HIGH PERFORMANCE COMPUTING

PRACTICAL LAB FILE

Submitted by:-
NAME: Divyansh Mehrotra
ROLL NO: 2019UCO1503

Q1. 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,
// 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:

```
terminator@terminator-VirtualBox:~/Downloads/hpc1$ gcc hpc1.c -o ./hpc1 -lpthre
ad
terminator@terminator-VirtualBox:~/Downloads/hpc1$ ./hpc1 4
Hello from thread 1 of 4
Hello from thread 3 of 4
Hello from the main thread
Hello from thread 2 of 4
Hello from thread 0 of 4
```

Q2. Run a program to find the sum of all elements of an array using 2 processors.

```
#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:

```
terminator@terminator-VirtualBox:~/Downloads/hpc2$ mpicc hpc2.c -o hpc2
terminator@terminator-VirtualBox:~/Downloads/hpc2$ mpirun -np 2 ./hpc2
6
Array is -
 [ 9384 887 2778 6916 7794 8336 ]
Time taken by process 0 : 0.000001
Total time for recieving : 1.777300
Process 1 sending sum 23046 back to main...
Time taken by process for addition 1 : 1.777325
Total sum of array is 36095
```

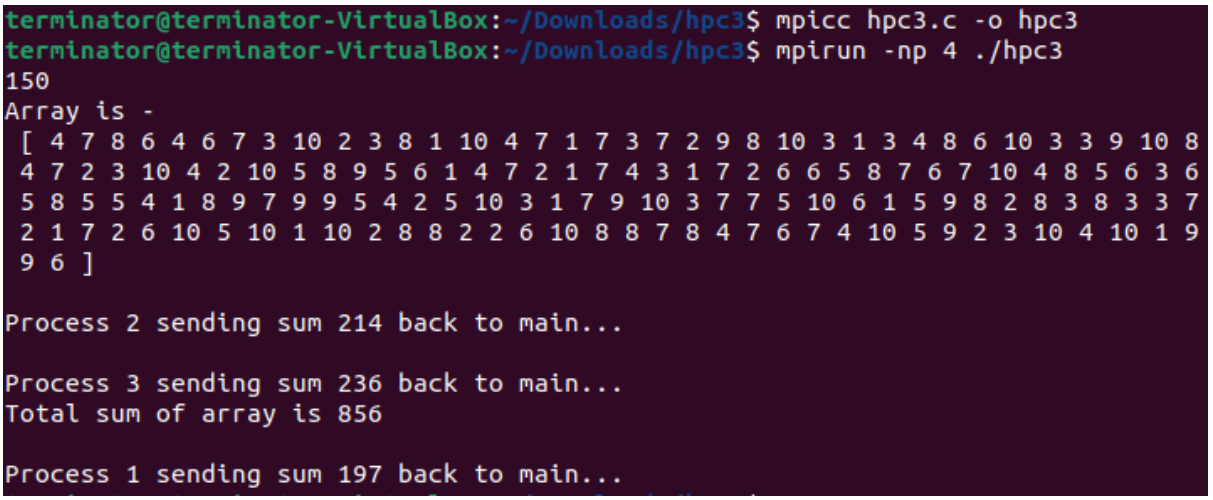
Q3. Compute the sum of all elements of an array using p processors.

```
#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:

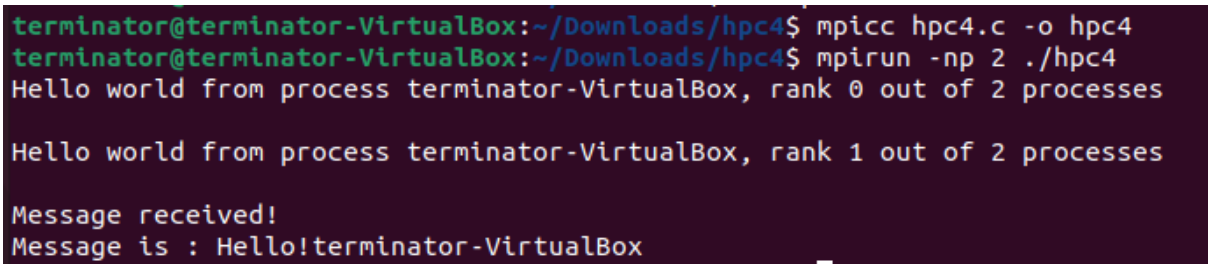


Q4. Write a program to illustrate basic MPI communication routines.

```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;
}
```

