HPC Assignment 3
OpenMP Report

Recommendation System
(Collaborative Filtering)

Source code and output files can be found here:
[ced18i042-project-openmp]

Roll No: CED18I042
Name: Reuben Skariah Mathew

Date: 23rd September, 2021

Hardware Configuration:

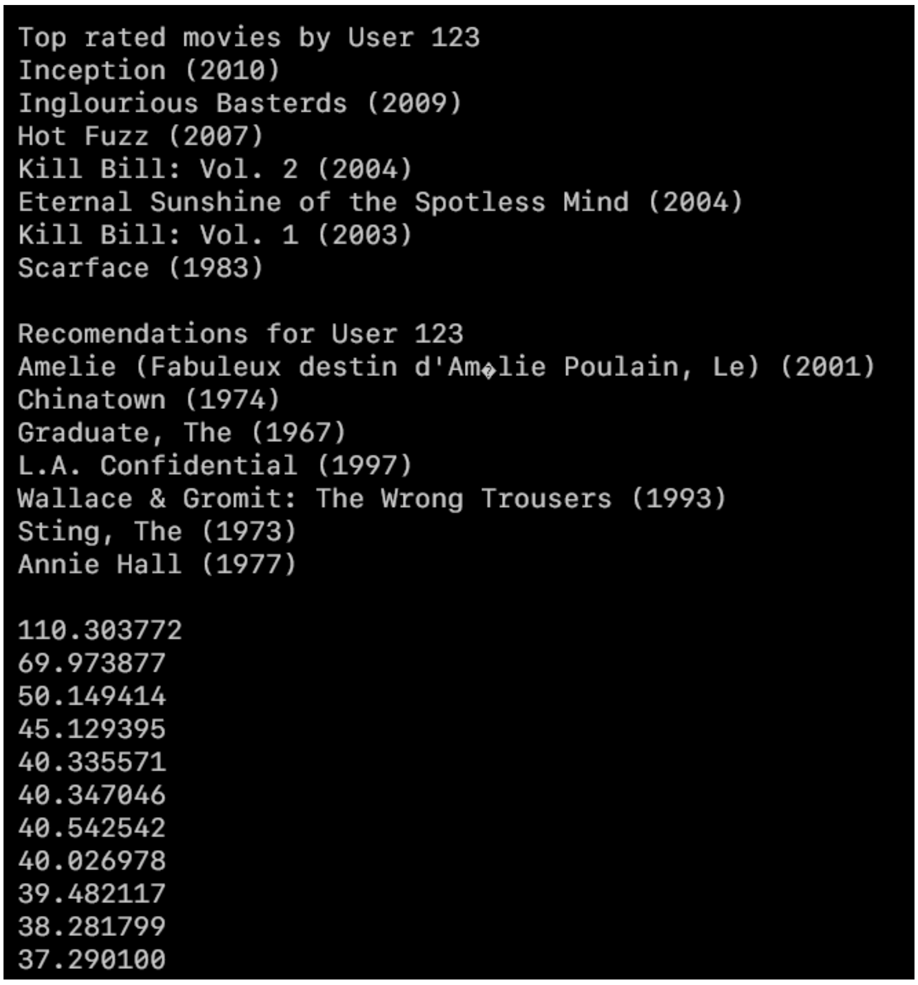
Processor: Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz
Sockets: 1
Cores per Socket: 4
Threads per Core: 2
L1 Cache: 32 kB
L2 Cache: 256 kB
L3 Cache: 6 MB
RAM: 8 GB

