Item-based Collaborative Filtering Movie Recommendation

Hsing-Wen Hsu hh2916

1 Overview & Background

Item-based Collaborative Filtering is an algorithm that computes item similarity based on ratings or interactions by users. With this algorithm, we can recommend users the items they might like by computing the similarity between the items they actually like and other items they haven't bought or interacted with yet.

Let the number of items be $|I|$, the number of users be $|U|$, an item can be represented by a 1D vector of length $|U|$. If a user has rated this item, the corresponding entry will be the rating that the user gave to the item. If a user hasn't rated this item, the corresponding will simply be 0. Here is an example of the utility matrix with 6 items and 12 users (min rating: 1, max rating: 5):

$i\backslash u$	1	2	3	4	5	6	7	8	9	10	11	12
1	1		3		?	5			5		4	
2			5	4			4			2	1	3
3	2	4		1	2		3		4	3	5	
4		2	4		5			4			2	
5			4	3	4	2					2	5
6	1		3		3			2			4	

Recommending a user items involves the following steps:

1. Mean-center the utility matrix
2. Compute the similarities between items with the mean-centered matrix
3. Predict the score of the items that the user hasn't rated yet and returns the list of items with higher score.

The following sections describes the exact method of the above steps:

Mean-centering

If an item is not rated by a user yet, the corresponding entry will remain 0 in the mean-centered matrix. If an item is already rated, then the mean-centered value will be the original value minus the average value. Let the rating of item i by user x be r_{ix}, c_i be the number of users that have rated item i, the mean-centered rating of an item by user x will be:

$$mean - centerered_{ix} = r_{ix} - \frac{\Sigma_{u \in Users} r_{iu}}{c_i} \quad (1)$$

Computing Cosine Similarity

In the mean-centered matrix, each row is a vector that represents an item. Let vec_i and vec_j be the vectors representing item i and item j respectively, the cosine similarity between the two items is:

$$Similarity_{ij} = \frac{vec_i \cdot vec_j}{|vec_i||vec_j|} \tag{2}$$

Rating Prediction

Let S_{ij} be the similarity between movie i and j, r_{jx} be the rating of user x on movie j, $N(i;x)$ be the items rated by user x that are similar to i. The predicted rating for item i by user x:

$$r_{ix} = \frac{\Sigma_{j \in N(i;x) S_{ij} \cdot r_{jx}}}{\Sigma_{j \in N(i;x) S_{ij}}} \tag{3}$$

, which is the weighted average of the items that are similar to item i.

2 Problem Formulation

In this project, the MovieLens dataset is used. The MovieLens dataset provides 100000 ratings (943 users, 1682 movies). The program will compute the similarities between the items, predict the rating of each of them, and recommend a list of items to the specified user.

The input to the program contains the user's id, the number of items that we wished to used to predict the ratings with, and the number of items that we want to recommend to the user. The output of the program will simply be a list of item ids.

3 Data Preprocessing

Each entry in the MovieLens dataset is as follows (Range of user id: 1-943, item id: 1-1682, rating: 1-5):

