Future Generation Computer Systems 146 (2023) 207–221

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Load balanced locality-aware parallel SGD on multicore architectures for latent factor based collaborative filtering

Selcuk Gulcan [a], Muhammet Mustafa Ozdal [b,1], Cevdet Aykanat [a,*]

[a] Computer Engineering Department, Bilkent University, Ankara, Turkey
[b] Facebook Inc., Menlo Park, CA, USA

ARTICLE INFO

Article history:
Received 2 August 2022
Received in revised form 10 February 2023
Accepted 7 April 2023
Available online 20 April 2023

Keywords:
Matrix completion
Recommendation system
Stochastic gradient descent
Shared memory parallel systems
Load balancing
Locality-aware scheduling

ABSTRACT

We investigate the parallelization of Stochastic Gradient Descent (SGD) for matrix completion on multicore architectures. We provide an experimental analysis of current SGD algorithms to find out their bottlenecks and limitations. Grid-based methods suffer from load imbalance among 2D blocks of the rating matrix, especially when datasets are skewed and sparse. Asynchronous methods, on the other hand, can face cache issues due to their memory access pattern. We propose bin-packing-based block balancing methods that are alternative to the recently proposed BaPa method. We then introduce Locality Aware SGD (LASGD), a grid-based asynchronous parallel SGD algorithm that efficiently utilizes cache by changing nonzero update sequence without affecting factor update order and carefully arranging latent factor matrices in the memory. Combined with our proposed load balancing methods, our experiments show that LASGD performs significantly better than alternative approaches in parallel shared-memory systems.

1. Introduction

Many real-life problems in various fields such as text processing [1], recommendation systems [2], bioinformatics [3] can be formulated as a sparse matrix completion problem to explain the relationship between two entities. Movie recommendation can be given as an example of a matrix completion problem, which aims to fill missing entries of a sparse matrix by using currently known values. Movie ratings given by users constitute a sparse matrix in which rows represent users and columns represent movies. The task is to predict missing scores of the rating matrix, thus have an idea on how likely a user will enjoy a particular movie and then recommend highly scored movies to the user. Collaborative filtering achieve this by considering other users' rating information to determine the rating values of an individual user.

Netflix competition [4–6] shows that low-rank matrix completion techniques, finding two low-rank P and Q matrices such that their product gives an approximation of nonzeros in the original rating matrix, are quite useful for this type of problems. Exact completion methods [7,8] are not effective due to the sparsity and size of real datasets. So, many optimization algorithms [4–6] have been proposed to tackle this problem, including alternating least squares (ALS), cyclic coordinate descent (CCD), and stochastic gradient descent (SGD). Being able to handle huge data and good convergence rates make gradient descent methods such as SGD superior. The SGD algorithm solves the problem by making noisy but quick updates to the P- and Q-matrix rows.

Improving the algorithm's execution time is important in both production and training. If we continue with the recommendation example, improving the execution time reduces the response time. The system can give recommendations to the customer faster with a better algorithm. Depending on the application, it can be critical. In the training step, the SGD procedure is run many times to find a good set of hyperparameters (hyperparameter tuning). In this algorithm, important hyperparameters are learning rate, regularization parameter, factor size, and P, Q weight initialization. The standard steps to find a good set of hyperparameters are (1) picking some hyperparameters randomly (e.g., factor size f, regularization parameter λ and learning rate γ) (2) running the algorithm to find P, Q matrices (model) (3) testing the model on a different set of data, known as the validation set. Therefore, the whole training process may take up days even if a single execution of the algorithm is completed in minutes. The SGD procedure deserves parallelization since reducing the execution time of a single execution instance improves the overall training time significantly.

The sequential nature of the algorithm and ever-increasing data led researchers to search for better parallel methods for SGD.

* Corresponding author.
E-mail addresses: selcuk.gulcan@bilkent.edu.tr (S. Gulcan), ozdal@fb.com, mustafa.ozdal@cs.bilkent.edu.tr (M.M. Ozdal), aykanat@cs.bilkent.edu.tr (C. Aykanat).

[1] This work was done while the author was a faculty member at Bilkent University.

https://doi.org/10.1016/j.future.2023.04.007
0167-739X/

Two main concerns of the parallelization of the SGD algorithm are task mapping and scheduling (i.e., mapping of tasks to different computing units) as well as load-balancing. Previous studies [9–14] explore new methods to tackle those challenges. Many of them focus their research on distributed-memory systems; however, many real datasets [4,15–17] can fit into the main memory and can be computed faster in shared-memory multicore systems because of low latency memory operations and fast access to recently modified data using caches. Shared memory studies offer various methods for improvements in accuracy and convergence, but they often lack quantitative analysis on underlying reasons for such improvements.

The main contributions of the paper are listed below:

- We analyzed and compared the state-of-the-art shared-memory SGD methods for matrix completion under the same experimental setup on real datasets to form a better understanding. Experimental findings show that the widely used grid-based methods suffer from load imbalance among 2D blocks of the rating matrix especially for skewed and sparse datasets and from memory underutilization due to the random access pattern in each 2D block.
- In order to address the load imbalance bottleneck, we propose two bin-packing-based load balancing algorithms that generate more balanced 2D blocks than the state-of-the-art load balancing methods proposed in the literature for matrix completion.
- In order to address the memory underutilization bottleneck, we propose a locality-aware task scheduling algorithm to utilize memory hierarchy better by carefully changing the rating matrix memory layout and nonzero update sequence without disturbing the stochastic nature of the algorithm.
- We perform extensive experiments on a wide range of matrix completion datasets and show the validity of the proposed load balancing and locality-aware scheduling

The proposed methods do not contradict each other, so load balancing algorithms and LASGD can be used together to address both load balancing and cache utilization problems simultaneously.

The organization of the paper is as follows: Section 2 gives a background on SGD for solving the matrix completion problem. Literature survey and discussions are given in Section 3. Table 1 gives the notation used throughout the paper. The proposed load balancing algorithms for 2D-grid partitioning and the proposed LASGD algorithm are respectively described in Sections 4 and 5. Experimental setup and results are given in Section 6. Finally, Section 7 concludes the paper.

2. Matrix completion with SGD

Given a sparse $m \times n$ matrix $V = (v_{ij})$, which holds the observed entries of a matrix $\tilde{V} = \tilde{v}_{ij}$, the matrix completion problem is finding an estimation matrix $\hat{V} = (\hat{v}_{ij})$ that is as close as possible to the original matrix $\tilde{V}$. The distance between the matrices is defined as the Frobenius norm of the difference between the original matrix and the estimation matrix, which leads to the following minimization:

$$minimize_{\hat{v}_{ij}} \sum_{\tilde{v}_{ij} \in \tilde{V}} (\tilde{v}_{ij} - \hat{v}_{ij})^2.$$

The challenge is to find a model that captures the patterns in the original matrix $\tilde{V}$ by using only a few entries of the matrix, stored in V.

In the context of recommender systems, m denotes the number of users, n denotes the number of items, $\tilde{v}_{ij}$ is a true rating value of the corresponding user–item pair, v_{ij} is the known rating value, and $\hat{v}_{ij}$ is the estimated rating value in reconstructed matrix $\hat{V}$.

Matrix completion-based collaborative filtering methods solve this problem by finding two matrices, P and Q, such that their matrix multiplication gives reconstructed matrix, $\hat{V}$. A row of $P_{n\times f}$ and a row of $Q_{m\times f}$ respectively represent a user and an item as a latent factor vector of size f. The latent factor size, f, is much smaller than n and m and tuned to get a better model. With this setup, any rating $\tilde{v}_{ij}$ in the matrix can be estimated by calculating the dot product of its corresponding user vector, $\mathbf{p}_i$, and item vector, $\mathbf{q}_j^T$ (see Fig. 1).

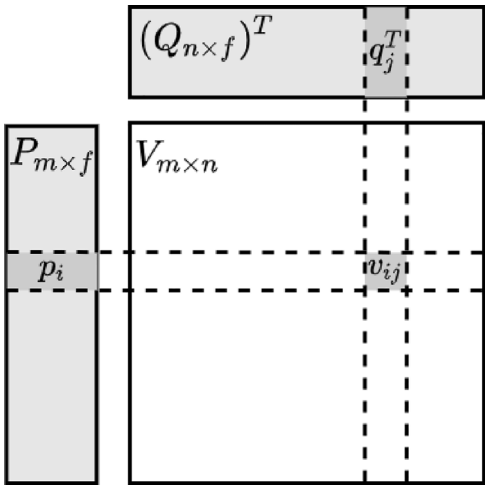

Fig. 1. Rating matrix V and latent factor matrices P and Q.

Table 1
The Notation table.

Symbol	Description
V	Rating matrix (Also denotes the set of nonzero ratings)
m, n	Number of rows and columns of matrix V
v_{ij}	Nonzero entry at row i and column j
$\hat{v}_{ij}$	Estimate for v_{ij}
r_i, c_j	ith row and jth column of matrix V. Also denotes the set of nonzeros in r_i and c_j
P, Q	Latent factor matrices
$\mathbf{p}_i$, $\mathbf{q}_j$	ith and jth rows of P- and Q-matrix
f	Factor size
T	Number of threads
γ	Learning rate
λ	Regularization parameter
$\Pi = \{V_1 \ldots V_T\}$	T-way nonzero partitioning of matrix V
V_t	Set of nonzeros assigned to thread t
$\langle \Pi \rangle$	Set of local nonzero update sequence
$\langle V \rangle$	Given nonzero update sequence for serial algorithm
$\langle V^* \rangle$	A nonzero update sequence for serial algorithm
$\langle V_t \rangle$	Nonzero update sequence for thread t
R_α, C_β	αth row slice, βth column slice
$V_{\alpha,\beta}$	2D block at the intersection of R_α and C_β
$c_{j,\alpha}$	Set of nonzeros in column-j that reside in R_α
B^r_α, B^c_β	Bin for row-slice α, bin for column-slice β
$B_{\alpha,\beta}$	Bin for 2D-block $V_{\alpha,\beta}$
$\lvert \cdot \rvert$	Number of nonzeros in the respective set

Gradient descent methods are popular iterative optimization techniques for finding a local minimum of multivariate differentiable functions. At each step, variables are updated proportionally to their gradients. The learning rate hyperparameter seen in Algorithm 1, α, controls how much the variables are changed at each step.

The loss function for P and Q matrices in matrix completion-based collaborative filtering methods can be written as

$$error = \sum_{i,j \in V} (v_{ij} - \mathbf{p}_i \mathbf{q}_j^T)^2 + \beta(\|P\|^2 + \|Q\|^2).$$

Here, β is the regularization parameter for L2 normalization added to avoid overfitting problems. The main difference between SGD and other gradient methods is that SGD updates latent factor matrices without computing the whole gradient. At each update step, the next nonzero in the given nonzero update sequence, $\langle V \rangle$, is fetched, and P- and Q-matrix rows are updated for this nonzero only as shown in Algorithm 1. Each iteration computes the error value through an inner product of $\mathbf{p}_i$ and $\mathbf{q}_j$ and then uses this error value for updating both $\mathbf{p}_i$ and $\mathbf{q}_j$ vectors through AXPY-type of operations.

Since other nonzero elements are not taken into account in a single update, updates of SGD may not be directed towards the point of convergence. However the performance gain of quick updates can surpass the error caused by noisy gradient [18]. For this reason, SGD converges faster than full gradient approaches for matrix completion. This is especially true for web-scale matrices where the number of training examples is huge but training time is limited. So SGD can give better results due to the fact that it can do many more updates in the given time.

