HIGH PERFORMANCE COMPUTING

SEMESTER 6, 2023

Name - Akhil Dubey

Roll - 2020UCO1673

Branch - COE 3

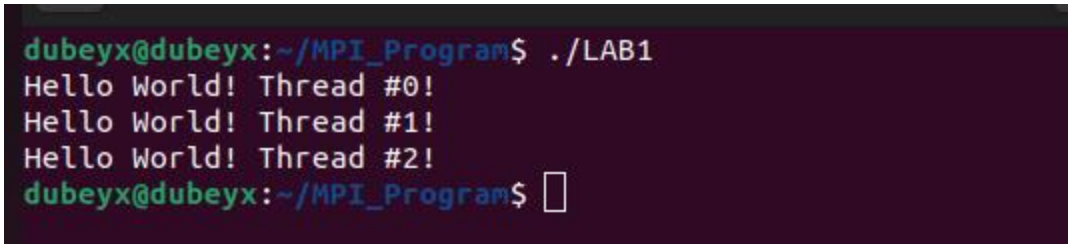
Submitted To:

Experiments

Experiment 1

Run a basic hello World program using pthreads

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

#define NUM_THREADS 3

void *PrintHello(void *threadid)
{
    long tid;
    tid = (long)threadid;
    printf("Hello World! Thread #%ld!\n", tid);
    pthread_exit(NULL);
}

int main(int argc, char *argv[])
{
    pthread_t threads[NUM_THREADS];
    int rc;
    long t;

    for (t = 0; t < NUM_THREADS; t++)
    {
        rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
        if (rc)
        {
            printf("ERROR; return code from pthread_create() is %d\n", rc);
            exit(-1);
        }
    }

    /* Last thing that main() should do */
    pthread_exit(NULL);
}
```

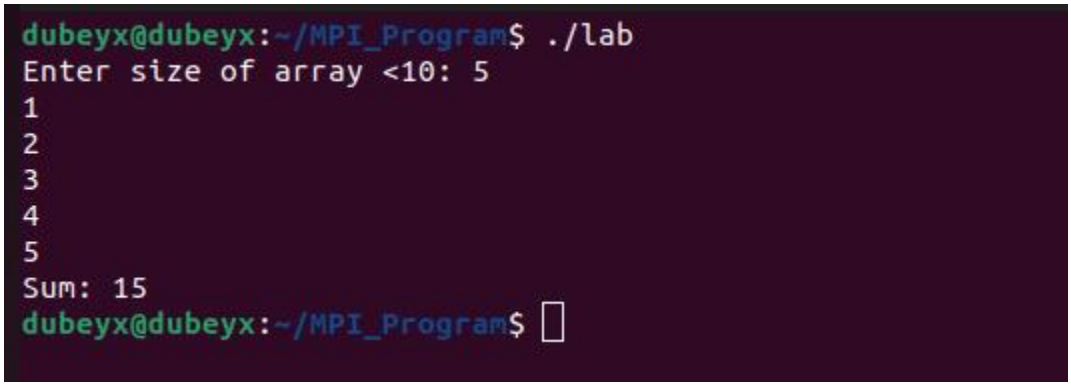

dubeyx@dubeyx:~/MPI_Program$ ./LAB1
Hello World! Thread #0!
Hello World! Thread #1!
Hello World! Thread #2!
dubeyx@dubeyx:~/MPI_Program$

Experiment 2

Find the sum of all elements of an array using two processors using pThreads

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

#define ARRAY_SIZE 10 // size of the array
#define NUM_THREADS 2 // number of threads

int array[ARRAY_SIZE];              // the array to sum
int partial_sums[NUM_THREADS];      // partial sums for each thread
pthread_mutex_t lock;              // mutex for accessing partial_sums

// thread function to compute partial sum
void *partial_sum(void *thread_id)
{
    int id = *(int *)thread_id;
    int start = id * (ARRAY_SIZE / NUM_THREADS);
    int end = (id + 1) * (ARRAY_SIZE / NUM_THREADS);
    int sum = 0;
    for (int i = start; i < end; i++)
    {
        sum += array[i];
    }
    pthread_mutex_lock(&lock);
    partial_sums[id] = sum;
    pthread_mutex_unlock(&lock);
    pthread_exit(NULL);
}

int main()
{
    int n;
    printf("Enter size of array <10: ");
    scanf("%d", &n);
```

```
    for (int i = 0; i < n; i++)
    {
        scanf("%d", &array[i]);
    }

    // initialize the mutex
    pthread_mutex_init(&lock, NULL);

    pthread_t threads[NUM_THREADS];
    int thread_ids[NUM_THREADS];

    // create the threads
    for (int i = 0; i < NUM_THREADS; i++)
    {
        thread_ids[i] = i;
        pthread_create(&threads[i], NULL, partial_sum, (void *)&thread_ids[i]);
    }

    // join the threads
    for (int i = 0; i < NUM_THREADS; i++)
    {
        pthread_join(threads[i], NULL);
    }

    // compute the final sum
    int sum = 0;
    for (int i = 0; i < NUM_THREADS; i++)
    {
        sum += partial_sums[i];
    }

    // print the final sum
    printf("Sum: %d\n", sum);

    // destroy the mutex
    pthread_mutex_destroy(&lock);

    return 0;
}
```


