Project Report: CS 240A – Applied Parallel Computing

K-Nearest Neighborhood based Music Recommendation System

Sivabalan Narayanan & Vivek Goswami

A. Introduction

Recommendation systems are an active topic in research and industry. The technique of collaborative filtering is especially successful in generating personalized recommendations. More than a decade of research has resulted in numerous algorithms, out of which, we have chosen K-Nearest Neighborhood (K-NN) model to predict the ratings for the songs. This model is an item-based algorithm which looks for neighbors among items (songs in this context) unlike user-based algorithms which look for neighbors among users. Our objectives for this project are as follows:

1. To improve prediction results using KNN algorithm.
2. Reduce overall execution time of KNN calculations by introducing parallelization
3. Find the optimum value of K(where K is the number of most similar neighbors chosen to predict the rating for a song for a given user)
4. Improve performance further by using multi cores.

B. K Nearest Neighborhood Algorithm

K Nearest Neighborhood Algorithm is a collaborative filtering algorithm. In order to predict rating for a new song “i” by user “u”, weighted average of ratings of songs common among the songs rated by user “u” and songs present in the neighborhood(k songs which are most similar to song “i”) of song “i”.

In order to find k nearest neighbors, we need to find the similarity between all pairs of songs. Once we have a similarity vector for song “i” having all other songs in the data-set in the similarity vector which contains the weight of their similarity. We can then sort in decreasing order and find k-most similar neighbors(they for set Ni). The formula for calculating the adjusted cosine similarity between a pair of song “i” and “j” rated by users in the set Uij(those users who have rated by song “i” and “j”), is given as follows, where wij is the similarity weight.

$$w_{ij} = \frac{\sum_{u \in U_{ij}} (r_{ui} - \bar{r}_u)(r_{uj} - \bar{r}_u)}{\sqrt{\sum_{u \in U_{ij}} (r_{ui} - \bar{r}_u)^2 \sum_{u \in U_{ij}} (r_{uj} - \bar{r}_u)^2}} .$$

These similarity weights are obtained for all other songs in the dataset with song “i”. Given k, we then create a neighborhood set Ni which has k most similar songs to song “i”.
Now, in order to predict the rating of song “i” by user “u”. We create a set Ru which contains tuples of type (song,rating) for all songs rated by user “u”.

Predicted rating for song “i” by user “u” is calculated by the following formulae:

$$\widehat{r}_{ui} = \frac{\sum_{j \in R_u \cap N_i} w_{ij} r_{uj}}{\sum_{j \in R_u \cap N_i} |w_{ij}|} .$$

C. Parallel KNN Algorithm

In this section, we discuss the implementation details about parallelizing the KNN algorithm. Figure 1, represents the entire workflow across modules for this algorithm.

I. System Workflow

The Yahoo! Music Data-set(training and validation) contains data in the following format:
UserId | Number of song(n)
song1 | rating
song2 | rating
…..... | …....
songn | rating

In order to process to perform neighborhood calculations, this data is modified by inverting the index of this data and indexing it on the basis of song. For this we created a module name *SongInvertedIndex* which takes the the yahoo music data-set as input and outputs the same data indexed on the basis of songs. The overall workflow is modularized as follows:

SongInvertedIndex:
-MapReduce task to calculated inverted index.
Input: Yahoo Music Dataset indexed on the basis of users.(Training Dataset)
Output: Yahoo Music Dataset indexed on songs.

InvertedKNN:
-MapRunner Task gets all pairs of songs for song "i" and feeds it to the mapper
-Mapper Task calculates the wij for all songs i and j which are related
Input: Output of *SongInvertedIndex*
Output: Neighborhood set (cardinal number K), for all songs in the dataset.

UserForwardIndex:
-Mapper Task calculates the set (Ru) for user "u"
Input: Yahoo Music Dataset indexed on the basis of users.(Training Dataset)
Output: Set Ru for all users in the dataset.

