# High Performance Computing (COCSC18)

## Practical Lab File

Submitted by:

Ashish Kumar 2019UCO1518

COE-I

# Table of content

1. Run a basic hello world program using pThreads.

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
int thread_count; // this global variable is shared by all threads
// compiling information -
// gcc name_of_file.c -o name_of_exe -lpthread (link p thread)
// this function is what we want to parallelize
void *Hello(void *rank);
// main driver function of the program
int main(int argc, char *argv[])
{
    long thread;
// /* Use long in case of a 64-bit system */
    pthread_t *thread_handles;
// /* Get number of threads from command line */
// since the command line arg would be string,
// we convert to the long value
    thread_count = strtol(argv[1], NULL, 10);
// get the thread handles equal to total num of threads
    thread_handles = malloc(thread_count * sizeof(pthread_t));
// note : we need to manually startup our threads
// for a particular function which we want to execute in
// the thread
// void* is a pretty nice concept,
// it is essentially a pointer to
// ANY type of memory,3
// you just dereference it with the type you expect
// it to be
    for (thread = 0; thread < thread_count; thread++)
        pthread_create(&thread_handles[thread], NULL, Hello, (void *)thread);
// Thread placement on cores is done by OS
    printf("Hello from the main thread\n");
    for (thread = 0; thread < thread_count; thread++)
        pthread_join(thread_handles[thread], NULL);
    free(thread_handles);
    return 0;
}
// /* main */
void *Hello(void *rank) // void * means a pointer, can be of any type
{
// Each thread has its own stack
// note : local variables of a thread are
// private to the thread and each thread
// will have its own local copy
    long my_rank = (long)rank;
// /* Use long in case of 64-bit system */
```

```
    printf("Hello from thread %ld of %d\n", my_rank, thread_count);
    return NULL;
}
```

Output

```
gourav@singal:~/Desktop/code-forces$ gcc prac1.c -o ./thread-basic -lpthread
gourav@singal:~/Desktop/code-forces$ ./thread-basic 5
Hello from thread 0 of 5
Hello from thread 1 of 5
Hello from thread 4 of 5
Hello from the main thread
Hello from thread 2 of 5
Hello from thread 3 of 5
```

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

Code

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
// it is a message passing interface
// processes live inside a COMM_WORLD
// processes are LIVING, and exist in a COMMUNICATOR

int main(int argc, char **argv)
{
    // start the MPI code     MPI_Init(NULL, NULL);
    int num_procs; // to store the size of the world / num of
procs     MPI_Comm_size(MPI_COMM_WORLD, &num_procs);     int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);     if (rank == 0)
    {
        // read the array         int n;
        // printf("Enter number of elements : ");         scanf("%d", &n);
        int arr[n];
        for (int i = 0; i < n; i++)
        {
            arr[i] = rand() % 10000 + 1;
        }
        printf("Array is -\n [ ");
        for (int i = 0; i < n; i++)
        {
            printf("%d ", arr[i]);
        }
        printf("]\n");
        int elem_to_send = n / 2;         if (n % 2)
            elem_to_send++;         // send the size
        MPI_Send(&elem_to_send, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
        // send the array
        MPI_Send(&arr[n / 2], elem_to_send, MPI_INT, 1, 1,
MPI_COMM_WORLD);         float t1 = clock();         int local =
0;          for (int i = 0; i < n / 2; i++)             local = local +
arr[i];          int s_rec = 0;         float t2 = clock();
        printf("Time taken by process %d : %f\n", rank, (t2 - t1) /
CLOCKS_PER_SEC);
        // recv the data into the local var s_rec
        MPI_Recv(&s_rec, 1, MPI_INT, 1, 2, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);         local = local + s_rec;
        printf("Total sum of array is %d\n", local);
    }
    else
```

```
    {
        // recieve the size of elements
        float t1 = clock();         int size;
        MPI_Recv(&size, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

        int arr[size];
        MPI_Recv(arr, size, MPI_INT, 0, 1, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);          float t2 = clock();
        printf("Total time for recieving : %f", (t2 - t1) /
CLOCKS_PER_SEC);          // lol, the time for recieving the elements is a
thousand times slower
        // than the processing, lol waste         t1 = clock();         int
local = 0;          for (int i = 0; i < size; i++)             local = local +
arr[i];
        printf("\nProcess %d sending sum %d back to main...\n", rank,
local);          t2 = clock();
        printf("Time taken by process for addition %d : %f\n", rank, (t2 - t1)
/ CLOCKS_PER_SEC);         MPI_Send(&local, 1, MPI_INT, 0, 2, MPI_COMM_WORLD);
    }
    MPI_Finalize();
}
```