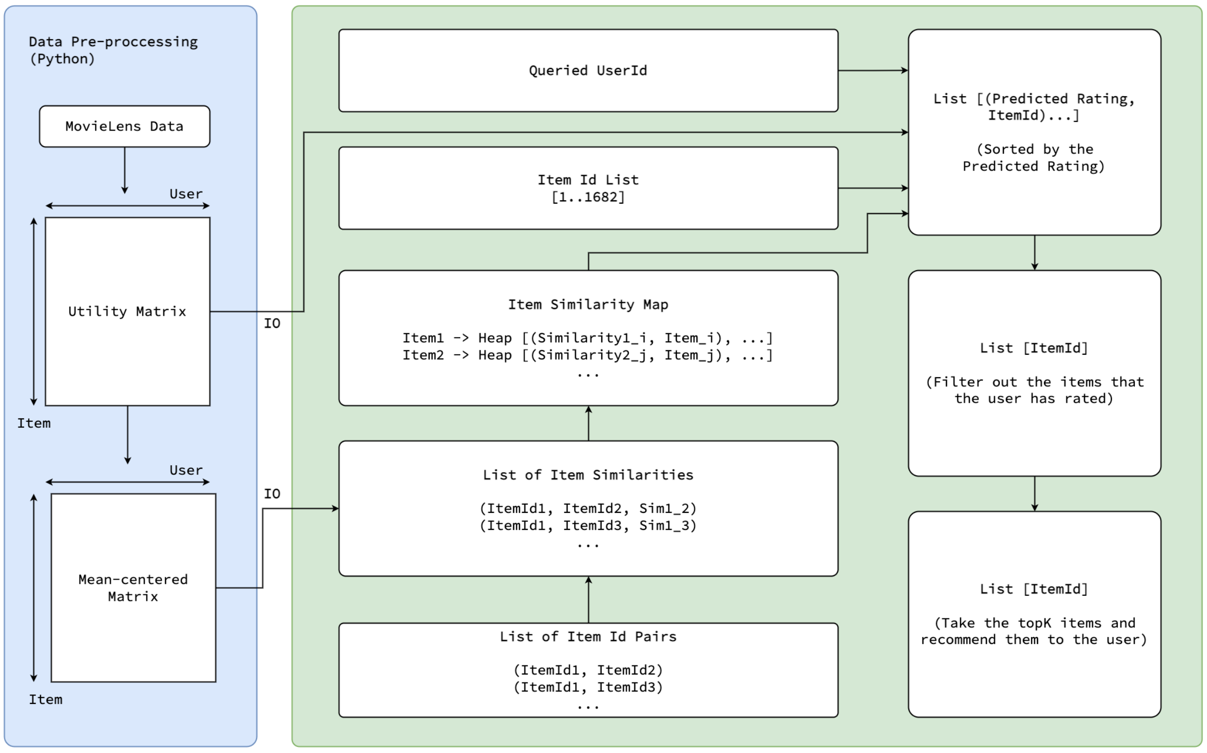
UserId ItemId Rating Timestamp

My original plan was to read the data line by line and initialize the utiltiy matrix with each of the entries. However, it takes a huge amount of time to initialized the matrix, such that the program just halts. I think this is because the `setElem` function returns an entirely new matrix every time an entry is set.:

```
setElem :: a
-> (Int, Int)
-> Matrix a
-> Matrix a
```

Therefore, I wrote a Python program that pre-processes the MovieLens data into the utility matrix and the mean-centered matrix. There are two input files to my Haskell program, one is the original utility matrix, and the other one is the mean-centered matrix. Each of them is text file with a $1682 * 643$ matrix.

4 Main Procedure


Data Pre-proccessing (Python)
MovieLens Data
User
Utility Matrix
Item
User
Mean-centered Matrix
Item
IO
IO
Queried UserId
Item Id List [1..1682]
Item Similarity Map
Item1 -> Heap [(Similarity1_i, Item_i), ...]
Item2 -> Heap [(Similarity2_j, Item_j), ...]
...
List of Item Similarities
(ItemId1, ItemId2, Sim1_2)
(ItemId1, ItemId3, Sim1_3)
...
List of Item Id Pairs
(ItemId1, ItemId2)
(ItemId1, ItemId3)
...
List [(Predicted Rating, ItemId)...]
(Sorted by the Predicted Rating)
List [ItemId]
(Filter out the items that the user has rated)
List [ItemId]
(Take the topK items and recommend them to the user)

The above chart shows the procedure of the program.

1. First, a Python program is used to read and preprocess the MovieLens data into two text files. One of them is the original utility matrix, and the other one is the mean-centered matrix. Both files contains 1682 rows and 943 columns.

2. The Haskell program reads the matrix text files into two `Data.Matrix`.

3. A list of item id pairs are generated. This is used to compute the similarity later. Example look of the lis `[(1, 2), (1, 3), ..., (1681, 1682)]`

4. A function that computes the cosine similarity between items will take the list generated in the last step and computes the cosine similarity between each items. After the computation we will get a list of tuples with the items' ids and the similarities between them: $[(item_1, item_2, similarity_{1,2})...]$

5. Obtaining the item similarity map: When we are computing the predicted score for an item, we need to be able to access the items that are highly similar to it quickly. A map is suitable in the situation. With the list of item pairs and similarities in the last steps, we can initialize an item similarity map. The key of the map is the item's id, and the value of the map is a Heap that that stores tuples of a similarity value and an item id.

6. For the queried user, we predicted the ratings of all 1682 items and return a list of type `[(Rating, ItemId)]`

7. The list `[(Rating, ItemId)]` is then sorted by the rating.

8. Items that are already rated by the user will be filtered from the list obtained in the previous step. The rating is also removed from the list, so we are only getting a list of item ids from this step.

9. For the final step, the top k items are recommended to the user. k is also a user input for the program.

5 Implementation

I have written several modules with different purposes for this project.

5.1 CFDataStructures

This module contains the data type that are used over the entire project so that the code would be more descriptive:

```
type Rating = Double
type UserId = Int
type ItemId = Int
type ItemRating = Vector Rating
type Similarity = Double
type ItemSimHeap = Heap.MaxPrioHeap Similarity ItemId

data RatingEntry = RatingEntry {
    rUserId :: UserId,
    rItemId :: ItemId,
    rRating :: Rating
} deriving (Eq, Show)
```

1. Rating: Data type of an element in the matrix (could be an element of the original utility matrix or the mean-centered matrix).
2. UserId: Data type that represents a user's id.
3. ItemId: Data type that represents an item's id.
4. ItemRating: A vector of Rating.
5. Similarity: Data type that represents the similarity between two items.
6. ItemSimHeap: This is a heap that is sorted by the similarity. Items with a larger similarity will be on the top of the heap. When we are doing the actual computation, each item will have it's own ItemSimHeap. This data structure is used to fetch the top items that are similiar to an item.
7. RatingEntry: This data structure stores a row of data of the MovieLens data (mentioned in section 3). This data structure is used to store the test data in the dataset. With a list of RatingEntry object, we can compute the RMSE (Root-mean-square Error) and verify the correctness of the program. However, this data structure is not directly related to the recommendation process.

