NETAJI SUBHAS UNIVERSITY OF TECHNOLOGY

High Performance Computing [COCSC18]

Lab File

SUBMITTED BY :-

Shivam Singla
2019UCO1526
COE-1

Index

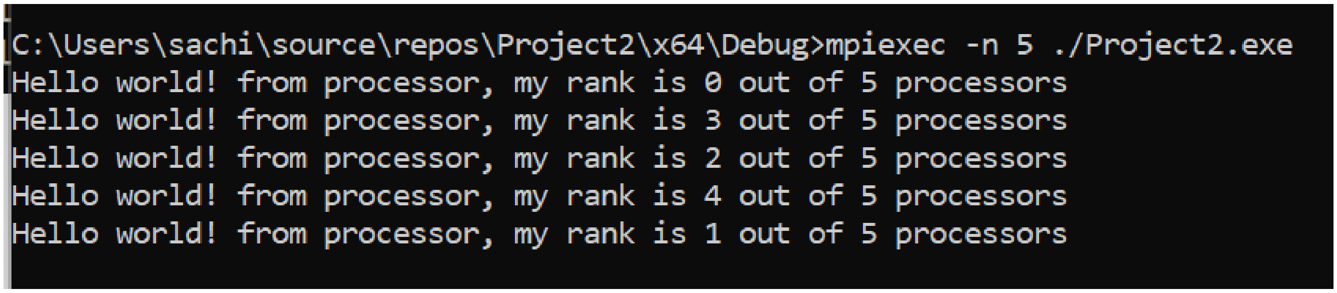
Experiment 1

Run a basic hello world program using pthreads

Code:

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

int main(int argc, char** argv) {
    MPI_Init(NULL, NULL);
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);
    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    int name_len;
    MPI_Get_processor_name(processor_name, &name_len);
    printf("Hello world! from processor, my rank is %d out of %d
processors\n", world_rank, world_size);
    MPI_Finalize();
}
```

Output:



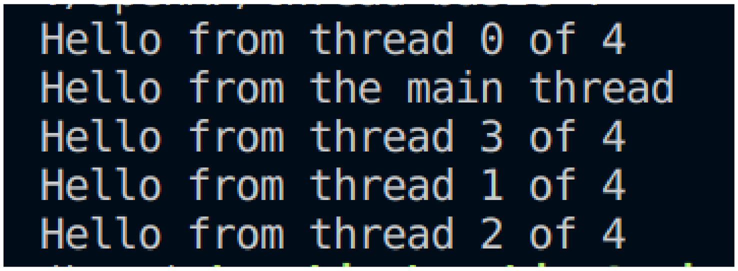
```
C:\Users\sachi\source\repos\Project2\x64\Debug>mpiexec -n 5 ./Project2.exe
Hello world! from processor, my rank is 0 out of 5 processors
Hello world! from processor, my rank is 3 out of 5 processors
Hello world! from processor, my rank is 2 out of 5 processors
Hello world! from processor, my rank is 4 out of 5 processors
Hello world! from processor, my rank is 1 out of 5 processors
```

Using Pthreads

Code:

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
int thread_count;
void* Hello(void* rank);
int main(int argc, char* argv[])
{
        long thread;
        pthread_t* thread_handles;
        thread_count = strtol(argv[1], NULL, 10);
        thread_handles = malloc(thread_count * sizeof(pthread_t));
        for (thread = 0; thread < thread_count; thread++)
                pthread_create(&thread_handles[thread], NULL, Hello, (void*)thread);
        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;
}
void* Hello(void* rank)
{
        long my_rank = (long)rank;
        printf("Hello from thread %ld of %d\n", my_rank, thread_count);
        return NULL;
}
```

Output:



```
Hello from thread 0 of 4
Hello from the main thread
Hello from thread 3 of 4
Hello from thread 1 of 4
Hello from thread 2 of 4
```

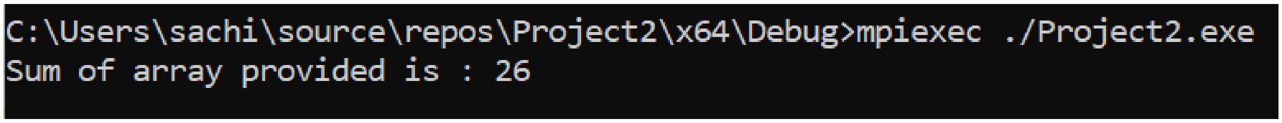
Experiment 2

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>

#define n 9

int a[] = { 1, 5, 2, 6, 2, 0, 1, 9 };
int a2[1000];

int main(int argc, char* argv[])
{

    int pid, np,
        elements_per_process,
        n_elements_recieved;
    MPI_Status status;
    MPI_Init(&argc, &argv);