dubeyx@dubeyx:~/MPI_Program$ ./lab
Enter size of array <10: 5
1
2
3
4
5
Sum: 15
dubeyx@dubeyx:~/MPI_Program$

Experiment 3

Find sum of all elements using p processors by using pthreads

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>

#define ARRAY_SIZE 10 // size of the array
int array[ARRAY_SIZE]; // the array to sum
int sum_global = 0; // global variable to store the sum
pthread_mutex_t lock; // mutex for accessing sum
int pthread_num; // global variable to store number of pthreads
pthread_once_t once_control = PTHREAD_ONCE_INIT; // control variable for
pthread_once

// function to initialize pthread_num using sysconf
void initialize_pthread_num() {
    pthread_num = sysconf(_SC_NPROCESSORS_ONLN);
}

// thread function to compute partial sum
void *partial_sum(void *thread_id) {
    int id = *(int *)thread_id;
    int start = id * (ARRAY_SIZE / pthread_num);
    int end = (id + 1) * (ARRAY_SIZE / pthread_num);
    int sum = 0;
    for (int i = start; i < end; i++) {
        sum += array[i];
    }
    pthread_mutex_lock(&lock);
    sum_global += sum;
    pthread_mutex_unlock(&lock);
    pthread_exit(NULL);
}

int main() {
    // initialize the array with random values
    int n;
    printf("Enter size of array <10: ");
    scanf("%d", &n);
```

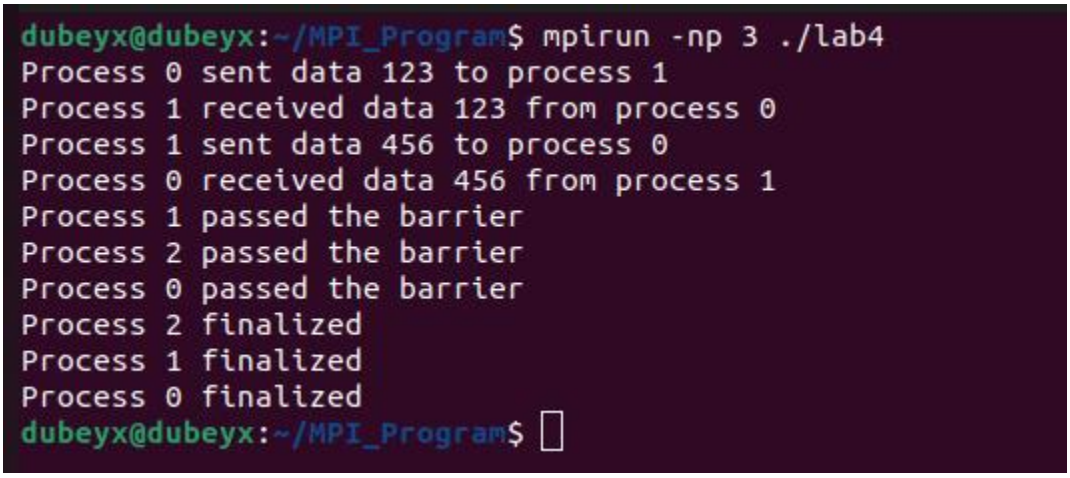
```
    for (int i = 0; i < n; i++) {
        scanf("%d", &array[i]);
    }

