High Performance

Computing (COCSC18)

Practical Lab File



Submitted by:

Ashish Kumar 2019UCO1518

COE-I

# Table of content

|  |  |  |
| --- | --- | --- |
| **S.no.** | **Topic** | **Page no.** |
| 1. | Run a basic hello world program using pthreads | 3 |
| 2. | Run a program to find the sum of all elements of an array using 2 processors | 5 |
| 3. | Compute the sum of all the elements of an array using p processors | 7 |
| 4. | Write a program to illustrate basic MPI communication routines | 9 |
| 5. | Design a parallel program for summing up an array, matrix multiplication and show logging and tracing MPI activity | 10 |
| 6. | Write a C program with openMP to implement loop work sharing | 17 |
| 7. | Write a C program with openMP to implement sections work sharing | 19 |
| 8. | Write a program to illustrate process synchronization and collective data movements | 21 |

1. Run a basic hello world program using pThreads.

#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;

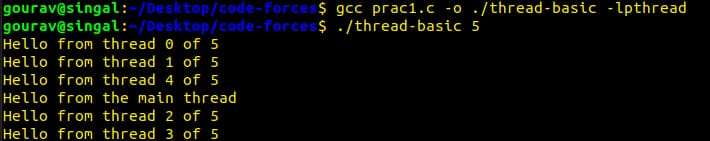
// /∗ Use long in case of 64−bit system ∗/

    printf("Hello from thread %ld of %d\n", my\_rank, thread\_count);

    return NULL;

}

Output



1. 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();

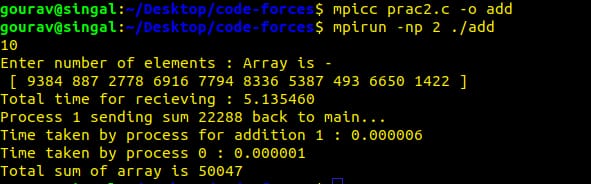
        printf("Time taken by process for addition %d : %f\n", rank, (t2 - t1) / CLOCKS\_PER\_SEC);         MPI\_Send(&local, 1, MPI\_INT, 0, 2, MPI\_COMM\_WORLD);

    }

    MPI\_Finalize();

}

Output



1. 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

        }

        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);

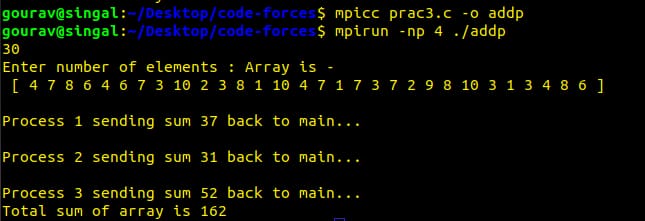
        MPI\_Send(&local, 1, MPI\_INT, 0, rank + num\_procs + 2, MPI\_COMM\_WORLD);

    }

    MPI\_Finalize();

}

Output



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);

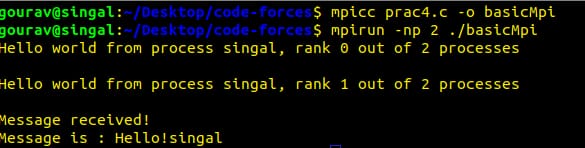
        printf("Message received!\n");         printf("Message is : %s\n", message);

    }

    // write message send and recieve here...     // Print off a hello world message     // Finalize the MPI environment.     MPI\_Finalize();     return 0;

}

Output:



1. 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++)