OpenMP Code:

```
#include <omp.h>
#include <vector>
#include <queue>
#include <string>
#include <cmath>
#include <vector>
#include <iostream>
#include <fstream>
#include <assert.h>
#include <functional>

using namespace std;

vector<string> moviesList;

void topRatings(vector<vector<double>> ratingsMat, int user)
{
    priority_queue<pair<double, int>> q;
    for (int i = 0; i < ratingsMat[user].size(); ++i)
    {
        q.push(pair<double, int>(ratingsMat[user][i], i));
    }
    int k = 7; // number of movies to be shown
    cout << "\nTop rated movies by User " << user << endl;
    for (int i = 0; i < k; ++i)
    {
        int ki = q.top().second;
        printf("%s\n", moviesList[ki].c_str());
        q.pop();
    }
}

void makeRec(vector<vector<double>> predict, int user)
{
    priority_queue<pair<double, int>> q;
    for (int i = 0; i < predict[user].size(); ++i)
    {
        q.push(pair<double, int>(predict[user][i], i));
    }
    int k = 7; // number of recomendations to be shown
    cout << "\nRecomendations for User " << user << endl;
    for (int i = 0; i < k; ++i)
    {
        int ki = q.top().second;
        printf("%s\n", moviesList[ki].c_str());

        q.pop();
    }
}

vector<vector<double>> matRead(string file, int row, int col)
{
    ifstream input(file);
    if (!input.is_open())
    {
        cerr << "File is not existing, check the path: \n"
             << file << endl;
        exit(1);
    }
    vector<vector<double>> data(row, vector<double>(col, 0));
    for (int i = 0; i < row; ++i)
    {
        for (int j = 0; j < col; ++j)
        {
            input >> data[i][j];
```

```cpp
        }
    }
    return data;
}

vector<string> movieRead(string file)
{
    vector<string> movies;
    ifstream input(file);
    if (!input.is_open())
    {
        cerr << "File is not existing, check the path: \n"
             << file << endl;
        exit(1);
    }
    string str;
    while (getline(input, str))
    {
        if (str.size() > 0)
            movies.push_back(str);
    }
    return movies;
}

void matWrite(vector<vector<double>> mat, string file)
{
    ofstream output(file);
    int row = mat.size();
    int col = mat[0].size();

    for (int i = 0; i < row; i++)
    {
        for (int j = 0; j < col; j++)
            output << mat[i][j] << " ";
        output << endl;
    }
}

double norm(vector<double> A)
{
    double res = 0;
#pragma omp parallel for reduction(+ : res)
    for (int i = 0; i < A.size(); ++i)
    {
        res += pow(A[i], 2);
    }
    return sqrt(res);
}

double dotProduct(vector<double> A, vector<double> B)
{
    double res = 0;
#pragma omp parallel for reduction(+ : res)
        for (int i = 0; i < A.size(); ++i)
        {
            res += A[i] * B[i];
        }
    return res;
}

double adjCosineSimilarity(vector<double> A, vector<double> B) //adjusted cosine
similarity (cosine similarity - mean)
{
    double A_mean = 0;
    double B_mean = 0;

//#pragma omp parallel for private(i) shared(A_mean, B_mean, A, B)
#pragma omp parallel for reduction(+ : A_mean, B_mean)
    for (int i = 0; i < A.size(); ++i)
    {
        A_mean += A[i];
        B_mean += B[i];
    }
    A_mean /= A.size();
    B_mean /= B.size();


    vector<double> C(A);
    vector<double> D(B);
    //#pragma omp parallel for private(i) shared(A, B, C, D, A_mean, B_mean)
#pragma omp parallel for //reduction(+ : A_mean, B_mean)
    for (int j = 0; j < A.size(); ++j)
    {
        C[j] = A[j] - A_mean;
        D[j] = B[j] - B_mean;
    }
    return dotProduct(C, D) / (norm(C) * norm(D)); //if output is nan then there
is no correlation
}

void checkCommon(vector<double> A, vector<double> B, vector<double> &C,
vector<double> &D) //to check if both A and B have rated
{

//#pragma omp parallel
//   {
//#pragma omp for
        for (int i = 0; i < A.size(); ++i)
        {
            if (A[i] && B[i])
            {
//#pragma omp critical
//             {
                    C.push_back(A[i]);
                    D.push_back(B[i]);
                }
            }
//        }
//    }
}

vector<vector<double>> colabFilter(vector<vector<double>> ratingsMat, int
usersNum, int itemsNum)
{
    vector<vector<double>> predict(usersNum, vector<double>(itemsNum, 0));

    #pragma omp parallel for collapse (2)
    for (int i = 0; i < usersNum; i++) //Make predictions for each user
    {
        for (int j = 0; j < itemsNum; j++) //Find item j that user i has not
scored, and predict user i's score for item j
        {
            if (ratingsMat[i][j]) //if movie has already been rated by the user
                continue;
            else //If item j has not been rated by user i, find out users who
have rated item j
            {
                vector<double> cosSim;
                vector<double> ratingsOld;
                for (int k = 0; k < usersNum; k++) //If user k has rated item j,
calculate the cosSimilarity between user k and user i
                {
                    if (ratingsMat[k][j]) //Find user k who has rated item j
                    {
                        vector<double> commonA, commonB;
//  Store the scores of the two items that have been jointly rated in two vectors
respectively
                        checkCommon(ratingsMat[i], ratingsMat[k], commonA,
commonB); //  check if item has been rated by both users
                        if (!commonA.empty())
//If the two have jointly rated items, calculate the cosine similarity
                        {
                            cosSim.push_back(adjCosineSimilarity(commonA,
commonB)); //cosine similarity
                            ratingsOld.push_back(ratingsMat[k][j]);
//old ratings
                        }
                    }
                }
                double cosSimSum = 0; //dot product of ratingsOld and cosSim
                if (!cosSim.empty())
                {
                    for (int m = 0; m < cosSim.size(); m++)
                    {
                        cosSimSum += cosSim[m];
                    }
                    predict[i][j] = dotProduct(cosSim, ratingsOld) / (cosSimSum);
                    cout << "user " << i << " item " << j << " with predicted
rating " << predict[i][j] << endl;
                }
            }
        }
    }
    return predict;
}

int main()
{
    string file1("ratings.txt");
    string file2("movies.txt");

    int row = 268;
    int col = 450;
    float startTime, endTime, runTime[4];
    vector<vector<double>> ratingsMat = matRead(file1, row, col);
    moviesList = movieRead(file2);

    int threads[] = {1, 2, 4, 6, 8, 10, 12, 16, 20, 32, 64, 128};
    for(int t = 0; t < 12; t++)
    {
        omp_set_num_threads(threads[t]);
        startTime = omp_get_wtime();
    vector<vector<double>> predict = colabFilter(ratingsMat, row, col);
    matWrite(predict, "predict.txt");

    //int uid = 123;
    topRatings(ratingsMat, 123);
    makeRec(predict, 123);

    endTime = omp_get_wtime();
    runTime[t] = endTime - startTime;
    }

    for(int t = 0; t < 12; t++)
        printf("\n%f",runTime[t]);
    return 0;
}
```

