HPC Project Report

# Recommendation System

(Collaborative Filtering)

**Roll No:** CED18I042
**Name:** Reuben Skariah Mathew

**Date:** 4th December, 2021

# Recommendation System

## (k-NN Based Collaborative Filtering)

**CED18I042**
**Reuben Skariah Mathew**

# Introduction

A **recommendation system**, or recommender system, is a subclass of information filtering system that seeks to **predict the "rating" or "preference" a user would give to an item**.

Recommender systems are used in a variety of areas, with commonly recognised examples taking the form of **playlist generators for video and music services, product recommenders for online stores, or content recommenders for social media platforms** and open web content recommenders.

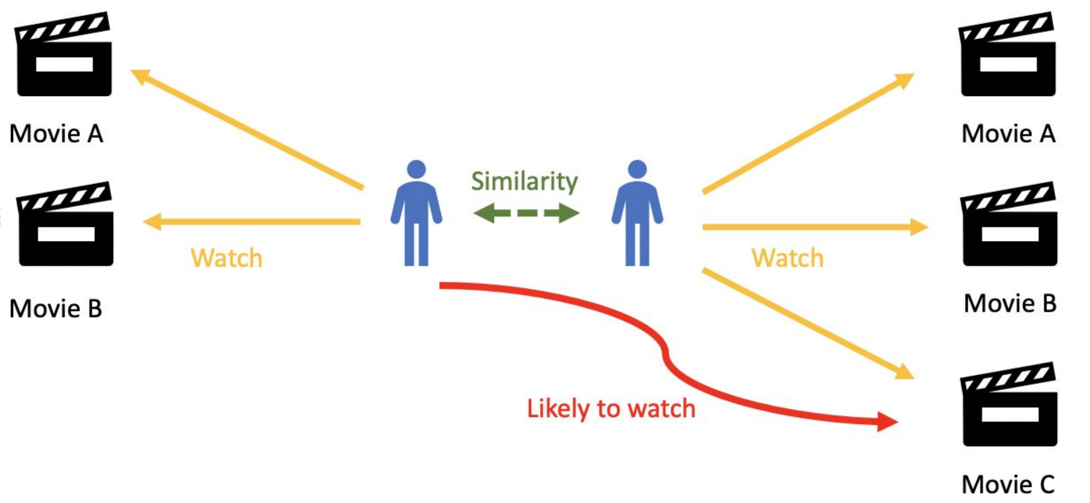
Most internet products we use today are powered by recommender systems. Youtube, Netflix, Amazon, Pinterest, and a long list of other internet products all rely on recommender systems to **filter millions of contents and make personalized recommendations to their users**.

These systems can operate using a single input, like music, or multiple inputs within and across platforms like news, books, and search queries.

# Collaborative Filtering

**Collaborative filtering** approach builds a model from a **user's past behaviors** (items previously purchased or selected and/or numerical ratings given to those items) as well as **similar decisions made by other users**. This model is then used to predict items (or ratings for items) that the user may have an interest in.

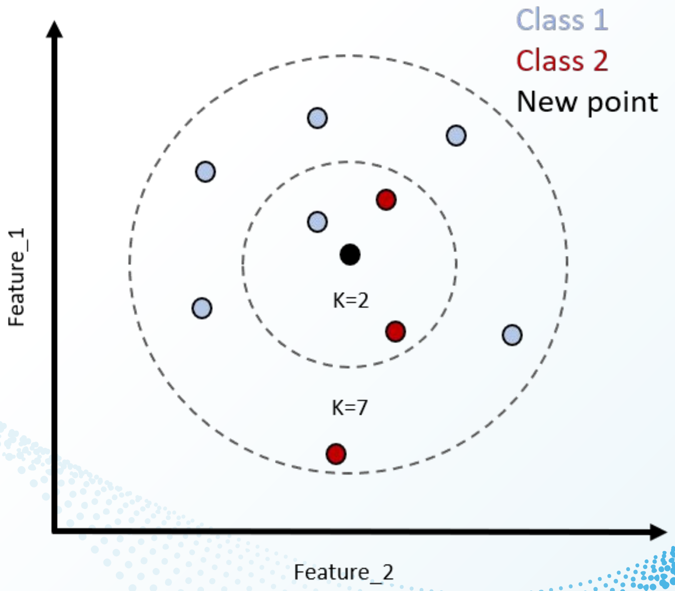
Collaborative filtering systems use the actions of users to recommend other movies. In general, they can either be user-based or item-based. **Item based** approach is usually preferred over user-based approach. To implement an item based collaborative filtering, **KNN** is a perfect go-to model and also a very good baseline for recommender system development.



# k-Nearest Neighbours (KNN)

The **k-nearest neighbors (k-NN)** algorithm is a **supervised machine learning algorithm** that can be used to solve both classification and regression problems. This algorithm is **non-parametric** in nature, hence it does not make any underlying assumptions about the distribution of data.

The k-NN algorithm is considered as one of the simplest machine learning algorithms. However, it is **computationally expensive** especially when the size of the training set becomes large which would cause the classification task to become very slow.



# Advantages

**1. No Training Period**: KNN is called Lazy Learner (Instance based learning). It stores the training dataset and learns from it only at the time of making real time predictions. This makes the KNN algorithm much faster than other algorithms that require training e.g. SVM, Linear Regression etc.

**2.** Since the KNN algorithm requires no training before making predictions, **new data can be added seamlessly** which will not impact the accuracy of the algorithm.

**3. KNN is easy to implement:** There are only two parameters required to implement KNN i.e. the value of K and the distance function (e.g. Euclidean or Manhattan etc.)

# Disadvantages

**1. Does not work well with large dataset:** In large datasets, the cost of calculating the distance between the new point and each existing points is huge which degrades the performance of the algorithm.

**2. Does not work well with high dimensions:** The KNN algorithm doesn't work well with high dimensional data because with large number of dimensions, it becomes difficult for the algorithm to calculate the distance in each dimension.

**3. Need feature scaling:** We need to do feature scaling (standardization and normalization) before applying KNN algorithm to any dataset. If we don't do so, KNN may generate wrong predictions.

# Working

K-nearest neighbors (k-NN) algorithm uses **'feature similarity'** to predict the values of new data points which further means that the new data point will be assigned a value based on how closely it matches the points in the training set.

Firstly datasets are loaded. This will contain the list of movies (movie ID), list of users (user ID) and their ratings for particular movies.

For an input movie (for which recommendations will be provided):

1. The data is converted to a matrix (users and their ratings for each movie) that can be given as input to the algorithm.
2. A distance value (Cosine Similarity) between each matrix entry to every other item is calculated and stored in a prediction matrix.
3. The k matrix entries for a particular user (row), with highest similarity will be given as recommendations.

# Code Balance

The number of floating point operations (FLOPs) can be found through line based profiling. In the output file we can see the number of times each line is executed/called. By adding the number of times lines with FLOPs is called we can estimate the number of FLOPs as well as number of words.A large number of FLOPs are found in the functions such as: norm, dotProduct, adjCosineSimilarity and colabFilter.

**Total Flops: 2843799997 = 2.843799997 gigaflops**
**Total Words: 4539698421**
*Calculation can be found in the report.

Code Balance = Words / FLOPs
4539698421 / 2843799997

**Code Balance = 1.59634940073**

# Recommendation System

(k-NN Based Collaborative Filtering)

HPC Report

**Roll No: CED18I042**
**Name: Reuben Skariah Mathew**

**Date: 8th September, 2021**

# Introduction

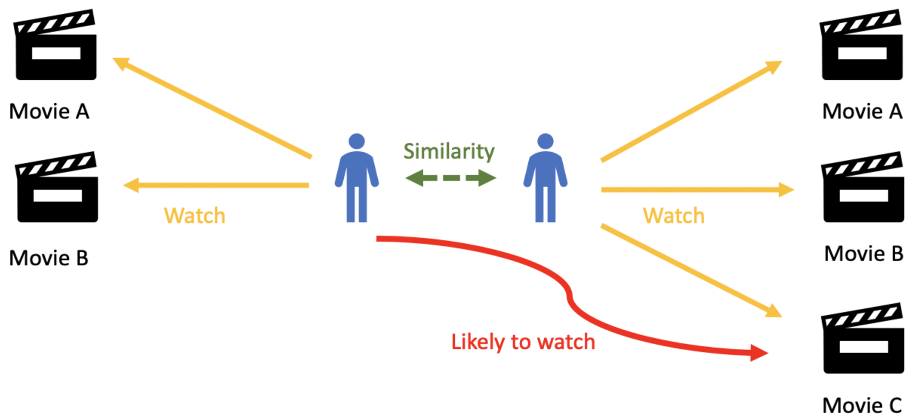
A **recommendation system**, or recommender system, is a subclass of information filtering system that seeks to **predict the "rating" or "preference" a user would give to an item**.

Recommender systems are used in a variety of areas, with commonly recognised examples taking the form of **playlist generators for video and music services, product recommenders for online stores, or content recommenders for social media platforms** and open web content recommenders.

Most internet products we use today are powered by recommender systems. Youtube, Netflix, Amazon, Pinterest, and a long list of other internet products all rely on recommender systems to **filter millions of contents and make personalized recommendations to their users**.

These systems can operate using a single input, like music, or multiple inputs within and across platforms like news, books, and search queries.