    // initialize the mutex
    pthread_mutex_init(&lock, NULL);

    // initialize pthread_num using pthread_once
    pthread_once(&once_control, initialize_pthread_num);

    // prompt user for number of pthreads to use
    printf("Enter number of pthreads to use (1-%d): ", pthread_num);
    scanf("%d", &pthread_num);
    if (pthread_num < 1 || pthread_num > pthread_num) {
        printf("Invalid number of pthreads\n");
        return 1;
    }

    pthread_t threads[pthread_num];
    int thread_ids[pthread_num];

    // create the threads
    for (int i = 0; i < pthread_num; i++) {
        thread_ids[i] = i;
        pthread_create(&threads[i], NULL, partial_sum, (void *)&thread_ids[i]);
    }

    // join the threads
    for (int i = 0; i < pthread_num; i++) {
        pthread_join(threads[i], NULL);
    }

    // print the final sum
    printf("Sum: %d\n", sum_global);

    // destroy the mutex
    pthread_mutex_destroy(&lock);

    return 0;
}
```

```
dubeyx@dubeyx:~/MPI_Program$ ./lab3
Enter size of array <10: 5
1
2
3
4
5
Enter number of pthreads to use (1-4): 4
Sum: 15
dubeyx@dubeyx:~/MPI_Program$
```

Experiment 4

Illustrate basic mpi communication routines

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

int main(int argc, char *argv[]) {
    int rank, size, data;
    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // Send and receive data
    if (rank == 0) {
        data = 123;
        MPI_Send(&data, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
        printf("Process %d sent data %d to process 1\n", rank, data);
        MPI_Recv(&data, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, &status);
        printf("Process %d received data %d from process 1\n", rank, data);
    } else if (rank == 1) {
        MPI_Recv(&data, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
        printf("Process %d received data %d from process 0\n", rank, data);
        data = 456;
        MPI_Send(&data, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
        printf("Process %d sent data %d to process 0\n", rank, data);
    }

    // Barrier
    MPI_Barrier(MPI_COMM_WORLD);
    printf("Process %d passed the barrier\n", rank);

    // Finalize
    MPI_Finalize();
    printf("Process %d finalized\n", rank);

    return 0;
}
```


dubeyx@dubeyx:~/MPI_Program$ mpirun -np 3 ./lab4
Process 0 sent data 123 to process 1
Process 1 received data 123 from process 0
Process 1 sent data 456 to process 0
Process 0 received data 456 from process 1
Process 1 passed the barrier
Process 2 passed the barrier
Process 0 passed the barrier
Process 2 finalized
Process 1 finalized
Process 0 finalized
dubeyx@dubeyx:~/MPI_Program$

