Report for HPC LAB

Name: Bhaskar R
Roll No: CED18I009
Programming Environment: OpenMP
Problem: Block based Matrix Multiplication
Date: 2nd September 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 <iostream>
#include <ctime>
#include <omp.h>
using namespace std;
#define n 200
int main()
{
	int blocksize[] = {10, 25, 50, 75, 100, 125, 150};
	double a[n][n], b[n][n], c[n][n];
	float startTime, endTime, execTime;
	int B;
	for (int blocks = 0; blocks < 7; blocks++)
	{
		B = blocksize[blocks];
		printf("\nB = %d ", B);
		startTime = omp_get_wtime();
#pragma omp for collapse(2)
		for (int i = 0; i < n; i++)
		{
			for (int j = 0; j < n; j++)
			{
				a[i][j] = (i + 1) * 10.236;
				b[i][j] = (i + 1) * 152.123;
				c[i][j] = 0.0;
			}
		}
		for (int jj = 0; jj < n; jj = jj + B)
		{
			for (int kk = 0; kk < n; kk += B)
			{
				for (int i = 0; i < n; i++)
				{
```

```
                                        for (int j = jj; j < min(jj + B, n); j++)
                                        {
                                                double r = 0;
                                                for (int k = kk; k < min(kk + B, n); k++)
                                                        r = r + a[i][k] * b[k][j];
                                                double temp = 3.0;
                                                for (int tt = 0; tt < n * 2; tt++)
                                                        temp = tt * 3.2618 * 7.32347 / r;
                                                c[i][j] = c[i][j] + r;
                                        }
                                }
                        }
                }
                endTime = omp_get_wtime();
                execTime = endTime - startTime;
                printf("rtime = %f\n", execTime);
        }
        return 0;
}
```

Parallel Code:

```
#include <iostream>
#include <ctime>
#include <omp.h>
using namespace std;
#define n 200
int main()
{
        int x = 0;
        int threads[] = {1, 2, 4, 6, 8, 10, 12, 16, 20, 32, 64, 128, 150};
        int blocksize[] = {10, 25, 50, 75, 100, 125, 150};
        double a[n][n], b[n][n], c[n][n];
        float startTime, endTime, execTime;
        int B;
        for (int blocks = 0; blocks < 7; blocks++)
        {
                x = 0;
                B = blocksize[blocks];

printf("\n\t\tB=%d\n_____________________________________________\n", B);
                for (int k = 0; k < 13; k++)
                {
                        omp_set_num_threads(threads[k]);
                        startTime = omp_get_wtime();
#pragma omp for collapse(2)
                        for (int i = 0; i < n; i++)
                        {
                                for (int j = 0; j < n; j++)
                                {
                                        a[i][j] = (i + 1) * 10.236;
                                        b[i][j] = (i + 1) * 152.123;
```

```
                                        c[i][j] = 0.0;
                                }
                        }
#pragma omp parallel
                        {
#pragma omp for collapse(3)
                                for (int jj = 0; jj < n; jj = jj + B)
                                {
                                        for (int kk = 0; kk < n; kk += B)
                                        {
                                                for (int i = 0; i < n; i++)
                                                {
                                                        for (int j = jj; j < min(jj + B, n); j++)
                                                        {
                                                                double r = 0;
                                                                for (int k = kk; k < min(kk + B,
n); k++)

                                                                        r = r + a[i][k] * b[k][j];
                                                                double temp = 3.0;
                                                                for (int tt = 0; tt < n * 2; tt++)
                                                                        temp = tt * 3.2618 *
7.32347 / r;