# Collaborative Filtering



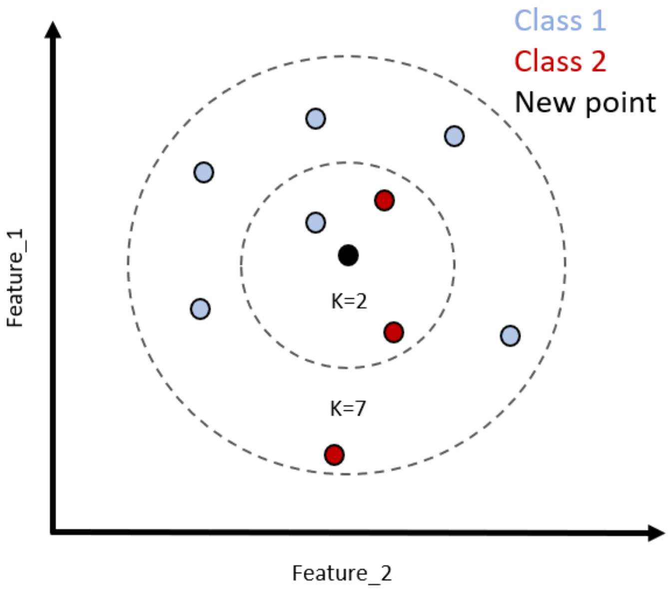

**Collaborative filtering** approach builds a model from a **user's past behaviors** (items previously purchased or selected and/or numerical ratings given to those items) as well as **similar decisions made by other users**. This model is then used to predict items (or ratings for items) that the user may have an interest in.

Collaborative filtering systems use the actions of users to recommend other movies. In general, they can either be user-based or item-based. **Item based** approach is usually preferred over user-based approach. To implement an item based collaborative filtering, **KNN** is a perfect go-to model and also a very good baseline for recommender system development.

# k-Nearest Neighbours (KNN)

The **k-nearest neighbors (k-NN)** algorithm is a **supervised machine learning algorithm** that can be used to solve both classification and regression problems. This algorithm is **non-parametric** in nature, hence it does not make any underlying assumptions about the distribution of data.

The k-NN algorithm is considered as one of the simplest machine learning algorithms. However, it is **computationally expensive** especially when the size of the training set becomes large which would cause the classification task to become very slow.




# Working

K-nearest neighbors (k-NN) algorithm uses **'feature similarity'** to predict the values of new data points which further means that the new data point will be assigned a value based on how closely it matches the points in the training set.

Firstly, datasets are loaded. This will contain the list of movies (movie ID), list of users (user ID) and their ratings for particular movies.

For an input movie (for which recommendations will be provided):

1. The data is converted to a matrix (users and their ratings for each movie) that can be given as input to the algorithm.
2. A distance value (Cosine Similarity) between each matrix entry to every other item is calculated and stored in a prediction matrix.
3. The k matrix entries for a particular user (row), with highest similarity will be given as recommendations.

# Code Balance

The number of floating point operations (FLOPs) can be found through line based profiling. In the output file we can see the number of times each line is executed/called. By adding the number of times lines with FLOPs is called we can estimate the number of FLOPs as well as number of words.

A large number of FLOPs are found in the functions such as: norm, dotProduct, adjCosineSimilarity and colabFilter.

**FLOPs Calculation:**

norm() - ( 529112946 * 2 = 1058225892 ) + ( 11993022 * 6 = 71958132 )

dotProduct() - ( 270552984 * 2 = 541105968 ) + ( 11993022 * 5 = 59965110 )

adjCosineSimilarity() - ( 264556473 * 4 = 1058225892 ) + ( 5996511 * 8 = 47972088 ) = 1106197980

colabFilter() - ( 5996511 * 1 ) + ( 87601 * 4 = 350404 ) = 6346915

**Total Flops: 2843799997 = 2.843799997 gigaflops**

**Word Count Calculation:**

norm() - 529112946 * 2 = 1058225892

dotProduct() - 270552984 * 3 = 811658952

adjCosineSimilarity() - ( 264556473 * 10 = 2645564730 ) + ( 5996511 * 2 = 11993022 ) = 2657557752

colabFilter() - ( 5996511 * 2 ) + ( 87601 * 3 ) = 12255825

**Total Words: 4539698421**

Code Balance = Words / FLOPs

4539698421 / 2843799997

**Code Balance = 1.59634940073**

# HPC Assignment 2
# Profiling Report

# Recommendation System
## (Collaborative Filtering)

Source code and output files can be found here:
**ced18i042-recommendation-system-1**

**Roll No:** CED18I042
**Name:** Reuben Skariah Mathew

**Date:** 20th 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

# Serial Code:

```
#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)
{
    std::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 = 5; // 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)
{
```

```
    std::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 = 5; // number of recommendations 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];
        }
    }
    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;
    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;
    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;
    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);
    for (int i = 0; i < A.size(); ++i)
    {
        C[i] = A[i] - A_mean;
        D[i] = B[i] - 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
{
    for (int i = 0; i < A.size(); ++i)
    {
        if (A[i] && B[i])
        {
            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));
    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;
    vector<vector<double>> ratingsMat = matRead(file1, row, col);
    moviesList = movieRead(file2);
    vector<vector<double>> predict = colabFilter(ratingsMat, row, col);
    matWrite(predict, "predict.txt");

    int uid, check;
    do
    {
        cout << "\nEnter User ID:" << endl;
        cin >> uid;
        topRatings(ratingsMat, uid);
        makeRec(predict, uid);
        cout << "\nRecommend for another user? (1 = Yes, 0 = No)" << endl;
        cin >> check;
    } while (check == 1);

    return 0; }
```

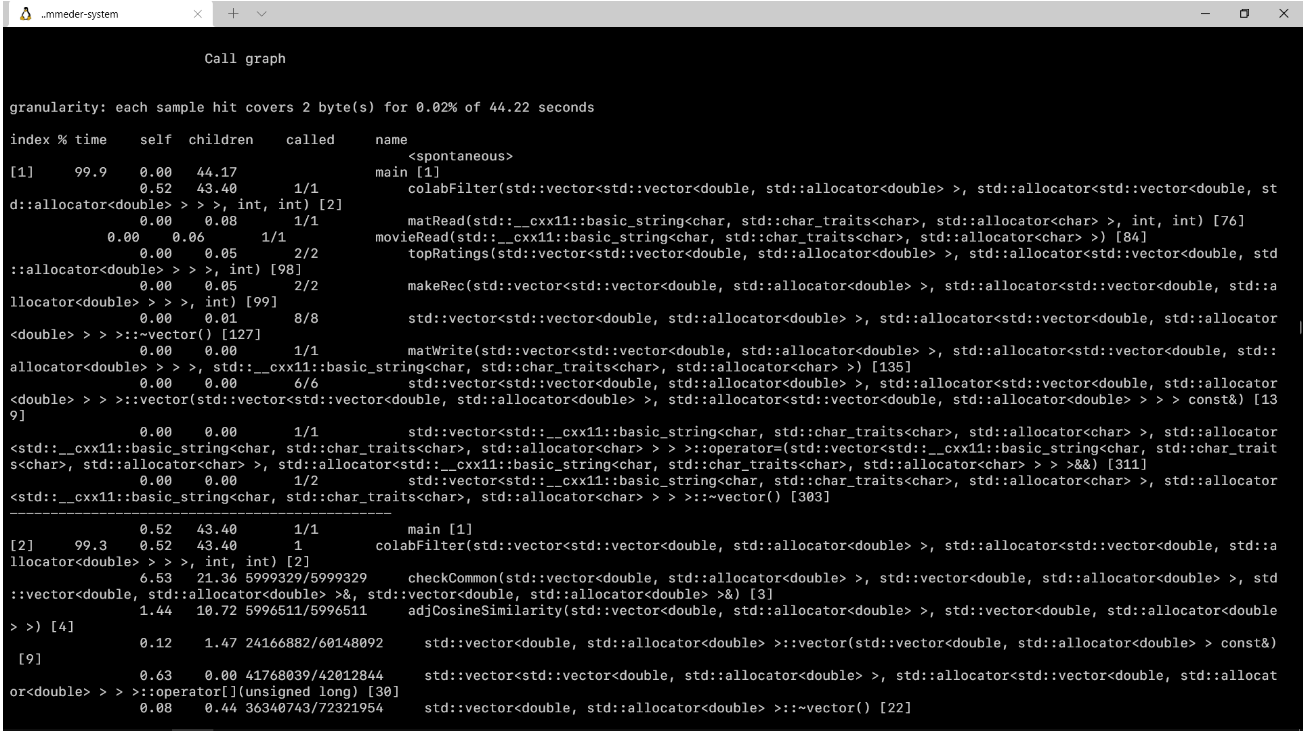
# Output:

```
Enter User ID:
123

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)

Recommend for another user? (1 = Yes, 0 = No)
0
→  recommeder-system
```

