HPC LAB 5 : SUM OF N NUMBERS

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 sum = 0;
   int num = 50000;

        for (int i = 0; i <= num; i = i+1)
        {
                for(int k = 0; k < num; k = k+1);
                sum = sum + i;
```

```
        }

        //printf("\n Sum of the first %d number is: %d", num, sum);

        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 sum = 0;
    int num = 50000;

        #pragma omp parallel for reduction(+ : sum)
        for (int i = 0; i <= num; i = i+1)
        {
                for(int k = 0; k < num; k = k+1);
                sum = sum + i;
        }

        //printf("\n Sum of the first %d number is: %d", num, sum);

        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 sum_of_n_numbers_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	11.375		
2	5.689453	1.999313467	0.9996566154
4	3.697266	3.076597681	0.8999541685
6	2.493164	4.562475633	0.9369848967
8	2.743164	4.146671508	0.8672488289
10	2.610352	4.35764985	0.8561316728
12	2.383789	4.771814955	0.862293962
16	2.428711	4.683554363	0.8389194081
20	2.348633	4.843242857	0.835291336
24	2.511719	4.528770933	0.8130673445
32	2.464258	4.615993942	0.8086316739
64	2.560547	4.442410157	0.7871969189
128	2.610742	4.356998892	0.7765510247

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;

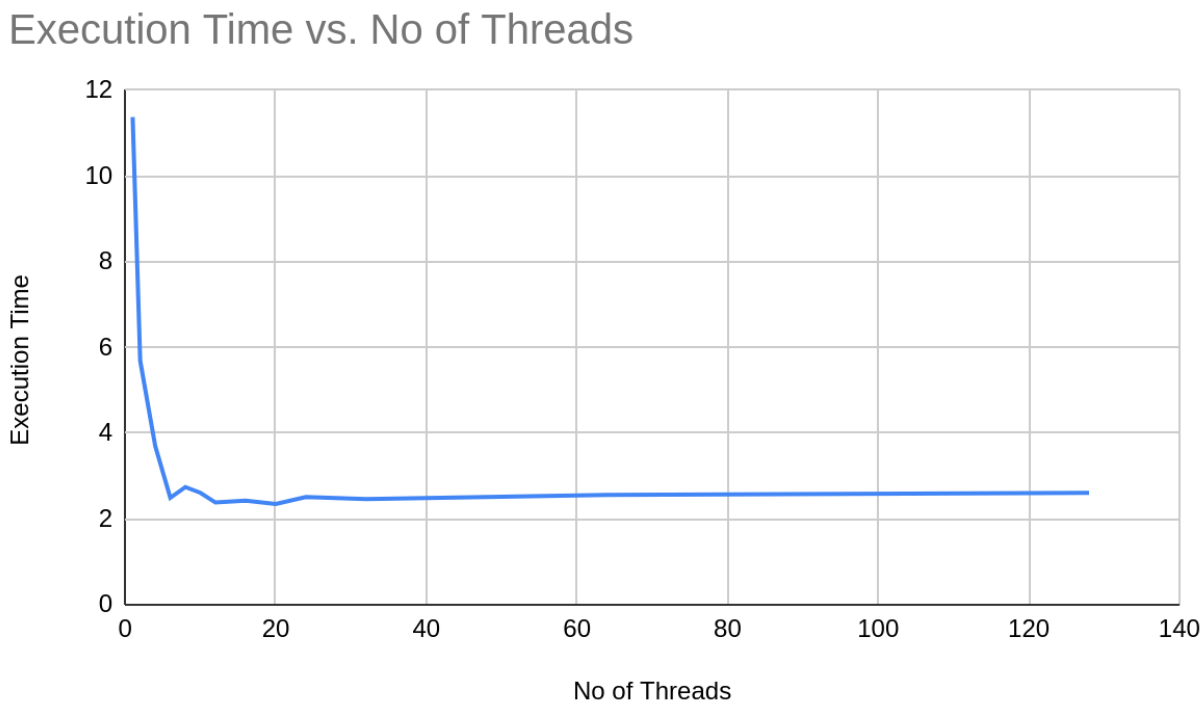
S(n)=1/((1 - p) + p/n where, S(n) = Speedup for thread count ‘n’
n = Number of threads
p = Parallelization fraction