Algorithm 1: SGD Algorithm

Input: $V_{m \times n}$, $\langle V \rangle$, f, α, β, *iterCount*
Initialize $P_{m \times f}$ and $Q_{n \times f}$;
for $c \leftarrow 0$ **to** *iterCount* **do**
  forall v_{ij} *of* V *in* $\langle V \rangle$ *order* **do**
    $e_{ij} \leftarrow v_{ij} - \mathbf{p}_i \mathbf{q}_j^T$;
    $\mathbf{p_i} \leftarrow \mathbf{p}_i + \alpha(e_{ij}\mathbf{q}_j - \beta \mathbf{p}_i)$;
    $\mathbf{q}_j \leftarrow \mathbf{q}_j + \alpha(e_{ij}\mathbf{p}_i - \beta \mathbf{q}_j)$;

One crucial property of the algorithm is the randomized order in the nonzero update sequence. It is a theoretical constraint [19] in order for the model to converge. Non-shuffled training points or sequences sorted in a particular way may cause similar gradient updates between subsequent iterations, and this situation may result in slower convergence or convergence to an underwhelming local minimum [19]. The study [10] shows that the following two properties should be ensured to get decent practical results:

- The selected nonzeros in iterations should cover a significant portion of the whole rating set.
- The sequence of nonzero updates should be randomized.

3. Literature survey and discussions

In this section, parallel SGD schemes proposed in the literature are analyzed. Each subsection follows a similar format; First, a brief introduction to the method is given. Second, the problems the method tries to solve are explained. Third, the main drawbacks of the method are shown.

3.1. Simple-parallel SGD (spSGD)

The main problem of threads concurrently updating the same P- and Q-matrix rows is data dependencies. When a thread tries to operate on a P- or Q-matrix row that is being processed by another thread, the following problems may occur depending on the execution order of instructions: A thread may disregard the latest value of the P- or Q-matrix row in its gradient calculation and operates on the old data. A thread may overwrite the P- or Q-matrix row. Hence, the previous update on the corresponding row is wasted.

Dependency problems cause threads to do extra computations to reach convergence. To avoid this, spSGD locks the rows of the P and Q matrices before doing the read or write operation and unlock them after it completes updating them as shown in Algorithm 2. Because each V-matrix nonzero is processed in a critical section, no data dependency problem can occur in this method.

The serializability of a parallel SGD algorithm is an important feature since it refers to the fact that there exists a sequential SGD execution for which parallel SGD algorithms show convergence to the same solution with the same convergence rate. We discuss the serializability of the spSGD algorithm in the following two paragraphs.

For a given parallel SGD instance, let $\Pi = \{V_1, V_2 \dots V_T\}$ denote the partitioning of the nonzeros of the rating matrix among T threads. Here V_t denotes the subset of nonzeros assigned to thread t for processing. The parts of Π ($V_t \in \Pi$) are mutually nonzero disjoint and exhaustive.

For a given nonzero partition Π, let $\langle \Pi \rangle = \{\langle V_1 \rangle, \langle V_2 \rangle \dots \langle V_T \rangle\}$ denote the set of local nonzero update sequences of threads. That is, $\langle V_t \rangle$ denotes the local update sequence of the nonzeros in V_t. spSGD ensures the serializability of the algorithm. That is, it ensures the existence of a sequential nonzero update sequence $\langle V^* \rangle$ so that serial SGD using $\langle V^* \rangle$ and spSGD using $\langle \Pi \rangle$ produce the same P and Q matrices. Note that although the overall update sequence $\langle V^* \rangle$ depends on the local update sequence set $\langle \Pi \rangle$, it cannot be determined apriori to the execution of the parallel algorithm. That is different parallel runs for a given $\langle \Pi \rangle$ may lead to different $\langle V^* \rangle$'s.

Algorithm 2: spSGD Algorithm

Input: $\langle \Pi \rangle$, Π, f, α, β, *iterCount*
Initialize $P_{m \times f}$ and $Q_{n \times f}$;
for *each thread t in parallel* **do**
  for $c \leftarrow 0$ **to** *iterCount* **do**
    forall v_{ij} *of* V_t *in* $\langle V_t \rangle$ *order* **do**
      Lock P- and Q-matrix rows, $\mathbf{p}_i$ and $\mathbf{q}_j$;
      Update P- and Q-matrix rows, $\mathbf{p}_i$ and $\mathbf{q}_j$;
      Unlock P- and Q-matrix rows, $\mathbf{p}_i$ and $\mathbf{q}_j$;

Although this naive parallelization leads to an inefficient algorithm, it provides some insights when the performances of the methods discussed in the following sections are analyzed against the performance of spSGD. In the following two paragraphs, we investigate the bottlenecks and potential improvements by using Intel's vTune profiling tool [20].

Table 2 reports average total spin time and overhead time as percentage values. The spin time is defined as the time spent waiting for another thread to release the synchronization object, and the overhead time is the time spent on acquiring an available lock object or releasing the lock object. Fig. 2 shows the lock wait percentage of a single nonzero on a single iteration. It is calculated as follows: the wait count value is incremented by one whenever a thread tries to access a P- or Q-matrix row when

Table 2
Percentage of CPU time spent on spin and overhead.

Dataset	Number of threads					
	2	4	8	16	28	56
amazon_items	87.3	87.3	87.4	87.2	87.1	87.7
amazon_books	91.6	91.4	91.6	91.9	91.2	91.3
amazon_clothings	92.3	92.8	92.1	91.9	91.8	92.7
amazon_electronics	90.1	88.8	90.1	90.4	89.7	91.1
amazon_movies	87.0	86.0	87.6	89.3	88.8	88.0
lastfm	82.9	85.8	86.9	86.9	87.2	83.4
movielens_20 m	80.9	84.2	85.9	86.2	85.8	83.7
movielens_latest	81.4	82.5	83.4	89.2	85.5	85.7
netflix	76.4	81.8	83.7	84.5	84.6	80.0
yahoo_music	83.8	85.7	86.2	86.1	85.5	81.9

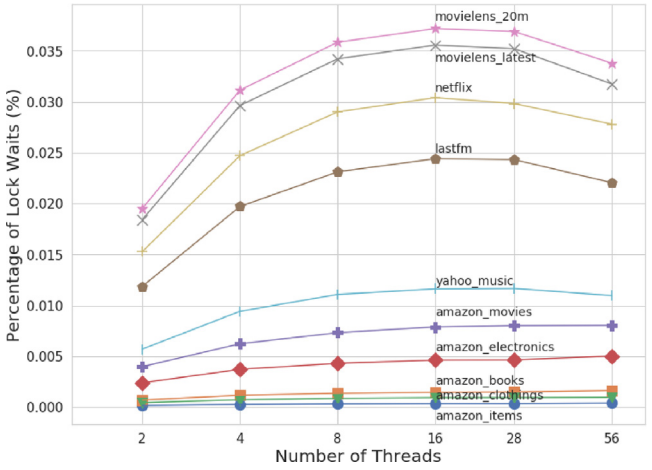

Fig. 2. Wait percentage for a nonzero update.

it is locked by another thread. At the end of execution, the wait count value is divided by the total number of nonzero updates to obtain normalized results. The total number of nonzero updates corresponds to the number of epochs multiplied by the number of nonzeros.

As seen in Fig. 2, lock waits are rare and vary between 0% and 0.04% depending on the density of the dataset. Around 75%–90% of CPU time is wasted to prevent data dependency problems that occur less than 0.04% of the time so we can gain significant performance if we can somehow avoid row synchronization on P and Q matrices.

3.2. Hogwild: Asynchronous SGD

Hogwild [11] states that possible data dependency problems which occur in parallel SGD are rare, and SGD can be implemented without any locking. Hogwild theoretically proves that even though data dependencies degrade the convergence rate, the convergence is still guaranteed given the assumption that the data matrix is sufficiently sparse. As Algorithm 3 shows, the only difference between the Hogwild and spSGD is the absence of locks on P- and Q-matrix rows.

Algorithm 3: Hogwild Algorithm

Input: $\langle \Pi \rangle$, Π, f, α, β, *iterCount*
Initialize $P_{m\times f}$ and $Q_{n\times f}$;
for *each thread t parallelly* **do**
  for $c \leftarrow 0$ **to** *iterCount* **do**
    forall v_{ij} *of* V_t *in* $\langle V_t \rangle$ *order* **do**
      Update P- and Q-matrix rows, p_i and q_j;

The problem with this approach is cache underutilization due to cache coherence between private caches. When a nonzero is processed in the sequential algorithm, its corresponding P- and Q-matrix rows are fetched from the memory and cached. The thread can read and write them on the cache quickly without accessing long latency DRAM until those rows are evicted since there no other threads trying to access those rows. However, this is not always possible in multicore systems because of the possibility of multiple cores trying to access/update the same P- or Q-matrix rows concurrently.

In multicore systems, cores have private caches, so there are cache coherence protocols to preserve data consistency across all cores. When a cache line is modified, lines having the same address in the cache(s) of other cores are invalidated through a snooping mechanism, so other cores need to fetch them from the DRAM or the core having the modified value. Since each thread has an equal chance to access a particular row or column in the Hogwild algorithm, the chance that a cache line is invalidated increases as more threads are used. Although this data migration does not affect the algorithm in terms of correctness, it degrades the performance since accessing P- and Q-matrix rows take longer times.

3.3. 2D grid partitioning

2D grid partitioning eliminates data migrations by dividing the rating matrix into independent 2D blocks. A 2D block is the intersection of a row slice and a column slice on the rating matrix. Two 2D blocks are said to be independent if they do not share any V-matrix rows or columns. Although this scheme is originally proposed for distributed-memory SGD in [10], we utilize it for shared-memory parallelization. A set of independent 2D blocks concurrently updated by different computing units is called a stratum. Since nonzeros of different 2D blocks in a single stratum do not incur updating the same P- and Q-matrix rows, the threads can operate on them without any data conflict.

The rating matrix should be divided into at least $T\times T$ blocks to ensure that all threads can work on independent blocks for a system with T threads. Besides the $T\times T$ 2D grid partitioning in [10], there are $(T+1)\times(T+1)$ [12], $T\times 2T$ [21], and $T\times dnT$ [13] partitioning schemes in the literature.

In this work, we consider $T\times T$ partitioning, where each row slice is assigned to a distinct thread. Fig. 3 displays a sample 4×4 2D grid partition with two different block scheduling for a system with 4 threads/cores. In the figure, letters show the thread assignments so that the first, second, third, and fourth-row slices are processed by threads A, B, C, and D, respectively. The numbers show different strata so that $\{A_k, B_k, C_k, D_k\}$ denotes the set of blocks constituting stratum k for $k = 1, 2, 3, 4$. At each epoch, the 2D blocks constituting stratum k are processed concurrently in the kth time period/subepoch.

The motivation behind row-wise partitioning of the V matrix is to exploit cache locality if P-matrix rows corresponding to nonzeros in the same row slice can fit the private cache of the core. That is, a thread will update the same P-matrix rows after it completes updating each block. However, it is not a constraint to mitigate the data migration problem. As long as cores work on independent blocks, any partitioning and block scheduling can be used [10].

Each thread in a stratum needs to wait for other threads after processing a block to avoid updates on the same P- and Q-matrix rows. This type of scheduling will be called *synchronous block scheduling*. On the other hand, Hogwild shows that synchronization points can be removed without any convergence concerns.

