Indian Institute of Information Technology, Design and Manufacturing, Kancheepuram

High Performance Computing Practice - COM403P

EXPERIMENT 5

Name	Roll Number
N Sree Dhyuti	CED19I027

---

1. Cluster Creation for MPI Programming

OBJECTIVE:

1. Using a Virtual Machine creates a cluster with a Master and Slave Relationship. Implement Vector Addition and Vector Multiplication using MPI on the above-distributed network.

Serial Code:

Vector Addition:

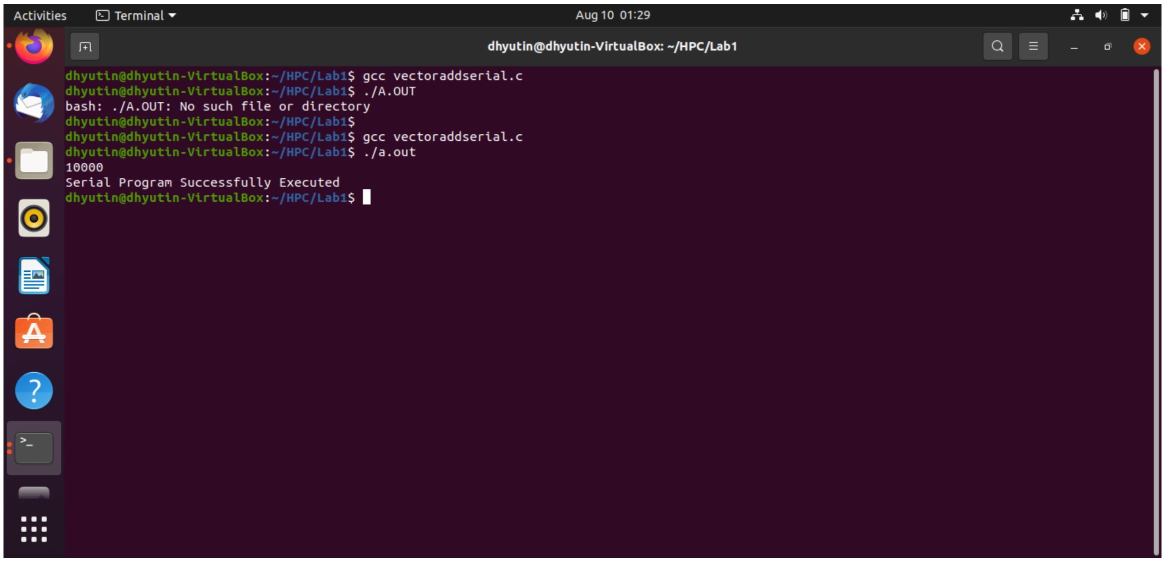
```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include "omp.h"

int main(){

    int n;
    scanf("%d", &n);

    double v1[n], v2[n], ans[n];

    for(int i = 0; i < n; i++){
        v1[i] = (float)rand()/(float)(RAND_MAX/n);
        v2[i] = (float)rand()/(float)(RAND_MAX/n);
```

```
    }

    for(int i = 0; i < n; i++){
        ans[i] = 0;
    }

     for(int i = 0; i < n; i++){
        ans[i] = v1[i] + v2[i];
     }

     printf("Serial Program Successfully Executed\n");

    return 0;
}
```

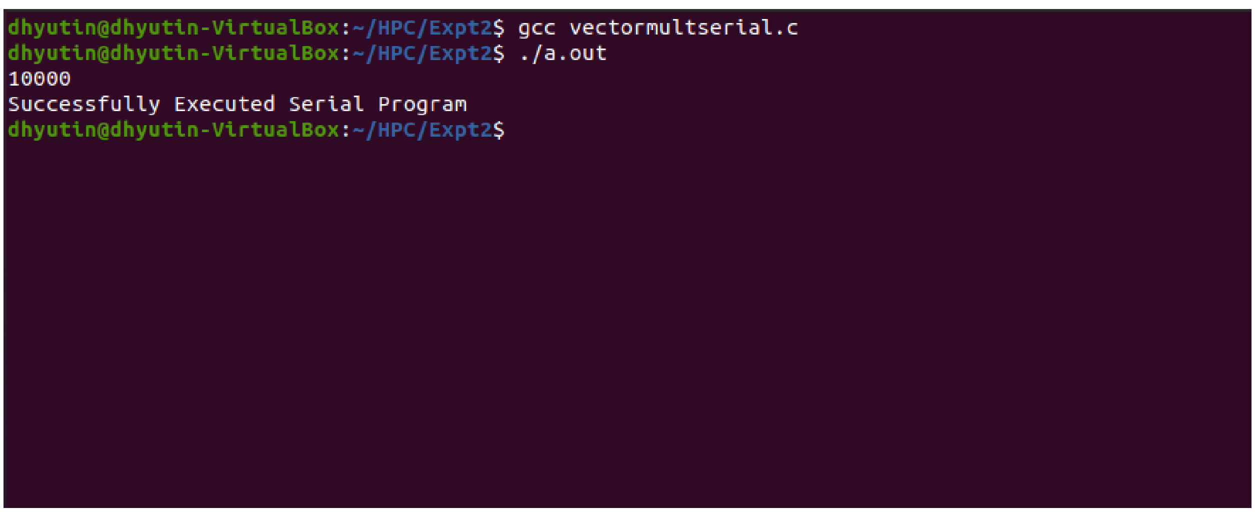
Vector Multiplication:

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include "omp.h"

int main(){

    int n;
    scanf("%d", &n);

    double v1[n], v2[n], v3[n], v4[n], v5[n], ans[n];

    for(int i = 0; i < n; i++){
        v1[i] = (float)rand()/(float)(RAND_MAX/n);
        v2[i] = (float)rand()/(float)(RAND_MAX/n);
        v3[i] = (float)rand()/(float)(RAND_MAX/n);
        v4[i] = (float)rand()/(float)(RAND_MAX/n);
        v5[i] = (float)rand()/(float)(RAND_MAX/n);
    }

    for(int i = 0; i < n; i++){
        ans[i] = 0;
    }

     for(int i = 0; i < n; i++){
        ans[i] = v1[i] * v2[i] * v3[i] * v4[i] * v5[i];
     }
     printf("Successfully Executed Serial Program\n");
    return 0;
}
```

Output:
Vector Addition:



Vector Multiplication:



Parallelized Code:
Vector Addition:

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

// size of array
#define n 10
```

```
int a[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

// Temporary array for slave process
int a2[1000];

int main(int argc, char* argv[])
{
        double t1, t2;
    int pid, np,
        elements_per_process,
        n_elements_recieved;
    // np -> no. of processes
    // pid -> process id

    MPI_Status status;

    // Creation of parallel processes
    MPI_Init(&argc, &argv);

    // find out process ID,
    // and how many processes were started
    MPI_Comm_rank(MPI_COMM_WORLD, &pid);
    MPI_Comm_size(MPI_COMM_WORLD, &np);

        t1 = MPI_Wtime();
    // master process
    if (pid == 0) {
        int index, i;
        elements_per_process = n / np;

        // check if more than 1 processes are run
        if (np > 1) {
            // distributes the portion of array
            // to child processes to calculate
            // their partial sums
            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);
            }

            // 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);
    }

    // master process add its own sub array
    int sum = 0;
    for (i = 0; i < elements_per_process; i++)
        sum += a[i];

    // collects partial sums from other processes
    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;
    }

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

    // stores the received array segment
    // in local array a2
    MPI_Recv(&a2, n_elements_recieved,
            MPI_INT, 0, 0,
            MPI_COMM_WORLD,
            &status);

    // calculates its partial sum
    int partial_sum = 0;
    for (int i = 0; i < n_elements_recieved; i++)
```

```
            partial_sum += a2[i];

        // sends the partial sum to the root process
        MPI_Send(&partial_sum, 1, MPI_INT,
                0, 0, MPI_COMM_WORLD);
    }

    // cleans up all MPI state before exit of process
    t2 = MPI_Wtime();

    MPI_Finalize();
    printf( "Elapsed time is %f\n", t2 - t1 );

    return 0;
}
```

Vector Multiplication:

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

// size of array
#define n 10

int a[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

// Temporary array for slave process
int a2[1000];

int main(int argc, char* argv[])
{
        double t1, t2;

    int pid, np,
        elements_per_process,
        n_elements_recieved;
    // np -> no. of processes
    // pid -> process id

    MPI_Status status;

    // Creation of parallel processes
    MPI_Init(&argc, &argv);
    MPI_Barrier(MPI_COMM_WORLD);

    // find out process ID,
    // and how many processes were started
    MPI_Comm_rank(MPI_COMM_WORLD, &pid);
```

```
MPI_Comm_size(MPI_COMM_WORLD, &np);

// master process
 t1 = MPI_Wtime();
if (pid == 0) {
    int index, i;
    elements_per_process = n / np;

    // check if more than 1 processes are run
    if (np > 1) {
        // distributes the portion of array
        // to child processes to calculate
        // their partial sums
        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);
        }

        // 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);
    }

    // master process add its own sub array
    int product = 1;
    for (i = 0; i < elements_per_process; i++)
        product *= a[i];

