HYBRID PARALLELIZATION OF STOCHASTIC GRADIENT DESCENT

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
Kemal Büyükkaya
February 2022

Hybrid Parallelization of Stochastic Gradient Descent
By Kemal Büyükkaya
February 2022

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)

Özcan Öztürk

Fahreddin Şükrü Torun

Approved for the Graduate School of Engineering and Science:

Ezhan Karaşan
Director of the Graduate School

ABSTRACT

HYBRID PARALLELIZATION OF STOCHASTIC GRADIENT DESCENT

Kemal Büyükkaya
M.S. in Computer Engineering
Advisor: Cevdet Aykanat
February 2022

The purpose of this study is to investigate the efficient parallelization of the Stochastic Gradient Descent (SGD) algorithm for solving the matrix completion problem on a high-performance computing (HPC) platform in distributed memory setting. We propose a hybrid parallel decentralized SGD framework with asynchronous communication between processors to show the scalability of parallel SGD up to hundreds of processors. We utilize Message Passing Interface (MPI) for inter-node communication and POSIX threads for intra-node parallelism. We tested our method by using four different real-world benchmark datasets. Experimental results show that the proposed algorithm yields up to 6× better throughput on relatively sparse datasets, and displays comparable performance to available state-of-the-art algorithms on relatively dense datasets while providing a flexible partitioning scheme and a highly scalable hybrid parallel architecture.

Keywords: Stochastic gradient descent, Matrix completion, Matrix factorization, Parallel distributed memory system.

ÖZET

OLASILIKSAL GRADYAN ALÇALMANIN HİBRİT PARALELLEŞTİRİLMESİ

Kemal Büyükkaya
Bilgisayar Mühendisliği, Yüksek Lisans
Tez Danışmanı: Cevdet Aykanat
Şubat 2022

Bu çalışmanın amacı, Olasılıklsal Gradyan Alçalma (SGD) algoritmasının dağınık bellekli yüksek başarımlı hesaplama platformlarında verimli bir şekilde paralelleştirmelerini araştırmaktır. Paralel SGD'nin yüzlerce işlemciye kadar ölçeklenebilirliğini göstermek için işlemciler arasında asenkron iletişim kurabilen, hibrit mimarye sahip ve merkezsizleştirilmiş bir SGD algoritması öneriyoruz. İş birimleri arası iletişim için Message Passing Interface'i (MPI) ve iş birimleri içersindeki paralellik içinse POSIX iş parçacıklarını kullanıyoruz. Dört farklı kıyaslama veri seti kullanarak yöntemimizi test ettik. Deneysel sonuçlar, önerilen algoritmanın görece seyrek veri setleri üzerinde 6 kata kadar daha fazla verim elde ettiğini ve hem esnek bir bölümleme şeması hemde yüksek düzeyde ölçeklenebilir hibrit bir mimari sağlarken görece yoğun veri setleri üzerinde de mevcut en gelişmiş algoritmalarla karşılaştırılabilir sonuçlar verdiğini göstermektedir.

Anahtar sözcükler: Olasılıksal gradyan alçalma, Matris tamamlama, Matris ayrışımı, Paralel dağınık bellekli sistemler.

Acknowledgement

First of all, I would like to thank my advisor Prof. Dr. Cevdet Aykanat for his support, guidance, and suggestions. It was a great honor for me to study under his supervision. I would like to thank Prof. Dr. Özcan Öztürk and Asst. Prof. Fahreddin Şükrü Torun for kindly accepting to be in my thesis committee, reading my thesis, and providing valuable comments. I would also like to thank Mustafa Ozan Karsavuran for sharing his technical knowledge throughout this research.

I would like to express my deepest gratitude to my family for always being by my side. Their endless love and invaluable support is my greatest motivation.

I owe special thanks to my beloved girlfriend, Buse Aktepe, for always supporting me and always being there for me.

Lastly, I would like to thank the Scientific and Technological Research Council of Turkey (TÜBİTAK) 1001 program for supporting me throughout my M.S. studies in the EEEAG-119E035 project.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Recommendation systems play a fundamental role in the success of e-commerce companies such as Amazon [1] and Netflix [2]. Such companies offer a huge number of products for their customers in different categories to meet various interests. However, many customers are overwhelmed with the numerous selection of products offered by e-commerce websites. Recommendation systems help customers to find products that suit their needs. Therefore, many e-commerce companies use recommendation systems to provide a personalized online store experience so that they can maximize customer satisfaction.

Recommendation systems rely on input data called a rating matrix, where one dimension represents a set of users, and the other dimension represents a set of products (e.g., books, movies, songs, etc.), whose nonzero values denote preferences expressed by users on products and zero values denote missing preferences. Since users indicate their preferences for a limited number of products, this rating matrices are quite sparse. Essentially, a recommendation system aims to estimate all the missing values in the given rating matrix. In other words, it intends to solve the matrix completion problem.

The prior research in this field showed that collaborative filtering approaches based on latent factors are highly useful to solve the matrix completion problem [3], [4], [5]. These approaches try to map users and items to a latent factor space and define the affinity between a user and an item as the inner product of their latent factor vectors. A latent factor, albeit not directly definable, contains useful information on user and product similarity. Hence, it can be used to estimate missing values in the rating matrix.

There are different prominent algorithms to achieve matrix completion in the literature, including stochastic gradient descent (SGD) [6], alternating least squares (ALS) [7], and cyclic coordinate descent (CCD) [8]. However, SGD is considered superior to others since it can achieve high completion accuracy while scaling to large-scale rating matrices. For this reason, we focus our research on the efficient parallelization of the SGD algorithm for matrix completion on a high performance computing (HPC) platform in distributed memory setting.

We propose a new distributed (i.e., shared-nothing) SGD algorithm with non-blocking communication between processors and fully asynchronous computation in individual processors. We introduce a flexible partitioning scheme and a three-layered hybrid parallel architecture to exploit the full potential of modern HPC platforms by utilizing thread-level parallelism. Finally, we compare our algorithm with an available state-of-the-art algorithm using one small, two medium, and one large real-world benchmark dataset. Experimental results show that the proposed framework yields up to 6× better throughput on relatively sparse datasets and displays comparable performance to available state-of-the-art algorithms on relatively dense datasets.

The rest of this thesis is organized as follows: Chapter 2 formally defines the matrix completion problem with the necessary notations and gives background information about the SGD algorithm. Chapter 3 presents an analysis of the related work. Chapter 4 describes Hybrid Parallel Stochastic Gradient (HPSGD) algorithm in detail. The experimental setup, datasets, and results are discussed in Chapter 5. Lastly, Chapter 6 concludes the thesis with possible future work and final remarks.

Chapter 2

Background

In this chapter, we formally define the matrix completion problem, establish the related notations, and give detailed information about the SGD algorithm.

2.1 The Matrix Completion Problem

The input is a sparse rating matrix $R \in \mathbb{R}^{m \times n}$, where m denotes the number of users and n denotes the number of products. Let $\Omega \subseteq \{1, \ldots, m\} \times \{1, \ldots, n\}$ denote the set of observed (i.e., nonzero) entries in R, that is $(i, j) \in \Omega$ implies that user i rated product j with rating r_{ij}. Matrix completion problem is defined as the task of predicting missing entries using observed entries in Ω.

To predict missing entries, popular collaborative filtering approaches based on latent factors apply a method called low-rank matrix factorization [6], [9], [10]. This method aims to find two low-rank matrices $W \in \mathbb{R}^{m \times f}$ and $H \in \mathbb{R}^{n \times f}$, with $f \ll \min(m, n)$, such that $R \approx WH^T$.

The main idea behind low-rank matrix factorization is that a row $w_i \in \mathbb{R}^f$ of W can be considered as the low-rank embedding of user i in f-dimensional latent factor space. Similarly, a row $h_j \in \mathbb{R}^f$ of H can be considered as the low-rank

embedding of product j in f-dimensional latent factor space. Hence, any rating in the rating matrix can be predicted as:

$$\hat{r}_{ij} = w_i h_j^T. \tag{2.1}$$

Figure 2.1 shows the relationship between a user embedding w_i and a product embedding h_j in detail.

