Accelerating Parallel ALS for Collaborative Filtering on Hadoop[⋆]

Yi Liang[1], Shaokang Zeng[1], Yande Liang[1], and Kaizhong Chen[1]

[1] Faculty of Information Technology, Beijing University of Technology, Beijing 100124, China
yliang@bjut.edu.cn

Abstract. Collaborative Filtering (CF) is an important building block of recommendation systems. Alternating Least Squares (ALS) is the most popular algorithm used in CF models to calculate the latent factor matrix factorization. Parallel ALS on Hadoop is widely used in the era of big data. However, existing work on the computational efficiency of parallel ALS on Hadoop have two defects. One is the imbalance of data distribution, the other is lacking the fine-grained parallel processing on the rating data. Aiming on these issues, we propose an integrated optimized solution. The solution first optimizes the rating data partition with the consideration of both the number of involved data records and the partitioned data size. Then, the multithread-based fine-grained parallelism is introduced to process rating data records within a map task concurrently. Experimental results demonstrate that our solution can reduce the overall runtime of Hadoop ALS by 82.17% by maximum.

Keywords: Collaborative Filtering · Alternating Least Squares · Hadoop.

1 Introduction

Recommendation[1][3][5][14] systems are designed to analyze and find available user data to recommend relevant and interesting items to consumers, which have become increasingly indispensable for the online businesses. Collaborative Filtering (CF)[2][4] is an important building block in recommendation systems, which predicts the user preference based on the preferences of a group of users who are considered to be similar to the active user. Alternating Least Squares (ALS) [16][22] is the most popular algorithm used in CF models to calculate the latent factor matrix factorization. ALS can easily be parallelized and efficiently handle the CF models that incorporate implicit data, and hence, it has been proved to be more general and efficient than traditional methods.

Map/Reduce is a parallel programming model as well as the corresponding framework that enables to process a massive volume of data with low-end computing nodes in a scalable and fault-tolerant manner. Hadoop is the most popular implementation of Map/Reduce. The parallel Hadoop implementations of ALS

[⋆] Supported by 2019 BenchCouncil AI System and Algorithm Challenge.

(for short, Hadoop ALS) are of great interest in the era of big data. In this paper, we mainly focus on the computational efficiency of parallel ALS on Hadoop. The user-item rating data are the major data to be processed in ALS. Existing works on Hadoop ALS have two limitations in the rating data processing. The first is the rating data distribution imbalance[13][15]. In Hadoop ALS, the rating data of a user or an item are represented as the $< key, value >$ record. The size of the records in the rating data are various. Existing works partition the rating data equally among the parallel map tasks in term of the data size and do not consider the different amounts of records involved in the partitioned data [23]. However, our observation demonstrates that the execution time of a parallel task is mainly determined by the amount of processed rating data records. Hence, existing rating data partition in Hadoop ALS leads the load among parallel tasks imbalance. The second limitation is lacking the fine-grained parallel processing on the rating data. In Hadoop ALS, the rating data are parallel processed by multiple map tasks. In each map task, a bunch of rating data records is processed in serial. However, these data records can be processed independently without any data dependency. Hence, more fine-grained parallelism should be exploited within a map task to accelerate Hadoop ALS.

Aiming to overcome the above limitations, an integrated optimized solution for Hadoop ALS is proposed. The solution first optimizes the rating data partition with the consideration of both the number of involved data records and the partitioned data size. Then, the multithread mechanism in each map task is introduced to process rating data records concurrently. With our solution, the load balance and fine-grained parallelism can be achieved to accelerate the rating data processing in Hadoop ALS. In generally, the main contributions of the paper can be summarized as follows.

1) Record-based rating data partition model. In this model, the rating data records are partitioned into HDFS files. Each data record is mapped into one HDFS file. To select the mapping file for a data record, the model partitions these HDFS files based on their size and the number of involved records. The model chooses the file with the minimum amount of records, when all files have the similar file size, so as to balance the amount of involved records in the rating data partitions and avoid the extreme situation that too many small-sized rating records are mapped into one partition.

2) Fine-grained parallelism within the map task. In our solution, the multithread mechanism is introduced into the map tasks. We adopt the producer-consumer model in rating data processing. That is, the main process of map task reads the rating data records and the forked threads parallel process these records. We also introduce the golden section search method to determine the map task number and the concurrent thread number, so as to balance the data parallel reading and processing capacity of Hadoop ALS.

3) Performance Evaluation. We evaluate our solution with the MovieLens 20M Dataset from GroupLens. Experimental results demonstrate that our solution can reduce the overall runtime of Hadoop ALS by 82.17% by maximum, while not hurting RMSE of Hadoop ALS significantly.