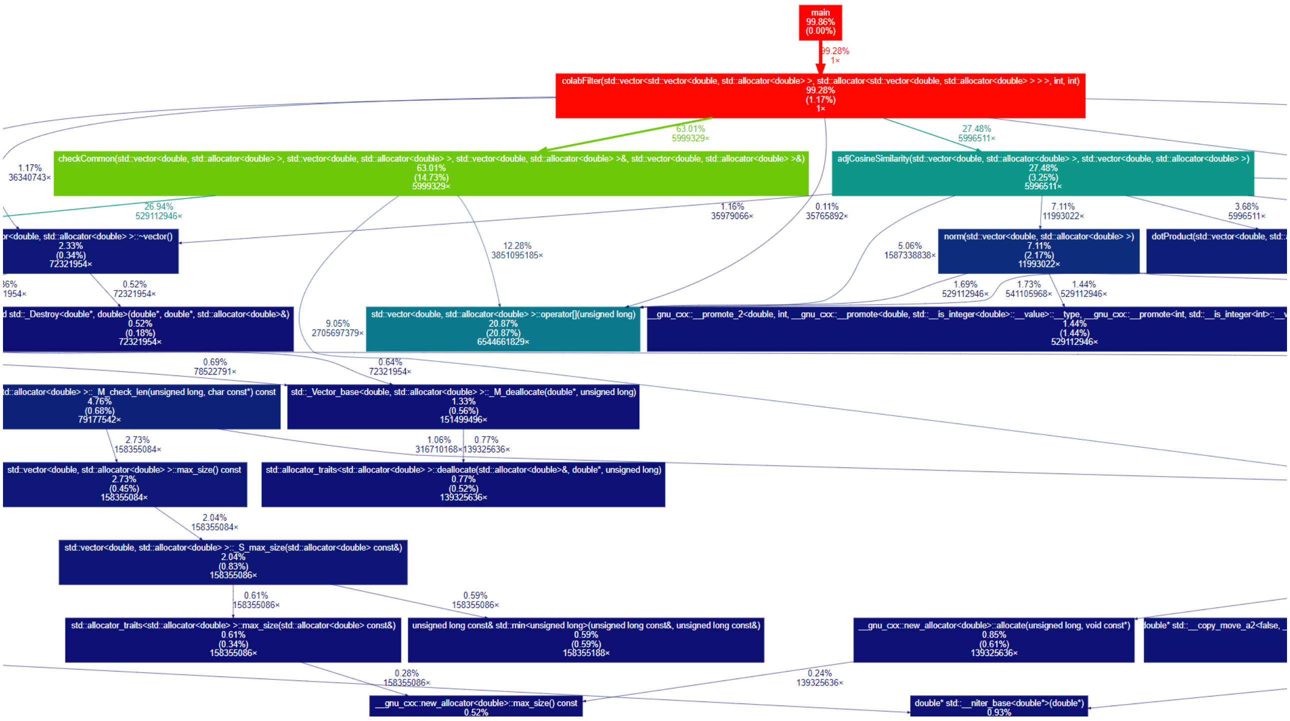
# Profiling

- ## Functional Profiling

```
..mmeder-system
→  recommeder-system gprof rec > rec.gprof
→  recommeder-system python3 gprof2dot.py < rec.gprof | dot -Tsvg -o gprof-output.svg
→  recommeder-system gprof -b rec
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  s/call   s/call  name
 20.94      9.25     9.25 6544661829     0.00     0.00  std::vector<double, std::allocator<double> >::operator[](unsigned long)
 14.95     15.86     6.61 4459483610     0.00     0.00  std::vector<double, std::allocator<double> >::size() const
 14.78     22.39     6.53  5999329     0.00     0.00  checkCommon(std::vector<double, std::allocator<double> >, std::vector<double, std::allocator<double> >
, std::vector<double, std::allocator<double> >&, std::vector<double, std::allocator<double> >&)
  3.26     23.83     1.44  5996511     0.00     0.00  adjCosineSimilarity(std::vector<double, std::allocator<double> >, std::vector<double, std::allocator<d
ouble> >)
  2.69     25.02     1.19 535109457     0.00     0.00  std::vector<double, std::allocator<double> >::push_back(double const&)
  2.57     26.16     1.14 535109457     0.00     0.00  void __gnu_cxx::new_allocator<double>::construct<double, double const&>(double*, double const&)
  2.35     27.20     1.04 535109457     0.00     0.00  void std::allocator_traits<std::allocator<double> >::construct<double, double const&>(std::allocator<
double>&, double*, double const&)
  2.23     28.19     0.99 78522791     0.00     0.00  void std::vector<double, std::allocator<double> >::_M_realloc_insert<double const&>(__gnu_cxx::__norma
l_iterator<double*, std::vector<double, std::allocator<double> > >, double const&)
  2.17     29.15     0.96 11993022     0.00     0.00  norm(std::vector<double, std::allocator<double> >)
  2.06     30.06     0.91 1148741705     0.00     0.00  double const& std::forward<double const&>(std::remove_reference<double const&>::type&)
  1.45     30.70     0.64 529112946     0.00     0.00  __gnu_cxx::__promote_2<double, int, __gnu_cxx::__promote<double, std::__is_integer<double>::__value>:
:__type, __gnu_cxx::__promote<int, std::__is_integer<int>::__value>::__type>::__type std::pow<double, int>(double, int)
  1.44     31.33     0.64 42012844     0.00     0.00  std::vector<std::vector<double, std::allocator<double> >, std::allocator<std::vector<double, std::allo
cator<double> > > >::operator[](unsigned long)
  1.29     31.90     0.57 541112917     0.00     0.00  operator new(unsigned long, void*)
  1.22     32.44     0.54 158355084     0.00     0.00  double* std::__relocate_a<double*, double*, std::allocator<double> >(double*, double*, double*, std::
allocator<double>&)
  1.20     32.97     0.53 158355084     0.00     0.00  std::vector<double, std::allocator<double> >::_S_relocate(double*, double*, double*, std::allocator<d
ouble>&)
  1.18     33.49     0.52        1     0.52    43.92  colabFilter(std::vector<std::vector<double, std::allocator<double> >, std::allocator<std::vector<doubl
e, std::allocator<double> > > >, int, int)
  1.09     33.97     0.48  6084112     0.00     0.00  dotProduct(std::vector<double, std::allocator<double> >, std::vector<double, std::allocator<double> >)

  1.00     34.41     0.44 158355084     0.00     0.00  std::enable_if<std::__is_bitwise_relocatable<double, void>::value, double*>::type std::__relocate_a_1
<double, double>(double*, double*, double*, std::allocator<double>&)
  0.93     34.82     0.41 535213346     0.00     0.00  double* std::__niter_base<double*>(double*)
  0.84     35.19     0.37 158355086     0.00     0.00  std::vector<double, std::allocator<double> >::_S_max_size(std::allocator<double> const&)
  0.68     35.49     0.30 79177542     0.00     0.00  std::vector<double, std::allocator<double> >::_M_check_len(unsigned long, char const*) const
  0.66     35.78     0.29 60148092     0.00     0.00  std::vector<double, std::allocator<double> >::vector(std::vector<double, std::allocator<double> > cons
t&)
```



```
                        Call graph


granularity: each sample hit covers 2 byte(s) for 0.02% of 44.22 seconds

index % time    self  children    called     name
                                                 <spontaneous>
[1]     99.9    0.00   44.17                 main [1]
                0.52   43.40       1/1           colabFilter(std::vector<std::vector<double, std::allocator<double> >, std::allocator<std::vector<double, st
d::allocator<double> > > >, int, int) [2]
                0.00    0.08       1/1           matRead(std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> >, int, int) [76]
            0.00    0.06       1/1           movieRead(std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> >) [84]
                0.00    0.05       2/2           topRatings(std::vector<std::vector<double, std::allocator<double> >, std::allocator<std::vector<double, std
::allocator<double> > > >, int) [98]
                0.00    0.05       2/2           makeRec(std::vector<std::vector<double, std::allocator<double> >, std::allocator<std::vector<double, std::a
llocator<double> > > >, int) [99]
                0.00    0.01       8/8           std::vector<std::vector<double, std::allocator<double> >, std::allocator<std::vector<double, std::allocator
<double> > > >::~vector() [127]
                0.00    0.00       1/1           matWrite(std::vector<std::vector<double, std::allocator<double> >, std::allocator<std::vector<double, std::
allocator<double> > > >, std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> >) [135]
                0.00    0.00       6/6           std::vector<std::vector<double, std::allocator<double> >, std::allocator<std::vector<double, std::allocator
<double> > > >::vector(std::vector<std::vector<double, std::allocator<double> >, std::allocator<std::vector<double, std::allocator<double> > > > const&) [13
9]
                0.00    0.00       1/1           std::vector<std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> >, std::allocator
<std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> > > >::operator=(std::vector<std::__cxx11::basic_string<char, std::char_trait
s<char>, std::allocator<char> >, std::allocator<std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> > > >&&) [311]
                0.00    0.00       1/2           std::vector<std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> >, std::allocator
<std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> > > >::~vector() [303]
-----------------------------------------------
                0.52   43.40       1/1           main [1]
[2]     99.3    0.52   43.40       1         colabFilter(std::vector<std::vector<double, std::allocator<double> >, std::allocator<std::vector<double, std::a
llocator<double> > > >, int, int) [2]
                6.53   21.36 5999329/5999329     checkCommon(std::vector<double, std::allocator<double> >, std::vector<double, std::allocator<double> >, std
::vector<double, std::allocator<double> >&, std::vector<double, std::allocator<double> >&) [3]
                1.44   10.72 5996511/5996511     adjCosineSimilarity(std::vector<double, std::allocator<double> >, std::vector<double, std::allocator<double
> >) [4]
                0.12    1.47 24166882/60148092     std::vector<double, std::allocator<double> >::vector(std::vector<double, std::allocator<double> > const&)
 [9]
                0.63    0.00 41768039/42012844     std::vector<std::vector<double, std::allocator<double> >, std::allocator<std::vector<double, std::allocat
or<double> > > >::operator[](unsigned long) [30]
                0.08    0.44 36340743/72321954     std::vector<double, std::allocator<double> >::~vector() [22]
```



