Report for HPC LAB

Name: Bhaskar R
Roll No: CED18I009
Programming Environment: OpenMP
Problem: Matrix Multiplication
Date: 19th August 2021

Hardware Configuration:

CPU NAME : Intel core i5 – 8250U @ 1.60 Ghz
Number of Sockets : 1
Cores per Socket : 4
Threads per core : 2
L1 Cache size : 32KB (Per Core)
L2 Cache size : 256KB (Per Core)
L3 Cache size : 6MB (Shared)
RAM : 8 GB

Serial Code:

```
#include <stdio.h>
#include <time.h>
#include <stdlib.h>
#include <omp.h>
#define n 100
#define m 1000
int main()
{
	double a[n][n], b[n][n], c[n][n];
	float startTime, endTime, execTime;
	int i, k;
	int omp_rank;
	float rtime;
	startTime = omp_get_wtime();
	for (int i = 0; i < n; i++)
	{
		for (int j = 0; j < n; j++)
		{
			a[i][j] = rand() % 500;
			b[i][j] = rand() % 500;
		}
	}
	for (int i = 0; i < n; i++)
	{
		for (int j = 0; j < n; j++)
		{
			c[i][k] = 0;
			for (int l = 0; l < m; l++)
			{
```

```
                    for (int k = 0; k < n; k++)
                    {
                        c[i][j] += a[i][k] * b[k][i];
                    }
                }
            }
        }
        endTime = omp_get_wtime();
        execTime = endTime - startTime;
        rtime = execTime;
        printf("\n rtime=%f\n", rtime);
        return (0);
}
```

Parallel Code:

```
#include <bits/stdc++.h>
using namespace std;
#include <omp.h>
#define n 100
#define m 1000
int main()
{
        srand(time(0));
        double a[n][n], b[n][n], c[n][n] = {{0}};
        float startTime, endTime, execTime;
        int i, k;
        int omp_rank;
        float rtime;
        startTime = omp_get_wtime();
        for (int i = 0; i < n; i++)
        {
            for (int j = 0; j < n; j++)
            {
                a[i][j] = rand() % 500;
                b[i][j] = rand() % 500;
            }
        }
        for (int i = 0; i < n; i++)
        {
            for (int j = 0; j < n; j++)
            {
                for (int k = 0; k < n; k++)
                {
                    c[i][j] += a[i][k] * b[k][j];
                }
            }
```

```
    }
    cout << "Matrix A " << endl;
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
            cout << a[i][j] << "\t";
        cout << endl;
    }
    cout << "Matrix B " << endl;
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
            cout << b[i][j] << "\t";
        cout << endl;
    }
    cout << "Resultant Matrix " << endl;
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
            cout << c[i][j] << "\t";
        cout << endl;
    }
    endTime = omp_get_wtime();
    execTime = endTime - startTime;
    rtime = execTime;
    printf("\n rtime=%f\n", rtime);
    return (0);
}
```

Compilation and Execution:

For enabling OpenMP environment use -fopenmp flag while

compiling using g++. **g++ -fopenmp matrixmul.cpp**

For execution use

./a.out

Observations:

Number of Threads	Execution Time	Speed-up	Parallelization Fraction
1	5.628906	1	
2	2.820312	1.99	99.4
4	1.445312	3.90	99.1
6	1.550781	3.63	86.9
8	1.339844	4.23	87.2
10	1.789062	3.16	75.9
12	1.796875	3.14	74.3
16	1.703125	3.30	74.3
20	1.617188	3.48	75.0
32	1.800781	3.12	70.1
64	1.820312	3.09	68.7
128	1.835938	3.06	67.8
150	1.875000	3.00	67.1