Works in this paper are supported by 2019 BenchCouncil AI System and Algorithm Challenge. 2019 BenchCouncil AI System and Algorithm Challenge includes three system tracks and an algorithm track, i.e, Cambricon track[24][25], RISC-V track[26], X86 track[27][28], and 3D face recognition track[29]. The challenge uses BenchCouncil AIBench as baseline[17][18], which is publicly available from `http://www.benchcouncil.org/benchhub/AIBench/`(Sign up to get access). Also, BenchCouncil provides AI benchmarks for HPC[19], AIoT[21] and Edge[20].

The rest of the paper is organized as follows. Section 2 describes the principle of the ALS algorithm and implementation of Hadoop ALS. Section 3 proposes the record-based rating data partitioning model. Section 4 present the multithread-based fine-grained parallelism with the map task. In Section 5, the performance evaluation is present in detail. Section 6 describes the conclusion and future works.

2 Related Work

Many works optimize the ALS algorithm at the algorithm level. Based on Hadoop, [12] proposed the collaborative filtering algorithm KASR to reduce the load on the system by increasing parallelism at the thread level, so that it can reduce the energy consumption. Based on spark, [10] proposed a GPU-based NMF algorithm, which can achieve high-speed operation and effectively handle non-negative decomposition of high-order matrices and greatly improve computational efficiency. [8] enables the ALS algorithm to achieve high precision results with less iterations and time. [9] extends the HALS to a distributed version and compare with the existing ALS in Spark MLlib to proves its superiority. However, these works all optimize and accelerate the ALS algorithm at the algorithm level, but ignoring the load imbalance problem of the sparse dataset on the distributed computing platform, and lacking the work of optimizing the ALS algorithm from the system level.

There are also works for layered parameter optimization and system modeling for Hadoop. [6] and [7] construct a fine-grained what-if performance prediction model, which integrates MapReduce job summary information, Hadoop cluster configuration information in job runtime, program input data and cluster hardware resource configuration information. Based on this model, the authors simulate the running effect of the program by adjusting the dataset size and changing the core configurations, so as to further optimize the system.

[11] analyses various adjustment parameters on system level (number of map task running simultaneously for each microserver node, block size of HDFS), application level (application type and input data size), and architecture level (operating voltage and frequency). The authors also discuss the impact of the performance, power and energy efficiency of the Hadoop micro-benchmark. Their work enlightens us to adjust the parameters and optimize the system at different levels.

3 Background of Hadoop ALS

In this section, we first describe the principle of ALS algorithm, then analyze the problems existing in the parallel implementation of ALS on Hadoop.

3.1 Principle of ALS

Collaborative filtering is an important building block in recommendation systems, which generates recommendation based on the similarity of users or items. In the famous Netflix Prize algorithm competition, [16] proposed the collaborative filtering based on Alternating Least Squares (ALS), which has significant advantages over other algorithms on the computational efficiency because of the embarrassing parallel characteristic of ALS. ALS adopts the iterative way to solve a series of least squares regression problems. For the user-item rating matrix R, a low rank matrix X is found to approximate the original matrix R. Solving R can be expressed as follows.

$$R \approx X = UI^T, \tag{1}$$

where $U \in C^{m \times d}$, $I \in C^{n \times d}$ and C represent complex numbers, m and n represent the numbers of rows and columns of the matrix R and d represents the number of eigenvalues.

The specific process is to randomly generate U_0, and fix it to solve I_0, then fix I_0 to solve U_1, which is called alternating computing. ALS is convergent because each iteration reduces the reconstruction error which has a lower bound. However, the Matrix Factorization speed of ALS is slow with high computing cost, which reflects the characteristics of computing-intensive workload.

3.2 Hadoop Implementation of ALS

Hadoop is the most popular implementation of Map/Reduce program model. A typical Hadoop job consists at most one map phase and one reduce phase. Each phase is executed with parallel tasks. In Hadoop job, the input data are represent as the $< key, value >$ pairs, the map/reduce task consumes one pair each time. The user-item rating data are the major input data of ALS. In Hadoop ALS, the rating data are organized in term of UserID or ItemID. That is, all rating data associated with one user or item are represent by one ¡key, value¿ pair, with UserID or ItemID as the key and all related rating data as the value. In Hadoop ALS, we call such $< key, value >$ pair as a rating data record.

Figure 1 shows the workflow of Hadoop ALS. In Hadoop ALS, the rating data are first preprocessed and partitioned.Then,for each data partition, an ALS map task is launched to process it. When all parallel map task finished, the new Matrix U and I are generated. The above operations are conducted iteratively until the number of iterations reaches the threshold.

