This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2020.2997051, IEEE Transactions on Computers

BaPa: A Novel Approach of Improving Load Balance in Parallel Matrix Factorization for Recommender Systems

Ruixin Guo, Feng Zhang, Lizhe Wang, Wusheng Zhang, Xinya Lei, Rajiv Ranjan, and Albert Y. Zomaya

Abstract—A simplified approach to accelerate matrix factorization of big data is to parallelize it. A commonly used method is to divide the matrix into multiple non-intersecting blocks and concurrently calculate them. This operation causes the Load balance problem, which significantly impacts parallel performance and is a big concern. A general belief is that the load balance across blocks is impossible by balancing rows and columns separately. We challenge the belief by proposing an approach of "Balanced Partitioning (BaPa)". We demonstrate under what circumstance independently balancing rows and columns can lead to the balanced intersection of rows and columns, why, and how. We formally prove the feasibility of BaPa by observing the variance of rating numbers across blocks, and empirically validate its soundness by applying it to two standard parallel matrix factorization algorithms, DSGD and CCD++. Besides, we establish a mathematical model of "Imbalance Degree" to explain further why BaPa works well. BaPa is applied to synchronous parallel matrix factorization, but as a general load balance solution, it has significant application potential.

Index Terms—Matrix Factorization, Stochastic Gradient Descent, Recommender Systems, Parallel and Distributed Computing, Load Balancing

1 Introduction

THE recommender system is being widely applied in e-commerce, social networks, location-based services, and other fields with the development of the Internet. Matrix factorization is a conventional method to implement collaborative filtering recommender systems [1], [2]: the rating matrix is decomposed into the product of two or more matrices, which are used to predict the missing ratings in the original matrix. Typical matrix decomposition algorithms include SGD (Stochastic Gradient Descent), CCD (Cyclic Coordinate Decent), ALS (Alternating Least Squares), and so on. SGD [3], [4] is based on gradient descent, CCD [5] is based on coordinate descent, and ALS [6] is easy to be parallelized, but its efficiency and accuracy are lower than the other two.

The sequential implementation of matrix decomposition becomes inefficient when the matrix grows to a massive scale. One acceleration method is to parallelize it. The difficulty of parallelizing SGD and CCD is that they are data-dependent: the inputs to an instruction depends on the outputs of the last instruction, rendering them inherently sequential [7]. At present, the parallelization often adopts the method of partitioning: the rating matrix is divided into multiple non-intersecting blocks, which can be calculated in parallel, and the outputs of different tasks are integrated through transmitting [8], [9]. DSGD is a typical parallel SGD algorithm based on partitioning [10], and CCD++ represents the parallel CCD algorithm based on partitioning [11], [12], [13].

Synchronization and asynchronization are two strategies to schedule parallel matrix factorization [14]. Synchronous parallel matrix factorization algorithms include DSGD, CCD++, DS-ADMM [15] and so on, and asynchronous algorithms cover FPSGD [16], Hogwild [17], NOMAD [18], MGLF-MF [19], Asy-GCD [20] and the like.

Load balance is a major factor influencing the efficiency of parallel computing [21], [22], specifically for synchronous computing. It is caused by the uneven workload of tasks [23]. For synchronous computing, tasks need to finish simultaneously in each iteration. However, if tasks complete one after another, the first finished must be blocked and idly wait for the last completed [24]. These delays are primarily due to the uneven data distribution across blocks, which is the primary reason causing the problem of load imbalance [25]. Load imbalance is a critical challenge that must be addressed in parallel computing.

The state-of-the-art approaches of improving load balance for matrix factorization are approximately divided into three categories: dynamic assignment, random permutation, and different-sized partitioning. Dynamic assignment [23] adjusts the workload of each task dynamically and is usually adopted by asynchronous algorithms such as Hogwild and NOMAD. But it is hard to be implemented in a distributed memory environment due to the complex scheduling and large communication cost [26]. Random permutation [27] rearranges the matrix by permuting rows and columns randomly and is adopted by FPSGD. However, a random permutation is not applicable in distributed memory environments due to its vast I/O cost [19]. Different-sized partitioning, known as General Block Distribution (GBD) and Semi-general Block Distribution (SBD) in [28], aims to adjust the size of blocks to balance the ratings in each block. Communication scheduling in SBD-based algorithms such as MGLF-MF is often complicated. We believe that GBD is more applicable to the synchronous matrix

- *Ruixin Guo, Feng Zhang, Lizhe Wang, and Xinya Lei are with the School of Computer Science, China University of Geosciences, Wuhan, P. R. China. E-mail: jeff.f.zhang@gmail.com*
- *Wusheng Zhang is with the Department of Computer Science, Tsinghua University, P. R. China.*
- *Rajiv Ranjan is with the School of Computing, University of Newcastle, U.K.*
- *Albert Y. Zomaya is with the School of Computer Science, University of Sydney, Australia.*

Manuscript received June 14, 2019; revised XXXX XX, 2019.

0018-9340 (c) 2020 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: Newcastle University. Downloaded on February 17,2021 at 12:44:56 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2020.2997051, IEEE Transactions on Computers

factorization algorithm than SBD. Other partitioning methods such as hypergraph partitioning [29], [30] and probabilistic modeling [31], [32] may also achieve good load balance, but they are too costly.

In this paper, we study synchronous parallel matrix factorization and introduce two pieces of innovative work to demonstrate why and how separately balancing rows and columns can lead to load balance across blocks.

(1) We introduce the approach of Balanced Partitioning (BaPa) to demonstrate under what circumstance, independently balancing rows and columns can prompt ratings to be distributed evenly across blocks, thereby improving load balance. We formally prove its feasibility by observing the variance of rating counts across blocks, and empirically demonstrate its high performance by applying it to DSGD and CCD++.

(2) We further introduce an "Imbalance Degree" model to explain why BaPa works well. Imbalance Degree measures how imbalanced the rating distribution across blocks is. Based on the Imbalance Degree model, we formally and empirically show that a more imbalanced dataset may lead to more significant speedup growth.

Last but not least, BaPa can be applied to a variety of block-based, synchronous parallel matrix factorization algorithms.

The remainder of the paper is organized as follows. Section 2 contains some preliminary knowledge of matrix factorization, DSGD, CCD++ and so on. Section 3 introduces the concept of BaPa and formally and empirically validates its feasibility of partitioning the ratings evenly across blocks. Section 4 demonstrates the superior performance of BaPa in speedup growth and introduces the Imbalance Degree model to explain why. Finally, Section 5 concludes the paper and identifies future work.

2 PRELIMINARIES

This section introduces matrix decomposition (Section 2.1), SGD and DSGD (Section 2.2), CCD++ (Section 2.3), and the challenges in DSGD and CCD++ (Section 2.4).

2.1 Matrix factorization

Matrix factorization is a commonly-used technology to implement collaborative filtering recommendation systems [1]. Rating matrices are usually sparse because the number of user-item ratings is small compared with a large number of users and items, and most of the user-item cells are null.

Let $R \in \mathbb{R}^{u \times v}$ be a rating matrix, in which u is the number of users and v is the number of items. Matrix factorization is to decompose R into a product of a user matrix $U^{u \times k}$ and an item matrix $V^{v \times k}$, i.e., $R \approx \hat{R} = UV^T$. U and V are derived based on R. Note that the ratings in $\hat{R}$ but not in R are the predictions.

2.2 SGD-based matrix factorization

There are many methods to achieve matrix factorization. SGD is one of them and is introduced by Koren et al. [3], [4]

2.2.1 SGD

Let r_{ij} be the rating of the i-th row, j-th column in R, u_{ij} and v_{ij} be the values of the i-th row, j-th column in U and V respectively. Let u_i be the i-th row vector of U, i.e., $u_i = (u_{i1}, u_{i2}, ..., u_{ik})$, v_j be the j-th row vector of V, i.e., $v_j = (v_{j1}, v_{j2}, ..., v_{jk})$.

Let the loss function of matrix factorization be formatted as the following regularized least squares.

$$J(U,V) = \sum_{r_{ij} \in R} [(r_{ij} - u_i v_j^T)^2 + \lambda(||u_i||^2 + ||v_j||^2)] \quad (1)$$

where $||.||^2$ is the second order norm and $\lambda(||u_i||^2 + ||v_j||^2)$ is the regularized parameter leveraged to avoid overfitting.

Then U and V can be obtained by solving the minimization problem.

$$\begin{aligned} &\arg\min_{U,V} J(U,V) = \\ &\arg\min_{U,V} \sum_{r_{ij} \in R} [(r_{ij} - u_i v_j^T)^2 + \lambda(||u_i||^2 + ||v_j||^2)] \end{aligned} \quad (2)$$

To solve the problem in Formula (2), we calculate the partial derivatives of the function $J(U,V)$ with respect to u_i, v_j respectively, obtaining their updating formulas as follows:

$$u_i \leftarrow u_i + \alpha(e_{ij} v_j - \lambda u_i) \quad (3)$$

$$v_j \leftarrow v_j + \alpha(e_{ij} u_i - \lambda v_j) \quad (4)$$

where α is the learning rate, and $e_{ij} = r_{ij} - u_i v_j^T$.

Evaluating the Formulas (3) and (4) once using a rating, we call it an update; after going through all the ratings, we call it one iteration. Two criteria can be used to decide the algorithm's convergence: the elapsed time reaches the maximum; the accuracy of the current iteration is worse than that of the previous one. RMSE is used to define the accuracy of prediction, shown as Formula (5).

$$RMSE = \sqrt{\frac{\sum_{r_{i,j} \in R} (r_{i,j} - u_i v_j^T)^2}{sum}} \quad (5)$$

where sum is the number of actual ratings in R. A smaller RMSE signifies a better factorization, leading to more accurate predictions.

The above procedure shows how SGD is applied to matrix factorization. Formulas (3) and (4) show u_i and v_j are interdependent, which renders SGD inherently sequential and not easy to be parallelized.

2.2.2 DSGD

DSGD is used to implement synchronously parallel SGD by the blocking technique [10].

DSGD decomposes $R^{u \times v}$ into $n \times n$ blocks, each with a size of $(u/n) \times (v/n)$. Accordingly, $U^{u \times k}$ is decomposed into n blocks, each with a size of $(u/n) \times k$, and $V^{v \times k}$ is decomposed into n blocks, each with a size of $(v/n) \times k$.

The blocking and scheduling strategy of DSGD is shown in Fig. 1. The rating matrix is divided into $n \times n$ blocks. An update of a block denotes utilizing all the ratings in the block to execute the SGD updates (Formulas (3) and (4)) once. Each iteration needs n rounds of updates. In each round of update, n tasks concurrently compute n non-intersecting blocks, respectively. Updates need to be synchronized: the next round of update should be started only after the current round of update is finished.