Output

```
gourav@singal:~/Desktop/code-forces$ mpicc prac2.c -o add
gourav@singal:~/Desktop/code-forces$ mpirun -np 2 ./add
10
Enter number of elements : Array is -
 [ 9384 887 2778 6916 7794 8336 5387 493 6650 1422 ]
Total time for recieving : 5.135460
Process 1 sending sum 22288 back to main...
Time taken by process for addition 1 : 0.000006
Time taken by process 0 : 0.000001
Total sum of array is 50047
```

3.) Compute the sum of all the elements of an array using p processors.

Code:

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h> #include <time.h>
int main(int argc, char **argv)
{
    // start the MPI code     MPI_Init(NULL, NULL);     int num_procs;
    MPI_Comm_size(MPI_COMM_WORLD, &num_procs);     int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);     if (rank == 0)
    {
        // read the array          int n;
        printf("Enter number of elements : ");
        scanf("%d", &n);
        int arr[n];
        for (int i = 0; i < n; i++)
        {
            arr[i] = rand() % 10 + 1;
        }
        printf("Array is -\n [ ");
        for (int i = 0; i < n; i++)
        {
            printf("%d ", arr[i]);
        }
        printf("]\n");
        int elem_to_send = n / num_procs;

        int tag = 0;
        for (int i = 1; i < num_procs; i++)
        {   // send the size
            if (i != num_procs - 1)
            {
                elem_to_send = n / num_procs;
                MPI_Send(&elem_to_send, 1, MPI_INT, i, i + num_procs,
MPI_COMM_WORLD);
                MPI_Send(&arr[i * (elem_to_send)], elem_to_send, MPI_INT, i, i
+ num_procs + 1, MPI_COMM_WORLD);
                continue;
            }
            // elements would be changed
            elem_to_send = n / num_procs + n % num_procs;
            MPI_Send(&elem_to_send, 1, MPI_INT, i, i + num_procs,
MPI_COMM_WORLD);
            MPI_Send(&arr[(num_procs - 1) * (n / num_procs)], elem_to_send,
MPI_INT, i, i + num_procs + 1, MPI_COMM_WORLD);
            // send the array
```

```c
        }
        int ans = 0;
        for (int i = 0; i < n / num_procs; i++)
            ans += arr[i];
        // recv the data into the local var s_rec         int s_rec;
        for (int i = 1; i < num_procs; i++)
        {
            s_rec = 0;
            MPI_Recv(&s_rec, 1, MPI_INT, i, i + num_procs + 2, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
            ans += s_rec;
        }
        printf("Total sum of array is %d\n", ans);
    }
    else
    {
        // receive the size of elements         int size;
        MPI_Recv(&size, 1, MPI_INT, 0, rank + num_procs, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);

        int arr[size];
        MPI_Recv(arr, size, MPI_INT, 0, rank + num_procs + 1, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
        int local = 0;
        for (int i = 0; i < size; i++)
            local = local + arr[i];
        printf("\nProcess %d sending sum %d back to main...\n", rank, local);
        MPI_Send(&local, 1, MPI_INT, 0, rank + num_procs + 2, MPI_COMM_WORLD);
    }
    MPI_Finalize();
}
```

Output

```
gourav@singal:~/Desktop/code-forces$ mpicc prac3.c -o addp
gourav@singal:~/Desktop/code-forces$ mpirun -np 4 ./addp
30
Enter number of elements : Array is -
 [ 4 7 8 6 4 6 7 3 10 2 3 8 1 10 4 7 1 7 3 7 2 9 8 10 3 1 3 4 8 6 ]

Process 1 sending sum 37 back to main...