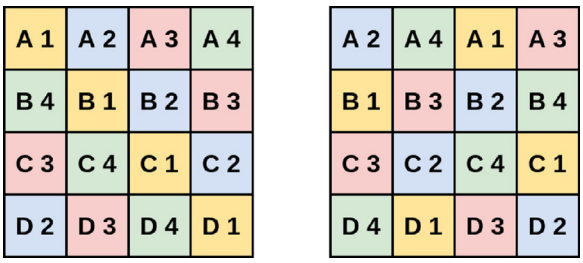

Fig. 3. Two different block scheduling for 4 × 4 partitioning.

This kind of scheduling in which threads continue operating on their blocks of the next stratum will be called *asynchronous block scheduling*. Our experiments show that there is no significant difference between these approaches in terms of throughput, convergence rate, and accuracy if nonzero distribution among 2D blocks is sufficiently balanced. Some studies [12,22] make this block assignment decision in the run time referred to as *standalone scheduling*. In this dynamic block scheduling, when a thread finishes updating all nonzeros in a block, a new block is assigned to the thread by finding an independent block from the set of blocks that are not being processed by other threads.

One of the key factors in the performance of the method utilizing 2D grid partitioning is the imbalance in the number of nonzeros among different blocks. Different problems may arise depending on the existing block scheduling mechanisms if nonzeros are distributed unevenly among blocks.

Synchronous block scheduling: Since threads need to wait for the slowest thread to complete its update, thread utilization greatly suffers from uneven nonzero partition.

Asynchronous block scheduling: Working on the same P- or Q-matrix rows creates data migrations that slow the execution due to cache coherence delays.

Standalone scheduling: Blocks with fewer ratings are updated more often than denser blocks because they will be updated quickly and become available to assign. This situation hurts the convergence performance because some portion of data is rarely used in iterative learning.

3.4. Load balancing in 2D grid partitioning

A common approach is to adopt random 2D grid partitioning. In this method, the rows and columns of V are randomly permuted. Then a uniform 2D grid partition is imposed on the permuted matrix so that the number of rows and columns assigned to each row slice and column slice differ by at most 1. However, random partitioning scales poorly with the increasing number of threads. Also, nonzero distribution among rows and columns in real datasets usually follows the power law. It requires more intelligent techniques to attain balanced blocks on scale-free datasets when the number of threads is large.

In order to obtain better-balanced blocks, [23] proposed a greedy partitioning algorithm (BaPa) that balances row slices and column slices separately. The algorithm assigns rows to the first available row slice in arbitrary order. When the number of nonzeros in the row slice reaches a threshold that can attain fairly balanced row slices, the row slice is considered full. The procedure continues until all rows are assigned. Columns are similarly assigned to column slices. Then, 2D blocks are created by meshing the row slices and column slices.

3.5. Other related studies

[24,25] proposed parallel SGD algorithms for GPUs. [26] proposed a lock-based SGD method for transactional memory in which data conflicts are handled in the hardware level. [27] applied Hogwild and DSGD ideas on the pairwise learning to rank problems. In this problem, instead of explicit ratings, the model utilizes the ranking of items to complete the matrix. [28] parallelized SGD algorithm on streaming data. [29,30] proposed parallel disk-based solutions for datasets that are too large for the main memory.

The articles we reviewed so far are focused on the speeding up the epoch processing time by parallelizing the SGD procedure, which reduces the total execution time eventually. Another way to reduce the total execution time is to make more effective updates so that the model converges in less number of epochs. In this regard, Momentum SGD [31] can be considered as a pioneer work in the context of recommender systems. In Momentum SGD, incorporating previous updates to P and Q matrices into the current update increases the convergence rate of the model. Following this idea, NF-SGD [32] improves the performance considerably by tuning the learning rate parameter adaptively. MISGD [33] utilizes the aggregate of past and current update information in a defined length of interactions efficiently in order to improve the convergence rate of SGD. GFSGD [34] adds fractional order gradient term to the update rules in order to improve the effect of previous updates by capturing rating history more effectively. Although this study uses basic SGD update rules without any momentum for the sake of simplicity, we should point out that our proposed methods and the mentioned solutions can be combined to get even more powerful models.

4. Proposed load balancing methods

In this section, we propose two load balancing algorithms for 2D grid partitioning. The proposed algorithms are based on bin-packing (BP) with a fixed number of bins [35]. These BP-based algorithms share the following common features:

- They apply a two-phase approach, where rows are assigned to row slices in one phase and columns are assigned to column slices in the other phase.
- Rows and columns are assigned to bins in decreasing order of their degrees.

The degree of a row/column refers to the number of nonzeros in the row/column. The motivation behind this row/column-to-bin assignment order is as follows: The relatively large number of sparse rows/columns have the potential of correcting the imbalance incurred by the assignment of the relatively small number of dense rows/columns at the initial steps.

These algorithms differ in the best-fit assignment heuristic utilized for row/column assignments in different phases.

4.1. Independent row and column partitioning (BP_{r+c})

In Algorithm 5, rows and columns are respectively assigned to row and column slices independently in two phases. For this purpose, in each phase, we maintain T bins, which correspond to either row or column slices. That is, in the row assignment phase, bin B^r_α represents row slice R_α, whereas in the column assignment phase bin B^c_β represents column slice C_β. In each phase, the weights of the bins are initialized to zero. Then in the row/column assignment phase, the best-fit heuristic utilized is assigning a row/column to the bin with the least weight as shown in Algorithm 4. After each assignment, the weight of

the respective bin is incremented by the degree of the assigned row/column.

The assignment of a V-matrix nonzero v_{ij} to a 2D block is induced by the assignment of row r_i and column c_j to the bins in row and column assignment phases. That is, the assignment of row-r_i to bin B^r_α and column c_j to bin B^c_β induces the assignment of nonzero v_{ij} to 2D block $V_{\alpha,\beta}$.

Algorithm 4: Decreasing order row/column assignment procedure (BP)

Input: $V_{m\times n}$, T
for $\alpha \leftarrow 1$ **to** T **do**
  $B^r_\alpha \leftarrow \emptyset$
Sort rows in decreasing order
for $i \leftarrow 1$ **to** m **do**
  Find min s.t. $|B^r_{min}| \leq |B^r_\alpha|$, $\alpha = 1, 2, \ldots m$
  $B^r_{min} \leftarrow B^r_{min} \cup \{r_i\}$
  $map(r_i) \leftarrow min$

Algorithm 5: BP_{r+c} Algorithm

Input: $V_{m\times n}$, T
Assign rows with BP procedure
Assign columns with BP procedure
$V_{\alpha,\beta} \leftarrow \emptyset$
forall $v_{ij} \in V$ **do**
  $\alpha \leftarrow map(r_i)$
  $\beta \leftarrow map(c_j)$
  $V_{\alpha,\beta} \leftarrow V_{\alpha,\beta} \cup \{v_{ij}\}$

4.2. First row then column partitioning ($BP_{r\rightarrow c}$)

BP_{r+c} does not utilize the row-to-slice assignment information obtained in one phase for the column-to-slice assignments in the other phase or vice versa. We propose a BP-based algorithm $BP_{r\rightarrow c}$ which utilizes the assignment information obtained in the first phase for the assignment in the second phase.

The first phase of $BP_{r\rightarrow c}$ is exactly the same as the row assignment phase of BP_{r+c}. The output of the first phase is the row-to-slice assignment shown by the function $map()$ where $map(r_i)$ denotes the row-slice containing row-r_i.

We propose a novel best-fit heuristic for the column assignment phase by defining a sum-of-squares cost of a given 2D nonzero-to-bin partition Π:

$$Cost(\Pi) = \sum_{\alpha=1}^{T}\sum_{\beta=1}^{T} |B_{\alpha,\beta}|^2. \tag{1}$$

Here $|B_{\alpha,\beta}|$ denotes the current load of bin $B_{\alpha,\beta}$ in terms of number of nonzeros. The sum-of-squares term enables the minimization of the cost function to encode minimizing the load of the most heavily loaded bin of the 2D bin array.

According to the cost function in (1), the cost of assigning a column c_j to column slice C_β can computed as follows:

$$\begin{aligned} cost(c_j, C_\beta) &= Cost(\Pi_{new}) - Cost(\Pi_{cur}) \\ &= \sum_{\alpha=1}^{T}((|B_{\alpha,\beta}| + |c_{j,\alpha}|)^2) - \sum_{\alpha=1}^{T}|B_{\alpha,\beta}|^2 \\ &= \sum_{\alpha=1}^{T}\left(|c_{j,\alpha}|^2 + 2|B_{\alpha,\beta}|\cdot|c_{j,\alpha}|\right). \end{aligned} \tag{2}$$

Π_{cur} denotes the current nonzero-to-bin distribution. Π_{new} denotes the nonzero-to-bin distribution to be obtained by assigning c_j to C_β in Π_{cur}. So, $cost(c_j, C_\beta)$ shows the increase in the overall cost of the current nonzero-to-bin partition Π_{cur} to be incurred if we assign c_j to C_β. In (2), $|c_{j,\alpha}|$ denotes the number of nonzeros of column c_j in row block R_α, i.e.,

$$|c_{j,\alpha}| = |\{v_{ij} \in V : map(r_i) = R_\alpha\}|.$$

The details of $BP_{r\rightarrow c}$ is given in Algorithm 6. The algorithm starts with initially empty $T\times T$ bins. Then, the columns are considered for assignment to column slices in decreasing order of their degrees in a similar way with the general framework. Then at each assignment step, we consider assigning the current column to each of T column slices (C_β, for $\beta = 1\cdots T$) and then realize the assignment that incurs the smallest amount of increase according to (2).

Algorithm 6: $BP_{r\rightarrow c}$ Algorithm

Input: $V_{m\times n}$, T
Assign rows with BP procedure
for $\alpha \leftarrow 1$ **to** m **do**
  for $\beta \leftarrow 1$ **to** n **do**
    $B_{\alpha,\beta} \leftarrow \emptyset$
Sort columns in descending order
for $j \leftarrow 1$ **to** n **do**
  for $\beta \leftarrow 1$ **to** T **do**
    $Cost_{j,\beta} \leftarrow \sum_{\alpha=1}^{T}\left(|c_{j,\alpha}|^2 + 2|B_{\alpha,\beta}|\cdot|c_{j,\alpha}|\right)$
      ▷ computed according to the Eq. (2)
  Find min s.t. $Cost_{j,min} \leq Cost_{j,\beta}$, $\beta = 1, 2, \ldots T$
  for $\alpha \leftarrow 1$ **to** T **do**
    $B_{\alpha,min} \leftarrow B_{\alpha,min} \cup c_{j,min}$
  $map(c_j) \leftarrow min$
forall $v_{ij} \in V$ **do**
  $\alpha \leftarrow map(r_i)$
  $\beta \leftarrow map(c_j)$
  $V_{\alpha,\beta} \leftarrow V_{\alpha,\beta} \cup \{v_{ij}\}$

As mentioned earlier, the algorithm performs row assignment in the first phase and the column assignment in the second phase. However, it can be easily modified to perform column assignments first and then row assignments. We call this variant of the algorithm as $BP_{c\rightarrow r}$.

5. Locality Aware SGD (LASGD)

Two definitions are given to express the motivation of the proposed LASGD algorithm.

Definition 1 (*Valid Nonzero Update Sequence*)**.** *For a given nonzero update sequence, there can be different nonzero update sequences such that the update order of P- and Q-matrix rows will be the same as the original sequence. Each such sequence is defined as a Valid Nonzero Update Sequence.*

Therefore, any valid nonzero update sequence can be used in the SGD process instead of the original one. This is because latent factor matrices P and Q will be the same as the original update sequence produces since the order of updates on each P- and Q-matrix row is exactly the same.

