HPC LAB 7 : BLOCK MATRIX MULTIPLICATION

Name: Mridul Harish
Roll No: CED18I034
Programming Environment: OpenMP
Problem: Matrix Multiplication
Date: 2nd September 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

NORMAL MATRIX MULTIPLICATION

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();

   double a[500][500]; double b[500][500]; double c[500][500];
         int row = 500, column = 500;

   //multiplication
   for(int i = 0; i < row; i = i+1)
         {
               for(int j = 0; j < column; j = j+1)
               {
```

```
                        c[i][j] = 0;
                        a[i][j] = i+j+12.549;
                        b[i][j] = i+j+97.949;
                        for(int k = 0; k < column; k = k+1)
                        {
                                c[i][j] += a[i][k]*b[k][j];
                        }
                }
        }
    end_time = omp_get_wtime();
    exec_time = end_time - start_time;
    printf("%f\n", exec_time);
}
```

Parallel Code:

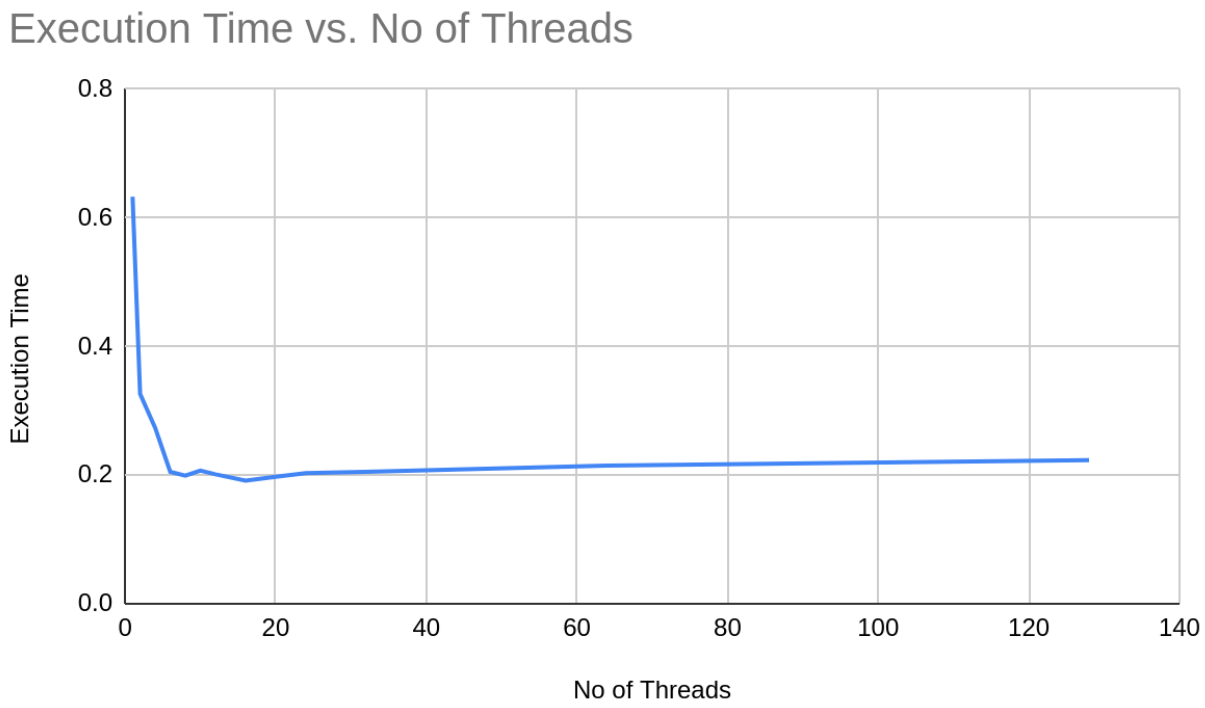
```
#include<omp.h>
#include<stdlib.h>
#include<stdio.h>

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

   double a[500][500]; double b[500][500]; double c[500][500];
          int row = 500, column = 500;

