Stochastic Gradient Descent for Matrix Completion: Hybrid Parallelization on Shared- and Distributed-Memory Systems

Kemal Büyükkaya[a,1], M. Ozan Karsavuran[b,1], Cevdet Aykanat[a,*]

[a]*Bilkent University, Computer Engineering Department, 06800 Ankara, Turkey*
[b]*Lawrance Berkeley National Laboratory, 1 Cyclotron Road, Berkeley, CA 94720, USA*

Abstract

The purpose of this study is to investigate the hybrid parallelization, where both shared-memory and distributed-memory parallelism is exploited, for the Stochastic Gradient Descent (SGD) algorithm for solving the matrix completion problem on a high-performance computing (HPC) platform. 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, collaborative filtering, matrix factorization, distributed-memory systems, shared-memory systems, hybrid parallelism

1. Introduction

Recommendation systems (Chen & Wang, 2022). play a fundamental role in the success of e-commerce companies such as Amazon (Linden et al., 2003) and Netflix (Koren et al., 2009). 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.). The nonzero values of this rating matrix denote preferences expressed by users on products and zero values denote missing preferences. Since users indicate their preferences for a limited number of products, the 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 (Mongia & Majumdar, 2021; Ramlatchan et al., 2018; Dror et al., 2012; Bennett et al., 2007; Takács et al., 2009). 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

[*] Corresponding author

Email addresses: kemal.buyukkaya@bilkent.edu.tr (Kemal Büyükkaya), MOKarsavuran@lbl.gov (M. Ozan Karsavuran), aykanat@cs.bilkent.edu.tr (Cevdet Aykanat)

[1] This work was done when Büyükkaya and M. Ozan Karsavuran was with Bilkent University.

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) (Shi et al., 2022; Gemulla et al., 2011), alternating least squares (ALS) (Priyati et al., 2022; Pilászy et al., 2010), and cyclic coordinate descent (CCD) (Chorobura et al., 2022; Yu et al., 2012). Nonnegative matrix factorization may be also used for latent factor analysis (Luo et al., 2016, 2021; Wu et al., 2022; Luo et al., 2022). 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 should note here that SGD is also utilized for sparse tensor completion (Singh et al., 2022).

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. Our contributions in this work are towards improving the parallel performance of the SGD algorithm. That is, our goal is developing scalable SGD algorithm while keeping its accuracy at the state-of-the-art level. For this purpose, we compare our algorithm with an available state-of-the-art algorithm on a real-benchmark dataset that contains one small, two medium, and one large real-world rating matrices. 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 paper is organized as follows: Section 2 formally defines the matrix completion problem with the necessary notations and gives background information about the SGD algorithm. Section 3 presents an analysis of the related work. Section 4 describes Hybrid Parallel Stochastic Gradient (HPSGD) algorithm in detail. The experimental setup, datasets, and results are discussed in Section 5. Lastly, Section 6 concludes the paper with possible future work and final remarks.

2. Background

In this section, 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 $\Omega \subseteq \{1, \dots, m\} \times \{1, \dots, n\}$ 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 (Gemulla et al., 2011; Teflioudi et al., 2012; Yun et al., 2014; Makari et al., 2015). 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:

$$\hat{r}_{ij} = w_i h_j^T. \tag{1}$$

Figure 1 shows the relationship between a user embedding w_i and a product embedding h_j in detail.

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

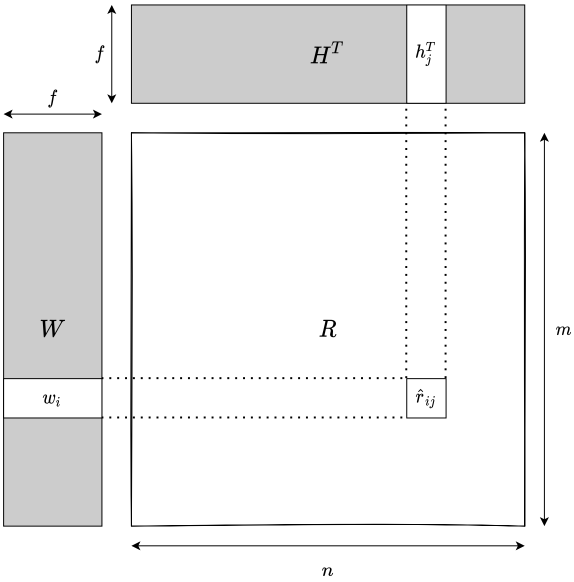


Figure 1: Rating matrix $R \in \mathbb{R}^{m\times n}$ and latent factor matrices $W \in \mathbb{R}^{m\times f}$ and $H^T \in \mathbb{R}^{f\times n}$.

function $L(W, H)$ is mostly defined as the regularized squared loss (Koren et al., 2009; Gemulla et al., 2011, 2015; Chin et al., 2015; Matsushima et al., 2017). That is,

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

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 (1).

2.2. Stochastic Gradient Descent (SGD)

The objective function (2) is a multivariate differentiable function, and there are different optimization algorithms to minimize multivariate differentiable functions such as stochastic gradient descent (SGD) (Shi et al., 2022; Gemulla et al., 2011), alternating least squares (ALS) (Priyati et al., 2022; Pilászy et al., 2010), and cyclic coordinate descent (CCD) (Chorobura et al., 2022; Yu et al., 2012). However, it is shown that SGD can achieve high completion accuracy while scaling to large scale rating matrices (Yun et al., 2014). Thus, we 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) at a fixed point r_{ij} can be defined as:

$$\nabla_{w_i} L(W, H) = (r_{ij} - \hat{r}_{ij})h_j + \lambda w_i, \quad (3)$$

$$\nabla_{h_j} L(W, H) = (r_{ij} - \hat{r}_{ij})w_i + \lambda h_j. \quad (4)$$

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:

$$w_i \leftarrow w_i - \gamma(r_{ij} - \hat{r}_{ij})h_j + \lambda w_i, \quad (5)$$

$$h_j \leftarrow h_j - \gamma(r_{ij} - \hat{r}_{ij})w_i + \lambda h_j. \quad (6)$$

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.

3. Related Work

In this section, we review the parallel SGD algorithms in the literature, discuss the methodology that they stand for, and mention their major drawbacks. In the literature, there are works involving parallel SGD which use GPU (Xie et al., 2017; Li et al., 2018; Elahi et al., 2022) as well as heterogeneous multi-CPU-GPU systems (Yu et al., 2021; Huang et al., 2021), include hardware level solutions for locks (Wu et al., 2018), utilize parallel disk systems (Lee et al., 2018; Oh et al., 2015), works on streaming data (Khan et al., 2019; Si et al., 2022), among many others. Another way of improving the SGD algorithm is making the updates more effective than the conventional SGD (updating according to (5) and (6)). Examples of such works may be found in (Luo et al., 2013; Khan et al., 2020, 2022b,a). These contributions may be applied to our work for further improvement. In this work, we focus on parallel SGD algorithms proposed for shared- and distributed-memory parallel systems as follows:

3.1. Shared-Memory Parallel SGD Algorithms

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

The SGD is an iterative algorithm, and its sequential implementations do not scale for large rating matrices. Parallel SGD algorithms aim to overcome this scalability problem by distributing the computations 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 of W and row h_j of H (Recht & Ré, 2013). 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.

Hogwild algorithm introduced by Recht et al. (2011) introduces an asynchronous lock-free algorithm assuming that the updates given in (5) and (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 (5) and (6) can be executed via different threads in parallel. Even though possible data hazards may occur during the execution (see Figure 2), 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.