Example 1. Fig. 4 shows a sample rating matrix with two rows labeled as x and y and two columns labeled as z and w. The original update sequence is given in the left part of the figure as $\langle v_{xz}, v_{yw}, v_{yz}\rangle$. First v_{xz}, then v_{yw} and finally v_{yz} is updated in the original sequence. The right part of the figure shows a different valid nonzero update sequence, $\langle v_{yw}, v_{xz}, v_{yz}\rangle$.

All update calculations on P- and Q-matrix rows are displayed in the bottom part of Fig. 4 to show that $\langle v_{xz}, v_{yw}, v_{yz}\rangle$ and

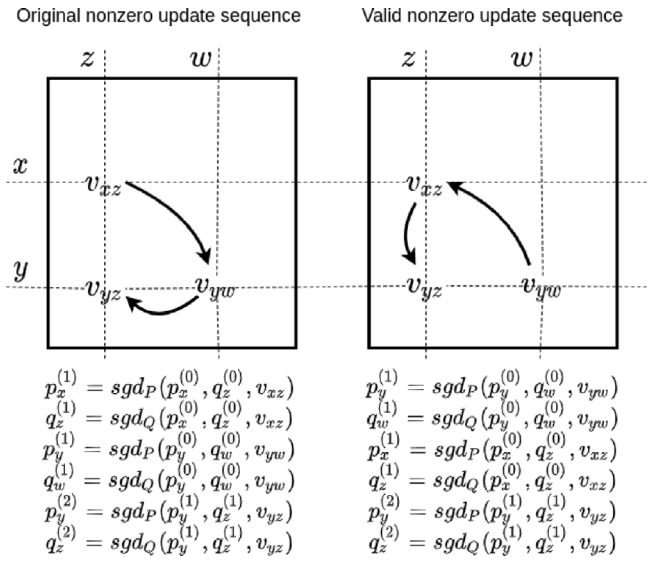

Fig. 4. The original SGD update sequence and a valid nonzero update sequence on a 2×2 sample matrix with three nonzeros.

$\langle v_{yw}, v_{xz}, v_{yz} \rangle$ update sequences modify P- and Q-matrix rows in exactly the same way. We encapsulate SGD calculation for P- and Q-matrix updates with functions called sgd_P and sgd_Q respectively, for the sake of clarity. *sgd* function takes a rating value and the corresponding P- and Q-matrix rows as arguments in order to compute a single rating update. In the calculations, $p_x^{(t)}$ shows the state of row x of P-matrix after the row is updated t times. Although P- and Q-matrix rows are updated in different orders, each P-matrix row, as well as each Q-matrix row, is updated in the same order. So, these latent matrix rows are exactly the same after all three nonzero updates. This shows that $\langle v_{yw}, v_{xz}, v_{yz} \rangle$ is a valid nonzero update sequence.

Definition 2 (*Nonzero Update Graph (NUG)*)**.** *For a given rating matrix V and nonzero update sequence $\langle V \rangle$, NUG is a directed graph, $G(V, \langle V \rangle) = (U, E)$, where each node $u_a \in U$ represents nonzero v_a. For a pair of nonzeros v_a and v_b, there is a row edge $(u_a, u_b) \in E$ only if the following 3 conditions are met:*

(i) *Nonzeros v_a and v_b reside in the same row.*
(ii) *v_a appears before v_b in the nonzero update sequence.*
(iii) *There is no nonzero v_c in the same row such that v_c is between v_a and v_b in the nonzero update sequence.*

The same construction logic applies to column edges as well.

A NUG is a directed acyclic graph. By the definition of NUG, any edge $(u_a, u_b) \in E$ refers to the fact that nonzero v_a is updated before nonzero v_b in the given update sequence. This property extends to any directed path by transitivity so that a path from node u_a to node u_b refers to the fact that v_a is updated before v_b. Since either v_a is before v_b or vice versa, either u_a is reachable from u_b or u_b is reachable from u_a in the NUG. This excludes the existence of any cycle in NUG. A NUG satisfies the following properties:

- Each node has at most two incoming edges.
- Each node has at most two outgoing edges.
- Two nodes sharing the same incoming neighbor cannot have an edge between them.

A NUG is a sparse graph as the total number of edges is upper bounded by two times the number of nonzeros.

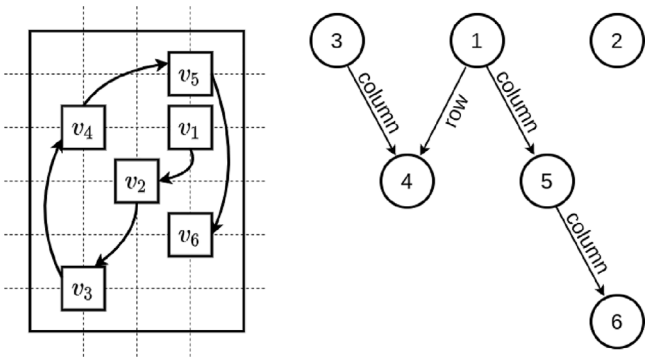

Fig. 5. Left: A sample 5×3 rating matrix with 6 nonzeros and the original update sequence $\langle v_1, v_2, v_3, v_4, v_5, v_6 \rangle$. Right: Respective NUG with 6 vertices and four edges.

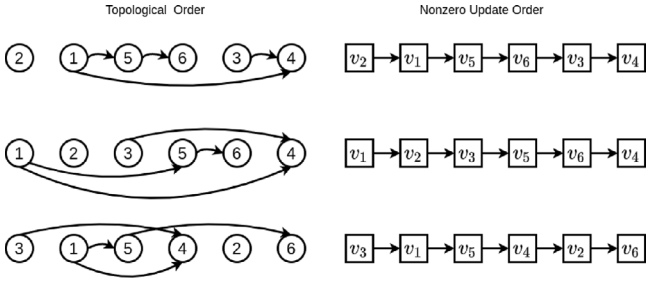

Fig. 6. Three possible valid nonzero update sequences for the sample NUG in Fig. 5.

Example 2. Fig. 5 shows an example 5×3 rating matrix and its corresponding NUG for the original update sequence $\langle v_1, v_2, v_3, v_4, v_5, v_6 \rangle$. Nonzero v_2 does not share any row or column with any other nonzero; therefore, node 2 has no incoming or outgoing edges. Other nodes are connected with row and/or column edges according to the given rules. Note that there is no edge between nodes 1 and 6 because of rule (*iii*) although they reside on the same column.

Since NUG G is acyclic, topological ordering(s) exists, where for every directed edge $(u_a, u_b) \in E$, vertex u_a comes before vertex u_b in the sorted order. Therefore any topological order of the NUG gives a valid nonzero update sequence according to Definition 1. In other words, a nonzero update sequence generated from any topological order of the nonzero update graph can be used in SGD updates instead of the original sequence, and the same result will be achieved, which is the main motivation of LASGD.

Example 3. Fig. 6 shows three different valid nonzero update sequences for the example NUG in Fig. 5. There can be many other update sequences as well.

LASGD tries to pick a valid update sequence that enables more effective use of memory hierarchies. Although LASGD is inherently an optimization over sequential SGD algorithm, it can be adapted for parallel algorithms as well. We describe LASGD in detail on the sequential case in the next section. Then, we discuss how it can be extended to cover parallel SGD methods that use grid-based partitioning.

5.1. LASGD on a single thread

Algorithm 7 shows the steps of the LASGD algorithm. As the nodes represent nonzeros, node and nonzero will be used interchangeably to refer to the same entity. The first step of LASGD

is to create the NUG for a given V matrix and a given update sequence $\langle V \rangle$. NUG is stored in compressed out-adjacency format, where $OutAdj(u)$ returns the set of vertices adjacent to vertex u. PQ denotes the priority queue that maintains the nodes with no incoming edges, which are effectively detected through zero in-degree values. Nodes in PQ are keyed with respect to the score values computed according to the criteria mentioned below. PQ supports extract max and insert operations. $\langle V^* \rangle$ denotes the valid nonzero update sequence to be generated by the LASGD algorithm.

The nodes in PQ constitute the set of candidate nonzeros for appending to the end of current $\langle V^* \rangle$ without disturbing the validity of $\langle V^* \rangle$. When a nonzero with the highest score is extracted from PQ, it is appended to the end of $\langle V^* \rangle$ being constructed, and the outgoing edges of the respective node are removed from the graph through decrementing the in-degree values of its out-adjacent nodes. The neighbor nodes whose in-degree values reduce to zero are inserted into PQ. This is repeated until PQ becomes empty.

The criteria considered while selecting a nonzero from PQ are listed below.

(i) **The corresponding P- or Q-matrix rows are cached** Since access time to data in caches is much faster than access time to DRAM, among two candidate nonzeros, the one whose latent factor rows are more probable to be in the cache is preferred. A least recently used (LRU) cache structure is used to simulate the real cache of the hardware. In this way, LASGD can predict whether a P- or Q-matrix row is in the cache. The logic of the real hardware is more complex, but our experiments show that an LRU cache structure can model the behavior of the hardware caches reasonably well.

(ii) **The corresponding P- or Q-matrix rows are close to recently used rows in DRAM** According to our microbenchmarks [36], write and read operations on 2 KB DRAM windows in which the last used data resides is much faster than other regions of DRAM. Therefore, if P- or Q-matrix rows are not found in the cache, LASGD prefers nonzeros whose latent factors are close to the last used memory location. These decisions are taken in the preprocessing step by modeling core caches and DRAM.

(iii) **The corresponding P- or Q-matrix rows are encountered the first time in the ordering step** It is directly related to the previous point. If P- or Q-matrix rows are not updated by any previous nonzeros in the modified sequence, it means that their location in DRAM is not fixed yet. So, we can move these rows near the recently used DRAM location. By doing so, SGD can update the rows more quickly.

The score of a nonzero is calculated by considering where corresponding P- and Q-matrix rows are located in the memory. The calculation method is summarized in Table 3. In the table, *cache* means that P- or Q-matrix row can be found in the cache. So, it refers to the criterion (i). For example, if both P-matrix and Q-matrix rows are in the cache, the score of the nonzero is calculated as 9. *close* means that the corresponding P- or Q-matrix row is not in the cache, but it is close to last accessed row (criterion (ii)). *new* refers to criterion (iii). *none* means that P- or Q-matrix row does not satisfy any condition given in the criteria. Although we used the scores given in the table for all our experiments, it is possible to get different results with different scoring systems. How scoring affects the performance of LASGD algorithm can be explored further in a future work.

The LRU cache simulation is conducted with a custom data structure that uses doubly linked list and hash map data structures to execute lookup, insertion, and eviction operations in constant time. Half of the L3 cache size is set as the size of the LRU cache. Since the number of unique score values is low, we store elements of PQ in buckets (implemented with hash sets) that are assigned to a unique score. Each element also has a reference to the bucket it is contained in. This setup allows us to insert elements to PQ, extract elements from PQ, and update the score of the elements quickly. It should be noted that updating the score of an element means that the node is moved from one bucket to another. While extracting the element with the maximum score, ties are broken arbitrarily.

Table 3
LASGD Score calculation. The states in the first two columns are interchangeable.