    //multiplication
    #pragma omp parallel for
    for(int i = 0; i < row; i = i+1)
          {
                #pragma omp parallel for
                for(int j = 0; j < column; j = j+1)
                {
                        c[i][j] = 0;
                        a[i][j] = i+j+12.549;
                        b[i][j] = i+j+97.949;
                        for(int k = 0; k < column; k = k+1)
                        {
                                c[i][j] += a[i][k]*b[k][j];
                        }
```

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

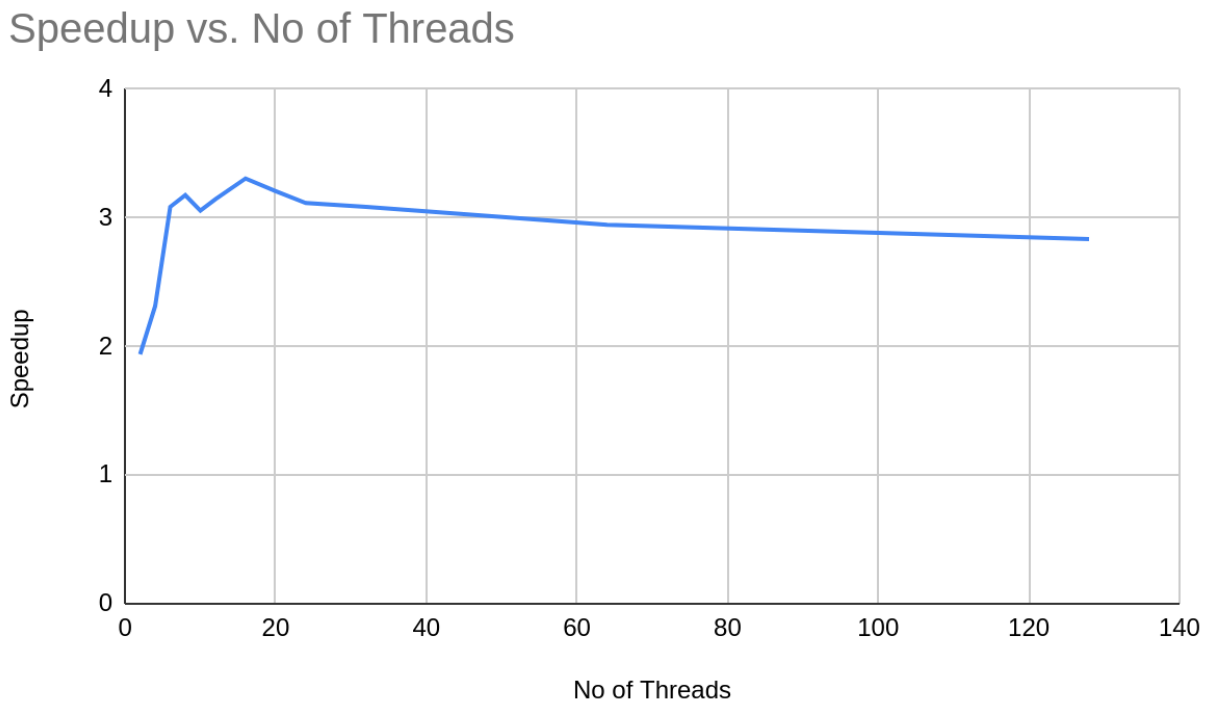
OBSERVATION TABLE

No of Threads	Execution Time	Speedup	Parallel Fraction
1	0.632812		
2	0.326172	1.940117484	0.9691345929
4	0.273438	2.314279654	0.7572001374
6	0.205078	3.085713728	0.8111110409
8	0.199219	3.176464092	0.7830680473
10	0.207031	3.056605049	0.747599603
12	0.201172	3.145626628	0.7441072546
16	0.191406	3.306124155	0.7440330883
20	0.197266	3.207912159	0.7244955432
24	0.203125	3.115382154	0.7085343569
32	0.205078	3.085713728	0.6977299276
64	0.214766	2.946518536	0.6711023982
128	0.223125	2.836132213	0.6525048227

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



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



BLOCK MATRIX MULTIPLICATION

Serial Code:

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

#define B 10

double random_num(double x)
{
        double num;
        num = (float)rand()/RAND_MAX * x ;
        return num;
}

int main(int argc, char* argv[])
{
        if (argc == 2)
        omp_set_num_threads(atoi(argv[1]));

        float startTime, endTime, execTime;
        int ii, jj, kk, i , j ,k;
        int n = 500;
        double a[n][n],b[n][n],c[n][n];
        startTime = omp_get_wtime();

        for (ii = 0; ii < n; ii+=B)
        {
        for (jj = 0; jj < n; jj+=B)
        {
        for (kk = 0; kk < n; kk+=B)
        {
                for (i = ii; i < ii+B; i++)
                {
                for (j = jj; j < jj+B; j++)
                {
                for (k = kk; k < kk+B; k++)
                {
                        a[i][j] = random_num(540012301.0);
                        b[i][j] = random_num(205465410.0);
                        c[i][j] = 0;
                        for (int l = 0; l < 100; l++);
```

```
                        c[i][j] += a[i][k] * b[k][j];
                }
                }
                }
        }
        }
        }

        endTime = omp_get_wtime();
        execTime = endTime - startTime;
        printf("%f \n", execTime);
}
```