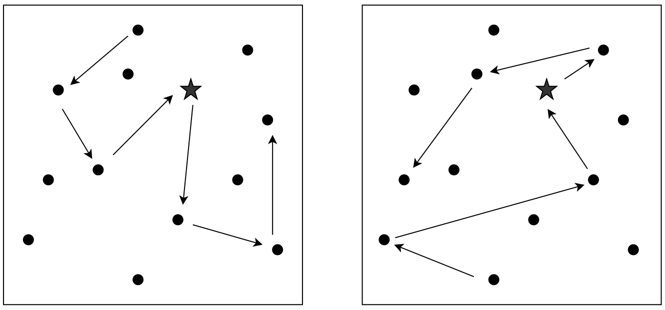


Figure 2: An example update sequence for two threads in HogWild. Black dots are randomly sampled nonzero entries. Arrows denote the order of the update sequence. The star indicates the potential data hazard since it is simultaneously accessed by two threads in their 4^{th} iterations.

Another popular strategy (Chin et al., 2015) 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 $(i, j) \in \Omega$.

Fast Parallel Stochastic Gradient (FPSGD) algorithm proposed by Chin et al. (2015) partitions the rating matrix R into $k' \times k'$ 2-dimensional (2D) blocks with $k' > k$, where k is the number of worker threads; and uses a task manager thread to distribute these 2D 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 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 2D 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 for the distributed-memory parallel systems.

Recently, an alternative asynchronous SGD (A^{2}SGD) (Qin & Luo, 2022) for shared-memory parallel systems is proposed. A^{2}SGD avoids the iteration concept during the training and in that way remove the synchronization requirement of the SGD completely. This alternative SGD algorithm still converges and attain high performance.

3.2. Distributed-memory Parallel SGD Algorithms

This subsection focuses on studies that solve the matrix completion problem on distributed-memory parallel systems.

By its nature, the SGD is an iterative algorithm, and the simplest strategy for distributing iterative algorithms among mul-

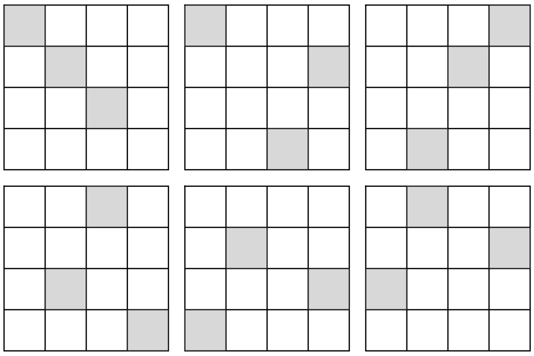


Figure 3: Examples of mutually independent block assignment in FPSGD for 4×4 2D blocks and 3 threads. In each matrix, 2D blocks which are assigned to threads are shown with shaded blocks. Each matrix shows different assignment during SGD.

tiple 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.

The Distributed Stochastic Gradient Descent (DSGD) algorithm of Gemulla et al. (2011) splits the rating matrix R into $k \times k$ 2D blocks, where k is the number of machines. Then, DSGD distributes these k^2 2D blocks to k machines in such a way that each machine has k mutually independent blocks. Two blocks are said to be mutually independent if they do not share any row and column of the rating matrix. DSGD defines this special type of block assignment scheme as stratum. In other words, mutually independent blocks of a stratum do not update the same rows of the latent factor matrices W and H. In DSGD, each stratum is executed in parallel within a single sub-epoch. A collection of stratums forms a strata if they cover the entire rating matrix when each stratum is executed within different sub-epochs. For example, Figure 4 indicates 6 different strata possible for a 3×3 2D blocks. 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.

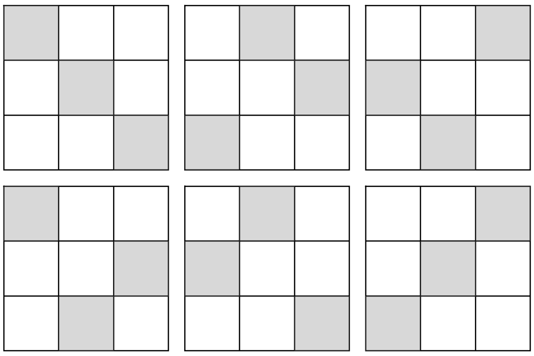
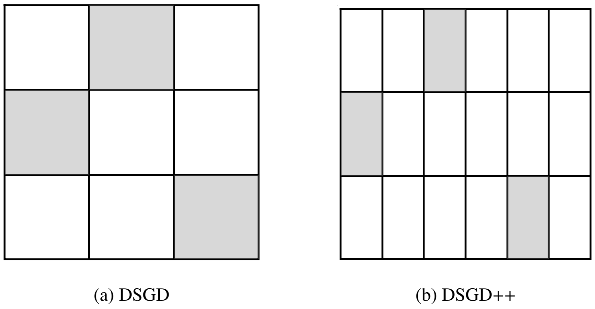


Figure 4: All possible mutually independent block assignments for a 3×3 2D blocks in DSGD. The first and second rows of the figure form two different strata.

In order to reduce processors' idle time, the DSGD++ algorithm of Teflioudi et al. (2012) divides the rating matrix R into $k \times 2k$ 2D blocks (see Figure 5b), 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 (Suri & Vassilvitskii, 2011).



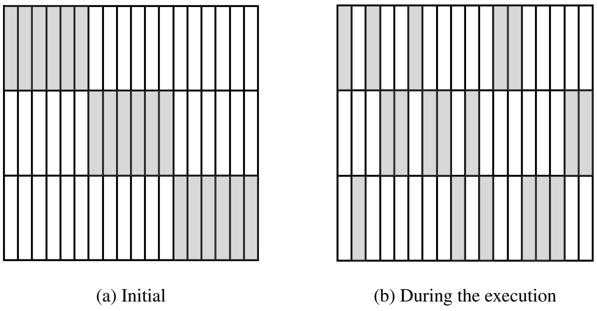
(a) DSGD

(b) DSGD++

Figure 5: A comparison of data partitioning schemes between DSGD and DSGD++ for $k = 3$ machines.

To overcome the curse of the last reducer problem, the NOMAD algorithm of Yun et al. (2014); Yu et al. (2016) follows the fine-grained partitioning approach and divides the rating matrix R into $k \times n$ 2D blocks where k is the number of machines and n is the number columns 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 6).

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.


(a) Initial

(b) During the execution

Figure 6: An illustration of block assignments of NOMAD for $k = 3$ machines and a rating matrix with $n = 12$ columns.

Alternating SGD (Luo et al., 2012; Shi et al., 2022) is a recent distributed asynchronous SGD model. In this model each iteration of the SGD algorithm is divided into two. In each of these two sub-iterations, similar to the ALS, one of the factor matrices is kept fixed, whereas other one is updated. In this way, asynchronous updates are limited to one of the factor matrices in each sub-iteration.

4. Hybrid Parallel Stochastic Gradient Descent (HPSGD)

In this section, 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 proposed HPSGD algorithm partitions the rating matrix R into $k \times c$ 2D 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 1D row blocks, and the latent factor matrix H into c 1D row blocks. Then, the algorithm assigns each row slice of the rating matrix R and 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 row slices of the latent factor matrix H to machines in such a way that each machine holds the ownership of c/k number of different row slices of latent factor matrix H. Figure 7 indicates the initial mutually independent block assignment in HPSGD with $k = 3$ and $c = 12$.