P-matrix row	Q-matrix row	Score
cache	cache	9
cache	close	8
cache	new	7
close	close	6
close	new	5
new	new	4
cache	none	3
active	none	2
new	none	1
none	none	0

Algorithm 7: LASGD Algorithm

```
Input: V, ⟨V⟩
Construct the NUG, G(V, ⟨V⟩) = (U, E) ;
foreach u ∈ U do
    InDeg[u] ← 0;
▷ Compute the number of incoming edges of nodes
foreach u ∈ U do
    foreach x ∈ OutAdj(u) do
        InDeg[x] ← InDeg[x] + 1;
PQ ← ∅ ;
foreach u ∈ U do
    if InDeg[u] = 0 then
        ▷ u is safe to add to PQ
        PQ ← PQ ∪ {u} ;
foreach u ∈ U do
    ▷ Assign the score according to Table 3
    Calculate the score of u ;
▷ Initialize the new nonzero update order ⟨V*⟩
⟨V*⟩ ← ⟨⟩
while PQ ≠ ∅ do
    u ← EXTRACT-MAX(PQ) ;
    ⟨V*⟩ ← ⟨⟨V*⟩, u⟩ ;
    foreach x ∈ OutAdj(u) do
        InDeg[x] ← InDeg[x] − 1 ;
        if InDeg[x] = 0 then
            ▷ x is safe to add to PQ
            INSERT(PQ, x) ;
    foreach u ∈ PQ do
        Calculate the score of u ;
SGD(V, ⟨V*⟩)
```

5.2. LASGD on multiple threads

The problem encountered when adapting LASGD to parallel methods is that the actual update sequence is not known beforehand. For two nonzeros assigned to different threads, one nonzero can be processed earlier in one execution, and the other one can be processed earlier in another execution depending on the processing speed of the local SGD computations. This situation

Table 4
Dataset Properties.

Dataset	Number of			Density	Row degree				Column degree			
	rows	columns	nonzeros		min	max	mean	cv	min	max	mean	cv
amazon_item_dedup	21,176,522	9,874,211	82,677,131	3.9e−07	1	44,557	3.90	4.93	1	25,368	8.37	7.76
amazon_books	8,026,324	2,330,066	22,507,155	1.2e−06	1	43,201	2.80	8.20	1	21,398	9.65	6.66
amazon_clothing	3,117,268	1,136,004	5,748,920	1.6e−06	1	349	1.84	1.32	1	3,047	5.06	4.59
amazon_electronics	4,201,696	476,002	7,824,482	3.9e−06	1	520	1.86	1.54	1	18,244	16.43	6.85
amazon_movies	2,088,620	200,941	4,607,047	1.0e−05	1	2,654	2.20	5.15	1	11,906	22.92	5.33
lastfm	359,349	268,758	17,559,530	1.8e−04	1	166	48.86	0.17	1	77,348	65.33	10.71
yahoo_music	249,012	296,111	61,944,406	8.4e−04	17	107,936	248.76	3.27	16	118,308	209.19	6.52
movielens_latest	270,896	45,115	26,024,289	2.1e−03	1	18,276	96.06	2.14	1	91,921	576.84	5.26
movielens_20m	138,493	26,744	20,000,263	5.3e−03	20	9,254	144.41	1.59	1	67,310	747.84	4.12
netflix	480,189	17,770	100,480,507	1.1e−02	1	17,653	209.25	1.14	13	232,944	5654.5	2.99

makes it difficult to construct a NUG since we cannot determine edges and their directions.

We can resolve this problem by using 2D grid-based parallel methods, in which nonzero update sequence is known in block level, instead of using methods like Hogwild, whose nonzero update sequence is more chaotic. If the NUG generation rules are analyzed carefully, it can be seen that LASGD does not need to know the nonzero update sequence completely. The relative update order between independent nonzeros, nonzeros that do not share a P- or Q-matrix row, is insignificant for the algorithm since corresponding nodes in the NUG are not connected at all. Grid-based methods with synchronous block scheduling ensure that threads work on independent blocks at all times. If blocks are sufficiently balanced, the execution of threads on independent blocks is expected to be satisfied for grid-based methods with asynchronous scheduling as well. If we go over Fig. 3, P- and Q-matrix updates incurred by nonzeros in blocks At, Bt, Ct and Dt are independent from each other in the same time interval t.

Because of this observation, LASGD can create a NUG by assuming that there exists some relative order between nonzeros processed in the same time interval as long as the update order in a single block is preserved. Only one NUG can be generated, regardless of the order in which blocks are processed within the same time interval. Therefore, the rating matrix is partitioned with a grid partitioning method in order to apply LASGD when multiple threads are used. Then, an arbitrary processing order is assumed to exist between blocks that are updated at the same time interval. Then, the same LASGD procedure given in Algorithm 7 for a single thread is applied in the multicore context. Any load balancing method can be used in the grid partitioning step. The performance of the load balancing method becomes important for the performance of LASGD ordering procedure in the multicore setting.

6. Experimental setup and results

6.1. Experimental framework

All algorithms are implemented in C++11, and OpenMP [37] library is used for parallelization. Some of the relevant papers do not offer an open implementation, or they are developed for distributed systems only but still applicable to shared memory systems. Therefore, we coded those methods. In fact, we have coded all compared methods from scratch even if implementation(s) already exists because our intention is not to compare exact implementations but to investigate the main ideas behind those methods and what makes them slower or faster than the others. For example, AVX2 SIMD instructions are used in both proposed algorithms and baseline algorithms to optimize the running the times.

P and Q matrices are initialized uniformly random between 0 and 1. The latent factor size f is chosen as 16. Since the aim is to estimate unseen ratings as correctly as possible, 20% of data, known as the test set, is set aside and used only to evaluate the quality of the solution. All evaluation error results are calculated with the root mean squared error (RMSE) [38] metric on the test set.

We use dual Intel Xeon Platinum 8280 CPUs having 28 physical cores each for performance experiments. Each core has a private 32 KB L1 data cache and 1024 KB L2 cache. 38.5 MB L3 cache is shared by all cores in a socket.

6.2. Datasets

The selected rating matrices and their properties are reported in Table 4. The main categories are shopping, movies, and music. Netflix and yahoo datasets are used in competitions of Netflix Challenge and KDD cup, respectively. Besides them, we tried to use less commonly known datasets like `amazon-electronics` [15]. Although they have extra attributes like timestamp, only rating, user id, and item id values are considered in the experiments.

In Table 4, the rating matrices are listed in increasing density order, where the density of a matrix refers to the ratio of the total number of nonzeros to $m \times n$. The first five matrices, *amazon* matrices, are considered as sparse matrices, whereas the remaining five matrices are considered as dense matrices. In the table, "CV" refers to the coefficient of variation in the degrees of the rows and columns. Higher CV values refer to more irregular nonzero distribution on rows and columns.

We prefer using coordinate list (COO) sparse matrix format to store the rating matrix as it is more flexible for permuting nonzero elements. We permute the update sequence of nonzeros to achieve two stochastic properties mentioned in Section 2. Thanks to this permutation, sequential SGD guarantees that each nonzero is used for updates exactly once at each iteration. In addition to permuting the order of updates, we also permute rows and columns of the rating matrix in order to prevent the effect of a particular distribution imposed by the original row/column ordering on the performance of the algorithms.

6.3. Load balancing performance analysis

Table 5 compares the performance of the proposed load balancing algorithms BP_{r+c} and $BP_{r \to c}$ against random partitioning (RP) and BaPa [23]. In our experiments, we did not find any considerable differences between load imbalance values attained by $BP_{r \to c}$ and $BP_{c \to r}$. So, we do not report $BP_{c \to r}$ results in this table and the following figures. However, load balancing performances of $BP_{r \to c}$ and $BP_{c \to r}$ may differ if rows and columns of the rating matrix have a substantially different nonzero distributions.

In the table, load imbalance values are computed as the ratio of the number of nonzeros in the block with the largest number of nonzeros to the number of nonzeros in the block with the least number of nonzeros. RP gives decent load balancing performance

Table 5
Load balancing performance comparison of 2D-grid partitioning algorithms with increasing grid dimensions.

Dataset	4 × 4				16 × 16				64 × 64			
	RP	BaPa	BP		RP	BaPa	BP		RP	BaPa	BP	
			$r+c$	$r \to c$			$r+c$	$r \to c$			$r+c$	$r \to c$
amazon_item_dedup	1.013	1.001	1.001	1.000	1.059	1.009	1.009	1.000	1.172	1.047	1.048	1.000
amazon_books	1.037	1.002	1.003	1.000	1.112	1.020	1.018	1.000	1.361	1.110	1.096	1.000
amazon_clothing	1.015	1.005	1.003	1.000	1.083	1.032	1.034	1.000	1.368	1.190	1.184	1.001
amazon_electronics	1.038	1.003	1.003	1.000	1.175	1.031	1.029	1.000	1.639	1.177	1.156	1.001
amazon_movies	1.067	1.008	1.004	1.000	1.227	1.041	1.031	1.000	1.834	1.356	1.189	1.000
lastfm	1.083	1.003	1.001	1.000	1.490	1.048	1.021	1.000	2.366	1.282	1.108	1.000
yahoo_music	1.101	1.001	1.001	1.000	1.336	1.014	1.009	1.000	1.855	1.110	1.059	1.000
movielens_latest	1.143	1.002	1.002	1.000	1.449	1.039	1.015	1.000	3.330	1.197	1.085	1.000
movielens_20m	1.071	1.009	1.001	1.000	1.443	1.039	1.015	1.000	3.902	1.189	1.088	1.000
netflix	1.127	1.003	1.001	1.000	1.491	1.015	1.006	1.000	2.543	1.129	1.046	1.001
geometric average	1.068	1.003	1.002	1.000	1.276	1.029	1.019	1.000	1.985	1.176	1.104	1.000

RP: Random partitioning. BaPa: Greedy independent row and column partitioning [23] BP_{r+c}: Independent BP-based row and column partitioning. $BP_{r \to c}$: First row then column BP-based partitioning. Load imbalance values indicate the ratio of the number of nonzeros in the block with the largest number of nonzeros to the number of nonzeros in the block with least number of nonzeros.

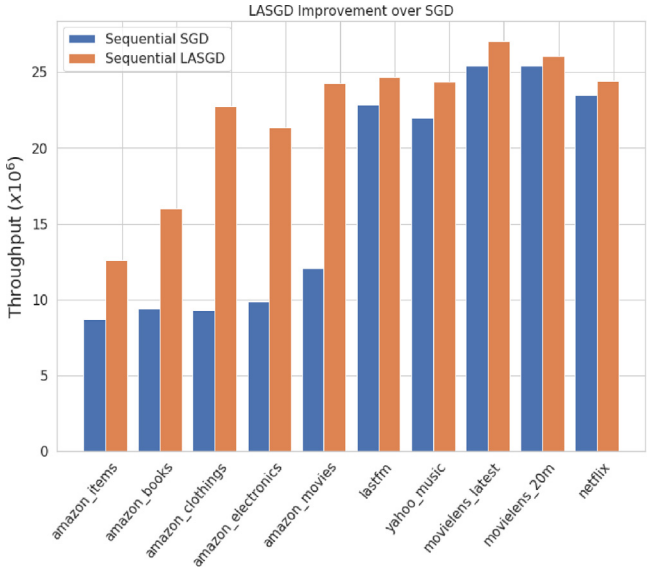

Fig. 7. Sequential LASGD compared to Sequential SGD.