- # Line Based Profiling

```
→  recommeder-system gcov -b -c rec.cpp
File 'rec.cpp'
Lines executed:96.72% of 122
Branches executed:94.50% of 218
Taken at least once:58.72% of 218
Calls executed:82.76% of 232
Creating 'rec.cpp.gcov'

File '/usr/include/c++/9/iostream'
No executable lines
No branches
No calls
Removing 'iostream.gcov'

File '/usr/include/c++/9/bits/stl_iterator.h'
Lines executed:58.54% of 41
No branches
Calls executed:77.78% of 9
Creating 'stl_iterator.h.gcov'

File '/usr/include/c++/9/bits/stl_algobase.h'
Lines executed:97.06% of 34
Branches executed:100.00% of 6
Taken at least once:83.33% of 6
Calls executed:100.00% of 3
Creating 'stl_algobase.h.gcov'

File '/usr/include/c++/9/bits/cpp_type_traits.h'
Lines executed:100.00% of 2
No branches
No calls
Creating 'cpp_type_traits.h.gcov'

File '/usr/include/c++/9/ext/new_allocator.h'
Lines executed:93.75% of 16
No branches
No calls
Creating 'new_allocator.h.gcov'

File '/usr/include/c++/9/bits/stl_construct.h'
Lines executed:100.00% of 15
No branches
```

```
→  recommeder-system cat rec.cpp.gcov
        -:    0:Source:rec.cpp
        -:    0:Graph:rec.gcno
        -:    0:Data:rec.gcda
        -:    0:Runs:1
        -:    1:#include <vector>
        -:    2:#include <queue>
        -:    3:#include <string>
        -:    4:#include <cmath>
        -:    5:#include <vector>
        -:    6:#include <iostream>
        -:    7:#include <fstream>
        -:    8:#include <assert.h>
        -:    9:#include <functional>
        -:   10:
        -:   11:using namespace std;
        -:   12:
        -:   13:vector<string> moviesList;
        -:   14:
function _Z10topRatingsSt6vectorIS_IdSaIdEESaIS1_EEi called 2 returned 100% blocks executed 86%
        2:   15:void topRatings(vector<vector<double>> ratingsMat, int user)        -:   16:{
        4:   17:    std::priority_queue<std::pair<double, int>> q;
call    0 returned 2
call    1 returned 2
call    2 never executed
      902:   18:    for (int i = 0; i < ratingsMat[user].size(); ++i)
call    0 returned 902
call    1 returned 902
branch  2 taken 900 (fallthrough)
branch  3 taken 2
        -:   19:    {
      900:   20:        q.push(std::pair<double, int>(ratingsMat[user][i], i));
call    0 returned 900
call    1 returned 900
call    2 returned 900
call    3 returned 900
branch  4 taken 900 (fallthrough)
branch  5 taken 0 (throw)
        -:   21:    }
        2:   22:    int k = 7; // number of movies to be shown
        2:   23:    cout << "\nTop rated movies by User " << user << endl;
call    0 returned 2
```

- # Hardware Profiling

```
➜  recommeder-system likwid-topology
--------------------------------------------------------------------------------
CPU name:       Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz
CPU type:       Intel Kabylake processor
CPU stepping:   10
********************************************************************************
Hardware Thread Topology
********************************************************************************
Sockets:                1
Cores per socket:       4
Threads per core:       2
--------------------------------------------------------------------------------
HWThread        Thread          Core            Socket          Available
0               0               0               0               *
1               1               0               0               *
2               0               1               0               *
3               1               1               0               *
4               0               2               0               *
5               1               2               0               *
6               0               3               0               *
7               1               3               0               *
--------------------------------------------------------------------------------
Socket 0:               ( 0 1 2 3 4 5 6 7 )
--------------------------------------------------------------------------------
********************************************************************************
Cache Topology
********************************************************************************
Level:                  1
Size:                   32 kB
Cache groups:           ( 0 1 ) ( 2 3 ) ( 4 5 ) ( 6 7 )
--------------------------------------------------------------------------------
Level:                  2
Size:                   256 kB
Cache groups:           ( 0 1 ) ( 2 3 ) ( 4 5 ) ( 6 7 )
--------------------------------------------------------------------------------
Level:                  3
Size:                   6 MB
Cache groups:           ( 0 1 2 3 4 5 6 7 )
--------------------------------------------------------------------------------
********************************************************************************
NUMA Topology
********************************************************************************
```

```
➜  recommeder-system sudo likwid-perfctr -C S0:0 -g L3 -m rec
[sudo] password for ubuntu:
--------------------------------------------------------------------------------
CPU name:       Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz
CPU type:       Intel Kabylake processor
CPU clock:      1.80 GHz
ERROR - [./src/access_client.c:189] No such file or directory
Exiting due to timeout: The socket file at '/tmp/likwid-132' could not be
opened within 10 seconds. Consult the error message above
this to find out why. If the error is 'no such file or directoy',
it usually means that likwid-accessD just failed to start.
➜  recommeder-system sudo likwid-perfctr -g 0-3 -g L3 -m rec
--------------------------------------------------------------------------------
CPU name:       Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz
CPU type:       Intel Kabylake processor
CPU clock:      1.80 GHz
Warning: The Marker API requires the application to run on the selected CPUs.
Warning: likwid-perfctr pins the application only when using the -C command line option.
Warning: LIKWID assumes that the application does it before the first instrumented code region is started.
Warning: You can use the string in the environment variable LIKWID_THREADS to pin you application to
Warning: to the CPUs specified after the -c command line option.
ERROR - [./src/access_client.c:189] No such file or directory
Exiting due to timeout: The socket file at '/tmp/likwid-151' could not be
opened within 10 seconds. Consult the error message above
this to find out why. If the error is 'no such file or directoy',
it usually means that likwid-accessD just failed to start.
➜  recommeder-system sudo likwid-perfctr -c 0-3 -g L3 -m rec
--------------------------------------------------------------------------------
CPU name:       Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz
CPU type:       Intel Kabylake processor
CPU clock:      1.80 GHz
Warning: The Marker API requires the application to run on the selected CPUs.
Warning: likwid-perfctr pins the application only when using the -C command line option.
Warning: LIKWID assumes that the application does it before the first instrumented code region is started.
Warning: You can use the string in the environment variable LIKWID_THREADS to pin you application to
Warning: to the CPUs specified after the -c command line option.
ERROR - [./src/access_client.c:189] No such file or directory
Exiting due to timeout: The socket file at '/tmp/likwid-170' could not be
opened within 10 seconds. Consult the error message above
this to find out why. If the error is 'no such file or directoy',
it usually means that likwid-accessD just failed to start.
➜  recommeder-system
```

# Observations:

**Functional Profiling:**

- From functional profiling we see that colabFilter() is called once from main.
- This inturn leads to checkCommon being called 59,99,329 times, and adjCosineSimilarity being called 59,96,511 times. Other functions that are called a large number of times are norm and dotProduct with 1,19,93,022 and 60,84,112 calls respectively.
- The predefined function .push_back() to insert values into a vector is called 53,51,09,457 times.
- We also see that the checkCommon() function takes 14.73% of the total time, and that adjCosineSimilarity() takes 3.25% of the total time.
- Hence the functions: checkCommon and adjCosineSimilarity can been seen as hotspots.

**Line Based Profiling:**

- From line profiling we observe that 96.72 lines are executed, 94.50% branches are executed and 82.76% calls are executed.
- We see that lines 118, 123 - 140 have a large number of iterations.

**Hardware Profiling:**

- likwid-topology was used to view system information.
- But likwid-perfctr was not able to be used as it was not supported by my system (on Windows Subsystem for Linux)

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];
```

```
        }
    }
    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
    {
```

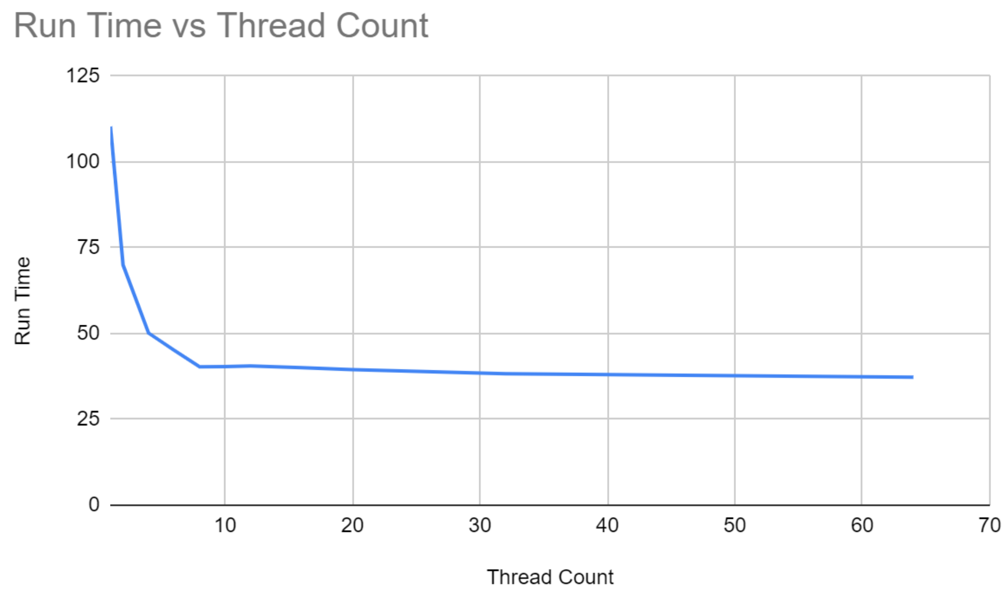
This change alone provided significant speedup. Following this, the norm() and dotProduct() functions were parallelized using reduction. The significant operation here was vector addition.

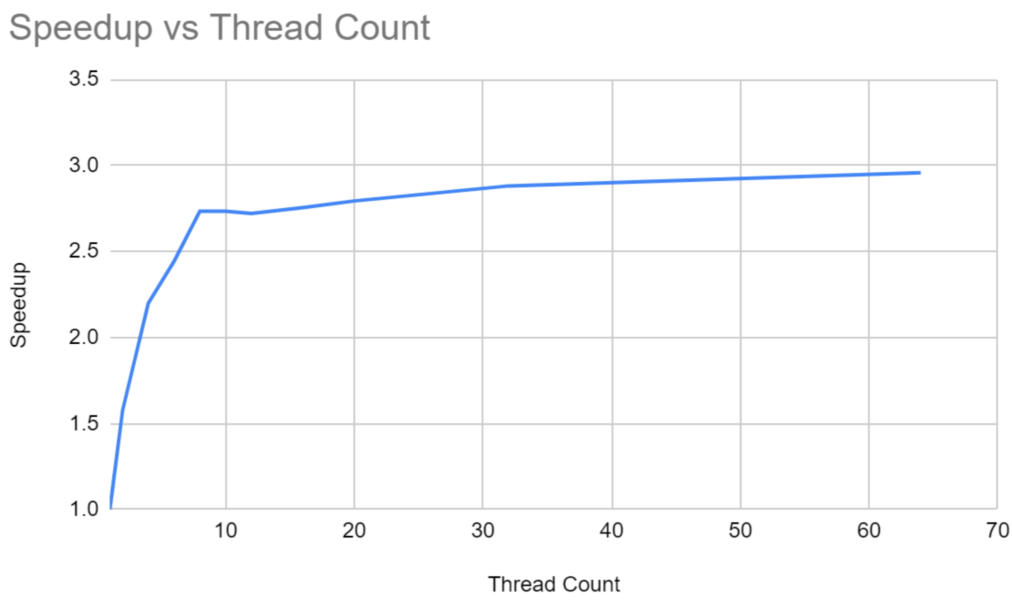
```
double norm(vector<double> A)
{
    double res = 0;
#pragma omp parallel for reduction(+ : res)
    for (int i = 0; i < A.size(); ++i)
    {
        res += pow(A[i], 2);
    }
    return sqrt(res);
}