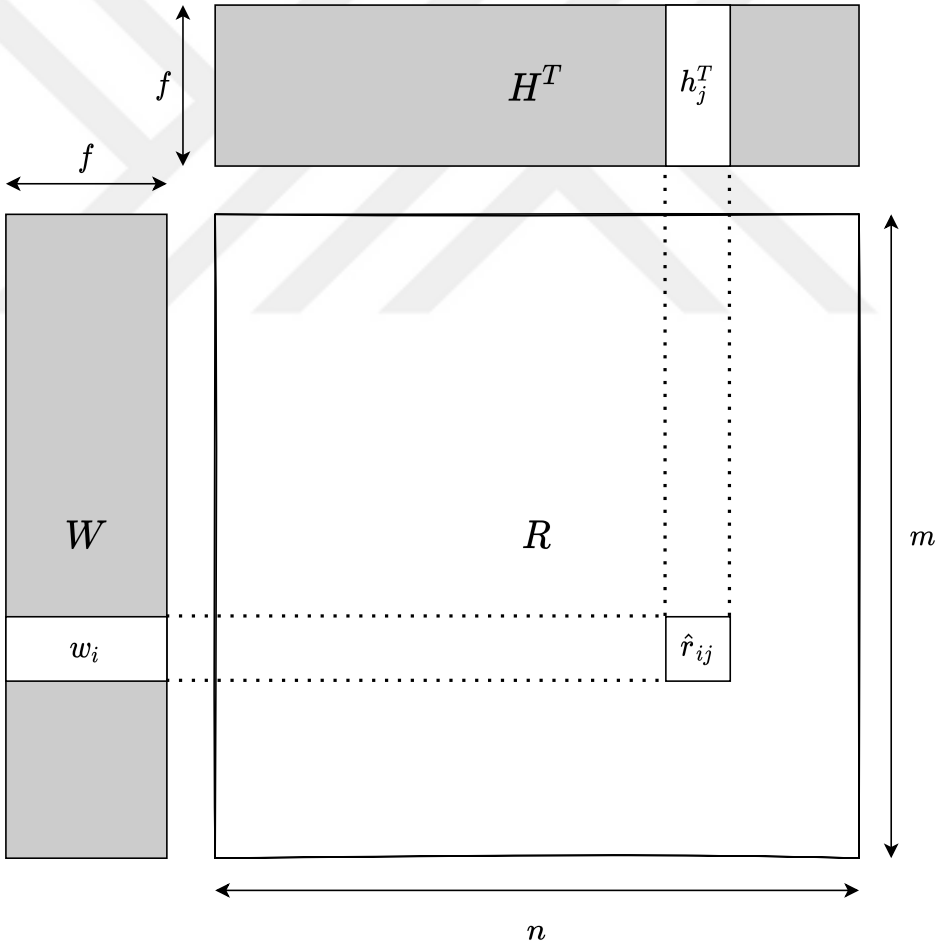


Figure 2.1: Rating matrix $R \in \mathbb{R}^{m \times n}$ and latent factor matrices $W \in \mathbb{R}^{m \times f}$ and $H^T \in \mathbb{R}^{f \times n}$.

The quality of the low-rank approximation is determined by an application dependent objective function $L(W, H)$, which measures the goodness of fit of the model. For collaborative filtering approaches based on latent factors, the

objective function $L(W, H)$ is mostly defined as the regularized squared loss [2],[6], [11]. That is,

$$L(W, H) = \underset{W,\, H}{\operatorname{argmin}} \sum_{(i,j)\in\Omega} \{(r_{ij} - \hat{r}_{ij})^2 + \lambda(||w_i||^2 + ||h_j||^2)\} \tag{2.2}$$

where $\lambda > 0$ is a regularization coefficient added to prevent over-fitting, $||.||^2$ is the L_2 norm of a vector, and $\hat{r}_{ij}$ can be calculated with (2.1).

2.2 Stochastic Gradient Descent (SGD)

The objective function (2.2) is a multivariate differentiable function, and there are different optimization algorithms to minimize multivariate differentiable functions such as stochastic gradient descent (SGD) [6], alternating least squares (ALS) [7], and cyclic coordinate descent (CCD) [8]. However, it is shown that SGD can achieve high completion accuracy while scaling to large scale rating matrices [10]. Thus, we only focus on the parallelization of the SGD algorithm.

SGD is an iterative algorithm and updates latent factor matrices W and H, at each step, with values proportional to the gradient of the objective function. For the matrix completion problem, the gradient of the objective function (2.2) at a fixed point r_{ij} can be defined as:

$$\bigtriangledown_{w_i} L(W, H) = (r_{ij} - \hat{r}_{ij})h_j + \lambda w_i, \tag{2.3}$$

$$\bigtriangledown_{h_j} L(W, H) = (r_{ij} - \hat{r}_{ij})w_i + \lambda h_j. \tag{2.4}$$

The key difference between the SGD algorithm and the above-mentioned optimization algorithms is that each SGD update only requires a single uniformly sampled random nonzero entry r_{ij} of the observed entries Ω.

Therefore, the corresponding update rules for w_i and h_j rows can be written as:

$$w_i \leftarrow w_i - \gamma(r_{ij} - \hat{r}_{ij})h_j + \lambda w_i, \tag{2.5}$$

$$h_j \leftarrow h_j - \gamma(r_{ij} - \hat{r}_{ij})w_i + \lambda h_j. \tag{2.6}$$

where γ is the step size, which can be a predefined constant value or can be dynamically managed during the execution. The algorithm continues to perform several iterations through the set of observed entries Ω until a convergence criterion is satisfied. A single iteration over the rating matrix R corresponds to an epoch.

Chapter 3

Related Work

In this chapter, we review the parallel SGD algorithms in the literature, discuss the methodology that they stand for, and mention their major drawbacks.

3.1 Single-machine Parallel SGD Algorithms

This subsection focuses on studies that solve the matrix completion problem in multi-core systems with shared memory.

3.1.1 HogWild

The SGD is an iterative algorithm, and its sequential implementations do not scale for large-sized rating matrices. Parallel SGD algorithms aim to overcome this scalability problem by distributing the computation among multiple processing units. A naive method to handle concurrent updates of shared latent factor matrices W and H is to lock, before sampling a random nonzero entry $(i, j) \in \Omega$, both row w_i and column h_j^T [12]. Nevertheless, lock-based methods put a limit on the scalability of the algorithm since they come with a memory locking overhead

which adversely affects the concurrency.

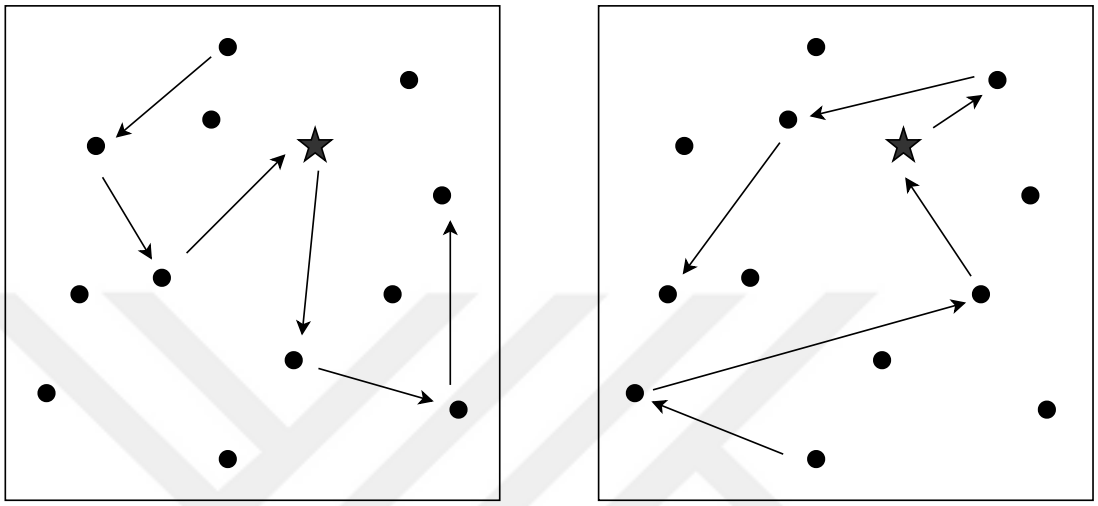


Figure 3.1: An example update sequence for two threads in HogWild. Black dots are randomly sampled nonzero entries. Arrows denote the order of the update sequence. The star indicates the potential data hazard since it is simultaneously accessed by two threads in their 4^{th} iterations.

Hogwild algorithm of Recht et al. [13] introduces an asynchronous lock-free algorithm assuming that the updates given in (2.5) and (2.6) are likely to be independent since the rating matrix R is quite sparse. In other words, randomly sampled nonzero entries to be updated are unlikely to share the same user i and the same product j, where $(i, j) \in \Omega$. Hence, the updates given in (2.5) and (2.6) can be executed via different threads in parallel. Even though possible data hazards may occur during the execution (see Figure 3.1), HogWild proves that convergence of the algorithm is inevitable if the rating matrix is sufficiently sparse. However, HogWild is a non-serializable algorithm which means there is no equivalent update sequence in a serial implementation; therefore, it does not guarantee faster convergence.

3.1.2 FPSGD

Another popular strategy to distribute the computation among multiple processing units is to ensure that processing units process on regions of the rating matrix

R such that randomly sampled nonzero entries to be updated do not share the same user i or the same product j, where $(i, j) \in \Omega$.

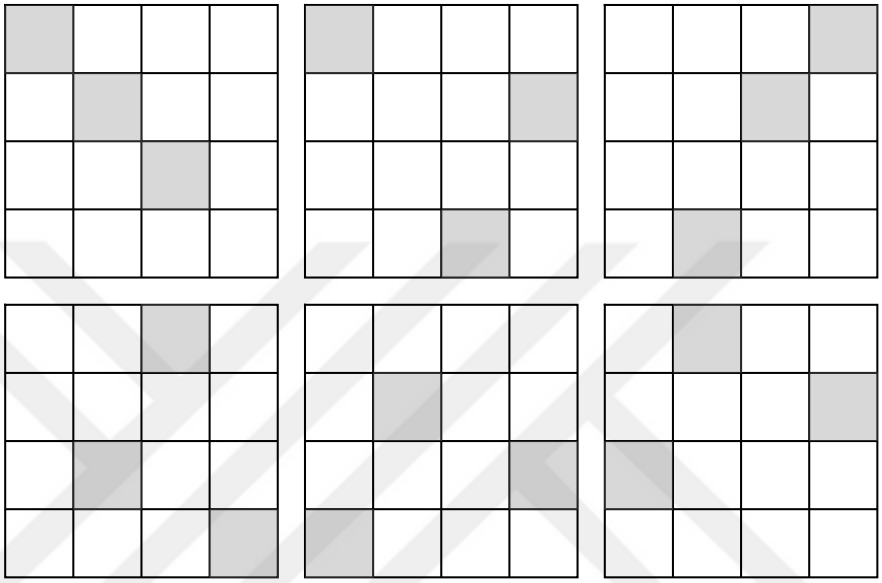


Figure 3.2: Examples of mutually independent block assignment in FPSGD.