5.2 IOUtils

This module includes helper functions that read the input matrix. I referenced my code for hw5 for the implementation.

1. readTableFile

```
readTableFileLambda :: Handle -> (Handle -> IO a) -> IO [a]
readTableFileLambda hdl getFunc = do
  t <- hIsEOF hdl
  if t then return []
  else do
    x <- getFunc hdl-- x is a string
    fmap ([x] ++) (readTableFileLambda hdl getFunc)

readTableFile :: String -> (Handle -> IO a) -> IO [a]
readTableFile fname getFunc = do
    withFile fname ReadMode $ \hdl -> readTableFileLambda hdl getFunc

```

2. One of the inputs of the readTableFile function is `getFunc`. The `getFunc` for reading a row of the input data is named `getMatrixRow`:

```haskell
getMatrixRow :: Handle -> IO [Rating]
getMatrixRow h = do
    inputs <- hGetLine h
    let rowString = words inputs
    let rowRating = map (\x -> read x :: Double) rowString
    return rowRating

```

3. Combining the above functions we get readMatrixData. The returned type of this function is `IO[[Rating]]`, which is actually 1682 lists of list of Ratings. Each list corresponds to an item:

```haskell
readMatrixData :: String -> IO[[Rating]]
readMatrixData filename = readTableFile filename getMatrixRow

```

The original utility matrix and the mean-centered matrix are read in Main.hs with the above functions. The resulting `IO[[Rating]]` is later used to initialize two `Data.Matrix` (`ratingMatrix, meanCentered`):

```haskell
ratingMatrixData <- readMatrixData filenameOri
mcMatrixData <- readMatrixData filenameMc
let ratingMatrix = Matrix.fromLists ratingMatrixData
let meanCentered = Matrix.fromLists mcMatrixData
```

5.3 SimilarityLib

This module contains functions that are used to compute the cosine similarity.

1. getDotProduct: Computes the dot product of two vectors

```haskell
getDotProduct :: ItemRating -> ItemRating -> Double
getDotProduct ir1 ir2 = Vector.sum $ Vector.zipWith (\x y -> x * y) ir1 ir2

```

2. getVectorLen: Computes the length of a vector:

```haskell
getVectorLen :: ItemRating -> Double
getVectorLen ir = sqrt $ (Vector.sum $ Vector.zipWith (\x y -> x * y) ir ir)

```

3. getCosSimilarityHelper: Computes the cosine similarity of two items. (An item is represented by a vector of rating, which is of type `ItemRating`).

```haskell
getCosSimilarityHelper :: ItemRating -> ItemRating -> Similarity
getCosSimilarityHelper ir1 ir2
    | dotProduct == 0 = 0
    | vecLen1 == 0 = 0
    | vecLen2 == 0 = 0
    | otherwise = dotProduct / (vecLen1 * vecLen2)
      where dotProduct = getDotProduct ir1 ir2
            vecLen1    = getVectorLen ir1
            vecLen2    = getVectorLen ir2

```

4. getCosSimilarity: Takes the mean-centered utility matrix and two items' ids and returns the similarity between the two items:

```haskell
getCosSimilarity :: Matrix Rating -> ItemId -> ItemId -> Similarity
getCosSimilarity mc iid1 iid2 =
    getCosSimilarityHelper (getRow iid1 mc) (getRow iid2 mc)

```

5.4 CFLib

This library contains the main functions for collaborative filtering.

1. `getItemPairs`: To compute the similarity between all pairs of item, we need a list of tuples that contains all combinations of items. Note that there are no duplicate pairs here (if (i, j) is present, (j, i) will not be in the list).

```
getItemPairs :: Int -> [(ItemId, ItemId)]
getItemPairs 0 = []
getItemPairs itemNum = [(i, j) | i <- [1..itemNum], j <- [(i + 1)..itemNum]]

```

2. `getSimTuple`: Takes a pair of items, and the mean-centered matrix, and computes the similarity between them. The function returns a tuple with the item ids and the similarity.

```
getSimTuple :: Matrix Rating -> (ItemId, ItemId) -> (ItemId, ItemId, Similarity)
getSimTuple mc (itemId1, itemId2) =
    (itemId1, itemId2, getCosSimilarity mc itemId1 itemId2)

```

3. Different flavors of `getSimTuples`:

   The goal of `getSimTuples` is to generate a list of `(ItemId, ItemId, Similarity)` for all the possible combinations. We can parallelize this part since the computation of two pairs of items will not affect each other.