0018-9340 (c) 2020 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: Newcastle University. Downloaded on February 17,2021 at 12:44:56 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2020.2997051, IEEE Transactions on Computers

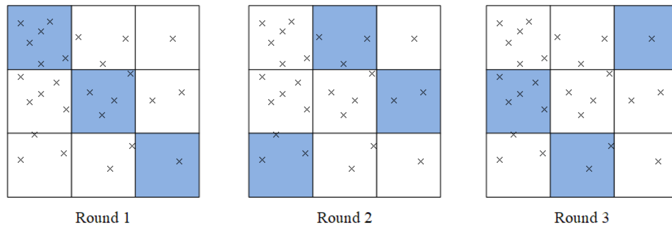

Fig. 1: The schedule of DSGD in one iteration.

2.3 CCD++-based matrix factorization

Based on the CCD algorithm, Hsiang-Fu Yu et al. propose the CCD++ algorithm [11], [12], which is used both in shared memory and distributed memory multi-core environments. Unlike SGD, CCD++ is based on coordinate descent rather than gradient descent.

Let Ω_i be the set of all the ratings of the ith row in R, and $\bar{\Omega}_i$ be the set of all the ratings of the jth column in R. Let $\bar{u}_q$ be the qth column vector of U, i.e., $\bar{u}_q = (u_{1q}, u_{2q}, ..., u_{uq})$; let $\bar{v}_q$ be the qth column vector of V, i.e., $\bar{v}_q = (v_{1q}, v_{2q}, ..., v_{vq})$. Let x be a vector of length u and y be a vector of length v. For u_{iq}, the loss function of CCD++ is defined as follows:

$$f(x_i) = \sum_{j \in \Omega_i} [(r_{ij} - u_i v_j^T + u_{iq} v_{jq} - x_i y_j)^2 + \lambda(||x||^2 + ||y||^2)] \quad (6)$$

$$f(y_j) = \sum_{i \in \bar{\Omega}_j} [(r_{ij} - u_i v_j^T + u_{iq} v_{jq} - x_i y_j)^2 + \lambda(||x||^2 + ||y||^2)] \quad (7)$$

Let $e_{ij} = r_{ij} - u_i v_j^T$, and all e_{ij} be maintained by a single matrix $E^{u \times v}$. RMSE can be directly calculated through $E^{u \times v}$. Our goal is to make $f(x_i)$ and $f(y_j)$ as small as possible, so that RMSE can converge faster. To achieve this goal, CCD++ is solved by the method of coordinate descent.

For Formula (6), taking the partial derivative with respect to x_i, we get

$$\frac{\sum_{j \in \Omega_i} (e_{ij} + u_{iq} v_{jq}) y_j}{\lambda + \sum_{j \in \Omega_i} y_j^2} \quad (8)$$

then the update of u_{iq} is denoted as

$$u_{iq} \leftarrow x_i \quad (9)$$

Similarly, for Formula (7), taking the partial derivative with respect to y_i, we get

$$\frac{\sum_{i \in \bar{\Omega}_j} (e_{ij} + u_{iq} v_{jq}) x_i}{\lambda + \sum_{i \in \bar{\Omega}_j} x_i^2} \quad (10)$$

then the update of v_{jq} can be denoted as

$$v_{jq} \leftarrow y_j \quad (11)$$

The update of e_{ij} can be denoted as

$$e_{ij} \leftarrow e_{ij} + u_{iq} v_{jq} - x_i y_j \quad (12)$$

In the CCD++ iteration, the updating order of U and V is: $\bar{u}_1, \bar{v}_1, \bar{u}_2, \bar{v}_2, ..., \bar{u}_q, \bar{v}_q, ..., \bar{u}_k, \bar{v}_k$

The update of $\bar{u}_q$ is to utilize Formulas (8) and (9) to calculate $u_{1q}, u_{2q}, ..., u_{uq}$ one by one; and the update of $\bar{v}_q$ is to utilize Formulas (10) and (11) to calculate $v_{1q}, v_{2q}, ..., v_{uq}$ one by one. After updating each group $(\bar{u}_q, \bar{v}_q)$, we get e_{ij} by Formula (12).

Similar to DSGD, CCD++ is also implemented in parallel by blocking technique. A partitioning example of CCD++ is shown in Fig. 2. $\bar{u}_q$ and $\bar{v}_q$ can be partitioned into sub-vectors by dividing the indices equally [12]. Suppose that CCD++ is parallelized under n tasks, $\bar{u}_q$ is equally partitioned into $\bar{u}_q^1, \bar{u}_q^2, ..., \bar{u}_q^n$, and $\bar{v}_q$ is partitioned into $\bar{v}_q^1, \bar{v}_q^2, ..., \bar{v}_q^n$. The rating matrix R is partitioned into n row blocks and n column blocks, respectively. The row indices of the ith row block correspond to $\bar{u}_q^i$, and the column indices of the ith column block correspond to $\bar{v}_q^i$.

CCD++ is parallelized by updating each $(\bar{u}_q, \bar{v}_q)$ in parallel. $\bar{u}_q$ and $\bar{v}_q$ are updated in parallel by calculating $\bar{u}_q^i$ and $\bar{v}_q^i$ with the ith task, where $i = 1, 2, ..., n$. After each $\bar{u}_q$ or $\bar{v}_q$ finishes updating, we need to synchronize the tasks.

2.4 Challenges in DSGD and CCD++

DSGD and CCD++ are both confronted with load imbalance challenge, but the causes are slightly different.

The update time of one block is determined by the number of ratings it contains. If a group of blocks is updated in parallel, the elapsed time needed to finish the update of all blocks is determined by the block with most ratings. If the number of ratings varies significantly between these blocks, load balance will become more serious a problem.

Fig. 1 demonstrates that the load imbalance of DSGD is due to the uneven distribution of ratings between sub-blocks, while Fig. 2 illustrates that the load imbalance of CCD++ is due to the uneven distribution of ratings between row blocks and between column blocks, respectively.

3 Balanced Partitioning and its Applications

To improve the load balance existing in algorithms like DSGD and CCD++, we introduce a novel partitioning approach BaPa to prompt ratings to be distributed as evenly as possible across row blocks, across column blocks, and across sub-blocks, respectively.

This section introduces the approach in terms of algorithm and definitions (Section 3.1), formal proof and empirical validation (Section 3.2), and application in DSGD and CCD++ (Section 3.3).

3.1 Algorithm and definitions

We partition matrix $R \in \mathbb{R}^{u \times v}$ into $m \times n$ blocks, i.e., m rows and n columns, where $m \neq n$ is allowed. Let $\mathcal{U} = \{1, 2, ..., u\}$ be the set of row indices. Fixing columns, we divide $\mathcal{U}$ into m disjoint subsets $\mathcal{U}_1, \mathcal{U}_2, ..., \mathcal{U}_m$ sorted by index in ascending order, i.e., $\mathcal{U}_1 = \{1, 2, ..., i\}, \mathcal{U}_2 = \{i+1, i+2, ..., j\}$, and so on. Let $\mathcal{U}^* = \{\mathcal{U}_1, \mathcal{U}_2, ..., \mathcal{U}_m\}$ denote the set of subsets obtained from $\mathcal{U}$. Similarly, fixing rows, we divide the column index set $\mathcal{V} = \{1, 2, ..., v\}$ into $\mathcal{V}^* = \{\mathcal{V}_1, \mathcal{V}_2, ..., \mathcal{V}_n\}$.

Definition 1 (Row block, Column block and Sub-block)**.** A block is denoted by $(\mathcal{X}, \mathcal{Y})$, where $\mathcal{X}$ represents the row index set of the block while $\mathcal{Y}$ represents the column index set. Let $i \in \{1, ..., m\}$ and $j \in \{1, ..., n\}$. The ith row block is denoted as $(\mathcal{U}_i, \mathcal{V})$, the jth column block is denoted as $(\mathcal{U}, \mathcal{V}_j)$, and the sub-block of the ith row and the jth column is denoted as $(\mathcal{U}_i, \mathcal{V}_j)$.

0018-9340 (c) 2020 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: Newcastle University. Downloaded on February 17,2021 at 12:44:56 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2020.2997051, IEEE Transactions on Computers

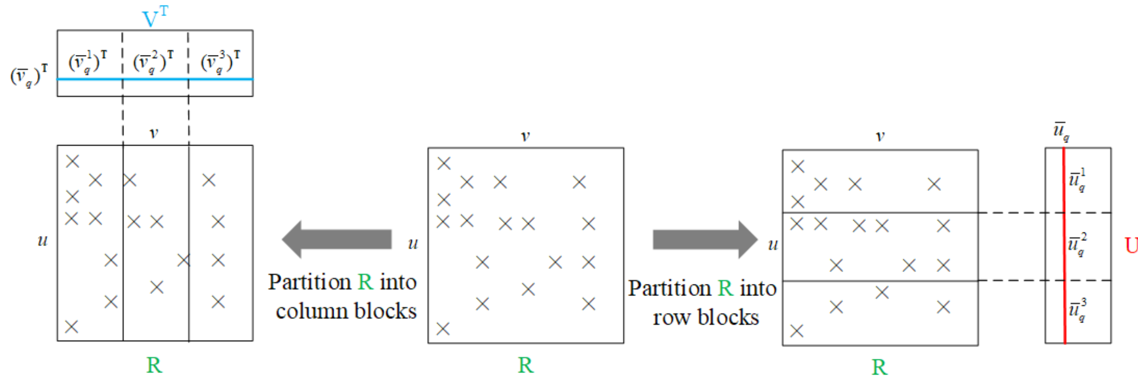

Fig. 2: A partitioning example for CCD++.

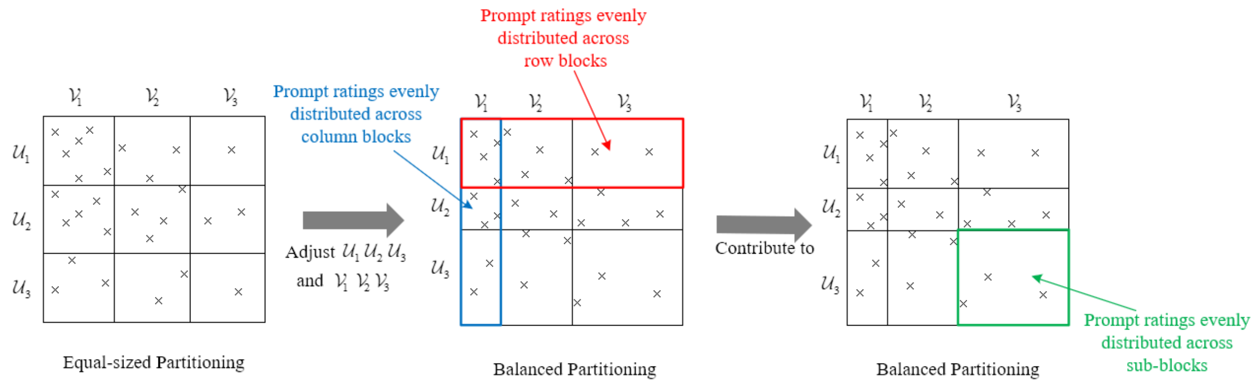

Fig. 3: Equal-sized Partitioning vs Balanced Partitioning.