Fast Parallel Stochastic Gradient (FPSGD) algorithm of Zhuang et al. [11] partitions the rating matrix R into $k' \times k'$ number of blocks with $k' > k$, where k is the number of worker threads; and uses a task manager thread to distribute these blocks among the worker threads in such a way that they share neither any row i nor any column j of the rating matrix R. Figure 3.2 shows some examples of mutually independent block assignment, where $k' = 4$ and $k = 3$, in FPSGD. When a worker thread finishes processing on its own block, it requests a new block from the task manager thread. Nevertheless, FPSGD shifts the overall complexity of the model to the task manager thread, and it is not straightforward to extend this strategy to multi-machine systems with distributed memory.

3.2 Multi-machine Parallel SGD Algorithms

This subsection focuses on studies that solve the matrix completion problem in multi-machine systems with distributed memory.

3.2.1 DSGD

By its nature, the SGD is an iterative algorithm, and the simplest strategy for distributing iterative algorithms among multiple machines is to apply a bulk synchronization at the end of each iteration to resynchronize updated local copies of the shared data across multiple machines.

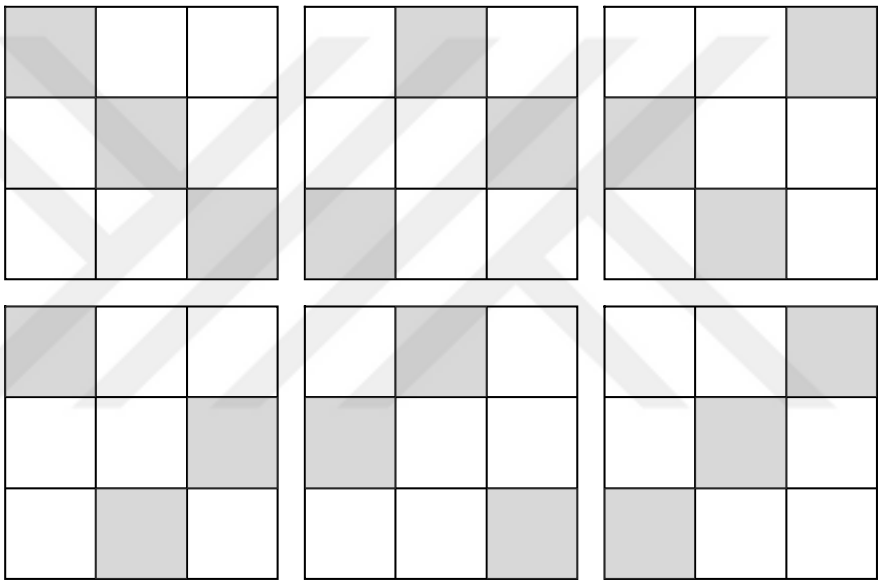


Figure 3.3: All possible mutually independent block assignments for a 3×3 matrix in DSGD. The first and second rows of the figure form two different strata.

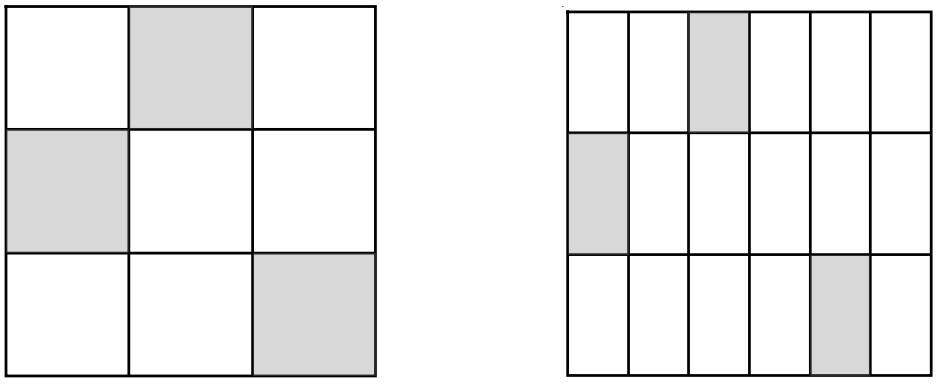
The Distributed Stochastic Gradient Descent (DSGD) algorithm of Gemulla et al. [6] splits the rating matrix R into $k \times k$ number of blocks, where k is the number of machines. Then, the algorithm distributes k mutually independent blocks to each machine. Two blocks are said to be mutually independent if they do not share any row and column. DSGD defines this special type of block assignment scheme as stratum. In other words, mutually independent blocks of a stratum does not share the same row w_i of the latent factor matrices W and the same column h_j^T of the latent factor matrix H. For example, Figure 3.3 indicates 6 different stratum possible for a 3×3 matrix. In DSGD, each stratum is executed in parallel within a single sub-epoch. A collection of stratum forms a strata if they cover the entire rating matrix when each strata is executed within different sub-epochs. At the end of each sub-epoch, DSGD carries out a bulk synchronization

step to synchronize all the updated rows of the latent factor matrix H.

The main disadvantage of approaches that rely on bulk synchronous processing is that the communication and the computation steps do not overlap since they are performed sequentially. Therefore, when the CPUs are busy, the network becomes idle. Likewise, when the network is busy, the CPUs become idle.

3.2.2 DSGD++

In order to keep both the CPUs and the network busy together, the DSGD++ algorithm of Teflioudi et al. [9] divides the rating matrix R into $k \times 2k$ number of blocks (see Figure 3.4), where k is the number of machines. Then, while k machines are processing k mutually independent blocks, the algorithm ensures that all updated rows of the latent factor matrix H corresponding to the other k mutually independent blocks are sent through the network. Unfortunately, in this strategy, all machines have to wait till the slowest machine gets the job done to proceed to the next step. This is also known as the curse of the last reducer problem [14].

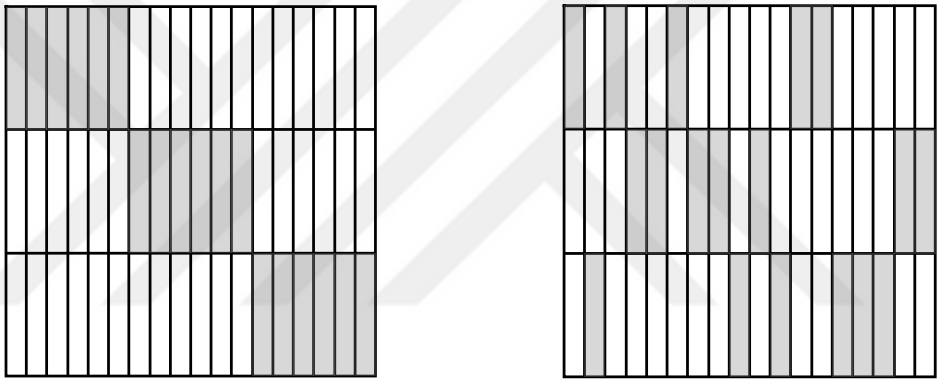


(a) DSGD (b) DSGD++

Figure 3.4: A comparison of data partitioning schemes between DSGD and DSGD++.

3.2.3 NOMAD

To overcome the curse of the last reducer problem, the NOMAD algorithm of Yun et al. [10] follows the fine-grained partitioning approach and divides the rating matrix R into $k \times n$ number of blocks where k is the number of machines and n is the number columns (i.e., products) of the rating matrix R. Then, the algorithm assigns these blocks to each machine in a mutually independent way and keeps track of the ownership of each block during the execution (see Figure 3.5).



(a) Initial (b) During the execution

Figure 3.5: An illustration of block assignments of NOMAD.

Nevertheless, this level of fine-grained partitioning moves the load balancing problem from the partitioning phase to the execution phase, and it requires additional procedures to maintain a balanced workload among the machines during the execution.

Chapter 4

Hybrid Parallel Stochastic Gradient Descent (HPSGD)

In this chapter, we propose a novel flexible partitioning scheme, discuss the details of the proposed architecture, and give the complexity analysis of the HPSGD algorithm.

4.1 Flexible Partitioning Scheme

The HPSGD algorithm partitions the rating matrix R into $k \times c$ number of blocks with $2k \leq c < n$, where c is a hyperparameter, and k is the number of machines such that each block has approximately the same number of nonzero entries. In the meanwhile, the algorithm also splits the latent factor matrix W into k number of blocks, and the latent factor matrix H into c number of blocks. Then, the algorithm assigns each row slice of the rating matrix R and its the corresponding row slice of latent factor matrix W to a single machine, and guarantees that they are never moved during the execution of the algorithm. Lastly, the algorithm distributes each column slice of the latent factor matrix H^T to machines in such a way that each machine holds the ownership of c/k number

of different column slices of latent factor matrix H^T. Figure 4.1 indicates the initial mutually independent block assignment in HPSGD with $k = 3$ and $c = 12$.

