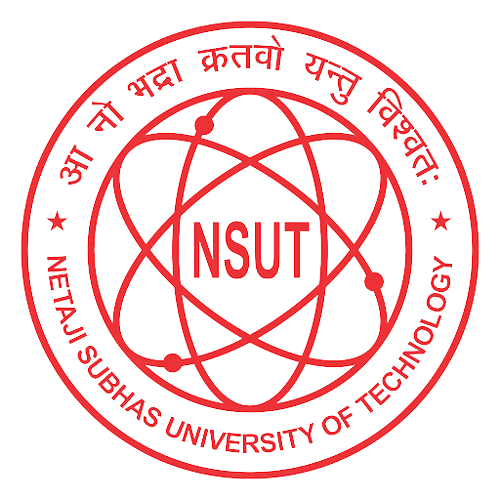
**HIGH PERFORMANCE**

**COMPUTING**

**SEMESTER 6, 2023**



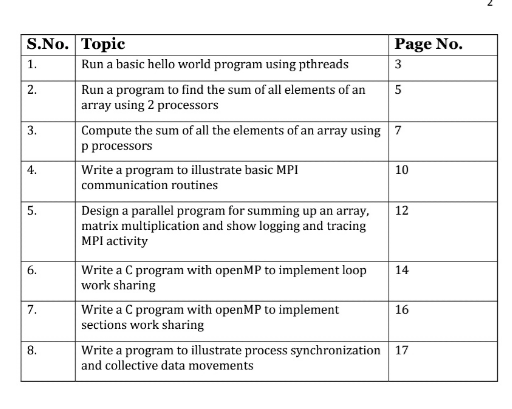
**Name - Akhil Dubey**

**Roll - 2020UCO1673**

**Branch - COE 3**

**Submitted To:**

**Experiments**



**Experiment 1**

**Run a basic hello World program using pthreads**

#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);

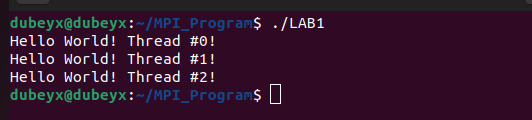
        }

    }

    /\* Last thing that main() should do \*/

    pthread\_exit(NULL);

}



**Experiment 2**

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

#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

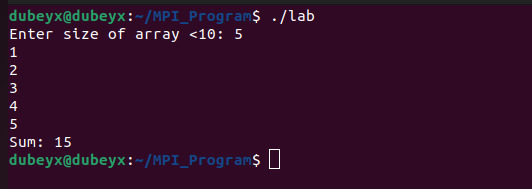
    printf("Sum: %d\n", sum);

    // destroy the mutex

    pthread\_mutex\_destroy(&lock);

    return 0;

}



**Experiment 3**

**Find sum of all elements using p processors by using pthreads**

#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

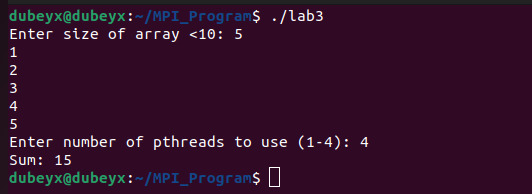
    printf("Sum: %d\n", sum\_global);

    // destroy the mutex

    pthread\_mutex\_destroy(&lock);

    return 0;

}



**Experiment 4**

**Illustrate basic mpi communication routines**

#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);

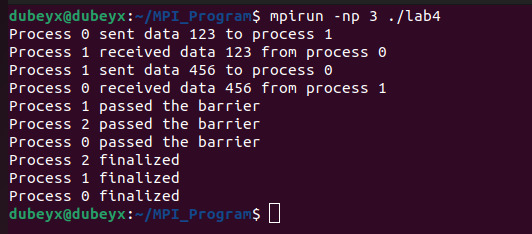
    // Finalize

    MPI\_Finalize();

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

    return 0;

}



**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");

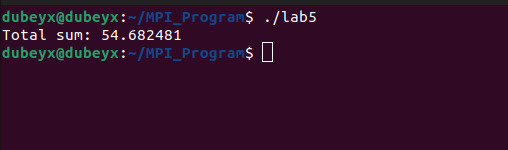
}

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++) {

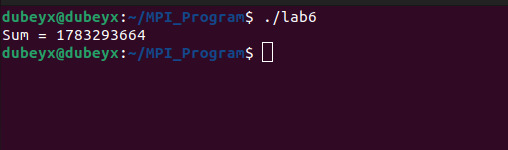
sum += a[i];

}

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

return 0;

}



**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

{

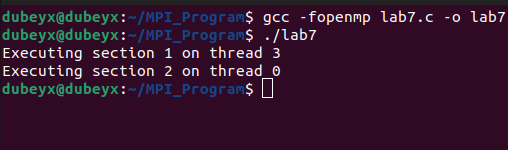
section2();

}

}

return 0;

}



**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;

}