Process 2 sending sum 31 back to main...

Process 3 sending sum 52 back to main...
Total sum of array is 162
```

4.)Write a program to illustrate basic MPI communication routines.

Code:

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

int main(int argc, char **argv)
{
    // Initialize the MPI environment
    MPI_Init(NULL, NULL);
    // Get the number of processes
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);
    // COMM_WORLD is the communicator world
    // a communicator is a group of processes
    // communicating with each other and HAVE BEEN
    // init
    // Get the rank of the process     int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
    // Get the name of the processor
    char processor_name[MPI_MAX_PROCESSOR_NAME];     int name_len;
    MPI_Get_processor_name(processor_name, &name_len);
    printf("Hello world from process %s, rank %d out of %d processes\n\n",
processor_name, world_rank, world_size);
    if (world_rank == 0)
    {
        char *message = "Hello!";
        MPI_Send(message, 6, MPI_CHAR, 1, 0, MPI_COMM_WORLD);
    }
    else
    {
        char message[6];
        MPI_Recv(message, 6, MPI_CHAR, 0, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
        printf("Message received!\n");         printf("Message is : %s\n",
message);
    }
    // write message send and recieve here...     // Print off a hello world
message     // Finalize the MPI environment.     MPI_Finalize();     return 0;
}
```

Output:

```
gourav@singal:~/Desktop/code-forces$ mpicc prac4.c -o basicMpi
gourav@singal:~/Desktop/code-forces$ mpirun -np 2 ./basicMpi
Hello world from process singal, rank 0 out of 2 processes

Hello world from process singal, rank 1 out of 2 processes

Message received!
Message is : Hello!singal
```

5.) Design a parallel program for summing up an array, matrix multiplication and show logging and tracing MPI activity.

Code:

```
#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
divisible 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("The matrix A is: \n");
    for (i = 0; i < NUM_ROWS_A; i++)
    {
        printf("\n");
        for (j = 0; j < NUM_COLUMNS_A; j++)
            printf("%8.2f ", A[i][j]);
    }
    printf("\n\n\n");
    printf("The 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("The result 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");
}
```

Output :

```
gourav@singal:~/Desktop/code-forces$ mpicc prac5.c -o prac5
gourav@singal:~/Desktop/code-forces$ mpirun -np 2 ./prac5
Process 1 calculating for rows 0 to 8 of Matrix A
The 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     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


The 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


The result 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
```

6.) Write a C program with openMP to implement loop work sharing.

Code:

```
#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 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[THREADS];
    reset_freq(freq, THREADS);
    // simple parallel for with unequal iterations
    #pragma omp parallel for num_threads(THREADS)
    for (i = 0; i < n; i++)
    {

        // printf("Thread num %d executing iter %d\n", omp_get_thread_num(),
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 %d : %d iters\n", i, freq[i]);
    }
    // using static scheduling
    int CHUNK;
    printf("\nUsing static scheduling...\n");
    printf("Enter the chunk size :");
    scanf("%d", &CHUNK);
    // using a static, round robin schedule for the loop
iterations     reset_freq(freq, THREADS);
    // useful when the workload is ~ same across each thread, not when
otherwise
    #pragma omp parallel for num_threads(THREADS) schedule(static, CHUNK)
    for (i = 0; i < n; i++)
    {
        // printf("Thread num %d executing iter %d\n", omp_get_thread_num(),
i);          freq[omp_get_thread_num()]++;
```

```
    }
    #pragma omp barrier
    printf("\nIn static scheduling, we have the following thread distribution
:- \n");
    for (int i = 0; i < THREADS; i++)
    {
        printf("Thread %d : %d iters\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(auto)
    for (i = 0; i < n; i++)
    {
        // printf("Thread num %d executing iter %d\n", omp_get_thread_num(),
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 %d : %d iters\n", i, freq[i]);
    }
    return 0;
}
```