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()).

HPC Project Part 1
MPI Report

# Recommendation System

(Collaborative Filtering)

Source code and output files can be found here:
**ced18i042-project-mpi**

**Roll No:** CED18I042
**Name:** Reuben Skariah Mathew

**Date:** 20th November, 2021

# Hardware Configuration (VM):

**Processor:** Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz
**Sockets:** 1
**Cores per Socket:** 1
**Threads per Core:** 1
**L1 Cache:** 32 kB
**L2 Cache:** 256 kB
**L3 Cache:** 6 MB
**RAM:** 981.262 MB

# MPI Code:

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

#define NR 268
#define NC 450
#define MASTER 0
#define FROM_MASTER 1
#define FROM_WORKER 2

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];
        }
    }
    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;
    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;
    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) //5996511
{
    double A_mean = 0;
    double B_mean = 0;
    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);
    for (int i = 0; i < A.size(); ++i)
    {
```

```
        C[i] = A[i] - A_mean;
        D[i] = B[i] - B_mean;
    }
    return dotProduct(C, D) / (norm(C) * norm(D)); //if output is nan then there
is no correlation //11993022
}

void checkCommon(vector<double> A, vector<double> B, vector<double> &C,
vector<double> &D) //to check if both A and B have rated //5999329
{
    for (int i = 0; i < A.size(); ++i) //2705697379
    {
        if (A[i] && B[i])
        {
            C.push_back(A[i]);
            D.push_back(B[i]);
        }
    }
}

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

    if (rows + offset <= 268)
    {
        for (int i = offset; i < rows + offset; 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;
```

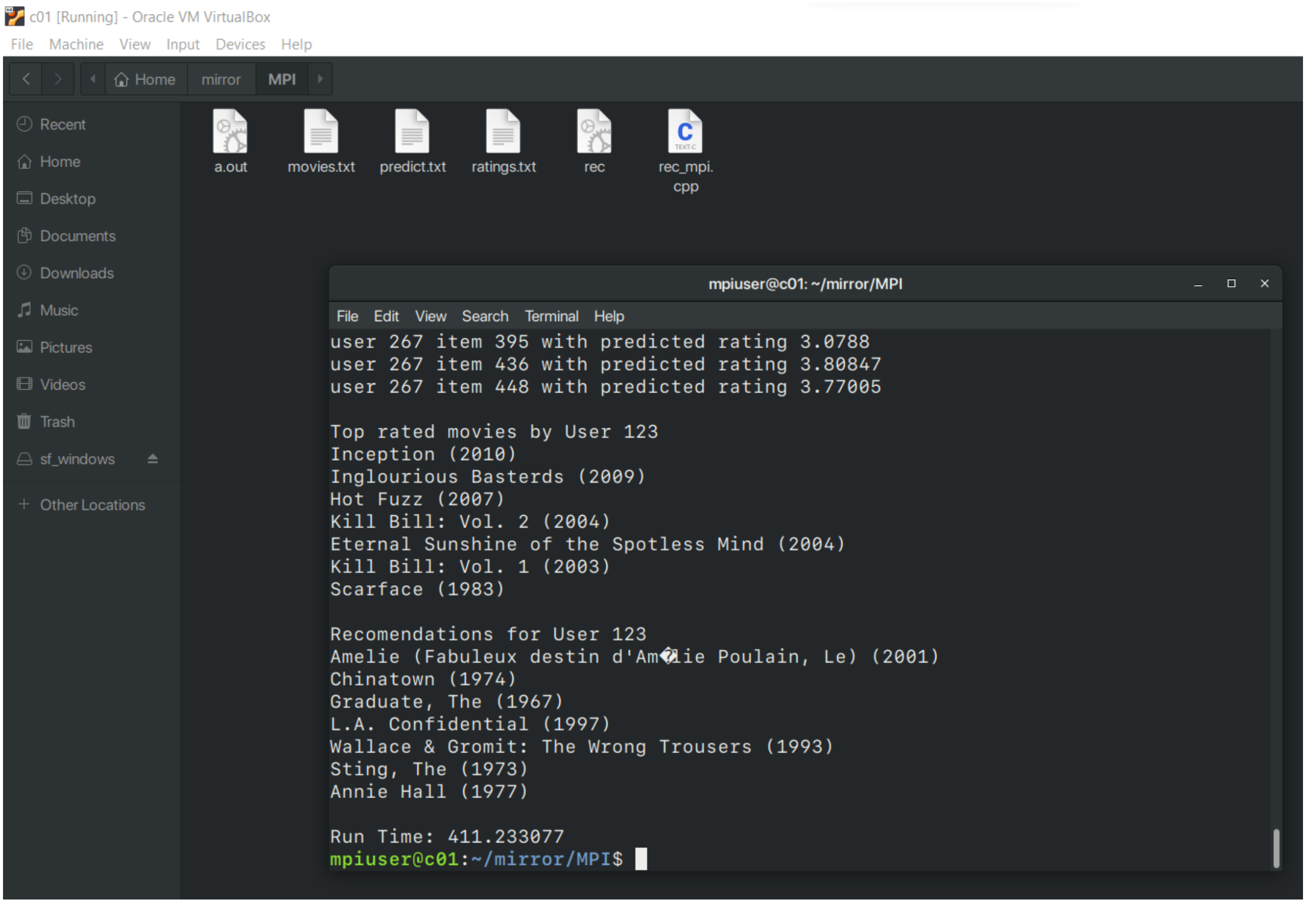
```
                    for (int k = 0; k < usersNum; k++) //If user k has rated item
j, calculate the cosSimilarity between user k and user i
                    {
                        if (ratingsMat[k][j]) //Find user k who has rated item j
                        {
                            vector<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(int argc, char *argv[])
{
```

```
    int numtasks, taskid, numworkers, source, dest, mtype, rows, averow, extra,
offset, i, j, k, rc;
    vector<vector<double>> ratingsMat;

    double start, end, runtime;

    int row = 268;
    int col = 450;

    vector<vector<double>> predict(row, vector<double>(col, 0));

    string file1("ratings.txt");
    string file2("movies.txt");

    ratingsMat = matRead(file1, row, col);
    moviesList = movieRead(file2);

    MPI_Status status;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &taskid);
    MPI_Comm_size(MPI_COMM_WORLD, &numtasks);

    start = MPI_Wtime();

    numworkers = numtasks - 1;

    //master task:

    if (taskid == MASTER)
    {

        averow = NR / numworkers;
        extra = NR % numworkers;
        offset = 0;
        mtype = FROM_MASTER;
        for (dest = 1; dest <= numworkers; dest++)
        {
            rows = (dest <= extra) ? averow + 1 : averow;
            MPI_Send(&offset, 1, MPI_INT, dest, mtype, MPI_COMM_WORLD);
```

```
            MPI_Send(&rows, 1, MPI_INT, dest, mtype, MPI_COMM_WORLD);
            offset = offset + rows;
        }

        mtype = FROM_WORKER;
        for (i = 1; i <= numworkers; i++)
        {
            source = i;
            MPI_Recv(&offset, 1, MPI_INT, source, mtype, MPI_COMM_WORLD,
&status);
            MPI_Recv(&rows, 1, MPI_INT, source, mtype, MPI_COMM_WORLD, &status);
            MPI_Recv(&predict[offset][0], rows * NC, MPI_DOUBLE, source, mtype,
MPI_COMM_WORLD, &status);
        }

        end = MPI_Wtime();

        //matWrite(predict, "predict.txt");
        topRatings(ratingsMat, 123);
        makeRec(predict, 123);
        //printf("\noffset: %d, rows: %d, sum: %d ", offset, rows, rows +
offset);
        runtime = end - start;
        printf("\nrun time: %f", runtime);
    }

    //wprker task
    if (taskid > MASTER)
    {
        mtype = FROM_MASTER;
        MPI_Recv(&offset, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD, &status);
        MPI_Recv(&rows, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD, &status);

        char pro_name[MPI_MAX_PROCESSOR_NAME];
        int name_len;
        MPI_Get_processor_name(pro_name, &name_len);
        //printf("\nWorking in Processor %s\n", pro_name);

        predict = colabFilter(ratingsMat, row, col, offset, rows);
```

```
        mtype = FROM_WORKER;
        MPI_Send(&offset, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD);
        MPI_Send(&rows, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD);
        MPI_Send(&predict[offset][0], rows * NC, MPI_DOUBLE, MASTER, mtype,
MPI_COMM_WORLD);
    }
    MPI_Finalize();
}
```

# Output:



# Approach:

For the most efficient implementation of MPI we need to divide the problem into equal parts. This will balance the computational load across all the nodes.

In the recommendation system, we use User Based Collaborative Filtering. In this method each user in the dataset is compared with every other user present. To equally balance the load we can divide the problem into an equal number of users per node. This can be done by segmenting the ratings matrix, which contains the rating each user has given for each movie. The rows of the matrix correspond to the users, and the columns correspond to the movies.

Hence we can divide the number of rows by the number of processors available and do operations on it parallelly. Note that each user is compared with every other user, thus each processor will require the entire dataset matrix for calculating the prediction matrix.