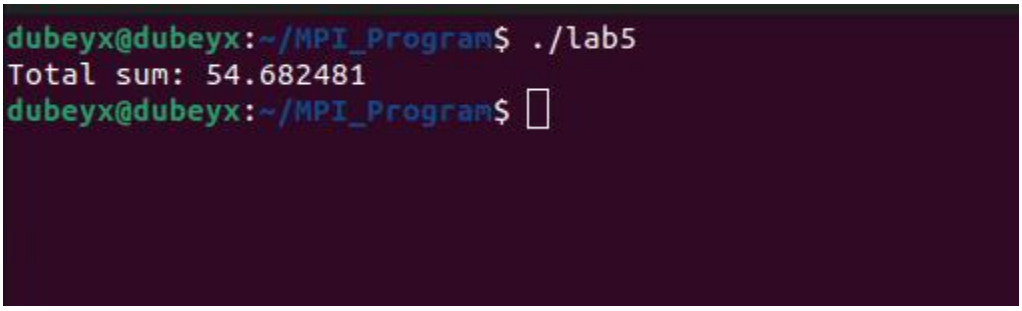
Experiment 5

Design a parallel program for summing up an array, matrix multiplication and show logging and tracing mpi activity

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

#define ARRAY_SIZE 100000
#define MATRIX_SIZE 1000

int main(int argc, char **argv) {
    int rank, size;
    int array[ARRAY_SIZE], sum = 0;
    int matrix[MATRIX_SIZE][MATRIX_SIZE],
result[MATRIX_SIZE][MATRIX_SIZE];
    int i, j, k;

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

    // Initialize array
    for (i = 0; i < ARRAY_SIZE; i++) {
        array[i] = i;
    }

    // Sum up array
    int local_sum = 0;
```

```
    int chunk_size = ARRAY_SIZE / size;
    int start = rank * chunk_size;
    int end = start + chunk_size;

    for (i = start; i < end; i++) {
        local_sum += array[i];
    }

    MPI_Reduce(&local_sum, &sum, 1, MPI_INT, MPI_SUM, 0,
MPI_COMM_WORLD);

    // Initialize matrix
    for (i = 0; i < MATRIX_SIZE; i++) {
        for (j = 0; j < MATRIX_SIZE; j++) {
            matrix[i][j] = i + j;
            result[i][j] = 0;
        }
    }

    // Multiply matrix
    int chunk_rows = MATRIX_SIZE / size;
    int start_row = rank * chunk_rows;
    int end_row = start_row + chunk_rows;

    for (i = start_row; i < end_row; i++) {
        for (j = 0; j < MATRIX_SIZE; j++) {
            for (k = 0; k < MATRIX_SIZE; k++) {
                result[i][j] += matrix[i][k] * matrix[k][j];
            }
        }
    }

    MPI_Barrier(MPI_COMM_WORLD);
```

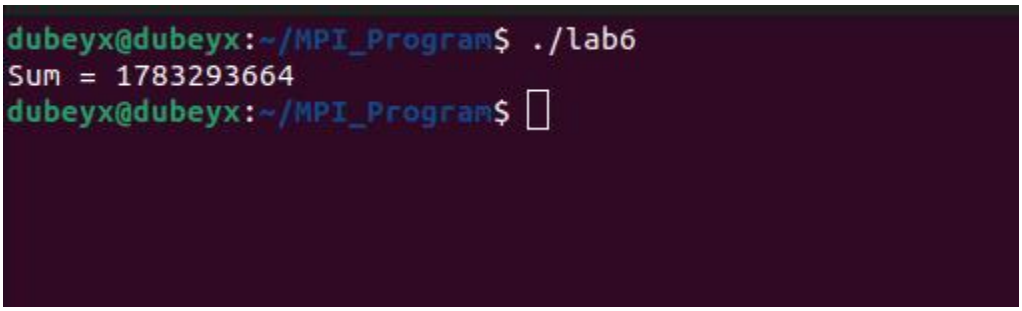
```
    // Logging/tracing MPI activity
    char log_filename[20];
    sprintf(log_filename, "log_%d.txt", rank);
    FILE *log_file = fopen(log_filename, "w");

    fprintf(log_file, "Rank %d: sum = %d\n", rank, sum);

    for (i = 0; i < MATRIX_SIZE; i++) {
        for (j = 0; j < MATRIX_SIZE; j++) {
            fprintf(log_file, "%d ", result[i][j]);
        }
        fprintf(log_file, "\n");
    }

    fclose(log_file);