Assumption

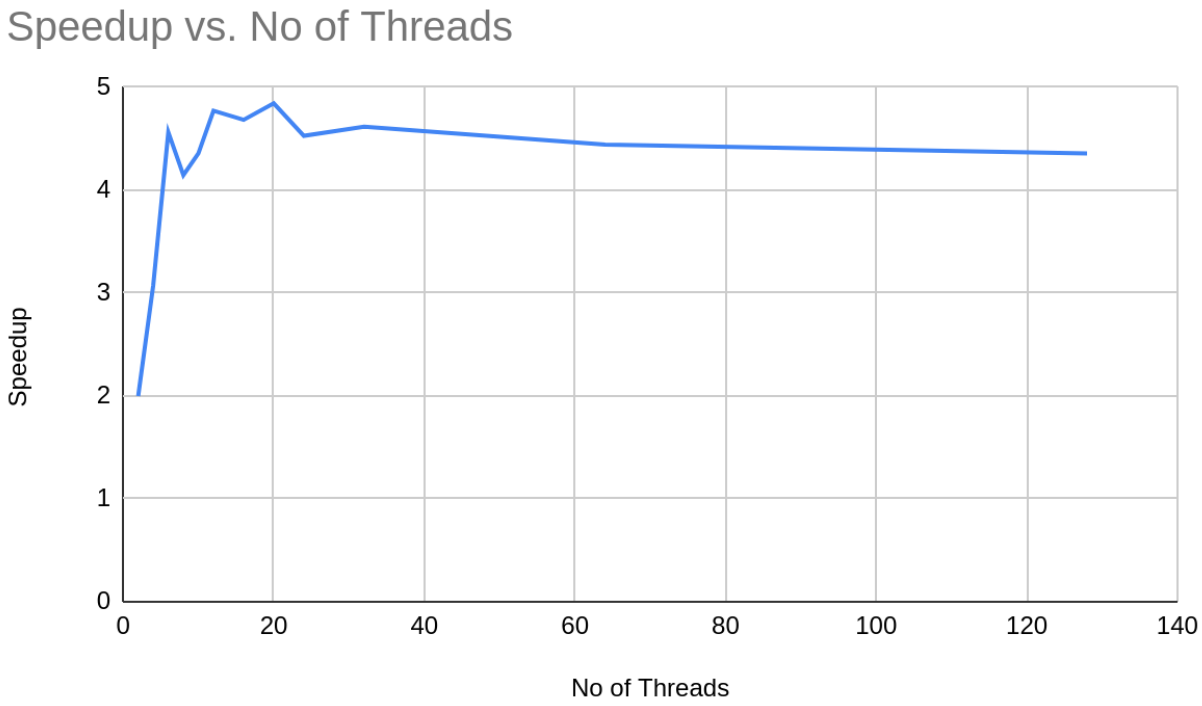
Following extra for loop is added to increase the number of operations in the parallel region to visualize the effect of multi-threading in sum of n numbers;

```
 for (k = 0; k < num; k++)
{
 Sum of N numbers
}
```

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 20 maximum speedup is observed.

• If thread count is more than 20 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 sum = 0; int psum = 0;
    int num = 50000; int i = 0;

                #pragma omp parallel for shared(num) private (i, psum)
        for (i = 0; i <= num; i = i+1)
        {
                for(int k = 0; k < num; k = k+1);
                psum = psum + i;
        }

                #pragma omp critical
                {
                        sum = sum + psum;
                }

        //printf("\n Sum of the first %d number is: %d", num, sum);

        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 sum_of_n_numbers_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	11.438477		
2	5.738281	1.993362995	0.9966704483
4	3.28125	3.486012038	0.9508523439
6	2.616211	4.372153852	0.9255357335
8	2.743164	4.169811575	0.8687782973
10	2.582031	4.430030856	0.8602977088
12	2.413086	4.740186218	0.8607685351
16	2.594727	4.408354713	0.8247018666
20	2.47168	4.627814685	0.8251739881
24	2.636719	4.338147903	0.8029428332
32	2.473633	4.6241609	0.8090266314
64	2.521484	4.536406735	0.7919352001
128	2.517969	4.542739406	0.7860092056