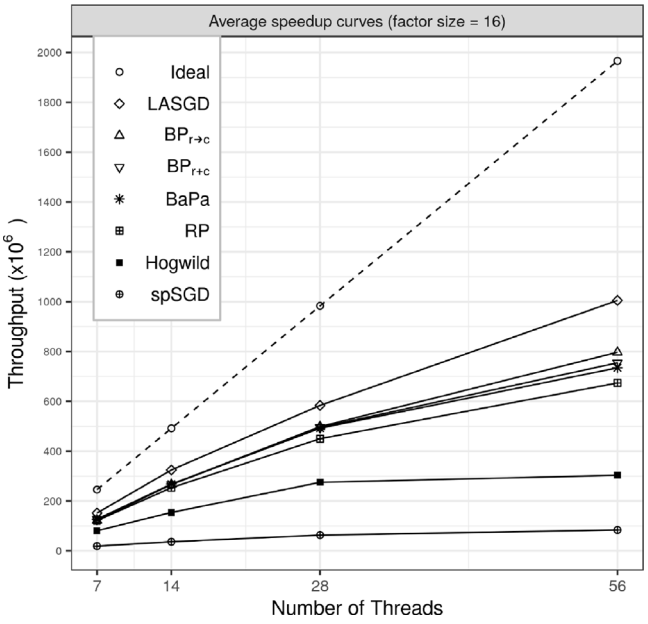

Fig. 8. Average speedup curves for $f = 16$. The y axis shows the throughput values which are computed as the number of nonzero updates per second. The proposed load balancing algorithms (BP_{r+c}, $BP_{r \to c}$) outperform the other load balancing algorithms on average. Our proposed algorithm, LASGD, improves the performance significantly with better scalability.

in small grid dimension (4 × 4); however, its performance degrades significantly with increasing grid dimensions (16 × 16 and 64 × 64) especially for dense matrices. For sparse rating matrices (*amazon* datasets), the load imbalance ratio attained by the RP remains below 1.9 even for 64 × 64 partitioning. BaPa performs much better than RP, especially in larger grid dimensions.

BP_{r+c} performs slightly better than BaPa where the performance gap increases with increasing grid dimensions in favor of BP_{r+c}. Both BaPa and BP_{r+c} attain rather low imbalance values even for large grid dimensions. For example, BaPa attains average load imbalance values of 1.029 and 1.176 on 16 × 16 and 64 × 64 grid dimensions respectively, whereas, BP_{r+c} attains 1.019 and 1.104. It may seem counter-intuitive that independent row and column partitioning (BaPa and BP_{r+c}) gives balanced blocks. These algorithms assume that nonzero distribution in a column is independent of the number of nonzeros in the column. This looks like a strong assumption, but BP_{r+c} provides a significant improvement over RP.

$BP_{r \to c}$ performs better than BP_{r+c} and BaPa, where the performance gap increases with increasing grid dimensions. For example, $BP_{r \to c}$ achieves 0.3%, 2.9%, 17.6% better load balancing performance than BaPa on 4 × 4, 16 × 16, 64 × 64 grid dimensions, respectively. This shows the benefit of using the partition information obtained over one matrix dimension in the first phase for the partitioning decisions to be performed over the other dimension in the second phase.

6.4. Throughput performance analysis

Figs. 7–9 compare the performance of the algorithm in terms of throughput values. The throughput values are computed as the number of nonzero updates per second.

Fig. 7 shows the performance improvement attained by LASGD algorithm in a sequential setting (single thread) for each dataset. As seen in the figure, LASGD attains considerably higher throughput than SGD in all datasets. On the other hand, LASGD performs significantly better than SGD on sparse datasets. For example, the improvement of LASGD over SGD varies between 14% and 101% on the sparse *amazon* matrices, whereas it remains between 2% and 8% on the dense matrices. This is because LASGD is expected to have larger freedom in nonzero selection on sparse matrices

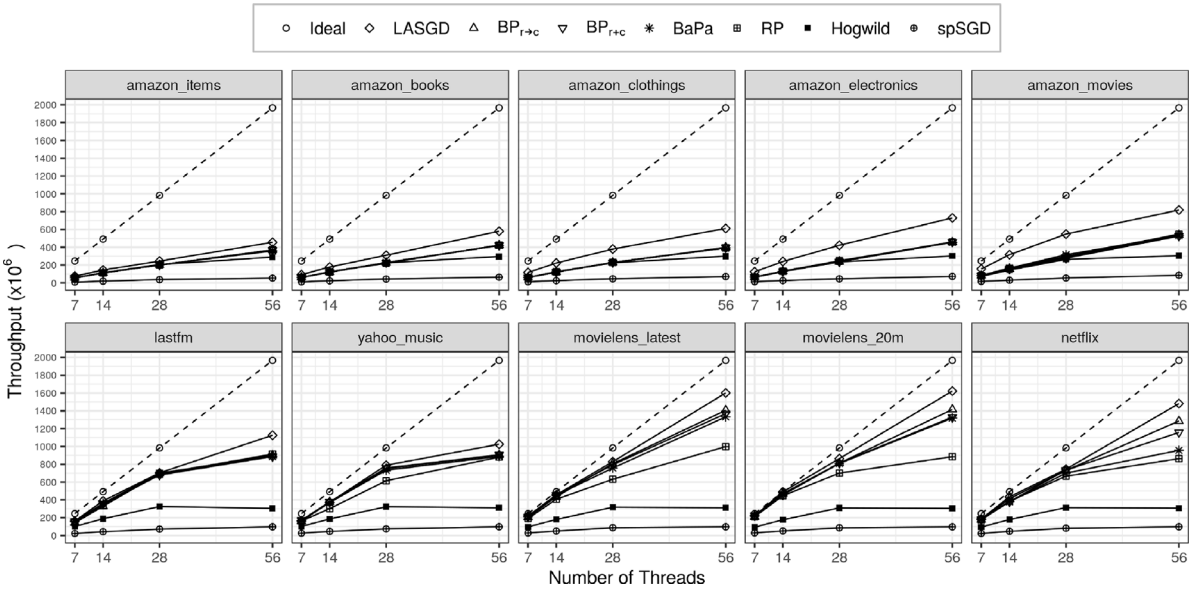

Fig. 9. Speedup curves. The *y* axis shows the throughput values which are computed as the number of nonzero updates per second. The proposed algorithms (LASGD, BP$_{r+c}$, BP$_{r\to c}$) perform better than the other algorithms and the performance gap increases with increasing number of threads in favor of the proposed algorithms. In dense matrices, the proposed algorithms approach the ideal speedup curve.

compared to dense matrices. That is, on sparse matrices, the priority queue is expected to contain a larger number of candidate nonzeros (nodes with zero in-degrees) for selection compared to dense matrices. In graph-theoretic view, this corresponds to the expectation that sparse acyclic graphs will have a larger number of different topological orderings compared to dense acyclic graphs with the same number of nodes.

Figs. 8 and 9 display the results of the strong scaling experiments as speedup curves in terms of increase in throughput with the increasing number of threads. Fig. 8 shows the average speedup curves, whereas Fig. 9 shows speedup curves for the individual datasets. All algorithms except spSGD and Hogwild utilize 2D-grid partitioning. LASGD utilizes BP$_{r\to c}$ for load balancing in grid partitioning.

As seen in Figs. 8 and 9, spSGD shows the worst performance as expected. Hogwild performs significantly better than spSGD. However, the average throughput attained by Hogwild does not change between 28 threads and 56 threads. When code profiling results and the architecture of the system are considered together, we conclude that data migrations between L3 caches of sockets cause this behavior. There are 28 cores on each socket, so when 56 threads are used, data migrations between L1 or L2 caches become migrations between L3 caches. Since L3 latency is much higher than L1 and L2 latency, almost no speedup is achieved. On the other hand, this problem does not occur in grid partitioning methods because data migration is avoided due to threads running on independent blocks.

All three grid-based methods show much better speedup performance than Hogwild. Among the grid-based methods, both bin-packing-based algorithms (BP$_{r+c}$ and BP$_{r\to c}$) perform considerably better than RP, where BP$_{r\to c}$ is the winner. *BP*$_{r\to c}$ performs slightly better than BaPa, but the difference is not distinguishable for most datasets. The difference becomes noticeable as *P* increases, especially when the dataset is dense. The significant difference of 34.4% more throughput achieved by BP$_{r\to c}$ compared to BaPa for `netflix` dataset on 56 threads, which has a small number of rows/columns but a large number of nonzeros, confirms this claim. On average, BP$_{r\to c}$ achieves 1.2% and 7.3% more throughput than BaPa on 28 and 56 threads, respectively. These relative speedup results conform with the relative load balancing comparison given in Table 5.

The proposed LASGD algorithm performs significantly better than all other algorithms. For example, on 56 threads, LASGD achieves 200% and 50% more throughput than Hogwild and RP, respectively. The performance gap between LASGD and other algorithms increases with increasing number of threads in favor of LASGD. As seen in Fig. 9, LASGD performs significantly better than BaPa in every dataset. This improvement gap increases with increasing number of processors in favor of LASGD. For example, LASGD achieves 20%, 21%, 18% and 33% more throughput than BaPa on 7, 14, 28 and 56 threads, respectively. These results show that LASGD increases the scalability of SGD.

In Fig. 9, when speedup curves for different datasets are examined separately, it becomes easier to understand why some ideas work better for some datasets. BP$_{r\to c}$ as well as LASGD (which also utilizes BP$_{r\to c}$) approach the ideal speedup for the three most dense matrices (`netflix`, `movielens_20m` and `movielens_latest`). On these datasets, there is an immense improvement gained through proposed BP-based load balancing algorithms over the random permutation. As seen in Table 5, `netflix` and `movielens` datasets suffer the most from load imbalance so it is logical to see that intelligent load balancing methods like BP$_{r+c}$ and BP$_{r\to c}$ affect the performance significantly.

Table 6 shows code profiling results obtained with vTune code profiler about memory accesses of BaPa and LASGD. Here, BaPa partitioning is used in the grid partitioning phase of LASGD so that the improvement of LASGD over BaPa will result from the nonzero update ordering and memory access patterns utilized in LASGD. The memory bound column shows the percentage of pipeline slots wasted due to the stalls during memory accesses. Columns L1, L2, and L3 show the percentage of stalled CPU clockticks due to read/write operations on L1, L2, and L3 caches, respectively. Similarly, the DRAM column gives the bound of DRAM-related operations in terms of the percentage of stalled CPU clockticks. The last column shows the number of last level cache (LLC) misses. LLC miss occurs when a requested cache line

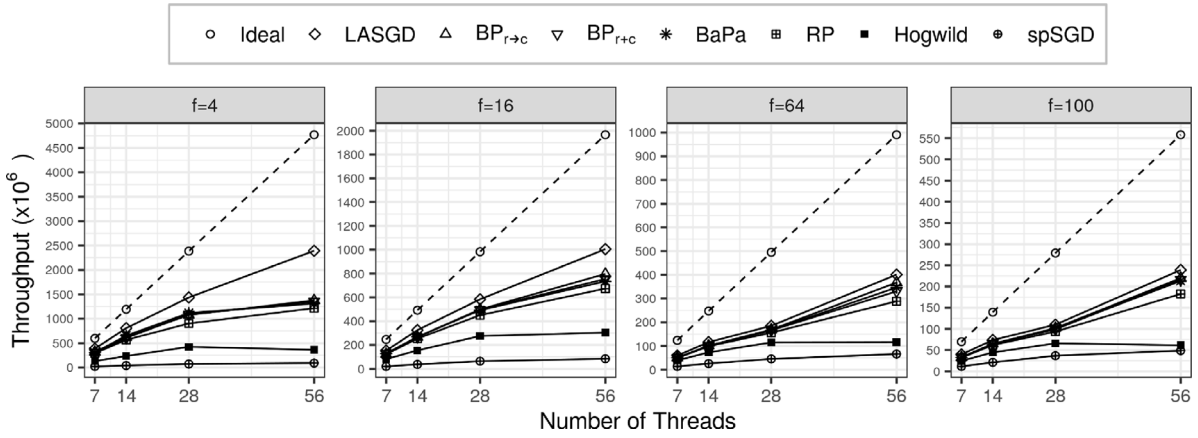

Fig. 10. Average speedup curves for different f values. The y axis shows the throughput values which are computed as the number of nonzero updates per second. The performance gap between LASGD and the other grid-based algorithms decreases with increasing f values.

