MATRIX FACTORIZATION WITH STOCHASTIC GRADIENT DESCENT FOR RECOMMENDER SYSTEMS

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF ENGINEERING AND SCIENCE
OF BILKENT UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR
THE DEGREE OF
MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

By
Ömer Faruk Aktulum
February 2019

Matrix Factorization with Stochastic Gradient Descent for Recommender Systems
By Ömer Faruk Aktulum
February 2019

We certify that we have read this thesis and that in our opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Cevdet Aykanat(Advisor)

Hamdi Dibeklioğlu

Tayfun Küçükyılmaz

Approved for the Graduate School of Engineering and Science:

Ezhan Karaşan
Director of the Graduate School

ABSTRACT

MATRIX FACTORIZATION WITH STOCHASTIC GRADIENT DESCENT FOR RECOMMENDER SYSTEMS

Ömer Faruk Aktulum
M.S. in Computer Engineering
Advisor: Cevdet Aykanat
February 2019

Matrix factorization is an efficient technique used for disclosing latent features of real-world data. It finds its application in areas such as text mining, image analysis, social network and more recently and popularly in recommendation systems. Alternating Least Squares (ALS), Stochastic Gradient Descent (SGD) and Coordinate Descent (CD) are among the methods used commonly while factorizing large matrices. SGD-based factorization has proven to be the most successful among these methods after Netflix and KDDCup competitions where the winners' algorithms relied on methods based on SGD. Parallelization of SGD then became a hot topic and studied extensively in the literature in recent years.

We focus on parallel SGD algorithms developed for shared memory and distributed memory systems. Shared memory parallelizations include works such as HogWild, FPSGD and MLGF-MF, and distributed memory parallelizations include works such as DSGD, GASGD and NOMAD. We design a survey that contains exhaustive analysis of these studies, and then particularly focus on DSGD by implementing it through message-passing paradigm and testing its performance in terms of convergence and speedup. In contrast to the existing works, many real-wold datasets are used in the experiments that we produce using published raw data. We show that DSGD is a robust algorithm for large-scale datasets and achieves near-linear speedup with fast convergence rates.

Keywords: Recommender system, Matrix Factorization, Stochastic Gradient Descent, Parallel Computing, Shared Memory Algorithms, Distributed Memory Algorithms.

ÖZET

ÖNERİ SİSTEMLERİ İÇİN OLASILIKSAL EĞİM İNİŞ İLE MATRİS ÇARPANLARINA AYIRMA

Ömer Faruk Aktulum
Bilgisayar Mühendisliği, Yüksek Lisans
Tez Danışmanı: Cevdet Aykanat
Şubat 2019

Matris çarpanlarına ayırma gerçek dünya verilerinin gizli özelliklerini ortaya çıkarmak için kullanılan verimli bir tekniktir. Bu teknik, metin madenciliği, görüntü analizi, sosyal ağlar ve son zamanlarda yaygın olarak öneri sistemleri gibi alanlarda uygulanmaktadır. Birbirini izleyen en küçük karaler (ALS), olasılıksal eğim iniş (SGD) ve koordinat iniş (CD) geniş matrisleri çarpanlarına ayırırken kullanılan yöntemler arasındadır. Bu üç yöntem arasında, SGD'ye dayalı çarpanlarına ayırma yöntemi, Netflix ve KDDCup yarışmalarından sonra en başarılı yöntem olarak ispatlanmıştır. Sonrasında, SGD'nin paralelleştirilmesi yaygınlaşmış ve literatürde geniş bir biçimde çalışılmıştır.

Biz paylaşımlı ve dağıtık bellek sistemleri için geliştirilmiş paralel SGD algoritmalarına odaklanıyoruz. Paylaşımlı bellek paralelleştirmeleri HogWild, FPSGD ve MLGF-MF gibi çalışmalar içerirken dağıtık bellek paralelleştirmeleri DSGD, GASGD ve NOMAD gibi çalışmalar içermektedir. Biz bu çalışmaların detaylı analizini içeren bir araştırma metni oluşturuyoruz, sonrasında ayrıntılı olarak DSGD'ye odaklanıp bu algoritmayı mesaj aktarma yaklaşımı ile uyguluyoruz ve performansını yakınsama ve hızlanma yönünden test ediyoruz. Mevcut çalışmaların aksine deneylerde kendi ürettiğimiz çok sayıda gerçek dünya veri kümeleri kullanıyoruz. DSGD'nin geniş ölçekli veri kümeleri için dirençli bir algoritma olduğunu ve hızlı yakınsama değerleri ile birlikte doğrusala yakın hızlanmayı başardığını gösteriyoruz.

Anahtar sözcükler: Öneri Sistemi, Matris Çarpanlarına Ayırma, Olasılıksal Eğim İniş, Paralel Hesaplama, Paylaşımlı Bellek Algoritmaları, Dağıtık Bellek Algoritmaları.

Acknowledgement

First and foremost, I am grateful to the one person who has stood beside me all my life, with full support under every circumstance; my dear father, Uğur Aktululm. I have always asked myself how a person can have so much love and compassion for another, the answer to which I found when I became a father myself to my baby boy. I feel truly blessed to have been born to such an amazing father. My mother, Halime Aktulum, holds an equally dear place in my heart. I would like to thank her for her tireless effort in my upbringing and for her unending love and support in all aspects of my life. Together, they have overcome many difficulties to present a better life for me and I greatly appreciate them for everything.

I would also like to express thanks to my dear wife, Melike, for coming into my life two years ago. My life has changed for the better after getting married to her. I appreciate her support, understanding and patience during the development of this thesis.

I would like to thank Assoc. Prof. Dr. Ünal Göktaş for his valuable contributions to me not only in the field of computer science but also in many other aspects of my life. He has made always helped me find the right path in life and helped me learn from my mistakes, just like how a father helps his son.

I also appreciate Assoc. Prof. Dr. Fatih Emekci who was my supervisor during my undergraduate studies. He is the best computer engineer I have ever seen, with deep knowledge in both academics and industry. He taught me how to approach and solve problems in the field of computer science in a self-motivated manner and introduced me to professional software development.

I would like to thank Reha Oğuz Selvitopi for his collaboration during my graduate studies. His guidance helped me find my way out of many dead ends. I am also grateful to Mustafa Özdal for his valuable contributions in our joint works.

I am grateful to Asst. Prof. Dr. Hamdi Dibeklioğlu and Asst. Prof. Dr.

Tayfun Küçükyılmaz for reading, commenting and sharing their ideas on the thesis.

I thank my thesis supervisor Prof. Dr. Cevdet Aykanat for giving me the opportunity to take part in the graduate program at Bilkent University.

Finally, I would like to thank my friends, Prasanna Kansakar and Bikash Poudel, from University of Nevada, Reno for supporting me in difficult situations not only when I was in United States but, also after I came back to Turkey.

To my grumpy son, Uğur...

Contents

1 Introduction **1**

2 Background **5**

2.1 Problem Definition 5

2.2 Existing Techniques for Loss Minimization 8

2.2.1 Alternating Least Squares (ALS) 8

2.2.2 Coordinate Descent (CD) 9

2.2.3 Stochastic Gradient Descent (SGD) 11

3 Literature Survey **13**

3.1 Hogwild:A Lock-Free Approach to Parallelizing Stochastic Gradient Descent 13

3.2 DSGD: Large-Scale Matrix Factorization with Distributed Stochastic Gradient Descent 15

3.3 FPSGD: A Fast Parallel SGD for Matrix Factorization in Shared Memory Systems 16

3.4 GASGD: Stochastic Gradient Descent for Distributed Asynchronous Matrix Completion via Graph Partitioning 20

3.5 NOMAD: Non-locking, stOchastic Multi-machine algorithm for Asynchronous and Decentralized matrix completion 24

3.6 MLGF-MF: Fast and Robust Parallel SGD Matrix Factorization . 27

4 DSGD: Large-Scale Matrix Factorization with Distributed Stochastic Gradient Descent 31

4.1 SSGD (Stratified SGD) 36

4.2 DSGD (Distributed SGD) 38

4.3 Implementation 41

5 Experimental Results 44

5.1 Datasets 44

5.1.1 Amazon Dataset 45

5.1.2 Last.fm Dataset 48

5.1.3 Movielens Dataset 49

5.1.4 Netflix Dataset 51

5.1.5 Yahoo Music Dataset 52

5.2 Experimental Setup 53

5.3 Results and Discussion 54

5.3.1 Load Imbalance 54

5.3.2 Speedup 59

5.3.3 Convergence 63

6 Conclusion **67**

List of Figures

3.1 Hogwild algorithm. 14

3.2 The issues in the Hogwild and DSGD algorithms. 17

3.3 The FPSGD model. 18

3.4 The NOMAD model. 25

3.5 The MLGF partitioning strategy (capacity=3). 28

4.1 The SSGD model. 37

4.2 The input data distribution in the DSGD algorithm. 39

4.3 A complete iteration in the DSGD algorithm. 41

5.1 Load imbalance comparison for static and random partitioning 1. 58

5.2 Load imbalance comparison for static and random partitioning 2. 59

5.3 Speedup results 1. 61

5.4 Speedup results 2. 62

5.5 Per-matrix convergence results. 64

5.6 Per-partition convergence results. 66

List of Tables

2.1 Loss functions used in matrix factorization. 7

5.1 Properties of produced datasets. 45

5.2 Amazon datasets. 47

5.3 Movielens datasets. 51

5.4 Yahoo Music datasets. 53

5.5 Parameters of the SGD algorithm used in the experiments. 53

5.6 Load imbalance results for static and random partitioning. 55

Chapter 1

Introduction

Number of online businesses is increasing in these days by making an expansion into different user services. Main goal of the businesses is to make their products more popular by increasing not only the number of services and customers they have but also attraction of the customers to existing services. Recommendation systems recently become a part of this trend by being able to capture latent relationships between the customers (*or users*) and the services (or *items*). Strength of the relationships is expressed through ratings given by the users to the items in a specified range. Many online platforms including social media (e.g., Facebook, Twitter, Instagram), commercial shopping websites (e.g., Amazon) and entertainment products (e.g., Netflix, Yahoo Music) make use of recommendation systems to increase user interest to their systems.

Content filtering and *collaborative filtering* are the most popular techniques among the existing approaches to build recommendation systems. In content filtering, each user or item is characterized by using its features such as age, city, gender for a user; and kind, actors, time for a movie recommender system. Content filtering makes a comparison between the features of the items and the features in the user profile while recommending new items to the users. However, formation of the profiles is an independent problem and incurs additional

cost. On the other hand, instead of forming profiles, collaborative filtering proposes a more effective solution by making use of actions followed by the users in the past. Collaborative filtering exploits the relationship among the users while recommending new items to the users. Among the existing usage areas of the collaborative filtering technique, the most popular one is *latent factor models* that summarizes the rating matrix with factor matrices. In this scenario, both the users and the items are characterized by using a factor matrix for each of them. The latent factor models find their most successful solutions in matrix factorization. [10, 11]

In the real-world systems, there are two different sets including the users and the items that can be moved to a matrix plane, called *rating matrix*, in such a way that the rows and the columns represent the users and the items, respectively. Meanwhile, an entry in a cell of the rating matrix represents a rating given by the user to the respective item. However, the rating matrix is highly sparse since the users rate only a small subset of the items, meaning that there are many missing entries in the rating matrix. The problem is to predict nonexisting ratings in the rating matrix using the latent factors and decide which items can possibly be recommended to the users. The latent factor models can be developed and applied using collaborative filtering approach to model the problem. At this point, discovering the latent factors boils down to matrix factorization solution.

The most important goal in matrix factorization is to minimize loss occurred during prediction of the missing entries. *Stochastic Gradient Descent* (SGD) [7, 2, 3] and *Alternating Least Squares* (ALS) [15, 16] are well-known algorithms used in the matrix factorization solutions to minimize the loss. Even though parallelization of the SGD algorithm is inherently difficult, it has become more popular than the ALS algorithm in recent years. For example, selected top three models in KDDCup 2011 are developed using the SGD algorithm [13]. Then, the SGD algorithm has been parallelized for both shared memory [1, 2, 5] and distributed memory environments [7, 8, 3, 4].

In this thesis, we focus on parallel SGD algorithms designed for both shared memory and distributed memory architectures. We walk through by starting from

the earliest proposed algorithms and investigate improvements work by work by exhaustively examining the developed models with their pros and cons. At the end, we create a literature survey that includes detailed analysis of popular parallel SGD algorithms. DSGD [7] is the leading work among the existing popular works [1, 7, 2, 3, 4, 5]. Although most of the algorithms developed in this area are influenced by this outstanding study, there is no such an extensive study that analyzes the DSGD model as far as we researched. Hence, we specifically focus on the DSGD algorithm among the existing works by implementing it using message-passing paradigm and testing its performance in detail. Our contributions are grouped under extensive analysis of the DSGD algorithm as follows.

- In contrast to the experiments performed in the existing works that include only a few real-world datasets, we produce many real-world datasets using published raw data as their features are given in Section 5.1. Then, we run our DSGD implementation with these datasets using different number of processors.
- In the DSGD study, there information regarding effect of applied random permutation on input data by the DSGD algorithm initially. Hence, we generate static and random partitioning files for each real-world dataset and different number of processors and calculate the load imbalance for both partitioning schemes by implementing a load imbalance calculator. Finally, we discuss how the random permutation increases the load balance among the processors and speed up the convergence in Section 5.3.1 and Section 5.3.3, respectively.
- In addition, we conduct the experiments regarding our DSGD implementation with different number of processors in terms of the load imbalance, the speedup and the convergence metrics and discuss them in detail in Chapter 5. There is no such a work regarding performance measurement of the DSGD algorithm in such an extensive manner.

The remainder of this thesis is organized as follows. Chapter 2 includes crucial background statements regarding this work by defining the problem and the

matrix factorization techniques in detail. The popular parallel SGD algorithms are examined in Chapter 3, and the selected model is described with its implementation details in Chapter 4. Then, we state properties of the datasets and discuss the experimental results in Chapter 5. Finally, we conclude our work in Chapter 6.

Chapter 2

Background

In this chapter, we first define the problem using a real-world scenario in Section 2.1. Then, we mention the matrix factorization solution and reinforce the comprehensibility of this concept by stating related mathematical background. Finally, we introduce the loss optimization techniques applied in matrix factorization and mention the parallelization strategies by stating their pros and cons in Section 2.2.

2.1 Problem Definition

We start to explain the problem by making use of a real-world example [7]. Assume that we sell course books online in a commercial website, where we allow the users to rate the books. In this example, we have four users and three books as illustrated with a rating matrix below, where the entries represent the existing ratings (e.g, $user_2$ rates $book_2$ as 5). The entries shown with (-) are unknown, and we call them *missing entries* that lead to cold start problem. We want to increase attraction of the users to the books by making use of the existing ratings.

$$\begin{array}{c} \text{Ratings} \\ user_1 \\ user_2 \\ user_3 \\ user_4 \end{array} \begin{array}{c} \begin{array}{ccc} book_1 & book_2 & book_3 \end{array} \\ \begin{pmatrix} - & 1 & - \\ - & 5 & - \\ 4 & - & - \\ - & - & 2 \end{pmatrix} \end{array}$$

To decide which book can be recommended to which user, we want to have information about the missing entries as certain as possible. The problem focused on this concept is predicting the missing entries accurately to build high quality recommendation systems. In the remainder of this section, we introduce the mathematical background of the concept.

