CS 359 Assignment - 5

Name: Somya Mehta

Roll No.: 190001058

Problem Statement:

Given two matrices A and B, multiply them in parallel to store the result in matrix C using OpenMP.

Approach/Logic:

In matrix multiplication of (A x B) matrix with a (B x C) matrix, a total number of operations required = A*B*C But, the computation of all entries of the resultant (A x C) matrix can be done in parallel. So, if we can have A x C threads available then we can perform the matrix multiplication in O(B) time. We can further parallelize this approach by performing all the multiplications and additions required for one cell parallel but that may lead to inconsistency issues. However, we can synchronize it by using locks to solve the inconsistency issues but then we will lose the benefits of parallelism. Hence, we will concentrate only on parallelizing the evaluation of all cells and not on parallelizing the evaluation of one cell.

We want to find the Time Taken, Speed up, and efficiency for the task of multiplying two matrices for a variable number of threads and variable size of matrices being multiplied.

Here, we take the number of threads as [1, 2, 4, 8] and the matrix sizes as [100, 1000, 2000]. So, for each thread-count and for each matrix size, we find the time taken, speed up, and efficiency by the formula:-

SpeedUp = Time Taken by 1 thread / Time taken by p threads

Efficiency = SpeedUp / p

CODE :

```
// 190001058
//  Somya Mehta
#include <bits/stdc++.h>
#include <ctime>
#include <omp.h>
using namespace std;


int numberOfRowsinA, numberOfColumnsinA;
int numOfThreads;
int numberOfRowsinB, numberOfColumnsinB;
// Initialize matrix with random values
void RandomlyInitialize(vector<vector<int>> &matrix) {
   srand(time(0));
   int row = matrix.size();
   int col = matrix[0].size();
```

```
    for (int i = 0; i < row; i++) {
        for (int j = 0; j < col; j++) {
            matrix[i][j] = rand() % 100;
            printf("%d\t", matrix[i][j]);
        }
        printf("\n");
    }
}

void matrixMult(vector<vector<int>> &matA, vector<vector<int>> &matB,
vector<vector<int>> &matC) {
#pragma omp parallel
    {
#pragma omp for collapse(2)
        for (int i = 0; i < numberOfRowsinA; i++) {
            for (int j = 0; j < numberOfColumnsinB; j++) {
                for (int k = 0; k < numberOfColumnsinA; k++) {
                    matC[i][j] += matA[i][k] * matB[k][j];
                }
            }
        }
    }
}
void OutputMatrix(vector<vector<int>> &matC) {
```

```
   printf("Output Matrix\n");
   for (int i = 0; i < numberOfRowsinA; i++) {
       for (int j = 0; j < numberOfColumnsinB; j++) {
           printf("%d \t", matC[i][j]);
       }
       printf("\n");
   }
}
int main() {
   printf("\n Enter dimensions of matrix A ");
   scanf("%d", &numberOfRowsinA);
   scanf("%d", &numberOfColumnsinA);

   vector<vector<int>> matA(numberOfRowsinA,
vector<int>(numberOfColumnsinA, 0));

   numberOfRowsinB = numberOfColumnsinA;
   printf("\n Enter dimensions numberof columns in matrix B ");
   scanf("%d", &numberOfColumnsinB);

   vector<vector<int>> matB(numberOfRowsinB,
vector<int>(numberOfColumnsinB, 0));
   // Randomly fill the matrices
   RandomlyInitialize(matB);
   RandomlyInitialize(matA);
```

```
    printf("\n Enter number of threads: ");
    scanf("%d", &numOfThreads);
    omp_set_num_threads(numOfThreads);

    vector<vector<int>> matC(numberOfRowsinA,
vector<int>(numberOfColumnsinB, 0));
    auto start = std::chrono::high_resolution_clock::now();
    matrixMult(matA, matB, matC);
    auto end = std::chrono::high_resolution_clock::now();
    auto duration =
std::chrono::duration_cast<std::chrono::microseconds>(end - start);
    int timeTaken = duration.count();
    OutputMatrix(matC);
    printf("\n Time taken to multiply %d X %d matrix with %d X %d matrix
using %d threads = %d microseconds\n", numberOfRowsinA,
numberOfColumnsinA, numberOfRowsinB, numberOfColumnsinB, numOfThreads,
timeTaken);
    return 0;
}
```

Data Obtained

Time Taken (in Microseconds)