    MPI_Comm_rank(MPI_COMM_WORLD, &pid);
    MPI_Comm_size(MPI_COMM_WORLD, &np);

    if (pid == 0) {
        int index, i;
        elements_per_process = n / np;
        if (np > 1) {
            for (i = 1; i < np - 1; i++) {
                index = i * elements_per_process;

                MPI_Send(&elements_per_process,
                    1, MPI_INT, i, 0,
                    MPI_COMM_WORLD);
                MPI_Send(&a[index],
                    elements_per_process,
                    MPI_INT, i, 0,
                    MPI_COMM_WORLD);
            }

            index = i * elements_per_process;
            int elements_left = n - index;

            MPI_Send(&elements_left,
                1, MPI_INT,
                i, 0,
                MPI_COMM_WORLD);
            MPI_Send(&a[index],
                elements_left,
                MPI_INT, i, 0,
                MPI_COMM_WORLD);
        }

        int sum = 0;
        for (i = 0; i < elements_per_process; i++)
            sum += a[i];

        int tmp;
        for (i = 1; i < np; i++) {
            MPI_Recv(&tmp, 1, MPI_INT,
                MPI_ANY_SOURCE, 0,
                MPI_COMM_WORLD,
                &status);
            int sender = status.MPI_SOURCE;

            sum += tmp;
        }

        printf("Sum of array is : %d\n", sum);
    }
    else {
        MPI_Recv(&n_elements_recieved,
            1, MPI_INT, 0, 0,
            MPI_COMM_WORLD,
            &status);

        MPI_Recv(&a2, n_elements_recieved,
            MPI_INT, 0, 0,
            MPI_COMM_WORLD,
            &status);

        int partial_sum = 0;
        for (int i = 0; i < n_elements_recieved; i++)
            partial_sum += a2[i];

        MPI_Send(&partial_sum, 1, MPI_INT,
            0, 0, MPI_COMM_WORLD);
    }

    MPI_Finalize();

    return 0;
}
```

Output:



```
C:\Users\sachi\source\repos\Project2\x64\Debug>mpiexec ./Project2.exe
Sum of array provided is : 26
```

Experiment 3

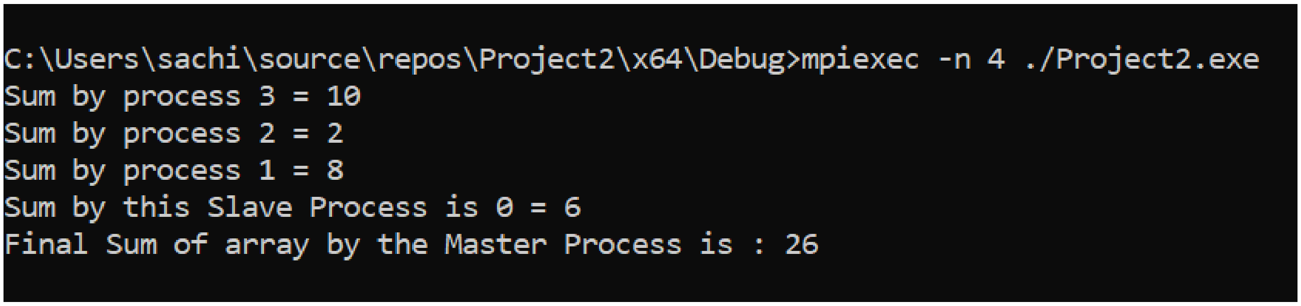
Compute the sum of all the elements of an array using p processors

Code:

```
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>
#define n 8
int a[] = { 1, 5, 2, 6, 2, 0, 1, 9 };
// Temporary array for other processes
int b[1000];
int main(int argc, char* argv[])
{
        int process_id, no_of_process,
                elements_per_process,
                n_elements_recieved;
        MPI_Status status;
        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &process_id);
        MPI_Comm_size(MPI_COMM_WORLD, &no_of_process);
        // For process 0
        if (process_id == 0) {
                int index, i;
                elements_per_process = n / no_of_process;
                if (no_of_process > 1) {
                        for (i = 1; i < no_of_process - 1; i++) {
                                index = i * elements_per_process;
                                MPI_Send(&elements_per_process, 1, MPI_INT, i, 0,
MPI_COMM_WORLD);
                                MPI_Send(&a[index], elements_per_process, MPI_INT, i, 0,
                                        MPI_COMM_WORLD);
                        }
                        // last process adds remaining elements
                                index = i * elements_per_process;
                        int elements_left = n - index;
                        MPI_Send(&elements_left, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
                        MPI_Send(&a[index], elements_left, MPI_INT, i, 0,
MPI_COMM_WORLD);
                }
                // sum by process 0
                int sum = 0;
                for (i = 0; i < elements_per_process; i++)
                        sum += a[i];
                printf("Sum by this Slave Process is %d = %d\n", process_id, sum);
                // partial sums from other processes
                int tmp;
                for (i = 1; i < no_of_process; i++) {
                        MPI_Recv(&tmp, 1, MPI_INT, MPI_ANY_SOURCE, 0, MPI_COMM_WORLD,
&status);
                        int sender = status.MPI_SOURCE;
```

```
            sum += tmp;
        }
        // prints the final sum of array
        printf("Final Sum of array by the Master Process is : %d\n", sum);
    }
    // Other processes
    else {
        MPI_Recv(&n_elements_recieved, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
&status);
        MPI_Recv(&b, n_elements_recieved, MPI_INT, 0, 0, MPI_COMM_WORLD,
&status);
        int partial_sum = 0;
        for (int i = 0; i < n_elements_recieved; i++)
            partial_sum += b[i];
        printf("Sum by process %d = %d\n", process_id, partial_sum);
        MPI_Send(&partial_sum, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
    }
    MPI_Finalize();
    return 0;
}
```

Output :



```
C:\Users\sachi\source\repos\Project2\x64\Debug>mpiexec -n 4 ./Project2.exe
Sum by process 3 = 10
Sum by process 2 = 2
Sum by process 1 = 8
Sum by this Slave Process is 0 = 6
Final Sum of array by the Master Process is : 26
```

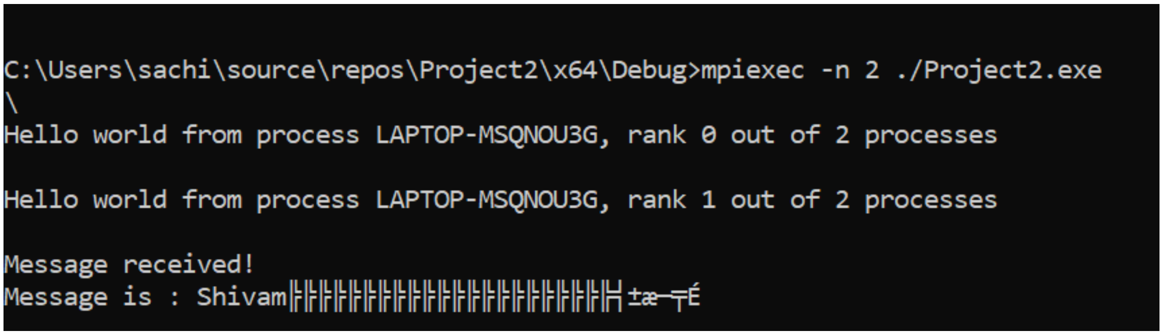
Experiment 4

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
        // 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[] = "Shivam";
                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 Successfully Received\n");
                printf("Message Recieved : %s\n", message);
        }
        MPI_Finalize();
        return 0;
}
```

Output:



```
C:\Users\sachi\source\repos\Project2\x64\Debug>mpiexec -n 2 ./Project2.exe
\
Hello world from process LAPTOP-MSQNOU3G, rank 0 out of 2 processes

Hello world from process LAPTOP-MSQNOU3G, rank 1 out of 2 processes

Message received!
Message is : Shivam╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠┤±æ─╤É
```