Given a rating matrix $\boldsymbol{R}$ with size $\boldsymbol{m} \times \boldsymbol{n}$ where m and n denote the number of the users and the items, respectively, and each nonzero entry $\boldsymbol{r_{i,j}}$ in the rating matrix R denotes the rating given by user i to item j. In the real-world systems, the number of the users is much more than the number of the items. Let $\boldsymbol{W^{m \times k}}$ and $\boldsymbol{H^{k \times n}}$ be user and item factor matrices, respectively, where $\boldsymbol{k}$ is the factor size, and W_x and ${H_y}^T$ denote the xth row vector of factor matrix W and yth row vector of factor matrix H, respectively, both size k. The rating matrix R is factorized by finding out the proper factor matrices W and H using a loss minimization technique to achieve $R \approx W \cdot H$. This process is known as matrix factorization (or low-rank approximation), and existing matrix factorization techniques are discussed in [11]. After the low-rank approximation process is completed, the missing ratings in the rating matrix R can be predicted using vectors of obtained factor matrices. For example, the rating $r_{2,3}$, given by $user_2$ to $book_3$, can be predicted by computing the dot product of the second row vector of the factor matrix W and the third row vector of the factor matrix H as $W_2 \cdot {H_3}^T$. Accuracy of the approximation is expressed with a loss function L that takes vectors of the factor matrices W and H as inputs and generates a loss based on the difference between the predicted value and the real value as an output. Hence, we need to minimize the loss to find out better predictions. [5, 7, 4]

Different loss functions are used in matrix factorization as the most popular

Loss Function	Definition
L_{S1}	$\sum_{r_{i,j} \in R \ and \ r_{i,j} \neq 0} (r_{i,j} - W_i \cdot {H_j}^T)^2$
L_{L2}	$L_{S1} + \sum_{t \in (0,1,\ldots,k)} (\lambda_W . \left\|W_i\right\|_2 + \lambda_H . \left\|H_j\right\|_2)$
L_{L2w}	$L_{S1} + \sum_{t \in (0,1,\ldots,k)} w_t (\lambda_W . \left\|W_i\right\|_2 + \lambda_H . \left\|H_j\right\|_2)$

Table 2.1: Loss functions used in matrix factorization.

three of them are given in Table 2.1 [8]. $\boldsymbol{L_{S1}}$ is the simplest loss function based on squared loss, which is called root mean squared error (RMSE). On the other hand, $\boldsymbol{L_{L2}}$ includes *L2* regularization to avoid overfitting. The regularization part is added to $\boldsymbol{L_{S1}}$ as shown in the table where λ is the regularization parameter. The last one, $\boldsymbol{L_{L2w}}$, is the weighted form of L2 regularization mostly used in parallel applications, where global loss is calculated as weighted sum of local losses. The weight of each local loss is expressed with the number of total local entries by setting the total weight on the rating matrix to 1. In this area, the L2 regularized loss functions are widely used among these existing loss functions in such a way that some of them use L_{L2}, whereas others use L_{L2w} according to proposed models. The general formula for the loss functions including L2 regularization is given as follows,

$$L(W,H) = \sum_{r_{i,j} \in R \ and \ r_{i,j} \neq 0} \left\| r_{i,j} - W_i \cdot {H_j}^T \right\|_2^2 + \lambda_W . \left\|W_i\right\|_2^2 + \lambda_H . \left\|H_j\right\|_2^2 \quad (2.1)$$

where $\boldsymbol{i}$ and $\boldsymbol{j}$ denote the row and the column indices of the ratings in the rating matrix R, respectively, and $\boldsymbol{\lambda_W}$ and $\boldsymbol{\lambda_H}$ are the regularization parameters (≥ 0) used in optimization to avoid overfitting that occurs due to non-convex nature of the problem based on the term $W_i \cdot {H_j}^T$. $\left\|.\right\|_2$ is the L2 norm, and $\left\|W_i\right\|_2^2$ and $\left\|H_j\right\|_2^2$ are equal to $W_i \cdot {W_i}^T$ and $H_j \cdot {H_j}^T$, respectively. Similarly, $\left\| r_{i,j} - W_i \cdot {H_j}^T \right\|_2^2$ is reduced to $\left(r_{i,j} - W_i \cdot {H_j}^T\right)^2$, and Equation 2.1 is reorganized as follows,

$$L(W,H) = \sum_{r_{i,j} \in R \ and \ r_{i,j} \neq 0} \left(r_{i,j} - W_i \cdot {H_j}^T\right)^2 + \lambda_W . W_i \cdot W_i^T + \lambda_H . H_j \cdot H_j^T \quad (2.2)$$

The existing optimization techniques to minimize the loss function in Equation 2.2 are covered in Section 2.2

2.2 Existing Techniques for Loss Minimization

In this section, the popular loss optimization techniques are described and compared in terms of applied update rule, convergence and parallelization strategies.

2.2.1 Alternating Least Squares (ALS)

The loss minimization is a non-convex problem, however, it is converted to a quadratic problem by fixing one factor side while working on the other factor side [18, 11]. Alternating Least Squares (ALS) uses this technique to minimize the loss function by solving the *least squares problems* during update procedures which are applied on related vectors of the factor matrices. The overall procedure of the ALS algorithm is given in Algorithm 1. Firstly, vectors of the factor matrix W, $(W_1, W_2, \ldots, W_m)$, are updated by fixing the matrices R and H. Then, vectors of the factor matrix H, $(H_1, H_2, \ldots, H_n)$, are updated by fixing the matrices R and W. During the update procedures, the least squares problems are solved with the following update procedures [17],

$$W_i = \sum_{r_{i,j} \in R_{i,*} \ and \ r_{i,j} \neq 0} (r_{i,j} H_j)/(H_j \cdot H_j^T + \lambda I) \tag{2.3a}$$

$$H_j = \sum_{r_{i,j} \in R_{*,j} \ and \ r_{i,j} \neq 0} (r_{i,j} W_i)/(W_i \cdot W_i^T + \lambda I) \tag{2.3b}$$

where $\boldsymbol{\lambda}$ is the regularization parameter (≥ 0), $R_{i,*}$ and $R_{*,j}$ denote the ratings in the ith row and jth column of the rating matrix R, respectively, and I the identity matrix. The crucial part of the update rules is applying the update procedures once for all the ratings in the same row or column, respectively. This makes the convergence of the ALS-based algorithm faster since the number of the ratings updated per iteration is increased. Moreover, the convergence of the ALS-based algorithms are generally completed in the first twenty iterations [15], and ALS is faster than SGD in terms of convergence.

In contrast to Stochastic Gradient Descent (SGD), ALS does not use the calculated vector values in the next iteration. Therefore, the update procedures

applied in the ALS algorithm are independent of each other. The independent updates make parallelization of the ALS algorithm easier, and allow workers[1] to work on the rating matrix R at the same time in such a way that the ratings in the rating matrix R can be simultaneously processed by the workers in row wise or column wise. This feature makes the ALS algorithm to be preferable to the SGD algorithm. In addition, the ALS algorithm applies the update procedures for all the ratings instead of a rating in the same row or column once. Hence, this increases the efficiency of ALS in terms of computation and makes the ALS algorithm to be preferable to the Coordinate Descent algorithm.

Algorithm 1 ALS Algorithm for Matrix Factorization

Input Rating matrix ($\boldsymbol{R}$), user and item factor matrices ($\boldsymbol{W}$ and $\boldsymbol{H}$) and $\boldsymbol{\lambda}$

1: **while** not converged **do**
2: **for** each vector i in W **do**
3: Update W_i by applying (2.3a)
4: **end for**
5: **for** each vector j in H **do**
6: Update H_j by applying (2.3b)
7: **end for**
8: **end while**

2.2.2 Coordinate Descent (CD)

Coordinate Descent (CD) is the another loss minimization technique used in various areas of large-scale optimization problems including big data [19], tensor factorization [20, 21, 22], support vector machines [23, 24] and matrix factorization [18, 25, 26]. The idea of moving the problem to quadratic environment is similar to the ALS algorithm. However, the CD algorithm applies a different update rule by updating only an entry in vectors of the factor matrices instead of an entire vector at a time by fixing the others. The update procedures of the

[1]Threads in shared memory systems or processors in distributed memory systems.

CD algorithm are given as follows.

$$w_{i,s} = \frac{\sum_{r_{i,j} \in R_{i,*} \text{ and } r_{i,j} \neq 0} (r_{i,j} + w_{i,s} h_{j,s}) h_{j,s}}{\lambda \sum_{r_{i,j} \in R_{i,*} \text{ and } r_{i,j} \neq 0} {h_{j,s}}^2} \tag{2.4a}$$

$$h_{j,s} = \frac{\sum_{r_{i,j} \in R_{*,j} \text{ and } r_{i,j} \neq 0} (r_{i,j} + w_{i,s} h_{j,s}) w_{i,s}}{\lambda \sum_{r_{i,j} \in R_{*,j} \text{ and } r_{i,j} \neq 0} {w_{i,s}}^2} \tag{2.4b}$$

There are two common variants of the CD algorithm used in the matrix factorization models developed for recommendation systems [18]. These variants propose different update schemes as feature wise and user (or item) wise. The algorithms apply user-wise updates are named Cyclic Coordinate Descent (CCD) algorithms, in which the applied update sequence has a cyclic order such as $w_{1,1->1,k}$, $w_{2,1->2,k}$, ..., $w_{m,1->m,k}$, $h_{1,1->1,k}$, $h_{2,1->2,k}$, ..., $w_{n,1->n,k}$, $w_{1,1->1,k}$ ..., and so forth. The overall procedure of the CCD algorithm for matrix factorization is stated in Algorithm 2. On the other hand, the CCD++ algorithm applies feature-wise update scheme such as $w_{1,1->1,k}$, $h_{1,1->1,k}$, $w_{2,1->2,k}$, $h_{2,1->2,k}$, ..., $w_{m,1->m,k}$, $h_{n,1->n,k}$, and so forth. The feature-wise update sequence improves the convergence due to applying the update procedures between the user and the item vectors more frequently. Similar techniques used in parallelization of the ALS algorithm can be directly applied to parallelize the CD algorithm. [18] and [25] are the most recent parallel CD algorithm developed for matrix factorization.

Algorithm 2 CCD Algorithm for Matrix Factorization

Input Rating matrix ($\boldsymbol{R}$) and user and item factor matrices ($\boldsymbol{W}$ and $\boldsymbol{H}$)

while not converged **do**
  for $i = 1, 2, \ldots m$ **do**
    for $s = 1, 2, \ldots k$ **do**
      Update $w_{i,s}$ by applying (2.4a)
    end for
  end for
  for $j = 1, 2, \ldots n$ **do**
    for $s = 1, 2, \ldots k$ **do**
      Update $h_{j,s}$ by applying (2.4b)
    end for
  end for
end while

2.2.3 Stochastic Gradient Descent (SGD)

SGD is an iterative algorithm mostly used in matrix factorization [1, 2, 5, 7, 8, 4, 3] and machine learning [27, 28, 29]. SGD applies gradient on Equation 2.2 regarding vectors of the factor matrices to optimize the loss function. The derivatives of the loss function based on W and H are calculated, and the update rules are obtained as following,

$$W_i = W_i - \alpha(e_{i,j}H_j - \lambda_W H_j) \tag{2.5a}$$

$$H_j = H_j - \alpha(e_{i,j}W_i - \lambda_H W_i) \tag{2.5b}$$

where $\boldsymbol{e_{i,j}}$ is the loss (or *error*) calculated for the rating $r_{i,j}$ in the current iteration. $\boldsymbol{\alpha}$ is the learning rate that can be selected different for each factor matrix, and $\boldsymbol{\lambda}$ is the regularization parameter used to avoid overfitting. The most important handicap in parallelization of the SGD algorithm is dependency of the update procedures which are inter-dependent as inferred from Equation 2.5a and Equation 2.5b. In other words, vectors of the factor matrices are updated using their current values calculated in the previous iteration. Hence, this makes parallelization of the SGD algorithm more difficult. The error for a rating in matrix R changes in each iteration due to applied updates in the related vectors of factor matrices. The error is calculated using the updated vectors as follows.

$$e_{i,j} = r_{i,j} - W_i \cdot {H_j}^T \tag{2.6}$$

where $r_{i,j}$ denotes real value of the rating, and the prediction value is obtained by calculating the dot product of the related user and item vectors. Hence, the error calculates the convergence for each rating by finding the difference between its real and predicted values in each iteration.

The update procedures in the ALS algorithm are not depending on each other as illustrated in Equation 2.3, where the current user or item vector values are not considered while updating them. On the other hand, the update procedures of vectors in SGD algorithm are inherently sequential as described above. Therefore,

Algorithm 3 SGD Algorithm for Matrix Factorization

Input Rating matrix ($\boldsymbol{R}$), user and item factor matrices ($\boldsymbol{W}$ and $\boldsymbol{H}$) and $\boldsymbol{\lambda}$

1: **while** not convergenced **do**
2: **for** each rating, $r_{i,j}$, in R **do**
3: Update W_i by applying (2.5a)
4: Update H_j by applying (2.5b)
5: **end for**
6: **end while**

developing the SGD-based parallel applications is more difficult than the ALS-based algorithms. However, the SGD-based parallelization has been widely used after the Netflix [12] and the KDD Cup [13] competitions, where the selected top models are developed using the SGD algorithm for large-scale applications.

Chapter 3

Literature Survey

This chapter contains detailed survey of popular parallel SGD algorithms developed for shared memory and distributed memory systems. We review the studies in chronological order to show improvements with proposed contributions.

3.1 Hogwild:A Lock-Free Approach to Parallelizing Stochastic Gradient Descent

Parallelization of the SGD algorithm has two important bottlenecks, which are locking and synchronization, that completely affect performance of the developed parallel algorithms. In shared memory, the locking issue arises when more than one thread wait (or idle) for accessing and changing a variable concurrently. The synchronization process is applied to use updated vectors of factor matrices during update procedures to speed up convergence, and the synchronization problem occurs when the processors or threads are not synchronized regularly. These two main problems stem from the sequential update procedure of the latent factors in the SGD algorithm as explained in Section 2.2.3. In parallel SGD algorithms developed for shared memory architecture, the locking issue occurs while applying the update procedures not only for the same rating but also for the ratings in the

same row or column of the rating matrix. In other words, simultaneous access to the ratings in the same row or column during the update procedures leads to the locking issue also in the related user or item vectors, and this case results in memory overwrites.

Niu et al. [1] develop a parallel SGD algorithm, called Hogwild, for shared memory systems by proposing a new update procedure plan to avoid the locking issue. Any locking mechanism is not used by Hogwild in such a way that each thread accesses to ratings randomly without concern about the memory overwrites regarding the update procedures. Although this can be thought as a serious problem at first glance in terms of using the most recent updated vectors, the authors prove that it does not cause computational error due to sparsity of data access. In other words, only small part of the factor matrices are updated, and the memory overwrites rarely happen. In addition, they theoretically show that convergence to ideal rates is almost achieved by the Hogwild algorithm. Although most of the existing works regarding parallelization of SGD in distributed systems prove global convergence without any rate, the authors prove the convergence of the Hogwild algorithm with rates according to selected step size. The overall procedure of the Hogwild algorithm is shown in Algorithm 4.

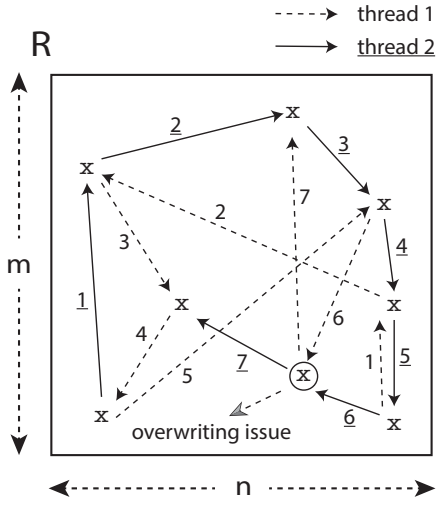


Figure 3.1: Hogwild algorithm.

An example of the proposed update sequence by Hogwild algorithm is illustrated in Figure 3.1. In this scenario, there is a rating matrix R where m and

n denote the number of the users and the items, respectively, and x represents the ratings. There are two threads as their mapping given on the top right side of the figure, whose update sequences are represented with arrows, and the numbers on the arrows show update order of the ratings. During working progress of the Hogwild algorithm for this example, there is only one memory overwrite that occurs in the 6th update, where both of the threads want to apply update procedure on the same rating at the same time as illustrated with a circle in the figure. The probability of this event happening is negligible, and it does not affect the convergence rate as the authors stated in the Hogwild study.

Algorithm 4 The overall procedure of Hogwild

Input Rating matrix (R), number of threads (T)
1: **for** each thread **do** // parallel task
2: **while** not converged **do**
3: Select a rating, $r_{i,j}$, from matrix R randomly
4: Apply update procedures in Equation 2.5 for $r_{i,j}$
5: **end while**
6: **end for**

Experimental results show that almost linear speedup for similar applications based on the sparsity is achieved by Hogwild algorithm, and its lock-free approach is faster than existing memory locking methods such as [9].

3.2 DSGD: Large-Scale Matrix Factorization with Distributed Stochastic Gradient Descent

Gemulla et al. [7] develop an efficient parallel SGD algorithm, named Distributed SGD (DSGD), for distributed memory systems. DSGD makes use of stratified SGD (SSGD) model that divides the rating matrix into blocks. The authors specialize the SSGD model by exploiting interchangeable blocks, which are defined as any two or more blocks that do not share any rows or columns of the rating matrix. Similarly, the user and item factor matrices are also partitioned into blocks and distributed among the processors. Then, the processors work on the

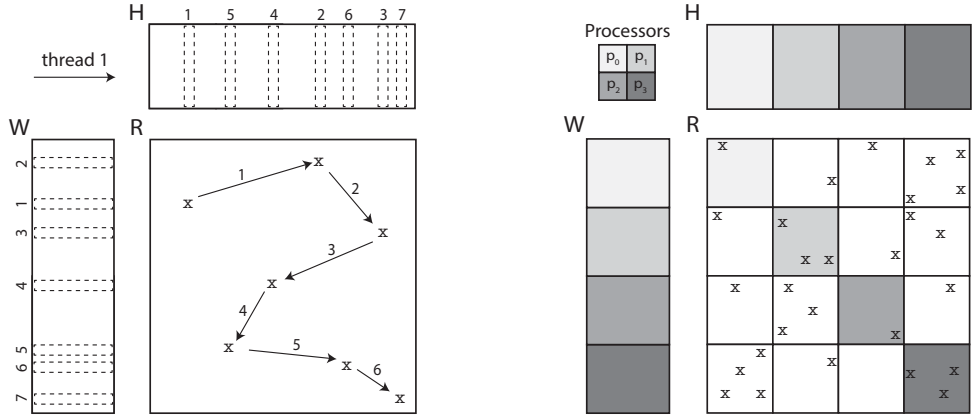
interchangeable blocks simultaneously and apply bulk synchronization process periodically to avoid steal data usage. During the bulk synchronization process, the processors communicate the updated item factor blocks among each other at the same time. Therefore, the processors apply the update procedures by using the most recent updated vectors of the factor matrices. In addition, the update sequence applied by the DSGD algorithm is the same with the serial SGD algorithm that makes convergence of the DSGD algorithm faster. Further details of the DSGD algorithm are described and our contributions in terms of analysis and experiments are given in Chapter 4.

3.3 FPSGD: A Fast Parallel SGD for Matrix Factorization in Shared Memory Systems

Zhuang et al. [2] develop Fast Parallel SGD (FPSGD) algorithm for shared memory systems by addressing two crucial problems in previously proposed shared memory SGD algorithms [1, 7], which are load imbalance and nonuniform memory access. Main contributions of this study are optimizing the load balance among the threads and decreasing high cache-miss rate during memory access. Note that shared memory implementation of DSGD is considered when we call the DSGD algorithm through this section.