    // collects partial sums from other processes
    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;

            product *= tmp;
        }

        // prints the final sum of array
        printf("product of elements of array is : %d\n", product);
    }
    // slave processes
    else {
        MPI_Recv(&n_elements_recieved,
                1, MPI_INT, 0, 0,
                MPI_COMM_WORLD,
                &status);

        // stores the received array segment
        // in local array a2
        MPI_Recv(&a2, n_elements_recieved,
                MPI_INT, 0, 0,
                MPI_COMM_WORLD,
                &status);

        // calculates its partial sum
        int partial_product = 1;
        for (int i = 0; i < n_elements_recieved; i++)
            partial_product *= a2[i];

        // sends the partial sum to the root process
        MPI_Send(&partial_product, 1, MPI_INT,
                0, 0, MPI_COMM_WORLD);
    }

    // cleans up all MPI state before exit of process
    MPI_Barrier(MPI_COMM_WORLD);
    t2 = MPI_Wtime();

    MPI_Finalize();
    printf( "Elapsed time is %f\n", t2 - t1 );
    return 0;
}
```

Output:

Vector Addition:

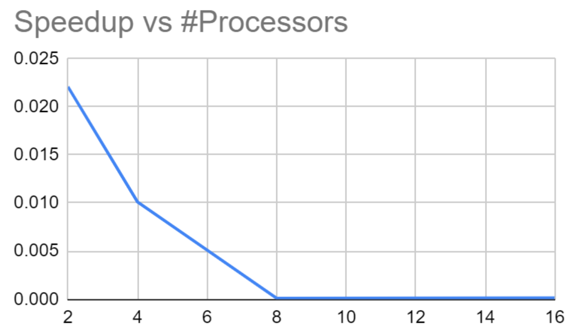
```
ubuntu@c01:~/mirror$ mpicxx ced19i027_vector_addition.cpp  && mpirun -np 1 ./a.out
Sum of array is : 55
Elapsed time is 0.000150
ubuntu@c01:~/mirror$ mpicxx ced19i027_vector_addition.cpp  && mpirun -np 2 ./a.out
Sum of array is : 55
Elapsed time is 0.000016
Elapsed time is 0.006798
ubuntu@c01:~/mirror$ mpicxx ced19i027_vector_addition.cpp  && mpirun -np 4 ./a.out
Sum of array is : 55
Elapsed time is 0.000020
Elapsed time is 0.014552
Elapsed time is 0.014555
Elapsed time is 0.014834
ubuntu@c01:~/mirror$ mpicxx ced19i027_vector_addition.cpp  && mpirun -np 8 ./a.out
Sum of array is : 55
Elapsed time is 0.000022
Elapsed time is 0.206625
Elapsed time is 0.100704
Elapsed time is 0.104521
Elapsed time is 0.207010
Elapsed time is 0.000019
Elapsed time is 0.214656
Elapsed time is 1.382564
ubuntu@c01:~/mirror$ mpicxx ced19i027_vector_addition.cpp  && mpirun -np 16 ./a.out
Sum of array is : 55
Elapsed time is 0.226768
Elapsed time is 0.226766
Elapsed time is 0.000018
Elapsed time is 0.000024
Elapsed time is 0.230745
Elapsed time is 0.000037
Elapsed time is 0.224363
Elapsed time is 0.000024
Elapsed time is 0.228268
Elapsed time is 0.000023
Elapsed time is 0.000018
Elapsed time is 0.000017
Elapsed time is 0.000027
Elapsed time is 0.000026
Elapsed time is 0.222771
Elapsed time is 1.220070
ubuntu@c01:~/mirror$
```

Vector Multiplication:

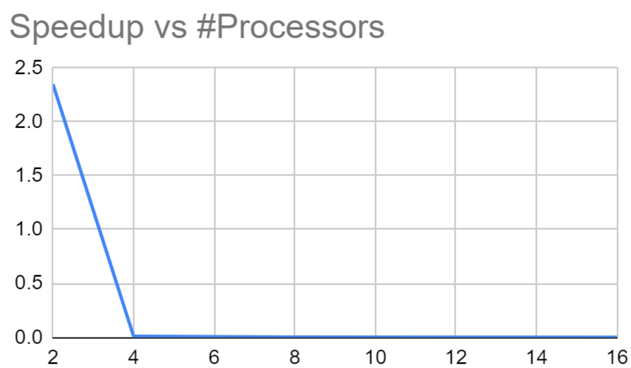
```
