DCF: A Dataflow-based Collaborative Filtering Training Algorithm

Xiangyu Ju[1], Quan Chen[1], Zhenning Wang[1], Minyi Guo[1], and Guang R. Gao[2]

[1] Shanghai Jiao Tong University, Shanghai 200240, China,
[2] University of Delaware, Newark 19716, USA
juxiangyu@sjtu.edu.cn

Abstract. Emerging recommender systems often adopt collaborative filtering techniques to improve the recommending accuracy. Existing collaborative filtering techniques are implemented with either Alternating Least Square (ALS) algorithm or Gradient Descent (GD) algorithm. However, both of the two algorithms are not scalable because ALS suffers from high computation complexity and GD suffers from severe synchronization problem and tremendous data movement. To solve the above problems, we proposed a Dataflow-based Collaborative Filtering (DCF) algorithm. More specifically, DCF exploits fine-grain asynchronous feature of dataflow model to minimize synchronization overhead; leverages mini-batch technique to reduce computation and communication complexities; uses dummy edge and multicasting techniques to avoid fine-grain overhead of dependency checking and reduce data movement. By utilizing all the above techniques, DCF is able to significantly improve the performance of collaborative filtering. Our experiment on a cluster with one master node and ten slave nodes show that DCF achieves 23x speedup over ALS on Spark and 18x speedup over GD on Graphlab in public datasets.

Keywords: DCF; dataflow; collaborative filtering; gradient descent; asynchronous; fine-grain; parallel;

1 Introduction

With the immense growth of e-commerce and social network, recommender systems have shown increasing popularity and importance in recent years. In order to achieve reasonable and user-specific recommendations, emerging recommender systems often adopt collaborative filtering technology that extracts user characters from multiple aspects and makes recommendation by synthesizing all the characters. Collaborative filtering can be implemented with either memory-based techniques (e.g user based technique, item based technique) [7] or model-based techniques (e.g., ALS, and GD) [3]. Prior work [15, 6] has shown that collaborative filtering system is able to achieve better prediction accuracy adopting model-based techniques. To this end, well-known big data platforms (e.g. Hadoop [1], Spark [18], Graphlab [17]) leverage model-based techniques, such as ALS and GD, to implement collaborative filtering.

However, recommender systems based on collaborative filtering suffer from poor performance in big data area. This is mainly because the volume of rating data from massive users increases dramatically but ALS and GD perform

poor when the data set is large. For ALS, its computation complexity increases with feature dimension quadratically [12] and this feature of ALS results in its poor scalability. For GD, although it has a much lower computation complexity compared with ALS and outperforms ALS when the dataset is large [19, 8, 12], it still suffers from poor performance due to the following challenges. **First, GD suffers from high communication complexity and the resulted network load becomes the new performance bottleneck. Second, individual operations of each data require fine-grain task scheduling and the scheduling overhead degrades the performance. Third, GD is not able to fully utilize computing resources due to the high data sparsity.**

In order to solve the above challenges, we propose a *Dataflow-based Collaborative Filtering* (DCF) algorithm to improve the performance of distributed recommender systems. Dataflow [9] is a parallel computation model that supports asynchronous fine-grain task scheduling. Prior work [21] has shown that dataflow model is able to minimize communication complexity, reduce synchronization overhead, and increase resource utilization, especially in high data sparsity scenario. However, simply implementing GD with dataflow model is not able to improve the performance of collaborative filtering, because the fine-grain dependency checking overhead and frequent data movement introduced by dataflow model becomes the new bottlenecks. To this end, we further propose two optimizations (*dummy edge* and *multicasting*) to minimize the extra overhead caused by dataflow model, and apply *mini-batch* technique to optimize the performance of GD. More specifically, dummy edge technique exploits local data transmission in dataflow graph to reduce dependency checking overhead and unnecessary data movement. Multicasting technique avoids sending data to same node for reducing data movement. Mini-batch limits the dependency number of each feature vector to further reduce computation and communication complexities on algorithmic level. The main contributions of this paper are summarized as follows:

- We design a *Dataflow-based Collaborative Filtering* (DCF) algorithm that performs better than traditional GD-based collaborative algorithms.
- We propose a dummy edge technique that handles fine-grain overhead of data dependency checking and reduce unnecessary data movement by exploiting data locality.
- We propose a multicasting technique that reduces the quantity of data transferred by network.
- We apply an mini-batch technique that reduces computation and communication complexity of DCF algorithm.