In Hadoop ALS job, there isn't the reduce phase. ALS map task consumes one rating data record each time. There is no data dependency among record

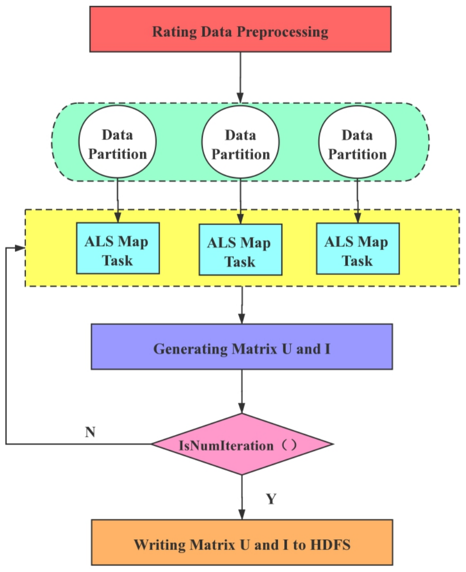

Fig. 1: Workflow of Hadoop ALS

processing. On the other hand, due to the uncertainty of user's rating behavior, the amount of rating data associated with users/items are various. Hence, the size of the data records are quite different in rating data.

4 User-Item Rating Data Partitioning

In this section, we first analyze the distribution and time consuming of the user-item rating data records, then propose the record-based partition model for the rating data.

4.1 Imbalance in Rating Data Processing

In Hadoop ALS, the user-item rating data are the major data to be processed iteratively. In each iteration, the user-item rating data need to be processed in term of user id and item id alternatively. Hence,in the Hadoop implementation of ALS, the user-item rating data are represented as $< key, value >$ list and stored in two copies. In one copy, a record has the key as the user id and the value as the collection of all rating data of a specific user and can be formatted as $\langle$*userid,[itemid:rating,itemid:rating...]*$\rangle$. The other copy has the key as the item id and the value as the collection of all rating data of a specific item, and can be formatted as $\langle$*itemid,[userid:rating,userid:rating...]*$\rangle$. On the data processing, the rating records are partitioned among the map tasks in the same data size. The

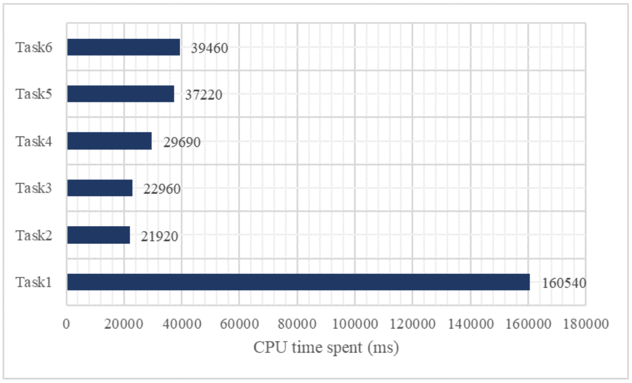

Fig. 2: Runtime of Map Tasks in Hadoop ALS

map task reads one key-value record in each time and works out the statistics of the comprehensive rating data of one user or item.

We evaluate the performance of ALS in Hadoop with the Movielen 20M data on three computing nodes. The total size of user-item rating data is 724MB. The number of map task is set to 6. In our evaluation, the jobs for the User and Item matrix training are dominant the execution time of parallel ALS in Hadoop. Among these, QR factorization is the core part, which takes up 86% of User and Item Matrix training time. Figure 2 demonstrates the load imbalance in the runtime of Item matrix training jobs. Among 6 map tasks, task 1 has the significantly longer execution time than the other five tasks, at least four times longer than others. We then analyze the data processed by each map task. Figure 3 shows the data size and the key-value records processed by the six tasks, respectively. Even though the rating data are partitioned equally in term of data size, the key-value records distribution among parallel map tasks are quite imbalanced. Particularly, the number of records processed by task 1 is 7 times larger than others at least. It is reasonable due to that with the amount of items growing up, the user has more possibility to give the explicit ratings to part of items randomly. This leads the various value numbers in key-value records, and thus, the various record numbers in rating data partitions. On the other hand, the computational complexity of QR factorization in Hadoop ALS is represent as km^3, where k is the record number and m is the feature number. Hence, in Hadoop ALS, tasks, which process the rating data partitions with larger record amount, will have longer runtime.

4.2 Record-based Rating Data Partitioning

Based on our evaluation, We propose a record-based rating data partition model in this section. Be different from the data partition in Hadoop ALS, for the two rating data copies, the model partitions the data with the consideration of both the partition data size and the records involved in each partition.