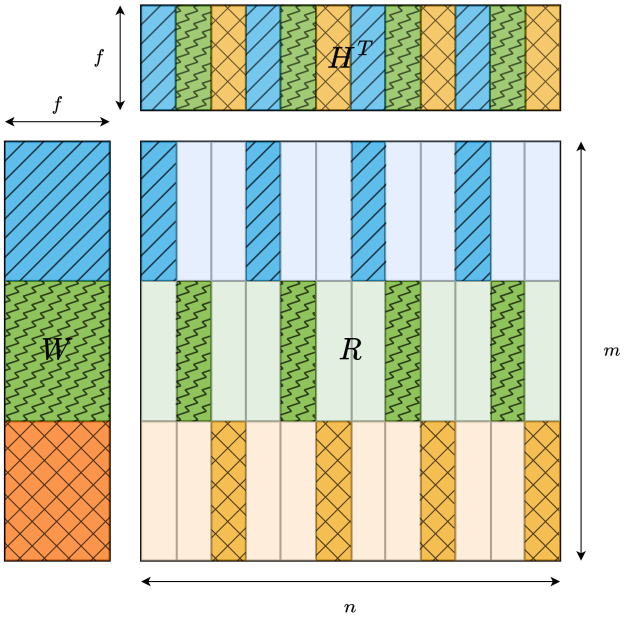


Figure 7: Initial mutually independent block assignment in HPSGD with $k = 3$ machines and $c = 12$ column blocks. 2D blocks with different colors/fill-in-patterns denote different machines.

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

4.2. Three-Layered Hybrid Architecture

In HPSGD, we utilize 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 threads and a single communicator thread (see Figure 8).

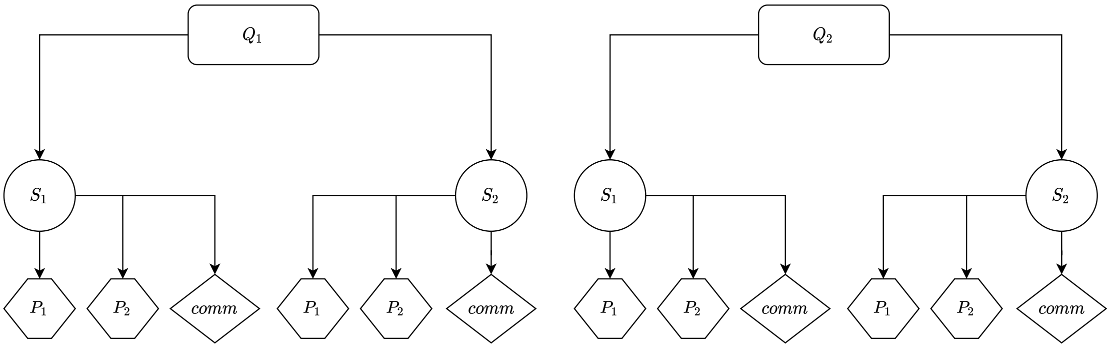


Figure 8: An illustration of the HPSGD architecture where the number of MPI processes, the number of MPI threads per MPI process, and the number of updater threads per MPI-thread are all equal to 2.

Each thread group S maintains two different concurrent queues; the first one contains indices of the row slices of the H-matrix to be updated, and the second one contains the indices of the row slices of the H-matrix to be sent. During the execution of the algorithm, each updater thread of the thread group S_t of the machine Q_p simultaneously pops an index from the first queue (multi-reader and single-writer), performs the related SGD updates given in (5) and (6) for each nonzero entry in the corresponding row slice. Then, each updater thread pushes these indices into the second queue (multi-writer and single-reader). In parallel, the communicator thread of the thread group S_t of machine Q_p pops an index from the second queue and sends the corresponding rows of the latent factor matrix H to the communicator thread of the thread group S_t of the machine $Q_{(p+1) \bmod k}$. 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 9 shows how the ownership of the row slices of the latent factor matrix H changes after all machines performed related SGD updates for their initially assigned blocks and sent the updated row slices of the latent factor matrix H to another machine. For example, the first three rows of the H-matrix is initially assigned to Q_1, Q_2, and Q_3, whereas during the execution assignment respectively changed to Q_2, Q_3, and Q_1.

Algorithms 1, 2 and 3 show the general structure of the HPSGD. Algorithm 1 is the main SGD algorithm. Lines 1–6 define the required parameters of the HPSGD. Then, in lines 7 and 8, factor matrices are initialized. In lines 9 and 10, initial assignment of the factor matrices are performed. The for-loop in lines 11–15 initializes the queues for each row slice of the H-factor matrix. Lastly, the for-loop in lines 16–18 launches updater and communicator threads.

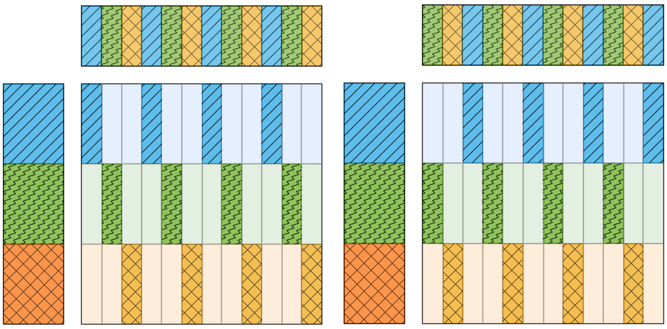
Algorithm 2 shows the operations performed by the updater threads. As seen in the algorithm, the threads continue to run while the update queue is not empty. In line 3, the H-matrix row to be updated is popped from the queue. Then, in line 4, related SGD updates are performed. Finally, the updated H-matrix row is pushed to send queue.

Algorithm 3 shows the operations performed by the communicator threads. As seen in the algorithm, the threads continue to run while the send queue is not empty. In line 3, the H-matrix row to be sent is dequeued from the queue. Then, in line 4, the row is sent to the next machine. In line 6, the rows updated by other machines are received and pushed to the update queue in line 7.

The use of asynchronous and non-blocking communication enables the computation and communication to overlap as long as both queues of a thread group are not empty, which can improve the overall performance of the algorithm. On the other hand, asynchronous nature of the proposed HPSGD necessitates a scheme to ensure complete iteration of data as described as below.

An epoch is completed when all the nonzero values in the rating matrix R have been processed once. This requires processing all the row slices of the latent factor matrix H, which are distributed across K machines. To achieve this, each machine communicates its row slices to the other machines, so that all the machines can update their respective slices of the H matrix. Once all the machines have processed all their row slices, the epoch is completed.

Similarly, to complete x number of epochs, each machine needs to communicate and update their respective row slices of the H matrix at least $x \times K$ times. To ensure that each row slice of the latent factor matrix H is updated the desired number of times during training, the number of updates that have taken place on each row slice is stored in its own header section. This allows each machine to keep track of the progress on the number of updates on its own row slices, and to stop processing them once the desired number of updates has been reached. By doing so, the algorithm ensures that each row slice is processed the same number of times, regardless of its position or size, which helps to maintain consistency and fairness in the training process.



(a) Initial

(b) During the execution

Figure 9: An illustration of the HPSGD algorithm with $k = 3$ machines and $c = 12$ column blocks. 2D blocks with different colors/fill-in-patterns denote different machines.

4.3. Complexity Analysis

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

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 row 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 $K \times C$ / $K$ / $C$ 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 row slices of $H$ is processed $K \times epochs$ times
```

Algorithm 2 The updater thread of HPSGD.

