HPC LAB 6 : VECTOR DOT PRODUCT

Name: Mridul Harish
Roll No: CED18I034
Programming Environment: OpenMP
Problem: Matrix Multiplication
Date: 26th August 2021

Hardware Configuration:

CPU NAME : Intel Core i5-8250U @ 8x 3.4GHz
Number of Sockets : 1
Cores per Socket : 4
Threads per core : 2
L1 Cache size : 32KB
L2 Cache size : 256KB
L3 Cache size(Shared): 6MB
RAM : 8 GB

REDUCTION

Serial Code:

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

int main(int argc, char* argv[])
{
   float start_time; float end_time; float exec_time;
   start_time = omp_get_wtime();

   int n = 50000;
   double a[n]; double b[n]; double c = 0;

   double random_num(double low, double high)
   {
                double d = (double)rand() / ((double)RAND_MAX + 1);
```

```
                return (low + d * (high - low));
    }

    for(int i = 0; i < n; i = i+1)
    {
        a[i] = random_num(0, 10010000100);
        b[i] = random_num(0, 10010000100);

        c = c + (a[i]*b[i]);
    }

    end_time = omp_get_wtime();
    exec_time = end_time - start_time;
    printf("%f\n", exec_time);
}
```

Parallel Code:

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

int main(int argc, char* argv[])
{
    float start_time; float end_time; float exec_time;
    start_time = omp_get_wtime();

    int n = 50000;
    double a[n]; double b[n]; double c = 0;

    double random_num(double low, double high)
    {
                double d = (double)rand() / ((double)RAND_MAX + 1);
                return (low + d * (high - low));
    }

    #pragma omp parallel for reduction(+ : c)
    for(int i = 0; i < n; i = i+1)
```

```
    {
        a[i] = random_num(0, 10010000100);
        b[i] = random_num(0, 10010000100);

        for(int k = 1; k < n; k = k+1);
        c = c + (a[i]*b[i]);
    }

    end_time = omp_get_wtime();
    exec_time = end_time - start_time;
    printf("%f\n", exec_time);
}
```

Compilation and Execution:

For enabling OpenMP environment use -fopenmp flag while compiling using gcc;

```
gcc -fopenmp dot_product_reduction.c
```

For execution use;

```
export OMP_NUM_THREADS = x (Where x is the number of threads we are deploying)
./a.out
```

OBSERVATION TABLE

No of Threads	Execution Time	Speedup	Parallel Fraction
1	11.439453		
2	5.722383	1.999071541	0.9995355547
4	2.914062	3.925603848	0.9936828273
6	2.463867	4.642885756	0.9415400544
8	2.493164	4.588327523	0.8937779005
10	2.334961	4.899205169	0.8843169531
12	2.408203	4.750202952	0.8612538316
16	2.295898	4.982561508	0.8525866869
20	2.262695	5.055676085	0.8444237031

24	2.21875	5.155809803	0.8410894411
32	2.321289	4.928060659	0.8227926914
64	2.311523	4.948881322	0.8105997531
128	2.354492	4.858565245	0.8004312904

Speed up can be found using the following formula;

S(n)=T(1)/T(n) where, S(n) = Speedup for thread count ‘n’

T(1) = Execution Time for Thread count ‘1’ (serial code)

T(n) = Execution Time for Thread count ‘n’ (serial code)

Parallelization Fraction can be found using the following formula;

S(n)=1/((1 - p) + p/n where, S(n) = Speedup for thread count ‘n’