   (a) `getSimTuples0`: The sequential version

```
getSimTuples0 :: Matrix Rating -> [(ItemId, ItemId)] -> [(ItemId, ItemId,
    Similarity)]
getSimTuples0 _ [] = []
getSimTuples0 mc x = map f x
  where f :: (ItemId, ItemId) -> (ItemId, ItemId, Similarity)
        f (itemId1, itemId2) = (itemId1, itemId2, getCosSimilarity mc
    itemId1 itemId2)

```

   (b) `getSimTuples1`: Parallelism with `parList rseq`

```
getSimTuples1 :: Matrix Rating -> [(ItemId, ItemId)] -> [(ItemId, ItemId,
    Similarity)]
getSimTuples1 _ [] = []
getSimTuples1 mc x = map (getSimTuple mc) x `using` parList rseq

```

   (c) `getSimTuples2`: Parallelism with `parListChunk chunkSize rseq`

```
getSimTuples2 :: Matrix Rating -> [(ItemId, ItemId)] -> Int -> [(ItemId,
    ItemId, Similarity)]
getSimTuples2 _ [] _ = []
getSimTuples2 mc x chunkSize = map(getSimTuple mc) x `using` parListChunk
    chunkSize rseq

```

   (d) `getSimTuplesStatic`: Partitions the item pair list into two and compute the similarity of items separately. This function is not involved in the main experiment because it is not paired with a `getPredictRatings` function. But the effect of it is explored in my presentation.

```
getSimTuplesStatic :: Matrix Rating -> [(ItemId, ItemId)] -> [(ItemId,
    ItemId, Similarity)]
getSimTuplesStatic _ [] = []
getSimTuplesStatic mc x = runEval $ do
    let (as, bs) = splitAt(length x `div` 2) x
    as' <- rpar (force (map (getSimTuple mc) as))
    bs' <- rpar (force (map (getSimTuple mc) bs))
    _ <- rseq as'
    _ <- rseq bs'
    return (as' ++ bs')

```

4. Functions for generating the item similarity map:

    (a) `createItemHeapList`: Creates a list of tuples for each item id and it's corresponding Heap.

```haskell
createItemHeapList :: Int
    -> ItemId
    -> [(ItemId, Heap.MaxPrioHeap Similarity ItemId)]
createItemHeapList 0 _ = []
createItemHeapList itemNum itemId
    = (itemId, Heap.fromList [] :: Heap.MaxPrioHeap Similarity ItemId) :
    createItemHeapList (itemNum - 1) (itemId + 1)

```

    (b) `createItemSimilarityMap`: Takes the list of tuples obtained with `createItemHeapList` and create a map that maps an item to its corresponding heap.

```haskell
createItemSimilarityMap :: Int
    -> Map.Map ItemId (Heap.MaxPrioHeap Similarity ItemId)
createItemSimilarityMap 0 = Map.fromList [] :: SimilarityMap
createItemSimilarityMap itemNum = Map.fromList $
    createItemHeapList itemNum 1

```

    (c) `setItemSimilarityMap`: Set the empty map created with the above functions with the computed item similarities.

```haskell
setItemSimilarityMap :: Map.Map ItemId (Heap.MaxPrioHeap Similarity ItemId)
    -> [(ItemId, ItemId, Similarity)]
    -> Map.Map ItemId (Heap.MaxPrioHeap Similarity ItemId)
setItemSimilarityMap sm [] = sm
setItemSimilarityMap sm ((itemId1, itemId2, sim):xs) = do
    case Map.lookup itemId1 sm of
        Just itemSimHeap1 -> do
            let itemSimHeap1' = Heap.insert(sim, itemId2) itemSimHeap1
            let sm' = Map.insert itemId1 itemSimHeap1' sm
            case Map.lookup itemId2 sm of
                Just itemSimHeap2 -> do
                    let itemSimHeap2' = Heap.insert(sim, itemId1)
    itemSimHeap2
                    let sm'' = Map.insert itemId2 itemSimHeap2' sm'
                    setItemSimilarityMap sm'' xs
                Nothing -> setItemSimilarityMap sm' xs
        Nothing -> do
            case Map.lookup itemId2 sm of
                Just itemSimHeap2 -> do
                    let itemSimHeap2' = Heap.insert (sim, itemId1)
    itemSimHeap2
                    let sm' = Map.insert itemId2 itemSimHeap2' sm
                    setItemSimilarityMap sm' xs
                Nothing -> setItemSimilarityMap sm xs

```

5. Functions for predicting the ratings:

    (a) `predictRating`: Takes the user's id and the id of the item that we are trying to predict and returns the predicted rating.

```haskell
predictRating :: Matrix Rating
    -> Map.Map ItemId (Heap.MaxPrioHeap Similarity ItemId)
    -> UserId
    -> Int
    -> ItemId
    -> Rating
predictRating ratingMatrix simMap userId k itemId =
    case Map.lookup itemId simMap of
        Just itemSimHeap ->
            predictRatingHelper ratingMatrix itemSimList userId k (0::
    Double) (0::Double)
                where itemSimList = Heap.toAscList itemSimHeap
        Nothing ->
            error "Item does not exist"

