Practical File

High Performance Computing

(COCSC18)

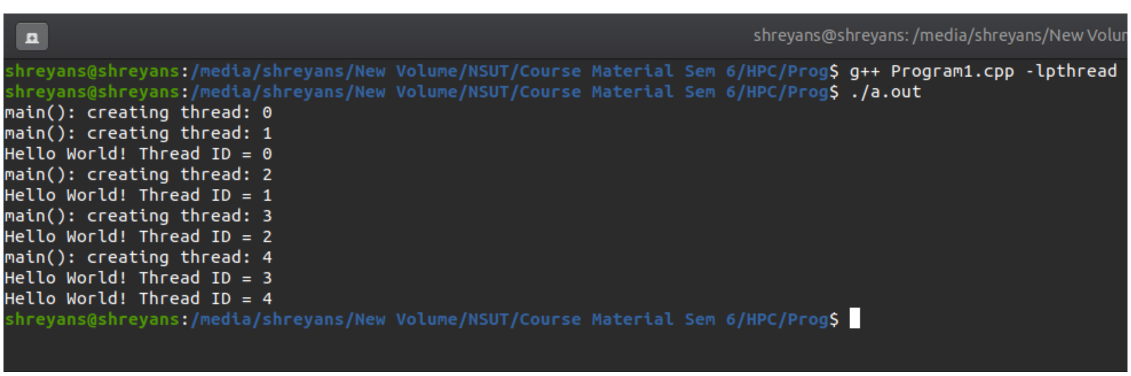
Submitted By:
Shreyans Jain
2019UCO1659
COE2

INDEX

Experiment 1:

Run a basic hello World Program using pthreads

```cpp
#include <iostream>
#include <cstdlib>
#include <pthread.h>
using namespace std;
#define NUM_THREAD 5
void *printHello(void *threadid);

int main()
{
    pthread_t threads[NUM_THREAD];

    int rc = 0;

    for (long i = 0; i < NUM_THREAD; i++)
    {
        cout << "main(): creating thread: " << i << endl;
        rc = pthread_create(&threads[i], NULL, printHello, (void *)i);

        if (rc)
        {
            cout << "Error: Unable to create thread" << rc << endl;
            exit(-1);
        }
    }

    pthread_exit(NULL);
}
void *printHello(void *threadid)
{
    long thread_id = (long)threadid;
    cout << "Hello World! Thread ID = " << thread_id << endl;
    pthread_exit(NULL);
}
```



Experiment 2: Run a program to find the sum of all elements of an array using 2 processors.

```
#include <iostream>
#include <pthread.h>
using namespace std;

#define MAX 16
#define MAX_THREAD 2
int part = 0;
int arr[MAX] = {1,3,5,7,9,2,4,6,8,10,11,13,15,12,14,18};
int sum[2] = {};

void* array_sum(void* array)
{
    int thread_part = part++;
    for (int i = thread_part*(MAX/MAX_THREAD); i < (thread_part+1)*(MAX/MAX_THREAD);
i++)
    {
        sum[thread_part] += arr[i];
    }

    pthread_exit(NULL);

}

int main()
{
    pthread_t threads[MAX_THREAD];

    // creating threads
    for (int i = 0; i < MAX_THREAD; i++)
    {
        pthread_create(&threads[i], NULL, array_sum, (void*)NULL);
    }

    // joining the threads after they have completed their actions
    for (int i = 0; i < MAX_THREAD; i++)
    {
        pthread_join(threads[i], NULL);
    }

    int sum_elements = 0;
    for (int i = 0; i < MAX_THREAD; i++)
    {
        sum_elements += sum[i];
```

```
    }

    int normal_sum = 0;
    for(int i = 0; i<MAX; i++)
    {
        normal_sum+= arr[i];
    }

    cout<<"Normal sum is "<< normal_sum<<endl;
    cout<<"The sum is "<<sum_elements<<endl;
    return 0;
}
```

```
shreyans@shreyans: /media/shreyans/New Volume/N
shreyans@shreyans:/media/shreyans/New Volume/NSUT/Course Material Sem 6/HPC/Prog$ g++ Program2.cpp -lpthread
shreyans@shreyans:/media/shreyans/New Volume/NSUT/Course Material Sem 6/HPC/Prog$ ./a.out
Normal sum is 138
The sum is 138
shreyans@shreyans:/media/shreyans/New Volume/NSUT/Course Material Sem 6/HPC/Prog$
```