QueryMain:
-Mapper Task outputs User-Song pair as key and actual rating given by the user
as value
-Reducer Task takes in User-song pair to calculate the predicted rating for the song
Input: Output of InvertedKNN and UserForwardIndex for all user song pair. Songs for which prediction is performed are taken from Validation dataset.
Output : For all songs present in validation dataset, this module outputs a tuple (Actual Rating, Predicted Rating).

II. Accuracy of Ratings

We used Root Mean Squared Error in order to measure the accuracy of the implemented KNN algorithm. We developed another module *RMSEMain*, which runs a reducer task on the output data of *QueryMain* module and calculates root mean squared error using the following formulae:

$$RMSE = \sqrt{\frac{1}{|\mathcal{T}_1|} \sum_{(u,i) \in \mathcal{T}_1} (r_{ui} - \hat{r}_{ui})^2}$$

where, rui is the actual rating for song "i" and rui(^) is the predicted rating.

III. Optimization

In order to process large amount of data(~5.6 GB) containing information about 1 Million Users, the map reduce tasks developed in the above modules were optimized to increase performance and to reduce memory usage. We changed the chunk-size for mapper task in *InvertedKNN* to 2MB and this led to creation of 20 mapper tasks and thus increased performance. Heap Space size was increased to 8GB while running MapReduce task in *QueryMain* module. Incremental Garbage Collector was enabled in *QueryMain* module to eliminate occasional garbage collection pauses during the execution.

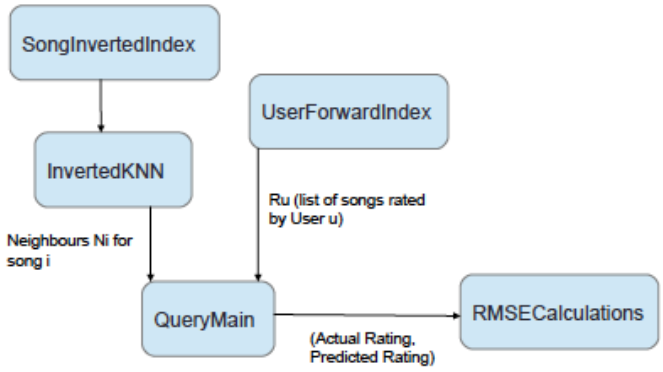

Figure 1: System Workflow

D. Performance Evaluation

In this section, we discuss performance of our implementation KNN algorithm when ran against the training dataset and validation dataset of track1 of music data-set released by Yahoo! for KDD-cup contest.

I. Dataset

We used the music data-set released by Yahoo for the KDD-cup contest. They released two datasets based on Yahoo! Music ratings. The Track1 dataset(larger dataset) comprises 262,810,175 ratings of 624,961 music items by 1,000,990 users collected during 1999-2010. Each item and each user have at least 20 ratings in the whole dataset. The available ratings were split into train, validation and test sets, such that the last 6 ratings of each user were placed in the test set and the preceding 4 ratings were used in the validation set.

II. SpeedUp

We executed our algorithm against 100K users from the validation dataset of track1. On increasing the number of cores from 1 to 2, we observed near perfect speedup(1.84). However, on increasing the number of nodes to 4, it the speedup increased but was restricted to 2.93. Using 8 cores, gave a speedup factor of 3.94. The following graph represents the increase in speedup factor on increasing the number of cores from 1 to 8.

