



# Indian Institute of Information Technology, Design and Manufacturing, Kancheeppuram

## High Performance Computing Practice - COM403P

### LAB REPORT 1

Name	Roll Number
N Sree Dhyuti	CED19I027

#### 1. Vector Addition

##### OBJECTIVE:

1. Perform vector addition on 'n' double precision floating point numbers

##### Serial Code:

```
#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];
}

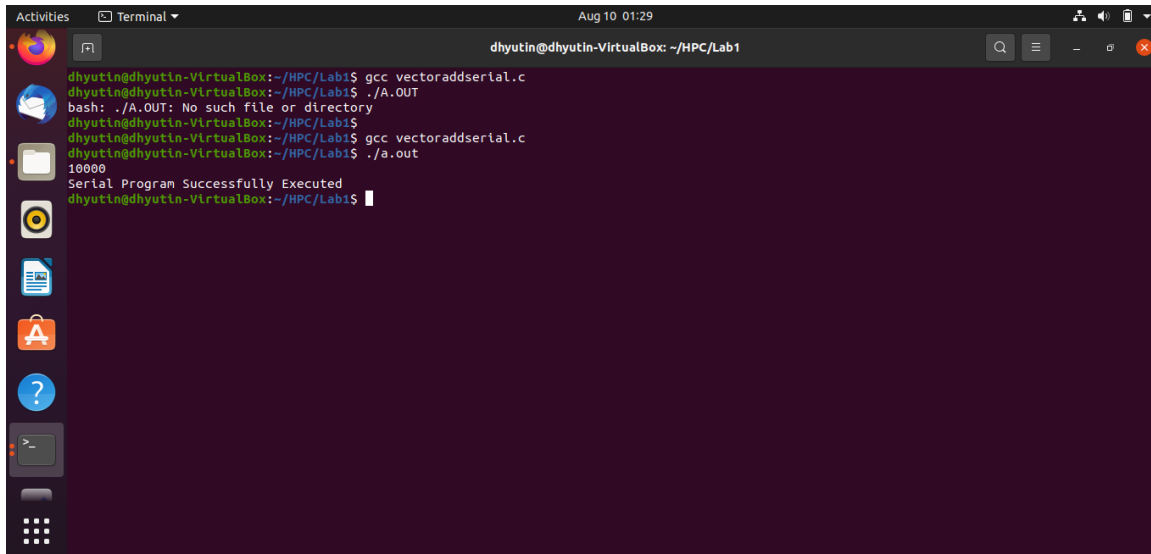
printf("Serial Program Successfully Executed\n");

return 0;
}

```

---

## Output:



```

Activities  Terminal  Aug 10 01:29
dhyutin@dhyutin-VirtualBox: ~/HPC/Lab1
dhyutin@dhyutin-VirtualBox:~/HPC/Lab1$ gcc vectoraddserial.c
dhyutin@dhyutin-VirtualBox:~/HPC/Lab1$ ./A.OUT
bash: ./A.OUT: No such file or directory
dhyutin@dhyutin-VirtualBox:~/HPC/Lab1$ gcc vectoraddserial.c
dhyutin@dhyutin-VirtualBox:~/HPC/Lab1$ ./a.out
10000
Serial Program Successfully Executed
dhyutin@dhyutin-VirtualBox:~/HPC/Lab1$

```

---

## Parallelized Code:

```

#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;
    }
    double wallclock_initial = omp_get_wtime();
    #pragma omp parallel
    {int id = omp_get_thread_num();
    #pragma omp for
    for(int i = 0; i < n; i++){
        ans[i] = v1[i] + v2[i];
    }

    }
    double wallclock_final = omp_get_wtime();

    printf("Time : %lf\n", wallclock_final - wallclock_initial);

    return 0;
}
```

