A Distributed Phoenix++ Framework for Big Data Recommendation Systems

Chongxiao Cao¹, Fengguang Song², Daniel G. Waddington³
Innovative Computing Lab, University of Tennessee at Knoxville¹
Computer Science Department, IUPUI²
Computer Science Lab, Samsung Research America³

Abstract

Recommendation systems are important big data applications that are used in many business sectors of the global economy. While many users utilize Hadoop-like MapReduce systems to implement recommendation systems, we utilize the high-performance shared-memory MapReduce system Phoenix++ to design a faster recommendation engine. In this paper, we design a distributed out-of-core recommendation algorithm to maximize the usage of main memory, and devise a framework that invokes Phoenix++ as a sub-module to achieve high performance. The design of the framework can be extended to support different types of big data applications. The experiments on Amazon Elastic Compute Cloud (Amazon EC2) demonstrate that our new recommendation system can be faster than its Hadoop counterpart by up to 225% without losing recommendation quality.

1. Introduction

Big data is sweeping into more sectors of the global economy due to innovation in computer systems, storage devices, sensor networks, mobile devices, as well as sophisticated capabilities to analyze data. A few important classic big data applications are: data mining, recommendation systems, enterprise analytics, and visualization [2]. In this paper, we focus on designing a distributed high-performance recommendation system.

Recommendation systems are typically used to predict a user's response given a list of options. Many companies such as Google, Amazon, Netflix, and the New York Times, have developed their own recommendation engines to offer customers options such as what products, books, or articles they might like. Recommendation systems can be divided into two broad categories: content-based systems and collaborative filtering systems [3]. Content-based systems recommend items that contain elements (e.g., text, image, movie star) similar to those the user liked in the past. Collaborative filtering systems, on the other hand, recommend items based

on “closeness” between users or between items. Although many other recommendation systems exist, the collaborative filtering systems so far have been thought of as the best recommendation systems [4], [5]. It requires users (or customers) to rate a relatively small subset of items. Based on the relationship between a subset of items, it can recommend an item that a user has not purchased, but is similar to what he or she liked in the past. Our work builds upon “item-based collaborative filtering recommendation algorithms” and supports both shared-memory and distributed-memory many-core systems.

Figure 1. Experiment of 1 GB word count using Phoenix++ and Hadoop. The y-axis is in a logarithmic scale.

Recommendation systems are often implemented with the MapReduce programming model due to the MapReduce's simplicity and robustness. Users can simply write some MapReduce jobs that contain a map function and a reduce function, then combine those jobs together and implement a recommendation system quickly [6], [7]. The most widely used MapReduce framework is Apache Hadoop [8], which supports distributed-memory clusters. However, researchers have also designed MapReduce systems on shared-memory multicore systems (e.g., Phoenix++ [1] from Stanford University).

In order to realize a high performance recommendation system, we first need to find the fastest MapReduce framework. Figure 1 compares the performance of Phoenix++ and Hadoop for 1GB word count running on Amazon Elastic Compute Cloud (Amazon EC2). Both experiments are

executed on one compute node (i.e., one worker node in Hadoop) with a different number of virtual CPUs. The result shows that Phoenix++ can outperform Hadoop greatly (note the log-scale in the y-axis). For instance, Phoenix++ is faster than Hadoop by 28.5× on four virtual CPUs, for 7.4 seconds versus 211 seconds.

Phoenix++ provides much better performance than Hadoop; however, it does not support clusters and assumes the input size is less than the main memory size. The goal of our work is to build a high performance recommendation system for both shared-memory and distributed-memory many-core systems. The new recommendation system utilizes Phoenix++'s in-memory processing capability to outperform Hadoop greatly. In our approach, we design a distributed out-of-core item-based collaborative filtering recommendation algorithm and build a framework that views and invokes Phoenix++ as a black box. Our new recommendation system can adapt to the following four types of scenarios: i) single node and in-core (i.e., data fit in memory), ii) single node and out-of-core, iii) multiple nodes and in-core, and iv) multiple nodes and out-of-core. More specifically, “in-core” and “out-of-core” mean whether the entire input can be stored in memory or not. Depending on a computing platform's memory size, our recommendation system is able to adapt to one of the four scenarios. The experimental results on an Amazon EC2 cluster show that our new recommendation system can achieve up to 225% speedup over the de facto standard machine-learning library of Apache Mahout [9].