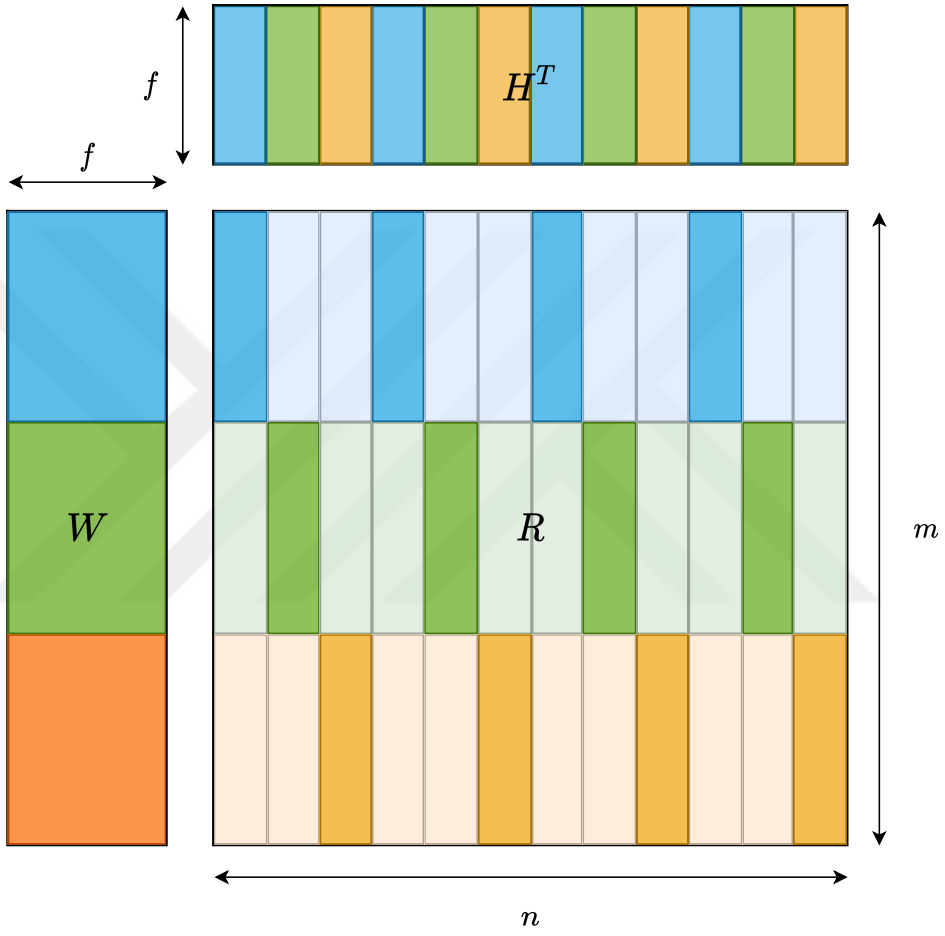


Figure 4.1: Initial of mutually independent block assignment in HPSGD.

The hyperparameter c enables us to find a unique partitioning for any rating matrix R that we can perfectly overlap the communication and the computation during the execution.

4.2 Three-Layered Hybrid Architecture

In HPSGD, we utilized multi-threaded MPI for inter-machine communication and POSIX threads for intra-machine parallelism. This is the reason why our

proposed architecture is called hybrid parallel. The architecture consists of three different layers such that each machine Q has S different thread groups. Each thread group S owns P different updater thread and a single communicator thread (see Figure 4.2).

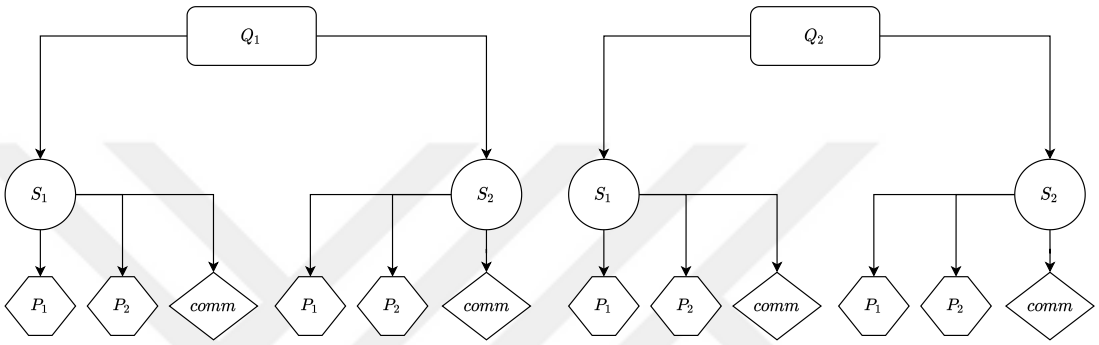
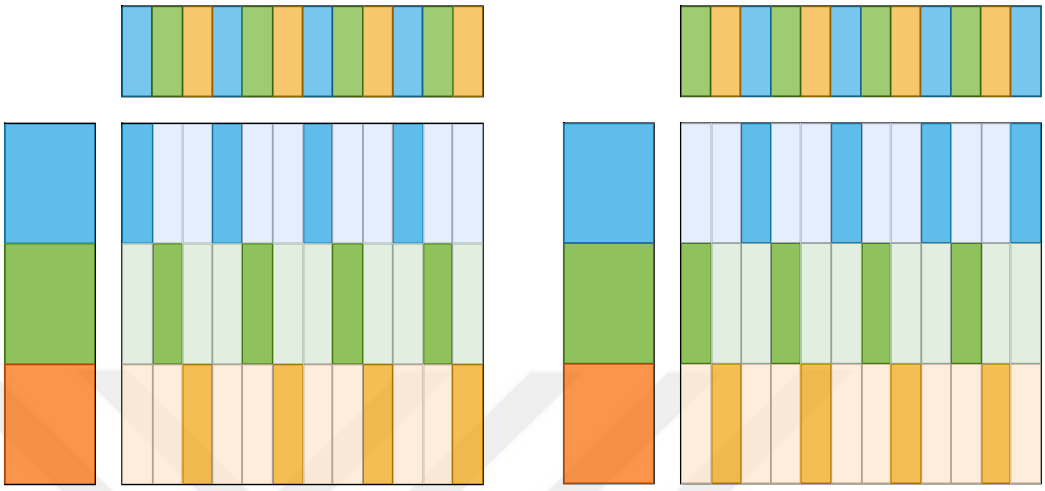


Figure 4.2: An illustration of the HPSGD architecture where the number of MPI-processes, the number of MPI-threads per MPI-process, and the number of updater threads per MPI-thread are all equal to 2.

Each thread group S maintains two different concurrent queues; the first one contains indices of the column slices to be updated, and the second one contains the indices of the column slices to be sent. During the execution of the algorithm, each updater thread of the thread group S_t of the machine Q_p simultaneously pop an index from the first queue (multi-reader and single-writer), perform the related SGD updates given in 2.5 and 2.6 for each nonzero entry in the corresponding column slice. Then, they push these indices into the second queue (multi-writer and single-reader). In parallel, the communicator thread of the thread group S_t of the machine Q_p pops an index from the second queue and sends the corresponding columns of the latent factor matrix H to the communicator thread of the thread group S_t of the machine $Q_{(p+1) \bmod k}$ (see Algorithms 1, 2 and 3). Since this is an asynchronous and non-blocking communication, the computation and the communication overlap with each other as long as both queues of an thread group S_t are not empty. Figure 4.3 shows how the ownership of the column slices of the latent factor matrix H^T changes after all machines performed related SGD updates for their initially assigned blocks and sent the updated column slices of the latent factor matrix H^T to another machine.



(a) Initial (b) During the execution

Figure 4.3: An illustration of the HPSGD algorithm.

4.3 Complexity Analysis

In the following subsections, it is assumed that problem is distributed among the k different machines.

4.3.1 Space Complexity

Each machine has to store the $1/k$ portion of the latent factor matrix W with size $m \times f$ and the $1/k$ portion of the rating matrix R that contains $|\Omega|$ nonzero entries. Each machine also stores c/k number of column slices of the latent factor matrix H^T each size of $(n \times f)/c$ (see Figure 4.4). Hence, the total space complexity of a single machine can be written as:

$$O(\frac{(m \times f) + |\Omega| + (n \times f)}{k}). \tag{4.1}$$

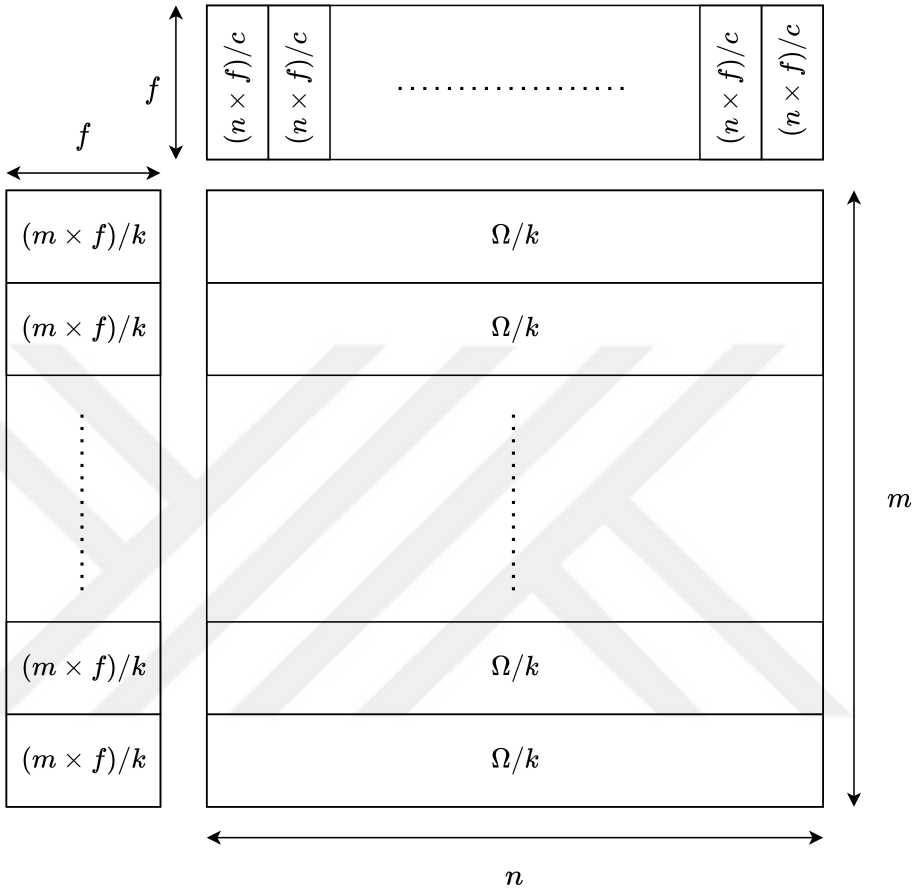


Figure 4.4: Space complexity of each row and column slice.