We have implemented DCF and evaluated it with public datasets. The experiment results demonstrate that DCF shows a 23x speedup over ALS on Spark and 18x speedup over GD on Graphlab with similar accuracy.

2 Background & Motivation

Model-based collaborative filtering algorithms are adopted by most of emerging distributed recommender systems. As shown in Figure 1, model-based collabo-

rative filtering algorithms convert a recommendation issue into a matrix factorization problem, transforming user-item-rating data into two separated matrixes which represent user features and item features. Predicted ratings are made by the products of user feature vectors and item feature vectors. ALS and GD are two main training algorithms for model-based collaborative filtering.
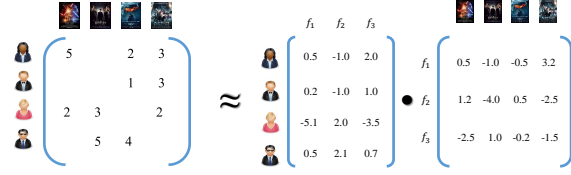

Fig. 1. Model-based Collaborative Filtering

2.1 Alternating Lest Square & Gradient Descent

ALS [20] converts the matrix factorization problem into a quadratic problem which can be optimally solved. According to the study of Gemulla et al. [12], the computation complexity of ALS is $O(k^3 + k^2(n_u + n_v) + kn_r)$ in one iteration, and it increases with feature dimension k quadratically because user number n_u or item number n_v is much larger than k. ALS shows low performance in large scale datasets because of the high computation complexity.

GD is an alternative training algorithm for model-based collaborative filtering. According to Eq. 1 and Eq. 2, user feature vector u_i or item feature vector v_j can be updated by given ratings $r_{i,j}$ and dependent feature vectors where λ represents the regularization parameter. The computation complexity of GD is $O(k\bar{d}(n_u + n_v))$ in one iteration where $\bar{d}$ represents the average dependency number. Compared with ALS, GD has a lower computation complexity. However, GD also faces several challenges. Firstly, it requires fine-grain operations because each user feature vector u_i or item feature vector v_j can be updated individually and asynchronously. Secondly, high data sparsity causes load imbalance because dependency number of each feature vector varies in large scale. Thirdly, the communication complexity is $O(k\bar{d}(n_u + n_v))$ and it leads to high network load.

$$u_i = u_i - \alpha(\sum_{r_{i,j} \neq 0}(u_i^T v_j - r_{i,j})v_j + \lambda u_i) \tag{1}$$

$$v_j = v_j - \alpha(\sum_{r_{i,j} \neq 0}(u_i^T v_j - r_{i,j})u_i + \lambda v_j) \tag{2}$$

In order to develop a high performance collaborative filtering algorithm, we select GD as the training algorithm for its low computation complexity. In addition, dataflow model is selected to solve the challenges of GD.

2.2 Dataflow

Dataflow [9] is a data-driven computation model optimized for the execution of fine-grain parallel algorithms. Dataflow treats each computation as a separate task and the execution order is maintained by data dependencies. Dataflow model can be represented as directed acyclic graph (DAG) where nodes represent computation and edges represent data dependencies.
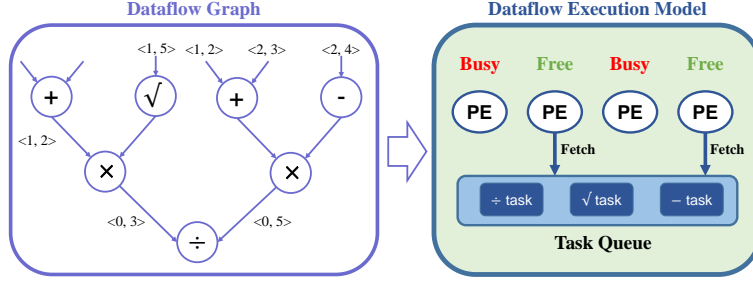
Fig. 2. Dataflow Execution Model

According to the work of Zuckerman et al. [21], the features of dataflow model can solve the challenges of GD well. Fine-grain feature supports individual updating operations of each feature vector. Asynchronization supports sparse dependencies well for fully utilizing computing resources. However, applying dataflow model to GD is not straightforward because we need to solve the challenges of dataflow model as well. Fine-grain dependency checking overhead and frequent data movement are the main bottlenecks because of massive rating data. As a result, we propose DCF algorithm with optimizations to solve the challenges of GD and dataflow model.