```
if (taskid == MASTER)
{

    averow = NR / numworkers;
    extra = NR % numworkers;
    offset = 0;
    mtype = FROM_MASTER;
    for (dest = 1; dest <= numworkers; dest++)
    {
        rows = (dest <= extra) ? averow + 1 : averow;
        MPI_Send(&offset, 1, MPI_INT, dest, mtype, MPI_COMM_WORLD);
        MPI_Send(&rows, 1, MPI_INT, dest, mtype, MPI_COMM_WORLD);
        offset = offset + rows;
    }

    mtype = FROM_WORKER;
    for (i = 1; i <= numworkers; i++)
    {
        source = i;
        MPI_Recv(&offset, 1, MPI_INT, source, mtype, MPI_COMM_WORLD, &status);
        MPI_Recv(&rows, 1, MPI_INT, source, mtype, MPI_COMM_WORLD, &status);
        MPI_Recv(&predict[offset][0], rows * NC, MPI_DOUBLE, source, mtype, MPI_COMM_WORLD, &status);
    }

    end = MPI_Wtime();

    //matWrite(predict, "predict.txt");
    topRatings(ratingsMat, 123);
    makeRec(predict, 123);
    //printf("\noffset: %d, rows: %d, sum: %d ", offset, rows, rows + offset);
    runtime = end - start;
    printf("\nRun Time: %f\n", runtime);
}
```

The approach to implement this would be to have a “rows” and “offset” variable which is sent to the workers. These variables will determine the segment of the dataset where that worker must do the calculations. Following which the worker will send back the section of the prediction matrix it has calculated.

```
//worker task
if (taskid > MASTER)
{
    mtype = FROM_MASTER;
    MPI_Recv(&offset, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD, &status);
    MPI_Recv(&rows, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD, &status);

    char pro_name[MPI_MAX_PROCESSOR_NAME];
    int name_len;
    MPI_Get_processor_name(pro_name, &name_len);
    //printf("\nWorking in Processor %s\n", pro_name);

    predict = colabFilter(ratingsMat, row, col, offset, rows);

    mtype = FROM_WORKER;
    MPI_Send(&offset, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD);
    MPI_Send(&rows, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD);
    MPI_Send(&predict[offset][0], rows * NC, MPI_DOUBLE, MASTER, mtype, MPI_COMM_WORLD);
}
```

Finally the master will take the final matrix and print the recommendations based on user ID and number of recommendations.

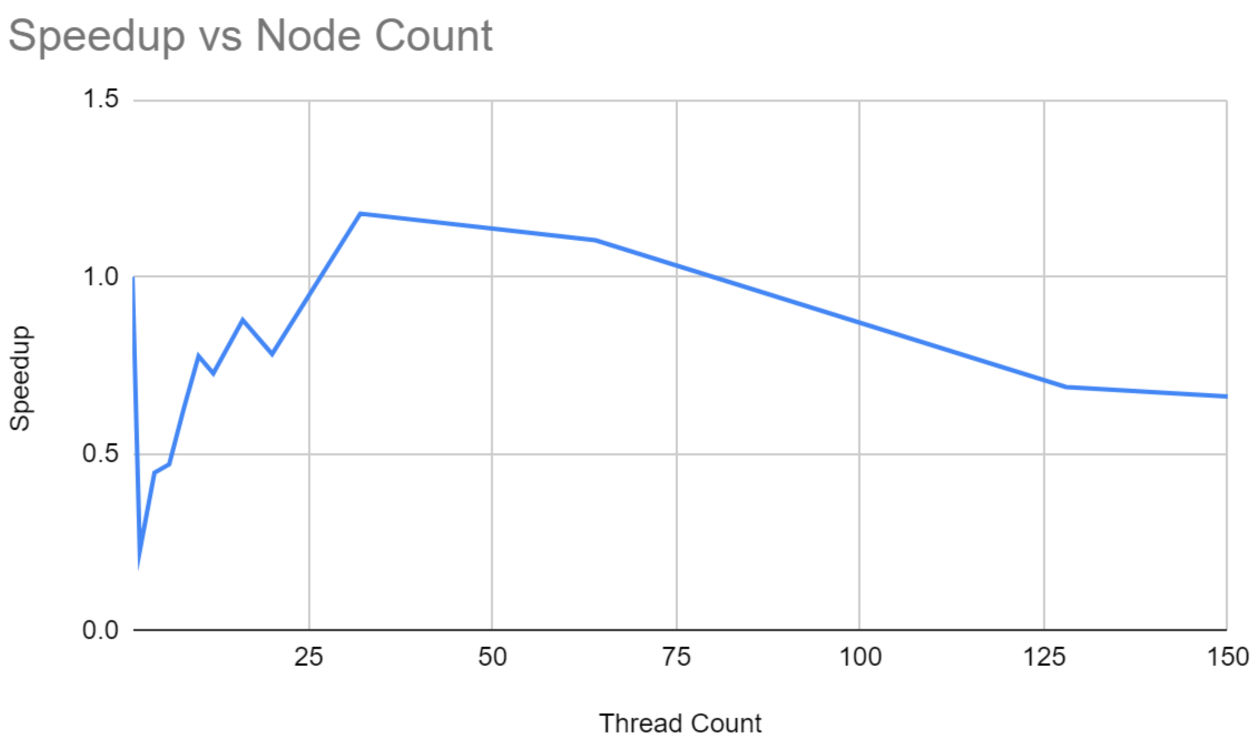
# Machine File Configuration:

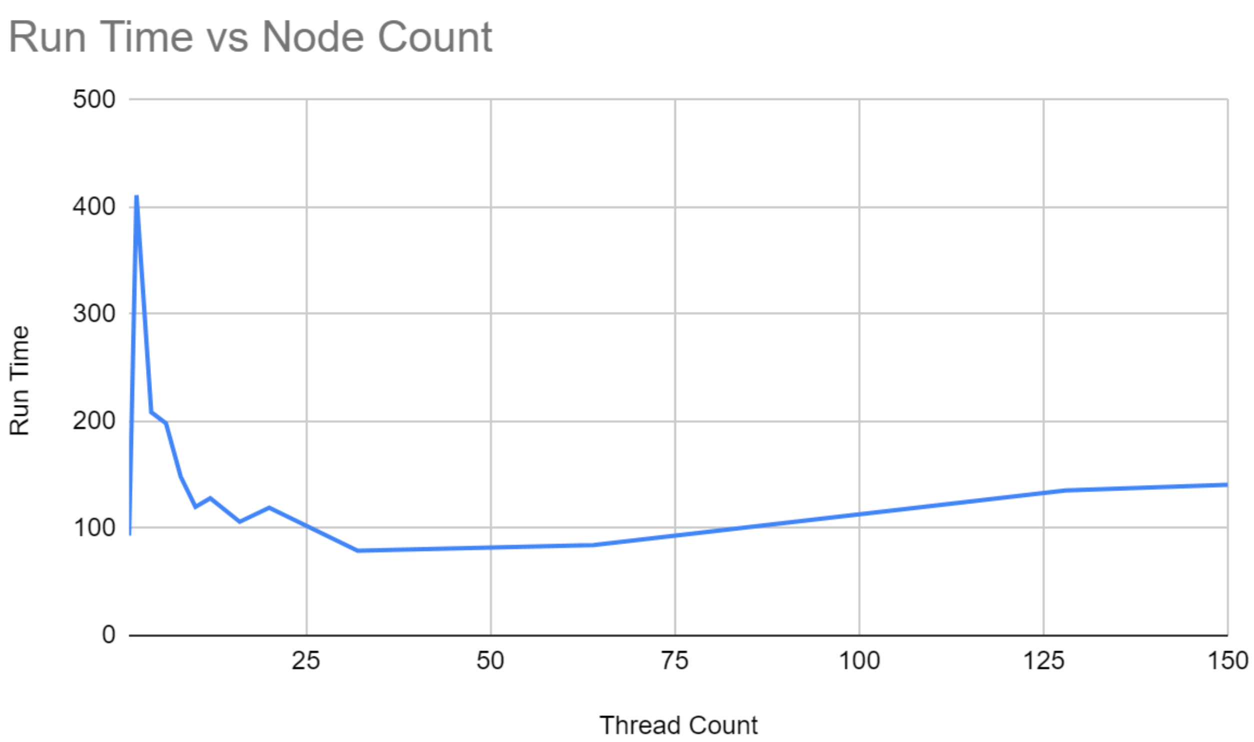
```
c01:5
c02:5
c03:5
```

# Analysis:

| Number of Nodes | Execution Time | Speed-Up | Parallelization Factor |
|---|---|---|---|
| 1 | 93.27756 | 1 | |
| 2 | 411.233077 | 0.2268240694 | -681.7406394 |
| 4 | 208.56105 | 0.447243433 | -164.7891733 |
| 6 | 198.134934 | 0.4707779598 | -134.8972344 |
| 8 | 148.320531 | 0.6288917615 | -67.43985646 |
| 10 | 120.121061 | 0.7765296046 | -31.97565655 |
| 12 | 128.204902 | 0.7275662517 | -40.84857592 |
| 16 | 106.20787 | 0.8782546905 | -14.78633303 |
| 20 | 119.28221 | 0.7819905416 | -29.34608902 |
| 32 | 79.14829 | 1.178516428 | 15.63618613 |
| 64 | 84.537585 | 1.103385672 | 9.518585994 |
| 128 | 135.37451 | 0.6890334081 | -45.48620488 |
| 150 | 140.792562 | 0.6625176691 | -51.2812456 |

# Graphs:


Speedup vs Node Count
1.5
1.0
0.5
0.0
Speedup
25
50
75
100
125
150
Thread Count


Run Time vs Node Count
500
400
300
200
100
0
Run Time
25
50
75
100
125
150
Thread Count

# Observations:

- The system is configured with 3 VMs each with 5 virtual nodes. Hence it is a virtual cluster with 15 nodes.

- After an initial drop in speedup we see an increase towards node count = 32.

- The run time decreases from node count = 2 to node count = 16. Then we see a slight increase at node count = 20, which subsequently drops at 32. Following this the run time tapers off with a slight increase.

- The reason for the initial drop is the fact that MPI uses a shared memory setup, hence in the case of master with 1 worker, the data has to be sent over the network which leads to an increase in run time.

HPC Project Part 2
CUDA Report

# Recommendation System

(Collaborative Filtering)

**Roll No:** CED18I042
**Name:** Reuben Skariah Mathew

**Date:** 4th December, 2021

# Hardware Configuration (VM):

**Architecture:** x86_64
**CPU op-mode(s):** 32-bit, 64-bit
**Byte Order:** Little Endian
**CPU(s):** 2
**On-line CPU(s) list:** 0,1
**Thread(s) per core:** 2
**Core(s) per socket:** 1
**Socket(s):** 1
**NUMA node(s):** 1
**Vendor ID:** GenuineIntel
**CPU family:** 6
**Model:** 63
**Model name:** Intel(R) Xeon(R) CPU @ 2.30GHz
**Stepping:** 0
**CPU MHz:** 2299.998
**BogoMIPS:** 4599.99
**Hypervisor vendor:** KVM
**Virtualization type:** full
**L1d cache:** 32K
**L1i cache:** 32K
**L2 cache:** 256K
**L3 cache:** 46080K
**NUMA node0 CPU(s):** 0,1
**RAM:** 12 GB

# CUDA Code:

```
%%cu

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

#define N 268
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];
        }
    }
    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();
```

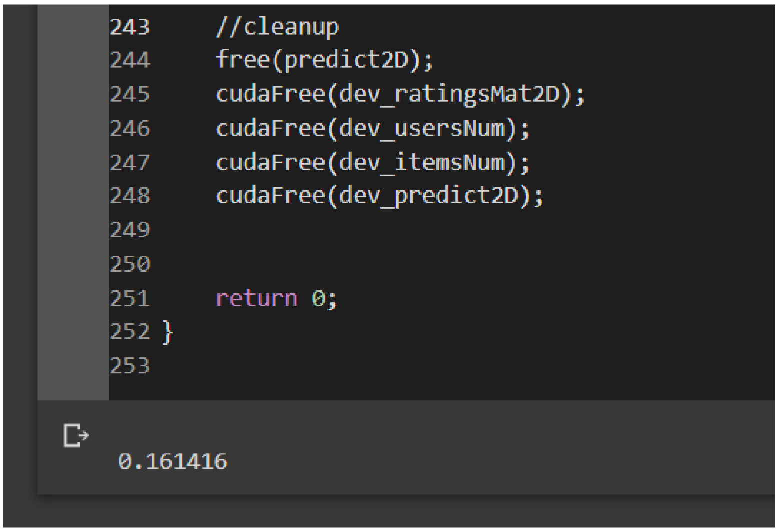
```
    int col = mat[0].size();

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

//*************************************************************//

__device__ double norm(double *A) //device function
{
    double res = 0;
    for (int i = 0; i < 450; ++i)
    {
        res += pow(A[i], 2);
    }
    return sqrt(res);
}

__device__ double dotProduct(double *A, double *B)
{
    double res = 0;
    for (int i = 0; i < 450; ++i)
    {
        res += A[i] * B[i];
    }
    return res;
}

__device__ double adjCosineSimilarity(double *A, double *B) //cosine similarity
(cosine similarity - mean)
{
    return dotProduct(A, B) / (norm(A) * norm(B)); //if output is nan then there
is no correlation
}

__device__ void checkCommon(double *A, double *B, double *&C, double *&D) //to
check if both A and B have rated
{
    int n = 0;
    for (int i = 0; i < 450; ++i)
    {
        if (A[i] && B[i])
        {
            C[n] = A[i];
            D[n] = B[i];
            n++;
```

```
        }
    }
}

__device__ void colabFilter(double **dev_ratingsMat, int *dev_usersNum, int
*dev_itemsNum, double **dev_predict, int userID)
{
    int usersNum = *dev_usersNum;
    int itemsNum = *dev_itemsNum;
    double **ratingsMat = dev_ratingsMat;
    double *A = new double[itemsNum];
    double *B = new double[itemsNum];
    double *C = new double[itemsNum];
    double *D = new double[itemsNum];
    for (int i = 0; i < itemsNum; ++i)
    {
        A[i] = ratingsMat[userID][i];
        B[i] = ratingsMat[userID][i];
    }
    checkCommon(A, B, C, D);
    for (int i = 0; i < itemsNum; ++i)
    {
        if (C[i] && D[i])
        {
            dev_predict[userID][i] = adjCosineSimilarity(C, D);
        }
    }
}

__global__ void rec(double **dev_ratingsMat, int *dev_usersNum, int
*dev_itemsNum, double **dev_predict)
{
    int usersNum = *dev_usersNum;
    int itemsNum = *dev_itemsNum;
    double **ratingsMat = dev_ratingsMat;
    double **predict = dev_predict;
    int userID = blockIdx.x * blockDim.x + threadIdx.x;
    if (userID < usersNum)
    {
        colabFilter(dev_ratingsMat, dev_usersNum, dev_itemsNum, dev_predict,
userID);
    }
}

double **convertVec2D(vector<vector<double>> &vals, int n, int m)
{
    double **temp;
    temp = new double *[n];
    for (unsigned int i = 0; (i < n); i++)
    {
```

```
        temp[i] = new double[m];
        for (unsigned int j = 0; (j < m); j++)
        {
            temp[i][j] = vals[i][j];
        }
    }
    return temp;
}

int main()
{
    double time_spent = 0.0;
    clock_t begin = clock();

    int usersNum = 268; //users
    int itemsNum = 450; //items:movies

    int size = usersNum * itemsNum * sizeof(double);

    vector<vector<double>> ratingsMat = matRead("ratings.txt", usersNum,
itemsNum);
    double **ratingsMat2D = convertVec2D(ratingsMat, usersNum, itemsNum);

    vector<string> moviesList = movieRead("movies.txt");

    double **predict2D;
    predict2D = (double **)malloc(size);

    //create device variables
    double **dev_ratingsMat2D;
    int *dev_usersNum;
    int *dev_itemsNum;
    double **dev_predict2D;

    cudaMalloc((void **)&dev_ratingsMat2D, size);
    cudaMalloc((void **)&dev_predict2D, size);
    cudaMalloc((void **)&dev_usersNum, sizeof(int));
    cudaMalloc((void **)&dev_itemsNum, sizeof(int));

    cudaMemcpy(dev_ratingsMat2D, ratingsMat2D, size, cudaMemcpyHostToDevice);
    cudaMemcpy(dev_usersNum, (int *)usersNum, sizeof(int *),
cudaMemcpyHostToDevice);
    cudaMemcpy(dev_itemsNum, (int *)itemsNum, sizeof(int *),
cudaMemcpyHostToDevice);

    //create device function
    rec<<<N, N/8>>>(dev_ratingsMat2D, dev_usersNum, dev_itemsNum, dev_predict2D);
    cudaDeviceSynchronize();

    cudaMemcpy(predict2D, dev_predict2D, size, cudaMemcpyDeviceToHost);
```

```
    clock_t end = clock();
    time_spent += (double)(end - begin) / CLOCKS_PER_SEC;
    printf("%f", time_spent);

    vector<vector<double>> predictFinal;


    for (int i = 0; i < usersNum; i++)
    {
        for (int j = 0; j < itemsNum; j++)
        {
            predictFinal[i][j] = &predict2D[i][j];
        }
    }

    //top ratings by user
    topRatings(ratingsMat, 123);

    //make recomendations
    makeRec(predictFinal, 123);

    //write predict matrix
    matWrite(predictFinal, "predict.txt");

    //cleanup
    free(predict2D);
    cudaFree(dev_ratingsMat2D);
    cudaFree(dev_usersNum);
    cudaFree(dev_itemsNum);
    cudaFree(dev_predict2D);
    return 0;
}
```

# Output:



# Approach:

The host (CPU) sends the code to the device (GPU) and the significant processing will take place parallely on the GPU. As there are a large number of cores in the GPU, our thread count can be high.

In this case each user is taken as a separate thread, as the same processing is done for every user parallely. The threads will be synchronized after the rows have been processed.

The ratings matrix, predictions matrix, number of users and number of items (movies) are sent from host to device.