4.3.2 Total Communication Volume

Since the rating matrix R is partitioned row-wise such that the corresponding updates of each row slice are performed by a single machine, the communication is restricted to the column slices of the latent factor matrix H^T. Each column slice of the latent factor matrix H^T has the size of $(n \times f)/c$. There are c different column slices and each of them has to be processed by k machines to complete an epoch. Therefore, the total communication volume for one epoch can be calculated as $O(n \times f \times k)$.

4.4 Discussion

Even though DSGD, DSGD++, NOMAD, and HPSGD apply a different partitioning approach, there is no difference in terms of total communication volume. They all cost the same. DSGD and DSGD++ are lock-free algorithms; however, they are not fully asynchronous. On the other hand, NOMAD is a lock-free and fully asynchronous algorithm; nevertheless, it suffers from the side-effects of the fine-grained partitioning. Therefore, it performs additional procedures during the execution of the algorithm to minimize the negative effects of fine-grained partitioning. We emphasize that HPSGD is not only a lock-free and fully asynchronous algorithm but also provides a flexible partitioning scheme to find a load-balanced partitioning for any rating matrix, and a three-layered hybrid architecture to employ the full potential of a modern HPC platform.

Algorithm 1 The overall HPSGD algorithm.

$\boldsymbol{R}$: the rating matrix ,
$\boldsymbol{W_0}$, **$\boldsymbol{H_0}$**: the initial latent factor matrices
$\boldsymbol{K}$: the number of machines,
$\boldsymbol{C}$: the number of column slices
$\boldsymbol{S}$: the number of thread groups
$\boldsymbol{P}$: the number of updater threads per thread group
▷ Initialize the latent factor matrices
$W \leftarrow W_0$
$H \leftarrow H_0$
▷ Initial independent block assignment
Divide R / W / H into $K \times C$ / K / K number of blocks
Assign each row slice R_p of R and each row slice W_p of W to the machine Q_p with $0 \leq p < K$
for $j \in \{0, \ldots, C-1\}$ **do**
    $p = j \mod K$
    $t = (j/K) \mod S$
    $Q_p.S_t$.queue-to-be-updated.push(H_j^T)
end for
▷ Launch updater and communicator threads
for each thread group S of a machine Q **do**
    Launch P different updater threads
    Launch single communicator thread
end for
Wait until all column slices of H^T is processed $K \times epochs$ times

Algorithm 2 The updater thread of HPSGD.

▷ An updater thread of the thread group S_t of the machine Q_p

while True **do**
  if $Q_p.S_t$.queue-to-be-updated **not** empty **then**
    $H_j^T = Q_p.S_t$.queue-to-be-updated.pop()
    Perform SGD updates (2.5) and (2.6) **for all** nonzeros in $R_{p,j}$
    $Q_p.S_t$.queue-to-be-sent.push(H_j^T)
  end if
end while

Algorithm 3 The communicator thread of HPSGD.

▷ The communicator thread of the thread group S_t of the machine Q_p

while True **do**
  if $Q_p.S_t$.queue-to-be-sent **not** empty **then**
    $H_j^T = Q_p.S_t$.queue-to-be-sent.pop()
    Send H_j^T **to** $Q_{(p+1) \bmod K}.S_t$
  end if
  Receive $H_{j'}^T$ **from** $Q_{(p-1+K) \bmod K}.S_t$
  $Q_p.S_t$.queue-to-be-updated.push($H_{j'}^T$)
end while

Chapter 5

Experiments and Results

In this chapter, we examine the performance of the HPSGD algorithm with various experiments.

5.1 Datasets

In our experiments, we used four benchmark datasets: Movielens [15], Netflix [4], Yahoo-medium [3], and Yahoo-large [3].

Dataset	rows(m)	columns(n)	nonzeros($\|\Omega\|$)	density(d)
Movielens	138,493	26,744	20,000,263	5.3 e-03
Netflix	480,189	17,770	100,480,507	1.1 e-02
Yahoo-medium	249,012	296,111	61,944,406	8.4 e-04
Yahoo-large	1,000,990	624,961	252,800,275	4.0 e-04

Table 5.1: Statistics for each dataset.

Table 5.1 indicates the number of rows, columns, nonzeros, and the density for each dataset, where the density d is defined as:

$$d = \frac{|\Omega|}{m \times n} \tag{5.1}$$

5.2 Experimental Setup

We partitioned nonzero entries of each dataset into two sets as 90% for training and 10% for testing. Ω^{train} and Ω^{test} represent the nonzero entries in the training set and the test set, respectively. We used the same training and test partition for all experiments. The initial values of the latent factor matrix W and H are determined by sampling independent random values from an uniform distribution in the range $(-1/\sqrt{f},\ +1/\sqrt{f})$ [16].

To evaluate the matrix completion accuracy of the algorithms, we used Root Mean Square Error (RMSE) as a comparison metric. That is,

$$RMSE = \sqrt{\frac{\sum_{(i,j)\in\Omega^{test}}(r_{ij} - \hat{r}_{ij})^2}{|\Omega^{test}|}}. \tag{5.2}$$

In HPSGD, we used a time-based decay function to schedule the step size γ during the execution and it is defined as:

$$\gamma = \frac{\gamma_i}{(1 + \beta t)}, \tag{5.3}$$

where γ_i is the initial step size, β is the decay rate and t denotes the number of SGD updates that were performed using a nonzero entry $(i, j) \in \Omega^{train}$.

We conducted our experiments on a distributed-memory high-performance computing platform. Each node of the platform has equipped with 2 AMD EPYC

7742 (Zen 2) processors and 256GB of RAM.

Lastly, Table 5.2 lists the parameters used in the experiments for each dataset.

Dataset	Movielens	Netflix	Yahoo-medium	Yahoo-large
$\lvert\Omega^{train}\rvert$	18,000,236	90,432,456	55,749,965	227,520,247
$\lvert\Omega^{test}\rvert$	2,000,027	10,048,051	6,194,441	25,280,028
c	105	280	1157	610
f	50	50	50	50
γ_i	0.015	0.015	0.00075	0.00075
β	0.03	0.03	0.65	0.65
λ	0.05	0.05	1.0	1.0

Table 5.2: Parameters for each dataset.

5.3 Results

In the following subsections, we report the outcomes of our experiments. For all experiments, we fixed the number of MPI threads per machine to 1 and the number of updater threads per MPI-thread to 4 in HPSGD. Similarly, we also fixed the number of updater threads per machine to 4 in NOMAD.

5.3.1 Matrix Completion Accuracy

To compare the matrix completion accuracy of HPSGD and NOMAD, we plotted the decrease in the training and test RMSE (see Equation 5.2) of both algorithms as a function of time for each dataset when the number of machines is varied. From Figures 5.1 and 5.2, we observe that HPSGD obtains better training RMSE on Movielens and Netflix, while both algorithms produce almost the same test RMSE. On the other hand, Figures 5.3 and 5.4 indicates that NOMAD obtains better training RMSE on Yahoo-medium and Yahoo-large, while both algorithms yield almost the same test RMSE. The reason for this may be the order of SGD updates that are not the same for HPSGD and NOMAD.

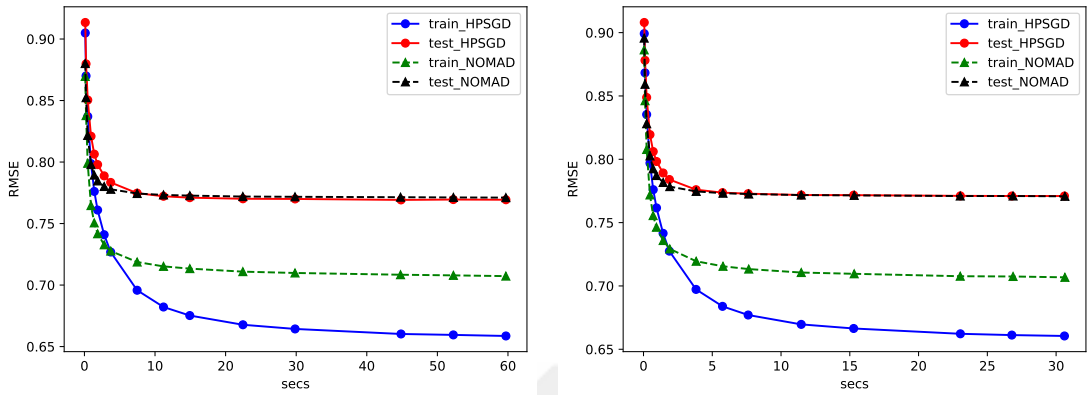
5.3.2 Scalability and Speedup

To investigate the scalability of HPSGD, we plotted the training and test RMSE as a function of the cost of HPC (time in seconds $\times$ number of machines $\times$ number of cores per machine). From Figures 5.5, 5.6, 5.7 and 5.8, we observe that all lines coincide with each other, which means that HPSGD linearly scales without sacrificing the matrix completion accuracy. We only observe a slight slowdown on the Netflix dataset when the number of machines is 32.