Experiment 5

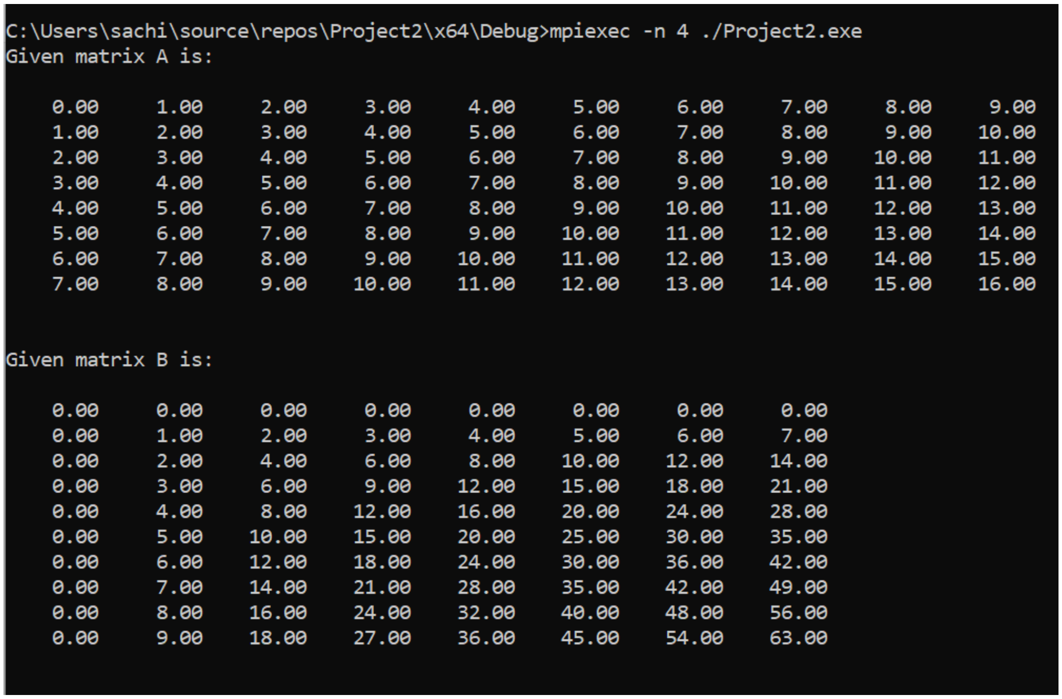
Design a parallel program for summing up an array, matrix multiplication and show logging and tracing MPI activity

Code:

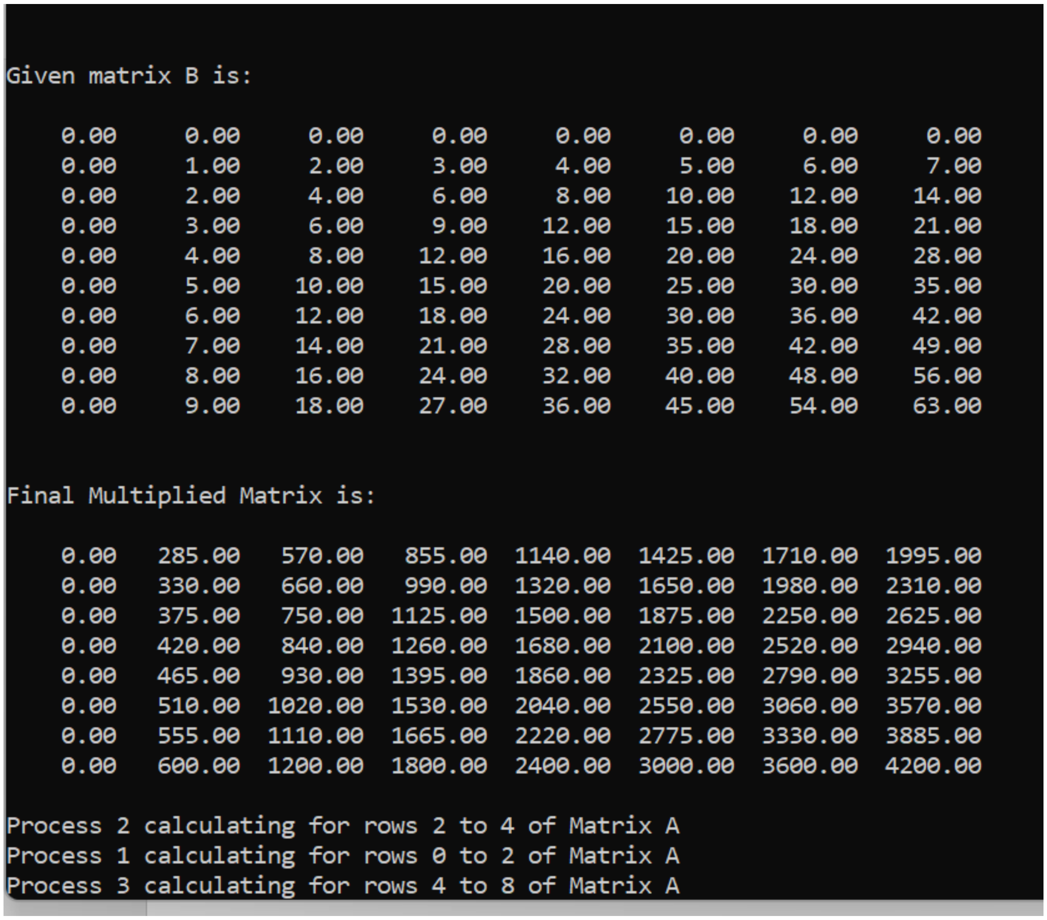
```
#include<stdio.h>
#include<iostream>
#include "mpi.h"
#define NUM_ROWS_A 8
#define NUM_COLUMNS_A 10
#define NUM_ROWS_B 10
#define NUM_COLUMNS_B 8
#define MASTER_TO_SLAVE_TAG 1 //tag for messages sent from master to slaves
#define SLAVE_TO_MASTER_TAG 4 //tag for messages sent from slaves to master
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++) {
		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");
}
```