Experiment 3: Run a program to find the sum of all elements of an array using p processors.

```
#include <iostream>
#include <pthread.h>
using namespace std;

#define MAX 16
#define MAX_THREAD 4
int part = 0;
int arr[MAX] = {1,3,5,7,9,2,4,6,8,10,11,13,15,12,14,18};
int sum[2] = {};

void* array_sum(void* array)
{
    int thread_part = part++;
    for (int i = thread_part*(MAX/MAX_THREAD); i < (thread_part+1)*(MAX/MAX_THREAD);
i++)
    {
        sum[thread_part] += arr[i];
    }

    pthread_exit(NULL);

}
```

```
int main()
{
    pthread_t threads[MAX_THREAD];

    // creating threads
    for (int i = 0; i < MAX_THREAD; i++)
    {
        pthread_create(&threads[i], NULL, array_sum, (void*)NULL);
    }

    // joining the threads after they have completed their actions
    for (int i = 0; i < MAX_THREAD; i++)
    {
        pthread_join(threads[i], NULL);
    }

    int sum_elements = 0;
    for (int i = 0; i < MAX_THREAD; i++)
    {
        sum_elements += sum[i];
    }

    int normal_sum = 0;
    for(int i = 0; i<MAX; i++)
    {
        normal_sum+= arr[i];
    }

    cout<<"Normal sum is "<< normal_sum<<endl;
    cout<<"The sum is "<<sum_elements<<endl;
    return 0;
}
```

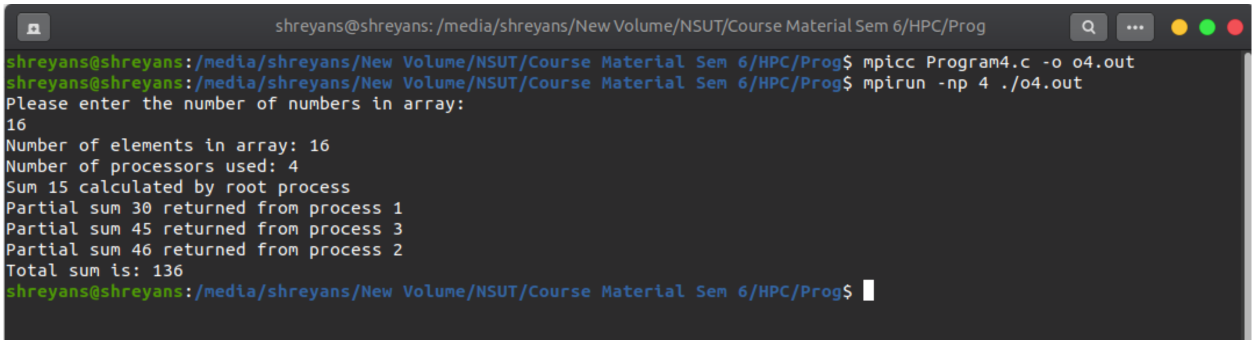
```
shreyans@shreyans: /media/shreyans/New Volume/N
shreyans@shreyans:/media/shreyans/New Volume/NSUT/Course Material Sem 6/HPC/Prog$ g++ Program3.cpp -lpthread
shreyans@shreyans:/media/shreyans/New Volume/NSUT/Course Material Sem 6/HPC/Prog$ ./a.out
Normal sum is 138
The sum is 138
shreyans@shreyans:/media/shreyans/New Volume/NSUT/Course Material Sem 6/HPC/Prog$
```