The authors point out existing nonuniform memory access issue in Hogwild and DSGD due to random access to ratings during update procedures. Applied randomization technique by these algorithms leads to increase in the cache-miss rate. Moreover, the random access to ratings results in nonuniform access to vectors of factor matrices as inferred from Equation 2.5. This issue is illustrated for the Hogwild algorithm in Figure 3.2a, where nonuniform memory access occurs through the update sequences not only in the rating matrix but also in both factor matrices. The arrows on the rating matrix denote the update sequences of a thread, whereas the numbers on the factor matrices show sequence of respective

accessed vectors. The memory access in the rating matrix and both factor matrices are nonuniform that results in cache misses. DSGD also suffers from the same issue due to selection of the ratings randomly during the update procedures.



(a) The cache-miss issue occurred in the Hogwild algorithm.

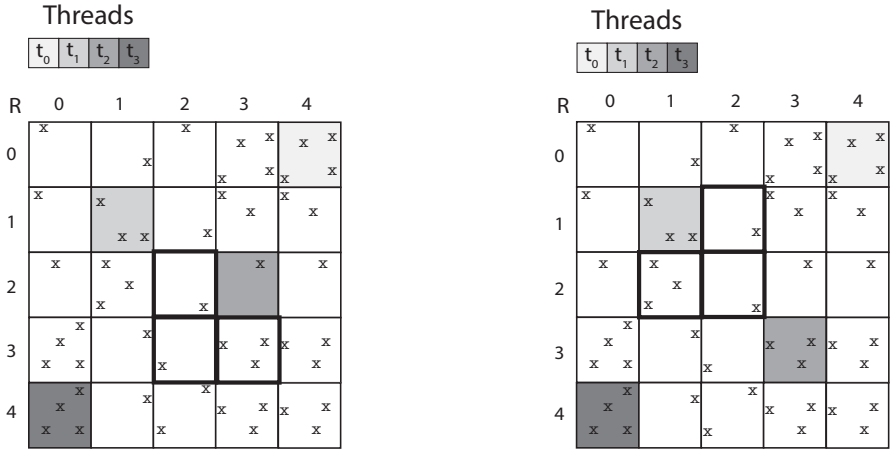
(b) The locking issue in shared memory design of the DSGD algorithm.

Figure 3.2: The issues in the Hogwild and DSGD algorithms.

The block partitioning technique of the DSGD algorithm is used in the FPSGD algorithm with a proposed new update scheme for alleviating the high cache-miss rates. The authors firstly consider an ordered method, in which the ratings in the blocks are selected sequentially, and then both user and item factor matrices do not suffer from the cache-miss issue. Although the ordered method provides uniform access to the factor matrices, convergence rate shows an alteration depending on the learning rate. Thus, there is a trade-off between uniform memory access and convergence in such a way that selecting the ratings randomly during the update procedures improves convergence, but increases the cache-miss rate, vice versa. Finally, they propose *partial random method*, where the blocks in the rating matrix are selected randomly, and the ratings in the blocks are picked sequentially. The partial random method achieves uniform memory access on the rating matrix and both factor matrices and converges faster the than random method even if the learning rate differs when root mean squared loss is applied.

The second contribution of this work is regarding the locking issue to keep

each thread working continuously as well as increasing the efficiency in terms of convergence. The authors address the locking issue in DSGD based on the difference among the number of the ratings in the blocks, meaning that if the ratings are not uniformly distributed among the blocks, some threads having fewer ratings in their blocks wait for the others. Figure 3.2b illustrates the locking issue with an example of the DSGD model in shared memory system with four threads, where the threads start to work on the diagonal blocks of the rating matrix in the first subiteration such that t_0, t_1, t_2 and t_3 have 1, 3, 1 and 3 ratings in their blocks, respectively. Although t_0 and t_3 complete their update procedures probably three times faster than other threads, they have to wait for the others to use the most recent updated factor vectors in the next iteration. If they do not wait and continue to work with the ratings in the next block, two threads start to access the same item vectors concurrently, and the locking issue occurs. The DSGD model is mainly developed for distributed systems, and optimizing the communication cost is more important than minimizing idling time among the processors. Therefore, it is not a critical issue for the DSGD algorithm in distributed memory systems.



(a) Subiteration 1 (b) Subiteration 2

Figure 3.3: The FPSGD model.

To avoid the locking issue, the authors firstly consider applying random permutation on the rating matrix. However, they claim that the random permutation

may not the existing solve the problem since computation time might be different even if the number of the ratings in the blocks are the same. Hence, they propose a lock-free model by gridding the rating matrix into $(t + 1)$ x $(t + 1)$ blocks instead of t x t where t is the number of threads. The developed model is illustrated in Figure 3.3 with an example, where the rating matrix is partitioned into 5x5 blocks where four threads work. Therefore, there is always at least one interchangeable block that can be assigned to a thread that completes its task. In the first subiteration as shown in Figure 3.3a, t_2 (thread 2) probably completes its update procedure earlier than others since it has only one rating in its block. Then, t_2 selects a new block among the free interchangeable blocks shown with bold frames as $R_{2,2}$, $R_{3,2}$ and $R_{3,3}$ and starts to work on the ratings in the new block without waiting for the others. Assume that t_2 selects $R_{3,3}$ and works on the ratings of that block as shown in Figure 3.3b. Then, t_1 finishes its job this time and works with one of the free interchangeable blocks shown with bold frames again without waiting for the others. Assignment of the new blocks to free threads are managed by a scheduler dynamically, called *lock-free scheduler*, that searches free blocks regarding their update counts and then sets one having at least update count to the respective thread. The advantage of dynamic scheduling is to keep update counts applied for each block similar through working progress. However, the update counts of the blocks differs from each other too much, when the data is intensely imbalanced. Hence, they calculate the imbalance by defining a degree of imbalance (DoI) to determine the size of difference in the update counts of the blocks. By this way, they obtain efficiency of the lock-free approach in such a way that smaller DoI means the update counts of the blocks are similar, whereas larger values of DoI means update counts of the blocks are too different. Experimental results show that DoI converges to zero after 50th iteration by using the lock-free scheduler, and idling problem disappears after that point. The overall procedure of the FPSGD algorithm in Algorithm 5.

By using single precision floating point instead of double precision and applying vectorization for inner products and additions, a speedup of 2.4 is achieved over normal FPSGD implementation. The authors implement another version of FPSGD, called FPSGD**, where the same block concept in DSGD model is

Algorithm 5 The overall procedure of FPSGD

Input Rating matrix (R), number of threads (t), number of updates (u)

1: **while** u is not reached **do**
2: Apply random permutation on R
3: Divide R into at least $(t+1)$ x $(t+1)$ blocks
4: Start lock-free scheduler and threads with initial parameters
5: Run SGD on the blocks with t threads
6: **end while**

applied by dividing the rating matrix into t x t blocks without using the lock-free scheduler to compare the performance of FPSGD with DSGD obviously. FPSGD with lock-free scheduler converges faster than FPSGD**, meaning that FPSGD does not suffer from the locking issue. In addition, FPSGD converges faster than Hogwild and DSGD in shared memory systems.

3.4 GASGD: Stochastic Gradient Descent for Distributed Asynchronous Matrix Completion via Graph Partitioning

Petroni et al. [3] analyze the performance of previously proposed shared memory SGD algorithms [1, 2] and claim that their performance becomes worse when problem size increases due to growth in frequency of accessed shared data among the threads. To avoid this bottleneck, the studies regarding parallelization of the SGD algorithm currently find its applications in distributed memory systems by solving the performance issue with large clusters. Although successful parallel SGD algorithms are proposed in distributed memory environment such as DSGD, they also have performance issues based on bulk synchronization process. The authors develop a new asynchronous SGD (ASGD) model for distributed memory systems, named GASGD, that proposes three contributions within the context of load balance and resynchronization frequency among the processors.

In distributed memory systems, the ASGD algorithms have important differences than synchronous SGD algorithms in terms of stored data and applied synchronization approach. In distributed ASGD algorithms, a unique master copy and working copies are created for each vector of factor matrices, and the processor owns the master copy of a vector is called master. Each processor works on its local copy by applying update procedures and communicates updated local vectors of factor matrices with related master processors periodically. This whole process is called asynchronous since workers simultaneously work on the same vectors of factor matrices. In contrast to distributed ASGD algorithms, the processors can not work on the same vectors of factor matrices at the same time in distributed synchronous SGD algorithms such as DSGD, where the processors work on the blocks on the rating matrix with using only two factor matrices, meaning that there is no local copy of the factor matrices, and communicate the updated factor blocks with each other during bulk synchronization process. Then, each processor starts to work on the ratings in its next block using received updated factor block, so this process is called synchronous. The time spent for communication of the updated vectors leads to idling issue which makes this type of algorithms inefficient. However, the most recent updated vectors of factor matrices are always used during update procedures.

The developed asynchronous models based on the SGD algorithm such as [4, 3] differ in terms of applied rules regarding the synchronization process. GASGD divides an iteration into f equal parts, named synchronization frequency, and applies bulk synchronization process at the end of each part after completing three stages. Firstly, the workers apply the update procedures on the local vector copies of the factor matrices in computation stage, and then the updated local vectors are communicated with related master processors during synchronization process. Finally, master processors calculate the new master copies as weighted sum of updated local vectors and resend them back to the workers. This process is repeated f times through an iteration, and the synchronization process is repeated until there is no more improvement in convergence. The overall procedure of the GASGD algorithm is given in Algorithm 6.

Resynchronization frequency plays an important role in terms of efficiency by

balancing communication cost and convergence rate since it is used not only to improve the convergence rate but also to optimize the communication cost. Hence, f is like a regularization parameter to adjust the trade-off between the convergence rate and the communication cost. Therefore, finding out the best value of f is a critical task for performance of the algorithm in terms of both convergence and communication cost. For example, synchronization of the working copies can be repeated continuously during an epoch to guarantee convergence, however, this idea may result in the communication bottleneck due to increased number of messages. On the other hand, the synchronization is applied after every iteration to decrease the communication cost, however, this time it converges slowly since the convergence is faster when the processors work on the most recent updated vectors of the factor matrices. None of the previously proposed distributed SGD algorithms for matrix factorization such as [7] considered to set and change the resynchronization frequency. In contrast, GASGD introduces a tuning mechanism for the resynchronization frequency by keeping overall stable quality of the update procedures in the SGD algorithm.

Algorithm 6 The overall procedure of GASGD

Input Synchronization frequency (f)
1: **for** each epoch until convergence **do** // parallel task
2: Permutes own data and divides the epoch into equal f subepochs
3: **for** each subepoch **do**
4: Local working copy of factor vectors are updated (*Computation stage*)
5: Updated working copies are sent to masters (*Communication stage*)
6: Masters calculate & resend updated master copy (*Synch. stage*)
7: **end for**
8: **end for**

The another contribution of this study is relied on the input data distribution, named *bipartite aware greedy algorithm*, by looking at greedy vertex-cut streaming algorithm, where the rating matrix is represented with a graph in such a way that vertices and edges represent users (or items) and ratings, respectively. In this scenario, main tasks are based on the vertices due to importance of communicating item or user vectors of the factor matrices. Hence, the vertex-cut approach is considered instead of the edge-cut approach to minimize the communication cost by assigning each vertex to only one partition. After the vertex-cut partitioning

is applied, each edge in the graph is being responsible by only one node, whereas the vertices are being responsible by more than one node. The number of different nodes of a vertex is located means replication factor (RF) that can be associated with communication volume that affects the running time of the developed algorithm. Hence, the authors use the greedy vertex-cutting algorithm to minimize the replication factor and provide the load balance by making edge counts of the processors similar. They make use of the bipartite feature of the graph to minimize the number of used same vector. The user and item partitioned approaches, named greedy-user partitioned (GUP) and greedy-item partitioned (GIP), are applied by keeping the user and item vectors in a single node, respectively, whereas other one is replicated. The GUP method works better in terms of replication factor, since the number of the users is much more than the number of items in the real-world systems. The authors optimize the replication factor by exploiting main characteristics of the input data. Hence, obtained deep knowledge of the rating matrix is used to decrease the communication cost with partitioning the data by considering the communicated vectors of the factor matrices.

The quality of developed partitioning techniques is compared with existing partitioning models in terms of the replication factor and the load balance among the processors. The GIP, GUP, grid (in DSGD), greedy and hashing partitioning methods are implemented and evaluated. Experimental results show that the GUP and GIP methods produce better results in terms of the replication factor and load balance metrics with datasets such as Movielens and Netflix, in which there is much more difference between the number of the users and the items. On the other hand, the greedy solution produces better results than GIP with Yahoo dataset since there is no much difference between the number of the users and the items. Although the grid partitioning technique works worse than greedy partitioning in terms of the load imbalance, it gives better results than the greedy partitioning with respect to the replication factor due to having complete knowledge of the rating matrix during stratification. In contrast, GUP is not affected by the growth in the number of the processors in terms of convergence rate, and achieves faster convergence. In addition, the communication cost is directly proportional to the resynchronization frequency in such a way that

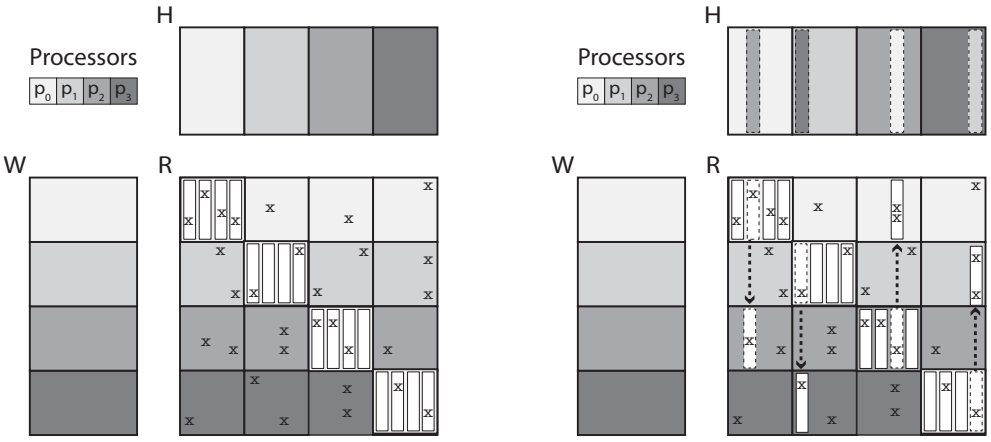
the cost decreases when a method has smaller replication factor in low frequencies, whereas it increases with larger replication factor. In Movielens and Netflix, smaller resynchronization frequencies are enough for faster convergence, whereas larger resynchronization frequency is necessary for Yahoo dataset.

3.5 NOMAD: Non-locking, stOchastic Multi-machine algorithm for Asynchronous and Decentralized matrix completion

Yun et al. [4] propose an efficient matrix factorization algorithm, named NOMAD, for distributed memory systems. NOMAD is developed as an asynchronous algorithm like GASGD, however, it is a fully asynchronous algorithm, where each processor simultaneously applies update procedures on its local data in a lock-free manner without using bulk synchronization process. NOMAD has a decentralized feature that provides load balance among the processors in terms of both computation and communication. In addition to all these capabilities, applied update sequence in the NOMAD algorithm is the same with the sequence of the serial SGD algorithm. Hence, workers always apply update procedures with using the most recent updated vectors of the factor matrices, even though most of the asynchronous algorithms such as [3] start to use steal data when the number of the workers is increased.

The synchronous SGD algorithms in distributed systems such as DSGD apply bulk synchronization process at the end of each subiteration, so they are not able to keep both CPU and network busy at the same time, since the computation and communication processes are sequential and applied by all the processors together. Hence, they suffer from the idling issue, where the slowest worker is waited by the others as explained in the previous section in detail. In contrast to this type of algorithms, NOMAD does not use bulk synchronization process such that the workers apply update procedures in their local data by communicating the updated vectors with respective masters anytime. In addition, to avoid the

idling issue the authors propose a fine-grained partitioning to process smaller number of ratings and communicate respective vectors in smaller time periods. After each communication, ownership of the communicated vector changes. In NOMAD, initial distribution of the input data including the rating matrix and the factor matrices is the same with DSGD. It means that the rating matrix and factor matrices are partitioned into p x p and p blocks, respectively, each worker owns the same data initially as in DSGD, and the first task of the workers is processing the ratings in diagonal blocks as illustrated in Figure 3.4a. Similarly, only updated item vectors are communicated even though both user and item vectors are updated. However, NOMAD differs from DSGD by partitioning each block again into n smaller pieces by forcing workers to work on the ratings in a piece of block. Thus, probability of finding a free piece in the blocks for a free worker is increased after the fine-grained partitioning. An example of working progress of the NOMAD algorithm is shown in Figure 3.4, where each piece of blocks includes only an item vector. Each processor works on its piece of block and communicates the updated vector of the factor matrix H with randomly selected processor. In this example, p_0 sends the updated item vector to p_2 as illustrated with arrows, and then p_2 is the new owner of this item vector and applies the update procedures on the respective ratings. Therefore, slower workers have less loads, so there is no locking issue.



(a) Subiteration 1 (b) Subiteration 2

Figure 3.4: The NOMAD model.

In NOMAD, the workers keep and maintain their tasks in a queue as a tuple including index of item vector and its data (item vector). The overall procedure of the NOMAD algorithm is shown in Algorithm 7. Each worker selects and pops an element from its queue and then applies update procedures in its local set of respective item vector. After the current update procedure is completed, a new worker for the updated vector is selected randomly, and the element is pushed into the new worker's queue.