	100x100 size	1000x1000 size	2000x2000 size
1 Thread	14300	24568124	275040044

2 Threads	11705	11890829	166470332
4 Threads	5816	8619521	111113505
8 Threads	5470	8200821	109881098

Speed Up

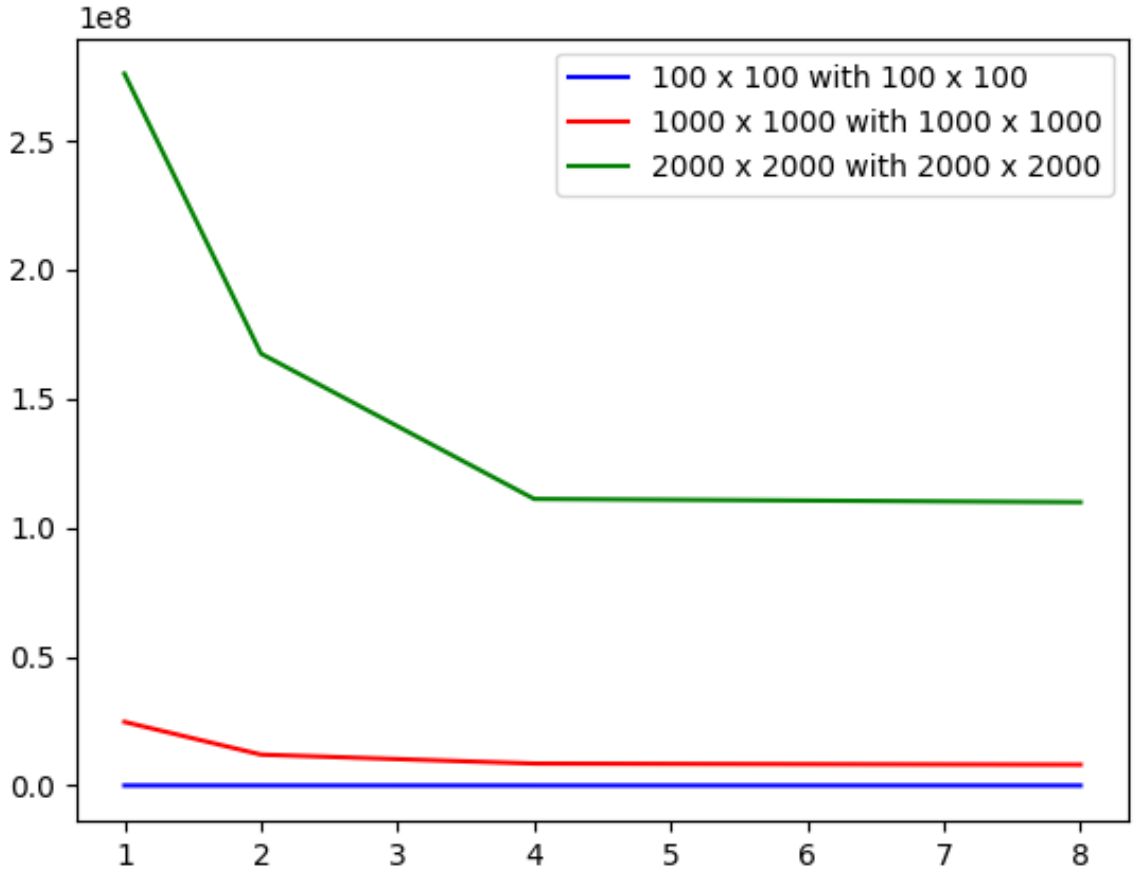
	100x100 size	**1000x1000 size**	**2000x2000 size**
2 Threads	1.58	1.89	1.89
4 Threads	2.91	3.95	3.88
8 Threads	3.74	6.49	6.63

Efficiency

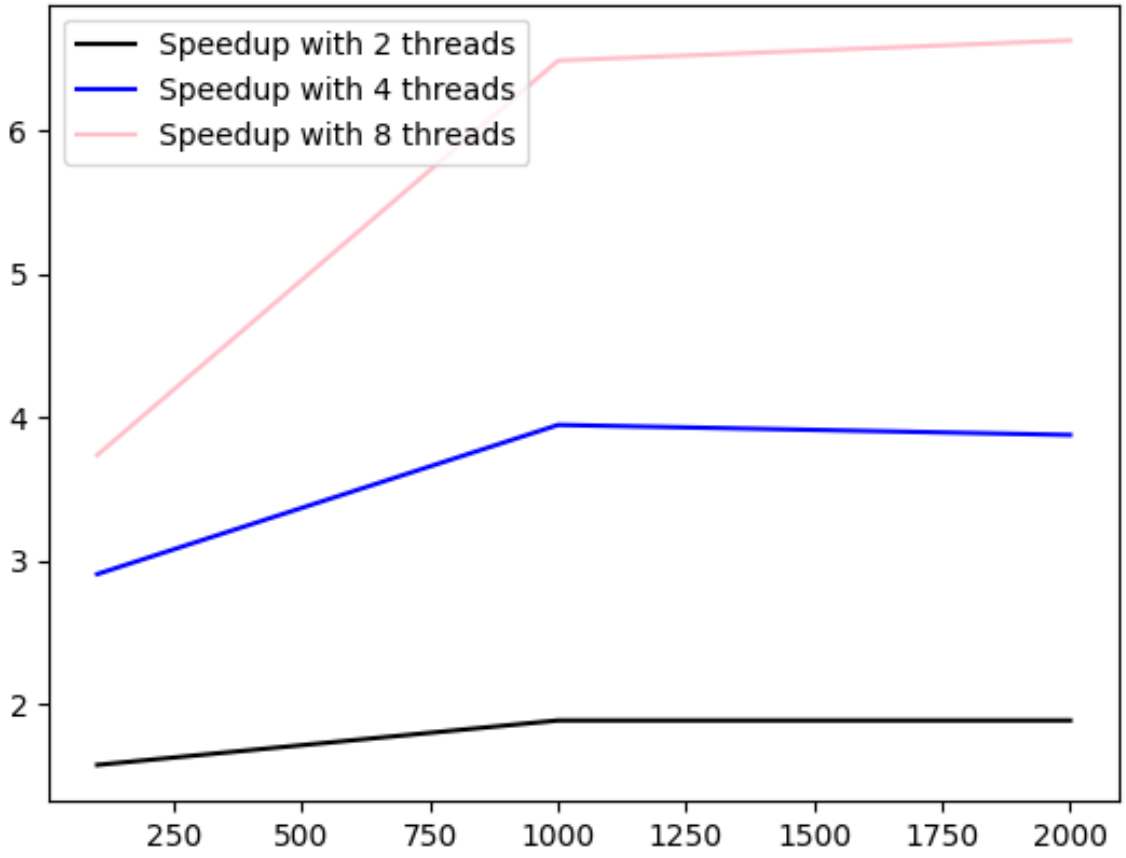
	100x100 size	**1000x1000 size**	**2000x2000 size**
2 Threads	1.58/2	1.89/2	1.89/2
4 Threads	2.91/4	3.95/4	3.88/4
8 Threads	3.74/8	6.49/8	6.63/8