This paper outlines the following contributions:

- A distributed out-of-core recommendation algorithm to maximize the usage of main memory.
- A new high-performance recommendation system that works on both shared-memory manycore systems and distributed-memory clusters. It outperforms Apache Mahout by up to 225%.
- A general methodology to extend shared-memory MapReduce frameworks such as Phoenix++ to distributed-memory clusters regardless of available memory sizes.

The rest of this paper is structured as follows. Section 2 explains the background for our work in more detail. Section 3 presents the design of our recommendation system. Section 4 shows the experimental results. Section 5 presents related work. Section 6 concludes our work and Section 7 summarizes the future work.

2. Background

This section briefly introduces the MapReduce programming model with two of its popular

implementations, and also introduces the item-based collaborative filtering (CF) algorithm.

2.1. MapReduce

The MapReduce programming model inherits the concept of *map* and *reduce* from functional programming languages to support parallel and distributed computing. Given a “daunting” task to develop large-scale parallel applications, now a user can simply provide a map and a reduce function to fulfill his or her goal quickly. The two functions are defined as follows:

Map: $(key1, value1) \rightarrow list(key2, value2)$

Reduce: $(key2, list(value2)) \rightarrow list(value2)$

A *map* function consumes an input pair of $\langle key1, value1 \rangle$ and produces a new set of intermediate $\langle key2, value2 \rangle$. Next, the runtime system groups all the intermediate *value2* based on the same *key2*. Finally, a *reduce* function takes as input an intermediate *key2* and its *value2* list, and produces an output [10]. The MapReduce model not only makes parallel programming much easier, but also automatically supports fault tolerance, parallelization, data distribution, and load balancing.

2.1.1. Hadoop implementation. Hadoop is an open-source software framework that is implemented in Java. A Hadoop cluster consists of a single master node and multiple worker nodes. The master node is responsible for scheduling the split tasks of a MapReduce job to the worker nodes, monitoring them, and re-executing the failed tasks. The worker nodes execute the tasks as directed by the master.

The Hadoop Distributed File System (HDFS) is a distributed, scalable, and portable file system in the Hadoop framework. It is designed for storing very large files with streaming data access patterns, running on clusters on commodity hardware [11]. Files in HDFS are split into blocks (64 MB by default), each block is replicated multiple times and distributed into different data nodes.

2.1.2. Phoenix++ implementation. Phoenix++ is a C++ implementation of the MapReduce programming model for shared-memory multicore systems. It allows users to write high-performance code easily, with scalability comparable to hand-coded pthreads solutions. The bottlenecks of the shared-memory MapReduce programs lie in the intermediate key-value data layout, memory allocation pressure, and framework overhead. To solve these bottlenecks, Phoenix++ provides a flexible intermediate key-value storage abstraction and an effective combiner to minimize the memory usage. Users can also adapt them to particular characteristics of a workload [1].

2.2. Item-based collaborative filtering (CF) algorithm

The Collaborative Filtering (CF) algorithm is motivated by the idea that people often get the best recommendations from someone with a similar taste. In a CF recommendation system, there is a list of m users $U = \{u_1, u_2, \dots, u_m\}$ and a list of n items $I = \{i_1, i_2, \dots, i_n\}$. Each user $u_i \in U$ gives some numerical ratings $r_{i,j} \in R$ for the corresponding item $i_j \in I$, where R is the rating set.

The CF algorithm has two forms: user-based CF algorithm and item-based CF algorithm. In this paper, we adopt the item-based CF algorithm to implement our recommendation system because it usually outperforms the user-based CF algorithm in large and sparse datasets [12].