```
▷ An updater thread of thread group $S_t$ of 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 (5)-(6) for all nnz( $R_{p,j}$)
        $Q_p.S_t$.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
```

4.3.1. Space Complexity

Each machine has to store the $1/k$ portion of the latent factor matrix W of 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 row slices of the latent factor matrix H each of size approximately $(n \times f)/c$ (see Figure 10). Hence, the total space complexity of a single machine can be written as:

$$O(\frac{(m \times f) + |\Omega| + (n \times f)}{k}). \tag{7}$$

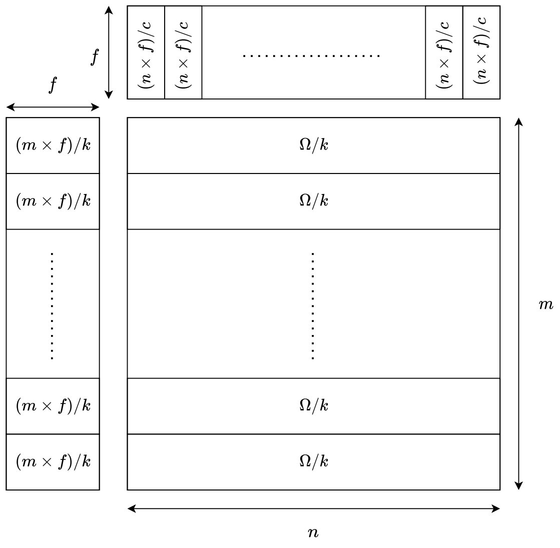


Figure 10: Space complexity of the nonzeros and factor matrix rows for each machine.

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 row slices of the latent factor matrix H. Each row slice of the latent factor matrix H has the size of approximately $((n \times f)/c)$. There are c different row 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.

5. Experiments and Results

5.1. Datasets

In our experiments, we used four benchmark datasets obtained from real-world data: `Movielens` (Harper & Konstan, 2015), `Netflix` (Bennett et al., 2007), `Yahoo-medium` (Dror et al., 2012), and `Yahoo-large` (Dror et al., 2012). Table 1 indicates the number of rows, columns, nonzeros, and the density for each dataset, where the density d is defined as:

$$d = \frac{|\Omega|}{m \times n}. \tag{8}$$

Table 1: Statistics for each dataset.

	number of			
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 2: Parameters for each dataset.

	Dataset			
	`Movielens`	`Netflix`	`Yahoo-medium`	`Yahoo-large`
$\|\Omega^{train}\|$	18,000,236	90,432,456	55,749,965	227,520,247
$\|\Omega^{test}\|$	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	1.00	1.00

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 $(-1/\sqrt{f}, +1/\sqrt{f})$ (Glorot & Bengio, 2010).

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

$$RMSE = \sqrt{\frac{\sum_{(i,j)\in\Omega^{test}}(r_{ij} - \hat{r}_{ij})^2}{|\Omega^{test}|}}. \tag{9}$$

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

$$\gamma = \frac{\gamma_i}{(1 + \beta t)}, \tag{10}$$

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 $(i, j) \in \Omega^{train}$.

In HPSGD, the hyperparameter c represents the number of row slices of the latent factor matrix H assigned to each machine. By varying the value of c, the algorithm can be tuned to find the optimal balance between computation and communication efficiency.

Grid search is a common technique used to find the optimal hyperparameters for a given machine learning model. To determine the optimal value of c for each dataset, we started with an intuition that each row slice of the H-matrix may have 1024 rows. Then, we determined the c values for each data that makes the number of rows in a row slice 128, 256, 512, 2048, and 5096 approximately. The optimal value of c was determined by evaluating the performance of the HPSGD algorithm on a validation set for each value of c. Table 2 lists the parameters used in the experiments for each dataset.

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.

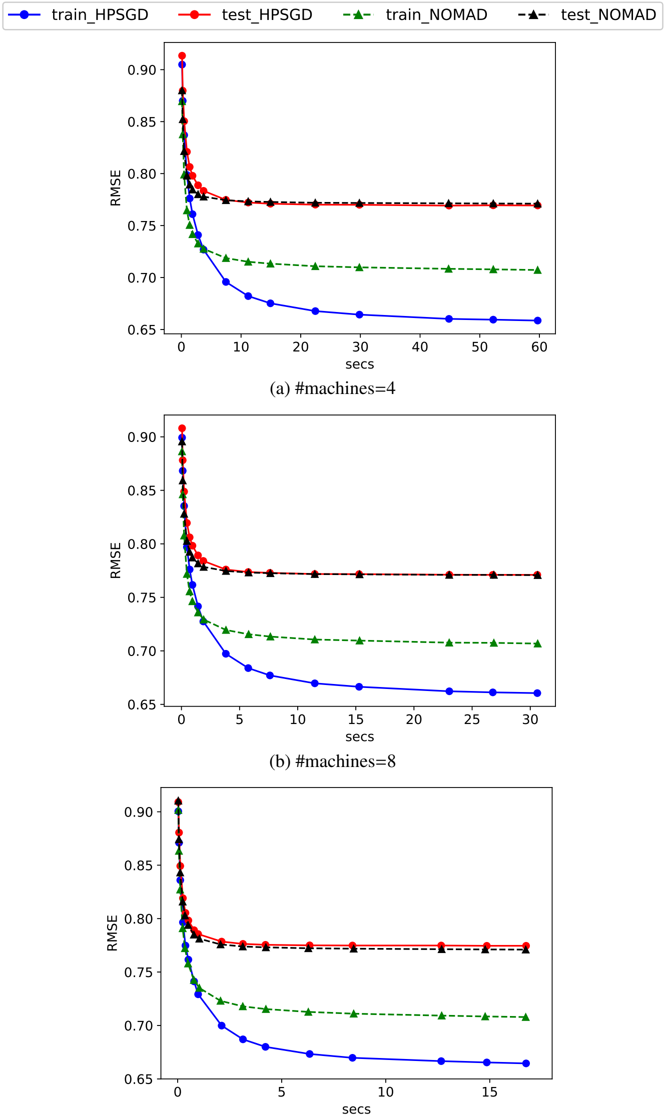
In each experiment, 4, 8, and 16 machines are used for `Movielens` dataset, whereas 4, 8, 16, and 32 machines are used for `Netflix` and `Yahoo-medium` datasets. Lastly, 4, 8, 16, 32, and 64 machines are used for `Yahoo-large` dataset. Note that the increasing number of machines relates to increasing size of the dataset as well as density and sparsity pattern.