Table 6
vTune code profiling results of BaPa and LASGD on 56 cores with 2000 iterations.

Dataset	Method	Memory Bound (%)	L1 (%)	L2 (%)	L3 (%)	DRAM (%)	LLC Miss
amazon_item_dedup	BaPa	74.0	2.1	0.8	10.3	61.4	306,989,686,424
	LASGD	68.8	2.5	2.3	10.6	54.1	193,636,394,682
amazon_books	BaPa	66.4	2.6	0.9	12.9	50.7	58,172,900,361
	LASGD	55.6	3.9	3.2	15.5	33.7	24,205,170,992
amazon_clothing	BaPa	61.8	3.1	0.8	13.7	44.7	13,575,605,624
	LASGD	46.2	4.7	4.4	19.7	18.2	2,286,232,826
amazon_electronics	BaPa	61.3	3.0	1.0	16.1	41.8	14,480,298,104
	LASGD	42.5	5.2	3.4	19.1	15.6	2,494,259,912
amazon_movies	BaPa	49.5	4.3	1.2	23.9	20.9	3,130,426,050
	LASGD	33.6	6.2	2.3	20.0	5.9	178,909,436
lastfm	BaPa	34.9	6.0	2.2	24.8	3.1	389,425,400
	LASGD	22.2	7.1	2.2	12.3	1.3	84,273,774
yahoo_music	BaPa	32.1	5.8	2.5	23.7	1.3	777,830,897
	LASGD	27.1	6.6	2.0	18.4	1.2	861,245,662
movielens_latest	BaPa	21.4	6.2	2.0	12.9	1.4	157,607,890
	LASGD	14.2	7.1	1.9	5.5	0.7	10,574,117
movielens_20m	BaPa	22.4	6.1	1.9	13.9	1.6	157,528,825
	LASGD	14.6	7.3	1.9	5.9	0.5	10,583,199
netflix	BaPa	18.0	6.2	2.2	9.4	1.2	986,055,979
	LASGD	12.8	7.2	2.4	3.8	0.4	167,809,989

cannot be found in L1, L2, and L3 caches, so it needs to be fetched from the DRAM.

As seen in Table 6, LASGD considerably decreases the memory bound of the program in all datasets (37.2% decrease on average). As also seen in the table, LASGD decreases LLC miss rates significantly in all datasets except yahoo_music. This results in fewer CPU cycles wasted on DRAM and consequently faster executions. For example, LASGD reduces LLC miss count by around 94% in amazon_movies dataset. This improvement in LLC miss count reflects as a decrease in DRAM bound from 20.9% to 5.9%. The increase in the lower-level cache bounds of LASGD suggests that more cache lines can be retrieved from L1 and L2 caches directly instead of fetching from DRAM. In order words, LASGD moves read/write operations from high latency DRAM to fast caches. That is why locality-aware ordering adopted in LASGD utilizes memory better and gives better execution performance.

6.5. Convergence and factor size analysis

LASGD on a single thread gives the same RMSE values as the sequential SGD algorithm since it does not affect the update order on individual P- and Q-matrix rows. Table 7 compares RMSE values of sequential SGD algorithm (run on 1 thread) and the LASGD algorithm run on 56 threads for various epoch numbers. As seen in the table, the differences between RMSE values are so small, thus showing that LASGD does not deteriorate the convergence rate.

The factor size parameter f directly affects the accuracy and the performance of the algorithm. Smaller or larger values than the optimal value produce poor accuracy results. When small f values are used, the model (P and Q matrices) is not strong enough to capture the underlying patterns in the rating matrix. Larger f, on the other hand, may overfit the training data. So, it again cannot predict unseen ratings correctly even though its accuracy on known values will be good. Therefore, f should be

Table 7
RMSE convergence values for LASGD on 56 threads and sequential SGD.

Dataset	Method	Epoch No									
		5	10	15	20	25	30	35	40	45	50
amazon_item_dedup	S.SGD	1.2796	1.2468	1.2337	1.2276	1.2250	1.2242	1.2246	1.2257	1.2273	1.2292
	LASGD	1.2795	1.2467	1.2335	1.2274	1.2248	1.2240	1.2244	1.2255	1.2271	1.2290
amazon_books	S.SGD	1.1792	1.1426	1.1261	1.1174	1.1126	1.1099	1.1086	1.1082	1.1084	1.1089
	LASGD	1.1792	1.1424	1.1260	1.1173	1.1124	1.1098	1.1084	1.1080	1.1081	1.1087
amazon_clothing	S.SGD	1.3693	1.3419	1.3292	1.3227	1.3194	1.3179	1.3175	1.3179	1.3188	1.3199
	LASGD	1.3693	1.3418	1.3292	1.3226	1.3193	1.3178	1.3174	1.3178	1.3186	1.3198
amazon_electronics	S.SGD	1.4223	1.3980	1.3872	1.3821	1.3801	1.3798	1.3807	1.3822	1.3843	1.3866
	LASGD	1.4222	1.3978	1.3871	1.3820	1.3799	1.3796	1.3805	1.3821	1.3840	1.3864
amazon_movies	S.SGD	1.2397	1.2124	1.2000	1.1936	1.1902	1.1886	1.1881	1.1884	1.1892	1.1904
	LASGD	1.2396	1.2123	1.2000	1.1934	1.1900	1.1884	1.1879	1.1882	1.1890	1.1900
lastfm	S.SGD	1.9942	1.9535	1.9359	1.9280	1.9246	1.9233	1.9228	1.9227	1.9227	1.9228
	LASGD	1.9944	1.9535	1.9358	1.9280	1.9245	1.9236	1.9228	1.9229	1.9230	1.9230
yahoo_music	S.SGD	27.5526	25.2026	24.2676	23.8323	23.5999	23.4661	23.3860	23.3372	23.3079	23.2912
	LASGD	27.5581	25.1934	24.2580	23.8197	23.5852	23.4510	23.3716	23.3222	23.2923	23.2735
movielens_latest	S.SGD	0.8836	0.8658	0.8530	0.8425	0.8334	0.8261	0.8203	0.8158	0.8123	0.8097
	LASGD	0.8835	0.8661	0.8530	0.8423	0.8331	0.8257	0.8198	0.8153	0.8117	0.8091
movielens_20m	S.SGD	0.8699	0.8583	0.8448	0.8342	0.8246	0.8167	0.8106	0.8060	0.8024	0.7996
	LASGD	0.8697	0.8581	0.8445	0.8339	0.8243	0.8164	0.8102	0.8055	0.8020	0.7991
netflix	S.SGD	0.9336	0.9223	0.9075	0.9019	0.8980	0.8944	0.8916	0.8893	0.8871	0.8850
	LASGD	0.9335	0.9225	0.9074	0.9019	0.8983	0.8947	0.8918	0.8895	0.8872	0.8852

picked in such a way that it should give good results on unseen data.

In the literature, we encountered studies using 10, 40, 50 or 100 [10,12,13,21,24,38] f values. Fig. 10 compares the performance of the algorithm for f values of 4, 16, 64, and 100. The LASGD algorithm improves the latency of reading/writing latent factor rows. So, the value f affects the performance of LASGD. For large f values, it is difficult for the algorithm to apply the same cache locality optimizations and DRAM locality optimization because the algorithm will place fewer P- and Q-matrix rows to the caches and to the active DRAM row. As a result, the larger f values reduce the performance gain of the LASGD algorithm, whereas the smaller f values increase the performance gain.

BaPa algorithm and the proposed load balancing algorithms (BP_{r+c}, $BP_{r\to c}$) reduce the threads' idle time, their optimization is not directly related to the factor size.

To verify the generality of the proposed algorithms, we randomly picked 10 datasets from SuiteSparse [39]. The throughput speedup curves for the selected 10 datasets and the average speedup curves of the algorithms are given in Figs. 11 and 12 respectively. It can be seen that the performance improvement of BaPa, BP_{r+c} and $BP_{r\to c}$ over RP is negligible. This is an expected result. Since the randomly picked matrices are not necessary scale-free, even RP algorithm yields a good load balance. Meanwhile, LASGD improves throughput performance by around 14% with respect to other grid-based algorithms on average. This shows that LASGD performs better even if the matrix does not show any specific nonzero pattern.

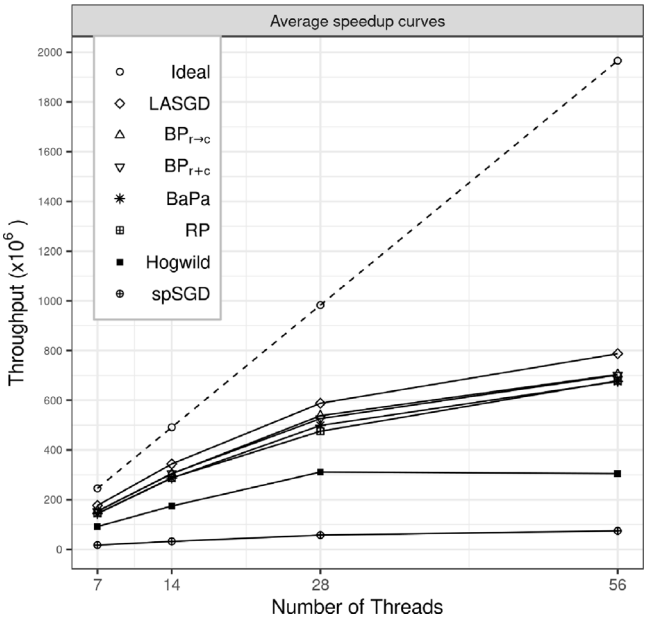

Fig. 11. Average speedup curves of the algorithm on a randomly selected datasets.

7. Conclusion

We investigated the load balancing and cache underutilization problems of parallel SGD methods for matrix completion on multicore systems. We proposed bin-packing-based algorithms for balancing the nonzero counts of the 2D blocks in 2D-grid partitioning. We proposed a memory-access improving method, LASGD, without disturbing the stochastic nature of the algorithm. Extensive experiments performed on a wide range of matrix completion datasets lead to the following findings:

- The proposed load-balancing algorithms perform better than the state-of-the-art load-balancing algorithms and the performance gap increases with increasing grid dimensions in favor of the proposed algorithms.
- In 2D grid based partitioning, instead of independent row and column partitioning, using the row partitioning information obtained in the first phase for the column partitioning to be performed in the second phase, or vice versa, significantly increases the load balancing performance.
- Both the proposed load balancing methods and LASGD favor skewed matrices. Although the proposed load balancing methods favor dense datasets more than sparse ones, LASGD favors sparse datasets.
- The proposed load-balancing and LASGD algorithms significantly increase the scalability of the SGD algorithms compared to existing state-of-the-art SGD algorithms.