Experiment 4: Program to illustrate MPI Communication Routines

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

#define max_count 1000
#define send_data_tag 2001
#define return_data_tag 2002

int array[max_count];
int array2[max_count];

int main(int argc, char **argv)
{
    int sum, partial_sum;
    MPI_Status status;
    int my_id, root_process, ierr, i, num_rows, num_procs,
        an_id, num_rows_to_receive, avg_rows_per_process,
        sender, num_rows_received, start_row, end_row, num_rows_to_send;

    /* Now replicte this process to create parallel processes.
     * From this point on, every process executes a seperate copy
     * of this program */

    ierr = MPI_Init(&argc, &argv);

    root_process = 0;

    /* find out MY process ID, and how many processes were started. */

    ierr = MPI_Comm_rank(MPI_COMM_WORLD, &my_id);
    ierr = MPI_Comm_size(MPI_COMM_WORLD, &num_procs);

    if (my_id == root_process)
    {

        /* determine how many numbers to sum */

        printf("Please enter the number of numbers in array: \n");
        scanf("%i", &num_rows);
        printf("Number of elements in array: %d\n", num_rows);
        printf("Number of processors used: %d\n", num_procs);

        if (num_rows > max_count)
```

```
{
    printf("Too many numbers.\n");
    exit(1);
}

avg_rows_per_process = num_rows / num_procs;

/* initialize an array */

for (i = 0; i < num_rows; i++)
{
    array[i] = i + 1;
}

/* distribute arrays to each child process */

for (an_id = 1; an_id < num_procs; an_id++)
{
    start_row = an_id * avg_rows_per_process + 1;
    end_row = (an_id + 1) * avg_rows_per_process;

    if ((num_rows - end_row) < avg_rows_per_process)
    {
        end_row = num_rows - 1;
    }

    num_rows_to_send = end_row - start_row + 1;

    ierr = MPI_Send(&num_rows_to_send, 1, MPI_INT,
                    an_id, send_data_tag, MPI_COMM_WORLD);

    ierr = MPI_Send(&array[start_row], num_rows_to_send, MPI_INT,
                    an_id, send_data_tag, MPI_COMM_WORLD);
}

/* and calculate the sum of the values in the segment assigned
 * to the root process */

sum = 0;
for (i = 0; i < avg_rows_per_process + 1; i++)
{
    sum += array[i];
}

printf("Sum %i calculated by root process\n", sum);

/* and, finally, I collet the partial sums from the slave processes,
```

```
         * print them, and add them to the grand sum, and print it */

        for (an_id = 1; an_id < num_procs; an_id++)
        {

            ierr = MPI_Recv(&partial_sum, 1, MPI_LONG, MPI_ANY_SOURCE,
                            return_data_tag, MPI_COMM_WORLD, &status);

            sender = status.MPI_SOURCE;

            printf("Partial sum %i returned from process %i\n", partial_sum, sender);

            sum += partial_sum;
        }

        printf("Total sum is: %i\n", sum);
    }

    else
    {

        /* receive the segment, storing it in a local array, array1 */

        ierr = MPI_Recv(&num_rows_to_receive, 1, MPI_INT,
                        root_process, send_data_tag, MPI_COMM_WORLD, &status);

        ierr = MPI_Recv(&array2, num_rows_to_receive, MPI_INT,
                        root_process, send_data_tag, MPI_COMM_WORLD, &status);

        num_rows_received = num_rows_to_receive;

        /* Calculate the sum of my portion of the array */

        partial_sum = 0;
        for (i = 0; i < num_rows_received; i++)
        {
            partial_sum += array2[i];
        }

        /* send partial sum to the root process */

        ierr = MPI_Send(&partial_sum, 1, MPI_LONG, root_process,
                        return_data_tag, MPI_COMM_WORLD);
    }
    ierr = MPI_Finalize();
}
```



Experiment 5: Implement and design a parallel algorithm to sum an array and matrix multiplication and show logging and tracing MPI activity