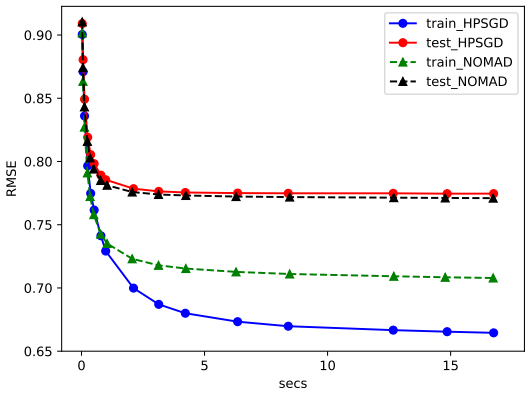
In addition, Figure 5.9 shows the speedup of the HPSGD algorithm for each dataset when the number of machines is varied. We define the speedup as T_k/T_4, where T_k is the time taken on k machines per epoch.

5.3.3 Throughput

To examine the throughput of HPSGD and NOMAD, we plotted the total number of SGD updates as a function of time for both algorithms. From Figure 5.10, we observe that both HPSGD and NOMAD display similar performances on Movielens and Netflix datasets. However, in compassion to NOMAD, HPSGD achieves $5\times$ and $6\times$ better throughput on Yahoo-medium and Yahoo-large datasets, respectively. This is due to the fact that NOMAD suffers from the overhead of the fine-grained partitioning scheme more in Yahoo datasets, since they are relatively sparse, and their number of columns are substantially higher in comparison to Movielens and Netflix datasets (see Table 5.1).

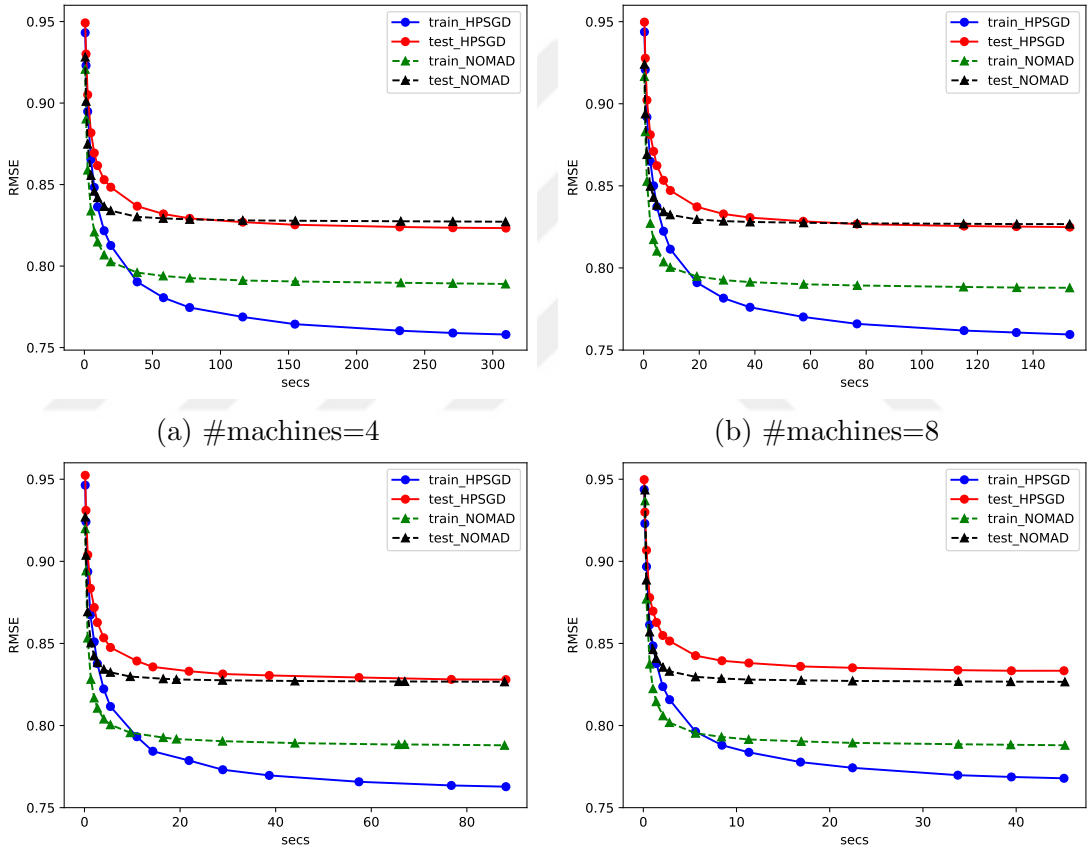


(a) #machines=4 (b) #machines=8



(c) #machines=16

Figure 5.1: Train and Test RMSE comparisons of HPSGD and NOMAD on Movilens dataset, when the number of machines is varied.



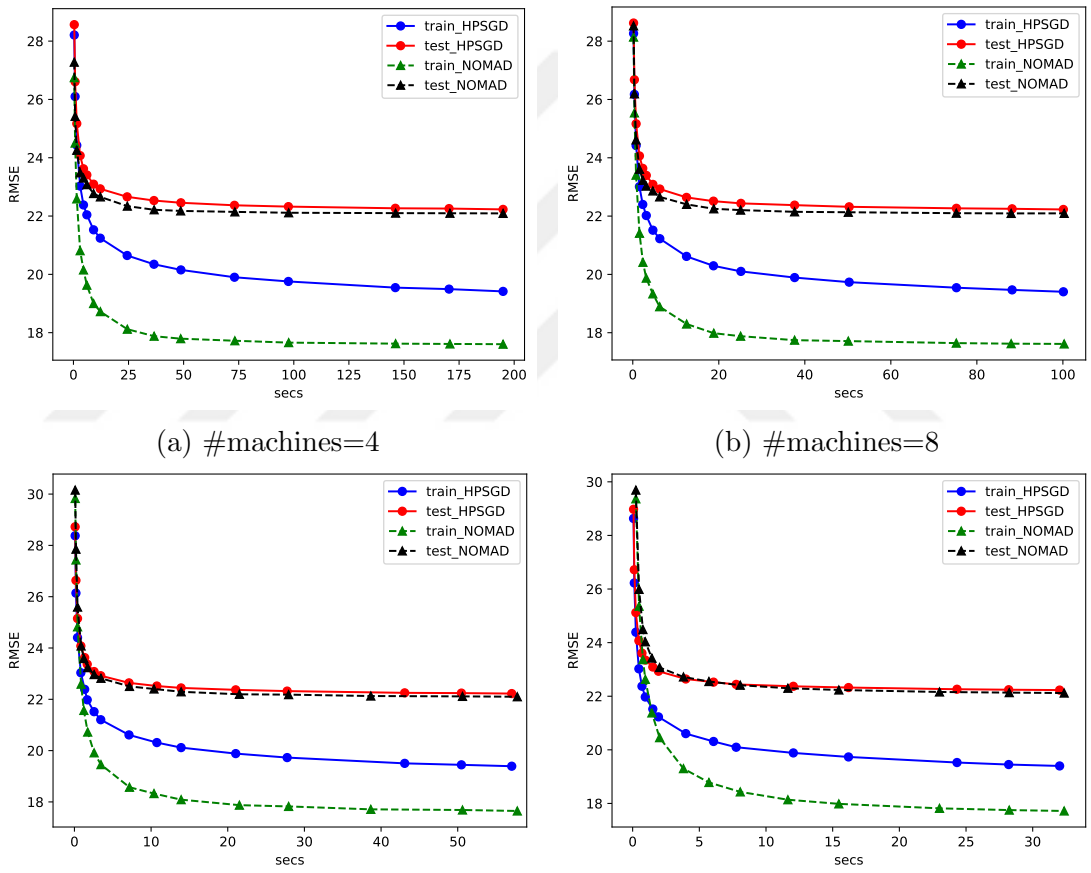
(a) #machines=4

(b) #machines=8

(c) #machines=16

(d) #machines=32

Figure 5.2: Train and Test RMSE comparisons of HPSGD and NOMAD on Netflix dataset, when the number of machines is varied.