                                                                c[i][j] = c[i][j] + r;
                                                        }
                                                }
                                        }
                                }
                        }
                        endTime = omp_get_wtime();
                        execTime = endTime - startTime;
                        printf("\nNumber of Threads = %d\t rtime = %f\n", threads[x++],
execTime);
                }
        }
        return 0;
}
```

Compilation and Execution:

For enabling OpenMP environment use -fopenmp flag while

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

For execution use

./a.out

Observations:

BLOCKS = 10

Number of Threads	Execution Time	Speed-up	Parallelization Fraction
1	0.739	1	
2	0.380	1.94	96.90
4	0.220	3.35	93.53
6	0.208	3.55	86.18
8	0.205	3.60	82.3
10	0.202	3.65	80.66
12	0.191	3.86	80.82
16	0.180	4.10	80.65
20	0.175	4.22	80.31
32	0.174	4.24	78.88
64	0.173	4.27	77.79
128	0.184	4.01	75.65
150	0.187	3.95	75.18

BLOCKS = 25

Number of Threads	Execution Time	Speed-up	Parallelization Fraction
1	0.299	1	
2	0.157	1.90	94.73
4	0.093	3.21	91.79
6	0.096	3.11	81.41
8	0.094	3.18	78.34
10	0.088	3.39	78.33

12	0.089	3.35	76.52
16	0.087	3.43	75.56
20	0.082	3.64	76.34
32	0.085	3.51	73.81
64	0.077	3.88	75.40
128	0.095	3.14	68.68