Output :

```
gourav@singal:~/Desktop/code-forces$ gcc -fopenmp prac6.c -o prac6
gourav@singal:~/Desktop/code-forces$ ./prac6
Enter the number of iterations :100
Enter the number of threads (max 8): 6

In default scheduling, we have the following thread distribution :-
Thread 0 : 17 iters
Thread 1 : 17 iters
Thread 2 : 17 iters
Thread 3 : 17 iters
Thread 4 : 16 iters
Thread 5 : 16 iters

Using static scheduling...
Enter the chunk size :20

In static scheduling, we have the following thread distribution :-
Thread 0 : 20 iters
Thread 1 : 20 iters
Thread 2 : 20 iters
Thread 3 : 20 iters
Thread 4 : 20 iters
Thread 5 : 0 iters

Using automatic scheduling...
In auto scheduling, we have the following thread distribution :-
Thread 0 : 17 iters
Thread 1 : 17 iters
Thread 2 : 17 iters
Thread 3 : 17 iters
Thread 4 : 16 iters
Thread 5 : 16 iters
```

7.) Write a C program with openMP to implement sections work
Code:

```c
#include <omp.h>
#include <stdio.h>
int main(int *argc, char **argv)
{   // invocation of the main program
// use the fopenmp flag for compiling
    int num_threads, THREAD_COUNT = 4;
    int thread_ID;
    int section_sizes[4] = {
        0, 100, 200, 300
    };
    printf("Work load 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("I am 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("Number of values computed : %d\n", value_count);
        }
        #pragma omp barrier
        if (thread_ID == 0)
        {
            printf("Total number of threads are %d", omp_get_num_threads());
        }
    }
    return 0;
}
```

Output:

```
gourav@singal:~/Desktop/code-forces$ gcc -fopenmp prac7.c -o thread-section
gourav@singal:~/Desktop/code-forces$ ./thread-section
Work load sharing of threads...
I am thread number 0!
I am thread number 2!
Number of values computed : 200
I am thread number 3!
Number of values computed : 300
I am thread number 1!
Number of values computed : 100
```

8.) Write a program to illustrate process synchronization and collective data movements.

Code:

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int thread_count; // this global variable is shared by all threads
// compiling information -
// 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;
    // /  Use long in case of a 64-bit system   /
    pthread_t *thread_handles;
    thread_count = 2; // using 2 threads only     // get the thread handles
equal to total num  // of threads
    thread_handles = malloc(thread_count * sizeof(pthread_t));
    printf("Enter the size of the vectors : ");     int n;
    scanf("%d", &n);
    printf("Enter the max_val of the vectors : ");     int max_val;
    scanf("%d", &max_val);
    struct arguments *args[2]; // array of pointer to structure
    // each element is a pointer
    for (int i = 0; i < 2; i++)
    {
        // allocate for the struct
```

```
        args[i] = malloc(sizeof(struct arguments) * 1);
        // allocate for the arrays
        args[i]->size = n;
        args[i]->arr1 = malloc(sizeof(int) * n);
        args[i]->arr2 = malloc(sizeof(int) * n);
        args[i]->dot = 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("Hello from 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];
    printf("Result is : \n");
    print_vector(n, result);     return 0;
}
void *add_into_one(void *argument)
{
    // de reference the argument
```

```
    struct arguments *args = 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;
}
```

Output

```
gourav@singal:~/Desktop/code-forces$ gcc prac8.c -o  prac8 -lpthread
gourav@singal:~/Desktop/code-forces$ ./prac8
Enter the size of the vectors : 5
Enter the max_val of the vectors : 3
Vectors are :
[ 1 0 2 1 0 ]
[ 1 1 1 0 1 ]
[ 2 2 2 0 1 ]
[ 1 1 1 0 1 ]
Multiplying 1 and 2 with thread 0...
Multiplying 2 and 3 with thread 1...
Hello from the main thread
Multiplication for vector 1 and 2
[ 1 0 2 0 0 ]

Multiplication for vector 2 and 3
[ 2 2 2 0 1 ]

Result is :
[ 3 2 4 0 1 ]
```