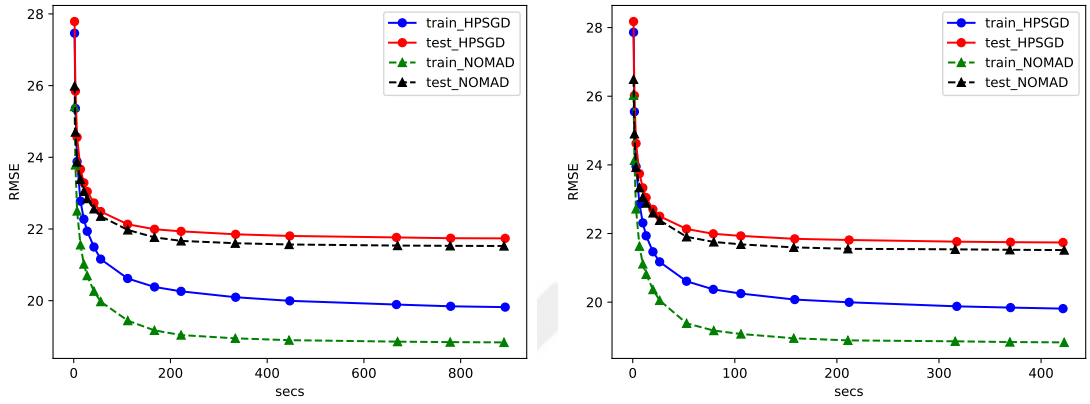
(a) #machines=4

(b) #machines=8

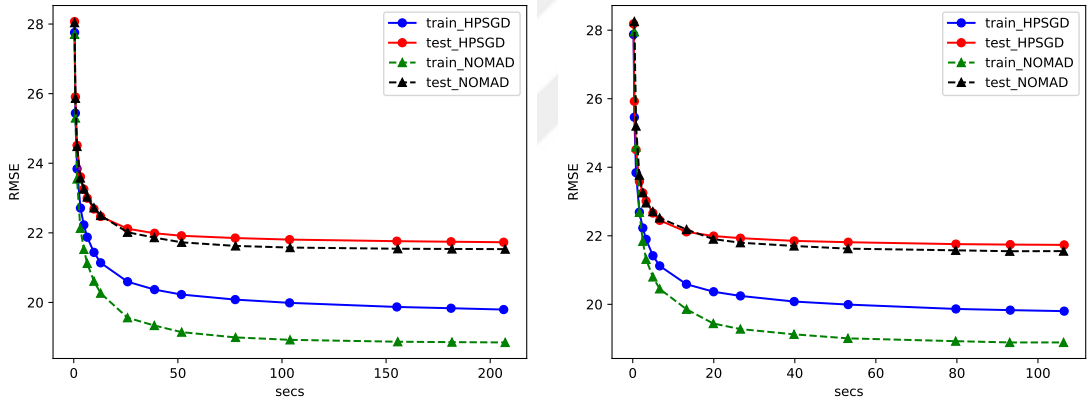
(c) #machines=16

(d) #machines=32

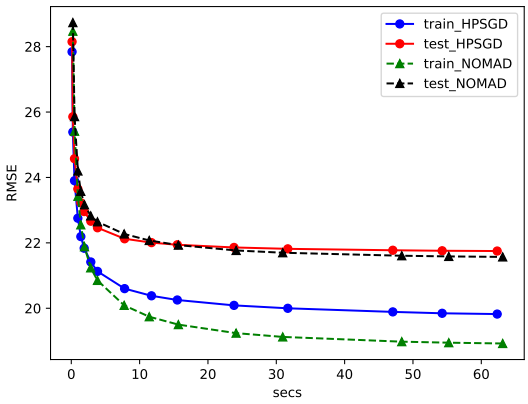
Figure 5.3: Train and Test RMSE comparisons of HPSGD and NOMAD on Yahoo-medium dataset, when the number of machines is varied.



(a) #machines=4 (b) #machines=8



(c) #machines=16 (d) #machines=32



(e) #machines=64

Figure 5.4: Train and Test RMSE comparisons of HPSGD and NOMAD on Yahoo-large dataset, when the number of machines is varied.

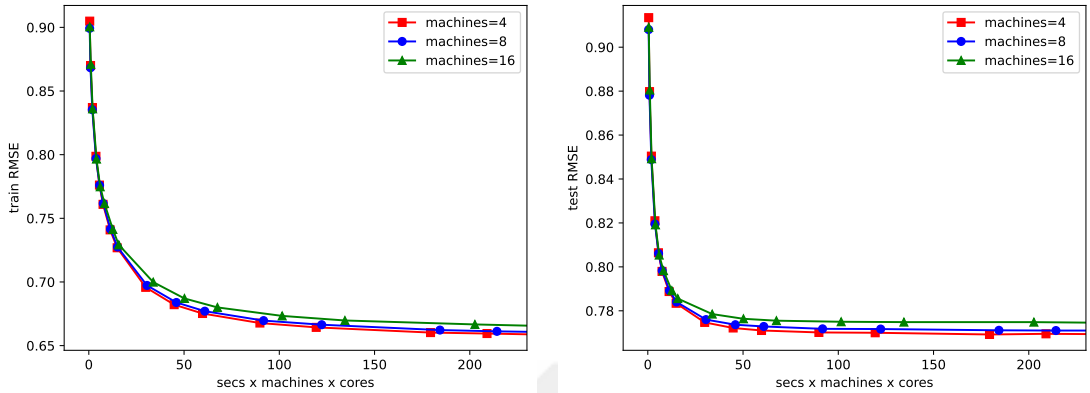


Figure 5.5: Train and Test RMSE of HPSGD as a function of the cost of HPC on Movielens dataset, when the number of machines is varied.

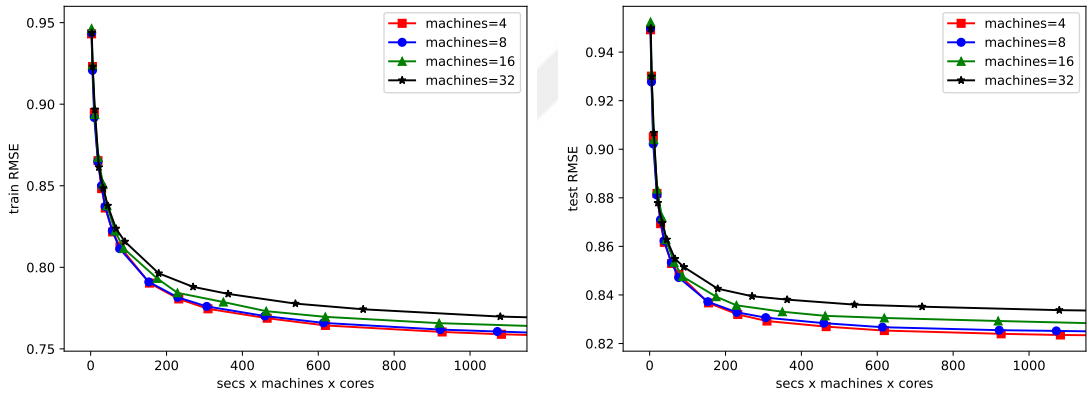


Figure 5.6: Train and Test RMSE of HPSGD as a function of the cost of HPC on Netflix dataset, when the number of machines is varied.

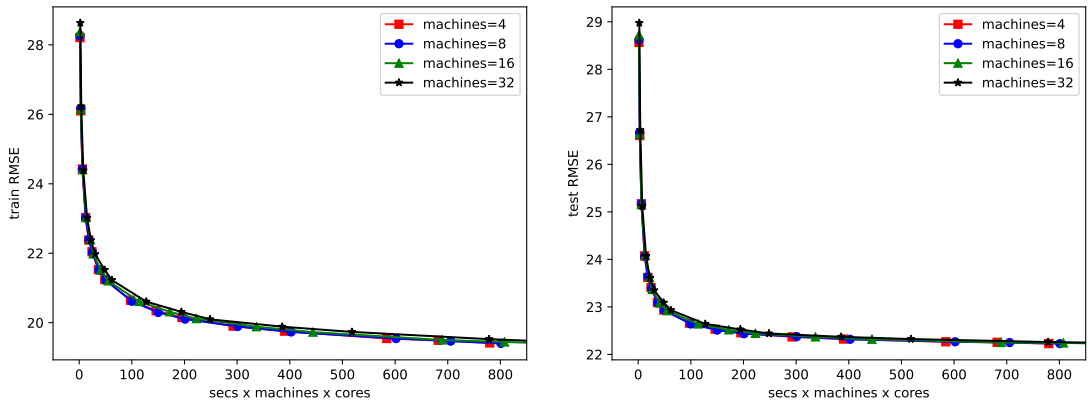


Figure 5.7: Train and Test RMSE of HPSGD as a function of the cost of HPC on Yahoo-medium dataset, when the number of machines is varied.

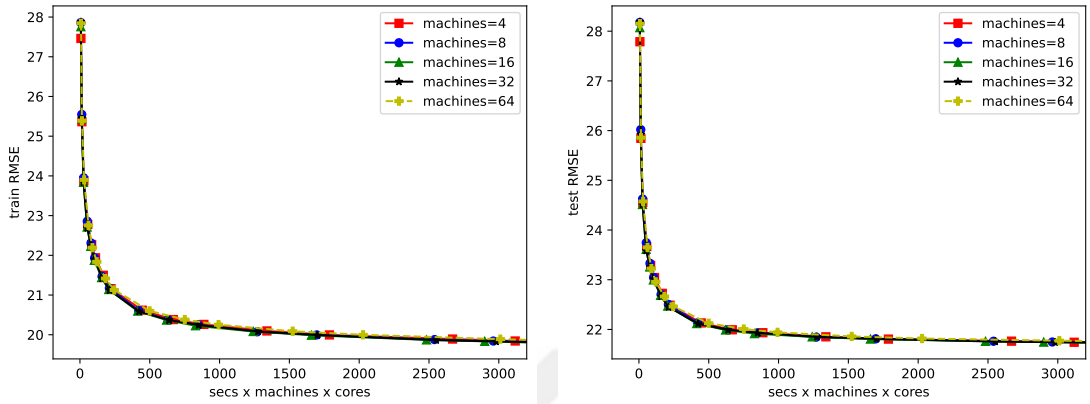
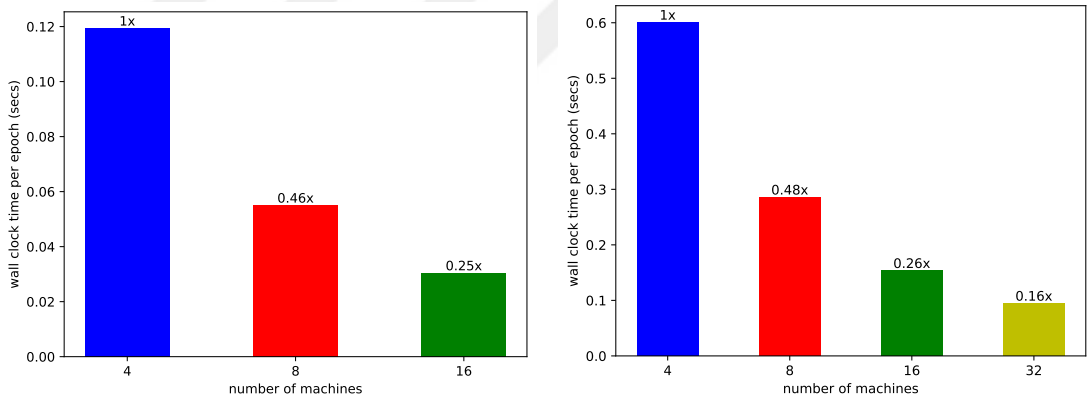
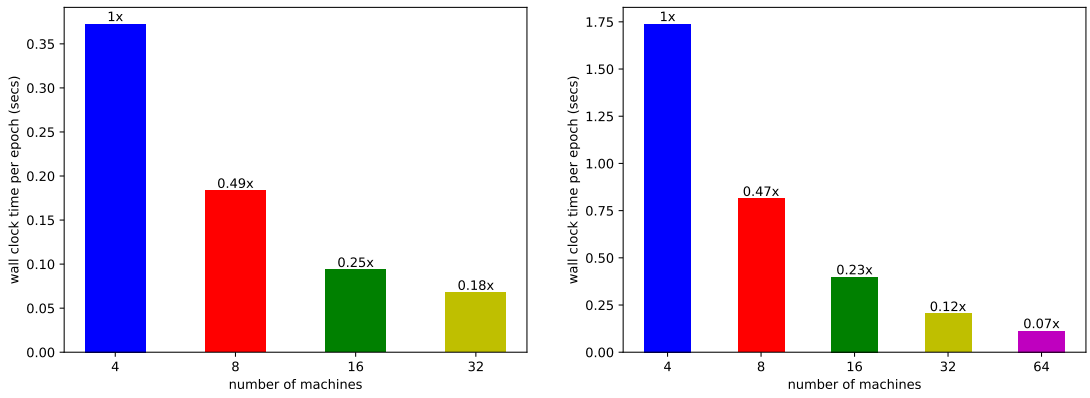