150	0.090	3.32	70.34

BLOCKS = 50

Number of Threads	Execution Time	Speed-up	Parallelization Fraction
1	0.186	1	
2	0.103	1.80	88.88
4	0.060	3.10	90.32
6	0.058	3.20	82.50
8	0.061	3.04	76.69
10	0.052	3.57	79.98
12	0.054	3.44	77.37
16	0.057	3.26	73.94
20	0.047	3.95	78.61
32	0.053	3.50	73.73
64	0.052	3.57	73.13
128	0.057	3.26	69.87
150	0.048	3.87	74.65

BLOCKS = 75

Number of Threads	Execution Time	Speed-up	Parallelization Fraction
1	0.165	1	
2	0.095	1.73	84.39
4	0.069	2.39	77.54
6	0.050	3.30	83.63
8	0.061	2.70	71.95
10	0.052	3.17	76.06
12	0.046	3.58	78.61
16	0.046	3.58	76.87
20	0.042	3.92	78.41
32	0.042	3.92	76.89
64	0.042	3.92	75.67
128	0.048	3.43	71.40
150	0.045	3.66	73.16

BLOCKS = 100

Number of Threads	Execution Time	Speed-up	Parallelization Fraction
1	0.137	1	
2	0.078	1.75	85.71
4	0.051	2.68	83.58
6	0.036	3.80	88.42
8	0.058	2.36	65.85
10	0.044	3.11	75.38
12	0.034	4.02	81.95

16	0.035	3.91	79.38
20	0.037	3.70	76.81
32	0.035	3.91	76.82
64	0.039	3.51	72.64
128	0.043	3.18	69.09
150	0.041	3.34	70.53

BLOCKS = 125

Number of Threads	Execution Time	Speed-up	Parallelization Fraction
1	0.149	1	
2	0.095	1.56	71.79
4	0.049	3.04	89.47
6	0.042	3.54	86.10
8	0.055	2.70	71.95
10	0.041	3.63	80.50
12	0.036	4.13	82.67
16	0.037	4.02	80.13
20	0.031	4.80	83.33
32	0.035	4.25	78.93
64	0.042	3.54	72.89
128	0.034	4.38	77.77
150	0.037	4.02	75.62

BLOCKS = 150

