Indian Institute of Information Technology, Design and Manufacturing, Kancheepuram

High Performance Computing Practice - COM403P

EXPERIMENT 2

Name	Roll Number
N Sree Dhyuti	CED19I027

Vector Multiplication

OBJECTIVE:

1. Perform vector multiplication on ‘n’ double precision floating point numbers

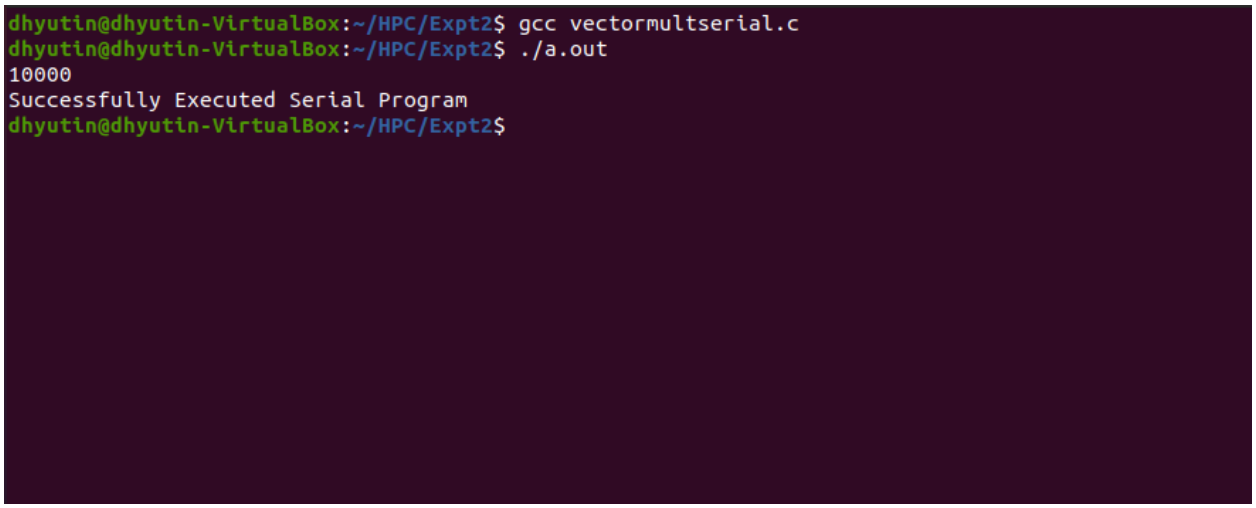
---

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], 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:



Parallelized Code:

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

int main(){

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

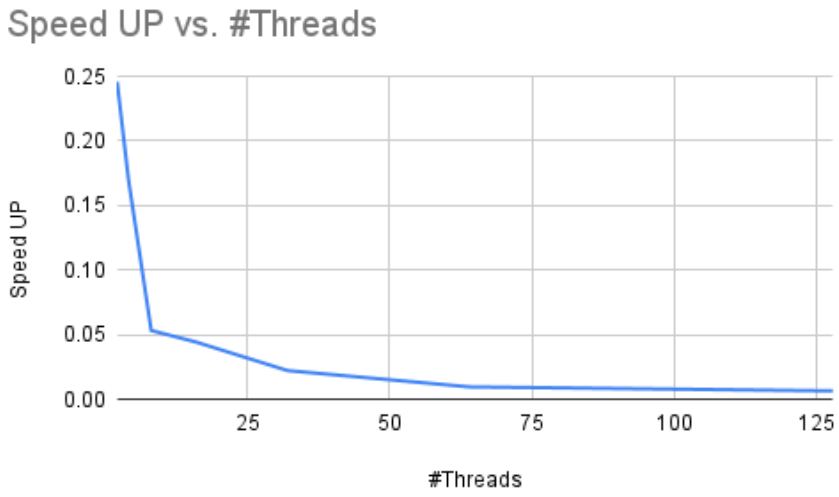
```
double v1[n], v2[n], v3[n], v4[n], v5[n], ans[n];

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;
}
double wallclock_initial = omp_get_wtime();
#pragma omp parallel
{int id = omp_get_thread_num();
//printf("Thread No - %d\n", id);
#pragma omp for
for(int i = 0; i < n; i++){
ans[i] = v1[i] * v2[i] * v3[i] * v4[i] * v5[i];
//printf("%lf ", ans[i]);
}
//printf("\n");
}
double wallclock_final = omp_get_wtime();

/*
double wc_i = omp_get_wtime();
for(int i = 0; i < n; i++){
ans[i] = v1[i] + v2[i];
printf("%lf ", ans[i]);
}
double wc_f = omp_get_wtime();
*/

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


return 0;
```