Parallel Code:

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

#define B 10

double random_num(double x)
{
        double num;
        num = (float)rand()/RAND_MAX * x ;
        return num;
}

int main(int argc, char* argv[])
{
        if (argc == 2)
        omp_set_num_threads(atoi(argv[1]));

        float startTime, endTime, execTime;
        int ii, jj, kk, i , j ,k;
        int n = 500;
        double a[n][n],b[n][n],c[n][n];
        startTime = omp_get_wtime();

        #pragma omp parallel for
        for (ii = 0; ii < n; ii+=B)
        {
        for (jj = 0; jj < n; jj+=B)
        {
```

```
        for (kk = 0; kk < n; kk+=B)
        {
                for (i = ii; i < ii+B; i++)
                {
                for (j = jj; j < jj+B; j++)
                {
                for (k = kk; k < kk+B; k++)
                {
                        a[i][j] = random_num(540012301.0);
                        b[i][j] = random_num(205465410.0);
                        c[i][j] = 0;
                        for (int l = 0; l < 100; l++);
                        c[i][j] += a[i][k] * b[k][j];
                }
                }
                }
        }
        }
        }

        endTime = omp_get_wtime();
        execTime = endTime - startTime;
        printf("%f \n", execTime);
}
```

Compilation and Execution:

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

```
gcc -fopenmp matrix_block.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	0.318359		
2	0.451172	0.705626679	-0.8343599521
4	0.389648	0.8170425615	-0.2985685971
6	0.333984	0.9532163217	-0.0588957749
8	0.24707	1.288537661	0.2559159404
10	0.144531	2.202703918	0.6066805783
12	0.15332	2.076434907	0.5655330789
16	0.155273	2.050317827	0.5464221209
20	0.137695	2.312059261	0.5973527734
24	0.09668	3.29291477	0.7265923608
32	0.09375	3.395829333	0.7282798715
64	0.094727	3.360805261	0.7136022989
128	0.089844	3.543464227	0.7234421854

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)

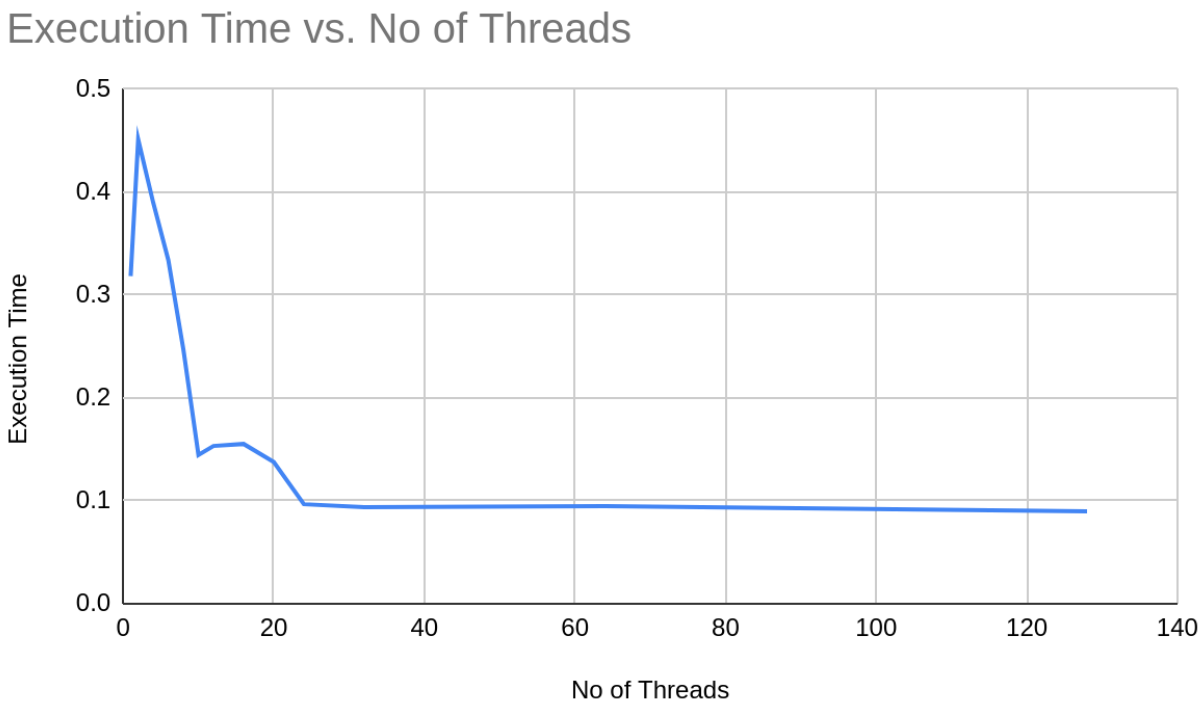
Parallelization Fraction can be found using the following formula;