3 DCF: Dataflow-based Collaborative Filtering Algorithm

In this section, we give a GD algorithm based on dataflow execution model. Firstly, we demonstrate the working mechanism of dataflow execution model. Secondly, we transform GD into dataflow graph and explain the working process of it. Thirdly, we propose optimizations to solve the challenges of dataflow model and GD. Finally, we give a detailed DCF algorithm by applying proposed optimizations.

3.1 Dataflow Execution Model

In order to design dataflow algorithm, we introduce the working process of dataflow execution model. As shown in Figure 2, we give an example to explain dataflow execution model explicitly. The figure shows a snapshot of a simple computing progress which consists of several basic operator tasks. Dataflow execution model schedules and processes tasks generated by dataflow graph. Dataflow graph is a DAG which edges represent data dependencies and nodes represent functions. The basic data structure transferred in dataflow graph is key-value pair. A task is composed of data from input edges with same key and the function of current node. Once dependent data of one node are satisfied, corresponding tasks will be generated and pushed to the task queue, such as $-$ task, $\sqrt{}$ task and $\div$ task. As the input data of $+$ node have different keys, the task will be not generated until dependent data with same key arrive. Processing elements (PE) will fetch new tasks in task queue until tasks are finished.

3.2 DCF Dataflow Graph

In order to design DCF algorithm, we need to transform the algorithm into dataflow graph expression. As shown in Figure 3, we construct dataflow graph of GD according to Eq. 1 and Eq. 2. Node R transforms the input data into the triplet form $< user\ id,\ item\ id,\ rating >$, and sends user id and item id information to $InitU$ and $InitV$ for initializing feature vectors. Meanwhile, node R sends rating data to U_i and V_i for gradient computing, where U_i and V_i represent user node and item node in the ith iteration. After initializing feature vectors, $InitU$ and $InitV$ send feature vectors to U_1 and V_1. In each iteration i, U_i and V_i are used to compute the gradients and update feature vectors until convergence. After the gradient computing tasks are finished in last iteration n, U_n and V_n output the trained feature vectors.
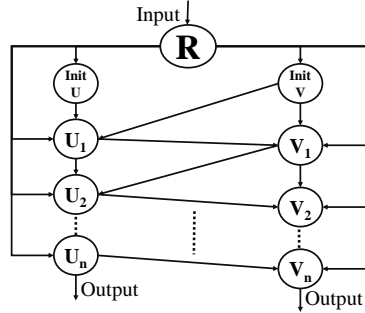

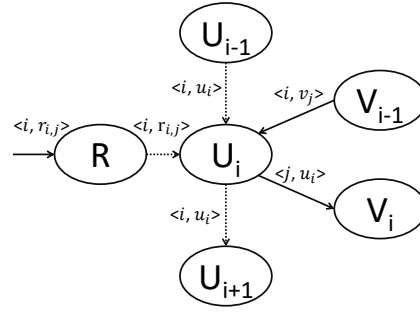**Fig. 3.** Dataflow Graph of Gradient Descent

Fig. 4. Dummy Edge Technique on User Node

By using dataflow model to implement GD, fine-grain operations and data sparsity can be easily handled. The fine-grain operations of GD are the gradient computing tasks of feature vectors. Fine-grain feature of dataflow model supports these tasks to be scheduled and executed individually. Due to the high data sparsity, the computation load of each feature vector varies in large scale. Asynchronization feature of dataflow model avoids unnecessary barrier time caused by sparse data, and better utilizes the computing resources.

