Experiment	CO
1.	Familiarization with Linux commands	CO1
2.	Familiarization with SLURM commands	CO1
3.	Write an OpenMP program to print Hello world with thread ID.	CO2
4.	Write your first Parallel Program, with which you should be able to print your NAME from 4 underline cores.	CO2
5.	Write a C program utilizing OpenMP directives to demonstrate the behavior of the private clause. ▪ The program should perform the following steps: ▪ Initialize OpenMP with 4 threads. ▪ Declare an integer variable val and initialize it to a value of 1234. ▪ Print the initial value of val outside the OpenMP parallel region. ▪ Enter an OpenMP parallel region using the omp parallel directive, with the firstprivate clause applied to the variable val. ▪ Inside the parallel region, each thread should print the current value of val, increment it by 1, and then print the updated value. ▪ Print the final value of val outside the parallel region.	CO2
6.	Write a C program utilizing OpenMP directives to demonstrate the behavior of the private clause. Steps to follow : ▪ Open text editor. ▪ write the below program in it. ▪ Save the file with .c extentation.	CO2

	▪ Compile and execuate with given commands.	
7.	Write a Parallel C program where the iterations of a loop should be scheduled statically across the team of threads. A thread should perform CHUNK iterations at a time before being scheduled for the next CHUNK of work.	CO2
8.	Write a Parallel C program which should print the series of 2 and 4. Make sure both should be executed by different threads.	CO2
9.	Write MPI Program to print "Hello World". MPI program to send and receive Hello World messages from all other processes to a Root process and print the received messages.	CO3
10.	MPI program to send two numbers (array elements) per process to a Root process and print the received messages.	CO3
11.	MPI program to find sum of first N integers using any given number of processes. Example, N=10,000 and no. of processes can be 4 or 8 or 12 etc.	CO4
12.	MPI program to find sum of n integers on in which processors are arranged in ring topology using MPI point-to-point blocking communication library calls.	CO4
13.	Write a CUDA program to perform two matrix addition.	CO5
14.	Write a CUDA program to perform two matrix multiplication.	CO5

Experiment No 3

Experiment Title: OpenMP program 1

Problem Statement: Write an OpenMP program to print Hello world with thread ID.

Source Code and Output /Screenshots:

```c
#include <stdio.h>
#include <omp.h>

int main() {
    // Start parallel region
    #pragma omp parallel
    {
        // Get the thread ID of the current thread
        int thread_id = omp_get_thread_num();

        // Print Hello World along with the thread ID
        printf("Hello, World! from thread %d\n", thread_id);
    }

    return 0;
}
```

```
Hello, World! from thread 0
Hello, World! from thread 3
Hello, World! from thread 2
Hello, World! from thread 1
```

Experiment No 4

Experiment Title: OpenMP Program 2

Problem Statement: Write your first Parallel Program, with which you should be able to print your NAME from 4 underline cores.

Source Code and Output /Screenshots:

main.c

```c
#include <stdio.h>
#include <omp.h>

int main() {
    // Set the number of threads to 4
    omp_set_num_threads(4);

    // Parallel region where each thread will print your name
    #pragma omp parallel
    {
        // Each thread prints your name
        printf("Hello, my name is Sourav! Printed by thread %d\n", omp_get_thread_num());
    }

    return 0;
}
```

input

```
Hello, my name is Sourav! Printed by thread 3
Hello, my name is Sourav! Printed by thread 2
Hello, my name is Sourav! Printed by thread 1
Hello, my name is Sourav! Printed by thread 0
```

Experiment No 5

Experiment Title: OpenMP Program 3

Problem Statement:

Write a C program utilizing OpenMP directives to demonstrate the behavior of the private clause. The program should perform the following steps:

- Initialize OpenMP with 4 threads.
- Declare an integer variable val and initialize it to a value of 1234.
- Print the initial value of val outside the OpenMP parallel region.
- Enter an OpenMP parallel region using the omp parallel directive, with the firstprivate clause applied to the variable val.
- Inside the parallel region, each thread should print the current value of val, increment it by 1, and then print the updated value.
- Print the final value of val outside the parallel region.

