Movie Recommendation Using Map Reduce

Sarvdeep Singh Bindra
Rochester Institute of Technology
Email:ssb2339@rit.edu

Abstract—Collaborative Filtering (CF) algorithms are used very widely in recommender systems. Collaborative Filtering algorithms are computationally very intensive. This hinders the use of collaborative filtering algorithms in large-scale recommender systems.

In this paper I discuss about research investigation on collaborative filtering algorithms and their implementation using the power of parallel computing. I have implemented an application, which supports collaborative filtering algorithm and thus recommends movies using the Parallel Java 2 Library. I will present the sequential and the parallel algorithm used in the implementation. I will also present the data and analysis for strong scaling and weak scaling.

I. COMPUTATIONAL PROBLEM

Recommender systems recommend products on the basis of many factors. One of the factors is rating of the product by different users. Computing the similarity between users based on rating of the product can lead to recommendation of products based on preferences of the most similar user. Computing the similarity of users in a large dataset is computationally intensive and time consuming. In order to provide a good user experience, fast recommendations are a necessity. Hence using the ability of parallel computing to reduce the running time of the recommender system is important.

II. RELATED WORK

A. Research Paper 1

User-based Collaborative-Filtering Recommendation Algorithms on Hadoop[1] discusses the implementation of Collaborative Filtering algorithm on a hadoop cluster.

The author mentions and discusses the use of map reduce in collaborative filtering algorithms. In the data partitioning phase, the author partitions the data among the nodes evenly so that the nodes do not spend much time in initializing the resources. The nodes should spend more time in the computation process. In the map phase, the nodes read the input and perform the computation. This reduces the running time of the program. The author suggests that partitioning the data evenly across the cores will provide better scalability. In the performance analysis part, the author found that the with larger data sets the hadoop cluster could achieve ideal scalability.

In the reduce phase the results from the map phase would be collected, sorted and presented to the user.

Ideas from the Research Paper 1

The idea of evenly partitioning the data across the cores in my project and performing the computation on the cores was taken from the paper.

B. Research Paper 2


The author mentions and discusses the use of map reduce on a many-core machine. A many-core machine is the one, which has more than a dozen cores on it. The machine, which is used for the purposes of testing in the research paper, has 48 cores on it. In the map stage, each core performs computations on the chunk of data allotted to it. Each core generates a result, which is passed on to the next stage. In the reduce stage a final pair of result is generated and is passed on to the sort phase. In the sort phase, the input from the reduce phase is sorted as per the requirements and is presented to the user. The author makes use of the pearson correlation formula to compute the similarity between the users.

Ideas from Research Paper 2

The idea to implement the project on a single machine with multiple cores was taken from this research paper. A variant of the pearson correlation formula was used in measuring the similarity between users in the project.

\[
sim(x,y) = \frac{\sum_{i \in C_{xy}} (r_{xi} - \bar{r}_x)(r_{yi} - \bar{r}_y)}{\sqrt{\sum_{i \in C_{xy}} (r_{xi} - \bar{r}_x)^2} \sum_{i \in C_{xy}} (r_{yi} - \bar{r}_y)^2}
\]

https://en.wikipedia.org/wiki/Collaborative_filtering

C. Research Paper 3

Parallel Implementation of the Slope One Algorithm for Collaborative Filtering [3] discusses the parallel implementation of the slope one algorithm. It explains the multithreaded implementation and hybrid implementation of the slope one algorithm for collaborative filtering.

Multithreaded Implementation:

The pseudocode below [3] shows the step to perform the computation for the predicted weighted rating of a user for an item in a using multiple threads.

Ideas from Research Paper 3

A variant of the multithreaded implementation has been implemented in the research project. Instead of the slope one algorithm, the use of pearson correlation formula has been adopted.
Pseudocode of multithreaded implementation.

Main procedure (Input: The txt MovieLens file. Output: The predictions)
{
1. Initialize OpenMP routines;
2. All threads compute ratings, differences, frequencies and deviation matrices;
3. For i=0 to users x items, if (ratings[i][i]==0) then call predictions and weighted predictions function;
} (Weighted) Predictions Function (Weighted) Predictions()
{
1. Calculate (weighted) prediction of a given user’s rating for a given item;
2. Return (weighted) prediction;
}

III. IMPLEMENTATION

The project uses Parallel Java 2 Library [4] for implementation.