OUTPUT:



Q5. Design a parallel program for summing up an array, matrix multiplication and show logging and tracing MPI activity.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <openmpi/mpi.h>
#define MAT_SIZE 5
int main(int argc, char *argv[])
{
  int np, pid;
  MPI_Status status;
  // int A[3][3] = {{2, 4, 1}, {6, 0, 0}, {3, -12, 6}};
  int A[MAT_SIZE][MAT_SIZE] = {{2, 4, 1}, {6, 0, 0}, {3, -12, 6}, {6, 0,
0}, {6, 0, 0}};

  // int B[3][3] = {{5, 5, 8}, {6, 2, 4}, {3, 5, 7}};
  int B[MAT_SIZE][MAT_SIZE] = {{5, 5, 8}, {6, 2, 4}, {3, 5, 7}, {6, 0,
0}, {6, 0, 0}};
  // int C[3][3];
  int C[MAT_SIZE][MAT_SIZE];

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &pid);
  MPI_Comm_size(MPI_COMM_WORLD, &np);

  if (pid == 0)
  {
    int num_rows_per_processor = MAT_SIZE / np;
    for (int i = 1; i < np - 1; i++)
    {
      int index = i * num_rows_per_processor;
      printf("Processor 0: Sending rows %d to %d to processor %d\n",
             index, index + num_rows_per_processor - 1, i);
      MPI_Send(&index, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
      MPI_Send(&num_rows_per_processor, 1, MPI_INT, i, 0,
               MPI_COMM_WORLD);
      MPI_Send(&A[index][0], num_rows_per_processor * MAT_SIZE,
               MPI_INT, i, 0, MPI_COMM_WORLD);
      MPI_Send(&B[0][0], MAT_SIZE * MAT_SIZE, MPI_INT, i, 0,
               MPI_COMM_WORLD);
    }
    int index = (np - 1) * num_rows_per_processor;
    int num_rows_sent = MAT_SIZE - index;
    printf("Processor 0: Sending rows %d to %d to processor %d\n",
index,
           index + num_rows_sent - 1, np - 1);
```

```
  MPI_Send(&index, 1, MPI_INT, np - 1, 0, MPI_COMM_WORLD);
  MPI_Send(&num_rows_sent, 1, MPI_INT, np - 1, 0, MPI_COMM_WORLD);

  MPI_Send(&A[index][0], num_rows_sent * MAT_SIZE, MPI_INT, np - 1, 0,
           MPI_COMM_WORLD);
  MPI_Send(&B[0][0], MAT_SIZE * MAT_SIZE, MPI_INT, np - 1, 0,
           MPI_COMM_WORLD);
  for (int r = 0; r < num_rows_per_processor; r++)
  {
    for (int c = 0; c < MAT_SIZE; c++)
    {
      C[r][c] = 0;
      for (int k = 0; k < MAT_SIZE; k++)
      {
        C[r][c] += A[r][k] * B[k][c];
      }
    }
  }
  for (int i = 1; i < np; i++)
  {
    int index, num_rows;
    MPI_Recv(&index, 1, MPI_INT, i, 2, MPI_COMM_WORLD, &status);
    MPI_Recv(&num_rows, 1, MPI_INT, i, 2, MPI_COMM_WORLD, &status);
    MPI_Recv(&C[index][0], num_rows * MAT_SIZE, MPI_INT, i, 2,
             MPI_COMM_WORLD, &status);
    printf("Processor 0: Received answer from processor %d\n",
           status.MPI_SOURCE);
  }
  // print matrix C here
  for (int i = 0; i < MAT_SIZE; i++)
  {
    for (int j = 0; j < MAT_SIZE; j++)
    {
      printf("%d ", C[i][j]);
    }
    printf("\n");
  }
}
else
{
  int num_rows, index;
  MPI_Recv(&index, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
  MPI_Recv(&num_rows, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
  MPI_Recv(&A, num_rows * MAT_SIZE, MPI_INT, 0, 0, MPI_COMM_WORLD,
           &status);
  MPI_Recv(&B, MAT_SIZE * MAT_SIZE, MPI_INT, 0, 0, MPI_COMM_WORLD,
```