```
#include <stdio.h>
#include <mpi.h>
#define NUM_ROWS_A 8
#define NUM_COLUMNS_A 10
#define NUM_ROWS_B 10
#define NUM_COLUMNS_B 8
#define MASTER_TO_SLAVE_TAG 1 // tag for messages sent from master to slaves
#define SLAVE_TO_MASTER_TAG 4 // tag for messages sent from slaves to master
void create_matrix();
void printArray();
int rank;
int size;
int i, j, k;
double A[NUM_ROWS_A][NUM_COLUMNS_A];
double B[NUM_ROWS_B][NUM_COLUMNS_B];
double result[NUM_ROWS_A][NUM_COLUMNS_B];
int low_bound;       // low bound of the number of rows of [A] allocated to a slave
int upper_bound;     // upper bound of the number of rows of [A] allocated to a slave
int portion;         // portion of the number of rows of [A] allocated to a slave
MPI_Status status;   // store status of a MPI_Recv
MPI_Request request; // capture request of a MPI_Send
int main(int argc, char *argv[])
{
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    if (rank == 0)
    { // master process
        create_matrix();
```

```
        for (i = 1; i < size; i++)
        {
            portion = (NUM_ROWS_A / (size - 1)); // portion without master
            low_bound = (i - 1) * portion;
            if (((i + 1) == size) && ((NUM_ROWS_A % (size - 1)) != 0))
            {                              // if rows of [A] cannot be equally divided
among slaves
                upper_bound = NUM_ROWS_A; // last slave gets all the  remaining rows

            }
            else
            {
                upper_bound = low_bound + portion; // rows of [A] are equally
divisable among slaves
            }
            MPI_Send(&low_bound, 1, MPI_INT, i, MASTER_TO_SLAVE_TAG,
                     MPI_COMM_WORLD);
            MPI_Send(&upper_bound, 1, MPI_INT, i, MASTER_TO_SLAVE_TAG + 1,
MPI_COMM_WORLD);
            MPI_Send(&A[low_bound][0], (upper_bound - low_bound) * NUM_COLUMNS_A,
                     MPI_DOUBLE, i, MASTER_TO_SLAVE_TAG + 2, MPI_COMM_WORLD);
        }
    }
    // broadcast [B] to all the slaves
    MPI_Bcast(&B, NUM_ROWS_B * NUM_COLUMNS_B, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    /* Slave process*/
    if (rank > 0)
    {
        MPI_Recv(&low_bound, 1, MPI_INT, 0, MASTER_TO_SLAVE_TAG,
                 MPI_COMM_WORLD,
                 &status);
        MPI_Recv(&upper_bound, 1, MPI_INT, 0, MASTER_TO_SLAVE_TAG + 1,
                 MPI_COMM_WORLD, &status);
        MPI_Recv(&A[low_bound][0], (upper_bound - low_bound) * NUM_COLUMNS_A,
                 MPI_DOUBLE, 0, MASTER_TO_SLAVE_TAG + 2, MPI_COMM_WORLD,
                 &status);
        printf("Process %d calculating for rows %d to %d of Matrix A\n", rank,
               low_bound, upper_bound);
        for (i = low_bound; i < upper_bound; i++)
        {
            for (j = 0; j < NUM_COLUMNS_B; j++)
            {
                for (k = 0; k < NUM_ROWS_B; k++)
                {
                    result[i][j] += (A[i][k] * B[k][j]);
                }
            }
```