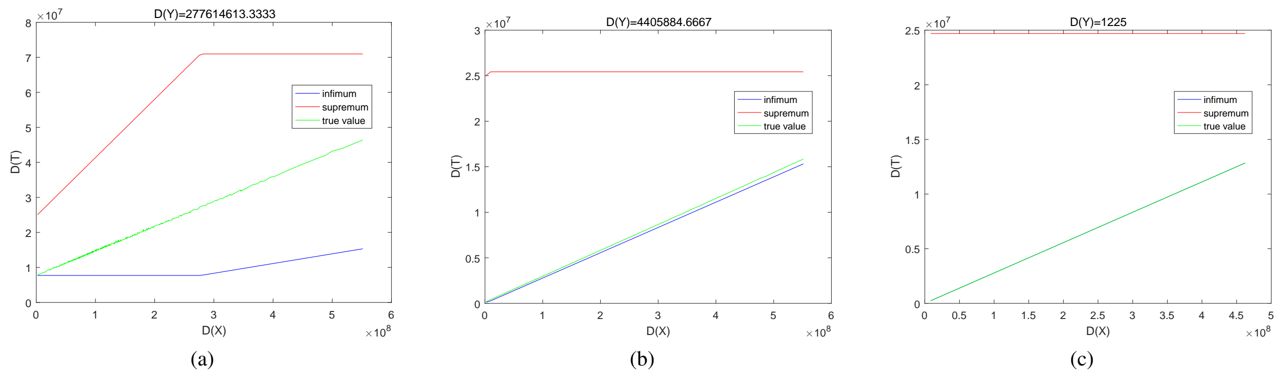

Fig. 4: Changes of $D(T)$ with $D(X)$ and $D(Y)$.

If not otherwise specified, we collectively refer to row block, column block and sub-block as block in the following texts.

Definition 2 (Number of ratings in blocks)**.** Let a_{ij} be the number of ratings in sub-block $(\mathcal{U}_i, \mathcal{V}_j)$.

We use $\langle \mathcal{U}_i, \mathcal{V}_j \rangle$ to represent the number of ratings in $(\mathcal{U}_i, \mathcal{V}_j)$.

$$\langle \mathcal{U}_i, \mathcal{V}_j \rangle = a_{ij} \tag{13}$$

The number of ratings in row block $(\mathcal{U}_i, \mathcal{V})$ is denoted as

$$\langle \mathcal{U}_i \rangle = \sum_{j=1}^{n} a_{ij} \tag{14}$$

The number of ratings in column block $(\mathcal{U}, \mathcal{V}_j)$ is denoted as

$$\langle \mathcal{V}_j \rangle = \sum_{i=1}^{m} a_{ij} \tag{15}$$

The original partitioning method of DSGD and CCD++ satisfies $|\mathcal{U}_1| = |\mathcal{U}_2| = ...|\mathcal{U}_m| = u/m, |\mathcal{V}_1| = |\mathcal{V}_2| = ...|\mathcal{V}_n| = v/n$. This method equally divides row (column) indices into subsets of equal size, so we named it Equal-sized Partitioning (EsPa). However, EsPa does not contribution to load balance, because it does not consider the issue of even distribution of ratings across blocks (equal size of blocks unnecessarily contain the equal number of ratings).

0018-9340 (c) 2020 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: Newcastle University. Downloaded on February 17,2021 at 12:44:56 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2020.2997051, IEEE Transactions on Computers

Algorithm 1: Balanced Partitioning (BaPa)

input : Rating Matrix $R^{u \times v}$, sum, m, n
output: row blocks $(\mathcal{U}_i, \mathcal{V})$, column blocks $(\mathcal{U}, \mathcal{V}_j)$ and sub-blocks $(\mathcal{U}_i, \mathcal{V}_j)$ for $\forall i \in \{1, 2, ..., m\}$ and $\forall j \in \{1, 2, ..., n\}$

```
1  begin
2    /* Part1: Scan all ratings in R to
        compute |Ω_i| and |Ω̄_j| */
3    Set |Ω_i|(i = 1,2,...,u) and |Ω̄_j|(j = 1,2,...,v) to
     0;
4    for each r_ij in R do
5      |Ω_i| ← |Ω_i| + 1;
6      |Ω̄_j| ← |Ω̄_j| + 1;
7    end
8    /* Part2: Partition rows via greedy
        algorithm */
9    Set U_i(i = 1,2,...,m) to empty;
10   t ← 1, s ← 0;
11   for i ← 1 to m − 1 do
12     while
        |s + |Ω_t| − i ∗ (sum/m)| < |s − i ∗ (sum/m)|
       do
13       Add t to U_i;
14       s ← s + |Ω_t|;
15       t ← t + 1;
16     end
17   end
18   Add t, t + 1, ..., u to U_m;
19   /* Part3: Partition columns via
        greedy algorithm */
20   Set V_j(j = 1,2,...,n) to empty;
21   t ← 1, s ← 0;
22   for j ← 1 to n − 1 do
23     while
        |s + |Ω̄_t| − j ∗ (sum/n)| < |s − j ∗ (sum/n)|
       do
24       Add t to V_j;
25       s ← s + |Ω̄_t|;
26       t ← t + 1;
27     end
28   end
29   Add t, t + 1, ..., v to V_n;
30   /* Part4: Sub-block partitioning */
31   Partition R into sub-blocks according to U*, V*;
32 end
```

To improve load balance, our goal is to render each $\langle \mathcal{U}_i, \mathcal{V}_j \rangle$ to be as close as possible to its expectation sum/mn. To achieve the goal, we propose an approach, **BaPa**, by partitioning $\mathcal{U}, \mathcal{V}$ to prompt each $\langle \mathcal{U}_i \rangle$ to be as close as possible to its expectation sum/m, and each $\langle \mathcal{V}_i \rangle$ to be as close as possible to its expectation sum/n.

BaPa makes sense only when **the load balance of sub-blocks can be achieved by the load balance of row blocks and the load balance of column blocks, respectively and independently**. Its rationale is formally proved and experimentally validated in section 3.2. BaPa outperforms EsPa in terms of load balance. Fig. 3 shows the fundamental ideas about EsPa and BaPa.

Let $|\Omega_i|$ be the number of ratings in the ith row, and $|\bar{\Omega}_j|$ be the number of ratings in the jth column. Another approach to compute $\langle \mathcal{U}_i \rangle$ and $\langle \mathcal{V}_j \rangle$ is let $\langle \mathcal{U}_i \rangle = \sum_{t \in \mathcal{U}_i} |\Omega_t|$ and $\langle \mathcal{V}_j \rangle = \sum_{t \in \mathcal{V}_j} |\bar{\Omega}_t|$, making BaPa easier to be implemented in a sparse matrix stored by indices.

Algorithm 1 shows the BaPa procedure. We use a greedy strategy to balance the ratings in row blocks (Part2) and column blocks (Part3) individually. The greedy strategy is not the only choice for 1D partitioning in Algorithm 1. The 1D partitioning problem can be considered as a Chains-on-Chains Partitioning Problem (CCPP) [33] or a Multiprocessor Scheduling Problem (MSP), so any algorithm that solves either CCPP or MSP can be used for 1D partitioning. We use a greedy algorithm here because it is effective and inexpensive. Note that there are three consecutive loops in the algorithm. The time complexity of the first loop is $O(sum)$ because it loops sum times. The complexity of the second loop is $O(u)$, and that of the third loop is $O(v)$, because in the worst case the operation $t \leftarrow t + 1$ runs u times in the second loop and v times in the third loop, respectively. Thus, the time complexity of Algorithm 1 is $O(sum + u + v)$.

In Section (3.2), we formally prove that Algorithm 1 achieves an even distribution of ratings in sub-blocks.

3.2 Rationale

We use variance to measure "the degree of balance". A smaller variance denotes a more uniform distribution of ratings across blocks. Let the random variables X, Y and T represent the number of ratings in row blocks, column blocks, and sub-blocks, respectively, then we use $D(X)$, $D(Y)$ and $D(T)$ to denote the variances of X, Y and T, respectively.

$$D(X) = \frac{1}{m} \sum_{i=1}^{m} (\langle \mathcal{U}_i \rangle - \frac{sum}{m})^2 \tag{16}$$

$$D(Y) = \frac{1}{n} \sum_{i=1}^{n} (\langle \mathcal{V}_i \rangle - \frac{sum}{n})^2 \tag{17}$$

$$D(T) = \frac{1}{mn} \sum_{i=1}^{m} \sum_{j=1}^{n} (\langle \mathcal{U}_i, \mathcal{V}_j \rangle - \frac{sum}{mn})^2 \tag{18}$$

The smaller $D(X)$, $D(Y)$, and $D(T)$ indicate the more uniform distribution of ratings across row blocks, across column blocks, and across sub-blocks, respectively.

3.2.1 Formal proof

To validate the claims, "the load balance of sub-blocks can be achieved by the load balance of row blocks and the load balance of column blocks, respectively and independently", we should prove that: $D(T)$ is reduced simultaneously by decreasing $D(X)$ and $D(Y)$, respectively and independently. We indicate such a mission as the following Theorem 1.

Theorem 1. *Both the supremum and the infimum of $D(T)$ are related to $D(X)$ and $D(Y)$, which has the following guarantees:*

$$\inf\{D(T)\} \leq D(T) \leq \sup\{D(T)\} \tag{19}$$

where

$$\inf\{D(T)\} = \max\{\frac{D(X)}{n^2}, \frac{D(Y)}{m^2}\}$$

$$\sup\{D(T)\} = \min\{\frac{D(X)}{n} + \frac{sum^2}{m^2 n}, \frac{D(Y)}{m} + \frac{sum^2}{mn^2}\} - \frac{sum^2}{m^2 n^2}$$

0018-9340 (c) 2020 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: Newcastle University. Downloaded on February 17,2021 at 12:44:56 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2020.2997051, IEEE Transactions on Computers

TABLE 1: Datasets and Parameter Settings

Dataset name	MovieLens 100K	MovieLens 1M	MovieLens 10M	MovieLens 20M	Netflix	Netflix2
u	943	6040	69878	138493	480189	480189
v	1682	3952	10677	26744	17770	17770
sum	80000	1000209	10000054	20000263	100480507	100480507
k	5	5	5	5	5	5
λ	0.05	0.05	0.05	0.05	0.05	0.05
α	0.0005	0.00005	0.00001	0.000005	0.000001	0.000001

The proof of Theorem 1 is shown in appendix A.

