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];**

**}**

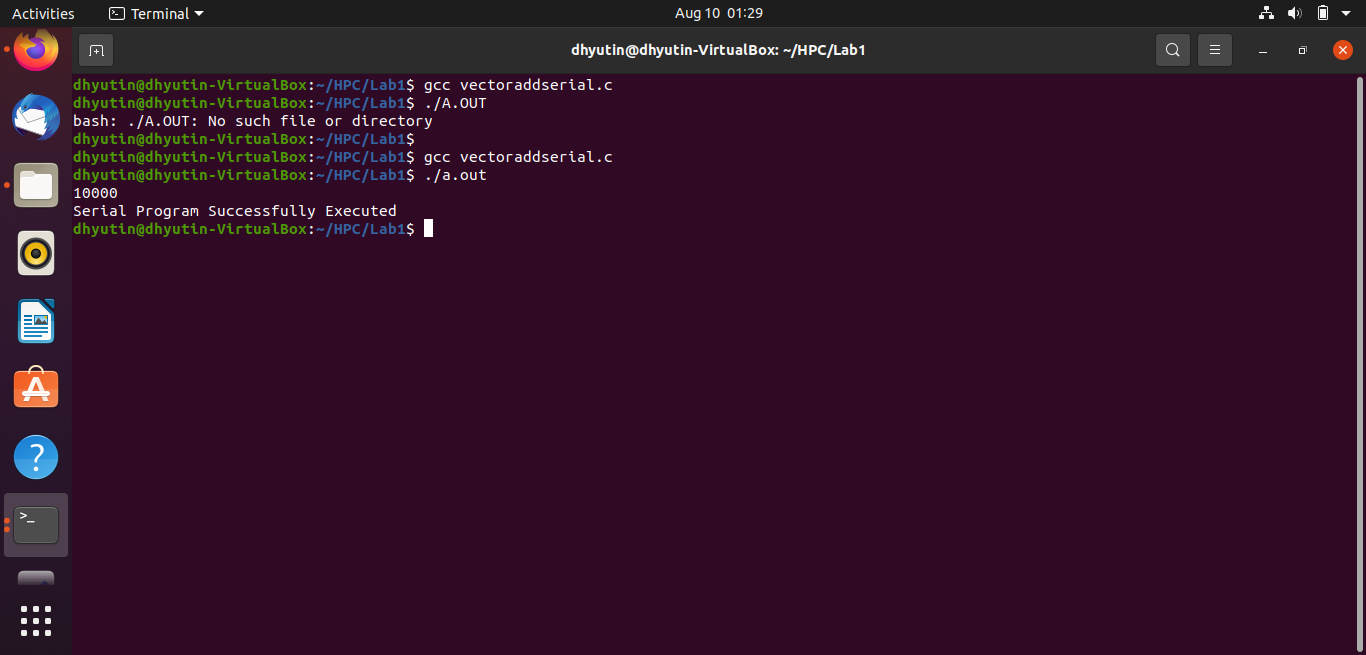
**printf("Successfully Executed Serial Program\n");**