Output:



```
C:\Users\sachi\source\repos\Project2\x64\Debug>mpiexec -n 4 ./Project2.exe
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     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
```



```
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

Process 2 calculating for rows 2 to 4 of Matrix A
Process 1 calculating for rows 0 to 2 of Matrix A
Process 3 calculating for rows 4 to 8 of Matrix A
```

Experiment 6

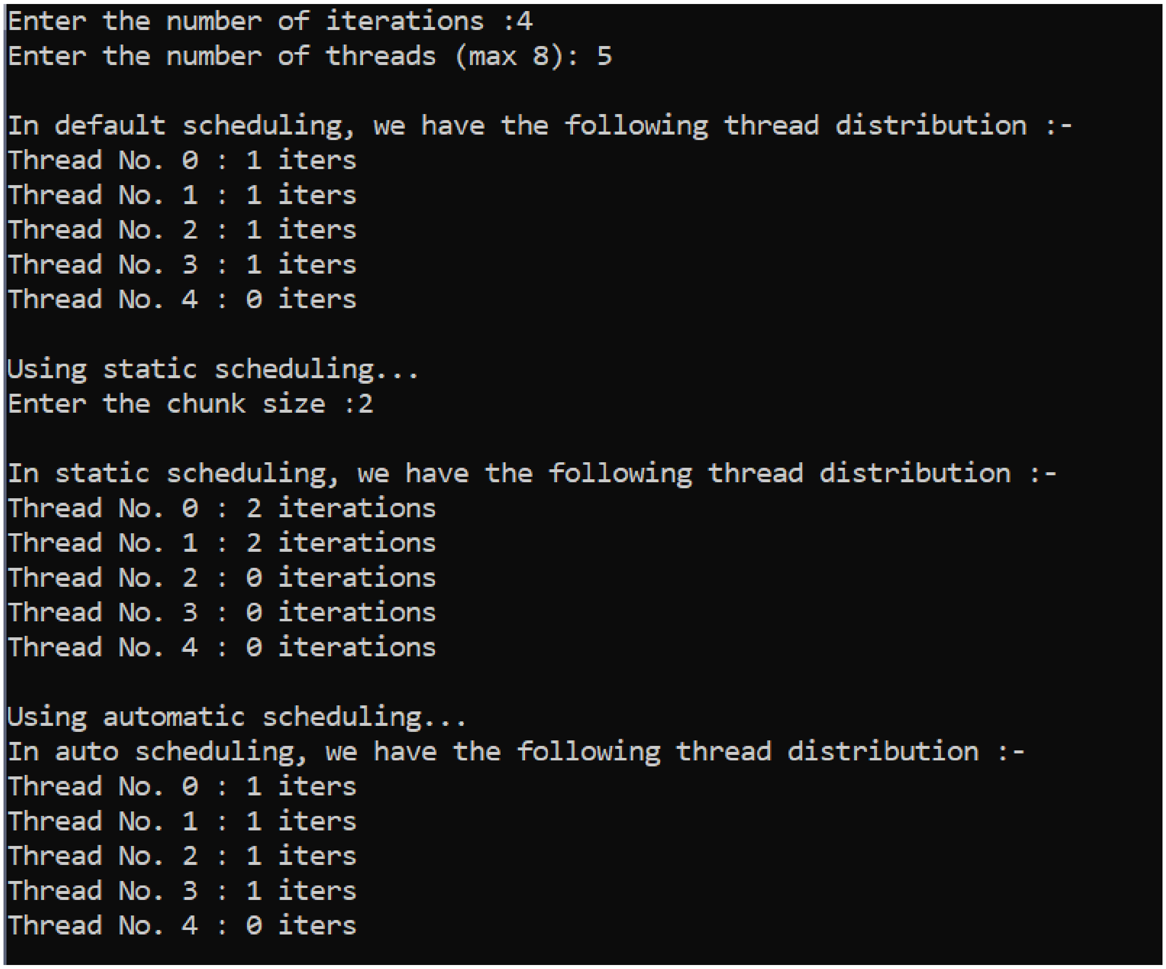
Write a C program with openMP to implement loop work sharing