n = Number of threads

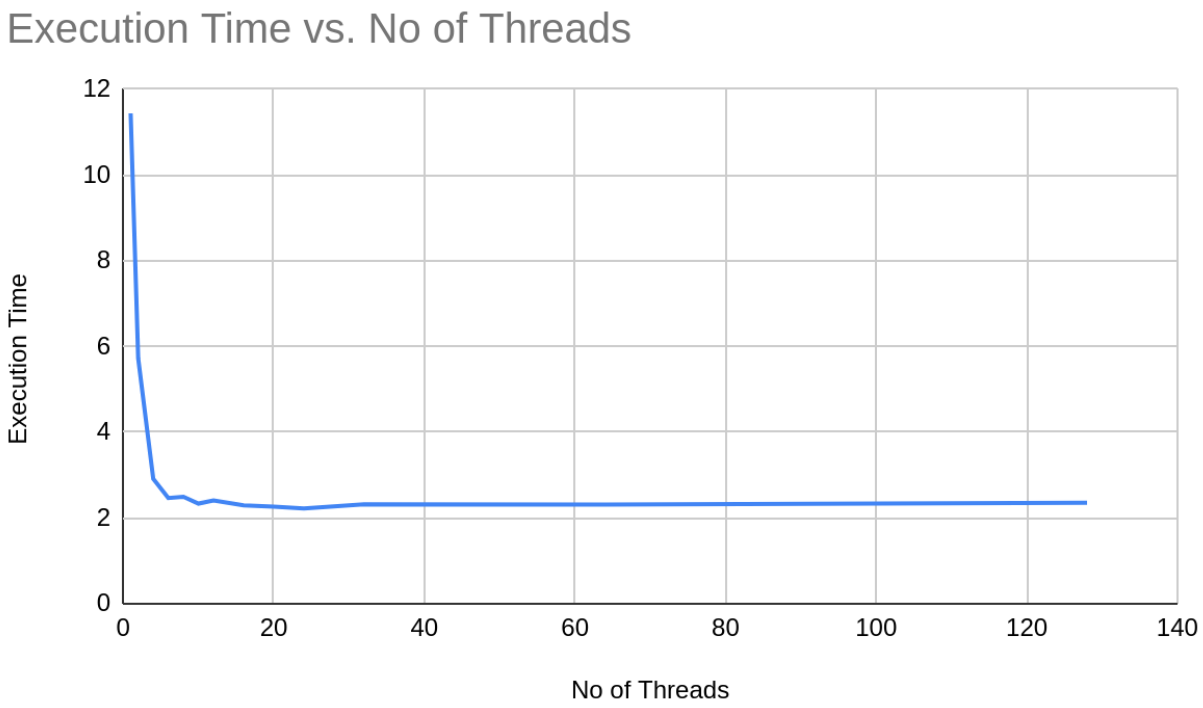
p = Parallelization fraction

Assumption

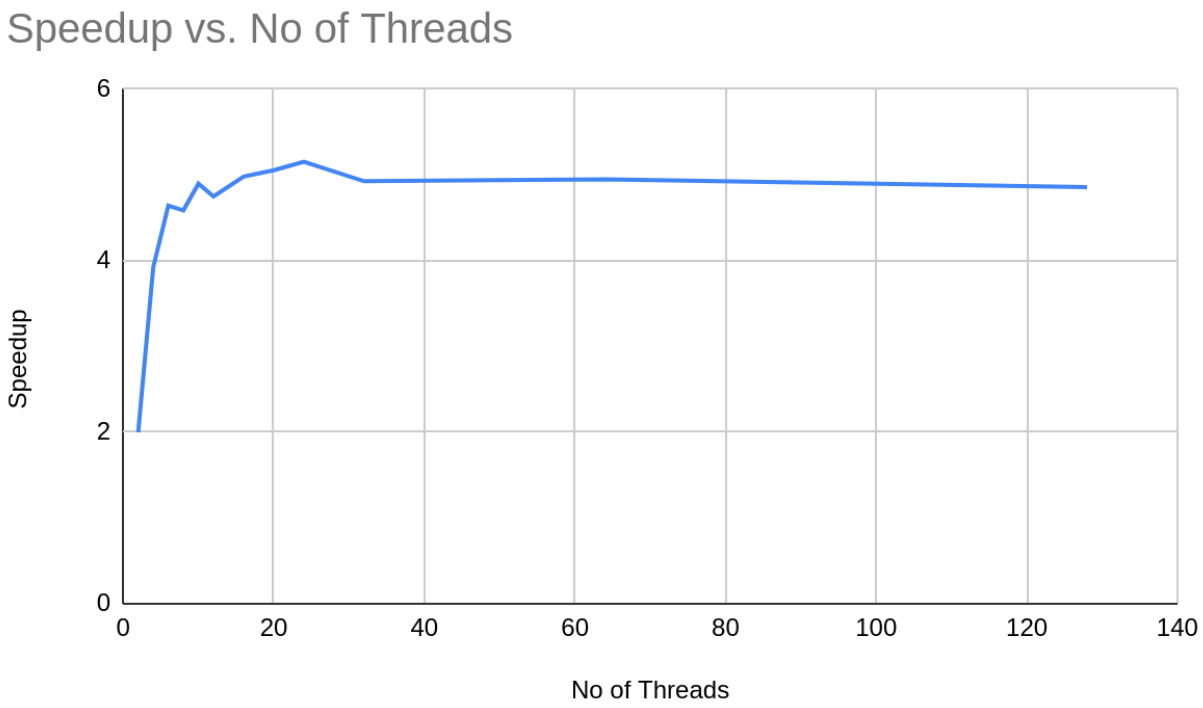
Following extra for loop is added to increase the number of operations in the parallel region to visualize the effect of multi-threading in Vector Dot Product;

```
 for (k = 0; k < n; k++)
{
 Vector Dot Product
}
```

Number of Threads(x axis) vs Execution Time(y axis) :



Number of Threads(x axis) vs Speedup(y axis):



Inference:

(Note: Execution time, graph and inference will be based on hardware configuration)

• At thread count 24 maximum speedup is observed.