Algorithm 7 The overall procedure of NOMAD

Input Rating matrix (R), user factor matrix (W), item factor matrix (H), array of queue (Q), number of processors (p) and rank of proc. (my_rank)

Initialize array of Q with size of p, and push all the item factor vectors ((j, H_j) where j=1 to n) into each queue in Q
while not converged **do** // parallel task
    if $Q[my\_rank]$ is not empty **then**
        Select and pop an element, (j, H_j), from the $Q[my\_rank]$
        for each $r_{i,j} \in R_{*,j}$ **do**
            Set number of applied updates on current nonzero $r_{i,j}$
            Update related factor vectors W_i and H_j respectively
        end for
        Push completed task, (j, H_j), into randomly selected processor's queue
    end if
end while

The workers might have different number of the ratings for the same item vectors, and this case probably results in running time differences among the workers. Therefore, selecting the next workers randomly might lead to the load imbalance and locking issues. To avoid these possible issues, a greedy-based scheduler is proposed, where decision for the next processor is made by selecting a worker who has minimum number of elements in its own queue among the available workers. Hence, slower workers have less loads. The developed scheduler solves the issue regarding not only being different number of the ratings for the same item vector, but also the difference among the workers in terms of hardware equipment. As a result, dynamic load balancing is achieved by using the greedy-based scheduler.

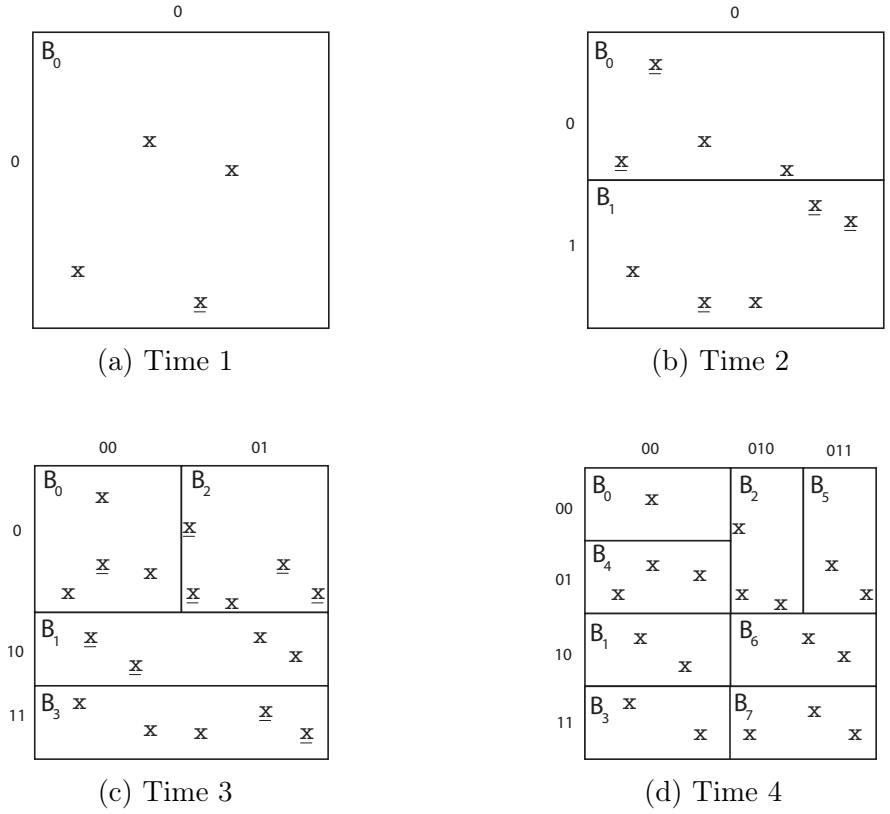
The piece of blocks are set to fixed number of vectors as a hundred to optimize the trade-off based on the synchronization frequency as discussed in Section 3.4. Experimental results show that almost linear speedup is achieved by the NOMAD algorithm on single machine and distributed multiple machines. Moreover, NOMAD outperforms DSGD and FPSGD** in terms of convergence in both shared memory and distributed memory environment. However, NOMAD is not compared with FPSGD in this study, and its performance is much better than FPSGD** as stated in the FPSGD study.

3.6 MLGF-MF: Fast and Robust Parallel SGD Matrix Factorization

Oh et al. [5] develop a parallel SGD algorithm for block-storage devices such as SSD disks, named Multi-level Grid File for Matrix Factorization (MLGF-MF), that introduces multi-level grid file partitioning technique to be robust for skewed datasets. The main consideration of this study is to avoid high scheduling cost arising from the load imbalance among workers. Hence, the authors make use of matched blocks obtained from the multi-level grid file partitioning scheme. MLGF-MF can be also adapted to shared memory environment.

Besides parallel SGD algorithms in both shared memory and distributed memory systems, the only existing algorithm developed for the block-storage devices is GraphChi [6] that is selected as a baseline for this study by the authors. GraphChi uses sequential order during update processes that leads to slow convergence, and it suffers from waiting I/O operations to execute CPU resources since the operations in I/O is slower than memory. MLGF divides the input data into blocks according to pre-specified capacity for each block as an input parameter before the partitioning. In other words, if a block has more ratings than the assigned capacity, MLGF partitions this block dynamically to overcome the load imbalance issue occurred especially in skewed matrices. In addition, MLGF-MF does not suffer from the idling issue to use CPU resources since it provides CPU utilization

by making I/O operations asynchronously. Therefore, CPU and I/O operations are overlapped which solves the idling issue.



(a) Time 1 (b) Time 2

(c) Time 3 (d) Time 4

Figure 3.5: The MLGF partitioning strategy (capacity=3).

In the multi-level grid file partitioning, hash function is used to represent the partitioned regions by naming them with a hash value hierarchically as shown in Figure 3.5, where x denote the ratings. In this example, the block capacity is determined as three, and initially (at $time_1$) there is only one block, B_0, that includes three ratings. At that time a rating is inserted (new ratings shown with $\underline{x}$), and the capacity for B_0 is exceeded. Hence, B_0 is partitioned into two regions as B_0 and B_1 at $time_2$, and the new block is represented with hash values as illustrated on the top and left side of the matrix. In this concept, an entry in a directory which is a set of region according to hash values, might point out another directory. As a result, the partitioning process continues recursively until all the ratings are placed in the rating matrix, and the number of the ratings in each

region can not be more than the pre-specified capacity as illustrated in Figure 3.5.

The MLGF partitioning for the given example is completed as shown in Figure 3.5d, where interchangeable regions are not directly realized in contrast to the DSGD model in such a way that a partitioned region can have shared rows or columns with another region, even if they are not overlapped. Hence, the authors propose *partial match query processing* to find noninterchangeable regions instead of the interchangeable regions by starting from root directory of obtained result from the MLGF partitioning. The process continues while a region has shared rows or columns with the query region, whereas it is terminated when an entry does not have any shared rows or columns with the query region. The overall procedure of the MLGF-MF algorithm is given in Algorithm 8.

Algorithm 8 The overall procedure of MLGF-MF

Input Rating matrix (R), user factor matrix (W), item factor matrix (H), number of processors (p), and number of total updates (u)

1: Initialize factor matrices W and H, and $total\_update = 0$
2: **while** $u > total\_update$ **do** // parallel task
3: Get an interchangeable block, $B_{selected}$ by locking other blocks
4: **for** each $r_{i,j} \in B_{selected}$ **do**
5: Update user and item vectors W_i and H_j, respectively
6: $total\_update$ ++
7: **end for**
8: **end while**

The idling issue in GraphChi for the CPU operations is solved by keeping both CPU and I/O operations busy in an asynchronous manner. While applying an update procedure for a rating, future block (will be updated after the current block) is found, and an I/O request is created asynchronously. After the update operations in the current block are completed, the future block is added as a new job. Therefore, the CPU and I/O operations are overlapped.

MLGF-MF is compared not only with GraphChi but also with shared memory algorithms such as FPSGD and NOMAD. Experimental results show that MLGF-MF outperforms NOMAD in terms of convergence with Netflix and Yahoo datasets. Although MLGF-MF produces almost the same results with FPSGD,

it outperforms FPSGD with skewed datasets generated by the authors. On the other hand, MLGF-MF produces much better results than GraphChi since there is no locking issue in MLGF-MF. In addition, they used different disks while conducting the experiments to make comparison, and obtain that page size does not affect the convergence rate.

Chapter 4

DSGD: Large-Scale Matrix Factorization with Distributed Stochastic Gradient Descent

Recommender system is an application used commonly by online businesses to increase popularity of the items. The provided services regarding the items are recently being expanded into many different areas including movie-rental services, social media and shopping. By making use of recommender systems, the businesses get knowledge of potential user interests to existing items, whereas the users easily find the items that probably appeal to their desires. Throughout this process, the latent relationships between the users and the items are discovered by considering the user preferences and features of the items. In the real-world systems, there are many missing ratings since only a small group of the users rate the items. The main goal of recommender systems is to predict the missing preferences of the users by making use of the existing ratings given by the users to the items in the past. During this process, there are two popular approaches used to build more accurate recommender systems as described below.

Content filtering and collaborating filtering are the techniques that take a set

of users, a set of items and existing ratings as input and predict the missing ratings as output to make decision which items can be recommended to which users. Content filtering forms a profile for each user by using features of the items rated by the user in the past. Then, existing features in the user profile are compared with features of an item to recommend the item to the user. Within this time period, the ratings given by other users to the item are not considered by the content filtering technique. Therefore, content filtering is capable of recommending new items which are not rated by any user yet. Conversely, collaborative filtering considers the relationship among the users and recommends the items without forming any profile. The past actions of a user are analyzed to find similar users in terms of the preferences, and then the items interested by similar users are recommended to the user. Therefore, the collaborative filtering technique makes use of the similarities among the users without considering the content of the items. However, this technique is stuck in case of a new user or item is added to the system, where it can not find similar users to the new user since there is no rating given by the new user to an item yet. Similarly, the new item also can not be recommended to a user since it is not rated by any user yet. This issue is called *cold start problem* and occurs in the collaborative filtering algorithms. On the other hand, the content filtering algorithms do not suffer from the cold start issue since a profile is created for each user and item by using their features without considering the relationships among each other. However, the collaborative filtering algorithms produce more precise results than the content filtering algorithms, meaning that the predictions of the collaborative filtering algorithms regarding the missing entries are more accurate. [10, 11]

The most well-known application area of collaborative filtering is the latent factor models which are developed to predict the missing ratings by applying matrix factorization [11]. The existing ratings are moved to a matrix plane where the set of the users and the set of the items are represented with the rows and the columns, respectively. Then, a factor matrix is created for each set as user factor matrix and item factor matrix. The rating matrix is summarized with the factor matrices by using a matrix factorization model. Both factor matrices are updated using the existing ratings in the rating matrix during the factorization process,

and finally the missing entries are predicted by using vectors of the updated factor matrices. The prediction quality is measured using the loss function in Equation 2.2 and increased when this function is minimized.

Stochastic gradient descent (SGD) algorithm is an iterative algorithm that has recently become the most popular technique applied for recommender systems among the loss optimization techniques covered in Chapter 2. Then, parallelization of SGD has been studied to factorize large-scale matrices while building recommender systems for real-world systems. There are two important situations should be considered while developing a parallel SGD algorithm to make it more efficient in terms of prediction accuracy and running time as following.

- The SGD algorithm contains dependent update procedures that make the parallelization difficult. The values of the updated vectors in the current iteration are used in the next iteration. Therefore, using the most recent updated vectors during the update procedures becomes harder which may result in slower convergence due to steal data usage. Hence, the developed parallel model should not allow the steal data usage by forcing workers[1] to apply update procedures with using the most recent vector values all the time. In other words, quality of the update procedures in the SGD algorithm should be maintained by applying the same update order with the sequential SGD algorithm.

- Load imbalance is a critical problem that leads performance issues in parallel applications. The usage of the most recent updated vectors are provided using a synchronization process in synchronous algorithms, where the workers communicate the updated vectors among each other simultaneously. During this process, the workers having fewer ratings might wait for the others since they finish their tasks earlier and have to wait for the others in order to continue to work by completing the communication process. Hence, this problem increases the running time of the developed parallel algorithm. To avoid such a problem, each worker should have similar number of ratings to be processed, meaning that the load balance among the

[1]Threads in shared memory systems or processors in distributed memory systems

workers should be considered while developing the parallel SGD algorithm.

In the light of the situations described above, Gemulla et al. [7] developed a parallel SGD algorithm, named Distributed SGD (DSGD), to factorize large-scale matrices while building efficient applications based on recommender systems. DSGD is the first fully distributed SGD algorithm and influences the studies in this area. The authors introduce *stratified stochastic gradient descent* (SSGD) model in which the rating matrix is partitioned into blocks. Each processor works independently on the ratings of a block at a time, and the total loss is calculated as weighted sum of the local losses in the blocks as given in Table 2.1. In DSGD, processors work on the most recent updated vectors of the factor matrices using *bulk synchronization process* in which the updated item vectors are communicated among the processors periodically. This makes DSGD a synchronous algorithm. The factorization of large matrices with fast convergence rates is achieved by DSGD in distributed memory environment.

The popular works [1, 7, 2, 3, 4, 5] regarding parallelization of the SGD algorithm are covered in Chapter 3. Most of them base their methods on DSGD and propose their algorithms by originating from this model. Thus, the algorithms developed in this area such as [2, 4] are inspired by this outstanding study to avoid the steal data usage during the update procedures. In addition, the DSGD model includes a flexible design which can be adapted to shared memory environment by replacing processors with threads. Although DSGD is the leading work due to having these crucial features, there is no such a study regarding detailed analysis of the DSGD algorithm as far as we researched. Hence, we particularly focus on the DSGD algorithm by implementing and testing it from different perspectives in detail. Our contributions regarding extensive analysis of the DSGD algorithm are stated as follows.

- From the real-world datasets perspective, the DSGD study includes experimental results regarding only Netflix dataset. When we consider the other models described in Chapter 3, the experiments are performed by using a few more real-world datasets in addition to Netflix dataset. Therefore,

there is only a few real-world datasets used while testing and analyzing the DSGD algorithm in the literature. In contrast, we produced many real-world datasets by processing published raw data and conducted the experiments regarding our DSGD implementation with these datasets using different number of processors.

- The DSGD algorithm shuffles the input rating matrix randomly, however, there is no information regarding how random permutation improves the load balance among the processors and makes the convergence faster in the DSGD study. To fill this gap, we generated static[2] and random[3] partitioning files for different dataset and each number of processors used in the experiments. Then, we implemented a load imbalance calculator and obtained the load imbalance among processors for both partitioning schemes. Moreover, we conducted the experiments by using both partitioning schemes and obtained the convergence rates for each partitioning scheme. Then, we discussed the effect of applied random permutation on the rating matrix with results of the load imbalance and the convergence rates in Chapter 5.

- In contrast to the studies covered in Chapter 3, we conducted the experiments in terms of the load imbalance, speedup and convergence metrics with using different number of processors. There is no such a work which measures performance of the DSGD algorithm in an extensive manner as far as we researched.

The remainder of this chapter is organized as follows. We describe the SSGD model that underlies of DSGD model in Section 4.1. Then, we provide structure of the DSGD model by examining its concept and algorithm in Section 4.2. Finally, we mention the implementation details in Section 4.3.

[2]The rating matrix is divided into blocks without any operation.

[3]The random permutation is applied on the rating matrix by row wise and column wise before dividing the rating matrix into blocks.

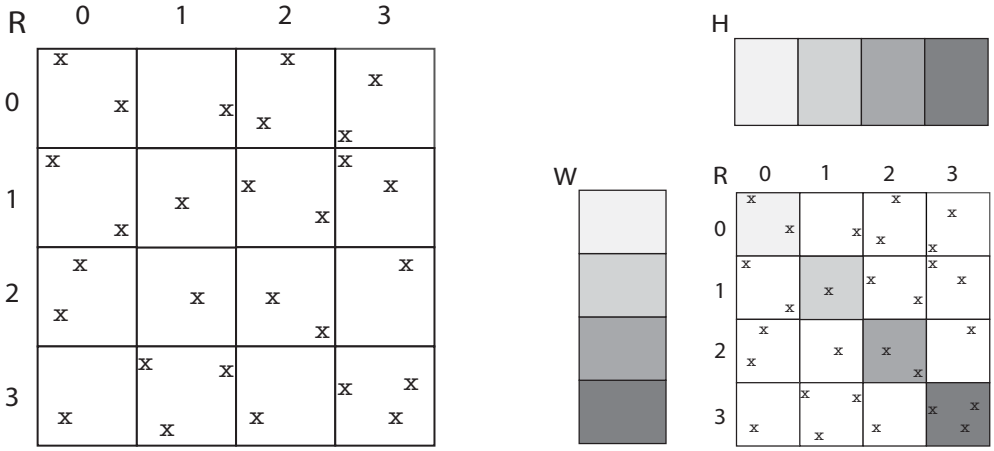
4.1 SSGD (Stratified SGD)

SSGD is an SGD variant algorithm specialized by the authors [7] which is later used to develop the DSGD model. DSGD gains feature of being fully distributed SGD algorithm by specializing the SSGD model. Figure 4.1a illustrates an example of the SSGD model where the rating matrix R is divided into blocks which are not overlapped, meaning that a rating $r_{i,j}$ in the rating matrix R can not be located in more than one block. The contribution of the authors while specializing the SSGD model is exploiting interchangeable blocks. The terms used in the specialized SSGD model are worth to define here to make the remaining parts more understandable. Interchangeable blocks are any of two or more blocks that do not have shared rows and columns of the rating matrix. Stratum is a set of the interchangeable blocks with size p[4]. Strata is the set of all possible stratum combinations for the rating matrix. A subiteration is completed when a stratum is processed, whereas an iteration is completed when a set of stratums with size p are processed. In other words, processing different p stratums means a complete iteration.