```

(b) `predictRatingHelper`: Iterate through the list of items and compute the weighted average of the rating of the item.

```haskell
predictRatingHelper :: Matrix Rating
    -> [(Similarity, ItemId)] -- Similarity List of the item that we are trying to predict
    -> UserId
    -> Int -- k
    -> Double -- numerator
    -> Double -- denominator
    -> Rating -- predicted rating
predictRatingHelper _ _ _ 0 numer deno
    | deno == 0 = 0
    | otherwise = numer / deno
predictRatingHelper ratingMatrix [] userId k numer deno
    | deno == 0 = 0
    | otherwise = numer / deno
predictRatingHelper ratingMatrix ((sim, itemJ):xs) userId k numer deno
    | rUJ == 0 = predictRatingHelper ratingMatrix xs userId k numer deno
    | otherwise = predictRatingHelper ratingMatrix xs userId (k - 1) (numer + rUJ * sim) (deno + sim)
        where rUJ = getElem itemJ userId ratingMatrix

```

6. Different flavors of `getPredictedRatings`. The function `getPredictedRatings` computes the predicted ratings for all 1682 items. There are different flavors of `getPredictedRatings`:

(a) `getPredictedRatings0`: The sequential version

```haskell
getPredictedRatings0 :: Matrix Rating -- seq
    -> Map.Map ItemId (Heap.MaxPrioHeap Similarity ItemId)
    -> UserId
    -> UserRating
    -> [ItemId] -- [1..1682]
    -> Int -- k
    -> [Rating]
getPredictedRatings0 _ _ _ _ [] _ = []
getPredictedRatings0 ratingMatrix itemSimMap userId userRating itemIds k
     = force (map (predictRating ratingMatrix itemSimMap userId k) itemIds)

```

(b) `getPredictedRatings1`: A version that uses `parList rdeepseq`

```haskell
getPredictedRatings1 :: Matrix Rating -- seq
    -> Map.Map ItemId (Heap.MaxPrioHeap Similarity ItemId)
    -> UserId
    -> UserRating
    -> [ItemId] -- [1..1682]
    -> Int -- k
    -> [Rating]
getPredictedRatings1 _ _ _ _ [] _ = []
getPredictedRatings1 ratingMatrix itemSimMap userId userRating itemIds k
     = force(map (predictRating ratingMatrix itemSimMap userId k) itemIds ‘using‘ parList rdeepseq)

```

(c) `getPredictedRatings2`: A version that uses `parListChunk chunkSize rdeepseq`

```haskell
getPredictedRatings2 :: Matrix Rating -- seq
    -> Map.Map ItemId (Heap.MaxPrioHeap Similarity ItemId)
    -> UserId
    -> UserRating
    -> [ItemId] -- [1..1682]
    -> Int -- k
    -> Int -- chunkSize
    -> [Rating]
getPredictedRatings2 _ _ _ _ [] _ _ = []
getPredictedRatings2 ratingMatrix itemSimMap userId userRating itemIds k chunkSize
     = force(map (predictRating ratingMatrix itemSimMap userId k) itemIds ‘using‘ parListChunk chunkSize rdeepseq)

```

7. `filterRatingItemPairs`: After all the ratings are computed, this function takes a list of `(Rating, ItemId)` and filters out the items that are not rated by the user with the user's column in the original utility matrix.

```
filterRatingItemPairs :: [(Rating, ItemId)] -> UserRating -> [ItemId]
filterRatingItemPairs [] _ = []
filterRatingItemPairs ((_, id):xs) userRating
    | userRating Vector.! (id - 1) == 0 = id : filterRatingItemPairs xs
    userRating
    | otherwise = filterRatingItemPairs xs userRating

```

8. `getTopK`: Takes the filtered list of item ids and return the top k items for the user.

```
getTopK :: [ItemId] -> Int -> [ItemId]
getTopK itemList k
    | length itemList < k = error "Not enough unrated items"
    | otherwise = take k itemList

```

6 Correctness

To verify that the algorithm is correctly implemented, I also implemented a function that computes the RMSE with the split datasets provided by the MovieLens dataset. In the MovieLens datasets, the original dataset `u.data` is 80%/20% split into u.base and u.test. There are five sets of split data (u1.base, u1.test, u2.base, u2.test, u3.base, u3.test, u4.base, u4.test, u5.base, u5.test). I made predictions with the data in u.base and computed the RMSE of the predicted ratings with u.test. Based on the statistics on the leaderboard of the MovieLens dataset [6], the resulting RMSE is in a reasonalbe range. (k is the number of items that are used to predict the rating of the item in consideration):

k	u1	u2	u3	u4	u5
3	1.13358	1.11366	1.09643	1.09571	1.10966

7 Experiments

As mentioned in implementation section, there are 3 versions of `getSimTuples` and 3 versions (ignoring the static partitioning one) of `getPredictedRatings`. The same version of `getSimTuples` and `getPredictedRatings` are tested together. Versions of functions:

Version	Implementation	getSimTuples	getPredictedRatings
0	Sequential	getSimTuples0	getPredictedRatings0
1	parList	getSimTuples1	getPredictedRatings1
2	parListChunk	getSimTuples2	getPredictedRatings2

7.1 Observations

Before implementing the program, I considered that the computation of item similarity between each pair of items will be the most time-consuming part. The time complexity of computing the similarity between each item is $O(|I|^2|U|^2)$, where $|I|$ is the number of items, and the $|U|$ is the number of users.

However, after observing the threadscope images and printing out the time consumed by each steps, I found out that predicting the ratings of the 1682 items is the most computationally heavy part of the entire procedure.