**return 0;**

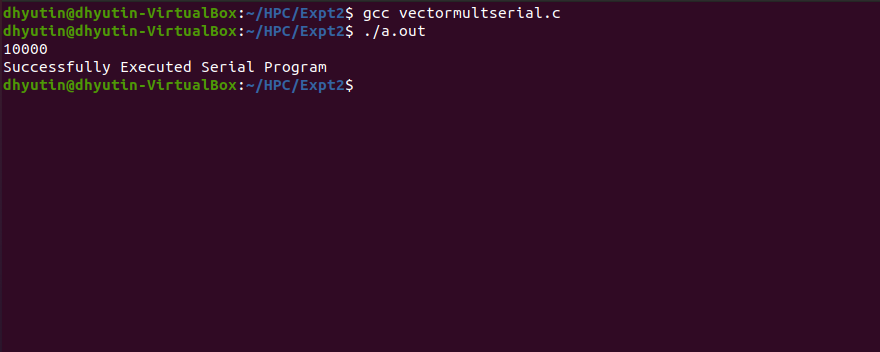
**}**

**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();**

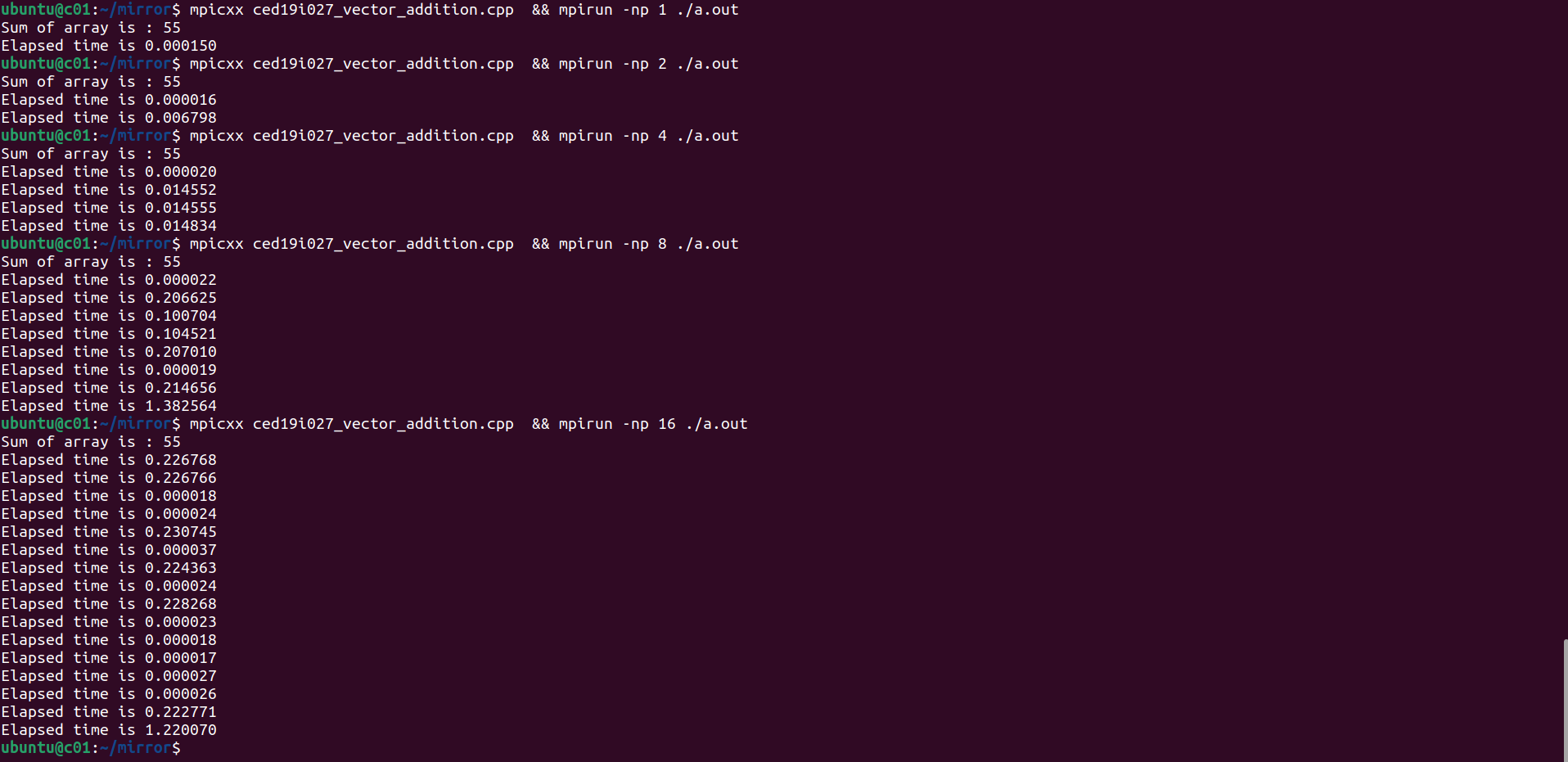
**printf( "Elapsed time is %f\n", t2 - t1 );**