}

Output:

```
dhyutin@dhyutin-VirtualBox:~/HPC/Expt2$ gcc -fopenmp vectormultparallel.c && export OMP_NUM_THREADS=1 && ./a.out
10000
Time : 0.000043
dhyutin@dhyutin-VirtualBox:~/HPC/Expt2$ gcc -fopenmp vectormultparallel.c && export OMP_NUM_THREADS=2 && ./a.out
10000
Time : 0.000175
dhyutin@dhyutin-VirtualBox:~/HPC/Expt2$ gcc -fopenmp vectormultparallel.c && export OMP_NUM_THREADS=4 && ./a.out
10000
Time : 0.000253
dhyutin@dhyutin-VirtualBox:~/HPC/Expt2$ gcc -fopenmp vectormultparallel.c && export OMP_NUM_THREADS=8 && ./a.out
10000
Time : 0.000805
dhyutin@dhyutin-VirtualBox:~/HPC/Expt2$ gcc -fopenmp vectormultparallel.c && export OMP_NUM_THREADS=16 && ./a.out
10000
Time : 0.000970
dhyutin@dhyutin-VirtualBox:~/HPC/Expt2$ gcc -fopenmp vectormultparallel.c && export OMP_NUM_THREADS=32 && ./a.out
10000
Time : 0.001919
dhyutin@dhyutin-VirtualBox:~/HPC/Expt2$ gcc -fopenmp vectormultparallel.c && export OMP_NUM_THREADS=64 && ./a.out
10000
Time : 0.004385
dhyutin@dhyutin-VirtualBox:~/HPC/Expt2$ gcc -fopenmp vectormultparallel.c && export OMP_NUM_THREADS=128 && ./a.out
10000
Time : 0.006401
dhyutin@dhyutin-VirtualBox:~/HPC/Expt2$
```

Speedup V/S Number of Processors(threads):

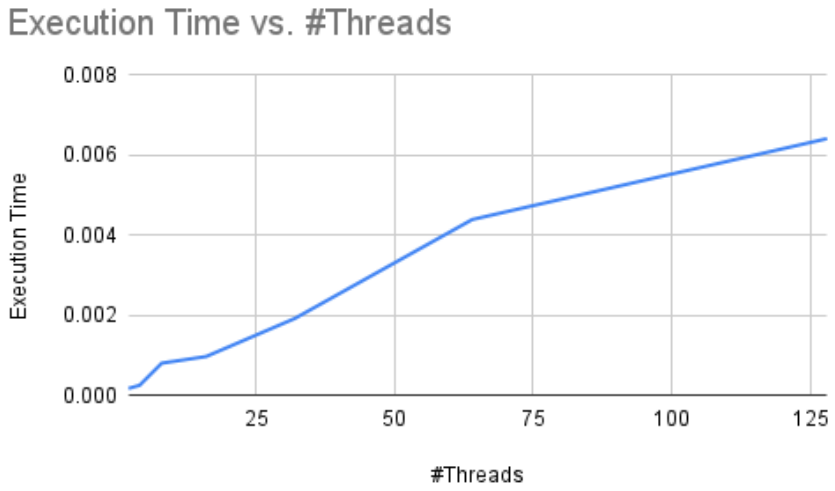


Inference:

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).

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 multiplication, one thread is sufficient to get optimum time.

Parallelization Factor (f):

#Threads	Execution Time	Speed UP	Efficiency (in %)	f
1	0.000043	1	100	n/a
2	0.000175	0.2457142857	12.28571429	-6.139534884

4	0.000253	0.1699604743	4.249011858	-6.511627907
8	0.000805	0.05341614907	0.6677018634	-20.25249169
16	0.00097	0.04432989691	0.2770618557	-22.99534884
32	0.001919	0.02240750391	0.07002344971	-45.03525881
64	0.004385	0.009806157355	0.01532212087	-102.5795496
128	0.006401	0.006717700359	0.005248203406	-149.0247207

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