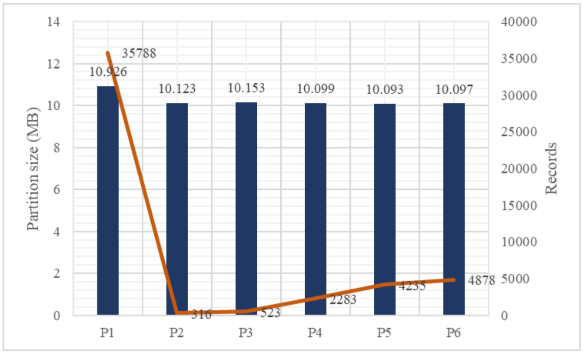

Fig. 3: Data Size and Record Numbers of Map Tasks in Hadoop ALS

Figure 4 demonstrates the framework of our data partition model. In this model, the rating data are partitioned and stored into several HDFS files. Each data record is stored into only one file. The number of files is equal to that of map tasks in Hadoop ALS jobs. We implement the rating data partitioning in the rating data pre-processing job. The data partitioning algorithm is described in Algorithm 1.

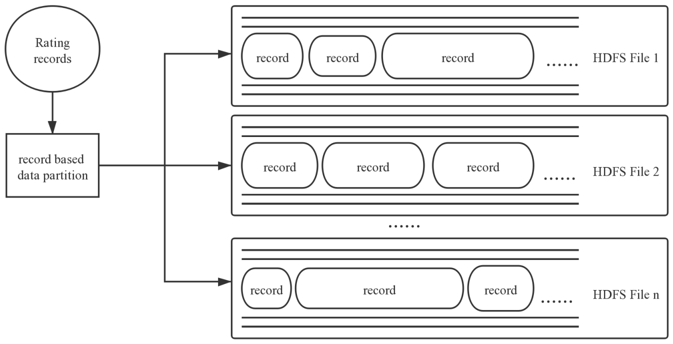

Fig. 4: Record-based Rating Data Partitioning Model

In this strategy, the inputs are the raw rating data record set and the number of map tasks, the output is the HDFS file set containing the partitioned rating data. The rating data are processed once a record. For each output file, a storing priority is assigned. The priority can be expressed as Equation 2.

$$weight = (nv_i/N_v) \times w_v + (nr_i/N_r) \times w_r, \tag{2}$$

where, nv_i is the number of values in the output file, nr_i is the amount of records involved in output file i, N_v is the total number of values of all output files, N_r is

Algorithm 1: Rating data partitioning algorithm

Input: *the rating records list RecordsList,the number of output files OutputNum*
Output: *the output files set OutputfileSet*

```
1:  Initialize:OutputfileSet $\leftarrow \emptyset$;file $\leftarrow \emptyset$ ;
2:  for $i \leftarrow 1$ to OutputNum do
3:      $file_i \leftarrow createOutputfile(i)$ ;
4:      $file_i.nv \leftarrow 0$ ;
5:      $file_i.nr \leftarrow 0$ ;
6:      $AddFile(file_i, OutputfileSet)$ ;
7:  for each $record_i \in RecordsList$ do
8:      $CalculateFilesWeight(OutputfileSet)$ ;
9:      $file \leftarrow FindTheMinWeightFile(OutputfileSet)$ ;
10:     $WriteRecord(record_i, file)$ ;
11:     $UpdateWeight(record_i, file)$ ;
12: return $OutputfileSet$ ;
```

the total amount of records stored in all output files, w_r and w_v are determined by empirical values, but they need to satisfy Equation 3.

$$w_r + w_v = 1 \tag{3}$$

We initialize an OutputfileSet, each file of it has four attribute which are nv, nr, weight and there records list (Line 1). nv_i and nr_i of each output file are initialized to 0 (Lines 2 to 6). For each rating record, the following steps are conducted. First, the storing priority of each output file is calculated with Equation 2 (Line 8). Second, the file with highest priority is selected as the record's target output. Once there are multiple files with the highest priority, the target output is selected randomly (Line 9). Finally, nv and nr of the selected output file are increased by the number of values in the rating record and by one, respectively (Line 10 to 11). Our partition strategy can balance the amount of records involved in output files and avoid the extreme situation that too many small-sized rating records are stored in one file. The priority weights w_r and w_v enable users to make trade-off between the balance of data size and the record amount among output files.

5 Multithread-based parallelism within map task

As described in Section 2, the processing of each rating data record is independent. Existing Hadoop ALS processes the rating data records in serial within a map task. Hence, we introduce the multithread mechanism into the map task to achieve the fine-grained parallelism in Hadoop ALS. Based on the multithread mechanism, we adopt the Producer-Consumer model in the rating record processing. The model is shown in Figure 5. The main process of map task acts as the producer to read the rating data records from HDFS files and put the records into the record queue. The multiple threads forked in a map task are taken as