```
        }
        MPI_Send(&low_bound, 1, MPI_INT, 0, SLAVE_TO_MASTER_TAG,
                 MPI_COMM_WORLD);
        MPI_Send(&upper_bound, 1, MPI_INT, 0, SLAVE_TO_MASTER_TAG + 1,
MPI_COMM_WORLD);
        MPI_Send(&result[low_bound][0], (upper_bound - low_bound) * NUM_COLUMNS_B,
                 MPI_DOUBLE, 0, SLAVE_TO_MASTER_TAG + 2, MPI_COMM_WORLD);
    }
    /* master gathers processed work*/
    if (rank == 0)
    {
        for (i = 1; i < size; i++)
        {
            MPI_Recv(&low_bound, 1, MPI_INT, i, SLAVE_TO_MASTER_TAG,
                     MPI_COMM_WORLD,
                     &status);
            MPI_Recv(&upper_bound, 1, MPI_INT, i, SLAVE_TO_MASTER_TAG + 1,
                     MPI_COMM_WORLD, &status);
            MPI_Recv(&result[low_bound][0], (upper_bound - low_bound) * NUM_COLUMNS_B,
MPI_DOUBLE, i, SLAVE_TO_MASTER_TAG + 2, MPI_COMM_WORLD, &status);
        }
        printArray();
    }
    MPI_Finalize();
    return 0;
}
void create_matrix()
{
    for (i = 0; i < NUM_ROWS_A; i++)
    {
        for (j = 0; j < NUM_COLUMNS_A; j++)
        {
            A[i][j] = i + j;
        }
    }
    for (i = 0; i < NUM_ROWS_B; i++)
    {
        for (j = 0; j < NUM_COLUMNS_B; j++)
        {
            B[i][j] = i * j;
        }
    }
}
void printArray()
{
    printf("Given matrix A is: \n");
    for (i = 0; i < NUM_ROWS_A; i++)
```

```c
    {
        printf("\n");
        for (j = 0; j < NUM_COLUMNS_A; j++)
            printf("%8.2f ", A[i][j]);
    }
    printf("\n\n\n");
    printf("Given matrix B is: \n");
    for (i = 0; i < NUM_ROWS_B; i++)
    {
        printf("\n");
        for (j = 0; j < NUM_COLUMNS_B; j++)
            printf("%8.2f ", B[i][j]);
    }
    printf("\n\n\n");
    printf("Final Multiplied Matrix is: \n");
    for (i = 0; i < NUM_ROWS_A; i++)
    {
        printf("\n");
        for (j = 0; j < NUM_COLUMNS_B; j++)
            printf("%8.2f ", result[i][j]);
    }
    printf("\n\n");
}
```

```
shreyans@shreyans: /media/shreyans/New Volume/NUSUT/Course Material Sem 6/
shreyans@shreyans:/media/shreyans/New Volume/NSUT/Course Material Sem 6/HPC/Prog$ mpicc Program5.c -o o5.out
shreyans@shreyans:/media/shreyans/New Volume/NSUT/Course Material Sem 6/HPC/Prog$ mpirun -np 4 ./o5.out
Process 1 calculating for rows 0 to 2 of Matrix A
Process 2 calculating for rows 2 to 4 of Matrix A
Given matrix A is:

    0.00     1.00     2.00     3.00     4.00     5.00     6.00     7.00     8.00     9.00
    1.00     2.00     3.00     4.00     5.00     6.00     7.00     8.00     9.00    10.00
    2.00     3.00     4.00     5.00     6.00     7.00     8.00     9.00    10.00    11.00
    3.00     4.00     5.00     6.00     7.00     8.00     9.00    10.00    11.00    12.00
    4.00     5.00     6.00     7.00 Process 3 calculating for rows 4 to 8 of Matrix A
    8.00     9.00    10.00    11.00    12.00    13.00
    5.00     6.00     7.00     8.00     9.00    10.00    11.00    12.00    13.00    14.00
    6.00     7.00     8.00     9.00    10.00    11.00    12.00    13.00    14.00    15.00
    7.00     8.00     9.00    10.00    11.00    12.00    13.00    14.00    15.00    16.00


Given matrix B is:

    0.00     0.00     0.00     0.00     0.00     0.00     0.00     0.00
    0.00     1.00     2.00     3.00     4.00     5.00     6.00     7.00
    0.00     2.00     4.00     6.00     8.00    10.00    12.00    14.00
    0.00     3.00     6.00     9.00    12.00    15.00    18.00    21.00
    0.00     4.00     8.00    12.00    16.00    20.00    24.00    28.00
    0.00     5.00    10.00    15.00    20.00    25.00    30.00    35.00
    0.00     6.00    12.00    18.00    24.00    30.00    36.00    42.00
    0.00     7.00    14.00    21.00    28.00    35.00    42.00    49.00
    0.00     8.00    16.00    24.00    32.00    40.00    48.00    56.00
    0.00     9.00    18.00    27.00    36.00    45.00    54.00    63.00


Final Multiplied Matrix is:

    0.00   285.00   570.00   855.00  1140.00  1425.00  1710.00  1995.00
    0.00   330.00   660.00   990.00  1320.00  1650.00  1980.00  2310.00
    0.00   375.00   750.00  1125.00  1500.00  1875.00  2250.00  2625.00
    0.00   420.00   840.00  1260.00  1680.00  2100.00  2520.00  2940.00
    0.00   465.00   930.00  1395.00  1860.00  2325.00  2790.00  3255.00
    0.00   510.00  1020.00  1530.00  2040.00  2550.00  3060.00  3570.00
    0.00   555.00  1110.00  1665.00  2220.00  2775.00  3330.00  3885.00
    0.00   600.00  1200.00  1800.00  2400.00  3000.00  3600.00  4200.00

shreyans@shreyans:/media/shreyans/New Volume/NSUT/Course Material Sem 6/HPC/Prog$
```

Experiment 6: Write a program with OPENMP to implement loop work-sharing

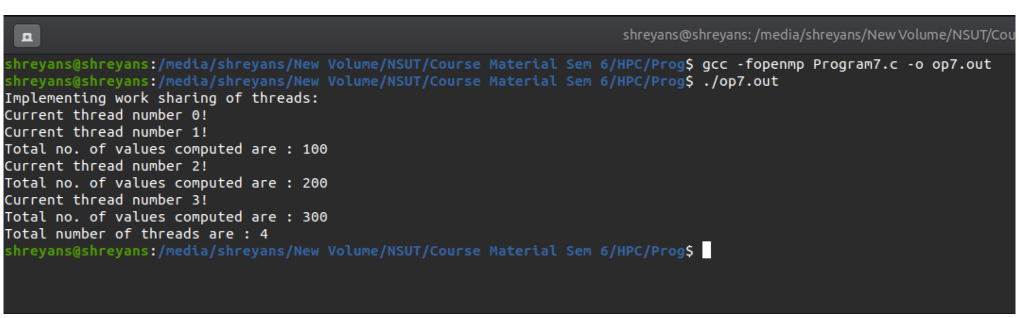
```
#include <omp.h>
#include <stdio.h>

void reset_freq(int *freq, int THREADS)
{
    for (int i = 0; i < THREADS; i++)
    {
        freq[i] = 0;
    }
}
int main(int *argc, char **argv)
{
    int n, THREADS, i;
    printf("Enter the number of iterations :");
    scanf("%d", &n);
    printf("Enter the number of threads (max 8): ");
    scanf("%d", &THREADS);
    int freq[6];
    reset_freq(freq, THREADS);
    // simple parallel for with unequal iterations
#pragma omp parallel for num_threads(THREADS)
    for (i = 0; i < n; i++)
    {
        freq[omp_get_thread_num()]++;
    }
#pragma omp barrier
    printf("\nIn default scheduling, we have the following thread distribution :-\n");
    for (int i = 0; i < THREADS; i++)
    {
        printf("Thread No. %d : %d iterations\n", i, freq[i]);
    }
    // using static scheduling
    int CHUNK;
    printf("\nUsing static scheduling...\n");
    printf("Enter the chunk size :");
    scanf("%d", &CHUNK);

    reset_freq(freq, THREADS);
#pragma omp parallel for num_threads(THREADS) schedule(static, CHUNK)
    for (i = 0; i < n; i++)
    {
        freq[omp_get_thread_num()]++;
    }
```