Theorem 1 says that $\sup\{D(T)\}$ and $\inf\{D(T)\}$ decrease with $D(X)$ and $D(Y)$ when m, n and sum remain unchanged. The decreases of $\sup\{D(T)\}$ and $\inf\{D(T)\}$ narrow the range of $D(T)$ and reduce $D(T)$, resulting in a more even rating distribution across blocks.

Both the DSGD and CCD++ use $n \times n$ partition, i.e., the number of row blocks is the same as column blocks. We consider the $m = n$ case of Theorem 1:

$$\inf\{D(T)\} = \max\{\frac{D(X)}{n^2}, \frac{D(Y)}{n^2}\}$$

$$\sup\{D(T)\} = \min\{\frac{D(X)}{n}, \frac{D(Y)}{n}\} + \frac{sum^2(n-1)}{n^4}$$

Note that even we achieve $D(X) = 0$ and $D(Y) = 0$, we may not have $D(T) = 0$. We explain this in the following corollary:

Corollary 1. Consider the $m = n$ case of Theorem 1. When $D(X) = 0$ and $D(Y) = 0$, $D(T)$ varies between 0 and $\frac{sum^2(n-1)}{n^4}$.

$D(T) = 0$ when the number of ratings in each sub-block is the same and equals to sum/n^2.

$D(T) = \frac{sum^2(n-1)}{n^4}$ when there are n sub-blocks, each of them has sum/n ratings, and any two of them are not in the same row or same column.

The proof of Corollary 1 is shown in Appendix B.

Corollary 1 says that whether $D(T)$ is closer to $\inf\{D(T)\}$ or $\sup\{D(T)\}$ depends on how the ratings distribute among sub-blocks after partitioning. Hence, whether BaPa will achieve a good balance or a bad one is unpredictable. Our experiment in Section 3.2.2 shows that BaPa usually achieves a good balance in recommendation datasets.

3.2.2 *Empirical validation*

To validate Theorem 1, we evaluate the impacts of $D(X)$ and $D(Y)$ on $D(T)$ by experiments. We run experiments on MovieLens 100K dataset, dividing the rating matrix into a 6×6 partition. We vary $D(X)$ and $D(Y)$ by manipulating the partitions of $\mathcal{U}$ and $\mathcal{V}$, respectively.

According to Theorem 1, $D(X)$ and $D(Y)$ affect the supremum and infimum of $D(T)$ independently. Fig. 4 shows the changes of $D(T)$ with $D(X)$ under 4 different $D(Y)$. Note that the horizontal parts of the curves of supremum and infimum are caused by the fixed $D(Y)$.

The experimental results as follows confirm Theorem 1.

(1) The real value of $D(T)$ is always between the supremum and the infimum.

(2) For smaller $D(X)$ with fixed $D(Y)$, $D(T)$ is closer to its infimum, and the supremum is often loose.

(3) Reducing $D(X)$ and $D(Y)$ simultaneously, rather than either $D(X)$ or $D(Y)$, is a more effective way to reduce $D(T)$.

3.3 Balanced Partitioning on DSGD and CCD++

In this section, we introduce how to optimize DSGD and CCD++ with BaPa, respectively.

According to Section 2.4, the load imbalance of DSGD is caused by uneven rating distribution across sub-blocks. Accordingly, we improve the load balance of DSGD by reducing $D(T)$ to prompt ratings to be distributed across sub-blocks as evenly as possible.

The load imbalance of CCD++ is caused by uneven rating distribution across row blocks and across column blocks, respectively. Similarly, we improve the load balance of CCD++ by reducing $D(X)$ and $D(Y)$ to prompt ratings to be distributed across row blocks and across column blocks as evenly as possible, respectively.

Since both DSGD and CCD++ adopt the $n \times n$ partition, we apply the $m = n$ case of BaPa to DSGD and CCD++.

4 EXPERIMENTS

For the convenience of description, we use EsPa-DSGD and EsPa-CCD++ to denote DSGD and CCD++ based on EsPa, respectively, and use BaPa-DSGD and BaPa-CCD++ to denote DSGD and CCD++ based on BaPa, respectively. We adopt EsPa-DSGD and EsPa-CCD++ as our baselines. For each dataset, we verify the effectiveness of BaPa by comparing BaPa-DSGD and EsPa-DSGD, BaPa-CCD++ and EsPa-CCD++, respectively. Our experiments are designed to answer the following three questions:

(1) Can BaPa reduce variance more than EsPa? (Section 4.2)

(2) Can BaPa increase speedup more than EsPa? (Section 4.3)

(3) Can we build up a model to explain the above two phenomena? (Section 4.4)

4.1 Environment and Parameter Settings

We run all experiments on an MPI-cluster of TianHe-2[1]. Over 700 computing nodes are available in the cluster, each node with two Intel(R) Xeon(R) E5-2692 v2 2.4GHz processors and 64GB of RAM, and each processor with 12 cores and 24 threads. We implement DSGD and CCD++ in C programs and use MPI for parallelization. These programs are parallelized by up to 1535 tasks, each of which is executed by a single core. The hardware architecture of the MPI-cluster is shown in Fig. 6, in which the nodes are connected over a network, and they communicate with each other via MPI.

We employ two methods to optimize our program: (1) Implement MPI communication with blocking sending and non-blocking receiving. More specifically, sending is performed by MPI_Send and receiving is performed by MPI_Irecv and MPI_Wait. This method largely reduces communication time,

1. http://en.nscc-gz.cn/

0018-9340 (c) 2020 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: Newcastle University. Downloaded on February 17,2021 at 12:44:56 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2020.2997051, IEEE Transactions on Computers

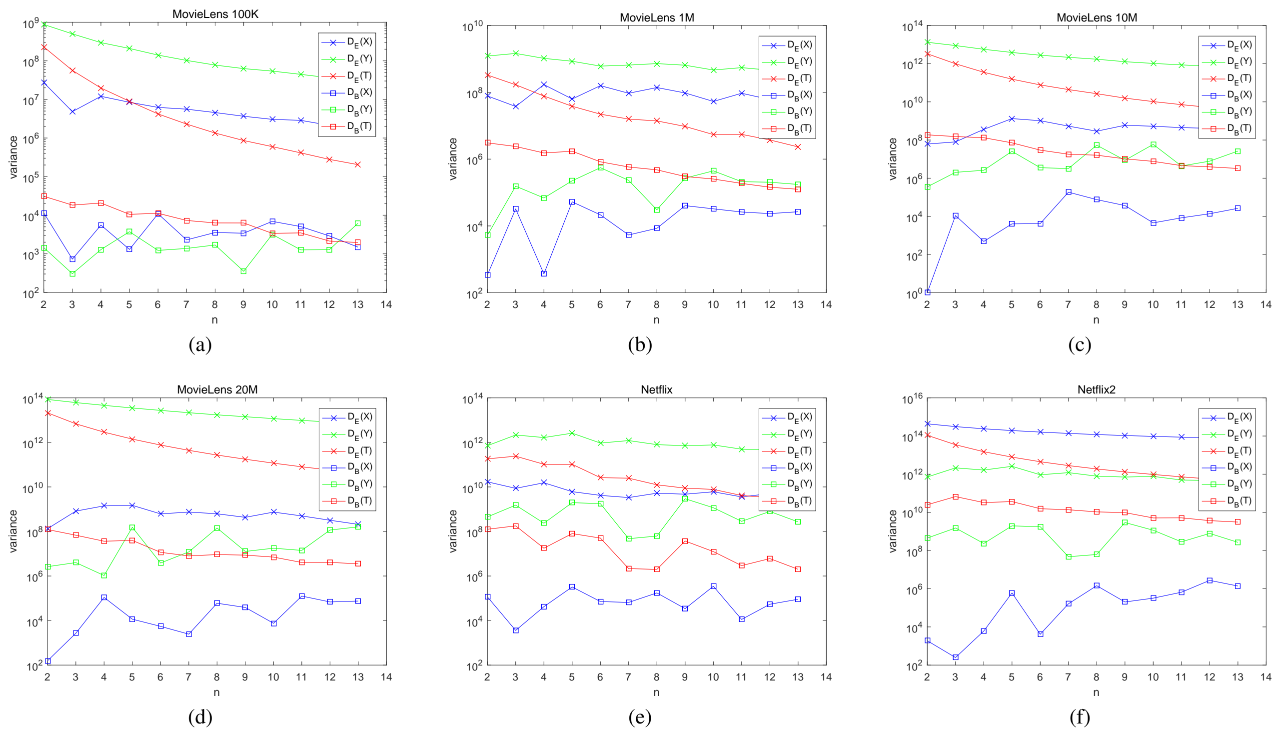

Fig. 5: $D(X)$, $D(Y)$,$D(T)$: Equal-sized Partitioning vs Balanced Partitioning.

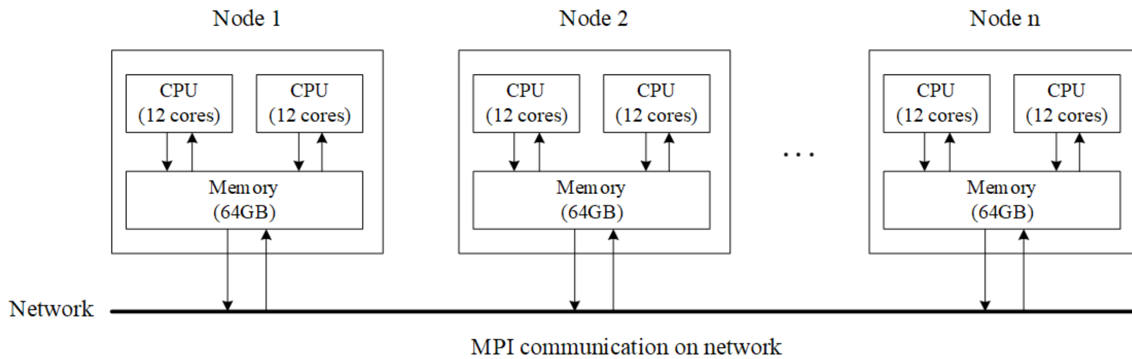

Fig. 6: Hardware architecture of the MPI-cluster of TianHe-2.

especially for the cases of hundreds or thousands of tasks. (2) Distribute the tasks to as many nodes as possible. Although one node has 24 cores and can run at most 24 tasks, we found that the more tasks running on the same node, the higher run time it causes. We are allowed to use up to 64 nodes on TianHe-2. Therefore, when the number of tasks does not exceed 64, we assign each task to a separate node; when the number of tasks exceeds 64, we distribute them evenly to 64 nodes.