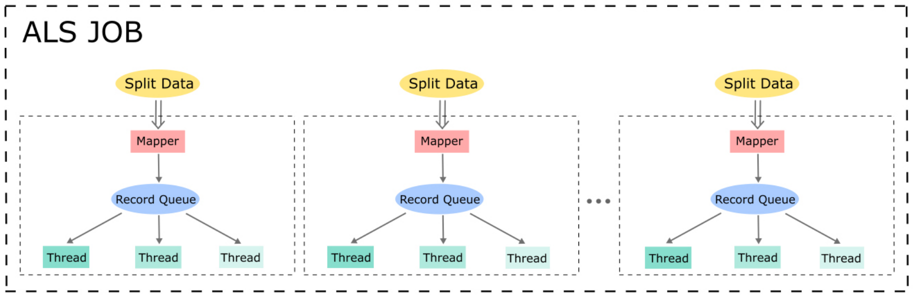

Fig. 5: Producer-Consumer Model

consumers to fetch and compute the queued records concurrently. The proposed Producer-Consumer model not only pipelines the rating data reading and computing, but also enables the fine-grained parallelism of rating data computation within a map task.

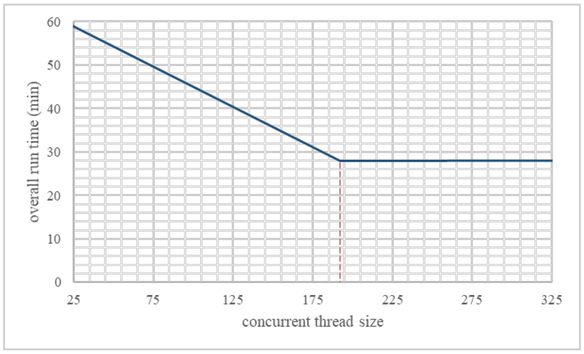

Fig. 6: Overall Run Time of Different Concurrent Thread Sizes

For a specific ALS job, the size of parallel map tasks determines its data reading capacity and the size of concurrent threads within the map task determines its data computing capacity. To achieve the capacity balance of data reading and computing, we need to make the trade-off between these two size settings. We first fix the map task size setting and examine Hadoop ALS performance under different concurrent thread size settings. As Figure 6 shown, the execution time of Hadoop ALS job decreases with the concurrent thread size increase, and reaches to the minimum when the total size of concurrent threads across all map tasks is equal to the size of allocated computing resources. It is reasonable for that Hadoop ALS workload is computing-intensive and each thread needs to consume one computing core during its computation. Larger thread size setting leads to the less average workload dispatched to a thread. However, when the total size of threads exceeds that of allocated resources, the surplus threads will be suspended due to the computing resource contention and make no contributions to the Hadoop ALS performance improvement.

We further examine the relationship between Hadoop ALS performance and the map task size setting under the constraint of the fixed size of allocated computing resources. Denoting N as the size of allocated resources and m as the map task size, we set the concurrent thread size as $\frac{N}{m}$. As shown in Figure 7, the relation function satisfies unimodal nonlinearity. Overall run time decreases with the map task size goes up, because the data reading capacity increases which matches the concurrent data computation capacity. Larger map task size leads to less inner concurrent threads, degrading rating data records processing throughput and in this situation, the capacity of data reading and computation is imbalance, which increases overall run time.

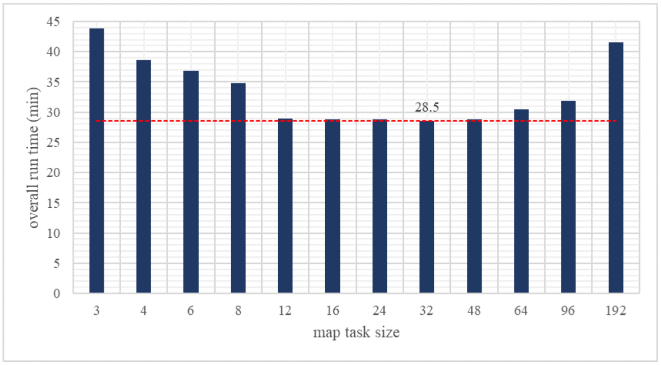

Fig. 7: Overall Run Time of Different Map Task Sizes

Hence, we adopt one-dimensional search strategy based on golden section heuristic method to find the optimal size settings. The search strategy can be described as Algorithm 2.

During the search, we attempt to find the optimal configuration for performance of ALS. In this algorithm, we record all attempted performance in the form of a tuple, (map task size, concurrent thread size), and all of tuples are stored in $Sampleset$. First, we initialize $Sampleset$ and the search boundary as M_{min} and M_{max}(Line 1 to 2). Next, we iterated the following processing: we employ Golden Section Method to calculate candidate lower and upper boundaries(Line 4 to 5), and we find performance by these in $Sampleset$. If not find, set the map task size as $b_l^{'}$ or $b_u^{'}$, the concurrent thread size as $\lceil \frac{M_{max}}{b_l^{'}} \rceil$ or $\lceil \frac{M_{max}}{b_u^{'}} \rceil$ to run ALS(Line 6 to 15). Then compare performance of two candidate boundaries, so as to determine how to update the search area(Line 16 to Line 19). The above operations are conducted iteratively until the lower and upper boundaries converge to small enough(Line 20). Finally, the optimal map task size is the one of the lower and upper boundaries that performs better(Line 21 to 22).