The idea behind an Item-based CF algorithm is that a user is most likely to purchase items, which are similar to the one he previously bought. The algorithm has two steps:

Similarity Computation: It computes the similarity between items and selects the most similar items.

Prediction Computation: It computes the prediction by taking the weighted average of the target user's ratings on the most similar items. The basic idea to compute the similarity between item i and item j is to consider them as two user-rating vectors and then apply a vector-similarity computation to determine their similarity $S_{i,j}$. There are several ways to compute the similarity (e.g., cosine, Pearson correlation, adjusted cosine, and so on). Our work uses the Pearson correlation [5] to measure the similarity $S_{i,j}$ as shown in Equation 1. Here U denotes the users who have both rated item i and item j , $R_{u,i}$ denotes the rating of user u on item i , and $\bar{R}_i$ denotes average rating of the i -th item.

$$S_{i,j} = \frac{\sum_{u \in U} (R_{u,i} - \bar{R}_i)(R_{u,j} - \bar{R}_j)}{\sqrt{\sum_{u \in U} (R_{u,i} - \bar{R}_i)^2} \sqrt{\sum_{u \in U} (R_{u,j} - \bar{R}_j)^2}} \quad (1)$$

To predict user u 's rating on item i , all the items similar to i will be considered. The weighted sum method is used, where each rating is weighted by the corresponding similarity $S_{i,j}$ between items i and j . The prediction $P_{u,i}$ can be computed as follows [5]:

$$P_{u,i} = \frac{\sum_{\text{all similar items } N} (S_{i,N} * R_{u,N})}{\sum_{\text{all similar items } N} (|S_{i,N}|)} \quad (2)$$

3. Design of our recommendation system

In this section, we first introduce how to extend the item-based CF algorithm to support the out-of-core scenario and distributed memory clusters. Then we introduce the general methodology of using

Phoenix++ and the implementation of our recommendation system. Finally, we discuss potential issues of our recommendation system.

3.1. The extended item-based CF algorithm

3.1.1. Formulating the problem. The Item-based CF algorithm can be transformed to the form of matrix computation [7]. Let A be a $|U| \times |I|$ rating matrix, where U denotes a set of users and I denotes a set of items. In addition, let A_u denote the u -th row of matrix A and A_j denote the j -th column of matrix A . Each row A_u represents a user u , each column A_j represents an item j , and $A_{u,j}$ represents user u 's rating on item j .

Let S be a $|I| \times |I|$ similarity matrix and $S_{i,j}$ denote the similarity between item i and item j . $S_{i,j}$ is computed by using Pearson correlation in Equation 1 between column A_i and column A_j . In our approach, we overload the dot product operator in matrix multiplication with Pearson correlation. The similarity matrix S can be computed by the following matrix multiplication:

$$S = A^T A \quad (3)$$

Similarly, let P be a $|U| \times |I|$ prediction matrix and $P_{u,i}$ denote the prediction of user u on item i . We also overload the dot product operator in matrix multiplication with the weighted sum method in Equation 2, after which P can be computed by the rating matrix A multiplying the similarity matrix S :

$$P = AS \quad (4)$$

The entire item-based CF recommendation algorithm can be represented as two matrix multiplications:

$$P = A(A^T A) \quad (5)$$

3.1.2. Distributed memory cluster support.

After modeling the item-based CF algorithm as matrix multiplications, the next step is to distribute the computation to different compute nodes to execute in parallel. Figure 2 explains how our parallelization is implemented. The rating matrix A is divided into different column panels A_i evenly and distributed to every compute node i . Every compute node i executes two steps of computation based on A_i simultaneously. In the step of similarity computation, compute node i computes similarity matrix panel S_i . In the step of prediction computation, compute node i computes prediction matrix panel P_i . Because every compute node produces its own similarity matrix panel needed for prediction, there is no global barrier for all compute nodes between these two steps.

Figure 2. Extending the item-based CF algorithm to support distributed cluster

3.1.3. Out-of-core support. After dividing the work into different compute nodes, the data size on each node may still be larger than the memory size, which makes in-memory computation infeasible. Figure 3 shows how we deal with the out-of-core cases.