Due to the deterministic feature of dataflow model, DCF transforms original GD algorithm into dataflow graph without changing the algorithm logic. Hence, DCF shows the same convergency performance with other GD algorithms. Though dataflow model supports GD well, high communication complexity and dependency checking overhead are still need be solved. Hence, we propose the following three optimizations.

3.3 Optimizations

Dummy Edge We propose dummy edge technique (dashed line in the dataflow graph) to avoid fine-grain overhead of dependency checking and reduce local data movement. As shown in Figure 4, we take the user node U_i as an example to explain dummy edge. The data transferred by dummy edge can be accessed

locally because the key of the data does not change. As local data are always ready to be accessed, dependency checking of dummy edge can be avoided.

The reconstructed dataflow graph with dummy edge is shown in Figure 5. Similar with R in Figure 3, R_U and R_V transforming rating data into triplet format, are indexed by user id and item id respectively, in order to be locally accessed by U and V. As for gradient computing nodes U and V, two input edges are replaced by two dummy edges , and the other input edge remains unchanged. Data movement can be reduced because only feature vectors from normal edge should be transferred by network.. There is no dependency checking operation of every node in reconstructed dataflow graph because there is only one normal input edge of them.
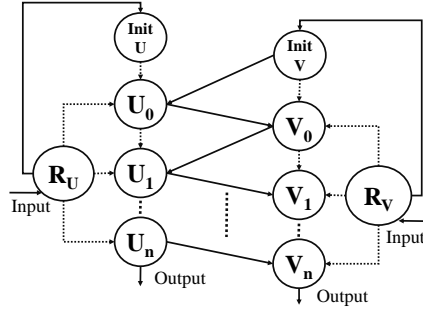

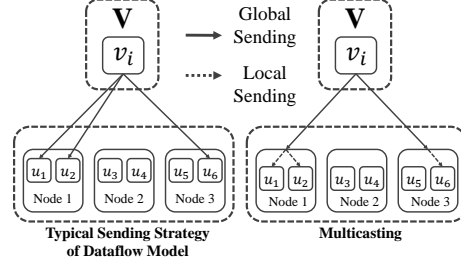**Fig. 5.** Dataflow Graph of Gradient Descent Using Dummy Edge

Fig. 6. Typical Sending Strategy of Dataflow Model and Multicasting

Multicasting With dummy edge, data movement can be reduced by avoiding sending local data. However, the quantity of data transferred by normal edges is much more than that by two dummy edges. Hence, we propose multicasting strategy to further optimize the data movement. As shown in Figure 6, global sending means sending data to different nodes in the cluster through network, while local sending means sending data in the node without network. In the scenario of sending data from V to U, typical sending strategy of dataflow model sends data to the location of dependent users by global sending directly. In this process, key-value pairs with the same value may be sent to different locations on the same node. For example, the key-value pairs sent to u_1 and u_2 on node 1 have the same value v_i. Repetitive data sent to the same node will increase the network load of the algorithm. As for multicasting strategy, key-value pairs of same feature vectors are sent to the nodes which have dependent feature vectors by global sending. For example, v_i depends on user u_1, u_2 and u_6, and there is no dependent data on node 2. Key-value pairs of v_i are sent to node 1 and 3 first by global sending. Then, each node sends the key-value pairs to dependent feature vectors by local sending. u_1 and u_2 get the key-value pair from node 1 and u_6 gets the key-value pair from node 3.

Compared with original communication complexity described in Section 2.1, the complexity by using multicasting is $O(ck(n_u + n_v))$ where c represents the node number of cluster. The complexity is reduced by $\bar{d}/c$ times where average

dependency number $\bar{d}$ increases with rating data linearly. In conclusion, multicasting can significantly reduce data movement in large scale dataset.

Mini-batch Mini-batch [10] was firstly proposed to reduce the computation complexity by limiting data batch size. In collaborative filtering, batch size of data is the dependency number of each feature vector, and feature vectors require to be sent after updating operations. Hence, we limit dependency number of each feature vector for reducing computation and communication complexities. As the dependency number of each feature vector is different, we select the minimal number between dependency number d and given mini-batch size b as the final dependency number for each feature vector. Specially in large rating scenario, the mini-batch b is always set as the final dependency number because real dependency number d is larger than mini-batch size b. The computation and communication complexities of one iteration are $O(kb(n_u + n_v))$, and the complexities are reduced by $\bar{d}/b$ times where average dependency number $\bar{d}$ increases with rating number linearly. As a result, mini-batch technique can significantly reduce the computation and communication complexities in large scale datasets.