**return 0;**

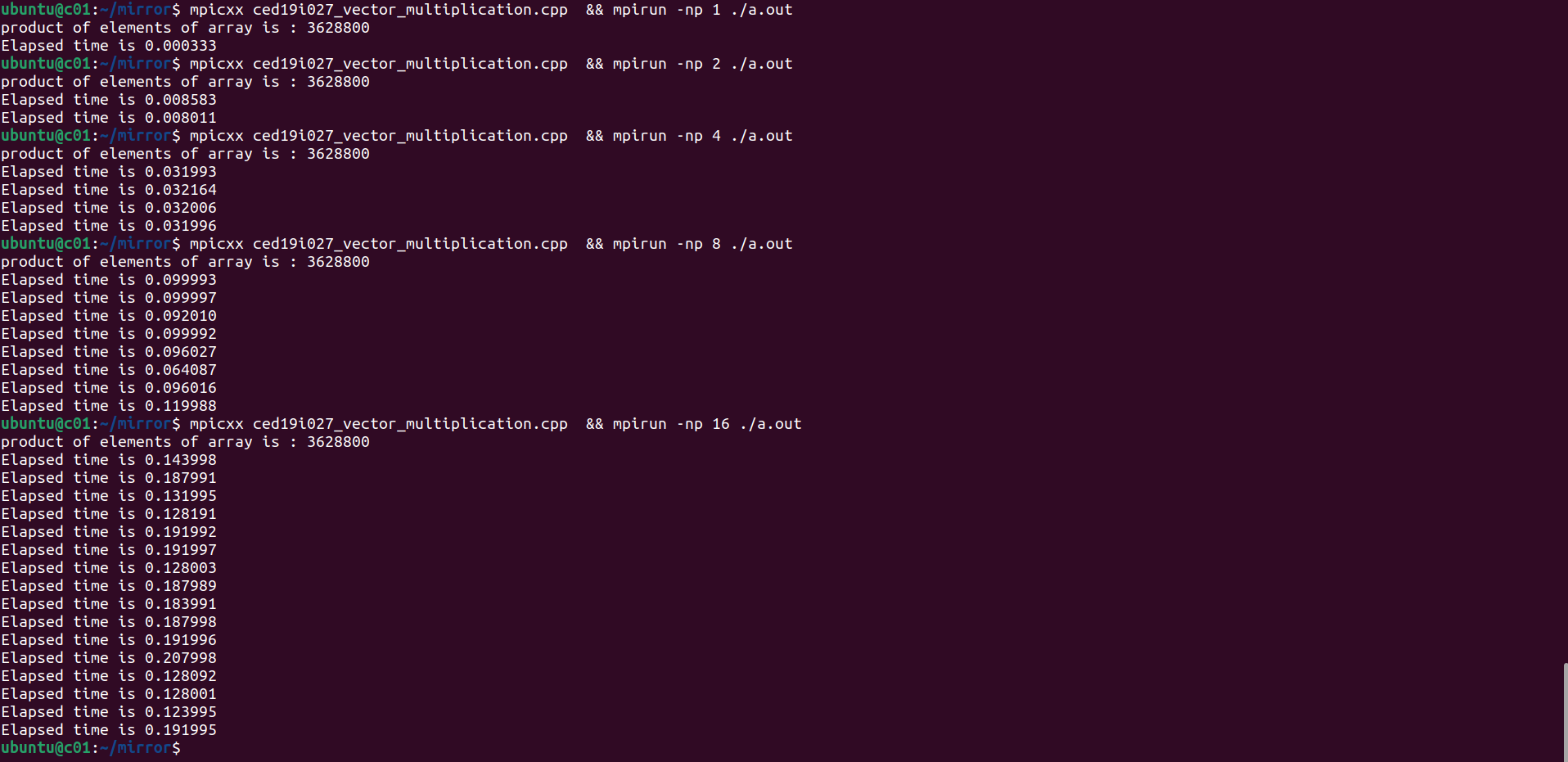
**}**

**Output:**

**Vector Addition:**

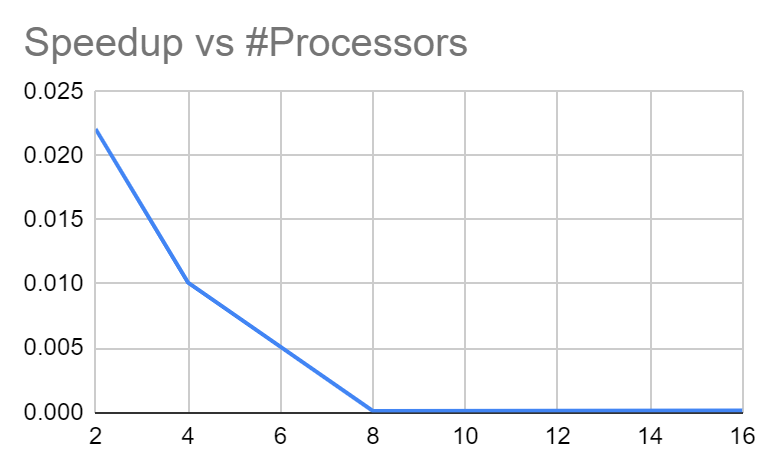


**Vector Multiplication:**

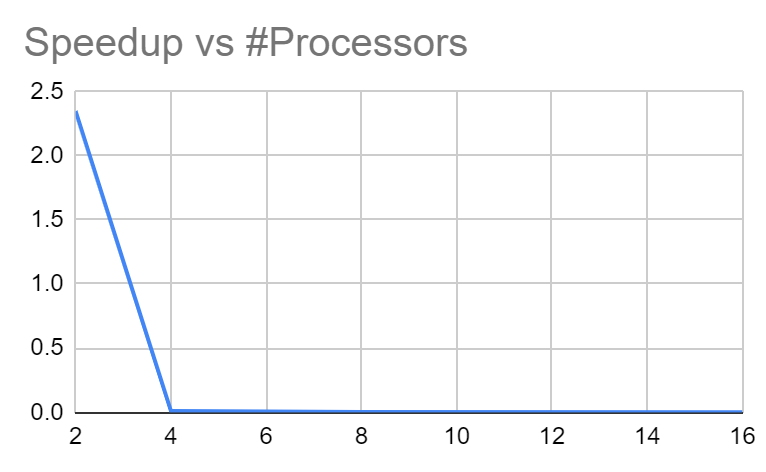