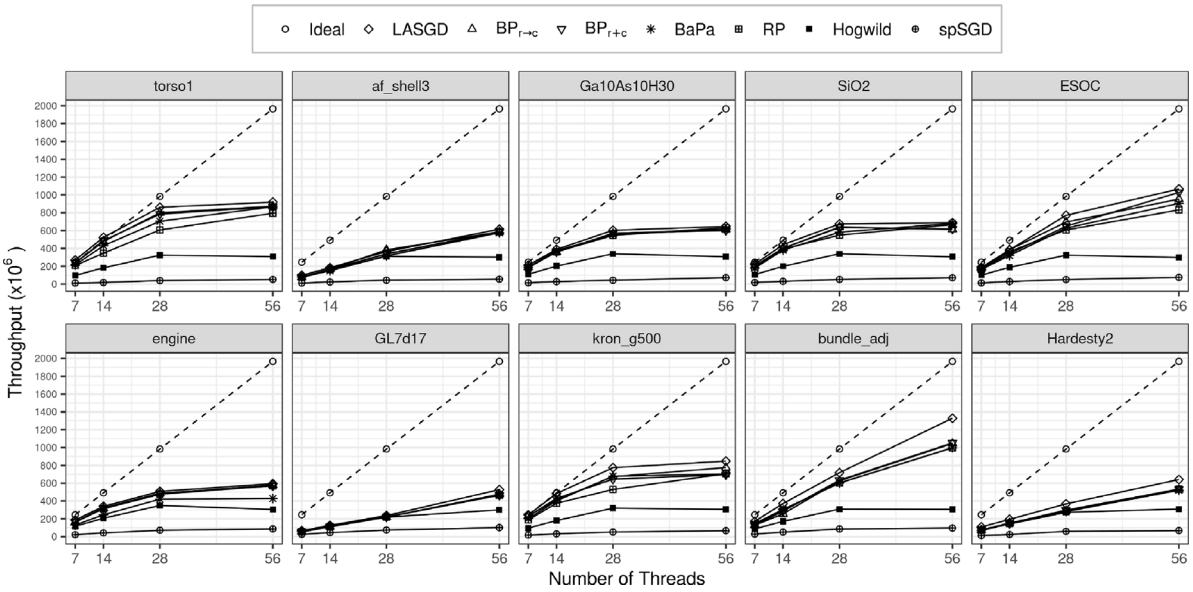

Fig. 12. Speedup curves of the algorithm on a randomly selected datasets.

As a future work, we consider exploring and experimenting with different scoring schema to find better nonzero orderings used in the LASGD algorithm.

CRediT authorship contribution statement

Selcuk Gulcan: Software, Validation, Investigation, Visualization, Writing – original draft, Writing – review & editing. **Muhammet Mustafa Ozdal:** Conceptualization, Methodology, Supervision, Writing – original draft. **Cevdet Aykanat:** Conceptualization, Methodology, Supervision, Funding acquisition, Writing – original draft, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

Acknowledgment

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

References

[1] O. Levy, Y. Goldberg, Neural word embedding as implicit matrix factorization, in: Advances in Neural Information Processing Systems, 2014, pp. 2177–2185.
[2] J. Lu, D. Wu, M. Mao, W. Wang, G. Zhang, Recommender system application developments: a survey, Decis. Support Syst. 74 (2015) 12–32.
[3] H. Kim, H. Park, Sparse non-negative matrix factorizations via alternating non-negativity-constrained least squares for microarray data analysis, Bioinformatics 23 (12) (2007) 1495–1502.
[4] J. Bennett, S. Lanning, et al., The netflix prize, in: Proceedings of KDD Cup and Workshop. Vol. 2007, Citeseer, 2007, p. 35.
[5] R.M. Bell, Y. Koren, C. Volinsky, All together now: A perspective on the netflix prize, Chance 23 (1) (2010) 24–29.
[6] R.M. Bell, Y. Koren, Lessons from the Netflix prize challenge, SiGKDD Explorations 9 (2) (2007) 75–79.
[7] E.J. Candès, B. Recht, Exact matrix completion via convex optimization, Found. Comput. Math. 9 (6) (2009) 717.
[8] P. Jain, P. Netrapalli, Fast exact matrix completion with finite samples, in: Conference on Learning Theory, 2015, pp. 1007–1034.
[9] J. Langford, A. Smola, M. Zinkevich, Slow learners are fast, 2009, arXiv preprint arXiv:0911.0491.
[10] R. Gemulla, E. Nijkamp, P.J. Haas, Y. Sismanis, Large-scale matrix factorization with distributed stochastic gradient descent, in: Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, ACM, 2011, pp. 69–77.
[11] B. Recht, C. Re, S. Wright, F. Niu, Hogwild: A lock-free approach to parallelizing stochastic gradient descent, in: Advances in Neural Information Processing Systems, 2011, pp. 693–701.
[12] Y. Zhuang, W.-S. Chin, Y.-C. Juan, C.-J. Lin, A fast parallel SGD for matrix factorization in shared memory systems, in: Proceedings of the 7th ACM Conference on Recommender Systems, ACM, 2013, pp. 249–256.
[13] H. Yun, H.-F. Yu, C.-J. Hsieh, S. Vishwanathan, I. Dhillon, NOMAD: Non-locking, stochastic multi-machine algorithm for asynchronous and decentralized matrix completion, Proc. VLDB Endow. 7 (11) (2014) 975–986.
[14] F. Petroni, L. Querzoni, GASGD: stochastic gradient descent for distributed asynchronous matrix completion via graph partitioning, in: Proceedings of the 8th ACM Conference on Recommender Systems, ACM, 2014, pp. 241–248.
[15] H. Lakkaraju, J. McAuley, J. Leskovec, What's in a name? understanding the interplay between titles, content, and communities in social media, in: Seventh International AAAI Conference on Weblogs and Social Media, 2013.
[16] T. Bertin-Mahieux, D.P. Ellis, B. Whitman, P. Lamere, The million song dataset, in: Proceedings of the 12th International Conference on Music Information Retrieval, ISMIR 2011, 2011.
[17] F.M. Harper, J.A. Konstan, The movielens datasets: History and context, Acm Trans Inter. Intell. Syst. (Tiis) 5 (4) (2015) 1–19.
[18] L. Bottou, Large-scale machine learning with stochastic gradient descent, in: Proceedings of COMPSTAT'2010, Springer, 2010, pp. 177–186.
[19] J. Kiefer, J. Wolfowitz, et al., Stochastic estimation of the maximum of a regression function, Ann. Math. Stat. 23 (3) (1952) 462–466.
[20] J. Reinders, VTune Performance Analyzer Essentials, Intel Press, 2005.
[21] F. Makari, C. Teflioudi, R. Gemulla, P. Haas, Y. Sismanis, Shared-memory and shared-nothing stochastic gradient descent algorithms for matrix completion, Knowl. Inf. Syst. 42 (3) (2015) 493–523.
[22] Y. Yu, D. Wen, Y. Zhang, X. Wang, W. Zhang, X. Lin, Efficient matrix factorization on heterogeneous CPU-GPU systems, 2020, arXiv preprint arXiv:2006.15980.
[23] R. Guo, F. Zhang, L. Wang, W. Zhang, X. Lei, R. Ranjan, A. Zomaya, BaPa: A novel approach of improving load balance in parallel matrix factorization for recommender systems, IEEE Trans. Comput. (2020).
[24] X. Xie, W. Tan, L.L. Fong, Y. Liang, 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, 2017, pp. 79–92.

[25] H. Li, K. Li, J. An, K. Li, MSGD: A novel matrix factorization approach for large-scale collaborative filtering recommender systems on GPUs, IEEE Trans. Parallel Distrib. Syst. 29 (7) (2017) 1530–1544.
[26] Z. Wu, Y. Luo, K. Lu, X. Wang, Parallelizing stochastic gradient descent with hardware transactional memory for matrix factorization, in: 2018 3rd International Conference on Information Systems Engineering, ICISE, IEEE, 2018, pp. 118–121.
[27] M. Yagci, T. Aytekin, F. Gurgen, On parallelizing SGD for pairwise learning to rank in collaborative filtering recommender systems, in: Proceedings of the Eleventh ACM Conference on Recommender Systems, 2017, pp. 37–41.
[28] Z.A. Khan, N.I. Chaudhary, S. Zubair, Fractional stochastic gradient descent for recommender systems, Electr. Mark. 29 (2) (2019) 275–285.
[29] J. Oh, W.-S. Han, H. Yu, X. Jiang, Fast and robust parallel SGD matrix factorization, in: Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, ACM, 2015, pp. 865–874.
[30] D. Lee, J. Oh, C. Faloutsos, B. Kim, H. Yu, Disk-based matrix completion for memory limited devices, in: Proceedings of the 27th ACM International Conference on Information and Knowledge Management, 2018, pp. 1093–1102.
[31] X. Luo, Y. Xia, Q. Zhu, Applying the learning rate adaptation to the matrix factorization based collaborative filtering, Knowl.-Based Syst. 37 (2013) 154–164.
[32] Z.A. Khan, S. Zubair, N.I. Chaudhary, M.A.Z. Raja, F.A. Khan, N. Dedovic, Design of normalized fractional SGD computing paradigm for recommender systems, Neural Comput. Appl. 32 (2020) 10245–10262.
[33] Z.A. Khan, M.A.Z. Raja, N.I. Chaudhary, K. Mehmood, Y. He, MISGD: Moving-information-based stochastic gradient descent paradigm for personalized fuzzy recommender systems, Int. J. Fuzzy Syst. (2022) 1–27.
[34] Z.A. Khan, N.I. Chaudhary, M.A.Z. Raja, Generalized fractional strategy for recommender systems with chaotic ratings behavior, Chaos Solitons Fractals 160 (2022) 112204.
[35] E. Horowitz, S. Sahni, Fundamentals of Computer Algorithms, Computer Science Press, 1978.
[36] S. Gulcan, Parallel Stochastic Gradient Descent on Multicore Architectures, Bilkent University (Turkey), 2020.
[37] L. Dagum, R. Menon, OpenMP: An industry-standard API for shared-memory programming, Comput. Sci. Eng. (1) (1998) 46–55.
[38] H.-F. Yu, C.-J. Hsieh, S. Si, I. Dhillon, Scalable coordinate descent approaches to parallel matrix factorization for recommender systems, in: 2012 IEEE 12th International Conference on Data Mining, IEEE, 2012, pp. 765–774.
[39] T.A. Davis, Y. Hu, The University of Florida sparse matrix collection, ACM Trans. Math. Softw. 38 (1) (2011) 1–25.

Selcuk Gulcan received the B.S. and M.S. degrees in computer engineering from Bilkent University, Turkey, in 2016 and 2020, respectively. He is currently working toward the Ph.D. degree at Bilkent University. His research interests include recommender systems and parallel computing in distributed and shared memory systems.

Muhammet Mustafa Ozdal received the Ph.D. degree in computer science from the University of Illinois at Urbana-Champaign, in 2005. He is currently a research scientist at Facebook's AI Systems Software/Hardware Co-Design group. Previously, he was an associate professor at the Computer Engineering Department of Bilkent University and a research scientist at the strategic CAD Labs of Intel Corporation. He has served in the executive and technical program committees of several conferences. He is a recipient of the IEEE/ACM William J. McCalla ICCAD Best Paper Award (2011), ACM SIGDA Technical Leadership Award (2012), TUBITAK Postdoctoral Reintegration Fellowship (2016), and the European Commission MSCA Individual Fellowship (2016). His research interests include high performance computing and software/hardware codesign of large scale AI problems.

Cevdet Aykanat received the B.S. and M.S. degrees from Middle East Technical University, Turkey, both in electrical engineering, and the Ph.D. degree from Ohio State University, Columbus, in electrical and computer engineering. He worked at the Intel Supercomputer Systems Division, Beaverton, Oregon, as a research associate. Since 1989, he has been affiliated with the Department of Computer Engineering, Bilkent University, Turkey, where he is currently a professor. His research interests mainly include parallel computing and its combinatorial aspects. He is the recipient of the 1995 Investigator Award of The Scientific and Technological Research Council of Turkey and 2007 Parlar Science Award. He has served as an Associate Editor of IEEE Transactions of Parallel and Distributed Systems between 2009 and 2013.