Plots Obtained

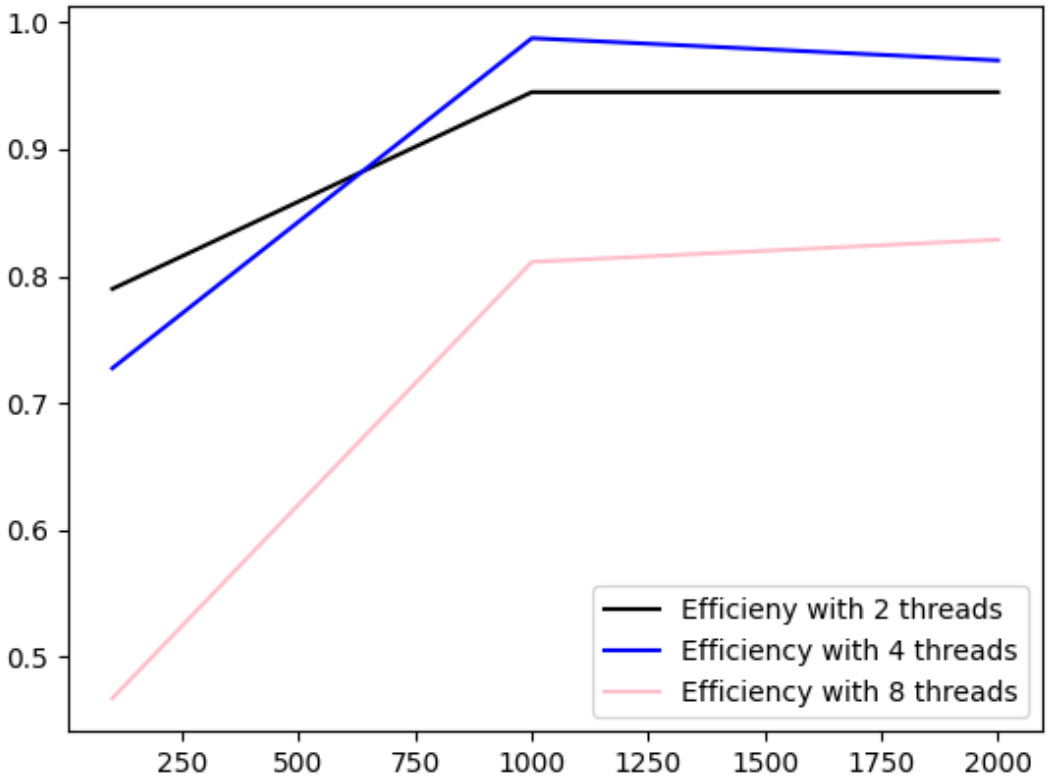
Time Taken



Speed Up


Speedup with 2 threads
Speedup with 4 threads
Speedup with 8 threads
6
5
4
3
2
250
500
750
1000
1250
1500
1750
2000

Efficiency



Observation from Graphs:-

As we can see from the graph of efficiency plotted that as the number of threads is increased, the efficiency obtained decreases.

But this is not in accordance with the expected result because we expect the efficiency to grow as the number of threads increases. The reason can be attributed to the fact that increasing the number of threads increases the number of context switches that occur, which increases the time taken, hence decreasing efficiency.