High Performance Computing

Practical Lab File

Submitted by :

NAME : Harshit Gupta

ROLL NO : 2019UCO1580

TABLE OF CONTENTS

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

1. Run a basic hello world program using pThreads

Code

```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,
```

```c
    // 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;
}
```

SCREENSHOTS

```
(base) harshit@harshit-Aspire-A315-55G:~/college/Sem-6/HPC$
gcc openMP/p_threads.c -o ./openMP/thread-basic -lpthread
(base) harshit@harshit-Aspire-A315-55G:~/college/Sem-6/HPC$
./openMP/thread-basic 4
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
```

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

Code

```c
#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();
}
```

SCREENSHOTS

```
(base) harshit@harshit-Aspire-A315-55G:~/college/Sem-6/HPC/MPI$ mpicc add-array-two-procs.c -o add
(base) harshit@harshit-Aspire-A315-55G:~/college/Sem-6/HPC/MPI$ mpirun -np 2 ./add
Enter number of elements : 15
Array is -
 [ 84 87 78 16 94 36 87 93 50 22 63 28 91 60 64 ]
Time taken by process 0 : 0.000001
Total time for recieving : 2.308675
Process 1 sending sum 471 back to main...
Time taken by process for addition 1 : 0.000003
Total sum of array is 953
```

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

Code

```c
#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();
}
```

SCREENSHOTS

```
(base) harshit@harshit-Aspire-A315-55G:~/college/Sem-6/HPC/MPI$ mpicc add-array-p-procs.c -o addp
(base) harshit@harshit-Aspire-A315-55G:~/college/Sem-6/HPC/MPI$ mpirun -np 4 ./addp
Enter number of elements : 150
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 10 3 3 9 10 8 4 7 2 3 10 4 2 10
5 8 9 5 6 1 4 7 2 1 7 4 3 1 7 2 6 6 5 8 7 6 7 10 4 8 5 6 3 6 5 8 5 5 4 1 8 9 7 9 9 5 4 2 5 10 3 1
7 9 10 3 7 7 5 10 6 1 5 9 8 2 8 3 8 3 3 7 2 1 7 2 6 10 5 10 1 10 2 8 8 2 2 6 10 8 8 7 8 4 7 6 7 4
10 5 9 2 3 10 4 10 1 9 9 6 ]

Process 1 sending sum 197 back to main...

Process 2 sending sum 214 back to main...
Total sum of array is 856

Process 3 sending sum 236 back to main...
```

4. Write a program to illustrate basic MPI communication routines

Code

```c
#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;
}
```

SCREENSHOTS

```
(base) harshit@harshit-Aspire-A315-55G:~/college/Sem-6/HPC/MPI$ mpicc hello-mpi.c -o basic_mpi
(base) harshit@harshit-Aspire-A315-55G:~/college/Sem-6/HPC/MPI$ mpirun -np 2 ./basic_mpi
Hello world from process harshit-Aspire-A315-55G, rank 1 out of 2 processes

Hello world from process harshit-Aspire-A315-55G, rank 0 out of 2 processes

Message received!
Message is : Hello!harshit-Aspire-A315-55G
```

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

Code - Sum of array

```c
#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();
}
```

SCREENSHOTS

```
(base) harshit@harshit-Aspire-A315-55G:~/college/Sem-6/HPC/MPI$ mpicc add-array-two-procs.c -o add
(base) harshit@harshit-Aspire-A315-55G:~/college/Sem-6/HPC/MPI$ mpirun -np 2 ./add
Enter number of elements : 15
Array is -
 [ 84 87 78 16 94 36 87 93 50 22 63 28 91 60 64 ]
Time taken by process 0 : 0.000001
Total time for recieving : 2.308675
Process 1 sending sum 471 back to main...
Time taken by process for addition 1 : 0.000003
Total sum of array is 953
```

Code - Matrix Multiplication

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 *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[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;
}
```

SCREENSHOTS

```
(base) harshit@harshit-Aspire-A315-55G:~/college/Sem-6/HPC/openMP$ ./forp
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 :21

In static scheduling, we have the following thread distribution :-
Thread 0 : 21 iters
Thread 1 : 21 iters
Thread 2 : 21 iters
Thread 3 : 21 iters
Thread 4 : 16 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 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("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;
}
```

SCREENSHOTS

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

A **B**
[ ] **[ ]**
. **.**
[ ] **[ ]**
C **D**
+
[ ]

Result

- There are 2 thread computing the dot product of the vectors simultaneously.
- This computation is parallelized, and then first thread in the program waits for the second one to complete the multiplication
- After the multiplication is done, the main program computes the sum of the two dot products

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++)
```

```
        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;
}
```

SCREENSHOTS

```
(base) harshit@harshit-Aspire-A315-55G:~/Desktop/college/Sem-6/HPC/openMP$ ./thread-vector
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 ]
```