We evaluate all optimizations in Section 4.2 and the experiments results demonstrate that these optimizations can significantly improve the performance of our algorithm.

3.4 DCF Algorithm

By applying dummy edge, multicasting and mini-batch optimizations, we get the finalized algorithm. In order to design a dataflow-based algorithm, we need to construct a dataflow graph and give detailed algorithms of each node in the graph. The finalized dataflow graph is shown in Figure 5. The key nodes are gradient computing nodes (U and V), while other nodes are used to preprocess data. As the algorithm logic is same for U and V, we give the detailed algorithm of user node U.

According to dummy edge technique, user vector u_i and rating $r_{i,j}$ are obtained by querying with user id i and item id j after receiving the key-value pair $< i, v_j >$ in lines 1-2. In lines 3-6, temporary gradient u_{sum} and dependency counter d_{u_i} are initialized to record the intermediate result of gradient computing, instead of checking redundant dependencies. In line 5 and 7-8, mini-batch strategy is applied to initialize the dependency counter d_{u_i} and the sending counter d_{v_j} for reducing computation and communication complexities. If $d_{u_i} > 0$, gradient will be added to the temporary gradient u_{sum} in lines 9-12. In lines 13-18, if the item dependencies for u_i are satisfied ($d_{u_i} = 0$), u_i will be updated by temporary gradient u_{sum} and old feature vector u_i. For every dependent item v_k of u_i, if the sending dependency is satisfied ($d_{v_k} > 0$), updated u_i will be sent to item v_k by multicasting.

4   Experiments

We have implemented DCF with Java and evaluated its performance on public datasets under both multicore and distributed platforms. Spark ALS [18] and Graphlab [17] GD are two most widely used distributed collaborative filtering

Algorithm 1 Execution Process of DCF User Node U

1: Receive $< i, v_j >$ from upstream node
2: Get u_i from user data, get $r_{i,j}$ from rating data
3: **if** d_{u_i} for u_i is not initialized **then**
4:    Get n_{u_i} from rating data
5:    $d_{u_i} = min(b, n_{u_i})$
6:    $u_{sum} = 0$
7: **if** d_{v_j} for v_j is not initialized **then**
8:    $d_{v_j} = b$
9: **if** $d_{u_i} > 0$ **then**
10:    Get $r_{i,j}$ from rating data
11:    $u_{sum} = u_{sum} + (u_i^T v_j - r_{i,j}) v_j$
12:    $d_{u_i} = d_{u_i} - 1$
13: **if** $d_{u_i} = 0$ **then**
14:    $u_i = u_i - \alpha(u_{sum} + \lambda u_i)$
15:    **for** all v_k of u_i **do**
16:      **if** $d_{v_k} > 0$ **then**
17:        Send $< k, u_i >$ to dependent v_k using multicasting
18:        $d_{v_k} = d_{v_k} - 1$

algorithms, so we select them as the contrasts of DCF. We evaluate the performance of optimizations (dummy edge, multicasting and mini-batch) separately. Scalability of DCF over contrast algorithm are assessed at the last of this section. In all the experiments, we use a cluster of one master node and ten slave nodes. The CPU of each node is Intel(R) Xeon(R) E5-2630 with 32 cores and the memory of each node is 128G. As for parameter setting, default iteration is 10 and mini-batch size is 40.

Movielens, Netflix and Yahoo Music are the three datasets used in the experiments. Detailed information of the datasets are shown in Table 1. We use 80% data for training and 20% data for testing in each dataset.