Figure 5.8: Train and Test RMSE of HPSGD as a function of the cost of HPC on Yahoo-large dataset, when the number of machines is varied.

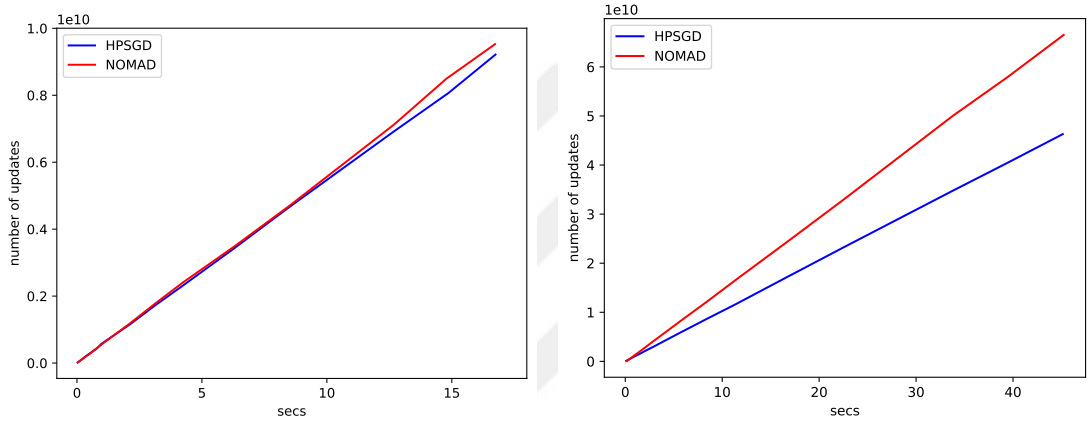


(a) Movielens (b) Netflix



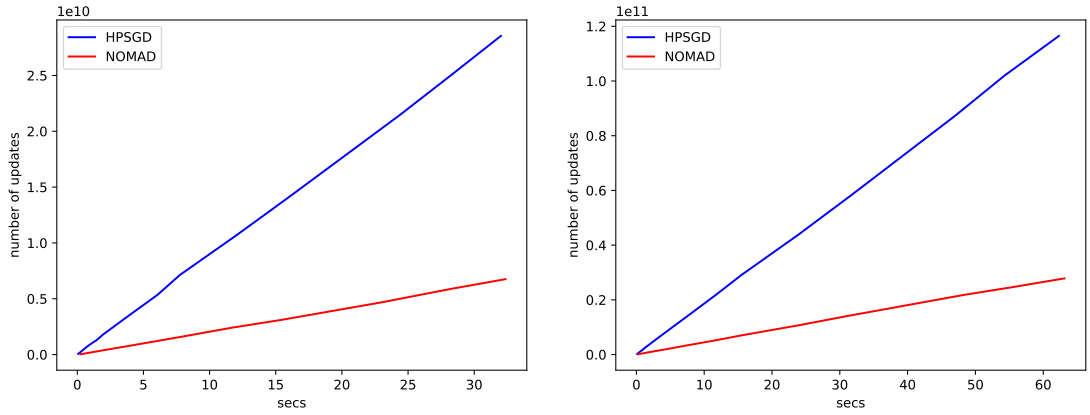
(c) Yahoo-medium (d) Yahoo-large

Figure 5.9: The speedup of HPSD for each dataset. The speedup defined as T_k/T_4, where T_k is the time taken on k machines per epoch.



(a) Movielens, #machines=16

(b) Netflix, #machines=32



(c) Yahoo-medium, #machines=32

(d) Yahoo-large, #machines=64

Figure 5.10: Total number of SGD updates of HPSGD and NOMAD as a function time for each dataset.

Chapter 6

Conclusion and Future Work

We focused on the efficient parallelization of the SGD algorithm for matrix completion on a high-performance computing platform in distributed memory setting. We proposed a new distributed SGD algorithm with non-blocking communication between processors and asynchronous computation in individual processors. We presented a flexible partitioning scheme to overlap the communication and the computation during the execution. We introduced a three-layered hybrid parallel architecture to exploit the full potential of modern high-performance computing platforms by utilizing thread-level parallelism. We showed the matrix completion accuracy, the scalability, and the throughput of our algorithm on four different real-world benchmark datasets.

Lastly, the exponential growth of web-scale data is an inevitable fact. Therefore, as future work, we are planning to show the validity of our proposed algorithm on much larger synthetic datasets with millions of rows, millions of columns, and billions of nonzero entries.

Bibliography

[1] G. Linden, B. Smith, and J. York, "Amazon. com recommendations: Item-to-item collaborative filtering," *IEEE Internet computing*, vol. 7, no. 1, pp. 76–80, 2003.

[2] Y. Koren, R. Bell, and C. Volinsky, "Matrix factorization techniques for recommender systems," *Computer*, vol. 42, no. 8, pp. 30–37, 2009.

[3] G. Dror, N. Koenigstein, Y. Koren, and M. Weimer, "The yahoo! music dataset and kdd-cup'11," in *Proceedings of KDD Cup 2011*, pp. 3–18, PMLR, 2012.

[4] J. Bennett, S. Lanning, *et al.*, "The netflix prize," in *Proceedings of KDD cup and workshop*, vol. 2007, p. 35, New York, NY, USA., 2007.

[5] G. Takács, I. Pilászy, B. Németh, and D. Tikk, "Scalable collaborative filtering approaches for large recommender systems," *The Journal of Machine Learning Research*, vol. 10, pp. 623–656, 2009.

[6] R. Gemulla, E. Nijkamp, P. J. Haas, and Y. Sismanis, "Large-scale matrix factorization with distributed stochastic gradient descent," in *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 69–77, 2011.

[7] I. Pilászy, D. Zibriczky, and D. Tikk, "Fast als-based matrix factorization for explicit and implicit feedback datasets," in *Proceedings of the fourth ACM conference on Recommender systems*, pp. 71–78, 2010.

[8] H.-F. Yu, C.-J. Hsieh, S. Si, and I. Dhillon, "Scalable coordinate descent approaches to parallel matrix factorization for recommender systems," in *2012 IEEE 12th international conference on data mining*, pp. 765–774, IEEE, 2012.

[9] C. Teflioudi, F. Makari, and R. Gemulla, "Distributed matrix completion," in *2012 ieee 12th international conference on data mining*, pp. 655–664, IEEE, 2012.

[10] H. Yun, H.-F. Yu, C.-J. Hsieh, S. Vishwanathan, and I. Dhillon, "Nomad: Non-locking, stochastic multi-machine algorithm for asynchronous and decentralized matrix completion," *Proceedings of the VLDB Endowment*, vol. 7, no. 11, 2014.

[11] W.-S. Chin, Y. Zhuang, Y.-C. Juan, and C.-J. Lin, "A fast parallel stochastic gradient method for matrix factorization in shared memory systems," *ACM Transactions on Intelligent Systems and Technology (TIST)*, vol. 6, no. 1, pp. 1–24, 2015.

[12] B. Recht and C. Ré, "Parallel stochastic gradient algorithms for large-scale matrix completion," *Mathematical Programming Computation*, vol. 5, no. 2, pp. 201–226, 2013.

[13] B. Recht, C. Re, S. Wright, and F. Niu, "Hogwild!: A lock-free approach to parallelizing stochastic gradient descent," *Advances in Neural Information Processing Systems*, vol. 24, pp. 693–701, 2011.

[14] S. Suri and S. Vassilvitskii, "Counting triangles and the curse of the last reducer," in *Proceedings of the 20th international conference on World wide web*, pp. 607–614, 2011.

[15] F. M. Harper and J. A. Konstan, "The movielens datasets: History and context," *Acm transactions on interactive intelligent systems (tiis)*, vol. 5, no. 4, pp. 1–19, 2015.

[16] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the thirteenth international*

conference on artificial intelligence and statistics, pp. 249–256, JMLR Workshop and Conference Proceedings, 2010.