Figure 3. Extending the item-based CF algorithm to support the out-of-core scenario

This strategy can be adapted in both steps of similarity computation and prediction computation. Assume the input is matrix A and matrix B , and the output is matrix C , also their total size is larger than the memory size. Matrix A is partitioned into m row strips by the block size $block_row$, while matrix B is partitioned into n column strips by the block size $block_col$. Each computation reads one row strip from matrix A and one column strip from matrix B into memory, produces a $block_row \times block_col$ block in matrix C and then writes back to disk. After $m \times n$ times of in-memory computations, the computation for matrix C is completed.

3.2. General methodology of using Phoenix++

Phoenix++ can be viewed as a black box to develop MapReduce applications. To develop a

specific MapReduce application, a user only needs to inherit the basic classes in Phoenix++ and provides four customized functions: *split()*, *map()*, *sort()* and *reduce()*.

The *map()* and *reduce()* functions are the same as those in the MapReduce programming model. The *split()* function is defined to assign portions of input to each map task. Additionally, in shared-memory system, sorting at the merge phase can cause significant performance overhead. Thus, the user can decide to enable or disable sorting in Phoenix++. If sorting is enabled, the user can provide a custom *sort()* function that compares key-value pairs, rather than only the keys.

3.3. Recommendation system implementation

We implement the in-memory MapReduce computation with Phoenix++. There are two typical ways to compute matrix multiplication in MapReduce:

- **Inner product:** Each map task computes an element of the output matrix based on the inner product of a row from the left matrix and a column from the right matrix. No reduce tasks are needed.
- **Outer product:** The i -th map task computes the outer product of the i -th column of the left matrix and the i -th row of the right matrix. Each reduce task simply sums up all the partial results.

The inner product method is highly efficient as no reduce task is required, which obviates the overhead for shuffling intermediate keys and values. We implement the similarity computation and the prediction computation as two MapReduce jobs using the inner product method with Phoenix++. Each map task computes 100×100 elements of the output matrix. This computation size can be further tuned to attain a better performance.

Considering the rating matrix A is very sparse, in order to accelerate matrix computation, we store it using compressed sparse column (CSC) format and compressed sparse row (CSR) format.

Our recommendation system is built on Network File System (NFS), the rating matrix A is copied to each compute node as means to minimize I/O in NFS.

The components of our recommendation system are as follows (see Figure 4):

- The input file is a rating history, which records users' ids, items' ids and corresponding rating points.
- The scripts (written in Python) are responsible for the preprocessing of the input file, including partitioning the input file based on the number of available compute nodes, transforming it into the sparse matrix format and sending it to each compute node.

- The recommendation engine runs on a distributed cluster, it takes the following three steps to calculate the predicted ratings: 1) The first step is similarity computation. It computes a partial similarity matrix on every compute node. 2) The second step is sorting. It sorts elements in every column of partial similarity matrix S_i , and prunes the elements that are less than the similarity threshold. The most similar items are selected based on a neighborhood size after sorting. 3) The final step is to compute the potential ratings of unrated items for every user on every compute node.
- The output file represents the prediction of users' potential ratings, which shows users' ids, items' ids and corresponding predicted rating points.

Figure 4. Recommendation system components.

The recommendation engine is implemented in C++ using MPI to realize parallelism on distributed memory cluster and Phoenix++ to execute in-memory MapReduce computation. Figure 5 shows the design of the recommendation engine. *SimMR* and *PtrMR* are two MapReduce classes in the engine, which are inherited from basic class in Phoenix++. Only *split()* and *map()* functions are provided in *SimMR* and *PtrMR*. The *split()* function divides input matrices into row strips and column strips, and assigns to every map task. The *map()* function implements inner product computation. The sort step prunes those dissimilar items as explained earlier.

Figure 5. Recommendation engine design.

3.4. Potential issues

3.4.1. Storage efficiency. For the purpose of minimizing the I/O communication in NFS, we send the input matrix to every node, which makes the storage inefficient in the disk. Utilizing a parallel file system is a feasible solution to this problem in the future.