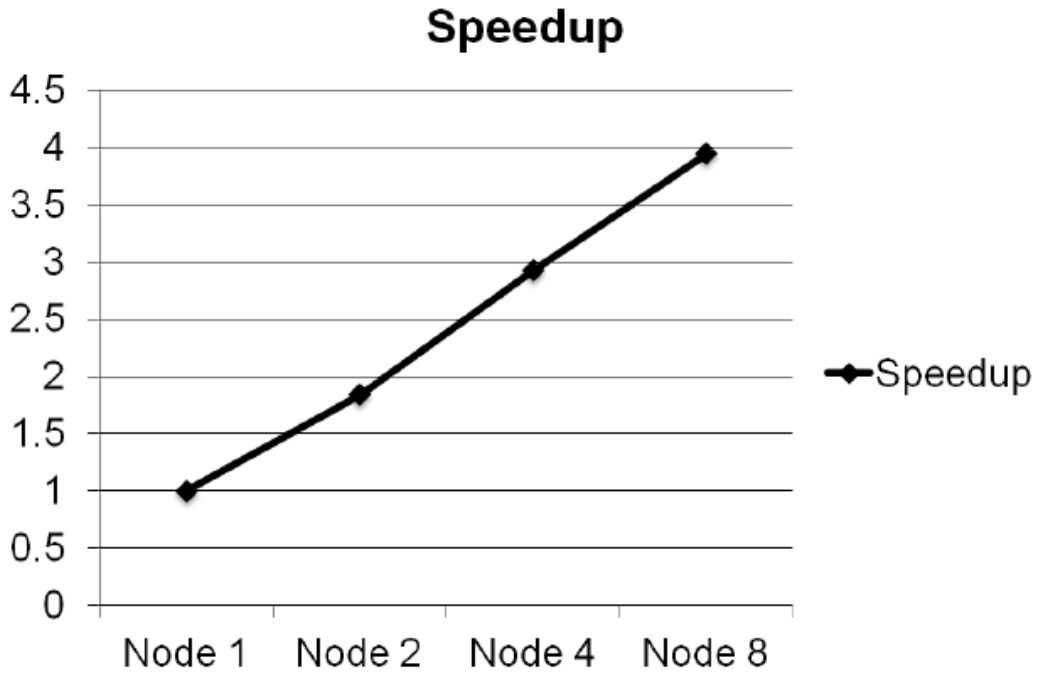

Table 1, reports the overall execution time along with time spent in each module for cores ranging from 1 to 8. From Table 1, we can easily conclude that the further parallelization efforts should be target towards *InvertedKNN* module since the time taken in this module is quite high has compared to other modules.

Number of Cores	***SongInverted Index*** **(Time taken)**	***UserForward Index*** **(Time taken)**	***InvertedKNN*** **(Time taken)**	***QueryMain*** **(Time taken)**	***RMSEMain*** **(Time taken)**	**Overall Running Time**	**Speedup**
1	3m 10.158s	47.546s	68m 58.15s	57.128s	24.091s	74.28m	1
2	2m 52.525s	44.102s	35m 26.66s	56.112s	24.031s	40.39m	1.84
4	3m 8.602s	50.187s	20m 3.91s	56.153s	23.982s	25.38m	2.93
8	3m 7.553s	51.122s	13m 30.41s	56.122s	24.030s	18.82m	3.94

Table 1: Overall Execution Time and Speedup

E. Conclusion

Working on this project was a great learning experience as well as helped us in realising the importance of how parallelization can reduce the running time drastically. We also, learnt to develop code which exploits parallelism using Hadoop MapReduce framework. Future work should be concentrated on exploiting parallelism in InvertedKNN module.

Acknowledgement

We would like to thank Maha Alabduljalil for her guidance during the course of this project.

References

[1] G. Dror, N. Koenigstein, Y. Koren, and M. Weimer, "The Yahoo! Music Dataset and KDD-Cup'11," In Proceedings of KDDCup 2011, 2011.

[2] Y. Wu, Q. Yan, D. Bickson, Y. Low and Q. Yang, "Efficient Multicore Collaborative Filtering", arXiv:1108.2580v2 [cs.LG] 17 Aug 2011.

[3] G. Takacs, I. Pilaszy, B. Nemeth, and D. Tikk., "Scalable collaborative Filtering approaches for large recommender systems", J. Mach. Learn. Res.,10:623-656, June 2009.