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 one and the number of updater threads per MPI thread to four in HPSGD. Similarly, we also fixed the number of updater threads per machine to four 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 (9)) of both algorithms as a function of time for each dataset when the number of machines is varied. Figures 11 and 12 display the training and test RMSE values respectively for Movielens and Netflix on different number of machines. From Figures 11 and 12, we observe that HPSGD obtains better training RMSE on Movielens and Netflix, while both algorithms produce almost the same test RMSE. This finding shows that the proposed HPSGD algorithm achieves better training RMSE compared to NOMAD at a given running time, or it achieves the same training RMSE in shorter running time. In other words, the advantage of the proposed algorithm is improving running time of the SGD algorithm, which is our aim in this work.

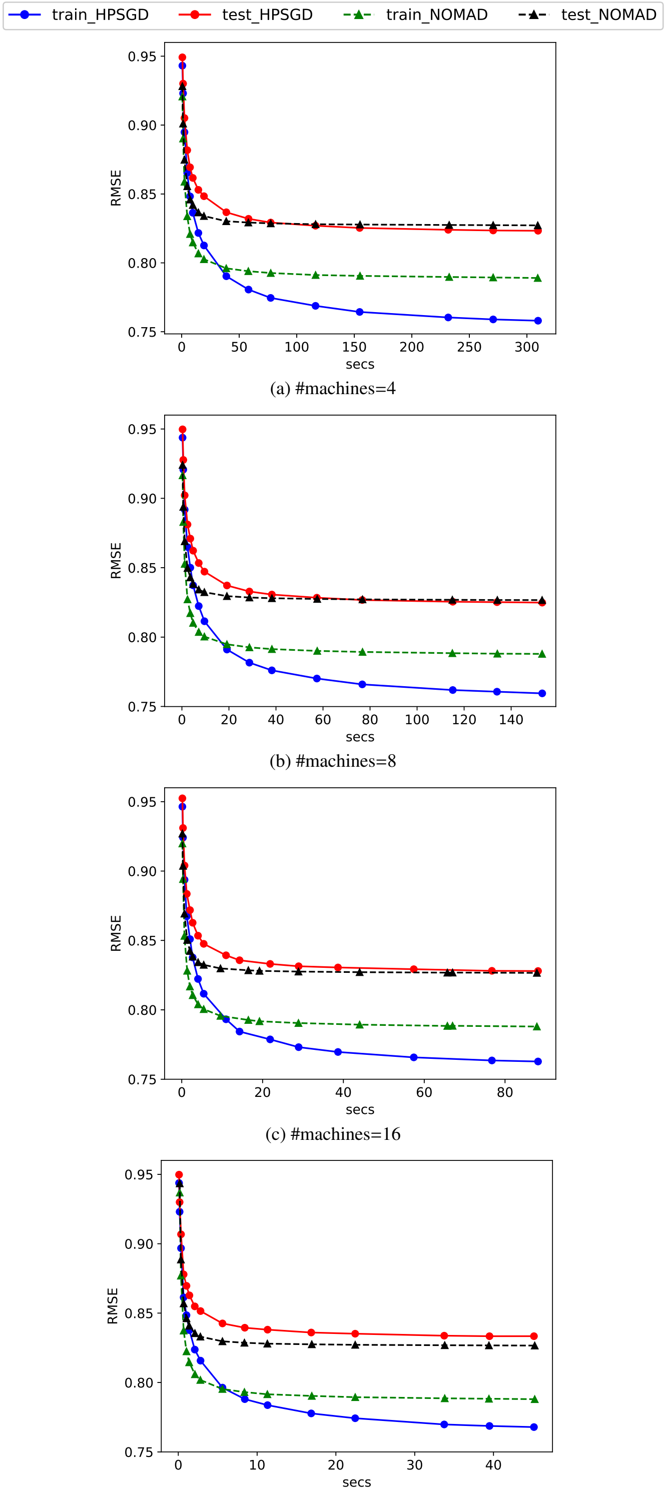


(a) #machines=4

(b) #machines=8

(c) #machines=16

Figure 11: Train and Test RMSE comparisons of HPSGD and NOMAD on Movielens dataset on 4, 8, and 16 machines.



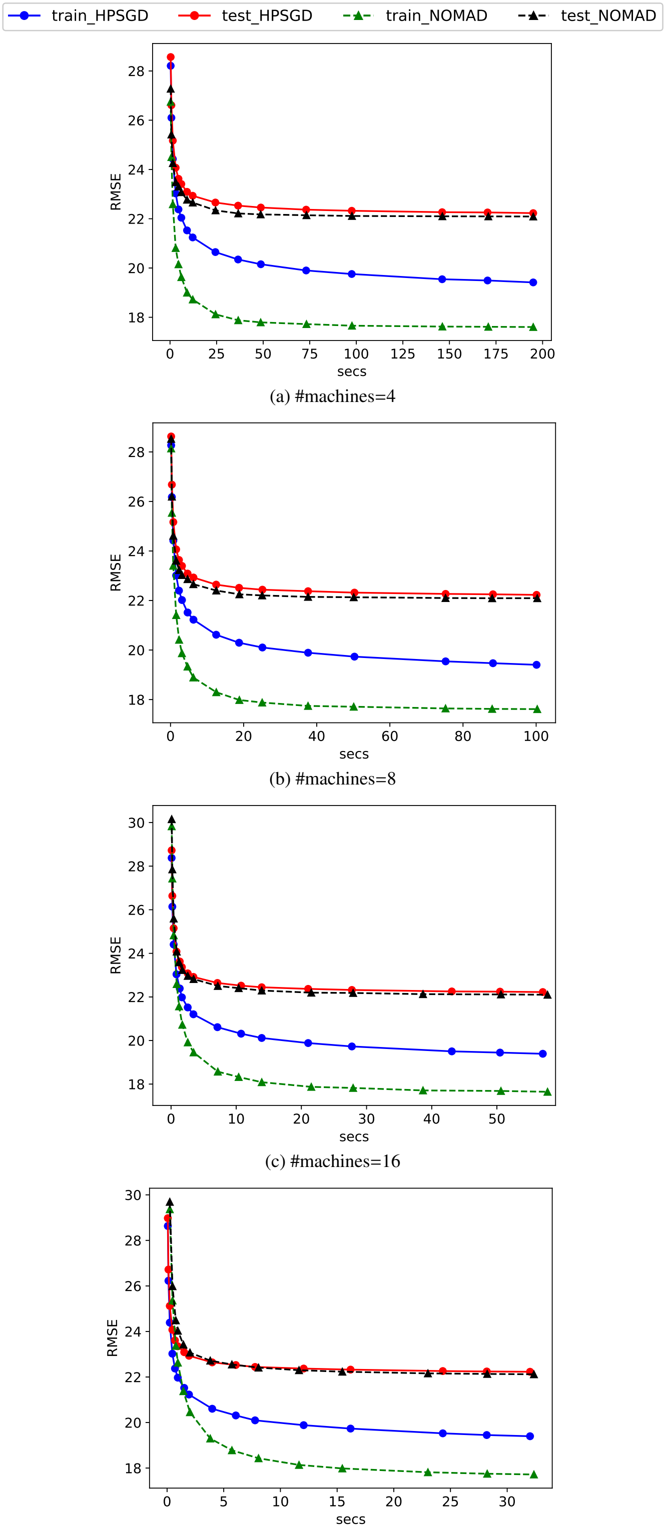
(a) #machines=4

(b) #machines=8

(c) #machines=16

(d) #machines=32

Figure 12: Train and Test RMSE comparisons of HPSGD and NOMAD on Netflix dataset on 4, 8, 16, and 32 machines.



(a) #machines=4

(b) #machines=8

(c) #machines=16

(d) #machines=32

Figure 13: Train and Test RMSE comparisons of HPSGD and NOMAD on `Yahoo-medium` dataset on 4, 8, 16, and 32 machines.

Figures 13 and 14 display the training and test RMSE values respectively for `Yahoo-medium` and `Yahoo-large` on different number of machines. As seen in the Figures 13 and 14, in contrast to `Movielens` and `Netflix` datasets, 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.

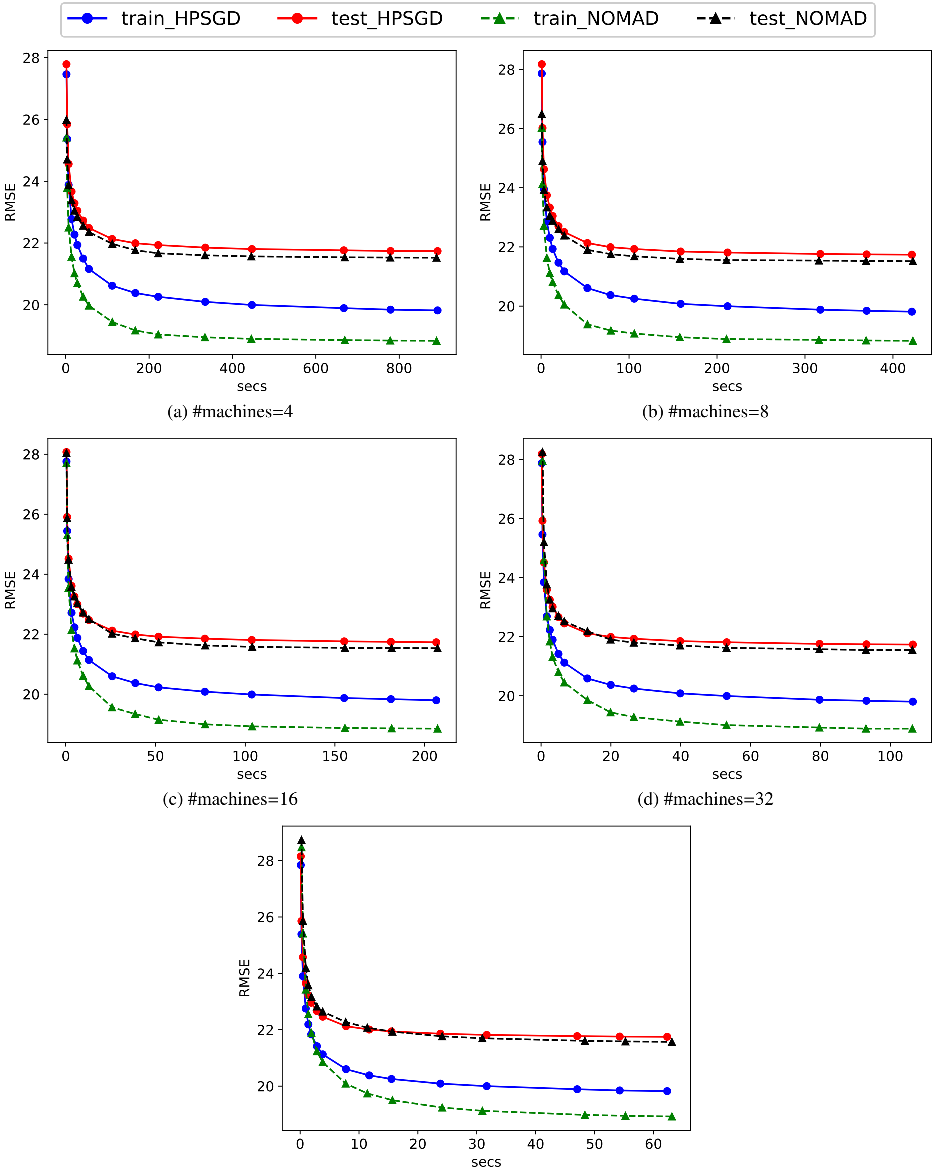
5.3.2. *Scalability and Speedup*

To investigate the scalability of HPSGD, we plotted the training and test RMSE as a function of the cost of HPC (time in seconds × number of machines × number of cores per machine). Figure 15 displays the cost values for `Movielens`, `Netflix`, `Yahoo-medium` and `Yahoo-large` datasets. As seen in the figures, we observe that all lines coincide with each other, which means that HPSGD linearly scales with almost no degradation in the matrix completion accuracy. We only observe a slight slowdown on the `Netflix` dataset when the number of machines is 32.

In addition, Figure 16 shows the strong scaling values in terms of parallel runtime for the HPSGD algorithm for each dataset when the number of machines is varied. We define the speedup as T_4/T_k, where T_k is the time taken on k machines per epoch. As seen in the figure, the proposed HPSGD achieves slightly super linear speedup on 8-machines on all datasets. Furthermore, HPSGD continues to achieve super linear speedup on `Yahoo-large` up to 32 machines as well. As seen in the figure, the HPSGD algorithm almost linearly scales up to 16 machines on all datasets. Furthermore, HPSGD continues to scale on `Netflix` and `Yahoo-medium` datasets on 32 machines with speedup values 6.37× and 5.49×, respectively, for an ideal speedup value of 8×. Finally, the HPSGD algorithm continues to scale on `Yahoo-medium` dataset on 64 machines almost linearly.

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



(a) #machines=4

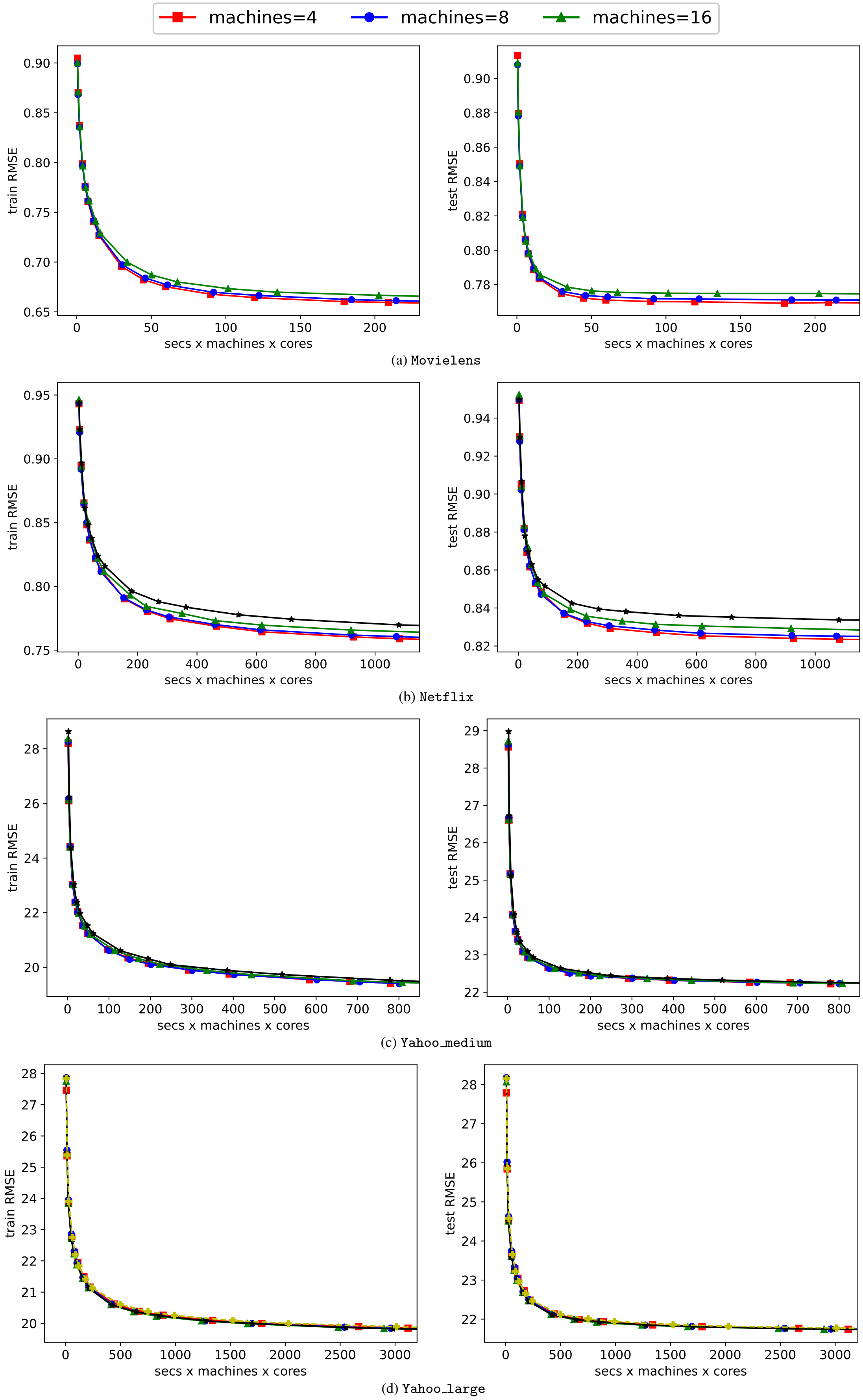
(b) #machines=8

(c) #machines=16

(d) #machines=32

(e) #machines=64

Figure 14: Train and Test RMSE comparisons of HPSGD and NOMAD on `Yahoo-large` dataset on 4, 8, 16, 32, and 64 machines.



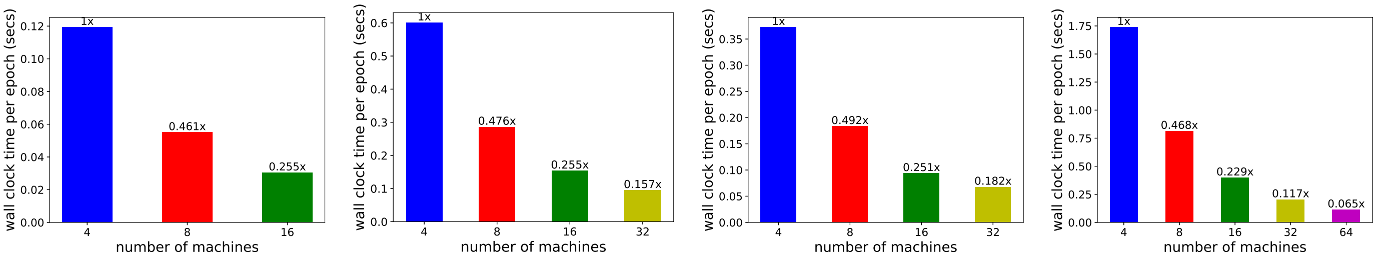
(a) `Movielens`

(b) `Netflix`

(c) `Yahoo_medium`

(d) `Yahoo_large`

Figure 15: Train and Test RMSE of HPSGD as a function of the cost of HPC, when the number of machines is varied.



(a) `Movielens` (b) `Netflix` (c) `Yahoo-medium` (d) `Yahoo-large`

Figure 16: The strong scaling values in terms of parallel runtime for the HPSGD algorithm for each dataset when the number of machines is varied.

for both algorithms. Figure 17 shows this throughput values for each dataset. As seen in Figure 17, both HPSGD and NOMAD display similar performances on `Movielens` and `Netflix` datasets. However, in comparison to NOMAD, HPSGD achieves 5× and 6× higher 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 have substantially larger number of columns in comparison to `Movielens` and `Netflix` datasets (see Table 1).