Output:



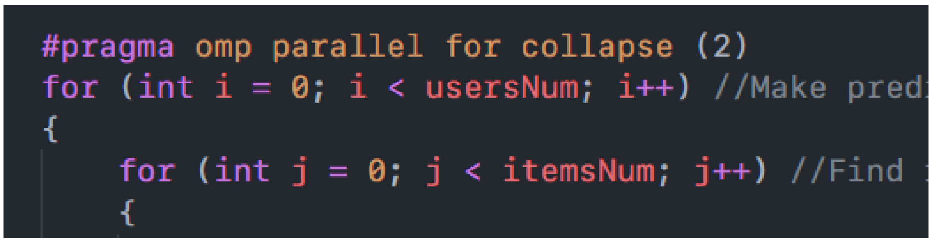
```
Top rated movies by User 123
Inception (2010)
Inglourious Basterds (2009)
Hot Fuzz (2007)
Kill Bill: Vol. 2 (2004)
Eternal Sunshine of the Spotless Mind (2004)
Kill Bill: Vol. 1 (2003)
Scarface (1983)

Recomendations for User 123
Amelie (Fabuleux destin d'Am�lie Poulain, Le) (2001)
Chinatown (1974)
Graduate, The (1967)
L.A. Confidential (1997)
Wallace & Gromit: The Wrong Trousers (1993)
Sting, The (1973)
Annie Hall (1977)

110.303772
69.973877
50.149414
45.129395
40.335571
40.347046
40.542542
40.026978
39.482117
38.281799
37.290100
```

***The last section of the output contains the runtimes for thread count 1, 2, 4, 6, 8, 10, 12, 16, 20, 32 and 64.**

Approach:

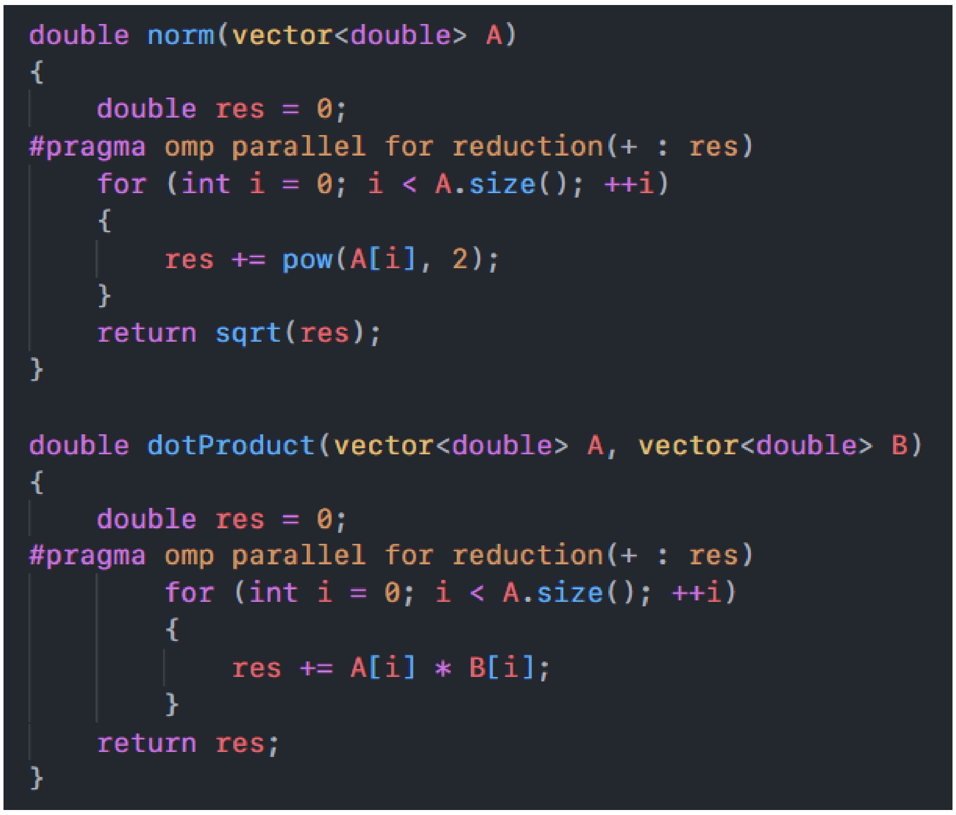
By profiling the serial code we were able to identify potential hotspots. Particularly, these were the function calls for checkCommon and adjCosineSimilarity. The norm() and dotProduct() function were also called a large number of times.

Initially parallelization was implemented in the driver function: colabFilter. As checkCommon() and adjCosineSimilarity() were called from here. This function iterates through the 2D matrix using a nested for loop.



```
#pragma omp parallel for collapse (2)
for (int i = 0; i < usersNum; i++) //Make pred
{
    for (int j = 0; j < itemsNum; j++) //Find
    {
```

This change alone provided significant speedup. Following this, the norm() and dotProduct() functions were parallelized using reduction. The significant operation here was vector addition.