Code:

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

using namespace std;
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_s("%d", &n);
        printf("Enter the number of threads (max 8): ");
        scanf_s("%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++)
        {
                // 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 No. %d : %d iters\n", i, freq[i]);
        }
        // using static scheduling
        int CHUNK;
        printf("\nUsing static scheduling...\n");
        printf("Enter the chunk size :");
        scanf_s("%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 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++)
    {
        // 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 No. %d : %d iters\n", i, freq[i]);
    }
    return 0;
}
```

Output:



```
Enter the number of iterations :4
Enter the number of threads (max 8): 5

In default scheduling, we have the following thread distribution :-
Thread No. 0 : 1 iters
Thread No. 1 : 1 iters
Thread No. 2 : 1 iters
Thread No. 3 : 1 iters
Thread No. 4 : 0 iters

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
Thread No. 4 : 0 iterations

Using automatic scheduling...
In auto scheduling, we have the following thread distribution :-
Thread No. 0 : 1 iters
Thread No. 1 : 1 iters
Thread No. 2 : 1 iters
Thread No. 3 : 1 iters
Thread No. 4 : 0 iters
```

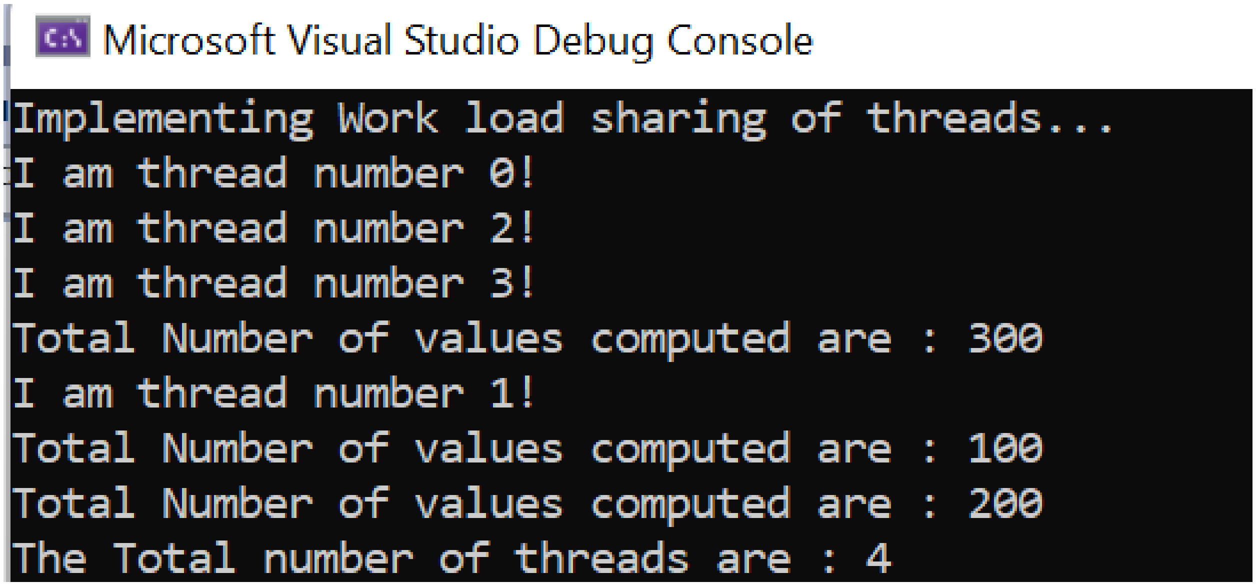
Experiment 7

Write a C program with openMP to implement loop work sharing

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("Implementing 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("Total Number of values computed are : %d\n",
value_count);
        }
#pragma omp barrier
        if (thread_ID == 0)
        {
            printf("The Total number of threads are : %d",
omp_get_num_threads());
        }
    }
    return 0;
}
```