3.4.2. Load balance. Currently, data is divided into column panels evenly, so the scalability of our recommendation system is not close to optimum if ratings are not uniformly distributed in every column. Those nodes receiving popular rated items would perform more computation than others and become the bottleneck. A fine-grained data distribution method like 1-D column cyclic distribution is a feasible way to achieve load balance on in our future work. Figure 6 shows the data distribution differences when using four compute nodes. With 1-D column cyclic distribution, each compute node can receive nearly the same amount data for computation.

Figure 6. Data distribution optimization

4. Experimental results

In this section we conducted experiments on Amazon EC2 to measure the prediction quality and performance of our recommendation system (we call it *dist-phoenix++*). We also compared our system with the item-based CF algorithm in Apache Mahout. Both real-world datasets and randomly generated datasets were tested.

4.1. Apache Mahout

Mahout is an open source machine learning library from Apache. It is written in Java and primarily focused on recommendation engines, clustering, and classification. Its core algorithms are implemented on top of Hadoop so that it can scale on large datasets [9]. The item-based CF algorithm implemented in Mahout is compared with our *dist-phoenix++*.

4.2. Experiment setup

Our experiments were conducted on Amazon EC2. We used one *m1.xlarge* instance as one compute node, which is a 64-bit virtual machine with four virtual CPUs and 15 GB memory.

We built an eight-nodes cluster for *dist-phoenix++*. NFS was used as file system and MPI version was MPICH2.

For experiments with Mahout, we used the existing Amazon Elastic MapReduce (Amazon EMR) directly. The Hadoop version on Amazon

EMR is 1.0.3. We allocated eight *m1.xlarge* instances as compute nodes and one *m1.xlarge* instance as the master node. Amazon Simple Storage Service (Amazon S3) was the file system used in this Hadoop cluster.

4.3. Prediction quality

To test the quality of recommendation system, we used the MovieLens 100k dataset [13]. This dataset contains 100,000 ratings by 943 users on 1682 movies. Both Mahout and our dist-phoenix++ system use Pearson Correlation as the similarity measurement, where the similarity threshold is 0.01. The prediction quality is measured by *Mean Absolute Error* (MAE):

$$MAE = \frac{\sum_{i=1}^N |p_i - q_i|}{N} \quad (6)$$

Here p_i is the predicted value and q_i is the real value. In Figure 7, 80% of the data was used as a training set and 20% of the data was used as a testing set. Total five different cases were tested. The default recommender in Mahout has the worst accuracy, because, by default, Mahout sets the similarity neighborhood size to 100 and only 10 items are considered for every user when doing prediction. In our dist-phoenix++ recommendation system, we set the similarity neighborhood size to the total number of items and all the items a user has rated will be considered during prediction. The MAE of dist-phoenix++ varies from 0.82 to 0.85. After tuning Mahout with the same setting as dist-phoenix++, its accuracy improves substantially to nearly 0.77.

Figure 7. Prediction quality on MovieLens 100k dataset

In the following performance experiments, for both dist-phoenix++ and Mahout, we use 100 as the similarity neighborhood size and 0.01 as the similarity threshold.

4.4. Single compute node performance

We first compare the performance on a single compute node when the data size is small enough to

fit into memory. All 100,000 ratings in MovieLens 100k dataset were used as input. The jester-3 dataset was also selected, which contains 24,938 users' ratings on 101 jokes [14]. In dist-phoenix++, all the data was loaded into memory only once and computed by Phoenix++. As shown in Figure 8, dist-phoenix++ outperforms Mahout greatly due to the advantage of in-memory computation.

Note that both Mahout tests ran for more than 10 minutes. This is because the launching overhead of one MapReduce job on Amazon EMR takes around 1 minute. And the item-based CF algorithm in Mahout contains 10 MapReduce jobs. This overhead can be reduced as the input data size increases.