A. Sequential Algorithm and Design

1. Read the dataset and create the shared resources (ArrayLists).
2. Read the Input data (movie ID and ratings) and create the ArrayList for the input user.
3. Call the computeSimilarity method to compute the similarity between the input user and all the other users in the dataset.
4. In the computeSimilarity method decide the most similar user from the dataset.
5. After finding the most similar user set the hashmap of final movies to be recommended.
6. Sort the hashmap based on the ratings of the movies. If the ratings of the movies are same then sort the hashmap based on the movie ID.
7. Print the results.

B. Parallel Algorithm and Design

1. Read the dataset and create the shared resources (ArrayLists).
2. Read the input data (movie ID and ratings) and create the ArrayList for the input user.
3. Divide the shared resources among the cores on the machine.
4. Compute the similarity between the input user and all other users allotted to every core.
5. Find the most similar user per core.
6. Find the most similar user among the results provided by every core.
7. After finding the most similar user set the hashmap of final movies to be recommended.
8. Sort the hashmap based on the ratings of the movies. If the ratings of the movies are same then sort the hashmap based on the movie ID.
9. Print the results.

C. Developers Manual

The sequential and the parallel versions of the code are tested on the nessieserver. Before executing and testing the project the following configuration need to be done using the command line.

1. export CLASSPATH=:/var/tmp/parajava/pj2/pj2.jar
2. compile the source code using the command javac *.java

D. User Manual

The sequential version should be executed using the following command: java pj2 Sequential <FileName1> <FileName2>

The parallel version should be executed using the following command: Java pj2 cores=<K>SequentialSmp <FileName1> <FileName2>

1. <FileName1>: FileName1 is data.txt. This file contains the ratings provided by the users in the dataset, movie ID in the dataset and the user ID.
2. <FileName2>: FileName2 is one out of movies1500.txt, movies2000.txt, movies2500.txt, movies3000.txt, movies3500.txt, movies3800.txt.
IV. PERFORMANCE ANALYSIS

The strong scaling performance was measured for 6 different input sizes starting from 1500 movies. The running time for different input sizes remained similar with the increase in the number of cores. This indicated that I got non-ideal scaling.

In figure 1, the efficiency decreases with the increase in the number of cores. In figure 2, the speedup is approximately constant with the increase in the number of cores. In figure 3,
the running time shows an aberrant behavior with respect to increasing number of cores.

The weak scaling performance was measured with 4 different input sizes starting from 1500 movies. The running time for different input sizes and increasing number of cores were similar which indicates that the running time is ideal. On the other hand the speedup increases but not in proportion to the number of cores. Therefore efficiency is not ideal.

In figure 4, the efficiency decreases with the increase in the input size as well increase in number of cores. In figure 5, the running time is almost similar with the increasing number of cores. In figure 6, the speedup increases but is not in proportion to the number of cores.

The reason for the above mentioned non-ideal scaling and efficiency is, reading the dataset and setting up the shared resources (ArrayLists) takes up a lot of time, which affects the overall running time of the program. Due to this reason the speedup is not ideal and hence the efficiency is not ideal.

V. Future Work

The recommender system can be improved by considering more factors than just ratings. These factors could be sex of the user and age of the user.

The scaling and efficiency of the parallel version can be improved by parallelizing the process of reading the dataset and setting up the shared resources.

VI. Lessons Learnt

1. I understood the working of collaborative filtering algorithms.
2. I understood the working of the working of recommender systems used in the real world.
3. I learnt about the application of parallel computing to recommender systems.
4. I learnt that by parallelizing the process of data reading and setting up the shared resources, an ideal scaling and efficiency of the parallel recommender system could be obtained.

VII. Team Contributions

I worked alone on the project from deciding the project topic till presentation 4. I worked alone on the Team Deliverables.

VIII. References

[1] Zhi-Dan Zhao and Ming-Sheng Shang, User-based Collaborative filtering recommendation algorithms on hadoop, Knowledge Discovery and Data mining 2010, Third International Conference, 9th Jan -10th Jan 2010, 478-481.