6. Conclusion and Future Work

We focused on the hybrid parallelization which utilizes both shared- and distributed-memory parallelization for scaling the SGD algorithm for matrix completion on a high-performance computing platform. We proposed a new distributed SGD algorithm with non-blocking communication between processors and asynchronous computation in individual processors. We utilized MPI for communications between nodes, whereas we utilized POSIX threads for shared-memory parallelism. 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. Overall, our algorithm achieves the state-of-the-art in accuracy while increasing the scalability and the throughput.

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.

Acknowledgments

This work was supported by the Scientific and Technological Research Council of Türkiye (TUBITAK) under project EEEAG-119E035.

References

Bennett, J., Lanning, S. et al. (2007). The netflix prize. In *Proceedings of KDD cup and workshop* (p. 35). New York, NY, USA. volume 2007.

Chen, Z., & Wang, S. (2022). A review on matrix completion for recommender systems. *Knowledge and Information Systems*, *64*, 1–34. doi:`10.1007/s10115-021-01629-6`.

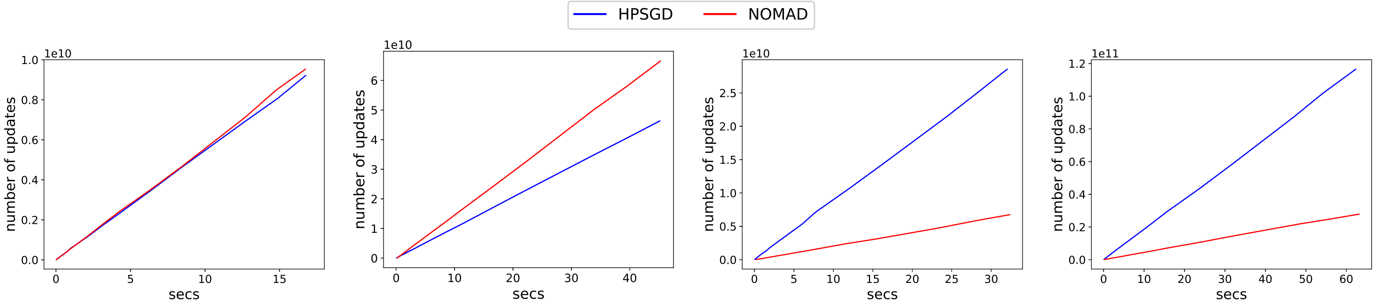
Chin, W.-S., Zhuang, Y., Juan, Y.-C., & Lin, C.-J. (2015). A fast parallel stochastic gradient method for matrix factorization in shared memory systems. *ACM Trans. Intell. Syst. Technol.*, *6*. doi:`10.1145/2668133`.

Chorobura, F., Lupu, D., & Necoara, I. (2022). Coordinate projected gradient descent minimization and its application to orthogonal nonnegative matrix factorization. In *2022 IEEE 61st Conference on Decision and Control (CDC)* (pp. 6929–6934). doi:`10.1109/CDC51059.2022.9992996`.

Dror, G., Koenigstein, N., Koren, Y., & Weimer, M. (2012). The yahoo! music dataset and kdd-cup'11. In *Proceedings of KDD Cup 2011* (pp. 3–18).

Elahi, F., Fazlali, M., Malazi, H. T., & Elahi, M. (2022). Parallel fractional stochastic gradient descent with adaptive learning for recommender systems. *IEEE Transactions on Parallel and Distributed Systems*, (pp. 1–14). doi:`10.1109/TPDS.2022.3185212`.

Gemulla, R., Haas, P. J., & Sismanis, J. (2015). Systems and methods for large-scale randomized optimization for problems with decomposable loss functions. US Patent 8,983,879.



(a) `Movielens`, #machines=16

(b) `Netflix`, #machines=32

(c) `Yahoo-medium`, #machines=32

(d) `Yahoo-large`, #machines=64

Figure 17: Throughput comparison of HPSGD and NOMAD as a function time for each dataset.

Gemulla, R., Nijkamp, E., Haas, P. J., & Sismanis, Y. (2011). Large-scale matrix factorization with distributed stochastic gradient descent. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* KDD '11 (p. 69–77). doi:`10.1145/2020408.2020426`.

Glorot, X., & Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics* (pp. 249–256). Chia Laguna Resort, Sardinia, Italy: PMLR volume 9. URL: `https://proceedings.mlr.press/v9/glorot10a.html`.

Harper, F. M., & Konstan, J. A. (2015). The movielens datasets: History and context. *ACM Trans. Interact. Intell. Syst.*, *5*. doi:`10.1145/2827872`.

Huang, Y., Yin, Y., Liu, Y., He, S., Bai, Y., & Li, R. (2021). A novel multi-CPU/GPU collaborative computing framework for SGD-based matrix factorization. In *50th International Conference on Parallel Processing* ICPP 2021. doi:`10.1145/3472456.3472520`.

Khan, Z. A., Chaudhary, N. I., & Raja, M. A. Z. (2022a). Generalized fractional strategy for recommender systems with chaotic ratings behavior. *Chaos, Solitons & Fractals*, *160*, 112204. doi:`10.1016/j.chaos.2022.112204`.

Khan, Z. A., Chaudhary, N. I., & Zubair, S. (2019). Fractional stochastic gradient descent for recommender systems. *Electronic Markets*, *29*, 275–285. doi:`10.1007/s12525-018-0297-2`.

Khan, Z. A., Raja, M. A. Z., Chaudhary, N. I., Mehmood, K., & He, Y. (2022b). MISGD: Moving-information-based stochastic gradient descent paradigm for personalized fuzzy recommender systems. *International Journal of Fuzzy Systems*, *24*, 686–712. doi:`10.1007/s40815-021-01177-9`.