```
cudaMemcpy(dev_ratingsMat2D, ratingsMat2D, size,
cudaMemcpyHostToDevice);
cudaMemcpy(dev_usersNum, (int *)usersNum, sizeof(int),
cudaMemcpyHostToDevice);
cudaMemcpy(dev_itemsNum, (int *)itemsNum, sizeof(int),
cudaMemcpyHostToDevice);
```

Then the function which calculates predicted values is called to run on the device.

```
//create device function
rec<<<usersNum / BLOCK_SIZE, BLOCK_SIZE>>>(dev_ratingsMat2D,
dev_usersNum, dev_itemsNum, dev_predict2D);
cudaDeviceSynchronize();
```

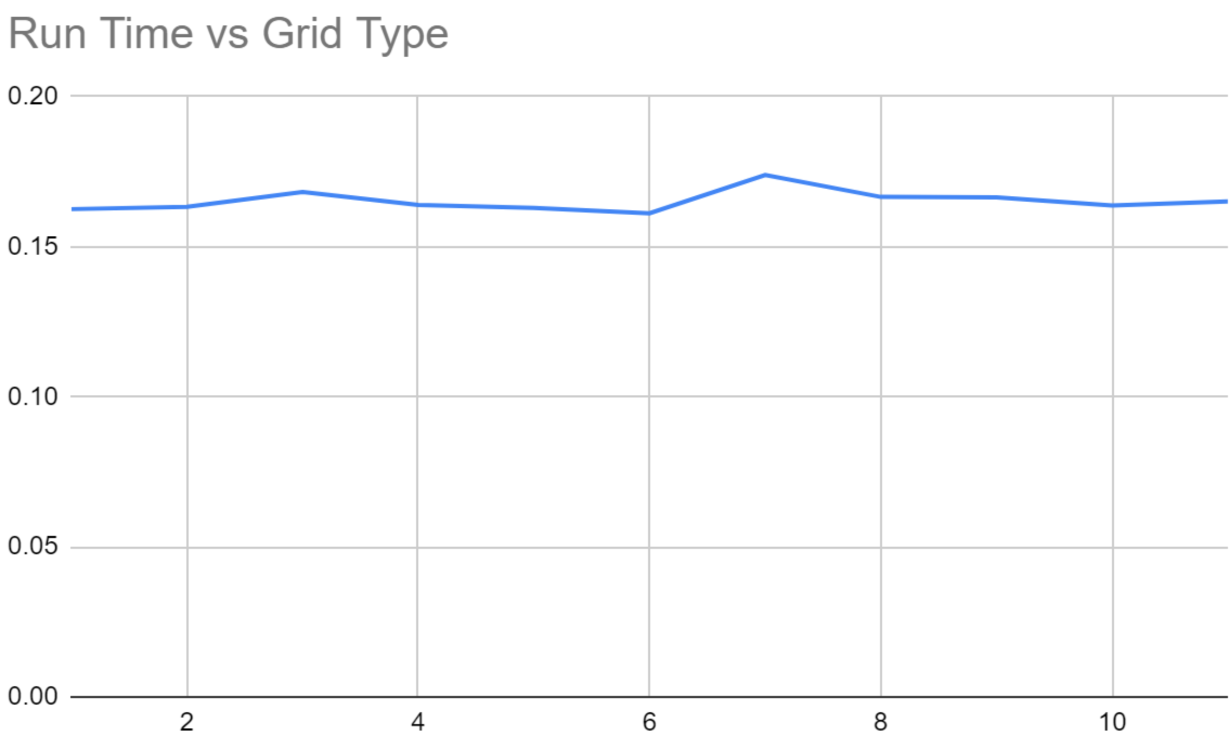
The __global__ function, which can be called from the host, runs for every user (taken as a separate thread).

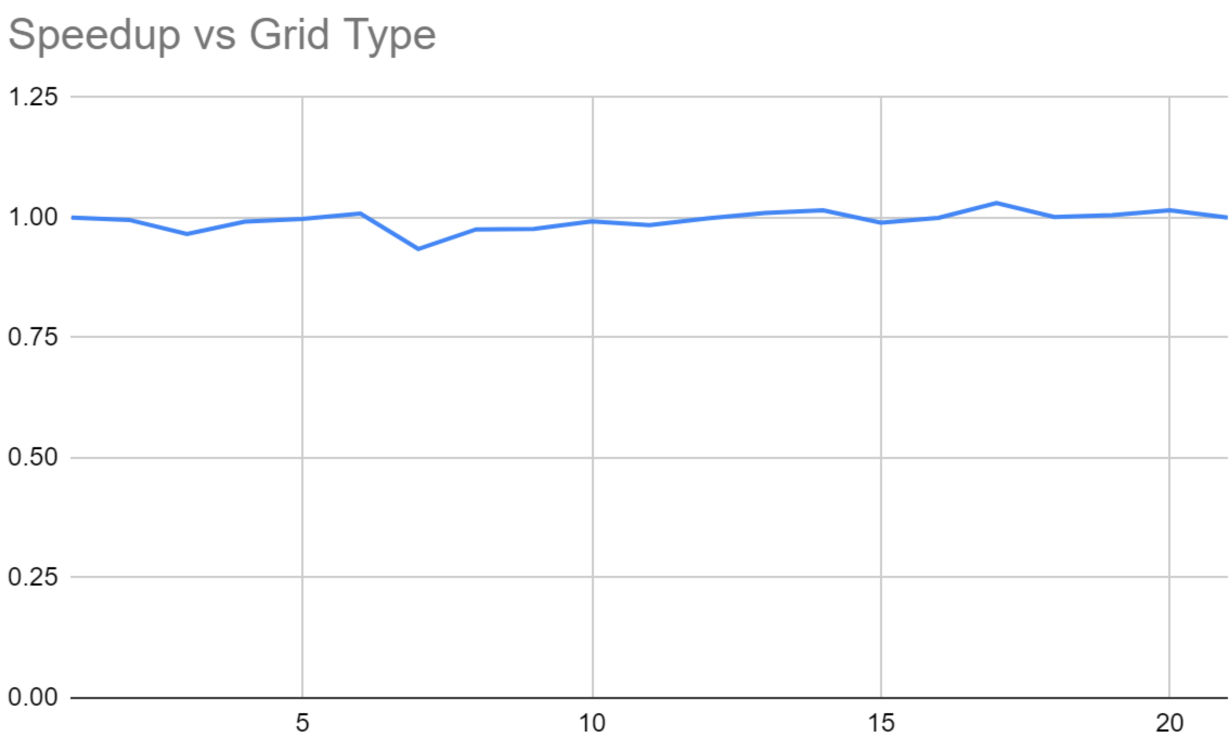
```
__global__ void rec(double **dev_ratingsMat, int *dev_usersNum,
int *dev_itemsNum, double **dev_predict)
{
    int usersNum = *dev_usersNum;
    int itemsNum = *dev_itemsNum;
    double **ratingsMat = dev_ratingsMat;
    double **predict = dev_predict;
    int userID = blockIdx.x * blockDim.x + threadIdx.x;
    if (userID < usersNum)
    {
        colabFilter(dev_ratingsMat, dev_usersNum, dev_itemsNum,
        dev_predict, userID);
    }
}
```

# Analysis:

| Number of Blocks | Threads per block | Execution Time | Speed-Up | Grid Type |
|---|---|---|---|---|
| 1 | 1 | 0.162512 | 1 | 1 |
| 1 | 10 | 0.163261 | 0.995412254 | 2 |
| 1 | 20 | 0.168251 | 0.9658902473 | 3 |
| 1 | 30 | 0.163881 | 0.9916463776 | 4 |
| 1 | 40 | 0.162941 | 0.9973671452 | 5 |
| 1 | 50 | 0.161126 | 1.008601964 | 6 |
| 10 | 10 | 0.173856 | 0.9347505982 | 7 |
| 20 | 10 | 0.166568 | 0.9756495846 | 8 |
| 30 | 10 | 0.16643 | 0.9764585712 | 9 |
| 40 | 10 | 0.163749 | 0.9924457554 | 10 |
| 50 | 10 | 0.165083 | 0.984426016 | 11 |
| 1 | N | 0.162733 | 0.9986419472 | 12 |
| N/8 | N/2 | 0.160906 | 1.009980983 | 13 |
| N/2 | N/8 | 0.160026 | 1.015534976 | 14 |
| N/4 | N/2 | 0.164225 | 0.9895691886 | 15 |
| N/2 | N/4 | 0.162534 | 0.9998646437 | 16 |
| N | N/8 | 0.157721 | 1.030376424 | 17 |
| N/2 | N/2 | 0.162269 | 1.001497513 | 18 |
| N/2 | N | 0.16163 | 1.005456908 | 19 |
| N | N/2 | 0.160045 | 1.015414415 | 20 |
| N | N | 0.162518 | 0.999963081 | 21 |

# Graphs:


Run Time vs Grid Type
0.20
0.15
0.10
0.05
0.00
2
4
6
8
10


Speedup vs Grid Type
1.25
1.00
0.75
0.50
0.25
0.00
5
10
15
20

# Inference:

- The runtimes are similar for all values of blocks and threads.
- The maximum speedup is 1.030376424 where the number of blocks is N and the number of threads per block is N/8. (where N = 268)
- The lack of significant speedup may be due to parallelization overhead as the inputs are relatively small, as well as due to the online runtime environment.