The specialized SSGD model is given for the rating matrix R in Figure 4.1b. There are four processors working on the rating matrix R which is divided into sixteen blocks. Similarly, the user and item factor matrices W and H, respectively, are divided into four blocks. The colors represent the processors. A stratum for the rating matrix R is illustrated where the processors work simultaneously on the interchangeable blocks. The current subiteration is completed when all the processors complete their update procedures for all the ratings in the current blocks. Similarly, an iteration is completed for matrix R in DSGD algorithm when four different stratums are processed as illustrated in Figure 4.3. The details regarding the DSGD algorithm including input data distribution among the processors and communication of the updated vectors will be stated in Section 4.2

In sequential SGD algorithm, global loss is calculated by consecutively applying update procedures for all the ratings in the rating matrix once, whereas in

[4]The number of processors working on the rating matrix.



(a) The stratification technique. (b) The interchangeable blocks.

Figure 4.1: The SSGD model.

parallel SGD algorithms it is calculated as sum of local losses which are called stratum losses. In the SSGD model, processors calculate their local loss by applying the update procedures on the ratings of their block in the current stratum independently. Then, the global loss is calculated as weighted sum of stratum losses. The weight of a stratum is generally considered as a constant that is proportional to the time spent for processing the stratum. In other words, the number of the ratings in a stratum represents its weight in such a way that total weight in the rating matrix is set to 1. The convergence of the SSGD model is determined under a set of conditions including learning rate (or step size), loss, stratification and stratum sequence. The learning rate (ϵ) is an input parameter of the SGD algorithm used to specify momentum of updating system parameters. Having a lower learning rate results in slower movements, and leads to slow convergence under other sufficient conditions or being stuck in local minima, however, it achieves finding the local minima. On the other hand, the momentum of the system is increased with using higher learning rates, but it might diverged instead of convergence. The authors state that the learning rate can be selected using the iteration number as $\epsilon = 1/I$, where I denotes the iteration number. They prove convergence of the specialized SSGD model under stratification technique. In addition, we also show the convergence of the SSGD model with experimental results as discussed in Chapter 5. We obtained that the number of blocks

generated in the stratification period does not affect the convergence, meaning that the DSGD model converges to the same rates even if different number of processors work on the same rating matrix. On the other hand, there is no exact information regarding the sequence of stratums, however, the idea is to find out which sequence provides fast convergence by trying different sequences.

4.2 DSGD (Distributed SGD)

DSGD makes use of the SSGD algorithm to handle inherent sequential update procedures of the SGD algorithm. The introduced SSGD model is specialized by integrating a new concept for selection of blocks interchangeably which is later used in development of the DSGD algorithm. The developed model allows processors to always work on the most recent updated vectors of the factor matrices. The bulk synchronization process is applied to make communication of the updated blocks of item factor matrix among processors at the same time. During the bulk synchronization process, all the processors communicate the updated factor blocks together without making computation, which makes DSGD a synchronous algorithm.

The rating matrix R is divided by row block wise to minimize the communication cost regarding size of the updated vectors during the bulk synchronization process since the number of the users is much more than the number of the items in the real-word systems. All the input data including the rating matrix R, the user factor matrix W and the item factor matrix H is partitioned into blocks using a data independent model based on the introduced SSGD algorithm. The input data distribution for the rating matrix R is illustrated in Figure 4.2. At the end of the input data partitioning, the rating matrix R with size $m \times n$ includes p x p blocks, and each block has m/p rows and n/p columns. On the other hand, the factor matrix W includes the number of m/p blocks, each of them has m/p rows and k columns, whereas the factor matrix H includes n/p blocks, each of them has n/p rows and k columns as illustrated in the figure. The colors on the blocks represent their owners after the input data distribution, and the mapping

for the colors and the processors are given on the top left side of the figure.

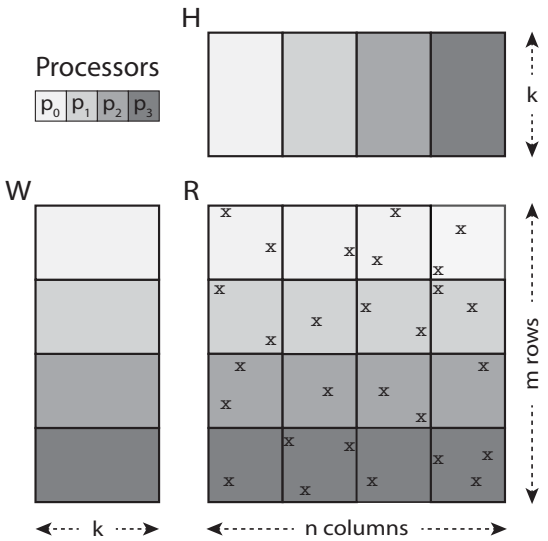


Figure 4.2: The input data distribution in the DSGD algorithm.

In the DSGD algorithm, processors work on the rating matrix by updating vectors of the different factor blocks by applying update procedure on the ratings of the interchangeable blocks as defined follows. For the rating matrix R^{mxn}, B_x and B_y are called interchangeable blocks if and only if the following statements are provided, $a \neq c$ and $b \neq d$, $r_{a,b} \in B_x$ and $r_{c,d} \in B_y$ $(0 \leq a, c < m,\ 0 \leq b, d < n)$, where m and n denote the number of the rows and the columns in the rating matrix R, respectively. A subiteration is completed when a stratum is processed, whereas a full iteration is completed when a set of stratum with size p is processed. After each subiteration, updated blocks of the factor matrix H are communicated among the processors using the bulk synchronization process. On the other hand, there is no need for communication of the updated blocks of the factor matrix W since they belong to a row stripe which is being processed by single processor.

The overall procedure of the DSGD algorithm is given in Algorithm 9. Initially, the rows and the columns of the rating matrix R are randomly shuffled to increase the load balance among the processors. The detailed information regarding how the random permutation increases the load balance and speed up the convergence are given in Section 5.3.1 and Section 5.3.3, respectively. Then, the input data is partitioned and distributed among the processors according to stratification

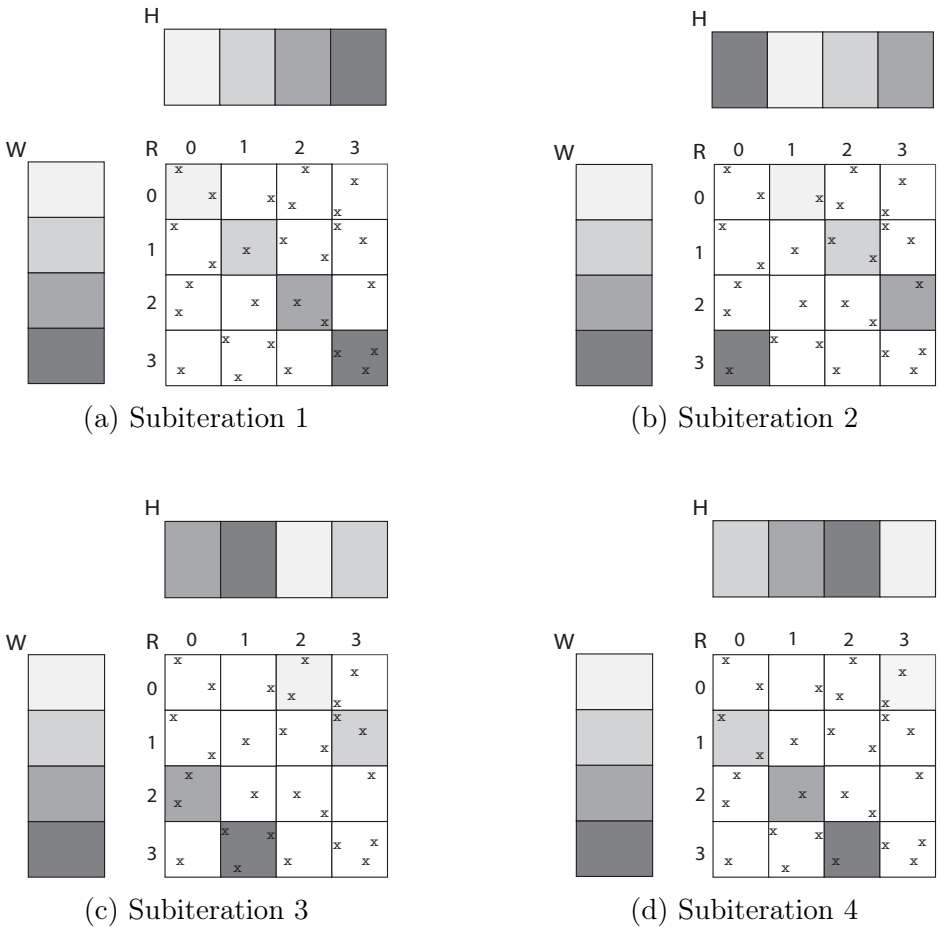
Algorithm 9 The overall procedure of DSGD

Input Rating matrix (R), user factor matrix (W), item factor matrix (H), number of processors (p) and iteration count ($iter$)

```
1: Partition matrix R, H and W into p x p, p and p blocks, respectively
2: Each processor has an array, B, to keep its own blocks in matrix R
3: for i = 1, 2, ..., iter do // parallel task
4:     for j = 1, 2,, ..., p do
5:         for each rating, r_{x,y}, in B[j] do
6:             Apply update procedure on r_{x,y}
7:         end for
8:     end for
9: end for
```

technique and the number of the processors used in SSGD model. Next, the processors start to work simultaneously on a stratum of the rating matrix R, and then this process is repeated the number of the processors times for a complete iteration. Finally, the execution is terminated when predetermined number of iterations is achieved or there is no more improvement in the convergence.

An example regarding working progress of the DSGD algorithm through an iteration is illustrated in Figure 4.3. Each subfigure represent a subiteration, in which each processor works on a block of a stratum. In the first subiteration, the processors start to work with the interchangeable blocks on diagonal of the rating matrix R as illustrated in Figure 4.3a, where p_0, p_1, p_2 and p_3 apply update procedures on the ratings of block $R_{0,0}$, $R_{1,1}$, $R_{2,2}$ and $R_{3,3}$, respectively. Then, blocks of the factor matrix H are communicated among the processors by applying a bulk synchronization process. Then, each processor starts to work with the ratings in the next block of the same row stripe using the received updated block of the factor matrix H in the second subiteration. This process is repeated p times for a complete iteration as the remaining subiterations are illustrated in Figure 4.3c and Figure 4.3d. At the end of each iteration, the convergence and the iteration number is checked, and then the execution is terminated if predetermined iteration number is reached or there is no more improvement in the convergence rate. Otherwise, the same steps through an iteration are entirely repeated.



(a) Subiteration 1

(b) Subiteration 2

(c) Subiteration 3

(d) Subiteration 4

Figure 4.3: A complete iteration in the DSGD algorithm.

4.3 Implementation

We implemented the DSGD model in C programming language using built-in message passing interface (MPI) library. We used OpenMPI as MPI version for both compilation and execution processes due to its more production oriented functionality among the existing MPI versions. All source files are compiled by enabling gcc -O3 optimization flag. Before mention implementation details, the input parameters used in our implementation are worth to define here as follows.

- `Matrix file` includes a rating matrix in matrix-market format.

- `Partition file` contains the distribution of the rows and the columns in the rating matrix among processors.
- `Iteration count` is the number of total iterations that specifies how many times the ratings in the rating matrix are updated.
- `Factor size` is the dimension of the factor matrices W and H.

We created the rating matrices in the matrix-market format by processing the published raw data of which features are described in Section 5.1. Then, we generated random partitioning files for each dataset and different number of the processors by applying the random permutation on the rows and the columns of the rating matrices, whereas no operation is applied while generating static partitioning files. Then, we implemented a load imbalance calculator to obtain the effect of the partitioning strategies on the load balance by considering the ratings are processed through an iteration by each processor. The formula regarding the load imbalance calculation is given in Equation 4.1. The iteration count and the factor size parameters are set to 100 and 8, respectively.

$$Imbalance = \frac{p \times max(\sum_{i=0}^{p} B_i)}{\sum_{i=0}^{p} B_i} \tag{4.1}$$

where p is the number of processors, B_i denotes the number of ratings owned by p_i through a complete iteration in the DSGD algorithm.

As mentioned previously, the number of the users is much more than the number of the items in most of the real-world systems. The produced real-world datasets are stated in Table 5.1, and features of the datasets prove this idea except `Yahoo Music Track-2` dataset. Hence, we partitioned the rating matrix by row block wise among the processors initially to minimize the communication cost regarding communication of the updated factor blocks during the bulk synchronization process. Therefore, only item factor blocks are communicated at the end of each subiteration since the user factor blocks are processed by single processor.

Most of the rating matrices are very sparse since only a small group of the users rate for the items. Therefore, keeping the matrices in memory efficiently

is very important to optimize the memory usage. We used a data structure, which is compressed sparse row format (CSR), to store the input rating matrix in memory. On the other hand, there is no need for using a data structure to keep the factor matrices in memory since they are dense matrices. The CSR format includes three one-dimensional arrays to keep sparse matrices, and not only occupies memory spaces for necessary data, but also avoids cache misses. In other words, only the existing ratings are stored in memory by using the CSR format and the local indices are used to provide uniform memory access. We kept each block within a CSR, and then each processor works on the ratings of p CSR blocks through an iteration in our DSGD implementation.

For communication of item factor blocks during the bulk synchronization process at the end of each subiteration, we used `MPI Sendrecv` function that includes an implicit barrier, meaning that a processor completes its communication task can start to work with the ratings in the next block. In addition, *bold driver* heuristic is used to make the convergence faster as the authors stated in the DSGD study. The bold driver heuristic dynamically updates the step size after each iteration by comparing the calculated most recent loss value with the previous loss value as shown below.

$$f(step\ size) = \begin{cases} step\ size \times 0.5, & \text{if } loss_{cur} > loss_{prev} \\ step\ size \times 1.05, & \text{if } loss_{cur} < loss_{prev} \end{cases}$$

The step size is increased by five percentage when the loss decreases, whereas it is decreased by fifty percentage when the loss is increased. According to our experimental results, the bold driver heuristic does not play an important role in the convergence of the DSGD algorithm since the loss starts to always decrease after the first a few iterations. Moreover, we obtained worse convergence rates by using smaller number of processors with denser datasets when the bold driver heuristic is applied.

Chapter 5

Experimental Results

In this chapter, we conduct experiments using different number of processors (K) as 2, 4, 8, 16, 32 and 64 with real-world datasets, properties of which are presented in Table 5.1. Then, the results are discussed in terms of the speedup and the convergence metrics. Moreover, we obtain the load imbalance statistics for each dataset based on the partitioning scheme and compare the load imbalance for static and random partitioning strategies.

The remainder of this chapter is organized as follows. Firstly, we describe the properties of the produced datasets. Then, we mention experimental setup by describing features of the system in which we conduct the experiments. Finally, we discuss the experimental results in terms of load imbalance, speedup and convergence.

5.1 Datasets

From the real-world datasets perspective, only Netflix is used in the experiments of DSGD study, whereas a few more datasets are used besides Netflix in the other studies covered through Chapter 3. Therefore, there is a few real-world datasets used in the experiments of the studies related to DSGD. In contrast, we produced

many real-world datasets by processing the published raw data and conducted the experiments regarding our DSGD implementation with these datasets. In this section, we examine the `Amazon`, `Last.fm`, `Movielens`, `Netflix` and `Yahoo Music` datasets with their subsets and describe their features. The properties of the produced datasets are described in the remaining of this section. For the experiments, we select some of them as given in Table 5.1.

Dataset	# users	# items	# ratings
Amazon Item	21,176,522	9,874,211	82,677,131
Amazon Books	8,026,324	2,330,066	22,507,155
Amazon Clothing	3,117,268	1,136,004	5,748,920
Amazon Electronics	4,201,696	476,002	7,824,482
Amazon Movies and TV	2,088,620	200 941	4,607,047
Last.fm	359,349	268,758	17,559,530
Movielens-20m	138,493	26,744	20,000,263
Movielens-latest	270,896	45,115	26,024,289
Netflix	480,189	17,770	100,480,507
Yahoo Music Track-1	1,000,990	624,961	252,800,275
Yahoo Music Track-2	249,012	296,111	61,944,406

Table 5.1: Properties of produced datasets.

5.1.1 Amazon Dataset

We produced rating matrices regarding the `Amazon` dataset by processing the published raw data [31] that contains product reviews in the Amazon website between 1996 and 2014. The raw data includes over 142 million reviews, however, some of them are duplicate since similar features of the products are merged by the Amazon website. Hence, we used the another version of the raw data, in which the duplicate reviews are removed, that contains over 82 million unique reviews by almost 21 million users to over 9 million items. In addition to the reviews, the raw data contains ratings, item-to-item relationships, timestamps,

helpfulness votes, product image, price, category and sales-rank information. The item-to-item relationships are established to reveal the relation among different items which are reviewed by the same users.

Each review is stored in a javascript object notation (JSON) file as following [31],


```
{
  "reviewerID": "A2SUAM1J3GNN3B",
  "asin": "0000013714",
  "reviewerName": "J. McDonald",
  "helpful": [2, 3],
  "reviewText": "I bought this for my husband who plays
      the piano.  He is having a wonderful time playing
      these old hymns.  The music  is at times hard to
     read because we think the book was published for
     singing from more than playing from.  Great
     purchase though!",
  "overall": 5.0,
  "summary": "Heavenly Highway Hymns",
  "unixReviewTime": 1252800000,
  "reviewTime": "09 13, 2009"
}
```


where `"reviewerID"` is id of the reviewer, `"asin"` is id of the product, `"reviewerName"` is name of the reviewer, `"helpful"` is helpfulness rating of the review, `"reviewerText"` is text of the review, `"overall"` is rating of the product, `"summary"` is summary of the product, `"unixReviewTime"` is time of the review in unix format and `"reviewTime"` is also time of the review but it is in raw format. We downloaded ratings version of the complete review data, called `item dedup`, that contains `"reviewerID"`, `"asin"`, `"overall"` and `"unixReviewTime"` information of each existing review in column-separated values (CSV) format. In addition to `item dedup`, we also downloaded 24 small real subsets commonly used for experimentation. These small subsets are created by categorizing the

products in Amazon website. The details of all the `Amazon` datasets including the number of the users, the items and the ratings are given in a descending order of the number of ratings in Table 5.2.