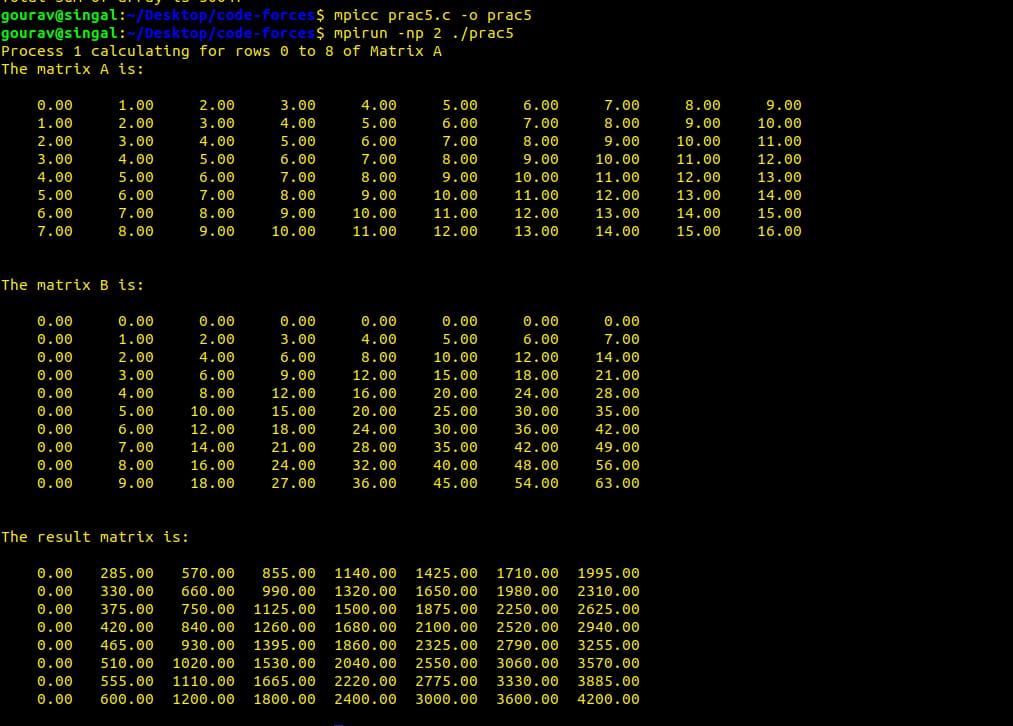
            printf("%8.2f ", result[i][j]);

    }

    printf("\n\n");

}

Output :



1. 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++)

    {

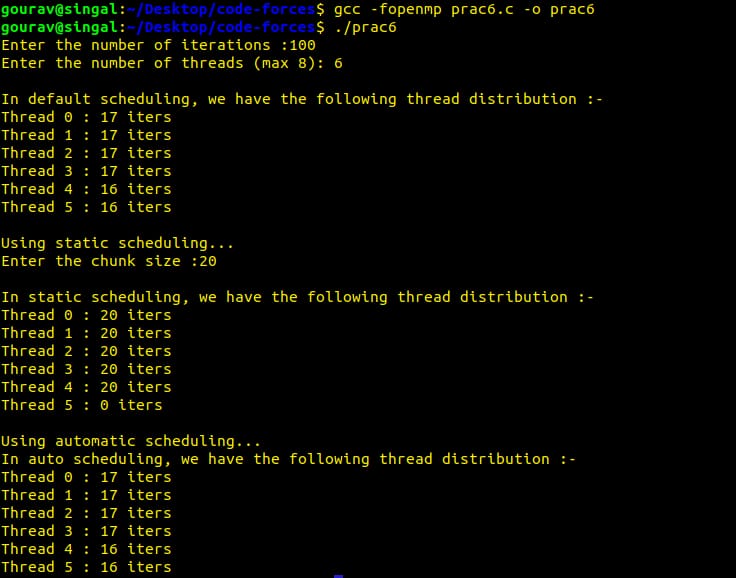
        printf("Thread %d : %d iters\n", i, freq[i]);

    }

    return 0;

}

Output :



1. Write a C program with openMP to implement sections work

Code:

#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());

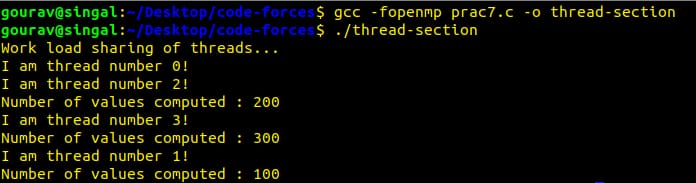
        }

    }

    return 0;

}

Output:



1. 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