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

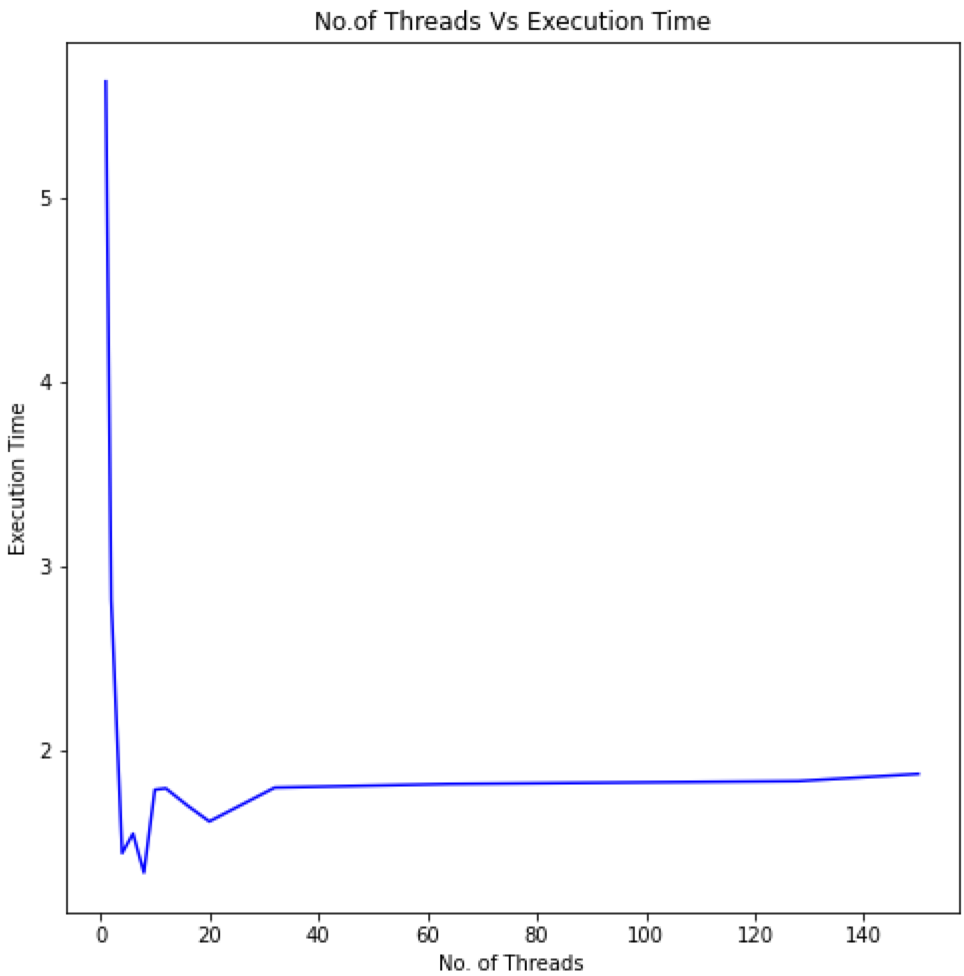
p = Parallelization fraction

Assumption:

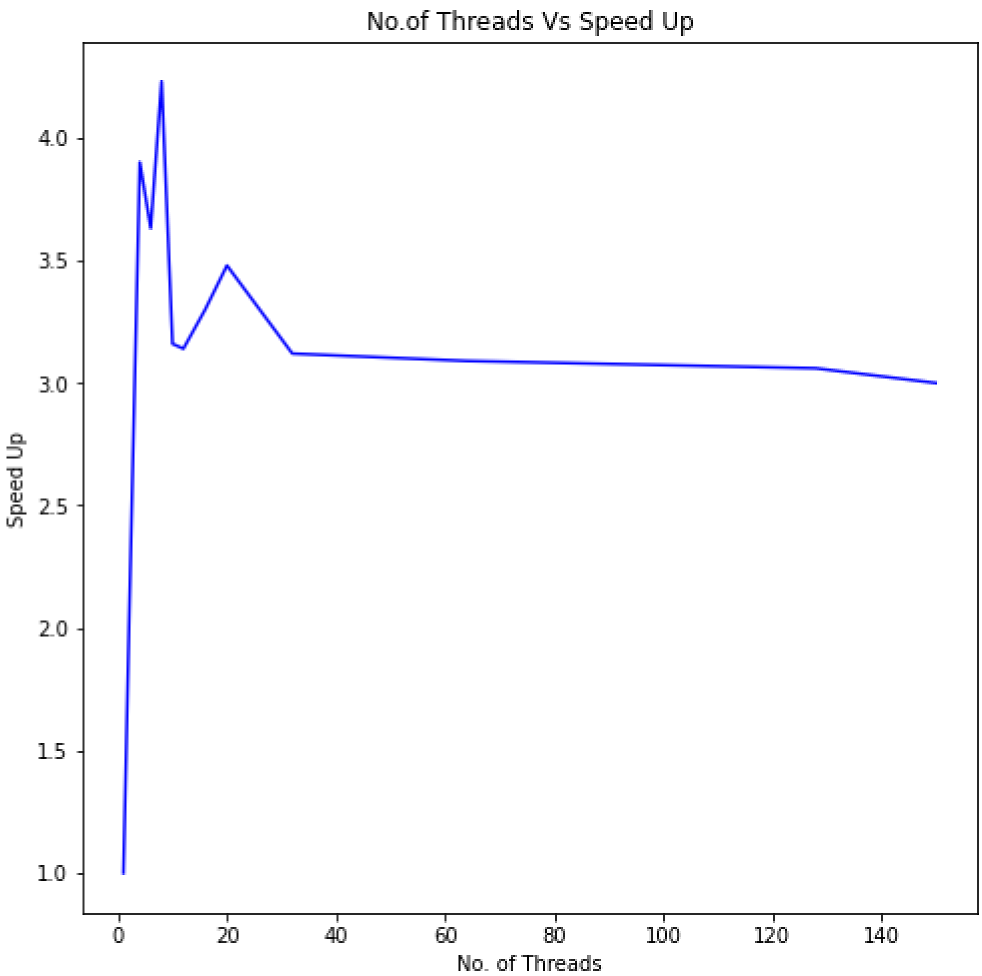
Following extra for loop is added to increase the number of operations in the parallel region to visualize the effect of multi-threading in Matrix Multiplication.

```
for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
                c[i][k] = 0;
                for (int l = 0; l < m; l++)
                        for (int k = 0; k < n; k++)
                                c[i][j] += a[i][k] * b[k][i];
```

Number of Threads vs Execution Time:



Number of Threads vs Speed Up:



Inference:
(Note: Execution time, graph and inference will be based on hardware configuration**)**
- At thread count 8 maximum speedup is observed as the maximum number of parallel threads supported by the hardware is 8.
- If the thread count is more than 10 then the execution time increases slightly and tapers out after 32 threads.