Table 1. The Statistical Information of Each Dataset

Dataset	Movielens	Netflix	Yahoo Music
user number	260,000	480,190	1,000,990
item number	40,000	17,770	624,961
rating	24,000,000	99,072,112	252,800,275

4.1 Comparing DCF with Other Distributed Algorithms

Performance: Figure 7 shows the results of DCF comparing with Spark ALS and Graphlab GD under public datasets. The trends of three algorithms are the same under different datasets. DCF shows the best performance among three algorithms. The speedup of DCF over Graphlab ranges from 5.4 to 18.01 and the average speedup is 9.76. Compared with Graphlab GD, the results demonstrate
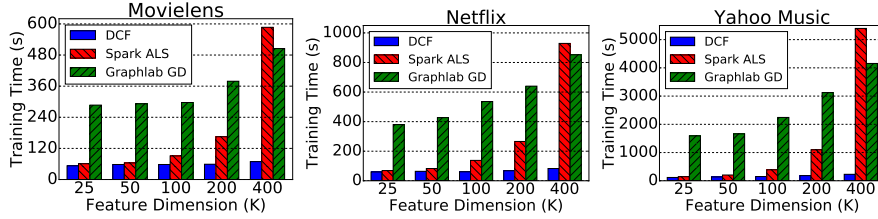
Fig. 7. Comparisons of DCF With Other Distributed Algorithms Under Public Datasets

that DCF significantly improves the performance of GD on distributed platform by applying dataflow with optimizations. The speedup of DCF over ALS ranges from 1.15 to 23.01 and the average speedup is 4.6. As k increases, the training time increases linearly with DCF but quadratically with ALS. When $k = 400$, DCF can achieve 14.34 speedup over ALS on average. The results prove the computation complexity analysis in Section 2, and show that DCF outperforms ALS due to algorithm selection and the design.

As rating size is different in three datasets, DCF shows better performance with increasing rating size. In Movielens, Netflix and Yahoo Music, DCF has an average speedup of 3.0, 3.9, 6.9 over ALS, and 5.81, 8.29, 15.11 over Graphlab GD. In three datasets, movielens has the smallest ratings and Yahoo Music has the largest ratings. DCF shows the best performance in the largest dataset.

Accuracy: We also evaluate the accuracy of three algorithms on Netflix dataset. Root mean square error (RMSE) [3] is the general metrics for model-based collaborative filtering algorithms. With smaller RMSE, the algorithms show better convergency performance. To get RMSE at 0.95, ALS, Graphlab GD and DCF converge with iterations of 6, 9 and 9 respectively. DCF shows the same convergency performance with other distributed GD algorithms. According to above data of iteration time, DCF achieves the same accuracy with 2.66x speedup over ALS and 8.51x speedup over Graphlab GD.

4.2 Optimizations Evaluation

In this subsection, we evaluate the optimizations of DCF algorithm in Netflix dataset. As shown in the left part of Figure 8, there are four experiment algorithms used to evaluate dummy edge and multicasting. DCF-Naive represents the algorithm without using proposed optimizations and mini-batch technique is not applied to four experiment algorithms. DCF-Naive with multicasting has a average speedup of 2.04 over DCF-Naive. When k increases, the speedup goes up to 3.87. The results prove that multicasting can significantly improve the performance by reducing data movement. The speedup of DCF-Naive with dummy edge over DCF-Naive is less than 5%. With multicasting, dummy edge can improve the performance by 30% on average. This difference indicates that data movement is the primary challenge in this algorithm. In conclusion, dummy edge and multicasting can significantly improve the performance.

As shown in the right part of Figure 8, we also evaluate mini-batch technique of DCF based on dummy edge and multicasting optimizations. $b = 1$ is a special case to set the dependency number to minimal value. Comparing $b = 40$ to with-
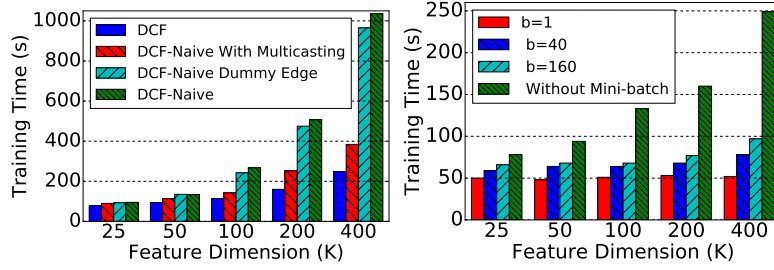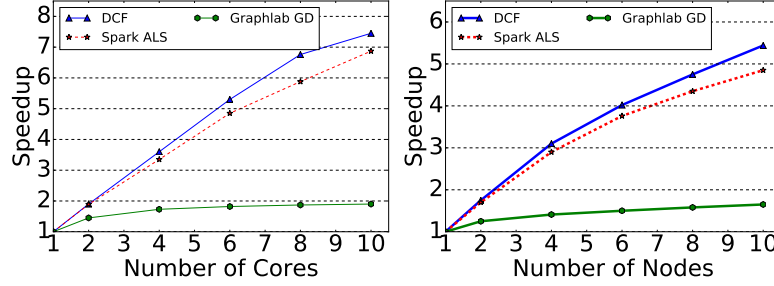
Fig. 8. Optimizations Evaluation of DCF