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

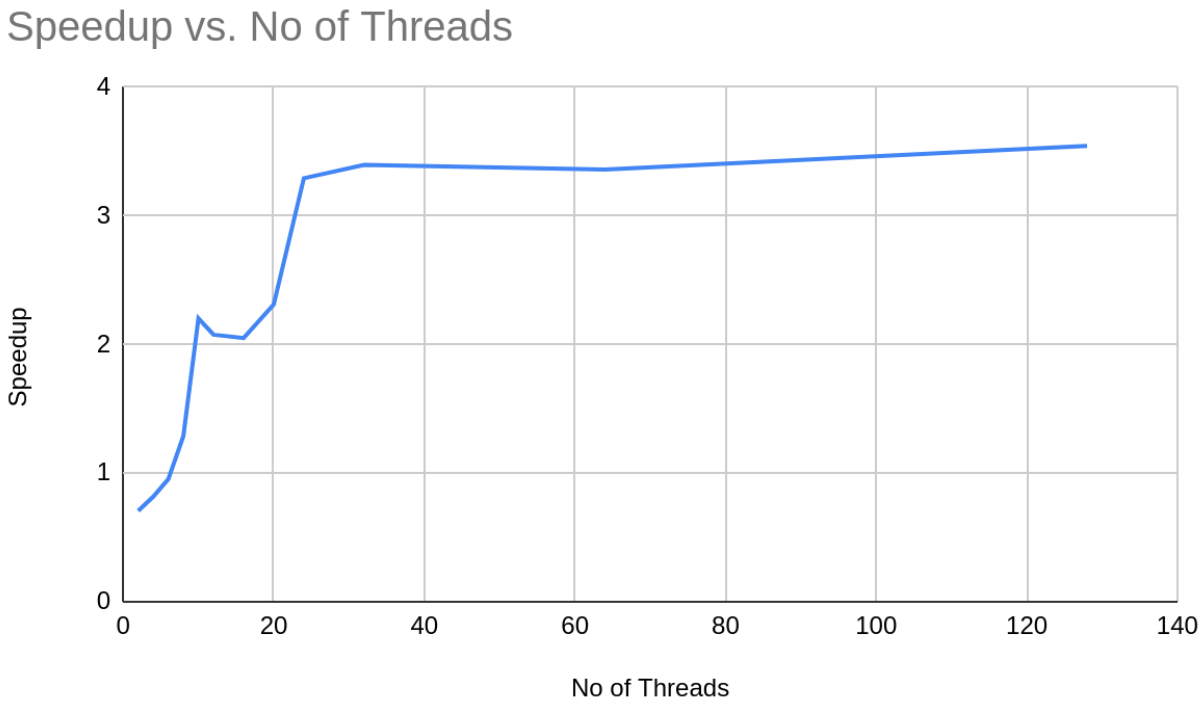
n = Number of threads

p = Parallelization fraction

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



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



Comparing Normal Matrix Multiplication and Block Matrix Multiplication

No of Threads	Execution Time(Block Matrix)	Execution Time(Normal matrix)
1	0.318359	0.632812
2	0.451172	0.326172
4	0.389648	0.273438
6	0.333984	0.205078
8	0.24707	0.199219
10	0.144531	0.207031
12	0.15332	0.201172
16	0.155273	0.191406
20	0.137695	0.197266
24	0.09668	0.203125
32	0.09375	0.205078
64	0.094727	0.214766
128	0.089844	0.223125

Inference:

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

- As no. of threads increase, execution time decreases and speedup increases.
- As thread count increases, speedup increases and is observed at maximum on thread no. 16.
- Matrix Block multiplication is almost doubly faster than normal matrix multiplication.