Khan, Z. A., Zubair, S., Chaudhary, N. I., Raja, M. A. Z., Khan, F. A., & Dedovic, N. (2020). Design of normalized fractional sgd computing paradigm for recommender systems. *Neural Computing and Applications*, *32*, 10245–10262. doi:`10.1007/s00521-019-04562-6`.

Koren, Y., Bell, R., & Volinsky, C. (2009). Matrix factorization techniques for recommender systems. *Computer*, *42*, 30–37. doi:`10.1109/MC.2009.263`.

Lee, D., Oh, J., Faloutsos, C., Kim, B., & Yu, H. (2018). Disk-based matrix completion for memory limited devices. In *Proceedings of the 27th ACM International Conference on Information and Knowledge Management* CIKM '18 (p. 1093–1102). doi:`10.1145/3269206.3271685`.

Li, H., Li, K., An, J., & Li, K. (2018). MSGD: A novel matrix factorization approach for large-scale collaborative filtering recommender systems on GPUs. *IEEE Transactions on Parallel and Distributed Systems*, *29*, 1530–1544. doi:`10.1109/TPDS.2017.2718515`.

Linden, G., Smith, B., & York, J. (2003). Amazon.com recommendations: item-to-item collaborative filtering. *IEEE Internet Computing*, *7*, 76–80. doi:`10.1109/MIC.2003.1167344`.

Luo, X., Liu, H., Gou, G., Xia, Y., & Zhu, Q. (2012). A parallel matrix factorization based recommender by alternating stochastic gradient decent. *Engineering Applications of Artificial Intelligence*, *25*, 1403–1412. doi:`10.1016/j.engappai.2011.10.011`.

Luo, X., Liu, Z., Jin, L., Zhou, Y., & Zhou, M. (2022). Symmetric nonnegative matrix factorization-based community detection models and their convergence analysis. *IEEE Transactions on Neural Networks and Learning Systems*, *33*, 1203–1215. doi:`10.1109/TNNLS.2020.3041360`.

Luo, X., Shang, M., & Li, S. (2016). Efficient extraction of non-negative latent factors from high-dimensional and sparse matrices in industrial applications. In *2016 IEEE 16th International Conference on Data Mining (ICDM)* (pp. 311–319). doi:`10.1109/ICDM.2016.0042`.

Luo, X., Wang, Z., & Shang, M. (2021). An instance-frequency-weighted regularization scheme for non-negative latent factor analysis on high-dimensional and sparse data. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, *51*, 3522–3532. doi:`10.1109/TSMC.2019.2930525`.

Luo, X., Xia, Y., & Zhu, Q. (2013). Applying the learning rate adaptation to the matrix factorization based collaborative filtering. *Knowledge-Based Systems*, *37*, 154–164. doi:`10.1016/j.knosys.2012.07.016`.

Makari, F., Teflioudi, C., Gemulla, R., Haas, P., & Sismanis, Y. (2015). Shared-memory and shared-nothing stochastic gradient descent algorithms for matrix completion. *Knowledge and Information Systems*, *42*, 493–523. doi:`10.1007/s10115-013-0718-7`.

Matsushima, S., Yun, H., Zhang, X., & Vishwanathan, S. V. N. (2017). Distributed stochastic optimization of regularized risk via saddle-point problem. In *Machine Learning and Knowledge Discovery in Databases* (pp. 460–476). Cham: Springer International Publishing.

Mongia, A., & Majumdar, A. (2021). Matrix completion on learnt graphs: Application to collaborative filtering. *Expert Systems with Applications*, *185*, 115652. doi:https://doi.org/10.1016/j.eswa.2021.115652.

Oh, J., Han, W.-S., Yu, H., & Jiang, X. (2015). Fast and robust parallel sgd matrix factorization. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* KDD '15 (p. 865–874). doi:10.1145/2783258.2783322.

Pilászy, I., Zibriczky, D., & Tikk, D. (2010). Fast als-based matrix factorization for explicit and implicit feedback datasets. In *Proceedings of the Fourth ACM Conference on Recommender Systems* RecSys '10 (p. 71–78). doi:10.1145/1864708.1864726.

Priyati, A., Laksito, A. D., & Sismoro, H. (2022). The comparison study of matrix factorization on collaborative filtering recommender system. In *2022 5th International Conference on Information and Communications Technology (ICOIACT)* (pp. 177–182). doi:10.1109/ICOIACT55506.2022.9972018.