• If thread count is more than 24 the execution time continues increasing and the speedup decreases and tapers out in the end.

CRITICAL SECTION

Parallel Code:

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

int main(int argc, char* argv[])
{
   float start_time; float end_time; float exec_time;
   start_time = omp_get_wtime();

   int n = 50000; int i = 0;
   double a[n]; double b[n]; double c = 0; double pc = 0;

   double random_num(double low, double high)
   {
            double d = (double)rand() / ((double)RAND_MAX + 1);
            return (low + d * (high - low));
   }

   #pragma omp parallel for shared(a, b, c, n) private (i, pc)
   for(i = 0; i < n; i = i+1)
   {
      a[i] = random_num(0, 10010000100);
      b[i] = random_num(0, 10010000100);

      for(int k = 1; k < n; k = k+1);
      pc = pc + (a[i]*b[i]);
   }
```

```
    #pragma omp critical
    {
          c = c + pc;
    }

    end_time = omp_get_wtime();
    exec_time = end_time - start_time;
    printf("%f\n", exec_time);
}
```

Compilation and Execution:

For enabling OpenMP environment use -fopenmp flag while compiling using gcc;

```
gcc -fopenmp dot_product_critical.c
```

For execution use;

```
export OMP_NUM_THREADS = x (Where x is the number of threads we are deploying)
./a.out
```

OBSERVATION TABLE

No of Threads	Execution Time	Speedup	Parallel Fraction
1	11.530273		
2	5.799961	1.987991471	0.9939594665
4	2.957031	3.89927363	0.9913893048