Dataset	# users	# items	# ratings
Item dedup	21,176,522	9,874,211	82,677,131
Books	8,026,324	2,330,066	22,507,155
Electronics	4,201,696	476,002	7,824,482
Clothing	3,117,268	1,136,004	5,748,920
Movies and TV	2,088,620	200,941	4,607,047
Home and Kitchen	2,511,610	410,243	4,253,926
CDs and Vinyl	1,578,597	486,360	3,749,004
Cell Phones and Accessories	2,261,045	319,678	3,447,249
Sports and Outdoors	1,990,521	478,898	3,268,695
Kindle Store	1,406,890	430,530	3,205,467
Health and Personal Care	1,851,132	252,331	2,982,326
Apps for Android	1,323,884	61,275	2,638,172
Toys and Games	1,342,911	327,698	2,252,771
Beauty	1,210,271	249,274	2,023,070
Tools and Home Improvement	1,212,468	260,659	1,926,047
Automotive	851,418	320,112	1,373,768
Video Games	826,767	50,210	1,324,753
Grocery and Gourmet Food	768,438	166,049	1,297,156
Office Products	909,314	130,006	1,243,186
Pet Supplies	740,985	103,288	1,235,316
Patio Lawn and Garden	714,791	105,984	993,490
Baby	531,890	64,426	915,446
Digital Music	478,235	266,414	836,006
Amazon Instant Video	426,922	23,965	583,933
Musical Instruments	339,231	83,046	500,176

Table 5.2: Amazon datasets.

We created matrix files in matrix market format by processing the downloaded raw data regarding the complete review data and the smaller subsets. Then, we selected `item dedup` and the largest four datasets from the smaller subsets in terms of the number of the ratings to use in the experiments as `Books`, `Electronics`, `Clothing` and `Movies and TV`.

5.1.2 Last.fm Dataset

We produced rating matrices regarding the `Last.fm` dataset by processing the published raw data [34] that is generated using the web-service application of Last.fm website in 2010. The `Last.fm` dataset includes two different versions such that smaller one contains a thousand users. We consider the larger `Last.fm` dataset that contains the ratings given by almost 360 thousands users to 270 thousands songs. The raw data are stored in two different files in tab-separated values (TSV) format. The files and their scheme are described as follows.

- The `usersha1-artmbid-artname-plays.tsv` file contains `"user-mboxsha1"`, `"musicbrainz artist id"`, `"artist name"` and `"plays"` attributes of the artists. `"musicbrainz artist id"` is id of the user in musicbrainz encyclopedia, and `"plays"` shows that songs of an artist is how many times played by a user. A sample line from the raw data file is given as follows [34], `"00000c289a1829a808ac09c00daf10bc3c4e223b f779ed95-66c8-4493-9f46-3967eba785a8 letzte instanz 387"`, where a user whose id is `"00000c289a1829a808ac09c00daf10bc3c4e223b"` plays the songs of Letzte Instanz 387 times.
- The `usersha1-profile.tsv` file contains `"user-mobxsha1"`, `"gender"`, `"age"`, `"country"` and `"signup"` information of the users. `"signup"` represents the registration date of a user to the Last.fm website. A sample line from the raw data file is given as follows [34], `"000163263d2a41a3966a3746855b8b75b7d7aa83 m 27 Sweden Jan 5, 2007"`, where a boy who is 27 years-old and from Sweden registered to Last.fm website in January 5, 2007.

We downloaded sanitized version of the larger `Last.fm` dataset in which the listen counts of the songs are quantized to 10, meaning that the ratings are normalized to values from 0 to 10. Then, we created a matrix file in matrix market format by processing the downloaded raw data including the files described above and used it in the experiments.

5.1.3 Movielens Dataset

We produced rating matrices regarding the `Movielens` dataset by processing the published raw data [35] that is generated from the Movielens website and lastly updated in 2018 by GroupLens Research. Movielens is a movie recommendation system, and the produced dataset includes different subsets as given in Table 5.3. The features of the subsets are described below.

`Movielens-20m` [36] is one of the largest `Movielens` dataset that contains over 20 million ratings given by 138,493 users to 26,744 movies between 1995 and 2015. The users have at least twenty ratings are selected randomly and used in this dataset. Privacy preservation is applied while collecting the ratings in such a way that only an ID is generated for each user and any personal information regarding the user is not stated in the dataset. The raw data is stored in six files in CSV format and grouped regarding their content as following.

- The `ratings.csv` file stores ratings data file structure that contains the ratings given by the users to the movies. Each line of the file includes `userId`, `movieId`, `rating` and `timestamp` attributes as sorted by `userId` and `movieId`, respectively. A rating given by a user to a movie is in the range from 0.5 to 5 with intervals of 0.5.

- The `tags.csv` file stores tags data file structure that contains the tags assigned to the movies by the users. Each line of the file includes `userId`, `movieId`, `tag` and `timestamp` features as sorted by `userId` and `movieId`, respectively. The tag is a single word created by the users to describe the movie.

- The `movies.csv` file stores movies data file structure that contains the movies with their `movieId`, `title` and `genres` attributes. The `title` attribute and release year of a movie are retrieved from themoviedb website. The `genres` for a movie is selected from the predetermined set of genres that includes 18 genres such comedy, fantasy, romance etc.

- The `links.csv` file stores links data file structure that contains `movieId`, `imdbId` and `tmdbId` information. These attributes are used as unique identifiers of a movie for its links in movielens, imdb and themoviedb websites.

- Tag genome contains the scores given to the movies for all the tags, meaning that each movie has a score for each tag. The tag genome data is generated to classify the movies by using the user reviews in a learning mechanism. Therefore, a movie's relevance to a tag can be known with this data structure. The genome data contains 12 million relevance scores for 1,100 different tags, and it is presented within two files as follows.

  - The `genome-scores.csv` file contains `movieId`, `tagId` and `relevance` features, where `relevance` is a floating number between 0 and 1.
  - The `genome-tags.csv` file includes the mapping between `tag` and `tagId`. The name of the tags used in `genome-scores.csv` file can be known by using this mapping regarding `tagId` attribute.

`Movielens-latest` [36] is the largest `Movielens` dataset that contains almost 26 million ratings given by 270,896 users to 45,115 movies between 1995 and 2018. The users having at least one rating are selected randomly and used in this dataset, and privacy preservation for the users is again applied in this dataset. All the data structures and their scheme are the same with the `Movielens-20m` dataset. The difference is regarding of the usage such that `movielens-latest` dataset is used as a development dataset and might be changed continuously. However, `movielens-20m` is more stable. In addition, the genome data of the `Movielens-latest` dataset contains 14 million relevance scores for 1,100 different tags. The smaller subsets of the `Movielens` dataset include the `Movielens-100k`, `Movielens-1m` and `Movielens-10m` datasets, and their schemes are also similar to `Movielens-20m` and `Movielens-latest` datasets.

Dataset	# users	# items	# ratings
Movielens-100k	943	1,682	100,000
Movielens-1m	6,040	3,706	1,000,209
Movielens-10m	69,878	10,677	10,000,54
Movielens-20m	138,493	26,744	20,000,263
Movielens-latest	270,896	45,115	26,024,289

Table 5.3: Movielens datasets.

We downloaded the raw data regarding all the subsets of the `Movielens` dataset and created matrix files in matrix market format for all of them by processing the raw data files described above. Then, we selected and used the largest two subsets in the experiments, which are `movielens-20m` and `movielens-latest`.

5.1.4 Netflix Dataset

We produced a rating matrix regarding the `Netflix` dataset by processing the published raw data [32] that is collected from the Netflix website between 1998 and 2005 and presented for the Netflix Prize competition [12] in 2009. The dataset contains over 100 million ratings given by 480,189 users to 17,770 movies in the range from 1 to 5. The files including raw data are described as follows.

- A text file is generated for each movie of which the first line includes `movieId`, whereas each remaining line includes `customerId`, `rating` and `date` attributes. `customerId` is set with the values from 1 to 2,649,429 even though there are 480,189 users. The reason is that only a group of the customers rate for the movies.
- All the information regarding a movie is stored in the `movie-titles.txt` file with its `movieId`, `yearOfRelease` and `title` attributes. `movieId` is generated randomly, and `yearOfRelease` includes a value between 1890 and 2005.

- The qualifying dataset is stored in the `qualifying.txt` file and used to test developed model for the contest. This file includes the customers and their interest to the movies.

We downloaded the raw data and created a matrix file in matrix market format by processing the related raw data files described above. Then, we used them in the experiments.

5.1.5 Yahoo Music Dataset

We produced rating matrices regarding the `Yahoo Music` dataset by processing the published training raw data [33] that is generated from the Yahoo Music website. The items and the users having at least 20 rates and 10 rates, respectively, are selected for this dataset, and privacy preservation is considered during generation of the dataset by relabeling the items and the users. In this dataset, an item can be a track, album, artist or genre and rated in range from 0 to 100. There are totally four different subsets of the `Yahoo Music` dataset as given in Table 5.4. The files containing the raw data regarding the users and different item types are described below.

- The `trainIdx.txt` file contains all the ratings given by each user to the items. The users are separated with a line that only includes `userId` and `numberOfRatings` features. Each line in a user part includes `itemId`, `score` and `time` attributes of the rating given by that user.
- The `trackData.txt` file contains the information regarding the tracks with using `trackId`, `albumId`, `artitstId` and optionally `genreId`(s) attributes.
- The `albumData.txt` file contains the information regarding the albums with using `albumId`, `artitstId` and optionally `genreId`(s) attributes.
- The `artistData.txt` file contains the artists with using `artistId` attribute.

- The `genreData.txt` file contains the genres with using `genreId` attribute.

Dataset	# users	# items	# ratings
Small	249,012	296,111	61,944,406
Medium	500,269	445,440	123,318,314
Large	1,000,990	624,961	252,800,275
All	5,014,136	1,158,226	1,279,358,021

Table 5.4: Yahoo Music datasets.

We downloaded the raw data regarding the `large` and `small` subsets among the existing four subsets of the `Yahoo Music` dataset and called them `Yahoo Music Track-1` and `Yahoo Music Track-2`, respectively. Then, we created matrix files in matrix market format by processing the related raw data files described above, and used them in the experiments.

5.2 Experimental Setup

In our experiments, we used a machine equipped with Intel Xeon CPU E7-8860 running at 2.20GHz. It has 256 GB of RAM and 72 CPUs. There are four sockets, each of them contains 18 cores. In addition, L1d, L1i, L2 and L3 are the caches on the machine with size of 32 KB, 32 KB, 256 KB and 46 MB, respectively.

Dataset	Learning Rate(α)	Regularization Constant(λ)
Amazon	0.002	0.05
Last.fm	0.001	0.01
Movielens	0.0001	1
Netflix	0.002	0.05
Yahoo Music	0.0001	1

Table 5.5: Parameters of the SGD algorithm used in the experiments.

The selected learning rate (α) and the regularization constant (λ) parameters used in the update procedures of SGD algorithm are stated in Table 5.5. These parameters should be determined carefully for each dataset to speedup the convergence. The datasets are grouped under the name of the main dataset, and the same parameters are used for its all subsets. We examined the values of the parameters used in the existing works covered in Chapter 3 and used the same parameters for some of the datasets. The parameters used for `Movielens`, `Netflix` and `Yahoo Music` datasets are got from [3] and [2]. On the other hand, we try different parameter values for `Amazon` and `Last.fm` datasets and obtain them ourselves. In addition, factor size k is set to 8 in all the experiments, and the entries in the factor matrices are randomly initialized with single precision floating point numbers in the range of [-1.0, 1.0].

5.3 Results and Discussion

5.3.1 Load Imbalance

As mentioned previously in Chapter 4.3, we applied random permutation on the rating matrix to provide load balance among the processors before running the DSGD algorithm as the authors stated. We implement a load imbalance calculator and calculate the load imbalance in the produced datasets by using different number of processors. During the load imbalance calculation, we take the data independent block distribution into consideration, in which the rating matrix is partitioned by row block wise among the processors and never communicated during the working process. In other words, the number of the ratings in a complete iteration, which consists of p subiterations, for the processors is considered while calculating the load imbalance as the formula given in Equation 4.1. In addition, we also generate static and random partitioning files and obtain the improvement with the random permutation in terms of the load balance as given in Table 5.6, where p denotes the number of processors and two rows in the part of each dataset contain the load imbalance values as percentage for static

partitioning and random partitioning, respectively. For example, the load imbalance in `Amazon Books` dataset for 32 processor is obtained as 437.6% with static partitioning, whereas it is only 6.6% with random partitioning. Moreover, the improvement by the random partitioning is stated on the left column, where the first row of each dataset contains the average improvement for applied different number of processors as 2, 4, 8, 16, 32 and 64. The second row of each dataset contains the improvement when the number of processors is 64.

Dataset	Improvement	p=2	p=4	p=8	p=16	p=32
Amazon Item	Average : 339.7x	48.56	98.40	166.2	254.6	383.4
	$p = 64$: 114.6x	0.150	0.429	0.338	0.353	2.391
Amazon Books	Average : 90.69x	47.53	106.8	191.3	308.5	437.6
	$p = 64$: 50.15x	0.586	0.706	2.196	2.842	6.635
Amazon Clothing	Average : 129.0x	23.88	43.61	58.72	75.69	87.60
	$p = 64$: 67.98x	0.110	0.331	0.385	0.587	1.118
Amazon Movies	Average : 124.8x	40.30	99.33	186.5	318.5	506.5
	$p = 64$: 72.20x	0.212	0.489	1.817	5.100	4.250
Lastfm	Average : 0.142x	0.029	0.038	0.084	0.154	0.273
	$p = 64$: 0.172x	0.141	0.450	0.868	1.094	1.839
Movielens-20m	Average : 0.843x	0.113	0.903	2.210	2.518	3.814
	$p = 64$: 0.871x	0.682	0.611	1.490	4.811	7.089
Movielens-latest	Average : 1.356x	0.174	0.808	1.548	3.498	6.886
	$p = 64$: 2.095x	0.061	1.501	2.254	3.557	7.079
Netflix	Average : 1.197x	0.258	0.854	1.324	1.462	1.992
	$p = 64$: 1.100x	0.257	0.589	0.680	1.820	2.263
Yahoo Music-1	Average : 0.803x	0.157	0.570	1.430	2.301	4.108
	$p = 64$: 0.868x	0.345	0.945	1.354	2.533	4.405
Yahoo Music-2	Average : 0.925x	0.775	1.637	2.363	4.828	6.822
	$p = 64$: 0.936x	0.467	2.747	4.246	5.040	8.038

Table 5.6: Load imbalance results for static and random partitioning.

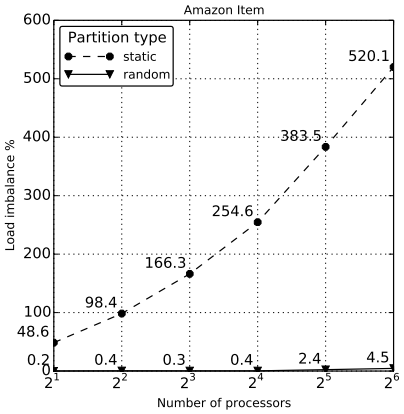
The figures regarding the load imbalance results are illustrated in Figure 5.1 and Figure 5.2. The results show that random permutation works well for much sparser datasets even though it is not too much effective for denser datasets including `last.fm`, `Netflix`, `Yahoo Music Track-1` and `Yahoo Music Track-2`. As shown in Figure 5.1, the load imbalance in `Amazon` datasets with static partitioning is very high, and random permutation improves the load balance by 67x-339x. On the other hand, there is no such a huge improvement for denser

datasets such as `Lastfm`, `Neflix` and `Yahoo Music`, where the results are almost the same for both partitioning schemes. The static partitioning already achieves good load balance for the denser datasets where the maximum load imbalance is around 2% and 5% in `Last.fm` and Netflix datasets, respectively, whereas it is around 10% in Yahoo Music datasets. Although the static partitioning produces a little bit better results than the random partitioning around ∼15%, these differences are not important since the load imbalance for the denser datasets is very low for both partitioning schemes such that it is between 1.9% and 8% even if 32 processors are used as illustrated in Figure 5.2. Therefore, we can not mention an improvement in such a case since the imbalance is already low and the difference between static and random partitioning in terms of load imbalance is also very low. Hence, we expect that the speedup does not show too much difference in both partitioning schemes for these datasets. On the other hand, the obtained load imbalance for Movielens datasets are very similar to obtained results in larger datasets, however, there is a tricky point that we have to consider. When SSGD model is directly used during the load imbalance calculation without considering row block wise partitioning through a complete iteration, static partitioning differs by producing higher imbalance with Movielens datasets. However, we obtained that static and random partitioning results are very similar when only row block wise partitioning is considered as applied for the other datasets during the load imbalance calculation. It means that the load imbalance on Movielens datasets in terms of row block wise partitioning is very low, whereas it is very high in SSGD-based (block by block) wise partitioning.