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

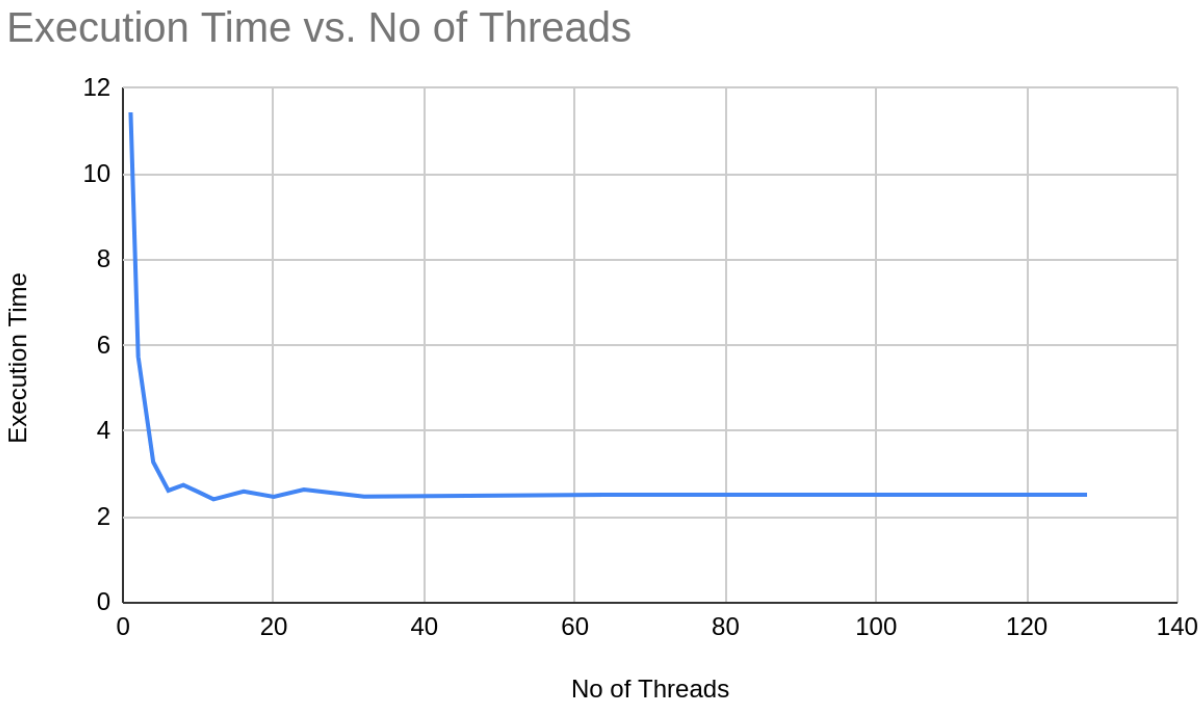
p = Parallelization fraction

Assumption

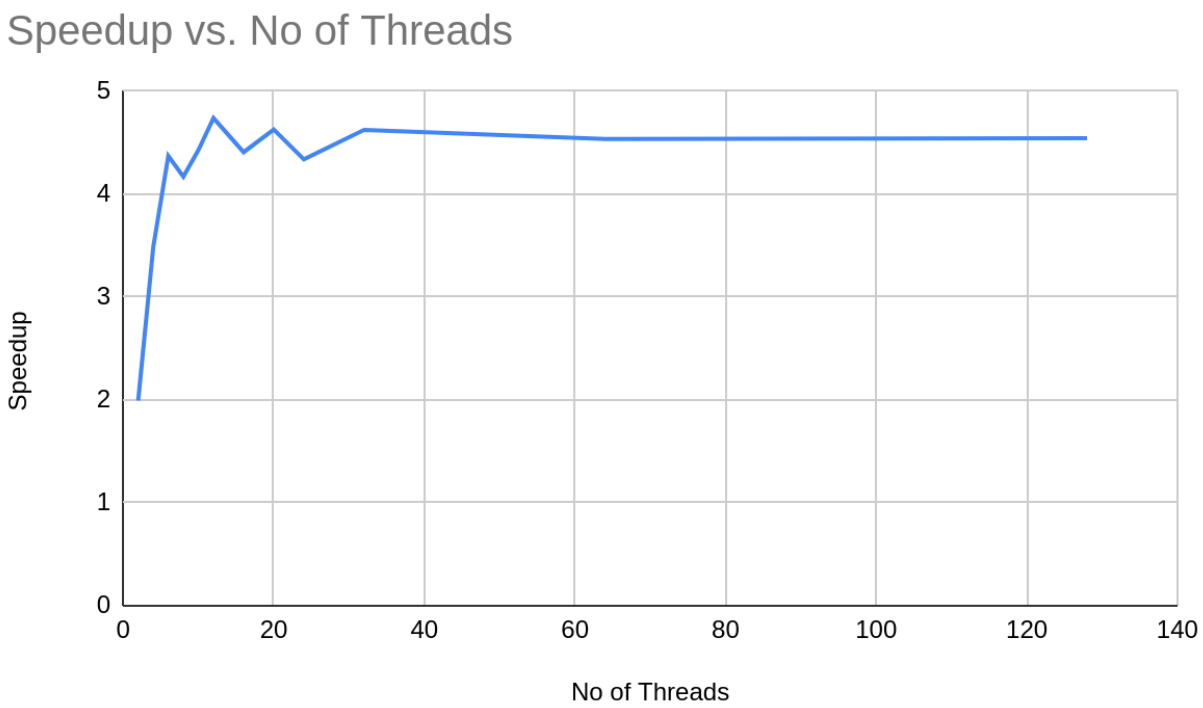
Following extra for loop is added to increase the number of operations in the parallel region to visualize the effect of multi-threading in sum of n numbers;

```
 for (k = 0; k < num; k++)
{
 Sum of N numbers
}
```

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 12 maximum speedup is observed.

• If thread count is more than 12 the execution time continues increasing and the speedup decreases and tapers out in the end.