Source Code and Output /Screenshots:

```c
#include <stdio.h>
#include <omp.h>

int main() {
    // Initialize OpenMP with 4 threads
    omp_set_num_threads(4);

    // Declare an integer variable val and initialize it to 1234
    int val = 1234;

    // Print the initial value of val outside the parallel region
    printf("Initial value of val (outside parallel region): %d\n", val);

    // Enter an OpenMP parallel region with the firstprivate clause applied to val
    #pragma omp parallel firstprivate(val)
    {
        // Get the thread number
        int thread_id = omp_get_thread_num();

        // Print the current value of val (private copy for each thread)
        printf("Thread %d: Current value of val = %d\n", thread_id, val);

        // Increment val by 1 (this only affects the private copy for this thread)
        val++;

        // Print the updated value of val (private copy after increment)
        printf("Thread %d: Updated value of val = %d\n", thread_id, val);
    }

    // Print the final value of val outside the parallel region
    printf("Final value of val (outside parallel region): %d\n", val);

    return 0;
}
```

```
Initial value of val (outside parallel region): 1234
Thread 2: Current value of val = 1234
Thread 2: Updated value of val = 1235
Thread 0: Current value of val = 1234
Thread 0: Updated value of val = 1235
Thread 1: Current value of val = 1234
Thread 1: Updated value of val = 1235
Thread 3: Current value of val = 1234
Thread 3: Updated value of val = 1235
Final value of val (outside parallel region): 1234
```

Experiment No 6

Experiment Title: OpenMP Program 4

Problem Statement:

Write a C program utilizing OpenMP directives to demonstrate the behavior of the private clause.

Steps to follow :
Open text editor.
write the below program in it.
Save the file with .c extentation.
Compile and execuate with given commands.

Source Code and Output /Screenshots:

```c
#include <stdio.h>
#include <omp.h>

int main() {
    // Initialize OpenMP with 4 threads
    omp_set_num_threads(4);

    // Declare an integer variable val and initialize it to 1234
    int val = 1234;

    // Print the initial value of val outside the parallel region
    printf("Initial value of val (outside parallel region): %d\n", val);

    // Enter an OpenMP parallel region with the private clause applied to val
    #pragma omp parallel private(val)
    {
        // Get the thread number
        int thread_id = omp_get_thread_num();

        // Set the private copy of val in each thread (no initial value)
        val = thread_id * 10;  // Each thread sets its private val to something unique

        // Print the current value of val (private copy for each thread)
        printf("Thread %d: Current value of val = %d\n", thread_id, val);
    }

    // Print the final value of val outside the parallel region
    printf("Final value of val (outside parallel region): %d\n", val);

    return 0;
}
```

```
Initial value of val (outside parallel region): 1234
Thread 1: Current value of val = 10
Thread 3: Current value of val = 30
Thread 0: Current value of val = 0
Thread 2: Current value of val = 20
Final value of val (outside parallel region): 1234
```

Experiment No 7

Experiment Title: OpenMP Program 5

Problem Statement: Write a Parallel C program where the iterations of a loop should be scheduled statically across the team of threads. A thread should perform CHUNK iterations at a time before being scheduled for the next CHUNK of work.