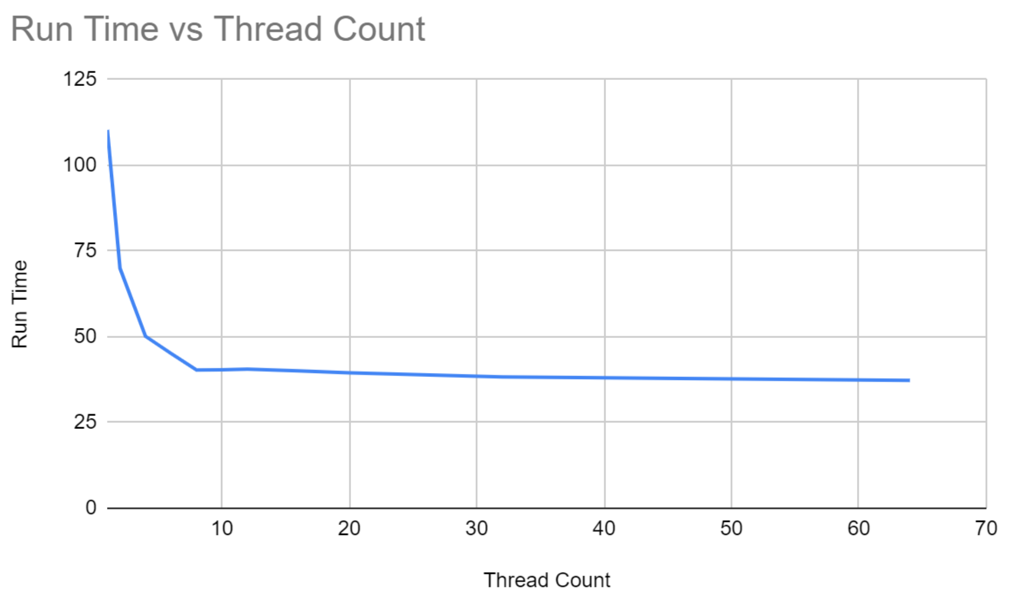


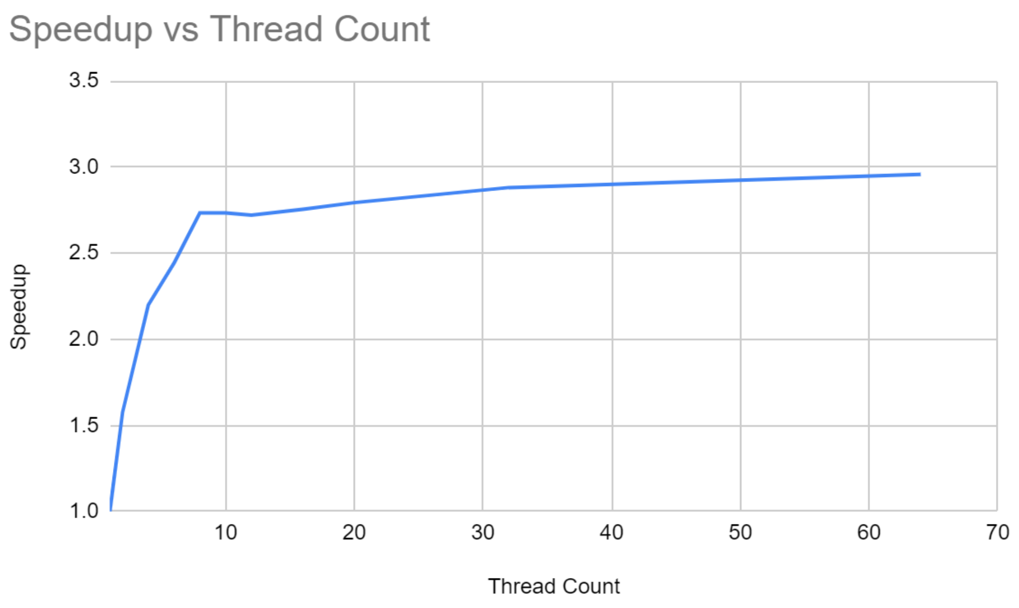
```
double norm(vector<double> A)
{
    double res = 0;
#pragma omp parallel for reduction(+ : res)
    for (int i = 0; i < A.size(); ++i)
    {
        res += pow(A[i], 2);
    }
    return sqrt(res);
}

double dotProduct(vector<double> A, vector<double> B)
{
    double res = 0;
#pragma omp parallel for reduction(+ : res)
        for (int i = 0; i < A.size(); ++i)
        {
            res += A[i] * B[i];
        }
    return res;
}
```

An attempt was made to parallelize the adjCosineSimilarity() and checkCommon(). But on trial running the code, the runtime had increased and speedup deteriorated. This may have been due to the communication overhead and critical section added (to keep <vector> functions thread safe). The lowest runtime was observed when the above three parallelizations were done: in colabFilter(), norm() and dotProduct(). The method used to parallelize checkCommon() and adjCosineSimilarity() are shown as comments in the code.

Analysis:

Number of Threads	Execution Time	Speed-Up	Parallelization Factor
1	110.303772	1	
2	69.973877	1.576356445	73.12514208
4	50.149414	2.19950271	72.71357018
6	45.129395	2.444166867	70.90351579
8	40.335561	2.734653226	72.49404827
10	40.347046	2.733874792	70.46875564
12	40.542542	2.720692057	68.99415915
16	40.026978	2.755735694	67.95951964
20	39.482117	2.79376539	67.58527762
32	38.281799	2.881363334	67.40047154
64	37.2901	2.95799078	67.24395533





Observations:

We observe that the runtime decreases upto thread count = 8, and then tapers off. The maximum threading supported by my system is 8. The speedup obtained due to threads is around 2.7. This greatly reduces the runtime as the serial program takes around 2 minutes to run and generate recommendations. It was also observed that critical sections deteriorated the speedup (which were required in the functions containing vector.push_back()).