## Output:

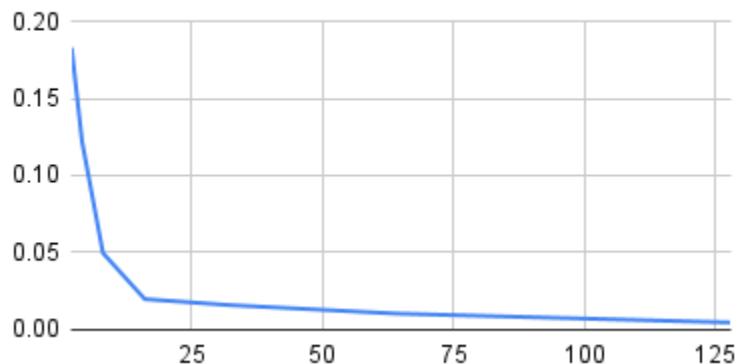
```

Activities Terminal Aug 10 01:25 dhyutin@dhyutin-VirtualBox: ~/HPC/Lab1
dhyutin@dhyutin-VirtualBox:~/HPC/Lab1$ gcc -fopenmp vectoraddparallel.c && export OMP_NUM_THREADS=1 && ./a.out
10000
Time : 0.000028
dhyutin@dhyutin-VirtualBox:~/HPC/Lab1$ gcc -fopenmp vectoraddparallel.c && export OMP_NUM_THREADS=2 && ./a.out
10000
Time : 0.000191
dhyutin@dhyutin-VirtualBox:~/HPC/Lab1$ gcc -fopenmp vectoraddparallel.c && export OMP_NUM_THREADS=4 && ./a.out
10000
Time : 0.004492
dhyutin@dhyutin-VirtualBox:~/HPC/Lab1$ gcc -fopenmp vectoraddparallel.c && export OMP_NUM_THREADS=8 && ./a.out
10000
Time : 0.000686
dhyutin@dhyutin-VirtualBox:~/HPC/Lab1$ gcc -fopenmp vectoraddparallel.c && export OMP_NUM_THREADS=16 && ./a.out
10000
Time : 0.001008
dhyutin@dhyutin-VirtualBox:~/HPC/Lab1$ gcc -fopenmp vectoraddparallel.c && export OMP_NUM_THREADS=32 && ./a.out
10000
Time : 0.001556
dhyutin@dhyutin-VirtualBox:~/HPC/Lab1$ gcc -fopenmp vectoraddparallel.c && export OMP_NUM_THREADS=64 && ./a.out
10000
Time : 0.004354
dhyutin@dhyutin-VirtualBox:~/HPC/Lab1$ gcc -fopenmp vectoraddparallel.c && export OMP_NUM_THREADS=128 && ./a.out
10000
Time : 0.006186
dhyutin@dhyutin-VirtualBox:~/HPC/Lab1$

```

## Speedup V/S Number of Processors:

Speedup vs #Processors



### Inference:

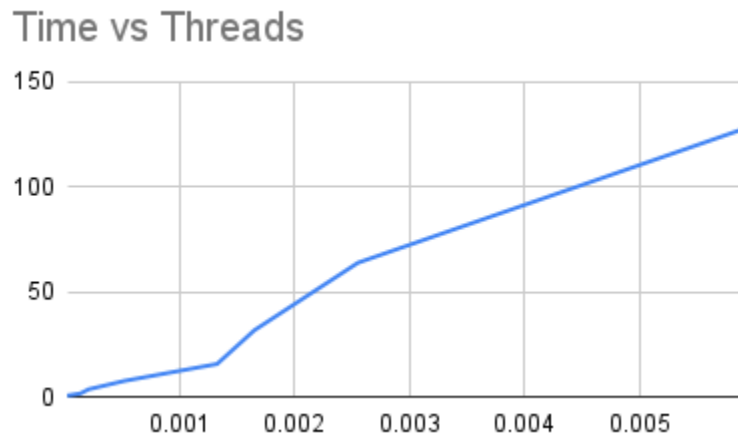
It can be observed from the above graph that the speedup value is decreasing as the number of processors involved are increasing. This is contrary to the general pattern. One reason for this is that the selected 'c program' is not computationally expensive. The code involved in this is just a simple 'for-loop' which fills a memory location with specific values.

One processor can do it at a certain rate, but when you use N processors simultaneously, the total memory bandwidth remains the same on Shared-Memory Multicore systems (i.e most PCs, laptops) and it increases on Distributed-Memory Multicore systems (high-end servers).

All their advantages can be explored in a better way in computationally expensive programs.

---

### Execution Time V/S Number of Threads:



#### Inference:

It can be observed from the above graph that as the number of threads increases, the total execution time also increases. This is contrary to the general pattern. One reason for this is that the selected 'c program' is not computationally expensive.

The code involved in this is just a simple 'for-loop' which fills a memory location with specific values.

A clear advantage of using multiple threads can be observed when a computationally expensive program is parallelized.

For simple vector addition, one thread is sufficient to get optimum time.

---

### Parallelization Factor (f):

#Threads	Execution Time	Speed UP	Efficiency (in %)	f
1	0.000026	1	100	n/a
2	0.000142	0.1830985915	9.154929577	-8.923076923
4	0.000213	0.1220657277	3.051643192	-9.58974359
8	0.000525	0.04952380952	0.619047619	-21.93406593
16	0.001328	0.01957831325	0.1223644578	-53.41538462

32	0.001652	0.01573849879	0.04918280872	-64.55583127
64	0.002551	0.01019208154	0.0159251274	-98.65689866
128	0.005927	0.00438670491	0.00342711321 1	-228.7486372

**Inference:**

It can be noticed that all the parallelization factor 'f' is in negative values. This is an indication that this code has taken the least amount of time while using a single thread. Anything more than that is not worth parallelizing and this f is negative.

---

## THE END

---