```c
#pragma omp barrier
    printf("\nIn static scheduling, we have the following thread distribution :- \n");
    for (int i = 0; i < THREADS; i++)
    {
        printf("Thread No. %d : %d iterations\n", i, freq[i]);
    }
    // auto scheduling depending on the compiler
    printf("\nUsing automatic scheduling...\n");
    reset_freq(freq, THREADS);
#pragma omp parallel for num_threads(THREADS) schedule(static)
    for (i = 0; i < n; i++)
    {
        freq[omp_get_thread_num()]++;
    }
#pragma omp barrier
    printf("In auto scheduling, we have the following thread distribution :-\n");
    for (int i = 0; i < THREADS; i++)
    {
        printf("Thread No. %d : %d iterations\n", i, freq[i]);
    }
    return 0;
}
```

```
shreyans@shreyans: /media/shreyans/New Volume/NSUT/Course Ma
shreyans@shreyans:/media/shreyans/New Volume/NSUT/Course Material Sem 6/HPC/Prog$ gcc -fopenmp Program6.c -o op6.out
shreyans@shreyans:/media/shreyans/New Volume/NSUT/Course Material Sem 6/HPC/Prog$ ./op6.out
Enter the number of iterations :4
Enter the number of threads (max 8): 4

In default scheduling, we have the following thread distribution :-
Thread No. 0 : 1 iterations
Thread No. 1 : 1 iterations
Thread No. 2 : 1 iterations
Thread No. 3 : 1 iterations

Using static scheduling...
Enter the chunk size :2

In static scheduling, we have the following thread distribution :-
Thread No. 0 : 2 iterations
Thread No. 1 : 2 iterations
Thread No. 2 : 0 iterations
Thread No. 3 : 0 iterations

Using automatic scheduling...
In auto scheduling, we have the following thread distribution :-
Thread No. 0 : 1 iterations
Thread No. 1 : 1 iterations
Thread No. 2 : 1 iterations
Thread No. 3 : 1 iterations
shreyans@shreyans:/media/shreyans/New Volume/NSUT/Course Material Sem 6/HPC/Prog$
```

Experiment 7: Write a c program with OPENMP to implement sections work-sharing