**Speedup V/S Number of Processors:**

**Vector Addition:**

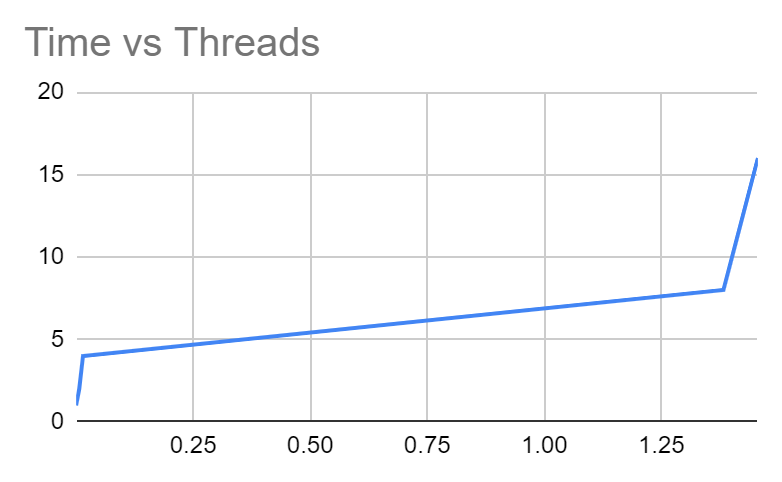
****

**Vector Multiplication:**

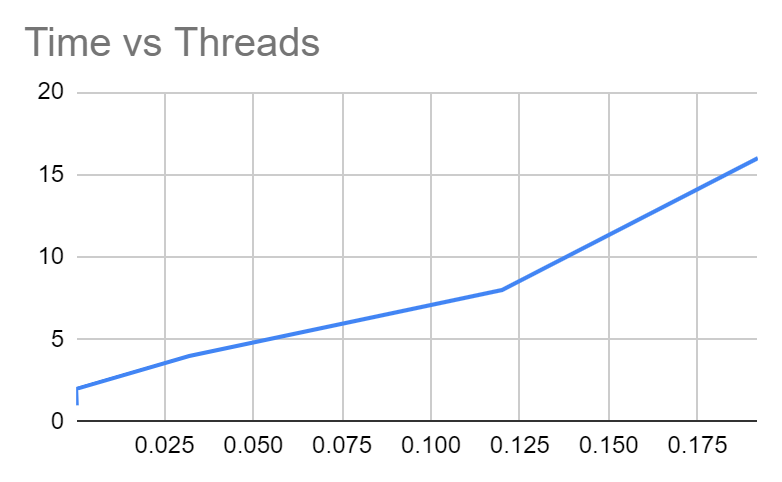
****

**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**