Algorithm 2: Optimized Map Task Size Searching Algorithm

Input: M_{min}: the minimum map task size, M_{max}: the maximum map task size, λ: the Golden Section coefficient, ϵ: the search threshold
Output: m: the optimal map task size

```
1:  $InitializeSampleset()$
2:  $b_l \leftarrow \mathrm{M}_{min}$; $b_u \leftarrow \mathrm{M}_{max}$
3:  repeat
4:    $b_l^{'} \leftarrow b_l + (1-\lambda)(b_u - b_l)$
5:    $b_u^{'} \leftarrow b_l + \lambda(b_u - b_l)$
6:    if($FindByKey(Sampleset, b_l^{'})$ == False)
7:      $t_l \leftarrow runALS(b_l^{'}, UpRct(b_l^{'}, M_{max}))$
8:      $AddElement(Sampleset, (b_l^{'}, t_l))$
9:    else
10:     $t_l \leftarrow FindByKey(Sampleset, b_l^{'})$
11:   if($FindByKey(Sampleset, b_u^{'})$ == False)
12:     $t_u \leftarrow runALS(b_u^{'}, UpRct(b_u^{'}, M_{max}))$
13:     $AddElement(Sampleset, (b_u^{'}, t_u))$
14:   else
15:     $t_u \leftarrow FindByKey(Sampleset, b_u^{'})$
16:   if ($t_l \geq t_u$)
17:     $b_l \leftarrow b_l^{'}$
18:   else
19:     $b_u \leftarrow b_u^{'}$
20: until $b_u - b_l \leq \epsilon$
21: $m \leftarrow GetMinByValue(Sampleset, b_l, b_u)$
22: return $m$
```

6 Performance Evaluation

We compare the performance of our optimized Hadoop ALS with the original Hadoop ALS from AIBench. Besides the record-based rating data partition model and the fine-grained parallelism within the map task, we optimize the JVM heap configurations of Hadoop ALS. In this section, we quantify the contributions of these optimization options to the performance enhancement of Hadoop ALS.

6.1 Experimental Design

The experiments are conducted on a cluster of 3 identical nodes supported by 2019 BenchCouncil AI System and Algorithm Challenge[17][18][19][20][21].There is always one master node. Configurations of the cluster are described in Table 1. We execute Hadoop ALS on MovieLen dataset, which contains rating data of multiple users for movies and is often used as the dataset for recommendation system and machine learning algorithm evaluations. We use two metrics in

Table 1: Experimental Environment

Name	Configuration
CPU	X86 platform, 64 cores, 8MB LLC
Memory	250 GB, 2667 MHz
OS	Kernel Linux 3.10.0
JVM	HotSpot JDK 8u221
Hadoop	Version 3.2.0
Mahout	Version 0.12.2

our performance evaluations, Overall Run Time (ORT) for the computational efficiency and Root Mean Square Error (RMSE) for the training accuracy.

$$ORT = t_e - t_s \tag{4}$$

$$RMSE = \sqrt{\frac{\sum_i (v_i - v_i^{'})^2}{n}} \tag{5}$$

where t_e and t_s represents the time stamp obtained at the end and the beginning of ALS algorithm, $v_i^{'}$ and v_i are the estimated value and the ground truth of instance i, respectively and n is the total number of observed values.

6.2 Performance Result Analysis

We first compare the overall performance of our optimized Hadoop ALS (OPHA) with the original Hadoop ALS (HA). The original Hadoop ALS are configured and executed in two ways, MP and MT. For MP, we disable the multithread mechanism in the map task and configure the size of map tasks as that of the allocated resources. For MT, we deploy one map task in each node and configure the thread size within a map task as the size of allocated resources in one node. The experimental results are shown in Figure 8.