6	2.56543	4.494479678	0.9330058013
8	2.448242	4.709613265	0.9001923892
10	2.30957	4.992389492	0.8885501285
12	2.488281	4.633830745	0.8554863595
16	2.355469	4.895107089	0.8487620024
20	2.517578	4.579906958	0.8227946874
24	2.282227	5.05220252	0.8369389828
32	2.391602	4.821150426	0.8181477437

64	2.291016	5.03282081	0.8140233864
128	2.492188	4.626566294	0.7900290846

Speed up can be found using the following formula;

S(n)=T(1)/T(n) where, S(n) = Speedup for thread count ‘n’

T(1) = Execution Time for Thread count ‘1’ (serial code)

T(n) = Execution Time for Thread count ‘n’ (serial code)

Parallelization Fraction can be found using the following formula;

S(n)=1/((1 - p) + p/n where, S(n) = Speedup for thread count ‘n’

n = Number of threads

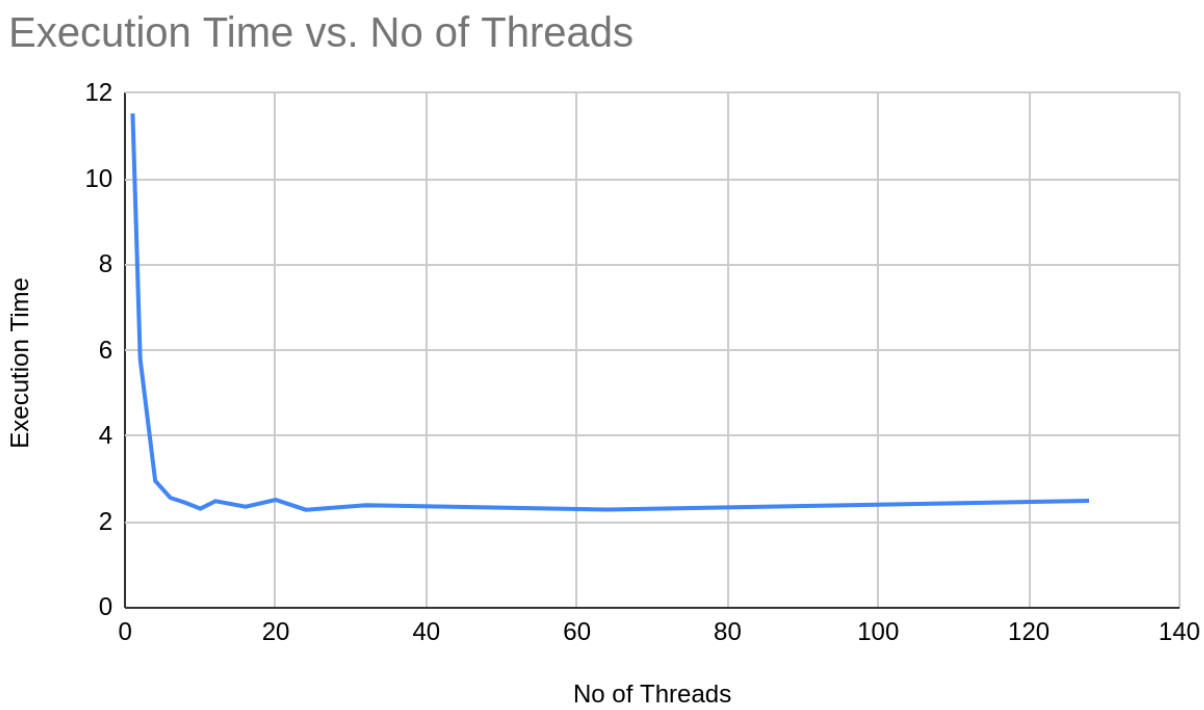
p = Parallelization fraction

Assumption

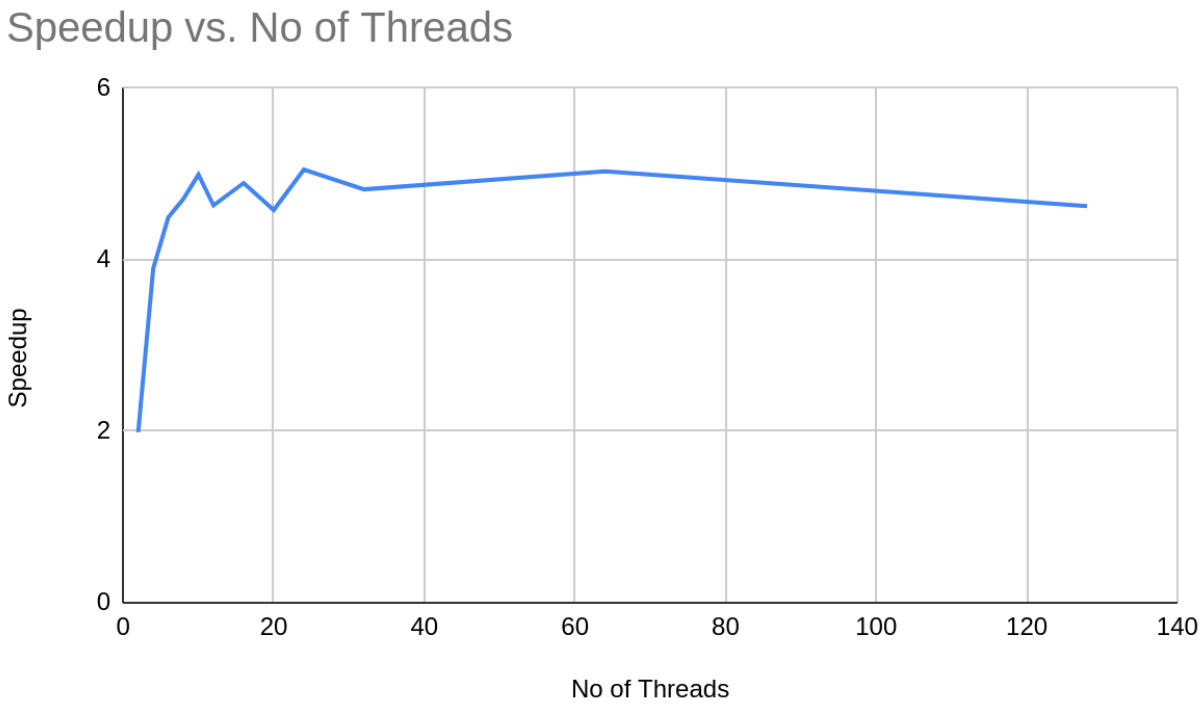
Following extra for loop is added to increase the number of operations in the parallel region to visualize the effect of multi-threading in Vector Addition;

```
 for (k = 0; k < n; k++)
{
 Vector Product
}
```

Number of Threads(x axis) vs Execution Time(y axis) :



Number of Threads(x axis) vs Speedup(y axis):



Inference:

(Note: Execution time, graph and inference will be based on hardware configuration)

• At thread count 24 maximum speedup is observed.

• If thread count is more than 24 the execution time continues increasing and the speedup decreases and tapers out in the end.