We run our experiments on five publicly available datasets: MovieLens[2] 100K, 1M, 10M, 20M and Netflix[3], and a dataset Netflix2 transformed from Netflix. Netflix 2 is obtained by reshuffling the rows of Netflix to make it be a dataset with a more uneven distribution of ratings across rows. In later experiments, it will be clear that Netflix2 is used to compare with Netflix to see how the rating distribution plays a role in the speedup performance.

2. https://grouplens.org/datasets/movielens/
3. https://www.kaggle.com/netflix-inc/netflix-prize-data

Details of these datasets and relevant parameters are shown in Table 1. Note that λ is used for both DSGD and CCD++ while α is used for DSGD only. For DSGD, we initialize each entry of U and V with 1. For CCD++, we initialize each entry of U with 0, and each entry of V with 1.

4.2 Variance: Equal-sized Partitioning vs Balanced Partitioning

In this section, we want to verify that BaPa achieves smaller variance than EsPa by empirically comparing $D(X)$, $D(Y)$, $D(T)$ between EsPa and BaPa.

For each of the datasets in Table 1, we apply EsPa and BaPa to perform the $n \times n$ partitioning, respectively, varying n from 2 to 13. For each partition, we calculate $D(X)$, $D(Y)$ and $D(T)$ by Formulas (16), (17) and (18), respectively. For the calculation based on EsPa, we record $D(X)$, $D(Y)$, $D(T)$ as $D_E(X)$, $D_E(Y)$, $D_E(T)$. For the calculation based on BaPa, we record $D(X)$, $D(Y)$, $D(T)$ as $D_B(X)$, $D_B(Y)$, $D_B(T)$.

0018-9340 (c) 2020 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: Newcastle University. Downloaded on February 17,2021 at 12:44:56 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2020.2997051, IEEE Transactions on Computers

TABLE 2: Convergence: BaPa vs EsPa for DSGD and CCD++

Dataset name	Partitioning	DSGD			CCD++		
		Iterations	Elapsed time ($R(12)$/ms)	RMSE	Iterations	Elapsed time ($R(12)$/ms)	RMSE
MovleLens 100K	EsPa	7389	0.3597	0.9145	1588	0.9927	0.7632
	BaPa	7327	0.2978	0.9145	1588	0.7684	0.7632
MovleLens 1M	EsPa	10000	2.3756	0.9036	3637	7.7842	0.8104
	BaPa	10000	1.6787	0.9036	3637	5.8323	0.8104
MovleLens 10M	EsPa	6447	39.4432	0.8615	5778	212.3144	0.7692
	BaPa	6447	16.4276	0.8615	5778	93.1716	0.7692
MovleLens 20M	EsPa	10000	137.2029	0.8553	3535	753.6797	0.7625
	BaPa	10000	39.8841	0.8553	3535	212.3136	0.7625
Netflix	EsPa	10000	267.9529	0.9216	2735	1101.9628	0.8333
	BaPa	10000	259.8416	0.9216	2735	1063.0161	0.8333
Netflix2	EsPa	10000	520.3908	1.0099	10000	2182.4294	0.9930
	BaPa	10000	163.3855	1.0099	10000	812.2552	0.9930

Fig. 5 illustrates the comparisons between $D_E(X)$ and $D_B(X)$, $D_E(Y)$ and $D_B(Y)$, $D_E(T)$ and $D_B(T)$ for all datasets. We see that $D_B(X)$, $D_B(Y)$, $D_B(T)$ are several orders of magnitude smaller than $D_E(X)$, $D_E(Y)$, $D_E(T)$, respectively. In conclusion, BaPa outperforms EsPa in terms of achieving smaller $D(X)$, $D(Y)$ and $D(T)$.

4.3 Performance of Balanced Partitioning

In this section, we compare the performance between BaPa and EsPa based on DSGD and CCD++ from two aspects: convergence (Section 4.3.1) and speedup (Section 4.3.2).

4.3.1 Convergence

We run experiments under 12 tasks. The iteration stops when either of the following conditions is met: (1) RMSE of current iteration becomes bigger than the previous; (2) The number of iterations reaches the pre-defined maximum 10000.

Table 2 shows the convergence performance of DSGD and CCD++ after BaPa and EsPa under various datasets. In the table, RMSE denotes the final predicated accuracy after convergence. For all datasets, the elapsed time of BaPa is shorter than that of EsPa when convergence occurs, which suggests that BaPa accelerates convergence without impairing accuracy. The major reason is that BaPa leads to a more uniform distribution of ratings across blocks.

Theoretically, the update order of ratings may slightly affect the RMSE of DSGD, for it affects the results of U and V in each iteration. As shown in Table 2, BaPa and EsPa reaches the same RMSE with different iterations for DSGD under MovieLens 100K. It is because BaPa changes the update order of some ratings and affects RMSE. However, the impact is trivial. BaPa does not affect the RMSE of CCD++, because the update order of each u_{iq} in $\bar{u}_q$ and each v_{iq} in $\bar{v}_q$ is independent from U and V at each iteration.

4.3.2 Speedup

We define $R(n)$ as the average elapsed time of an iteration:

$$R(n) = \left(\frac{\text{elapsed time of the program}}{\text{number of iterations}} \right)_{\text{under } n \text{ tasks}} \quad (20)$$

where "elapsed time of the program" and "number of iterations" are measured by experiments. The elapsed time includes two parts: the time of block computation and the time of MPI communication. BaPa prompts the balance of ratings across blocks, so it mainly reduces the time of block computation, not the time of MPI communication. Indeed, the communication time is usually far less than computation time except that the number of tasks becomes too large. For the synchronous algorithm, we need to calculate RMSE once after each iteration, so the number of iterations is equal to the number of RMSE calculations.

We evaluate the speed of a program by $R(n)$. $R(n)$ takes the average of multiple tests.

Let $Speedup(n)$ be the speedup of the program under n tasks, i.e.,

$$Speedup(n) = \frac{R(1)}{R(n)} \quad (21)$$

Fig. 7 shows $Speedup(n)$s of BaPa and EsPa for both DSGD and CCD++ under all datasets, varying n from 1 to 13. Note that n tasks correspond to the $n \times n$ partitioning of the rating matrix. Further, we use more tasks to test the scalability of BaPa. We test the speedups under 16, 32, 64, 96, 128, 192, 256, 384, 512, 1024, and 1535 tasks, respectively, and the results are shown in Fig. 8.

According to Fig. 7 and Fig. 8, we have the following three judgments.

(1) In most cases, BaPa performs higher speedup than EsPa for both DSGD and CCD++ under all datasets. The reason is that BaPa achieves a more even rating distribution across blocks than EsPa.

(2) BaPa has good scalability: it outperforms EsPa under tens to hundreds of tasks. Fig. 8 shows that, in most cases, EsPa's speedups are smaller than the peak speedups of BaPa, which is the advantage of BaPa. As the number of tasks increases, BaPa's speedup performance may decline or even be smaller than EsPa in some cases. The main reason is that when the number of tasks is too large, BaPa costs more communication time than EsPa, because blocks of the former are more unbalanced. However, in no case can EsPa surpass the peak of BaPa, which demonstrates the higher scalability performance of BaPa.

(3) The speedup growth is significant in MovieLens 10M, MovieLens 20M, and Netflix2, trivial in MovieLens 1M and Netflix, but divided in MovieLens 100K. The reason leading to the results seems divided, but we give it a consistent explanation by introducing the Imbalance Degree Model.

0018-9340 (c) 2020 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: Newcastle University. Downloaded on February 17,2021 at 12:44:56 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2020.2997051, IEEE Transactions on Computers

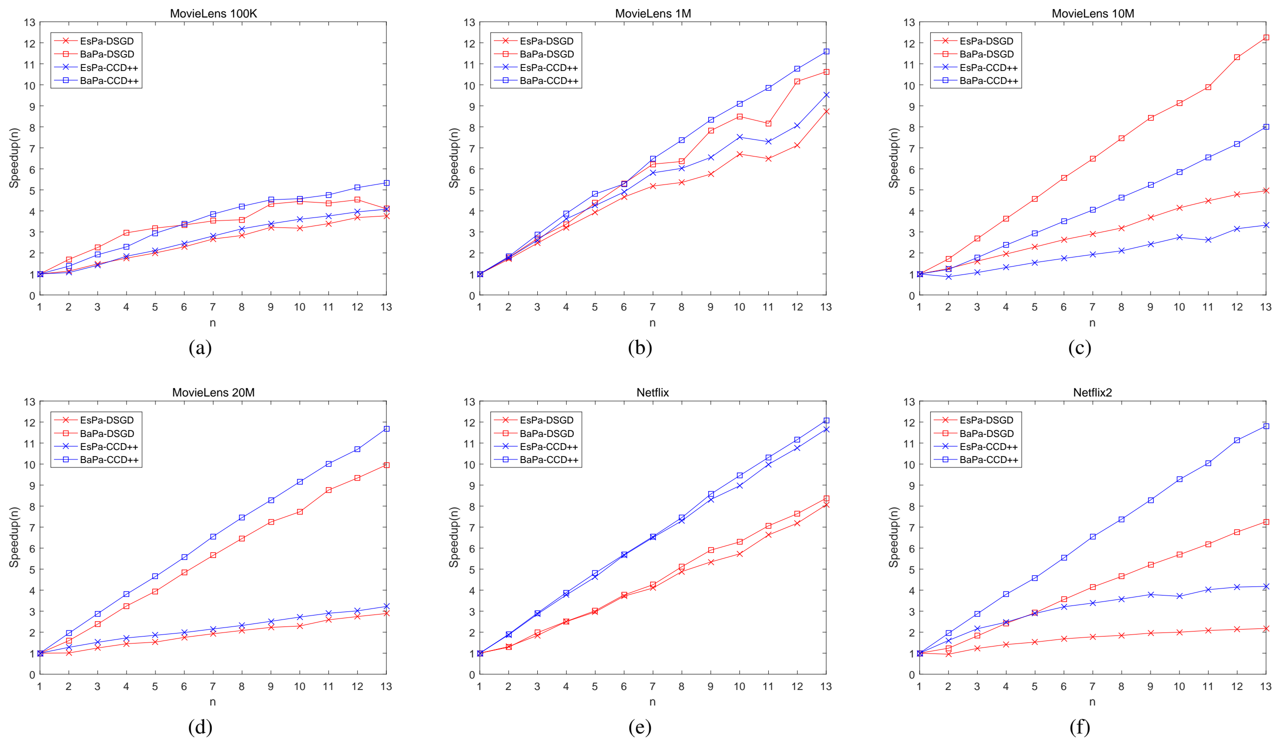

Fig. 7: Speedup of 1-13 tasks: EsPa-DSGD vs BaPa-DSGD, EsPa-CCD++ vs BaPa-CCD++.