```c
#include <omp.h>
#include <stdio.h>
int main(int *argc, char **argv)
{
    int num_threads, THREAD_COUNT = 4;
    int thread_ID;
    int section_sizes[4] = {
        0, 100, 200, 300};
    printf("Implementing work sharing of threads:\n");
#pragma omp parallel private(thread_ID) num_threads(THREAD_COUNT)
    {
        // private means each thread will have a private variable
        // thread_ID
        thread_ID = omp_get_thread_num();
        printf("Current thread number %d!\n", thread_ID);
        int value_count = 0;
        if (thread_ID > 0)
        {
            int work_load = section_sizes[thread_ID];
            // each thread has a different section size
            for (int i = 0; i < work_load; i++)
                value_count++;
            printf("Total no. of values computed are : %d\n",
                   value_count);
        }
#pragma omp barrier
        if (thread_ID == 0)
        {
            printf("Total number of threads are : %d\n",
                   omp_get_num_threads());
        }
    }
    return 0;
}
```



Experiment 8: Write a program to illustrate process synchronization and collective data movements

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int thread_count;
// gcc name_of_file.c -o name_of_exe -lpthread (link p thread)
// necessary for referencing in the thread
struct arguments
{
    int size;
    int *arr1;
    int *arr2;
    int *dot;
};

// function to parallelize`
void *add_into_one(void *arguments);
// util
void print_vector(int n, int *arr)
{
    printf("[ ");
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);
    printf("] \n");
}
// main driver function of the program
int main(int argc, char *argv[])
{
    long thread;
    pthread_t *thread_handles;
    thread_count = 2; // using 2 threads only
    // get the thread handles equal to total num
    thread_handles = (pthread_t *)malloc(thread_count * sizeof(pthread_t));
    printf("Enter the vector size: ");
    int n;
    scanf("%d", &n);
    printf("Enter vectors maximal element: ");
    int max_val;
    scanf("%d", &max_val);
    struct arguments *args[2]; // array of pointer to structure
    for (int i = 0; i < 2; i++)
    {
```

```c
        // allocate for the struct
        args[i] = (struct arguments *)malloc(sizeof(struct arguments) * 1);
        // allocate for the arrays
        args[i]->size = n;
        args[i]->arr1 = (int *)malloc(sizeof(int) * n);
        args[i]->arr2 = (int *)malloc(sizeof(int) * n);
        args[i]->dot = (int *)malloc(sizeof(int) * n);
        for (int j = 0; j < n; j++)
        {
            args[i]->arr1[j] = rand() % max_val;
            args[i]->arr2[j] = rand() % max_val;
        }
    }
    printf("Vectors are : \n");
    print_vector(n, args[0]->arr1);
    print_vector(n, args[0]->arr2);
    print_vector(n, args[1]->arr1);
    print_vector(n, args[1]->arr2);
    int result[n];

    memset(result, 0, n * sizeof(int));

    // note : we need to manually startup our threads
    // for a particular function which we want to execute in
    // the thread
    for (thread = 0; thread < thread_count; thread++)
    {
        printf("Multiplying %ld and %ld with thread %ld...\n", thread + 1,
               thread + 2,
               thread);
        pthread_create(&thread_handles[thread], NULL, add_into_one,
                       (void *)args[thread]);
    }
    printf("Currently in the main thread\n");
    // wait for completion
    for (thread = 0; thread < thread_count; thread++)
        pthread_join(thread_handles[thread], NULL);
    for (int i = 0; i < 2; i++)
    {
        printf("Multiplication for vector %d and %d \n", i + 1, i + 2);
        print_vector(n, args[i]->dot);
        printf("\n");
    }
    free(thread_handles);
    // now compute the summation of results
    for (int i = 0; i < n; i++)
        result[i] = args[0]->dot[i] + args[1]->dot[i];
```

```c
    printf("Result is : \n");
    print_vector(n, result);
    return 0;
}
void *add_into_one(void *argument)
{
    // de reference the argument
    struct arguments *args = (struct arguments *)argument;
    // compute the dot product into the
    // array dot
    int n = args->size;
    for (int i = 0; i < n; i++)
        args->dot[i] = args->arr1[i] * args->arr2[i];
    return NULL;
}
```

```
shreyans@shreyans: /media/shreyans/New Vol
shreyans@shreyans:/media/shreyans/New Volume/NSUT/Course Material Sem 6/HPC/Prog$ gcc Program8.c -lpthread
shreyans@shreyans:/media/shreyans/New Volume/NSUT/Course Material Sem 6/HPC/Prog$ ./a.out
Enter the vector size: 7
Enter vectors maximal element: 34
Vectors are :
[ 27 9 15 20 11 22 20 ]
[ 32 17 3 14 19 31 19 ]
[ 29 28 6 1 19 14 8 ]
[ 22 30 14 8 5 22 21 ]
Multiplying 1 and 2 with thread 0...
Multiplying 2 and 3 with thread 1...
Currently in the main thread
Multiplication for vector 1 and 2
[ 864 153 45 280 209 682 380 ]

Multiplication for vector 2 and 3
[ 638 840 84 8 95 308 168 ]

Result is :
[ 1502 993 129 288 304 990 548 ]
shreyans@shreyans:/media/shreyans/New Volume/NSUT/Course Material Sem 6/HPC/Prog$
```