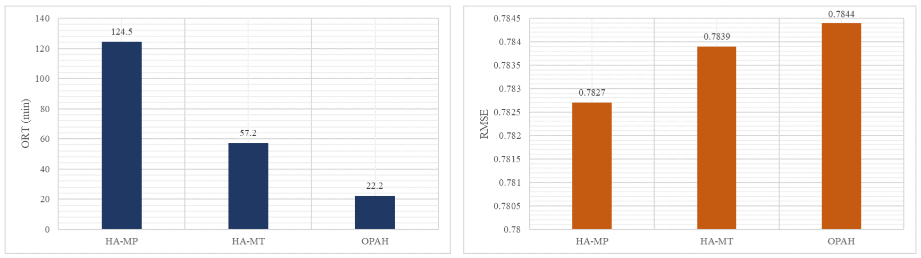

Fig. 8: ORT and RMSE of OPHA and Original Hadoop ALS

The experimental results demonstrate that OPHA outperforms both HA-MP and HA-MT. The overall runtime is decreased by 82.17% and 61.19% respectively. However, RMSE is only increased by 0.22% by maximum. Even though all three Hadoop ALS implementations are executed with the same parallelism (that is, HA-MP with 180 parallel task processes, HA-MT and OPHA with 180 concurrent threads), HA-MP achieves the worst performance on overall runtime. This is for that, the process-level parallelism makes the rating data partitioned into more pieces and processed by map tasks separately, which leads to the higher possibility of load imbalance among these parallel tasks. On the other hand, the thread-level parallelism enables the data sharing among concurrent threads, and hence, increases the rating data records processing throughput within a map task and lower the possibility of load imbalance among map tasks. Compared to HA-MT, OPHA dispatches the rating data records among map tasks evenly and optimizes the map task size and concurrent thread size settings to balance the data reading and computation capacity, and hence, achieves better performance.

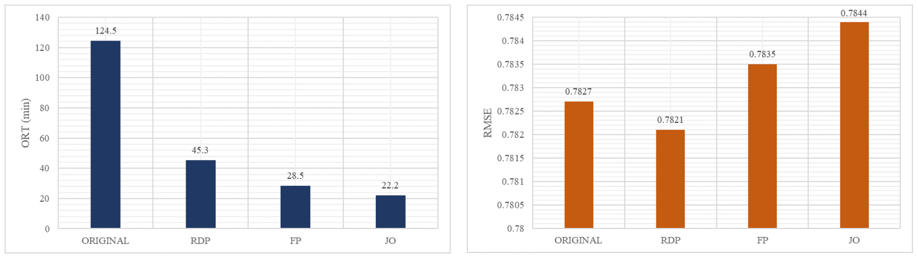

Fig. 9: Performance Contributions of Proposed Optimization Options

Figure 9 shows the performance enhancement contributions of all proposed optimization options, which include Record-based rating Data Partitioning (RDP), Fine-grained Parallelism (FP) and JVM heap size Optimization (JO). The overall runtime of Hadoop ALS can be reduced by 63.59% with RDP, further by 37.07% with FP, and finally by 22.18% with JO. The results prove that the rating data distribution imbalance and the serial data processing within the map task are the main sources of inefficiency of the original Hadoop ALS, and our integrated solution solve these two problems effectively.

7 Conclusion

In our work, we propose an integrated optimized solution for the parallel ALS on Hadoop, which incorporates the record-based rating data partition and multithread-based fine-grained parallelism within the map task. Experimental results demonstrate that our solution can reduce the overall runtime of Hadoop ALS by 82.17% by maximum, while not hurting RMSE of Hadoop ALS significantly. In the future work, we will optimize the rating data partitioning strategy to accommodate the situation of larger-sized rating data and multi-waved map task execution.

References

1. Bokde D., Girase S., Mukhopadhyay D.: Matrix factorization model in collaborative filtering algorithms:A survey, J, .Procedia Computer Science, 49(1), 136-146(2015).
2. Hernando A., Bobadilla J., Ortega F.: A non negative matrix factorization for collaborative filtering recommender systems based on a Bayesian probabilistic model. Knowledge-Based Systems, 97(4),188-202(2016) .
3. Deshpande, Mukund, George Karypis.: Item-based top-n recommendation algorithms. ACM Transactions on Information Systems(TOIS). 22(1), 143-177 (2004).
4. Hanmin Y., Qiuling Zhang, Xue Bai.: A New Collaborative Filtering Algorithm based on Modified Matrix Factorization. In: Electronic and Automation Control Conference (IAEAC), pp.147-151. IEEE(2017).
5. Zhen Yang, Weitong Chen, Jian Huang.: Enhancing recommendation on extremely sparse data with blocks-coupled non-negative matrix factorization. J. Neurocomputing, 278, 126-133(2018).
6. Herodotou H., Dong F., Babu S.: Mapreduce programming and cost-based optimization crossing this chasm with starfish. J. Proceedings of the VLDB Endowment, 4(12), 1446-1449(2011).
7. Herodotou H.: Hadoop performance models. J. arXiv preprint arXiv, 1106.0940(2011).
8. Manda W., Michael B., Anthony L., Hans D.: Algorithmic Acceleration of Parallel ALS for Collaborative Filtering:Speeding up Distributed Big Data Recommendation in Spark. In: 21st International Conference on Parallel and Distributed Systems(ICPADS), pp. 682-691. IEEE(2015) .
9. Krzysztof F., Rafal Z.: Distributed Nonnegative Matrix Factorization with HALS Algorithm on Apache Spark. In: Artificial Intelligence and Soft Computing - 17th International Conference(ICAISC), pp. 333-342(2018).
10. Bing T., Linyao K., Yanmin Xia., Li Zhang: GPU-accelerated Large-Scale Nonnegative Matrix Factorization Using Spark. In: Collaborative Computing: Networking, Applications and Worksharing- 14th International Conference(EAI), pp. 189–201(2018).
11. Maria M., Katayoun N., Setareh R., Houman H.: Hadoop Workloads Characterization for Performance and Energy Efficiency Optimizations on Microservers. J. IEEE Trans. Multi-Scale Computing Systems. 4(3), 355-368(2018).
12. Jyotindra T., Dr. Mahesh P., Dr. Anjana P.: A Hadoop based collaborative filtering recommender system accelerated on GPU using OpenCL. J. International Journal of Engineering Sciences & Research Technology, 6(9), 195-209(2017).
13. C. Teflioudi, F. Makari, and R. Gemulla: Distributed matrix completion. In 12th International Conference on Data Mining (ICDM), pp. 655–664. IEEE(2012).
14. H.-F. Yu, C.-J. Hsieh, I. Dhillon et al.: Scalable coordinate descent approaches to parallel matrix factorization for recommender systems. In: 12th International Conference on Data Mining (ICDM), pp. 765–774. IEEE(2012).
15. M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica: Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In: Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation, pp. 15–28(2012).
16. Y. Zhou, D. Wilkinson, R. Schreiber, and R. Pan,: Large-scale parallel collaborative filtering for the Netflix prize. In: Proceedings of the 4th International Conference on Algorithmic Aspects in Information and Management, pp. 337–348(2008).