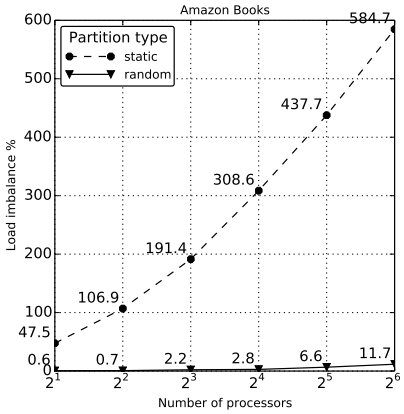
The load imbalance results are calculated as percentage and illustrated in Figure 5.1 and Figure 5.2. The results show that random permutation works well for much sparser datasets even though it is not too much effective for denser datasets including Last.fm, Netflix, Yahoo Music Track-1 and Yahoo Music Track-2. As shown in Figure 5.1, the load imbalance in Amazon datasets with static partitioning is very high, and random permutation improves the load balance by 67x-339x. On the other hand, static partitioning produces a little bit (∼15%) better results than random partitioning in terms of load balance as illustrated in Figure 5.2. However, static partitioning in Netflix, Yahoo Music Track-1 and Yahoo Music

Track-2 datasets already achieves good load balance where the maximum load imbalance is around 2% and 5% in Last.fm and Netflix datasets, respectively, whereas it is around 10% in Yahoo Music datasets. Therefore, we can not mention an improvement in such a case since the imbalance is already low and the difference between static and random partitioning in terms of load imbalance is also very low. Hence, we expect that the speedup does not show too much difference in both partitioning schemes for these datasets. On the other hand, the obtained load imbalance for Movielens datasets are very similar to obtained results in larger datasets, however, there is a tricky point that we have to consider. When SSGD model is directly used during the load imbalance calculation without considering row block wise partitioning through a complete iteration, static partitioning differs by producing higher imbalance with Movielens datasets. However, we obtained that static and random partitioning results are very similar when only row block wise partitioning is considered as applied for the other datasets during the load imbalance calculation. It means that the load imbalance on Movielens datasets in terms of row block wise partitioning is very low, whereas it is very high in SSGD-based (block by block) wise partitioning.

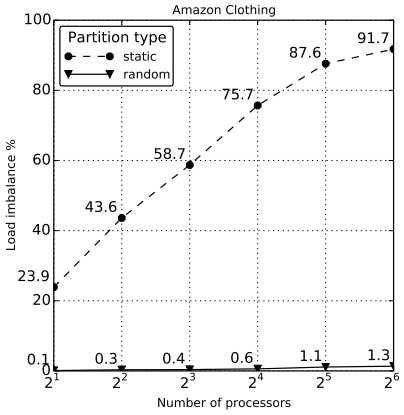
Thus, we completely ensured that the ratings are spread nonuniformly through column block wise in Movielens datasets, meaning that some of the blocks have much more entries than others in the same row block. Therefore, we expect that static partitioning works slower than random partitioning with Movielens datasets since some of the processors idle during bulk synchronization progress due to imbalance among the blocks located in the same row block stripe as explained above.



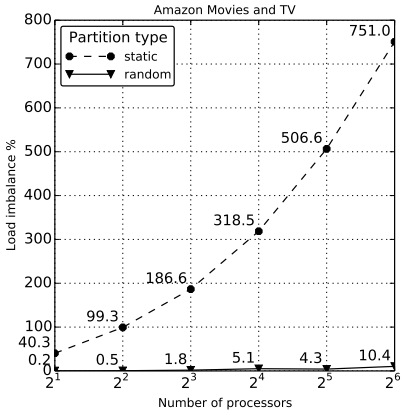
(a) Amazon Item.



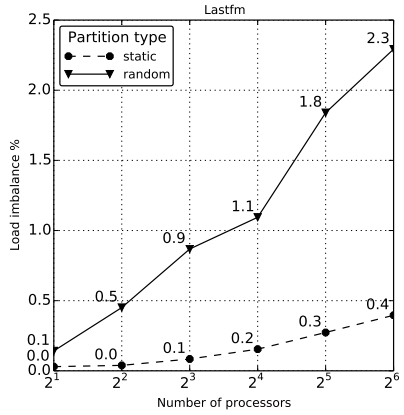
(b) Amazon Books.



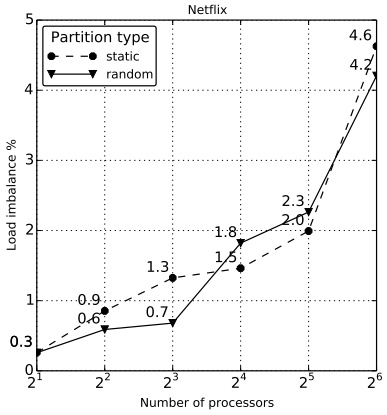
(c) Amazon Clothing.



(d) Amazon Movies and TV.

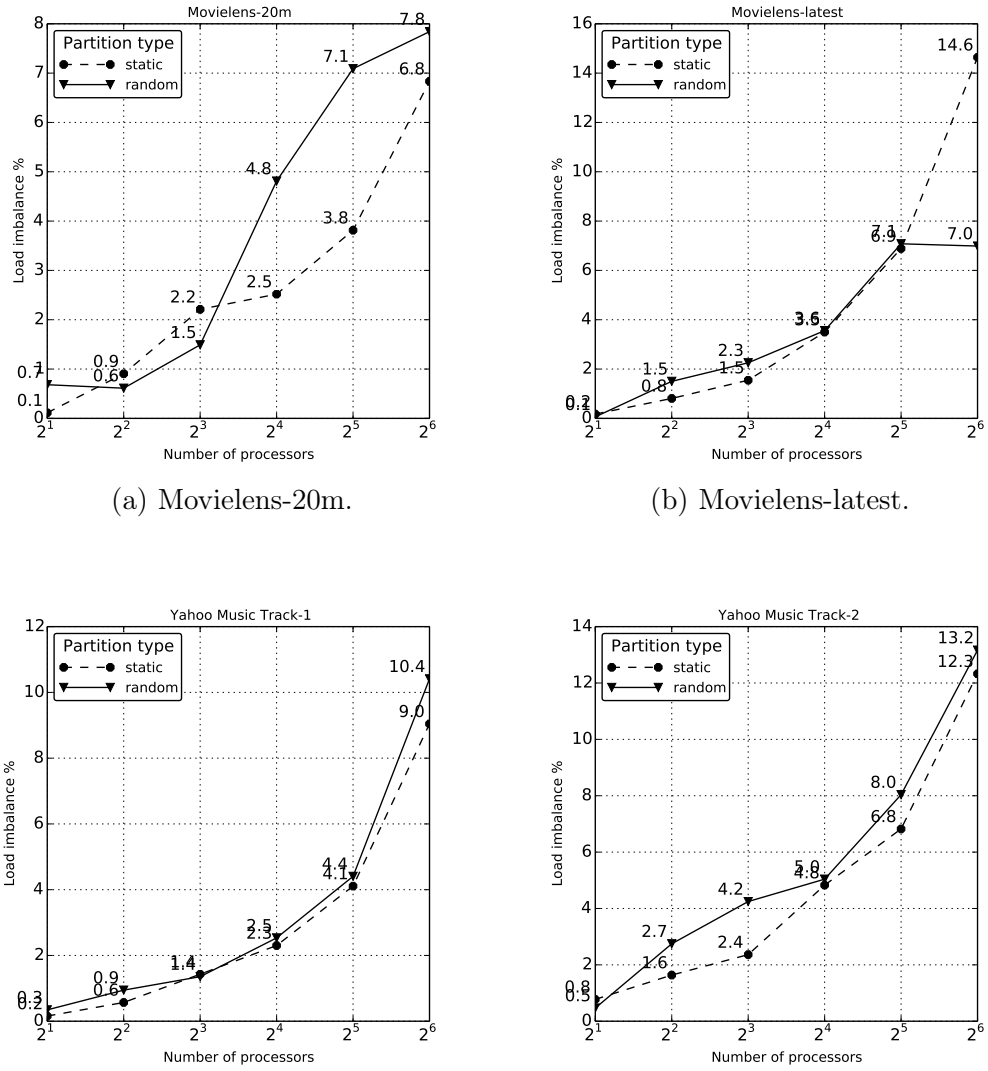


(e) Last.fm.



(f) Netflix.

Figure 5.1: Load imbalance comparison for static and random partitioning 1.



(a) Movielens-20m.

(b) Movielens-latest.

(c) Yahoo Music Track-1.

(d) Yahoo Music Track-2.

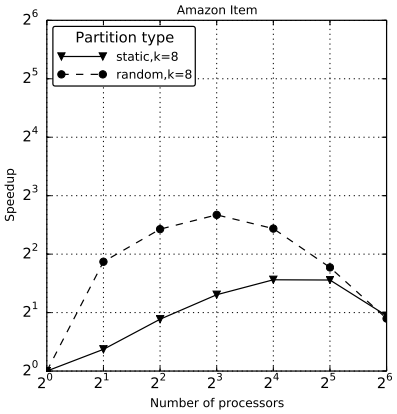
Figure 5.2: Load imbalance comparison for static and random partitioning 2.

5.3.2 Speedup

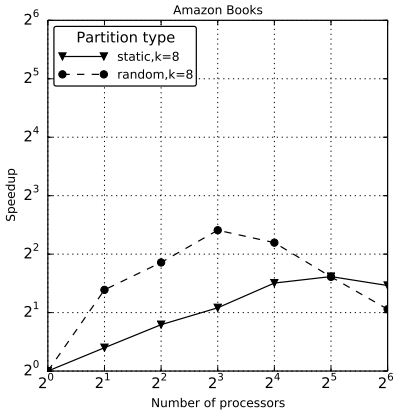
We used built-in `MPI Wtime` function to measure the running time of our DSGD implementation. The initial operations before running the DSGD algorithm including reading and distributing input data are not included in timing. We selected the maximum one among the elapsed times measured by all the processors

and calculated the speedup by comparing the elapsed times in parallel and sequential executions for each dataset. Meanwhile, our sequential SGD implementation is also executed on the same machine with the same input parameters and random seed.

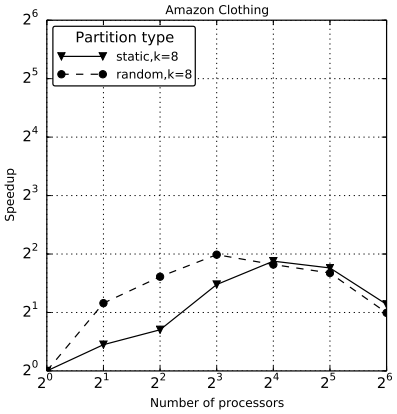
Almost linear speedup is achieved for denser datasets by the DSGD algorithm as illustrated in Figure 5.4. The respective speedup curves are almost the same for both partitioning schemes since the calculated load imbalance values for these datasets are very similar to each other as discussed in Section 5.3.1. In addition, the random partitioning scales a little bit better than the static partitioning as we expected. The speedup results for the `Movielens` datasets prove us right about characteristics of the dataset which is described while discussing the load imbalance results. Even if the load imbalance results for both partitioning types are similar in these datasets, the speedup curves show too much difference, where the random partitioning works much better than the static partitioning as we expected. Meanwhile, we obtained better speedup results by using the random partitioning also in smaller Amazon datasets, which are `Amazon Books`, `Amazon Clothing`, `Amazon Electronics` and `Amazon Movies` as shown in Figure 5.3. In these datasets, the speedup begins well as expected when the random partitioning is applied, however, it does not interestingly scale after 8 processors. We analyzed the elapsed times separately for computation and communication processes and found out that communication bottleneck occurs after 8 processors for these datasets, in which the time spent during the communication process is five times the time spent during the computation process. The same scenario occurs in `item dedup` dataset where the speedup does not scale after 8 processor, although it scales well at the beginning as we calculated as 3.8 when 2 processors are used. This abnormal situation arises from the cache, especially L3 cache, since the size of communicated data fits into L3 cache on the machine used in the experiments.



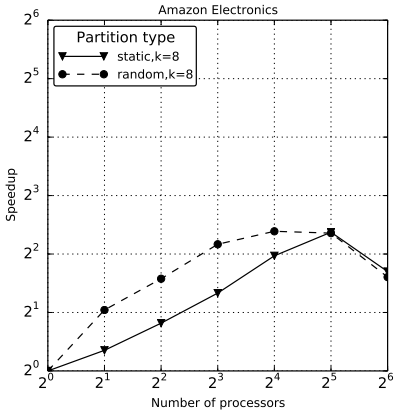
(a) Amazon Item.



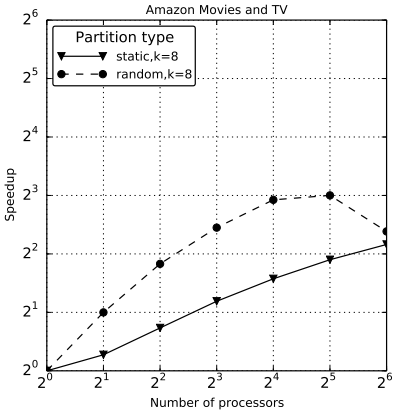
(b) Amazon Books.



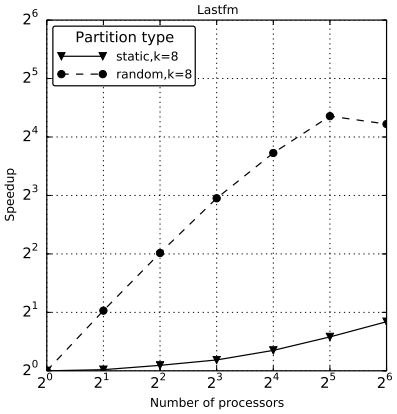
(c) Amazon Clothing.



(d) Amazon Electronics.

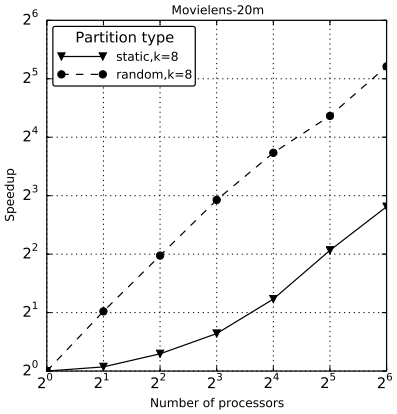


(e) Amazon Movies.

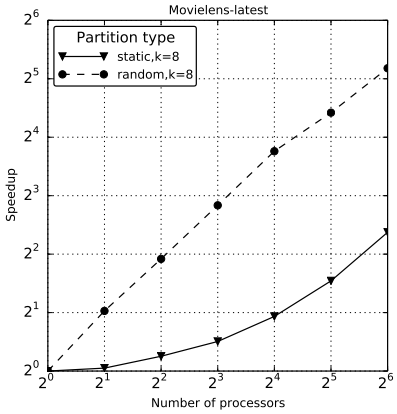


(f) Last.fm.

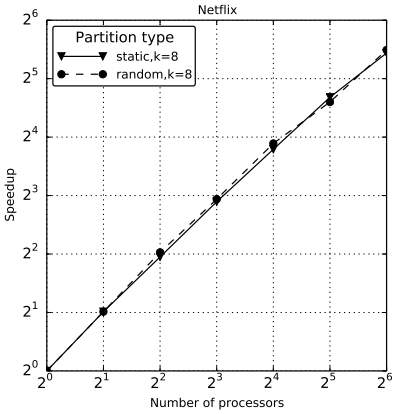
Figure 5.3: Speedup results 1.



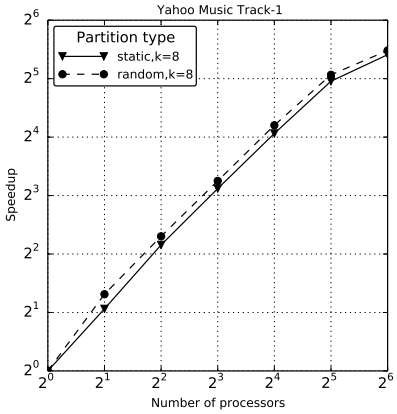
(a) Movielens-20m



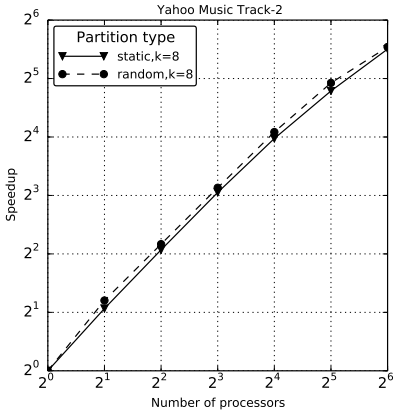
(b) Movielens-latest.



(c) Netflix.



(d) Yahoo Music Track-1.



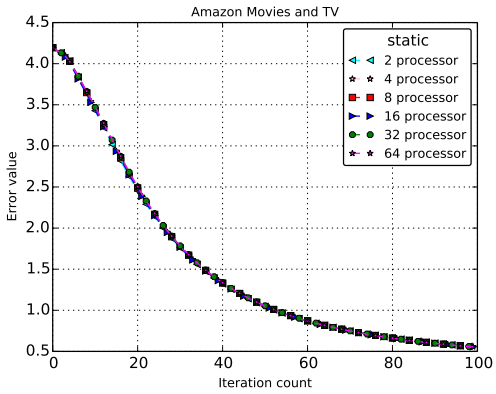
(e) Yahoo Music Track-2.

Figure 5.4: Speedup results 2.