Figure 8. Performance on single compute node when data can fit into memory

Figure 9. Performance on single compute node for MovieLens 10M dataset when block size changes

To investigate the impact of disk I/O on the overall performance, we varied the size of the data that can be loaded into memory. The dataset we used is MovieLens-10M, which contains 10,000,054 ratings applied to 10,681 movies and 71,567 users [13]. As shown in Figure 9, 100% means all the input data is loaded into memory once, 50% means each time 50% of input data is loaded into memory, and so forth for 25% and 12.5%. When disk I/O increases, the execution time of dist-phoenix++ increases slightly, and all of them are much faster than Mahout.

4.5. Multiple compute nodes performance

To investigate the multiple compute nodes performance of our recommendation system, we increased the number of compute nodes from 1 to 8.

Also, we simply copied MovieLens-10M dataset 10 times to get a 100M ratings dataset applied to 10,681 movies and 715,670 users. The out-of-core block sizes *block_row* and *block_col* mentioned in section 3.1.3 were set to 2,000 and 1,000 in similarity computation, while 100,000 and 1,000 in prediction computation.

Figure 10 and Figure 11 show the running time and scalability of dist-phoenix++ versus Mahout. For 8 compute nodes, the running time of dist-phoenix++ is 1.07 hours, compared with 2.4 hours in Mahout, it gets a 2.25 $\times$ speedup.

However, the scalability of dist-phoenix++ is not as good as Mahout. It has a 2.82 $\times$ speedup when increasing the number of compute nodes from 1 to 8. We believe this is caused by the load balancing issue explained in Section 3.4. In the MovieLens dataset, popular rated movies are associated with a small number of items and are distributed to the first compute node. This makes the first node execute more computation than the other nodes and hence become the bottleneck.

Figure 10. Running time of dist-phoenix++ and Mahout on 100M ratings dataset

Figure 11. Speedup of dist-phoenix++ and Mahout on 100M ratings dataset

To further investigate the scalability of our dist-phoenix++, we created a uniformly distributed 50M ratings dataset. This dataset was randomly generated and contained 10,000 items and 500,000 users. Ratings were evenly distributed such that each item had 5,000 ratings. The out-of-core block sizes were the same as the previous test. Figure 12 shows that dist-phoenix++ has a linear speedup. With 8 compute nodes, dist-phoenix++ achieves a speedup of 7.44 $\times$.

Figure 12. Speedup for a growing number of compute nodes for dist-phoenix++ on a 50M ratings uniformly distributed dataset

5. Related work

There are several shared-memory MapReduce libraries optimized for multicore architectures. Phoenix++ has demonstrated good performance comparable to hand-tuned parallel programs using pthreads [1]. Metis, from MIT, implements a new data structure (i.e., hash table and B+ tree) to group intermediate key/value pairs to provide high performance [15]. Tiled-MapReduce extends the MapReduce model with a tiling algorithm [16]. However, these libraries only work on shared-memory machines and assume the input must fit in the main memory. Our work extends the shared-memory library to support distributed-memory clusters.

Spark is a tool developed at the University of California at Berkeley to support in-memory cluster computing [17]. It provides a number of primitives to allow users to perform data mining efficiently. The primitives include not only map, reduce, but also union, join, filter, sample, and so on. Spark is often applied to iterative data mining applications. It works on distributed-memory clusters but is not designed to handle disk I/O for intermediate results automatically.

Twister is another widely used MapReduce runtime system that supports iterative MapReduce computations [18]. It first reads data from disks to local memories on distributed nodes, then starts to compute iteratively, and finally writes the output to disk. It is possible to extend Twister to store large intermediate results in local disks instead of buffering in memory. Unlike Spark and Twister, our approach is designed to scale up within a single node first, then scales out to many nodes.

There has also been several work on implementing a recommendation system in MapReduce: Schelter et al. presented a recommendation system based on a scalable similarity-based neighborhood method [7]. Jiang et al. presented a scaling-up Item-based CF algorithm using four MapReduce jobs [6]. However all of those

implementations used Hadoop, whereas our work implemented the recommendation system in MapReduce with Phoenix++ on distributed-memory clusters.