Source Code and Output /Screenshots:

```c
#include <stdio.h>
#include <omp.h>

int main() {
    // Define the number of iterations in the loop
    int num_iterations = 100;

    // Define the chunk size
    int chunk = 10;

    // Initialize OpenMP with a specific number of threads (e.g., 4 threads)
    omp_set_num_threads(4);

    // Print a message indicating the start of parallel execution
    printf("Starting parallel loop with static scheduling and chunk size %d:\n", chunk);

    // Parallel for loop with static scheduling and chunk size
    #pragma omp parallel for schedule(static, chunk)
    for (int i = 0; i < num_iterations; i++) {
        // Get the thread ID
        int thread_id = omp_get_thread_num();

        // Print the iteration index and the corresponding thread ID
        printf("Thread %d: Iteration %d\n", thread_id, i);
    }

    // Print a message indicating the end of the parallel execution
    printf("Parallel loop execution completed.\n");

    return 0;
}
```

```
Starting parallel loop with static scheduling and chunk size 10:
Thread 3: Iteration 30
Thread 3: Iteration 31
Thread 3: Iteration 32
Thread 3: Iteration 33
Thread 3: Iteration 34
Thread 3: Iteration 35
Thread 3: Iteration 36
Thread 3: Iteration 37
Thread 3: Iteration 38
Thread 3: Iteration 39
Thread 3: Iteration 70
Thread 3: Iteration 71
Thread 3: Iteration 72
Thread 3: Iteration 73
Thread 3: Iteration 74
Thread 3: Iteration 75
Thread 3: Iteration 76
Thread 3: Iteration 77
Thread 3: Iteration 78
Thread 3: Iteration 79
Thread 1: Iteration 10
Thread 1: Iteration 11
Thread 1: Iteration 12
Thread 1: Iteration 13
Thread 1: Iteration 14
Thread 1: Iteration 15
Thread 1: Iteration 16
Thread 1: Iteration 17
Thread 1: Iteration 18
Thread 1: Iteration 19
Thread 1: Iteration 50
Thread 1: Iteration 51
Thread 1: Iteration 52
Thread 1: Iteration 53
Thread 1: Iteration 54
```

```
Thread 1: Iteration 55
Thread 1: Iteration 56
Thread 1: Iteration 57
Thread 1: Iteration 58
Thread 1: Iteration 59
Thread 1: Iteration 90
Thread 1: Iteration 91
Thread 1: Iteration 92
Thread 1: Iteration 93
Thread 1: Iteration 94
Thread 1: Iteration 95
Thread 1: Iteration 96
Thread 1: Iteration 97
Thread 1: Iteration 98
Thread 1: Iteration 99
Thread 2: Iteration 20
Thread 2: Iteration 21
Thread 2: Iteration 22
Thread 2: Iteration 23
Thread 2: Iteration 24
Thread 2: Iteration 25
Thread 2: Iteration 26
Thread 2: Iteration 27
Thread 2: Iteration 28
Thread 2: Iteration 29
Thread 2: Iteration 60
Thread 2: Iteration 61
Thread 2: Iteration 62
Thread 2: Iteration 63
Thread 2: Iteration 64
Thread 2: Iteration 65
Thread 2: Iteration 66
Thread 2: Iteration 67
Thread 2: Iteration 68
Thread 2: Iteration 69
Thread 0: Iteration 0
```

```
Thread 0: Iteration 1
Thread 0: Iteration 2
Thread 0: Iteration 3
Thread 0: Iteration 4
Thread 0: Iteration 5
Thread 0: Iteration 6
Thread 0: Iteration 7
Thread 0: Iteration 8
Thread 0: Iteration 9
Thread 0: Iteration 40
Thread 0: Iteration 41
Thread 0: Iteration 42
Thread 0: Iteration 43
Thread 0: Iteration 44
Thread 0: Iteration 45
Thread 0: Iteration 46
Thread 0: Iteration 47
Thread 0: Iteration 48
Thread 0: Iteration 49
Thread 0: Iteration 80
Thread 0: Iteration 81
Thread 0: Iteration 82
Thread 0: Iteration 83
Thread 0: Iteration 84
Thread 0: Iteration 85
Thread 0: Iteration 86
Thread 0: Iteration 87
Thread 0: Iteration 88
Thread 0: Iteration 89
Parallel loop execution completed.
```

Experiment No 8

Experiment Title: OpenMP Program 6

Problem Statement:

Write a Parallel C program which should print the series of 2 and 4. Make sure both should be executed by different threads.

Source Code and Output /Screenshots:

```c
#include <stdio.h>
#include <omp.h>

int main() {
    // Set the number of threads to 2 (one for each series)
    omp_set_num_threads(2);

    // Parallel region where each thread will execute different tasks
    #pragma omp parallel
    {
        int thread_id = omp_get_thread_num();

        // If thread_id is 0, print the series of 2
        if (thread_id == 0) {
            for (int i = 1; i <= 5; i++) { // Print 5 terms of the series 2, 2, 2, ...
                printf("Thread %d: %d\n", thread_id, 2);
            }
        }
        // If thread_id is 1, print the series of 4
        else if (thread_id == 1) {
            for (int i = 1; i <= 5; i++) { // Print 5 terms of the series 4, 4, 4, ...
                printf("Thread %d: %d\n", thread_id, 4);
            }
        }
    }

    return 0;
}
```

```
Thread 1: 4
Thread 1: 4
Thread 1: 4
Thread 1: 4
Thread 1: 4
Thread 0: 2
Thread 0: 2
Thread 0: 2
Thread 0: 2
Thread 0: 2
```

Experiment No 9

Experiment Title: MPI Program 1

Problem Statement: Write MPI Program to print "Hello World".
MPI program to send and receive Hello World messages from all other processes to a Root process and print the received messages.

