Experiment No 3

Experiment Title: OpenMP program 1

Source Code and Output /Screenshots:

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

int main() {
    // Initialize the OpenMP parallel region
    #pragma omp parallel
    {
        int thread_id = omp_get_thread_num(); // Get the thread ID
        int num_threads = omp_get_num_threads(); // Get the total number of threads

        // Print a message with thread ID
        printf("Hello, World from thread %d out of %d threads!\n", thread_id, num_threads);
    }

    return 0;
}
```

```
Hello, World from thread 0 out of 4 threads!
Hello, World from thread 1 out of 4 threads!
Hello, World from thread 2 out of 4 threads!
Hello, World from thread 3 out of 4 threads!
```

Experiment No 4

Experiment Title: OpenMP Program 2

Source Code and Output /Screenshots:

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

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

    // Parallel region
    #pragma omp parallel
    {
        int thread_id = omp_get_thread_num(); // Get the thread ID
        printf("Hello from thread %d: My name is [Your Name]\n", thread_id);
    }

    return 0;
}
```

```
Hello from thread 0: My name is [Your Name]
Hello from thread 1: My name is [Your Name]
Hello from thread 2: My name is [Your Name]
Hello from thread 3: My name is [Your Name]
```

Experiment No 5

Experiment Title: OpenMP Program 3

Source Code and Output /Screenshots:

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

int main() {
    int i, private_var = 0; // Declare and initialize a variable

    printf("Initial value of private_var: %d\n", private_var);

    // Parallel region with private clause
    #pragma omp parallel private(private_var)
    {
        int thread_id = omp_get_thread_num(); // Get thread ID
        private_var = thread_id + 10;          // Assign a value based on thread ID
        printf("Thread %d: private_var = %d\n", thread_id, private_var);
    }

    // Outside the parallel region
    printf("Value of private_var after parallel region: %d\n", private_var);

    return 0;
}
```

```
Initial value of private_var: 0
Thread 0: private_var = 10
Thread 1: private_var = 11
Thread 2: private_var = 12
Thread 3: private_var = 13
Value of private_var after parallel region: 0
```

Experiment No 6

Experiment Title: OpenMP Program 4

Source Code and Output /Screenshots:

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

int main() {
    int i, private_var = 0; // Declare and initialize a variable

    printf("Initial value of private_var: %d\n", private_var);

    // Parallel region with private clause
    #pragma omp parallel private(private_var)
    {
        int thread_id = omp_get_thread_num(); // Get thread ID
        private_var = thread_id + 10;         // Assign a value based on thread ID
        printf("Thread %d: private_var = %d\n", thread_id, private_var);
    }

    // Outside the parallel region
    printf("Value of private_var after parallel region: %d\n", private_var);

    return 0;
}
```

```
Initial value of private_var: 0
Thread 0: private_var = 10
Thread 1: private_var = 11
Thread 2: private_var = 12
Thread 3: private_var = 13
Value of private_var after parallel region: 0
```

Experiment No 7

Experiment Title: OpenMP Program 5

Source Code and Output /Screenshots:

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

#define N 20 // Total number of iterations
#define CHUNK 4 // Number of iterations per chunk

int main() {
    int i;

    // Parallel region with static scheduling and chunk size
    #pragma omp parallel for schedule(static, CHUNK)
    for (i = 0; i < N; i++) {
        int thread_id = omp_get_thread_num(); // Get the thread ID
        printf("Thread %d is processing iteration %d\n", thread_id, i);
    }

    return 0;
}
```

```
Thread 0 is processing iteration 0
Thread 1 is processing iteration 4
Thread 2 is processing iteration 8
Thread 3 is processing iteration 12
Thread 0 is processing iteration 1
Thread 1 is processing iteration 5
Thread 2 is processing iteration 9
Thread 3 is processing iteration 13
Thread 0 is processing iteration 2
Thread 1 is processing iteration 6
Thread 2 is processing iteration 10
Thread 3 is processing iteration 14
Thread 0 is processing iteration 3
Thread 1 is processing iteration 7
Thread 2 is processing iteration 11
Thread 3 is processing iteration 15
```

Experiment No 8

Experiment Title: OpenMP Program 6

Source Code and Output /Screenshots:

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

int main() {
    #pragma omp parallel
    {
        int thread_id = omp_get_thread_num();

        if (thread_id == 0) {
            printf("Thread %d: Series of 2: ", thread_id);
            for (int i = 1; i <= 5; i++) {
                printf("%d ", 2 * i);
            }
            printf("\n");
        }

        if (thread_id == 1) {
            printf("Thread %d: Series of 4: ", thread_id);
            for (int i = 1; i <= 5; i++) {
                printf("%d ", 4 * i);
            }
            printf("\n");
        }
    }

    return 0;
}
```

```
Thread 0: Series of 2: 2 4 6 8 10
Thread 1: Series of 4: 4 8 12 16 20
```

Experiment No 9

Experiment Title: MPI Program 1

Source Code and Output /Screenshots:

```c
#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

Source Code and Output /Screenshots:

```
#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

Source Code and Output /Screenshots:

```c
#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

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

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

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
```