4.4 Imbalance Degree model

The following Theorem 2 shows the Imbalance Degree model and its essential properties, which indicate how the rating distribution across blocks is used to estimate the speedup growth and explain why BaPa significantly impacts on some datasets but trivially on others.

Theorem 2 (Imbalance Degree Model)**.** *Let sum be the number of ratings in the matrix, and n be the number of tasks (equivalent to $n \times n$ partition). We use P to represent Imbalance Degree (IbD).*

Specifically, P_D is for DSGD,

$$P_D = \theta \cdot \frac{n^2}{sum} \sqrt{D_E(T)} \tag{22}$$

and P_C is for CCD++,

$$P_C = \theta \cdot \frac{n}{2 \cdot sum} (\sqrt{D_E(X)} + \sqrt{D_E(Y)}) \tag{23}$$

where $\theta = \Phi^{-1}(\sqrt[n]{0.5})$ and Φ is the cumulative distribution function of the standard normal distribution.

Let $Speedup_B(n)$ be the $Speedup(n)$ of BaPa and $Speedup_E(n)$ be the $Speedup(n)$ of EsPa. We define the speedup growth S as follows:

$$S = \frac{Speedup_B(n)}{Speedup_E(n)} \tag{24}$$

The following Formula indicates the relationship between S with P:

$$S \approx 1 + P \tag{25}$$

The proof of Theorem 2 is shown in appendix C.

According to Formulas (22) and (23), P is determined by variances $D_E(X)$, $D_E(Y)$ and $D_E(T)$, as well as sum and n. P is positively correlated to the variances and n, and is negatively correlated to sum.

A large S, as well as a large P, indicates the speedup growth is significant. In addition, a larger P indicates a more imbalanced rating distribution across blocks. We conduct experiments to obtain S and calculate P by Formulas (22) and (23). The results are illustrated in Fig. 9, which demonstrates that the relationship between S and P coincides with Formula (25).

More importantly, Fig. 9 explains further the Judgment (3) in Section (4.3.2). In the figure, S_D is the S for DSGD and S_C is the S for CCD++.

(1) The reason why the speedup growth of MovieLens 10M, MovieLens 20M, and Netflix2 are significant is the P values of these datasets are big; while the reason why the speedup growth of MovieLens 1M and Netflix are trivial is the P values of these datasets are small.

(2) The speedup growth of MovieLens 100K seems divided. When n is small, the above rule still satisfies. When n is big enough (bigger than 5 in the experiment), it is no longer possible to enhance speedup only by increasing n since MovieLens 100K is a dataset of small size.

Imbalance Degree model matches well with the experiment results in Section (4.2) and Section (4.3). It explains under what conditions BaPa performs well, and can be used to predict the potential speedup from applying BaPa.

4.5 Summary

We summarize the above experimental results as follows, which answer the three questions raised at the beginning of Section 4:

(1) BaPa reduces the variances $D(X), D(Y), D(T)$ by several orders of magnitude more than EsPa.

0018-9340 (c) 2020 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: Newcastle University. Downloaded on February 17,2021 at 12:44:56 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2020.2997051, IEEE Transactions on Computers

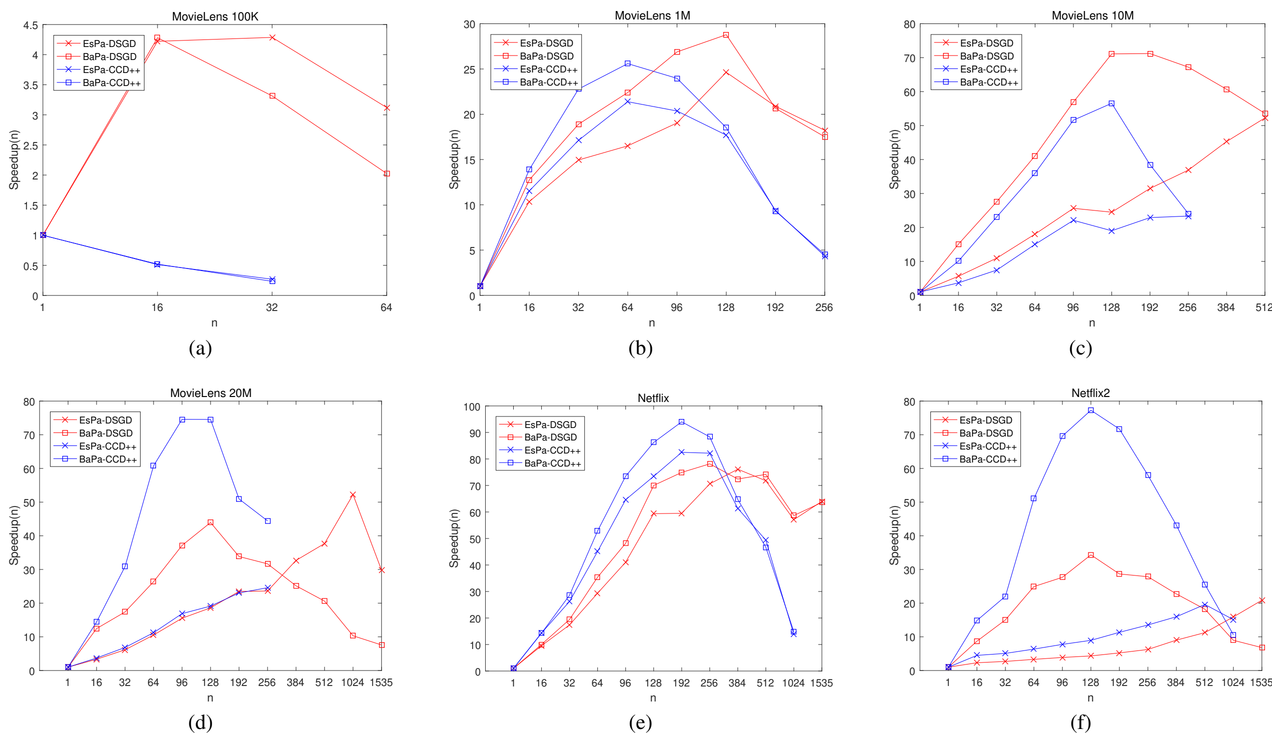

Fig. 8: Speedup of 1-1535 tasks: EsPa-DSGD vs BaPa-DSGD, EsPa-CCD++ vs BaPa-CCD++.

(2) BaPa outperforms EsPa in terms of speedup in all cases, and it performs better in more imbalanced datasets.

(3) Imbalance Degree model explains why BaPa outperforms EsPa.

5 CONCLUSIONS AND FUTURE WORK

BaPa seems simple, but it challenges the common belief that the balancing of sub-blocks cannot be achieved by independently balancing rows and columns. The contributions rest in the formal proofs and the experimental results. We formally show why and how BaPa works through Theorem 1 and empirically demonstrate its soundness; then we establish an Imbalance Degree model (Theorem 2) to reveal its working mechanism further.

Based on all the efforts and evidence, we believe that BaPa is a piece of significant work. Although it is merely applied to DSGD and CCD++ in our current study, it can be similarly applied to other matrix factorization algorithms since it is a general blocking model.

Although BaPa and Imbalance Degree work effectively, they merely provide a rough partitioning of sub-blocks and a rough estimate of speedup growth. We notice that the ideal row block partitioning and the ideal column block partitioning are an NP-hard problem [12]. Therefore, we believe that an ideal sub-block partitioning (the variance of ratings in sub-blocks reaches the minimum) is also NP-hard and requires a new idea to achieve it. Besides, an accurate estimate of speedup growth is hard to achieve, because it involves many other factors such as communication latency and memory discontinuity. In addition, the heterogeneous system is a system in which processor and coprocessor cooperate to process computing tasks. The processor is CPU, and the coprocessor can be GPU, MIC, FPGA, etc. The coprocessor has a hardware architecture different from the processor; it is usually used for parallel computing and carries the main computational load. We believe BaPa can benefit from these promising systems. We identify the following three as our future work.

(1) To establish an ideal sub-block partitioning approach and its NP-hard proof.

(2) To establish a more accurate model to estimate speedup growth.

(3) To apply the heterogeneous system to optimize BaPa.

6 ACKNOWLEDGMENT

The study is supported by the National Science Foundation of China (No. U1711266 and No. 41925007).

0018-9340 (c) 2020 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: Newcastle University. Downloaded on February 17,2021 at 12:44:56 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2020.2997051, IEEE Transactions on Computers

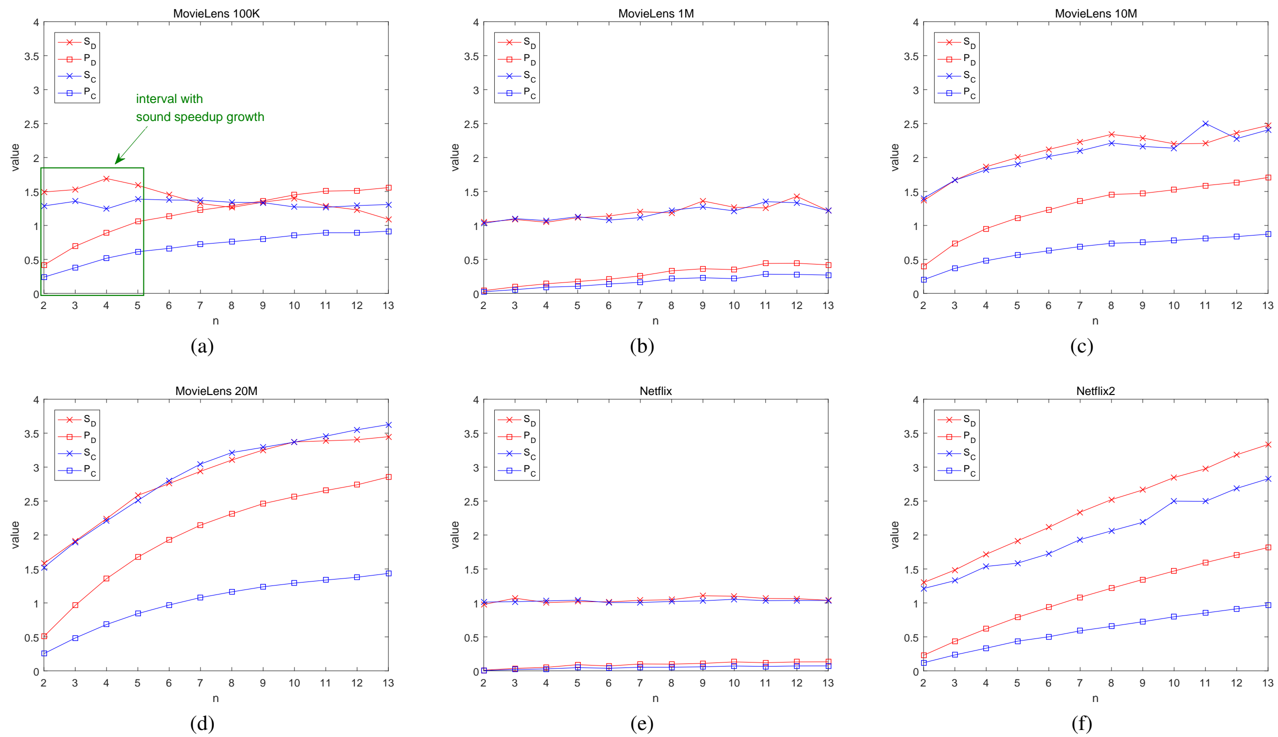

Fig. 9: S and P Values of DSGD and CCD++.