Qin, W., & Luo, X. (2022). An asynchronously alternative stochastic gradient descent algorithm for efficiently parallel latent feature analysis on shared-memory. In *2022 IEEE International Conference on Knowledge Graph (ICKG)* (pp. 217–224). doi:10.1109/ICKG55886.2022.00035.

Ramlatchan, A., Yang, M., Liu, Q., Li, M., Wang, J., & Li, Y. (2018). A survey of matrix completion methods for recommendation systems. *Big Data Mining and Analytics*, *1*, 308–323. doi:10.26599/BDMA.2018.9020008.

Recht, B., & Ré, C. (2013). Parallel stochastic gradient algorithms for large-scale matrix completion. *Mathematical Programming Computation*, *5*, 201–226. doi:10.1007/s12532-013-0053-8.

Recht, B., Re, C., Wright, S., & Niu, F. (2011). Hogwild!: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in Neural Information Processing Systems* (pp. 693–701). Curran Associates, Inc. volume 24. URL: https://proceedings.neurips.cc/paper/2011/file/218a0aefd1d1a4be65601cc6ddc1520e-Paper.pdf.

Shi, X., He, Q., Luo, X., Bai, Y., & Shang, M. (2022). Large-scale and scalable latent factor analysis via distributed alternative stochastic gradient descent for recommender systems. *IEEE Transactions on Big Data*, *8*, 420–431. doi:10.1109/TBDATA.2020.2973141.

Si, T. N., Van Hung, T., Ngoc, D. V., & Le, Q. N. (2022). Using stochastic gradient descent on parallel recommender system with stream data. In *2022 IEEE/ACIS 7th International Conference on Big Data, Cloud Computing, and Data Science (BCD)* (pp. 88–93). doi:10.1109/BCD54882.2022.9900664.

Singh, N., Zhang, Z., Wu, X., Zhang, N., Zhang, S., & Solomonik, E. (2022). Distributed-memory tensor completion for generalized loss functions in python using new sparse tensor kernels. *Journal of Parallel and Distributed Computing*, *169*, 269–285. doi:10.1016/j.jpdc.2022.07.005.

Suri, S., & Vassilvitskii, S. (2011). Counting triangles and the curse of the last reducer. In *Proceedings of the 20th International Conference on World Wide Web* WWW '11 (p. 607–614). doi:10.1145/1963405.1963491.

Takács, G., Pilászy, I., Németh, B., & Tikk, D. (2009). Scalable collaborative filtering approaches for large recommender systems. *The Journal of Machine Learning Research*, *10*, 623–656.

Teflioudi, C., Makari, F., & Gemulla, R. (2012). Distributed matrix completion. In *2012 IEEE 12th International Conference on Data Mining* (pp. 655–664). doi:10.1109/ICDM.2012.120.

Wu, D., He, Y., Luo, X., & Zhou, M. (2022). A latent factor analysis-based approach to online sparse streaming feature selection. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, *52*, 6744–6758. doi:10.1109/TSMC.2021.3096065.

Wu, Z., Luo, Y., Lu, K., & Wang, X. (2018). Parallelizing stochastic gradient descent with hardware transactional memory for matrix factorization. In *2018 3rd International Conference on Information Systems Engineering (ICISE)* (pp. 118–121). doi:10.1109/ICISE.2018.00029.

Xie, X., Tan, W., Fong, L. L., & Liang, Y. (2017). CuMF_SGD: Parallelized stochastic gradient descent for matrix factorization on GPUs. In *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing* HPDC '17 (p. 79–92). doi:10.1145/3078597.3078602.

Yu, H.-F., Hsieh, C.-J., Si, S., & Dhillon, I. (2012). Scalable coordinate descent approaches to parallel matrix factorization for recommender systems. In *2012 IEEE 12th International Conference on Data Mining* (pp. 765–774). doi:10.1109/ICDM.2012.168.

Yu, H.-F., Hsieh, C.-J., Yun, H., Vishwanathan, S., & Dhillon, I. (2016). Nomadic computing for big data analytics. *Computer*, *49*, 52–60. doi:10.1109/MC.2016.116.

Yu, Y., Wen, D., Zhang, Y., Wang, X., Zhang, W., & Lin, X. (2021). Efficient matrix factorization on heterogeneous CPU-GPU systems. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)* (pp. 1871–1876). doi:10.1109/ICDE51399.2021.00169.

Yun, H., Yu, H.-F., Hsieh, C.-J., Vishwanathan, S. V. N., & Dhillon, I. (2014). NOMAD: Non-locking, stochastic multi-machine algorithm for asynchronous and decentralized matrix completion. *Proceedings of the VLDB Endowment*, *7*, 975–986. doi:10.14778/2732967.2732973.