7.1.1 Amdahl's law

For the sequential version, the elapsed time is 40.419, and the time spent on computing the item similarity map and the ratings is 37.514143. Therefore, the ideal speedup that we can get with 8

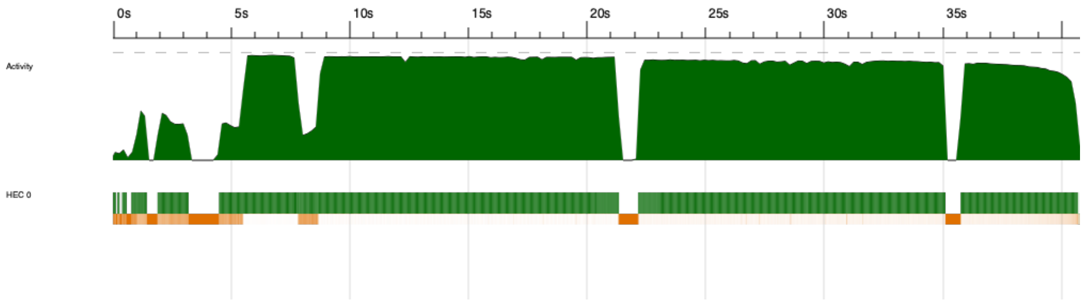


Figure 1: Threadscope image for sequential version, 1 core

cores is:

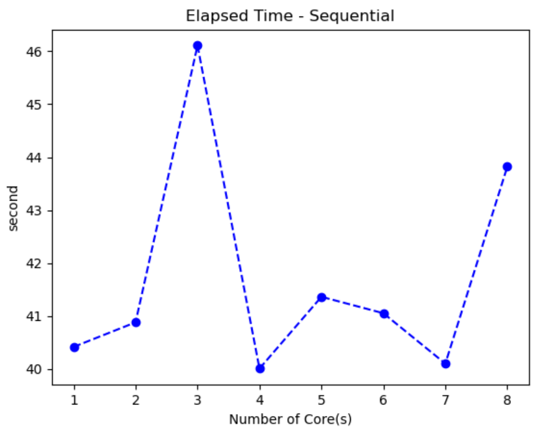
$$\frac{1}{(1-P)+\frac{P}{S}} = \frac{1}{(1-\frac{37.514143}{40.419})+\frac{37.514143}{8}} = 5.322 \tag{4}$$

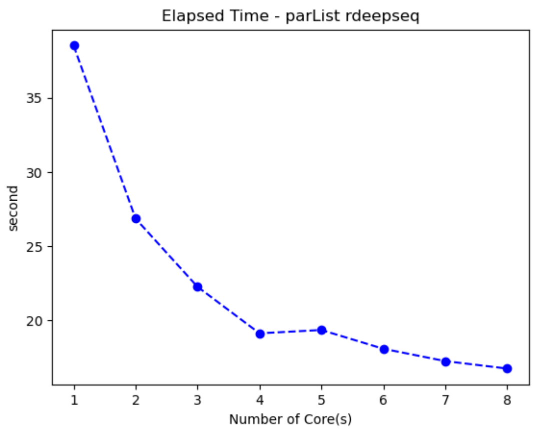
7.2 No Parallelism vs parList

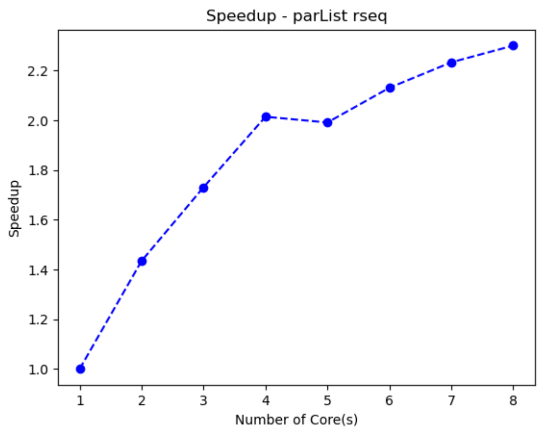
For the first two version (label 0 and label 1), the query is run on 3 different user for the same number of cores (1-8) . The ratings are all predicted with the top 3 items that are similar to the item in consideration. The number of sparks and elapsed time shown in the plots are averaged over the three samples.

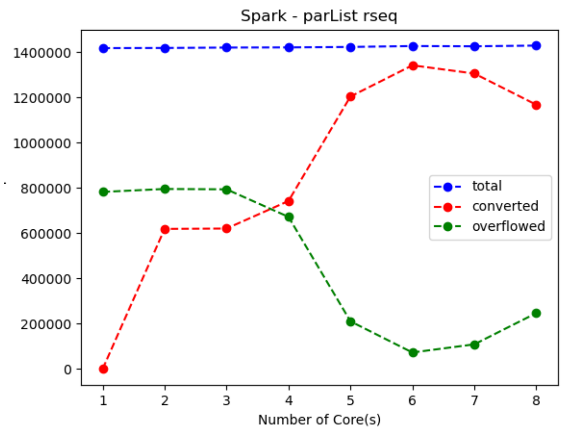
7.2.1 Elapsed Time and Speedup

It is visible that the performance of the version using `parList rdeepseq` is significantly improved. Also, from the plot for spark, we can see that as the number of cores increases, the number of converted sparks increases too. However, the best speedup is not even close to the ideal speedup computed.