Appendix A
Proof of Theorem 1

Lemma 1. Consider n non-negative rational $a_1, a_2, ..., a_n$ subjecting to $\sum_{i=1}^{n} a_i = t$, where t is a non-negative variable. Let $s = \sum_{i=1}^{n} a_i^2$, then the relationship between s and t can be formulated as $\frac{t^2}{n} \leq s \leq t^2$, which is equivalent to

$$\frac{1}{n}(\sum_{i=1}^{n} a_i)^2 \leq \sum_{i=1}^{n} a_i^2 \leq (\sum_{i=1}^{n} a_i)^2 \tag{26}$$

Proof. Formula (26) can be deduced with Lagrange multiplier method. $s_{min} = \frac{t^2}{n}$ if and only if $a_1 = a_2 = ... = a_n = \frac{t}{n}$. $s_{max} = t^2$ when $a_i = t$ and for each $j \neq i$, $a_j = 0$. □

We consider the effects of either $D(X)$ or $D(Y)$ on $D(T)$ individually. And we discuss $D(X)$ on $D(T)$ first. In Formula (16), $D(X)$ and $D(T)$ can be reduced to

$$\begin{aligned} D(X) &= \frac{1}{m}[\sum_{i=1}^{m}(\sum_{j=1}^{n} a_{ij})^2 - \frac{sum^2}{m}] \\ D(T) &= \frac{1}{mn}[\sum_{i=1}^{m}\sum_{j=1}^{n} a_{ij}^2 - \frac{sum^2}{mn}] \end{aligned} \tag{27}$$

We define $L(X)$ and $L(T)$ as follows

$$\begin{aligned} L(X) &= mD(X) + \frac{sum^2}{m} = \sum_{i=1}^{m}(\sum_{j=1}^{n} a_{ij})^2 \\ L(T) &= mnD(T) + \frac{sum^2}{mn} = \sum_{i=1}^{m}\sum_{j=1}^{n} a_{ij}^2 \end{aligned} \tag{28}$$

From Lemma 1, we can derive that

$$\frac{1}{n}\sum_{i=1}^{m}(\sum_{j=1}^{n} a_{ij})^2 \leq \sum_{i=1}^{m}\sum_{j=1}^{n} a_{ij}^2 \leq \sum_{i=1}^{m}(\sum_{j=1}^{n} a_{ij})^2 \tag{29}$$

which is equivalent to

$$\frac{L(X)}{n} \leq L(T) \leq L(X) \tag{30}$$

By substituting Formula (28) into Formula (30), we have

$$\frac{D(X)}{n^2} \leq D(T) \leq \frac{D(X)}{n} + \frac{sum^2}{m^2 n} - \frac{sum^2}{m^2 n^2} \tag{31}$$

Similar to Formula (31), the relationship between $D(Y)$ and $D(T)$ can be formulated as

$$\frac{D(Y)}{m^2} \leq D(T) \leq \frac{D(Y)}{m} + \frac{sum^2}{mn^2} - \frac{sum^2}{m^2 n^2} \tag{32}$$

Combining Formula (31) and Formula (32), we have

$$\begin{aligned} &\max\{\frac{D(X)}{n^2}, \frac{D(Y)}{m^2}\} \leq D(T) \leq \\ &\min\{\frac{D(X)}{n} + \frac{sum^2}{m^2 n}, \frac{D(Y)}{m} + \frac{sum^2}{mn^2}\} - \frac{sum^2}{m^2 n^2} \end{aligned} \tag{33}$$

Formula (33) is what presented in Theorem 1.

Appendix B
Proof of Corollary 1

$D(X) = 0$ means that the number of ratings in each row block is the same, i.e.,

$$\sum_{j=1}^{n} a_{ij} = \frac{sum}{n} \text{ for each } 1 \leq i \leq n \tag{34}$$

0018-9340 (c) 2020 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: Newcastle University. Downloaded on February 17,2021 at 12:44:56 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2020.2997051, IEEE Transactions on Computers

$D(Y) = 0$ means that the number of ratings in each column block is the same, i.e.,

$$\sum_{i=1}^{n} a_{ij} = \frac{sum}{n} \text{ for each } 1 \leq j \leq n \tag{35}$$

From Lemma 1, $D(T) = \inf\{D(T)\}$ when s_{min} is met, and $D(T) = \sup\{D(T)\}$ when s_{max} is met.

Formula (34) meets s_{min} when

$$a_{i1} = a_{i2} = ... = a_{in} = \frac{sum}{n^2} \text{ for each } 1 \leq i \leq n \tag{36}$$

Formula (35) meets s_{min} when

$$a_{1j} = a_{2j} = ... = a_{nj} = \frac{sum}{n^2} \text{ for each } 1 \leq j \leq n \tag{37}$$

Formula (36) and (37) show that $D(T) = \inf\{D(T)\}$ when $a_{ij} = sum/n^2$ for each $1 \leq i \leq n$ and each $1 \leq j \leq n$.

Formula (34) meets s_{max} when

$$a_{ij} = \frac{sum}{n} \text{ and } a_{ik} = 0 (k \neq j) \text{ for each } 1 \leq i \leq n \tag{38}$$

Formula (35) meets s_{max} when

$$a_{ij} = \frac{sum}{n} \text{ and } a_{kj} = 0 (k \neq i) \text{ for each } 1 \leq j \leq n \tag{39}$$

Formula (38) and (39) show that $D(T) = \sup\{D(T)\}$ when there is only one sub-block having sum/n ratings in each row and each column. That is, there are n sub-blocks, each of them has sum/n ratings and any two of them are not in the same row or same column, while other $n(n-1)$ sub-blocks has no ratings.

Appendix C
Proof of Theorem 2

Lemma 2. Let X_t i.i.d. $N(\mu, \sigma^2)$, where $t = 1, 2, ..., n$. Then we have

$$\begin{aligned} &\Pr[\max\{X_1, X_2, ..., X_n\} > \mu + \theta \cdot \sigma] \\ &= 1 - \prod_{t=1}^{n} (\Pr[X_t < \mu + \theta \cdot \sigma]) \end{aligned} \tag{40}$$

where $\Phi(x)$ is the cumulative distribution function of $N(0, 1)$ and θ is a variable.

For either algorithm, let $R(n)$ be the average runtime of one iteration with n processes, and each iteration consists of s rounds of synchronous updating ($s = n$ for DSGD, $s = 2k$ for CCD++). In each round, we update n blocks in parallel (sub-blocks for DSGD, row or column blocks for CCD++). Let X_t be the number of ratings of the tth block, where $t = 1, 2, ..., n$, and we define $M(n) = \max\{X_1, X_2, ..., X_n\}$. For simplicity, we suppose $M(n)$ is identical for each round. Then, the runtime of each round can be denoted as $c \cdot M(n)$, and $R(n) \approx c \cdot s \cdot M(n)$, where c is a constant. Therefore, S is derived as

$$\begin{aligned} S &= \frac{Speedup_B(n)}{Speedup_E(n)} = \frac{R_B(1)/R_B(n)}{R_E(1)/R_E(n)} = \frac{R_E(n)}{R_B(n)} \\ &\approx \frac{c \cdot s \cdot M_E(n)}{c \cdot s \cdot M_B(n)} = \frac{M_E(n)}{M_B(n)} \end{aligned} \tag{41}$$

TABLE 3: The mapping of $n \to \theta$

n	$\Phi(\theta) = \sqrt[n]{0.5}$	$\theta = \Phi^{-1}(\sqrt[n]{0.5})$
2	0.7071	0.55
3	0.7937	0.82
4	0.8409	1.00
5	0.8706	1.13
6	0.8909	1.23
7	0.9057	1.32
8	0.9170	1.39
9	0.9259	1.45
10	0.9330	1.50
11	0.9389	1.55
12	0.9439	1.59
13	0.9481	1.63

Formula (41) shows S can be indirectly calculated by estimating $M_E(n)$ and $M_B(n)$.

Let's consider the situation of DSGD first. We establish an interval estimation for $M(n)$. Suppose X_t i.i.d. $N(\mu, \sigma^2)$, where $\mu = sum/n^2$ and $\sigma^2 = D(T)$. Let $sum/n^2 + \theta\sqrt{D(T)}$ be the estimated value of $M(n)$, where θ is a variable. According to lemma 2, for any $\theta \in \mathbb{R}$,

$$\begin{aligned} &\Pr[M(n) > \frac{sum}{n^2} + \theta\sqrt{D(T)}] \\ &= \Pr[\max\{X_1, X_2, ..., X_n\} > \frac{sum}{n^2} + \theta\sqrt{D(T)}] \\ &= 1 - [\Phi(\theta)]^n \end{aligned} \tag{42}$$

Note that the probability is related to θ and n, but independent of μ and σ. To make a reasonable estimation, we keep the probability identical for any θ and n by adjusting θ according to n. The identity value of probability is optional, we set 0.5 as the default, i.e., $1 - [\Phi(\theta)]^n = 0.5$. Then the mapping $n \to \theta$ is denoted as

$$\theta = \Phi^{-1}(\sqrt[n]{0.5}) \tag{43}$$

In the formula above, Φ^{-1} can be easily solved by referring a standard normal table. We list the mapping for $n = 2, 3, ..., 13$ in Table 3. It is obvious that θ is a positive correlation with n.