    MPI_Finalize();

    return 0;
}
```



Experiment 6

Write a c program with openmp to implement loop work sharing

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

#define N 1000000

int main() {
    int i, sum = 0;
    int a[N];

    // Initialize array
    for (i = 0; i < N; i++) {
        a[i] = i;
    }

    // Use OpenMP to parallelize loop
    #pragma omp parallel for reduction(+:sum)
    for (i = 0; i < N; i++) {
        sum += a[i];
    }

    printf("Sum = %d\n", sum);

    return 0;
}
```

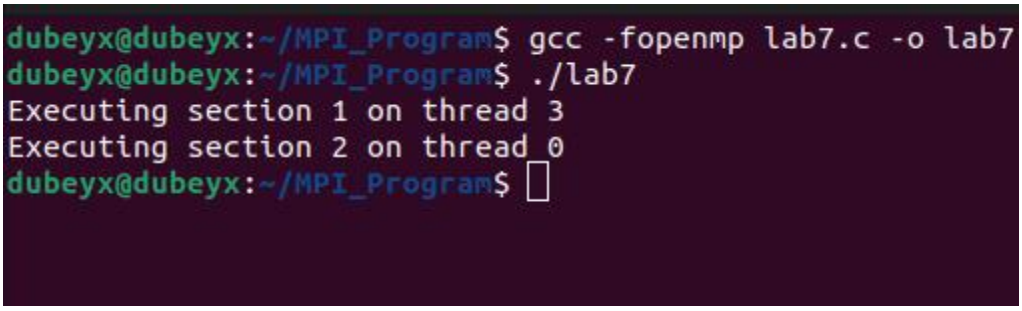

dubeyx@dubeyx:~/MPI_Program$ ./lab6
Sum = 1783293664
dubeyx@dubeyx:~/MPI_Program$

Experiment 7

Write a c program with openmp to implement section work sharing

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

void section1() {
    printf("Executing section 1 on thread %d\n", omp_get_thread_num());
}

void section2() {
    printf("Executing section 2 on thread %d\n", omp_get_thread_num());
}

int main() {
    // Use OpenMP to parallelize sections
    #pragma omp parallel sections
    {
        #pragma omp section
        {
            section1();
        }

        #pragma omp section
        {
            section2();
        }
    }
```

```
    return 0;
}
```


dubeyx@dubeyx:~/MPI_Program$ gcc -fopenmp lab7.c -o lab7
dubeyx@dubeyx:~/MPI_Program$ ./lab7
Executing section 1 on thread 3
Executing section 2 on thread 0
dubeyx@dubeyx:~/MPI_Program$

Experiment 8

Write a c program to illlustrate process synchronization and collective data movement

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

#define ARRAY_SIZE 10

int main(int argc, char** argv) {
    int rank, size;
    int array[ARRAY_SIZE];
    int sum = 0;

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

    if (rank == 0) {
        // initialize array
        for (int i = 0; i < ARRAY_SIZE; i++) {
            array[i] = i + 1;
        }
    }

    // Synchronize all processes
    MPI_Barrier(MPI_COMM_WORLD);
```

```
    // Scatter the array across all processes
    MPI_Scatter(array, ARRAY_SIZE/size, MPI_INT, array, ARRAY_SIZE/size,
MPI_INT, 0, MPI_COMM_WORLD);

    // Compute the sum of the local portion of the array
    for (int i = 0; i < ARRAY_SIZE/size; i++) {
        sum += array[i];
    }

    // Compute the global sum of the array
    int global_sum;
    MPI_Reduce(&sum, &global_sum, 1, MPI_INT, MPI_SUM, 0,
MPI_COMM_WORLD);

    if (rank == 0) {
        printf("Global sum: %d\n", global_sum);
    }

    MPI_Finalize();

    return 0;
}
```