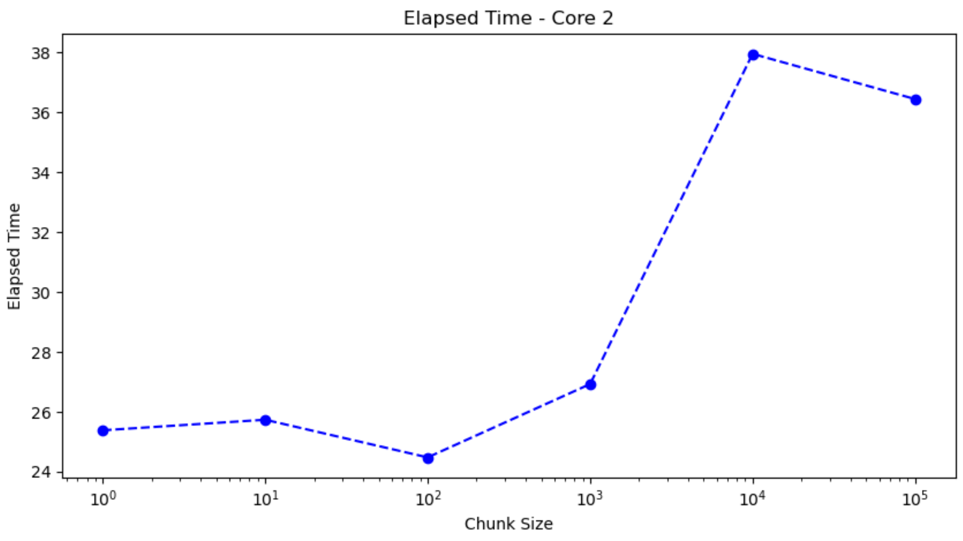


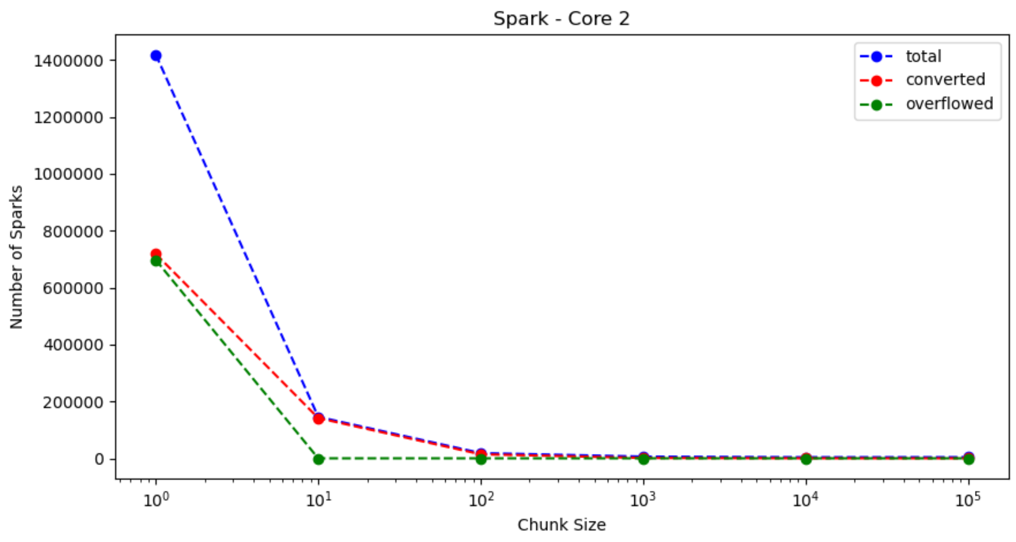


7.3 parListChunk

In addition to the parameters that are used for the first two versions, the parListChunk version uses an additional variable - chunkSize. The program is tested with chunk sizes - 1, 10, 100, 1000, 10000, 100000.

A very interesting observation is that this version only does the best when the chunk size is at 10 or 100. After looking at the output of threadscope, I think the explanation to this is that when the chunk size is too large, the load is becomes unbalanced among cores. As shwon in Figure 4, the second core is barely used when the chunk size is 100000.