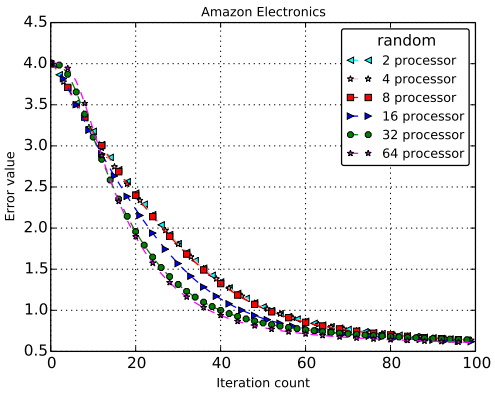
5.3.3 Convergence

The weighted loss function based on L2 regularization is used while calculating total loss in the DSGD model, where the stratum losses are reduced by master processor at the end of each subiteration. We observed that convergence for both partitioning schemes with all the datasets is achieved by our DSGD implementation. We divided convergence results into two groups, which are *per-matrix* and *per-partition*, to test the proposed statements in the DSGD study. Per-matrix results contain comparison of the convergence curves when different number of processors and different partitioning schemes are applied on the same dataset. On the other hand, per-partition results include comparison of the convergence curves when different partitioning schemes are applied on the same dataset with the same number of processor. To sum up, the per-matrix results show correctness of the stated theory regarding convergence of the SSGD model, whereas the per-partition results illustrate that random partitioning achieves faster convergence than static partitioning as stated in the paper.

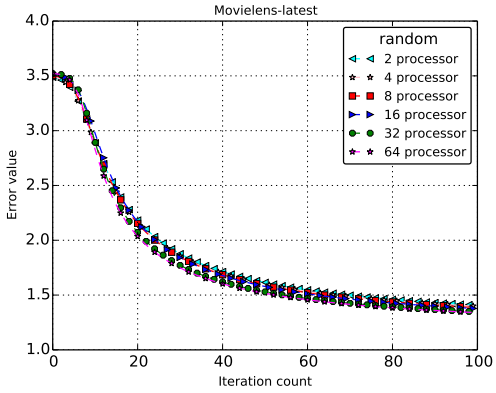
Figure 5.5 contains the per-matrix convergence results, where we picked only six figures randomly to not occupy more place here. Otherwise, we need to show more than a hundred figures in total like for each dataset, partitioning type and number of processor. We selected different datasets and partitioning types to test the proposed theory. We obtained that the convergence is achieved by the specialized SSGD model, and it is not affected by different number of processors, meaning that the number of blocks used during the stratification process does not cause any computational error. We obtained the same convergence rate on the same data without considering the applied partitioning type and the number of processors as illustrated in the subfigures. The only difference is based on the partitioning schemes in such a way that the convergence is a little bit faster with more number of processors. However, the reached convergence rates are the same.



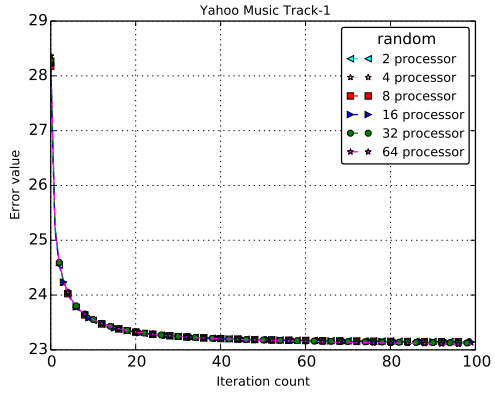
(a) Amazon Movies, Static.



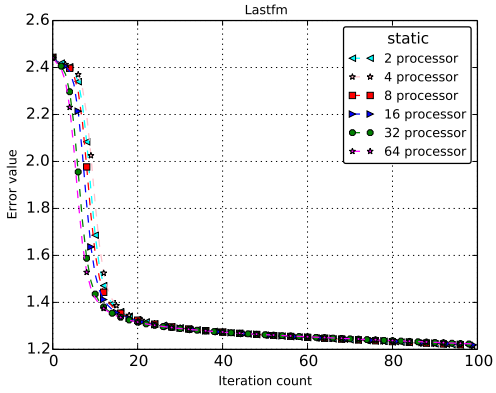
(b) Amazon Electronics, Random.



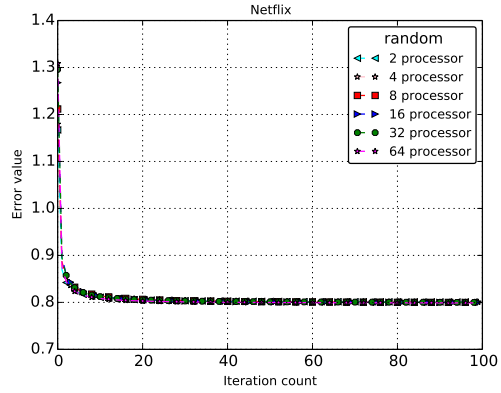
(c) Movielens-latest, Random.



(d) Yahoo Music Track-1, Random.



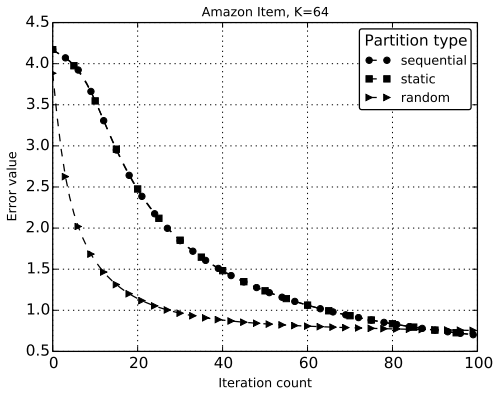
(e) Last.fm, Static.



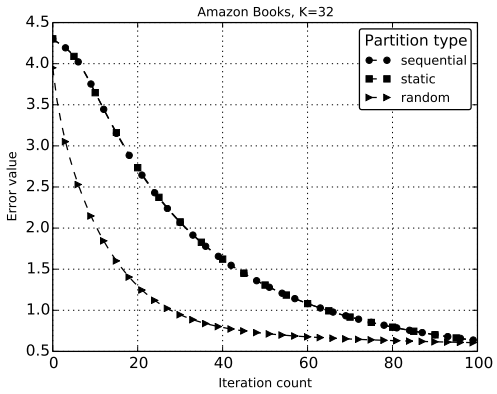
(f) Netflix, Random.

Figure 5.5: Per-matrix convergence results.

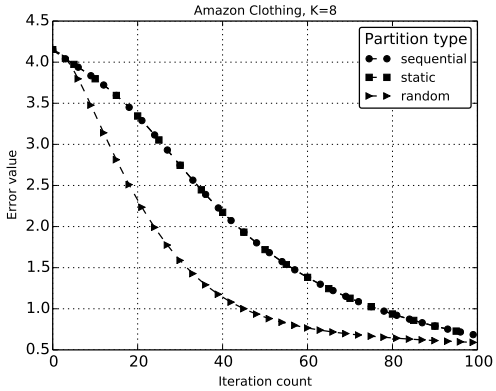
The per-partition convergence results are illustrated in Figure 5.6. The random partitioning achieves much faster convergence without considering the applied number of processors when compared with static partitioning and sequential execution. The SGD-based algorithms inherently work better with randomized data as exhaustively explained in Chapter 3. The convergence rates obtained with the static partitioning and the sequential execution are almost the same since they pick the ratings in the same sequence which shows that the SSGD model achieves the same update sequence as serial SGD algorithm. The convergence results are different for `Netflix` dataset when the static and random partitioning schemes applied with different number of the processors. The static partitioning leads to a little bit better convergence as illustrated in Figure 5.6e and Figure 5.6f. The obtained convergence rate is 0.8 with the random partitioning, whereas it is 0.65 with the static partitioning. Although these convergence rates are very close to each other, it probably converges to local minima due to the changed update orders.



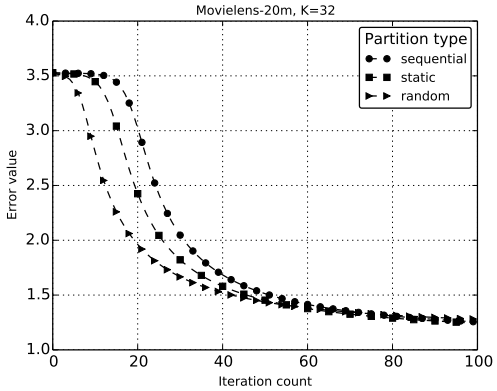
(a) Amazon Item, K=64.



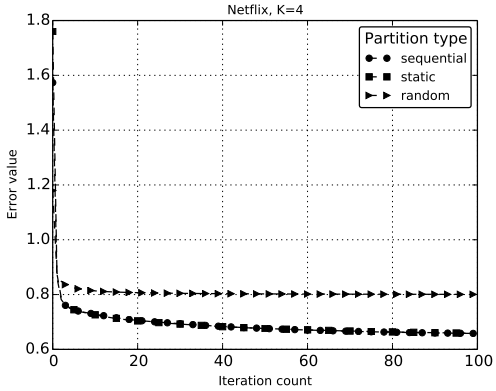
(b) Amazon Books, K=32.



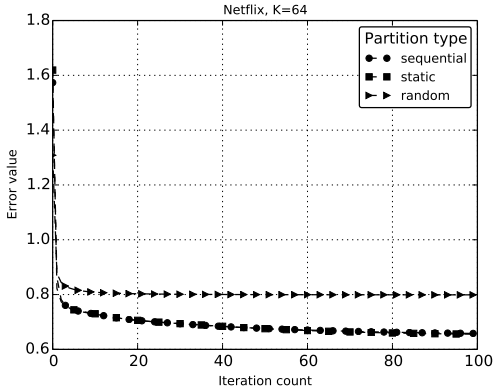
(c) Amazon Clothing, K=8.



(d) Movielens-20m, K=32.



(e) Netflix, K=4.



(f) Netflix, K=64.

Figure 5.6: Per-partition convergence results.

Chapter 6

Conclusion

In this thesis, we focused on the parallel stochastic gradient descent (SGD) models used popularly while building recommendation systems in recent years. Recommender systems are used to predict the missing preferences of the users to decide which items can be recommended to which users. There are two strategies used while building recommender systems as content filtering and collaborative filtering. Although content filtering is more robust to cold start problem, collaborative filtering achieves better predictions. Collaborative filtering find its applications in mostly latent factor models which boils down to matrix factorization, where factor matrices are used for the users and the items. The goal is to update vectors of the factor matrices by avoiding sequential nature of update procedures in the SGD algorithm while discovering the latent user and item factors. In this direction, we examined popular parallel SGD algorithms by describing the developed models and created a detailed survey consisting of these algorithms developed for shared memory and distributed memory systems. Among the existing models, distributed SGD (DSGD) [7] is the leading work since most of the existing works covered in Chapter 3 developed for both memory environments are influenced by this prominent study. Although DSGD is such an outstanding study in this research area, there is no work regarding detailed performance analysis of the DSGD algorithm as far as we researched. Hence, we implement the DSGD model using message-passing paradigm and test its performance in terms of the load

imbalance, speedup and convergence metrics. In addition, we produced many real-world datasets and conduct the experiments with these datasets in contrast to the studies described in the survey where there is only a few datasets used in the experiments. We also filled the gap in the DSGD study regarding the effect of the random permutation of the rating matrix on the load imbalance and the convergence. We implemented a load imbalance calculator and show how the random permutation increases the load balance and speed up the convergence by conducting the experiments for static and random partitioning techniques with different number of processors.

Experimental results show that the random permutation improves the load balance among the processors and speeds up the convergence of the DSGD algorithm. The specialized SSGD model converges with all the datasets of which properties are given in Table 5.1. Correctness of the described theory regarding the convergence of the specialized SSGD model in the DSGD study is also shown with the experimental results such that the developed model converges to the same rates even if different number of processors are applied in the rating matrix. In addition, the most recent updated vectors of the factor matrices are used by the processors which makes the convergence faster. Although the speedup curves for sparser datasets such as `Amazon` subsets do not scale when the number of the processors are increased due to occurred communication bottleneck, almost linear speedup is achieved with the denser datasets such as `Movielens`, `Netflix` and `Yahoo Music`. All these results show that the DSGD model is suitable to parallelize the SGD algorithm by avoiding its sequential update procedure. As a result, DSGD is a robust algorithm and does not suffer from the growing size of datasets.

Bibliography

[1] F. Niu, B. Recht, C. Re, and S. J. Wright, "Hogwild!: A lock-free approach to parallelizing stochastic gradient descent," in *NIPS*, 2011.

[2] Y. Zhou, D. Wilkinson, R. Schreiber, and R. Pan. Large-scale parallel collaborative filtering for the netflix prize. In *Algorithmic Aspects in Information and Management, pages 337–348. Springer,* 2008.

[3] F. Petroni and L. Querzoni. Gasgd: stochastic gradient descent for distributed asynchronous matrix completion via graph partitioning. *ACM Conference on Recommender systems*, 2014.

[4] 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. *arXiv preprint arXiv:1312.0193*, 2013.

[5] J. Oh, W.-S. Han, H. Yu, and X. Jiang. Fast and robust parallel sgd matrix factorization. In *KDD*, 2015.

[6] A. Kyrola, G. Blelloch, and C. Guestrin. Graphchi: Large-scale graph computation on just a pc. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 31–46. USENIX, 2012.

[7] R. Gemulla, E. Nijkamp, P. J. Haas, and Y. Sismanis. Large-scale matrix factorization with distributed stochas- tic gradient descent. In *Proceedings of the conference on Knowledge Discovery and Data Mining*, pages 69–77, 2011.

[8] C. Teflioudi, F. Makari, and R. Gemulla. Distributed matrix completion. In *Proceedings of the International Con- ference on Data Mining*, pages 655–664, 2012.

[9] J. Langford, A. J. Smola, and M. Zinkevich. Slow learners are fast. In *Advances in Neural Information Processing Systems*, 2009.

[10] Lops, P., de Gemmis, M., Semeraro, G.: Content-based recommender systems: State of the art and trends. In: F. Ricci, L. Rokach, B. Shapira, P.B. Kantor (eds.) Recommender Systems Handbook, pp. 73–105. Springer Verlag (2011)

[11] Y. Koren, R. Bell, and C. Volinsky. Matrix factorization techniques for recommender systems. *IEEE Computer*, 42(8):30–37, 2009.

[12] J. Bennett and S. Lanning. The netflix prize. In *Proceedings of KDD cup and workshop*, 2007.

[13] G. Dror, N. Koenigstein, Y. Koren, and M. Weimer. The yahoo! music dataset and kdd-cup'11. *Journal of Machine Learning Research-Proceedings Track*, 18:8–18, 2012.

[14] Y. Zhou, D. Wilkinson, R. Schreiber, and R. Pan. Large-scale parallel collaborative filtering for the Netflix prize. In *Proceedings of the Fourth International Conference on Algorithmic Aspects in Information and Management*, pages 337–348, 2008.

[15] W. Tan, L. Cao, and L. Fong. Faster and cheaper: Parallelizing large-scale matrix factorization on gpus. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, pages 219–230. ACM, 2016.

[16] I. Pilaszy, 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*, pages 71–78, 2010.

[17] http://stanford.edu/~rezab/classes/cme323/S15/notes/lec14.pdf

[18] H.-F. Yu, C.-J. Hsieh, S. Si, and I. S. Dhillon. Scalable coordinate descent approaches to parallel matrix factorization for recommender systems. In *ICDM*, 2012.

[19] P. Richtarik and M. Takac. Parallel coordinate descent methods for big data optimization, 2012.

[20] Y. Xu and W. Yin, A block coordinate descent method for multi-convex optimization with applications to nonnegative tensor factorization and completion, 2012.

[21] A. Cichocki and A.-H. Phan, "Fast local algorithms for large scale nonnegative matrix and tensor factorizations," *IEICE Transactions on Fundamentals of Electronics Communications and Computer Sciences*, vol. E92-A, no. 3, pp. 708–721, 2009.

[22] Kim, J., He, Y., Park, H.: Algorithms for nonnegative matrix and tensor factorizations: A unified view based on block coordinate descent framework. *Journal of Global Optimization (2013)*. doi:10.1007/s10898- 013-0035-4

[23] C.-J. Hsieh, K.-W. Chang, C.-J. Lin, S. S. Keerthi, and S. Sundararajan. A dual coordinate descent method for large-scale linear SVM. In ICML, 2008

[24] Kai-Wei Chang, Cho-Jui Hsieh, and Chih-Jen Lin. Coordinate descent method for large-scale L2- loss linear SVM. *Journal of Machine Learning Research*, 9:1369–1398, 2008

[25] C.-J. Hsieh and I. S. Dhillon, "Fast coordinate descent methods with variable selection for non-negative matrix factorization," in *ACM KDD*, 2011.

[26] S. Sheen, A Coordinate Descent Method for Robust Matrix Factorization and Applications,

[27] L. Bottou, "Large-scale machine learning with stochastic gradient descent," in *Proceedings of the 19th International Conference on Computational Statistics*, 2010.

[28] Sixin Zhang, Anna Choromanska, and Yann LeCun. Deep learning with Elastic Averaging SGD. *Neural Information Processing Systems Conference (NIPS 2015)*, pages 1–24, 2015.

[29] Schaul, Tom, Zhang, Sixin, and LeCun, Yann. No more pesky learning rates, 2012.

[30] Smith, S., Park, J., Karypis, G.: An exploration of optimization algorithms for high performance tensor completion. In: *Proceedings of the 2016 ACM/IEEE conference on Supercomputing* (2016)

[31] Amazon datasets, http://jmcauley.ucsd.edu/data/amazon/links.html

[32] Netflix dataset, http://academictorrents.com/

[33] Yahoo datasets, https://webscope.sandbox.yahoo.com/myrequests.php

[34] Lastfm dataset, http://www.dtic.upf.edu/ ocelma/MusicRecommendationDataset/index.html

[35] Movielens datasets, https://grouplens.org/datasets/movielens/

[36] F. Maxwell Harper and Joseph A. Konstan. 2015. The MovieLens Datasets: History and Context. *ACM Transactions on Interactive Intelligent Systems (TiiS)* 5, 4, Article 19 (December 2015).