Fig. 9. Scalability Evaluation on Multicore and Distributed System

out using mini-batch situation, the speedup of using mini-batch strategy ranges from 1.32 to 3.19, and the average speedup is 2.08. The results demonstrate that mini-batch technique can improve the performance by optimizing computation and communication complexities. With decreasing b, mini-batch technique does not show obvious improvement when b is less than 40. $b = 40$ situation takes 31.02% longer training time than $b = 1$, which can be considered as a similar performance with $b = 1$. However, the convergence speed of $b = 1$ is much lower than other situations. Empirically, we set $b = 40$ for balancing the performance and the convergency.

4.3 Scalability

In the last subsection, we evaluate the scalability of DCF in comparison with Spark ALS and Graphlab GD. The evaluation of the three algorithms is conducted on multicore system and distributed system. Yahoo Music has the largest rating size, which leads to high computation and communication load, so we select it as the dataset for this experiment. As shown in the left part of Figure 9, DCF has a near linear scalability when the core number increases, and achieves a speedup of 7.45 on 10 cores. ALS has similar scalability with DCF and achieve a speedup of 6.87 on 10 cores. Graphlab GD has the worst scalability and only achieve a speedup of 1.9 on 10 cores.

In distributed system, we evaluate the scalability on 10 nodes. As shown in the right part of Figure 9, DCF achieves a speedup of 5.44 on 10 nodes, and ALS has similar speedup of 4.85. Graphlab GD also shows the worst scalability on nodes and only achieves a speedup of 1.65 on 10 nodes.

In conclusion, DCF shows the best scalability in three algorithms and has superior performance in both multicore system and distributed system.

5 Related Work

In recent years, parallel researches on collaborative filtering were developing parallel algorithms for ALS and GD. Mahout, the machine learning library for Hadoop [1], implemented model-based algorithms with MapReduce [11] computation model. Spark [18] implemented ALS [13] and utilized its Dataframe [4] data structure to improve the performance. However, high computation complexity of ALS becomes the bottleneck of these systems.

In order to achieve higher performance, prior studies focused on parallel GD algorithm for its low computation complexity. Graphlab [17], a distributed graph computing system, provided GD based on its vertex programming model. Due to massive data movement between nodes, graphlab shows low performance in large data set because of the network bottleneck. In recent years, parameter server architecture were proposed to support distributed GD. STRADS [14] in Petuum was proposed to schedule model parallel machine learning algorithms on distributed platform. Parameter Server [16] proposed bounded synchronous technique and several network optimizations to support GD. As for these parameter sever architecture based systems, the task scheduling level is on data partition but not on key-value pair. Tensorflow [2], another dataflow inspired system, only adopted pipelining feature of dataflow model to support GD. StarPU [5] transformed the algorithm into DAG and executed tasks according to the DAG, but it mainly focused on the multicore system rather than distributed system.

There are also studies exploiting parallelism on algorithmic level rather than runtime system. Jinoh et al. [19] proposed MLGF-MF for partitioning rating matrix and asynchronously computing gradients, but the partition strategy is not fit for distributed systems due to the network load. Yong Zhuang et al. [8] proposed FPSGD and used locking free strategy to improve the performance, but it changed the original algorithm logic of GD. In addition, these parallel collaborative filtering algorithms were developed on multicore systems but not distributed systems.