ubuntu@c01:~/mirror$ mpicxx ced19i027_vector_multiplication.cpp  && mpirun -np 1 ./a.out
product of elements of array is : 3628800
Elapsed time is 0.000333
ubuntu@c01:~/mirror$ mpicxx ced19i027_vector_multiplication.cpp  && mpirun -np 2 ./a.out
product of elements of array is : 3628800
Elapsed time is 0.008583
Elapsed time is 0.008011
ubuntu@c01:~/mirror$ mpicxx ced19i027_vector_multiplication.cpp  && mpirun -np 4 ./a.out
product of elements of array is : 3628800
Elapsed time is 0.031993
Elapsed time is 0.032164
Elapsed time is 0.032006
Elapsed time is 0.031996
ubuntu@c01:~/mirror$ mpicxx ced19i027_vector_multiplication.cpp  && mpirun -np 8 ./a.out
product of elements of array is : 3628800
Elapsed time is 0.099993
Elapsed time is 0.099997
Elapsed time is 0.092010
Elapsed time is 0.099992
Elapsed time is 0.096027
Elapsed time is 0.064087
Elapsed time is 0.096016
Elapsed time is 0.119988
ubuntu@c01:~/mirror$ mpicxx ced19i027_vector_multiplication.cpp  && mpirun -np 16 ./a.out
product of elements of array is : 3628800
Elapsed time is 0.143998
Elapsed time is 0.187991
Elapsed time is 0.131995
Elapsed time is 0.128191
Elapsed time is 0.191992
Elapsed time is 0.191997
Elapsed time is 0.128003
Elapsed time is 0.187989
Elapsed time is 0.183991
Elapsed time is 0.187998
Elapsed time is 0.191996
Elapsed time is 0.207998
Elapsed time is 0.128092
Elapsed time is 0.128001
Elapsed time is 0.123995
Elapsed time is 0.191995
ubuntu@c01:~/mirror$
```

Speedup V/S Number of Processors:

Vector Addition:

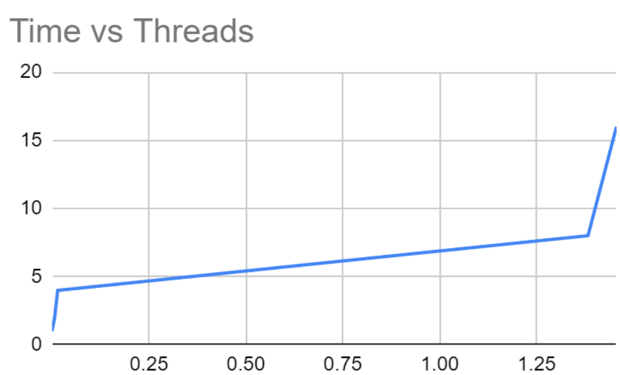


Vector Multiplication:

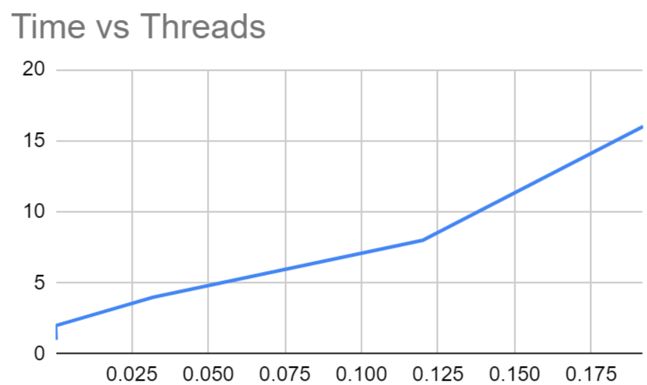


Execution Time V/S Number of Threads:

Vector Addition:



Vector Multiplication:



Inference:

Parallelization Factor (f):

Vector Addition:

#Threads	Execution Time	Speed UP	Efficiency (in %)	f
1	0.00015	1	100	n/a
2	0.006798	0.02206531333	1.103265666	-88.64
4	0.014834	0.01011190508	0.2527976271	-130.5244444
8	1.382564	0.0001084940733	0.001356175917	-10532.6781
16	1.4563	0.0001030007553	0.0006437547209	-10354.84444

Vector Multiplication:

#Threads	Execution Time	Speed UP	Efficiency (in %)	f
1	0.000333	1	100	n/a
2	0.000142	2.345070423	117.2535211	1.147147147
4	0.031996	0.01040755094	0.2601887736	-126.7787788
8	0.119988	0.002775277528	0.0346909691	-410.6563707
16	0.191995	0.001734420167	0.01084012604	-613.9323323

Inference:

In both vector addition and multiplication, it can be observed that the parallelizing factor f is reducing as the number of threads increases. This indicates that parallelizing is only increasing our cost in this case. Therefore, using only one thread for vector addition and multiplication would be optimal.

THE END