```
            &status);
    printf("Processor %d: Received rows %d to %d from processor 0\n",
           pid, index, index + num_rows - 1);
    for (int r = 0; r < num_rows; r++)
    {
      for (int c = 0; c < MAT_SIZE; c++)
      {
        C[r][c] = 0;
        for (int k = 0; k < MAT_SIZE; k++)
        {

          C[r][c] += A[r][k] * B[k][c];
        }
      }
    }
    printf("Processor %d: sending answer to processor 0\n", pid);
    MPI_Send(&index, 1, MPI_INT, 0, 2, MPI_COMM_WORLD);
    MPI_Send(&num_rows, 1, MPI_INT, 0, 2, MPI_COMM_WORLD);
    MPI_Send(&C, num_rows * MAT_SIZE, MPI_INT, 0, 2, MPI_COMM_WORLD);
  }
  MPI_Finalize();
  return 0;
}
```

OUTPUT:

```
terminator@terminator-VirtualBox:~/Downloads$ mpicc hpc5.c -o hpc5
terminator@terminator-VirtualBox:~/Downloads$ mpirun -np 2 ./hpc5
Processor 0: Sending rows 2 to 4 to processor 1
Processor 0: Received answer from processor 1
37 23 39 0 0
30 30 48 0 0
Trash 18 0 0
30 30 48 0 0
30 30 48 0 0
Processor 1: Received rows 2 to 4 from processor 0
Processor 1: sending answer to processor 0
```

Q6.Write a C program with openMP to implement loop work sharing.

```
#include <omp.h>
#include <stdio.h>
void reset_freq(int *freq, int THREADS)
{
for (int i = 0; i < THREADS; i++)
freq[i] = 0;
}
int main(int *argc, char **argv)
{
int n, THREADS, i;

printf("Enter the number of iterations :");
scanf("%d", &n);

printf("Enter the number of threads (max 8): ");
scanf("%d", &THREADS);

int freq[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:

```
terminator@terminator-VirtualBox:~/Downloads/hpc6$ gcc -fopenmp hpc6.c -o hpc6
terminator@terminator-VirtualBox:~/Downloads/hpc6$ ./hpc6
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 :24

In static scheduling, we have the following thread distribution :-
Thread 0 : 24 iters
Thread 1 : 24 iters
Thread 2 : 24 iters
Thread 3 : 24 iters
Thread 4 : 4 iters
Thread 5 : 0 iters

Using automatic scheduling...
In auto scheduling, we have the following thread distribution :-
Thread 0 : 17 iters
```

```
Thread 2 : 17 iters
Thread 3 : 17 iters
Thread 4 : 16 iters
Thread 5 : 16 iters
```

Q7. Write a C program with openMP to implement sections work sharing.

```
#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:

```
terminator@terminator-VirtualBox:~/Downloads/hpc7$ gcc -fopenmp hpc7.c -o hpc7
terminator@terminator-VirtualBox:~/Downloads/hpc7$ ./hpc7
Work load sharing of threads...
I am thread number 3!
Number of values computed : 300
I am thread number 2!
Number of values computed : 200
I am thread number 0!
I am thread number 1!
Number of values computed : 100
Total number of threads are 4terminator@terminator-VirtualBox:~/Downloads/hpc7$
```

Q8. Write a program to illustrate process synchronization and collective data movements.

```
#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));
```

```c
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:

```
terminator@terminator-VirtualBox:~/Downloads/hpc8$ mpirun -np 1 hpc8
8
9
Enter the size of the vectors : Enter the max_val of the vectors : Vectors are
:
[ 1 0 5 1 6 5 5 5 ]
[ 7 7 7 3 1 4 7 4 ]
[ 6 7 8 6 8 8 1 5 ]
[ 0 1 8 6 8 4 1 0 ]
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
[ 7 0 35 3 6 20 35 20 ]

Multiplication for vector 2 and 3
[ 0 7 64 36 64 32 1 0 ]

Result is :
[ 7 7 99 39 70 52 36 20 ]
```