6 Conclusion

Dataflow model is a natural parallel model which can exploit maximum parallelism of algorithms by analyzing data dependency to asynchronously execute tasks in runtime execution. At present, recommender systems are facing challenges from the increasing data size. We design and implement *Dataflow-based Collaborative Filtering* (DCF) algorithm to improve the performance. We propose three optimizations (*dummy edge, multicasting and mini-bach*) to further improve our design by avoiding fine-grain overhead and reducing computation load and data movement. The experiment results illustrate that DCF can significantly improve the performance on distributed system.

References

1. Apache hadoop project. http://hadoop.apache.org/ (2017)

2. Abadi, M., Barham, P., et al., C.: Tensorflow: A system for large-scale machine learning. In: OSDI. Savannah, Georgia, USA (2016)
3. Adomavicius, G., Tuzhilin, A.: Toward the next generation of recommender systems: A survey of the state-of-the-art and possible extensions. IEEE transactions on knowledge and data engineering 17(6), 734–749 (2005)
4. Armbrust, M., Xin, R.S., et al., L.: Spark sql: Relational data processing in spark. In: SIGMOD. pp. 1383–1394. ACM (2015)
5. Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.A.: StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. Euro-Par 23, 187–198 (Feb 2011)
6. Bobadilla, J., Ortega, F., Hernando, A., Gutiérrez, A.: Recommender systems survey. Knowledge-based systems 46, 109–132 (2013)
7. Breese, J.S., Heckerman, D., Kadie, C.: Empirical analysis of predictive algorithms for collaborative filtering. In: Proceedings of the Fourteenth conference on Uncertainty in artificial intelligence. pp. 43–52. Morgan Kaufmann Publishers Inc. (1998)
8. Chin, W.S., Zhuang, Y., Juan, Y.C., Lin, C.J.: A fast parallel stochastic gradient method for matrix factorization in shared memory systems. ACM Transactions on Intelligent Systems and Technology (TIST) 6(1), 2 (2015)
9. Culler, D.E.: Dataflow architectures. Tech. rep., DTIC Document (1986)
10. Dean, J., Corrado, G., et al., M.: Large scale distributed deep networks. In: Advances in neural information processing systems. pp. 1223–1231 (2012)
11. Dean, J., Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters. Commun. ACM 51(1), 107–113 (Jan 2008)
12. Gemulla, R., Nijkamp, E., Haas, P.J., Sismanis, Y.: Large-scale matrix factorization with distributed stochastic gradient descent. In: SIGKDD. pp. 69–77. ACM (2011)
13. Hu, Y., Koren, Y., Volinsky, C.: Collaborative filtering for implicit feedback datasets. In: ICDM. pp. 263–272. Ieee (2008)
14. Kim, J.K., Ho, Q., Lee, S., Zheng, X., Dai, W., Gibson, G.A., Xing, E.P.: Strads: a distributed framework for scheduled model parallel machine learning. In: Eurosys. p. 5 (2016)
15. Koren, Y., Bell, R., Volinsky, C., et al.: Matrix factorization techniques for recommender systems. Computer 42(8), 30–37 (2009)
16. Li, M., Andersen, D.G., Park, J.W., et al., S.: Scaling distributed machine learning with the parameter server.
17. Low, Y., Gonzalez, J.E., Kyrola, A., Bickson, D., Guestrin, C.E., Hellerstein, J.: Graphlab: A new framework for parallel machine learning. arXiv preprint arXiv:1408.2041 (2014)
18. Meng, X., Bradley, J., et al., Y.: Mllib: Machine learning in apache spark. JMLR 17(34), 1–7 (2016)
19. Oh, J., Han, W.S., Yu, H., Jiang, X.: Fast and robust parallel sgd matrix factorization. In: SIGKDD. pp. 865–874. ACM (2015)
20. Takane, Y., Young, F.W., De Leeuw, J.: Nonmetric individual differences multidimensional scaling: An alternating least squares method with optimal scaling features. Psychometrika 42(1), 7–67 (1977)
21. Zuckerman, S., Suetterlein, J., Knauerhase, R., Gao, G.R.: Using a codelet program execution model for exascale machines: position paper. In: Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era. pp. 64–69. ACM (2011)