With θ following Formula (43), $M_E(n)$ and $M_B(n)$ is denoted as:

$$\begin{aligned} M_E(n) &\approx \frac{sum}{n^2} + \theta\sqrt{D_E(T)} \\ M_B(n) &\approx \frac{sum}{n^2} + \theta\sqrt{D_B(T)} \end{aligned} \tag{44}$$

By substituting Formula (44) into Formula (41), we derive S as

$$S \approx \frac{\frac{sum}{n^2} + \theta\sqrt{D_E(T)}}{\frac{sum}{n^2} + \theta\sqrt{D_B(T)}} \approx 1 + \theta \cdot \frac{n^2}{sum}\sqrt{D_E(T)} \tag{45}$$

By referring data in Fig. 5, we find $sum/n^2 \gg \theta\sqrt{D_B(T)}$, so we omit $D_B(T)$ in the formula above. Formula (45) is our result, which is equivalent to Formula (25) and validates Theorem 2 for DSGD.

Now we discuss the situation of CCD++. CCD++ updates row blocks or column blocks alternatively in each iteration. We

0018-9340 (c) 2020 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: Newcastle University. Downloaded on February 17,2021 at 12:44:56 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2020.2997051, IEEE Transactions on Computers

calculate the $M(n)$ of row blocks and column blocks respectively, and evaluate the final $M(n)$ by taking their average. Let $M_r(n)$ and $M_c(n)$ be the $M(n)$ of row blocks and column blocks respectively, i.e., $M_r(n) \approx sum/n + \theta\sqrt{D(X)}$, $M_c(n) \approx sum/n + \theta\sqrt{D(Y)}$. The average $M(n)$ of each round is denoted as $M(n) = (M_r(n) + M_c(n))/2$, and we have

$$\begin{aligned} M_E(n) &\approx \frac{1}{2}(2 \cdot \frac{sum}{n} + \theta\sqrt{D_E(X)} + \theta\sqrt{D_E(Y)}) \\ M_B(n) &\approx \frac{1}{2}(2 \cdot \frac{sum}{n} + \theta\sqrt{D_B(X)} + \theta\sqrt{D_B(Y)}) \end{aligned} \tag{46}$$

After substituting Formula (46) into Formula (41), we derive S as

$$\begin{aligned} S &\approx \frac{\frac{1}{2}(2 \cdot \frac{sum}{n} + \theta\sqrt{D_E(X)} + \theta\sqrt{D_E(Y)})}{\frac{1}{2}(2 \cdot \frac{sum}{n} + \theta\sqrt{D_B(X)} + \theta\sqrt{D_B(Y)})} \\ &\approx 1 + \theta \cdot \frac{n}{2 \cdot sum}(\sqrt{D_E(X)} + \sqrt{D_E(Y)}) \end{aligned} \tag{47}$$

Similarly to DSGD, we omit $D_B(X)$ and $D_B(Y)$ in the formula above. Formula (47) is our result, which is equivalent to Formula (25) and validates Theorem 2 for CCD++.

References

[1] C. C. Aggarwal, *Recommender systems*. Springer, 2016.
[2] A. Ramlatchan, M. Yang, Q. Liu, M. Li, J. Wang, and Y. Li, "A survey of matrix completion methods for recommendation systems," *Big Data Mining and Analytics*, vol. 1, no. 4, pp. 308–323, 2018.
[3] Y. Koren, "Factorization meets the neighborhood: a multifaceted collaborative filtering model," in *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2008, pp. 426–434.
[4] Y. Koren, R. Bell, and C. Volinsky, "Matrix factorization techniques for recommender systems," *Computer*, no. 8, pp. 30–37, 2009.
[5] S. J. Wright, "Coordinate descent algorithms," *Mathematical Programming*, vol. 151, no. 1, pp. 3–34, 2015.
[6] I. Pilászy, D. Zibriczky, and D. Tikk, "Fast als-based matrix factorization for explicit and implicit feedback datasets," in *Proceedings of the fourth ACM conference on Recommender systems*. ACM, 2010, pp. 71–78.
[7] Z. Wu, Y. Luo, K. Lu, and X. Wang, "Parallelizing stochastic gradient descent with hardware transactional memory for matrix factorization," in *2018 3rd International Conference on Information Systems Engineering (ICISE)*. IEEE, 2018, pp. 118–121.
[8] O. Kaya, R. Kannan, and G. Ballard, "Partitioning and communication strategies for sparse non-negative matrix factorization," in *47th International Conference on Parallel Processing*. IEEE, 2018.
[9] H. Li, K. Li, J. An, and K. Li, "Msgd: a novel matrix factorization approach for large-scale collaborative filtering recommender systems on gpus," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 7, pp. 1530–1544, 2018.
[10] R. Gemulla, E. Nijkamp, P. J. Haas, and Y. Sismanis, "Large-scale matrix factorization with distributed stochastic gradient descent," in *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2011, pp. 69–77.
[11] H.-F. Yu, C.-J. Hsieh, S. Si, and I. Dhillon, "Scalable coordinate descent approaches to parallel matrix factorization for recommender systems," in *2012 IEEE 12th International Conference on Data Mining*. IEEE, 2012, pp. 765–774.
[12] H.-F. Yu, C.-J. Hsieh, S. Si, and I. S. Dhillon, "Parallel matrix factorization for recommender systems," *Knowledge and Information Systems*, vol. 41, no. 3, pp. 793–819, 2014.
[13] I. Nisa, A. Sukumaran-Rajam, R. Kunchum, and P. Sadayappan, "Parallel ccd++ on gpu for matrix factorization," in *Proceedings of the General Purpose GPUs*. ACM, 2017, pp. 73–83.
[14] B. Joshi, F. Iutzeler, and M.-R. Amini, "Large-scale asynchronous distributed learning based on parameter exchanges," *International Journal of Data Science and Analytics*, vol. 5, no. 4, pp. 223–232, 2018.
[15] Z.-Q. Yu, X.-J. Shi, L. Yan, and W.-J. Li, "Distributed stochastic admm for matrix factorization," in *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management*. ACM, 2014, pp. 1259–1268.
[16] W.-S. Chin, Y. Zhuang, Y.-C. Juan, and C.-J. Lin, "A fast parallel stochastic gradient method for matrix factorization in shared memory systems," *ACM Transactions on Intelligent Systems and Technology (TIST)*, vol. 6, no. 1, p. 2, 2015.
[17] B. Recht, C. Re, S. Wright, and F. Niu, "Hogwild: A lock-free approach to parallelizing stochastic gradient descent," in *Advances in Neural Information Processing Systems*. MIT Press, 2011, pp. 693–701.
[18] H. Yun, H.-F. Yu, C.-J. Hsieh, S. Vishwanathan, and I. Dhillon, "Nomad: Non-locking, stochastic multi-machine algorithm for asynchronous and decentralized matrix completion," *Proceedings of the VLDB Endowment*, vol. 7, no. 11, pp. 975–986, 2014.
[19] J. Oh, W.-S. Han, H. Yu, and X. Jiang, "Fast and robust parallel sgd matrix factorization," in *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 2015, pp. 865–874.
[20] F. Pedregosa, R. Leblond, and S. Lacoste-Julien, "Breaking the nonsmooth barrier: A scalable parallel method for composite optimization," in *Advances in Neural Information Processing Systems*. MIT Press, 2017, pp. 56–65.
[21] Z. Tang, X. Zhang, K. Li, and K. Li, "An intermediate data placement algorithm for load balancing in spark computing environment," *Future Generation Computer Systems*, vol. 78, pp. 287–301, 2018.
[22] J. Zhang, H. Guo, F. Hong, X. Yuan, and T. Peterka, "Dynamic load balancing based on constrained kd tree decomposition for parallel particle tracing," *IEEE transactions on visualization and computer graphics*, vol. 24, no. 1, pp. 954–963, 2018.
[23] E. Wang, Q. Zhang, B. Shen, G. Zhang, X. Lu, Q. Wu, and Y. Wang, *High-Performance Computing on the Intel® Xeon PhiTM: How to Fully Exploit MIC Architectures*. Springer, 2014.
[24] Q. Ho, J. Cipar, H. Cui, S. Lee, J. K. Kim, P. B. Gibbons, G. A. Gibson, G. Ganger, and E. P. Xing, "More effective distributed ml via a stale synchronous parallel parameter server," in *Advances in Neural Information Processing Systems*. MIT Press, 2013, pp. 1223–1231.
[25] A. Beutel, M. Weimer, V. Narayanan, and Y. T. Minka, "Elastic distributed bayesian collaborative filtering," in *NIPS workshop on Distributed Machine Learning and Matrix Computations*. MIT Press, 2014.
[26] A. Das, I. Upadhyaya, X. Meng, and A. Talwalkar, "Collaborative filtering as a case-study for model parallelism on bulk synchronous systems," in *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*. ACM, 2017, pp. 969–977.
[27] A. Buluç and J. R. Gilbert, "Parallel sparse matrix-matrix multiplication and indexing: Implementation and experiments," *SIAM Journal on Scientific Computing*, vol. 34, no. 4, pp. C170–C191, 2012.
[28] F. Manne and T. Sørevik, "Partitioning an array onto a mesh of processors," in *International Workshop on Applied Parallel Computing*. Springer, 1996, pp. 467–477.
[29] Ü. i. t. V. Çatalyürek, C. Aykanat, and B. Uçar, "On two-dimensional sparse matrix partitioning: Models, methods, and a recipe," *SIAM Journal on Scientific Computing*, vol. 32, no. 2, pp. 656–683, 2010.
[30] E. G. Boman, K. D. Devine, and S. Rajamanickam, "Scalable matrix computations on large scale-free graphs using 2d graph partitioning," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2013, pp. 1–12.
[31] K. Li, W. Yang, and K. Li, "Performance analysis and optimization for spmv on gpu using probabilistic modeling," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 1, pp. 196–205, 2014.
[32] W. Yang, K. Li, Z. Mo, and K. Li, "Performance optimization using partitioned spmv on gpus and multicore cpus," *IEEE Transactions on Computers*, vol. 64, no. 9, pp. 2623–2636, 2014.
[33] A. Pınar and C. Aykanat, "Fast optimal load balancing algorithms for 1d partitioning," *Journal of Parallel and Distributed Computing*, vol. 64, no. 8, pp. 974–996, 2004.

0018-9340 (c) 2020 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: Newcastle University. Downloaded on February 17,2021 at 12:44:56 UTC from IEEE Xplore. Restrictions apply.