Source Code and Output /Screenshots:

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

int main(int argc, char** argv) {
    int rank, size;
    char message[20];
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    sprintf(message, "Hello from process %d", rank);

    if (rank == 0) {
        printf("Root process %d: Receiving messages:\n", rank);
        for (int i = 1; i < size; i++) {
            MPI_Recv(message, 20, MPI_CHAR, i, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            printf("Received message: %s\n", message);
        }
        printf("Received message: Hello from process 0\n");
    }
    else {
        MPI_Send(message, 20, MPI_CHAR, 0, 0, MPI_COMM_WORLD);
    }
    MPI_Finalize();

    return 0;
}
```

```
Root process 0: Receiving messages:
Received message: Hello from process 1
Received message: Hello from process 2
Received message: Hello from process 3
Received message: Hello from process 0
```

Experiment No 10

Experiment Title: MPI Program 2

Problem Statement: MPI program to send two numbers (array elements) per process to a Root process and print the received messages.

Source Code and Output /Screenshots:

```c
#include <mpi.h>
#include <stdio.h>
#define NUM_ELEMENTS 2

int main(int argc, char** argv) {
    int rank, size;
    int data[NUM_ELEMENTS];
    MPI_Init(&argc, &argv);

    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    data[0] = rank * 10 + 1;
    data[1] = rank * 10 + 2;

    if (rank == 0) {
        printf("Root process %d: Receiving messages:\n", rank);
        for (int i = 1; i < size; i++) {
            MPI_Recv(data, NUM_ELEMENTS, MPI_INT, i, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            printf("Received from process %d: %d, %d\n", i, data[0], data[1]);
        }
        printf("Received from process %d: %d, %d\n", rank, data[0], data[1]);
    }
    else {
        MPI_Send(data, NUM_ELEMENTS, MPI_INT, 0, 0, MPI_COMM_WORLD);
    }

    MPI_Finalize();

    return 0;
```

```
Root process 0: Receiving messages:
Received from process 1: 11, 12
Received from process 2: 21, 22
Received from process 3: 31, 32
Received from process 0: 1, 2
```

Experiment No 11

Experiment Title: MPI Program 3

Problem Statement: MPI program to find sum of first N integers using any given number of processes. Example, N=10,000 and no. of processes can be 4 or 8 or 12 etc.

Source Code and Output /Screenshots:

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

int main(int argc, char** argv) {
    int rank, size;
    long long N = 10000;
    long long local_sum = 0, global_sum = 0;
    long long start, end;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    long long range_per_process = N / size;
    start = rank * range_per_process + 1;
    end = (rank + 1) * range_per_process;
    if (rank == size - 1) {
        end = N;
    }
    for (long long i = start; i <= end; i++) {
        local_sum += i;
    }
    MPI_Reduce(&local_sum, &global_sum, 1, MPI_LONG_LONG, MPI_SUM, 0, MPI_COMM_WORLD);

    if (rank == 0) {
        printf("The sum of the first %lld integers is: %lld\n", N, global_sum);
    }
    MPI_Finalize();

    return 0;
}
```

```
The sum of the first 10000 integers is: 50005000
```

Experiment No 12

Experiment Title: MPI Program 4

Problem Statement: MPI program to find sum of n integers in which processors are arranged in ring topology using MPI point-to-point blocking communication library calls.

Source Code and Output /Screenshots:

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

int main(int argc, char** argv) {
    int rank, size;
    long long N = 10000;
    long long local_sum = 0, total_sum = 0;
    long long start, end;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &siz
    long long range_per_process = N / size;
    start = rank * range_per_process + 1;
    end = (rank + 1) * range_per_process;
    if (rank == size - 1) {
        end = N;
    }
    for (long long i = start; i <= end; i++) {
        local_sum += i;
    }
    int prev = (rank - 1 + size) % size;
    int next = (rank + 1) % size;
    if (rank == 0) {
        MPI_Send(&local_sum, 1, MPI_LONG_LONG, next, 0, MPI_COMM_WORLD);
        MPI_Recv(&total_sum, 1, MPI_LONG_LONG, prev, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        total_sum += local_sum;
        printf("The total sum of the first %lld integers is: %lld\n", N, total_sum);
    }
    else
        MPI_Send(&local_sum, 1, MPI_LONG_LONG, next, 0, MPI_COMM_WORLD);
        MPI_Recv(&total_sum, 1, MPI_LONG_LONG, prev, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        total_sum += local_sum;
        if (rank != size - 1) {
            MPI_Send(&total_sum, 1, MPI_LONG_LONG, next, 0, MPI_COMM_WORLD);
        }
    }
    MPI_Finalize();

    return 0;
```

```
The total sum of the first 10000 integers is: 50005000
```

Experiment No 13

Experiment Title: CUDA Program 1

Problem Statement: Write a CUDA program to perform two matrix additions.

Source Code and Output /Screenshots:

```
#include <stdio.h>
#include <cuda_runtime.h>
#define N 1024
__global__ void matrixAdd(float *A, float *B, float *C, int width) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    int idy = blockIdx.y * blockDim.y + threadIdx.y;

    if (idx < width && idy < width) {
        int index = idy * width + idx;
        C[index] = A[index] + B[index];
    }
}

int main() {
    int size = N * N * sizeof(float);
    float *h_A, *h_B, *h_C, *h_D, *h_E;
    h_A = (float*)malloc(size);
    h_B = (float*)malloc(size);
    h_C = (float*)malloc(size);
    h_D = (float*)malloc(size);
    h_E = (float*)malloc(size);
    for (int i = 0; i < N * N; i++) {
        h_A[i] = rand() % 10;
        h_B[i] = rand() % 10;
        h_D[i] = rand() % 10;
    }
    float *d_A, *d_B, *d_C, *d_D, *d_E;
    cudaMalloc(&d_A, size);
    cudaMalloc(&d_B, size);
    cudaMalloc(&d_C, size);
    cudaMalloc(&d_D, size);
    cudaMalloc(&d_E, size);
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
```

```
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_D, h_D, size, cudaMemcpyHostToDevice);
    dim3 threadsPerBlock(16, 16);
    dim3 numBlocks((N + 15) / 16, (N + 15) / 16);
    matrixAdd<<<numBlocks, threadsPerBlock>>>(d_A, d_B, d_C, N);

    cudaDeviceSynchronize();
    matrixAdd<<<numBlocks, threadsPerBlock>>>(d_C, d_D, d_E, N);
    cudaDeviceSynchronize();
    cudaMemcpy(h_E, d_E, size, cudaMemcpyDeviceToHost);
    printf("Result matrix E (a few elements):\n");
    for (int i = 0; i < 5; i++) {
        printf("%f ", h_E[i]);
    }
    printf("\n");
    free(h_A);
    free(h_B);
    free(h_C);
    free(h_D);
    free(h_E);
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);
    cudaFree(d_D);
    cudaFree(d_E);

    return 0;
}
```

```
Result matrix E (a few elements):
42.000000 32.000000 24.000000 55.000000 19.000000
```

Experiment No 14

Experiment Title: CUDA Program 2

Problem Statement: Write a CUDA program to perform two matrix multiplications.

Source Code and Output /Screenshots:

```
#include <stdio.h>
#include <cuda_runtime.h>

#define N 1024
__global__ void matrixMultiply(float *A, float *B, float *C, int width) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    if (row < width && col < width) {
        float value = 0;
        for (int k = 0; k < width; k++) {
            value += A[row * width + k] * B[k * width + col];
        }
        C[row * width + col] = value;
    }
}

int main() {
    int size = N * N * sizeof(float);
    float *h_A, *h_B, *h_C, *h_D, *h_E;
    h_A = (float*)malloc(size);
    h_B = (float*)malloc(size);
    h_C = (float*)malloc(size);
    h_D = (float*)malloc(size);
    h_E = (float*)malloc(size);
    for (int i = 0; i < N * N; i++) {
        h_A[i] = rand() % 10;
        h_B[i] = rand() % 10;
        h_D[i] = rand() % 10;
    }
    float *d_A, *d_B, *d_C, *d_D, *d_E;
    cudaMalloc(&d_A, size);
    cudaMalloc(&d_B, size);
    cudaMalloc(&d_C, size);
```

```
    cudaMalloc(&d_A, size);
    cudaMalloc(&d_B, size);
    cudaMalloc(&d_C, size);
    cudaMalloc(&d_D, size);
    cudaMalloc(&d_E, size);
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_D, h_D, size, cudaMemcpyHostToDevice);
    dim3 threadsPerBlock(16, 16);  // 16x16 bloc(N + 15) / 16);
    matrixMultiply<<<numBlocks, threadsPerBlock>>>(d_A, d_B, d_C, N);
    cudaDeviceSynchronize();
    matrixMultiply<<<numBlocks, threadsPerBlock>>>(d_C, d_D, d_E, N);
    cudaDeviceSynchronize();
    cudaMemcpy(h_E, d_E, size, cudaMemcpyDeviceToHost);

    printf("Result matrix E (a few elements):\n");
    for (int i = 0; i < 5; i++) {
        printf("%f ", h_E[i]);
    }
    printf("\n");
    free(h_A);
    free(h_B);
    free(h_C);
    free(h_D);
    free(h_E);
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);
    cudaFree(d_D);
    cudaFree(d_E);

    return 0;
```

```
Result matrix E (a few elements):
256.000000 320.000000 128.000000 512.000000 384.000000
```