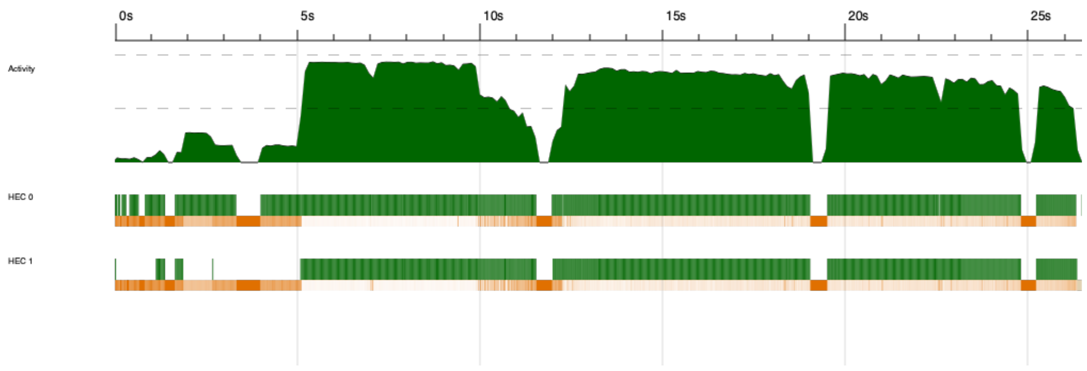


Figure 2: Threadscope image of the parListChunk version, 2 cores, 10 chunks

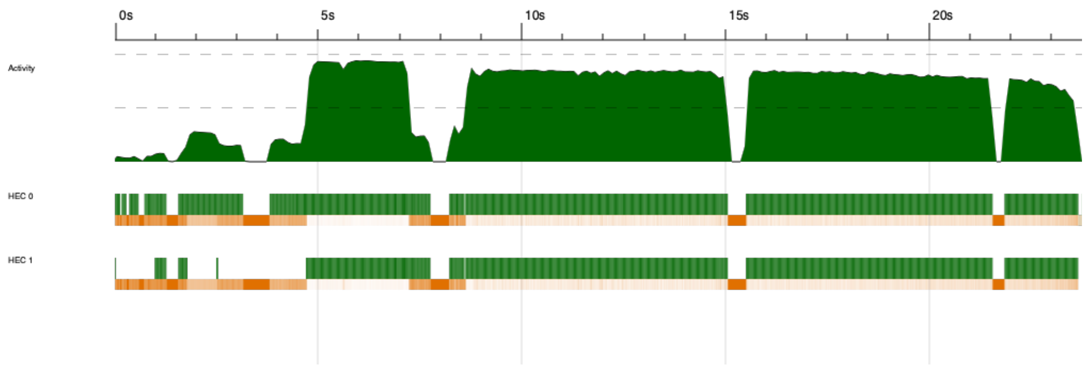


Figure 3: Threadscope image of the parListChunk version, 2 cores, 100 chunks

8 Conclusion and Future Work

In this project, I implemented a movie recommendation system based on item-based collaborative filtering. Three versions of the program is introduced and analyzed.

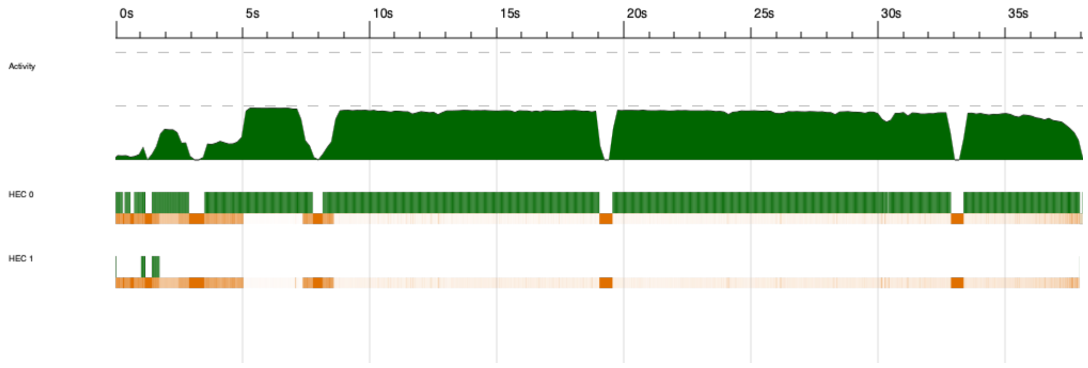


Figure 4: Threadscope image for parListChunk version, 2 cores, 100000 chunks

Besides speeding up the process of computing item similarities and predicting ratings, the speedup of IO and initializing data structures is also very worth exploring. For example, is there any way to solve the issue of initializing the Data.Matrix? Is it possible to read the raw data from the MovieLens dataset and initialize the Data.Matrix entry by entry quickly?

References and Resources

1. https://web.stanford.edu/class/cs124/lec/collaborativefiltering21.pdf

2. Item-based Collaborative Filtering :

   Build Your own Recommender System!: https://www.analyticsvidhya.com/blog/2021/05/item-based-collaborative-filtering-build-your-own-recommender-system/

3. Jure Leskovec, CS246 and J. Leskovec, A. Rajaraman, J. Ullman: Mining of Massive Datasets

4. G. Linden, B. Smith and J. York, "Amazon.com recommendations: item-to-item collaborative filtering," in IEEE Internet Computing, vol. 7, no. 1, pp. 76-80, Jan.-Feb. 2003, doi: 10.1109/MIC.2003.1167344.

5. MovieLens: https://grouplens.org/datasets/movielens/

6. MovieLens 1M Benchmark Leaderboard: https://paperswithcode.com/sota/collaborative-filtering-on-movielens-1m