6. Conclusion

In this paper, we proposed a method to utilize shared-memory multicore MapReduce systems to implement a distributed high-performance recommendation system. It is inspired by the fact that Phoenix++ can be faster than Hadoop by 28.5 times on Amazon EC2 to run word count. To utilize the sub-module of Phoenix++, we extend the item-based collaborative filtering recommendation algorithm to support both distributed-memory clusters and out-of-core computations. The experimental results show that the recommendation quality of our new system is comparable to that of Apache Mahout, while the performance is faster than Mahout by up to 2.25 times on eight compute nodes. The reason we can achieve better performance is because the shared-memory MapReduce system is highly optimized to conduct in-memory computations and can reduce the number of I/O operations greatly.

7. Future work

We have proposed a new approach for utilizing Phoenix++ to build a high-performance recommendation system. The same approach can be used to build other types of big data applications. Our future work is to make the framework more general so that different application developers can use the framework as a library.

8. References

- [1] J. Talbot, R. M. Yoo, and C. Kozyrakis, "Phoenix++: Modular mapreduce for shared-memory systems," in Proceedings of the second international workshop on MapReduce and its applications, 2011, pp. 9-16.
- [2] J. Manyika, M. Chui, B. Brown, J. Bughin, R. Dobbs, C. Roxburgh, and A. H. Byers, "Big data: The next frontier for innovation, competition, and productivity," McKinsey Global Institute, June 2011.
- [3] A. Rajaraman and J. D. Ullman, Mining of Massive Datasets. New York, NY, USA: Cambridge University Press, 2011.
- [4] D. Almazro, G. Shahatah, L. Abdulkarim, M. Khrees, R. Martinez, and W. Nzoukou, "A survey paper on recommender systems," CoRR, vol. abs/1006.5278, 2010.
- [5] B. Sarwar, G. Karypis, J. Konstan, and J. Riedl, "Item-based collaborative filtering recommendation algorithms," in Proceedings of the 10th international conference on World Wide Web, 2001, pp. 285-295.
- [6] J. Jiang, J. Lu, G. Zhang, and G. Long, "Scaling-up item-based collaborative filtering recommendation algorithm based on hadoop," in Proceedings of the 2011 IEEE World Congress on Services, 2011, pp. 490-497.
- [7] S. Schelter, C. Boden, and V. Markl, "Scalable similarity-based neighborhood methods with mapreduce," in Proceedings of the sixth ACM conference on Recommender systems, 2012, pp. 163-170.
- [8] Hadoop. The Apache Software Foundation. <http://hadoop.apache.org/>, (Access date: 2013-08-02).
- [9] Apache Mahout. The Apache Software Foundation. <http://mahout.apache.org/>, (Access date: 2013-08-03).
- [10] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation, 2004, pp. 137-150.
- [11] T. White, Hadoop: The Definitive Guide, 1st ed. O'Reilly Media, Inc., 2009.
- [12] T. Segaran, Programming collective intelligence, 1st ed. O'Reilly, 2007.
- [13] Movielens Data Sets. GroupLens Research. <http://www.grouplens.org/node/73>, (Access date: 2013-08-04).
- [14] K. Goldberg. Jester Collaborative Filtering Dataset. <http://goldberg.berkeley.edu/jester-data/>, (Access date: 2013-08-04).
- [15] Y. Mao, R. Morris, and M. F. Kaashoek, "Optimizing mapreduce for multicore architectures," in Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Tech. Rep, 2010.
- [16] R. Chen, H. Chen, and B. Zang, "Tiled-mapreduce: optimizing resource usages of data-parallel applications on multicore with tiling," in Proceedings of the 19th international conference on Parallel architectures and compilation techniques, 2010, pp. 523-534.
- [17] Spark. UC Berkeley AMPLab. <http://spark-project.org/>, (Access date: 2013-08-05).
- [18] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S. H. Bae, J. Qiu, and G. Fox, "Twister: a runtime for iterative mapreduce," in Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing, 2010, pp. 810-818.