Number of Threads	Execution Time	Speed-up	Parallelization Fraction
1	0.153	1	

2	0.097	1.57	72.61
4	0.062	2.46	79.13
6	0.053	2.88	78.33
8	0.069	2.21	62.57
10	0.033	4.63	87.11
12	0.042	3.64	79.12
16	0.035	4.37	82.25
20	0.038	4.02	79.07
32	0.035	4.37	79.60
64	0.034	4.5	79.01
128	0.037	4.13	76.38
150	0.033	4.63	78.92

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

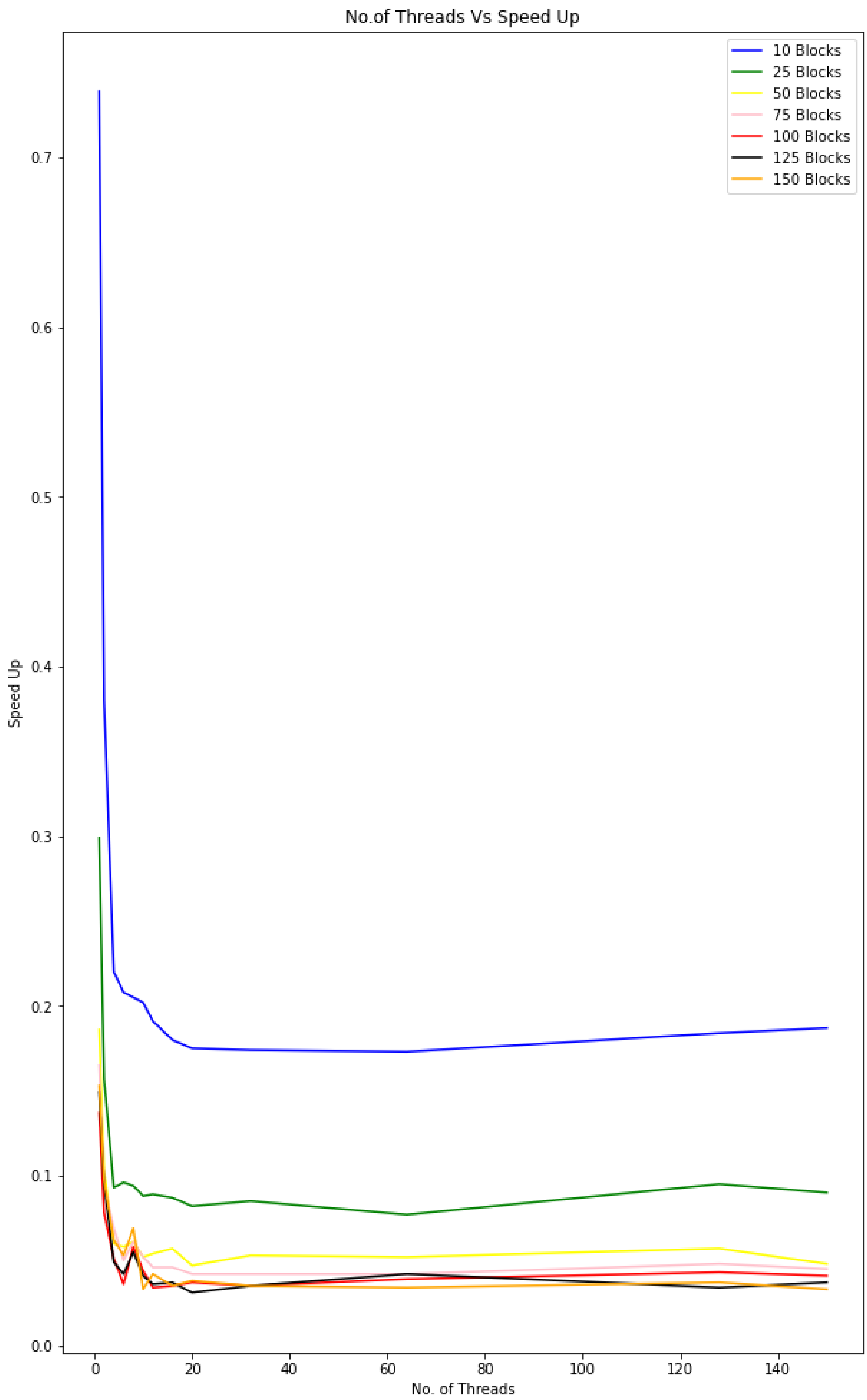
p = Parallelization fraction

Assumption:

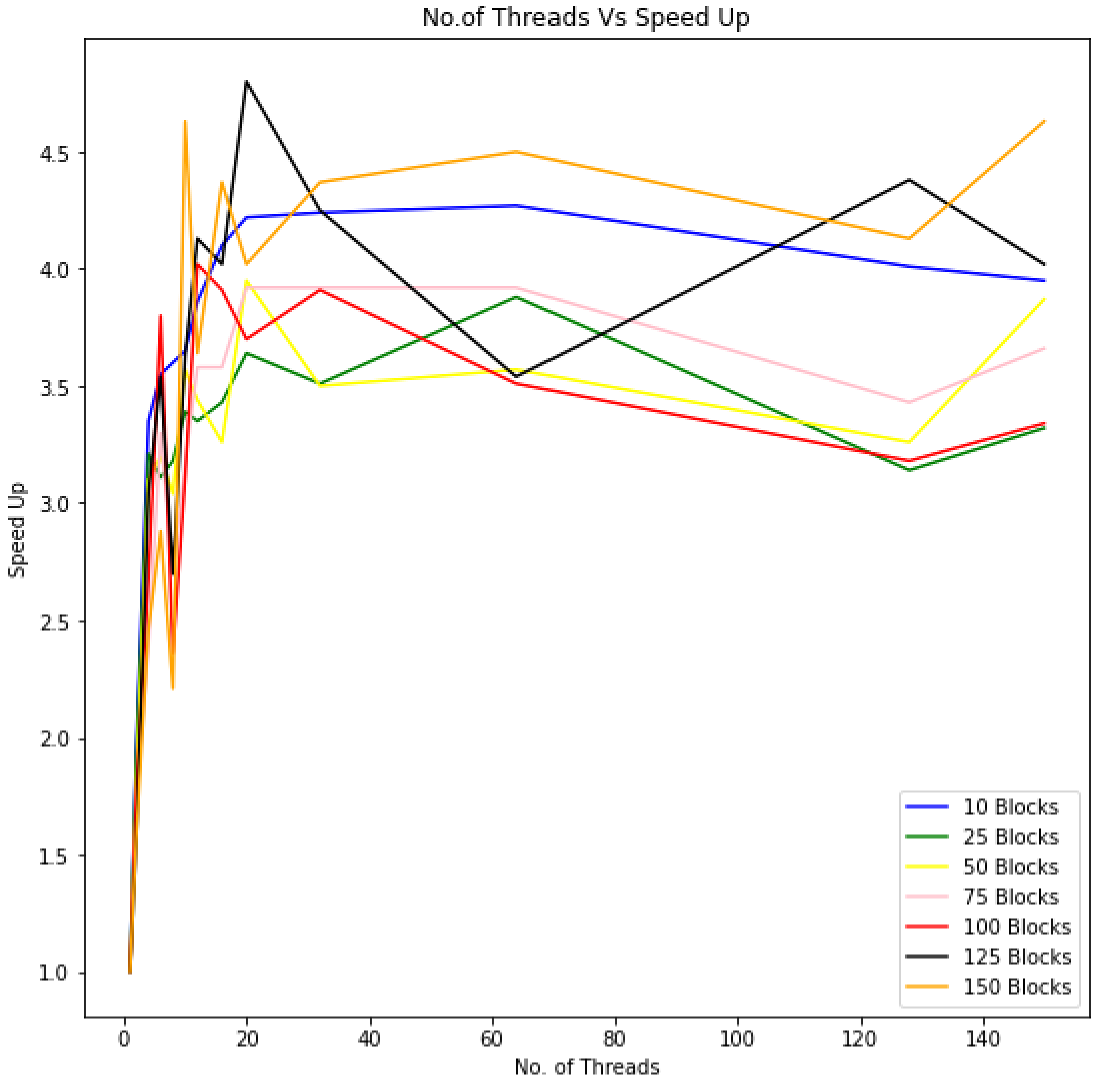
Following extra for loop is added to increase the number of operations in the parallel region to visualize the effect of multi-threading in the block-based matrix multiplication.

```
double temp=3.0;

for(int tt=0;tt<n*2;tt++)

temp=tt*3.2618*7.32347/r;
```

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)
• Compared to the experiment done previously(matrix multiplication), block-based matrix multiplication seems to produce better results in terms of execution times, especially on the matrices with large sizes.
• The normal method is limited by cache size and memory access.
• As we increase the size of the blocks, the execution time is decreasing gradually mainly due to a decrease in block switch overhead.
• The maximum speedup is observed mostly at thread count 16, but it is different for a few block sizes, as the maximum number of parallel threads depends on the hardware as well as the compile