17. Gao Wanling, Tang Fei, Wang, Lei, Zhan Jianfeng, Lan Chunxin,et. al.: AIBench: An Industry Standard Internet Service AI Benchmark Suite. J. arXiv preprint arXiv, 1908. 08998(2019).
18. Gao Wanling, Luo Chunjie, Wang Lei et al.: AIBench: towards scalable and comprehensive datacenter AI benchmarking. In: BenchCouncil International Symposium on Benchmarking, Measuring and Optimizing (Bench18), pp. 3-9. Springer(2018).
19. Jiang Zihan, Gao Wanling, Wang Lei, Xiong Xingwang et al.: HPC AI500: A Benchmark Suite for HPC AI Systems. In: BenchCouncil International Symposium on Benchmarking, Measuring and Optimizing (Bench18). Springer(2018).
20. Hao Tianshu, Huang Yunyou, Wen Xu, Gao Wanling et al.: Edge AIBench: Towards Comprehensive End-to-end Edge Computing Benchmarking. In: BenchCouncil International Symposium on Benchmarking, Measuring and Optimizing (Bench18). Springer(2018).
21. Luo Chunjie, Zhang Fan, Huang Cheng, Xiong Xingwang et al.: AIoT Bench: Towards Comprehensive Benchmarking Mobile and Embedded device Intelligence. In: BenchCouncil International Symposium on Benchmarking, Measuring and Optimizing (Bench18). Springer(2018).
22. Comon P., Luciani X., de Almeida A.L.F.: Tensor decompositions, alternating least squares and other tales. J. Chemom., 23 , pp. 393-405(2009).
23. L. Liu,: Computing infrastructure for big data processing. Front. Comput. Sci., 7, pp. 165-170(2013).
24. Guangli Li, Xueying Wang, Xiu Ma, Lei Liu, Xiaobing Feng.: Towards efficient inference of residual neural networks on cambricon chips. In: International Symposium on Benchmarking, Measuring and Optimization (Bench'19). Springer(2019).
25. Jiansong Li, Zihan Jiang: Performance analysis of cambricon mlu100. In: International Symposium on Benchmarking, Measuring and Optimization (Bench'19). Springer(2019).
26. Pengpeng Hou, Jiageng Yu, Yuxia Miao, Yang Tai, Yanjun Wu, Chen Zhao.: A light-weight neural network inference framework based on the risc-v architecture. In: International Symposium on Benchmarking, Measuring and Optimization (Bench'19). Springer(2019).
27. Weixin Deng, Pengyu Wang, Jing Wang, Chao Li, Minyi Guo.: Psl: Exploiting parallelism, sparsity and locality to accelerate matrix factorization on x86 platforms. In: International Symposium on Benchmarking, Measuring and Optimization (Bench'19). Springer(2019).
28. Tianshu Hao, Ziping Zheng.: The implementation and optimization of matrix decomposition based collaborative filtering task on x86 platform. In: International Symposium on Benchmarking, Measuring and Optimization (Bench'19). Springer(2019).
29. Xingwang Xiong, Xu Wen, Cheng Huang: Improving rgb-d face recognition via transfer learning from a pretrained 2d network. In: International Symposium on Benchmarking, Measuring and Optimization (Bench'19). Springer(2019).