Output:



```
Implementing Work load sharing of threads...
I am thread number 0!
I am thread number 2!
I am thread number 3!
Total Number of values computed are : 300
I am thread number 1!
Total Number of values computed are : 100
Total Number of values computed are : 200
The Total number of threads are : 4
```

Experiment 8

Write a program to illustrate process synchronization and collective data movements

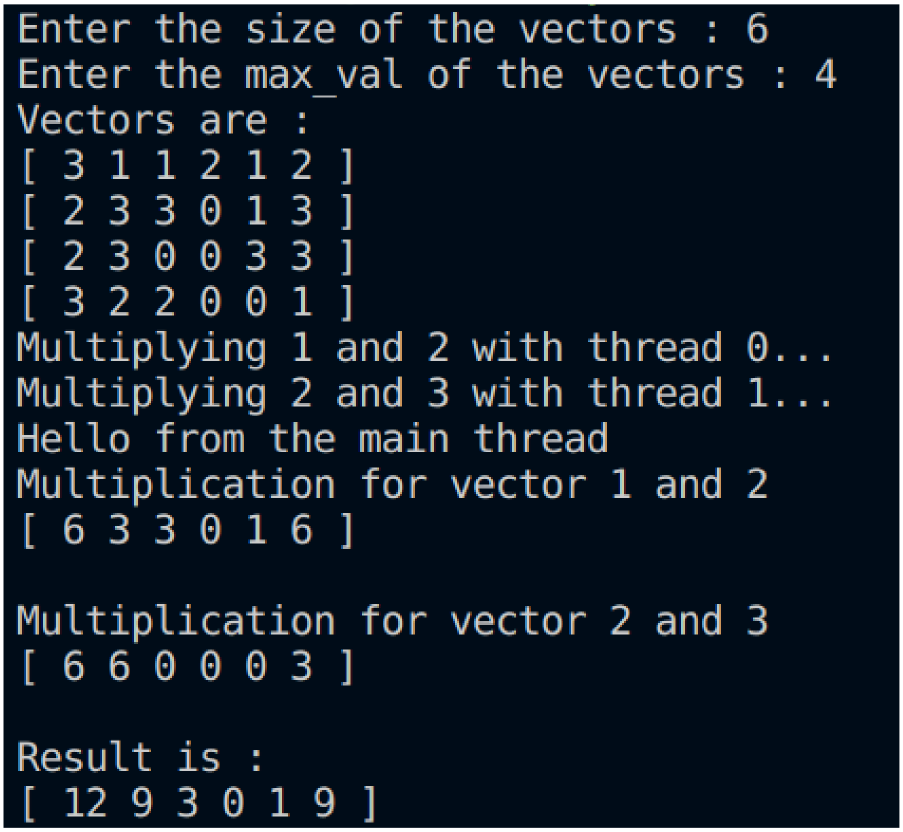
Code:

```c
#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++)
                22
                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++)
                     23
              {
                     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++)
              24
              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:



```
Enter the size of the vectors : 6
Enter the max_val of the vectors : 4
Vectors are :
[ 3 1 1 2 1 2 ]
[ 2 3 3 0 1 3 ]
[ 2 3 0 0 3 3 ]
[ 3 2 2 0 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
[ 6 3 3 0 1 6 ]

Multiplication for vector 2 and 3
[ 6 6 0 0 0 3 ]

Result is :
[ 12 9 3 0 1 9 ]
```