PARALLEL STOCHASTIC GRADIENT DESCENT ON MULTICORE ARCHITECTURES

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

By
Selçuk Gülcan
September 2020

Parallel Stochastic Gradient Descent on Multicore Architectures
By Selçuk Gülcan
September 2020

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

M. Mustafa Özdal(Advisor)

Cevdet Aykanat(Co-Advisor)

Özcan Öztürk

Tayfun Küçükyılmaz

Approved for the Graduate School of Engineering and Science:

Ezhan Karaşan
Director of the Graduate School

ABSTRACT

PARALLEL STOCHASTIC GRADIENT DESCENT ON MULTICORE ARCHITECTURES

Selçuk Gülcan
M.S. in Computer Engineering
Advisor: M. Mustafa Özdal
September 2020

The focus of the thesis is efficient parallelization of the Stochastic Gradient Descent (SGD) algorithm for matrix completion problems on multicore architectures. Asynchronous methods and block-based methods utilizing 2D grid partitioning for task-to-thread assignment are commonly used approaches for shared-memory parallelization. However, asynchronous methods can have performance issues due to their memory access patterns, whereas grid-based methods can suffer from load imbalance especially when data sets are skewed and sparse. In this thesis, we first analyze parallel performance bottlenecks of the existing SGD algorithms in detail. Then, we propose new algorithms to alleviate these performance bottlenecks. Specifically, we propose bin-packing-based algorithms to balance thread loads under 2D partitioning. We also propose a grid-based asynchronous parallel SGD algorithm that improves cache utilization by changing the entry update order without affecting the factor update order and rearranging the memory layouts of the latent factor matrices. Our experiments show that the proposed methods perform significantly better than the existing approaches on shared-memory multi-core systems.

Keywords: Stochastic gradient descent, Parallel shared memory system, Matrix completion, Performance analysis, Load balancing.

ÖZET

ÇOK ÇEKİRDEKLİ SİSTEMLERDE PARALEL OLASILIKSAL GRADYAN ALÇALMA

Selçuk Gülcan
Bilgisayar Mühendisliği, Yüksek Lisans
Tez Danışmanı: M. Mustafa Özdal
Eylül 2020

Tezin odak noktası Olasılıksal Gradyan Alçalma (SGD) algoritmasının çok çekirdekli sistemlerde parallelleştirilmesidir. Asenkron yöntemler ve 2 boyutlu ızgara bölümlemeden yararlanan blok tabanlı yöntemler, algoritmanın paylaşımlı bellekli sistemlerde paralelleştirilmesi için kullanılan yaygın yaklaşımlardır. Asenkron yöntemler bellek üzerindeki düzensiz erişimleri sebebiyle ön-bellek sorunlarıyla karşılaşmaktadır. Izgara tabanlı yöntemlerde ise işlek arasında yük dengesizliği problemi görülebilmektedir. Tezimizde mevcut paralel SGD algoritmalarının performans seviyeleri ve paralel darboğaz noktaları incelenmiştir ve bu darboğaz noktalarını hedef alan yeni algoritmalar önerilmiştir. 2 boyutlu bölümlemede işlekler arasında yük dengesizliği problemini çözmek için kutulama tabanlı algoritmalar önerilmiştir. Belleği etkili bir şekilde kullanmak için de ızgara tabanlı asenkron paralel SGD algoritması önerilmiştir. Bu algoritma belleği daha etkili kullanmak için sıfırdışı güncelleme sırasını, faktör güncelleme sırasını bozmadan değiştirmekte ve buna uygun olarak saklı faktör matrislerini bellek üzerinde değiştirmektedir. Elde ettiğimiz deney sonuçları, önerdiğimiz yöntemlerin alternatif yöntemlere göre önemli ölçüde daha iyi çalıştığını göstermektedir.

Anahtar sözcükler: Olasılıksal gradyan alçalma, Paralel paylaşımlı bellekli sistemler, Matris tamamlama, Performans analizi, Yük dengeleme.

Acknowledgement

I would like to express my deepest gratitude to my advisors, Assoc. Prof. Dr. M. Mustafa Özdal and Prof. Dr. Cevdet Aykanat, for their guidance, support, and patience. This was all possible thanks to their extensive knowledge and their efforts to help me through my research. I would also like to thank Prof. Dr. Özcan Öztürk and Asst. Prof. Dr. Tayfun Küçükyılmaz who kindly agreed to be in my thesis committee.

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

I would like to thank former members of the room-51 squad, Fatih Atalı, Celal Selçuk Karaca, Berkay Kebeci, Halil İbrahim Özercan, and Şeref Ali Yataman. Thanks to the pure friendship of these brilliant people, I found myself motivated to keep going.

Lastly, I give my special thanks to my parents for their endless support and unconditional love. I'm grateful that they support me even when they don't understand my decisions.

Contents

List of Figures

List of Tables

Chapter 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 $V = (v_{ij})$ in which rows represent users and columns represent movies. A nonzero v_{ij} is the rating score of user i to movie j within some predefined range. 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.

Netflix competition [4–6] shows that low-rank matrix completion techniques, finding two low-rank P and Q matrices such that their product is an approximation of nonzeros in the 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 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. SGD algorithm solves the problem by making noisy but quick updates to

the P-matrix rows and Q-matrix rows.

The sequential nature of the algorithm and ever-increasing data lead researchers to search for better parallel methods for SGD. 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 the distributed-memory system 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. On the other hand, shared memory studies offer various methods for improvements in accuracy and convergence but they often lack quantitative analysis on underlying reasons for such improvements. To form a better understanding of current methods, we analyzed and compared them under the same environment on real datasets with code profiling tools and in light of gathered results, we propose load balancing methods for task mapping and a locality aware task scheduling.

The prior research has shown that SGD can be applied to parallel systems with 2D grid-based and non-locking approaches. The present work advances those studies by offering novel algorithms to alleviate two main performance-limiting problems of those methods: load balancing problem among threads and cache underutilization. Proposed bin packing based algorithms balance the works of the threads better than commonly used permutation method in the 2D grid partitioning-based parallel SGD approach and consequently they provide more performance. Locality aware task scheduling algorithm, which will be reffered as LASGD, is proposed to utilize memory better by carefully changing the rating matrix memory layout and nonzero update sequence without breaking the main properties of the algorithm. The mentioned 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. Ideas behind the proposed algorithms are often supplemented with code profile results to show how they solve bottleneck problems.

The organization of the paper is as follows: Chapter 2 gives background information on SGD and how it is used to complete sparse matrices while introducing related notations. A literature survey is included in Chapter 3 in which an embarrassingly parallel method called Simple-Parallel SGD, a lock-free SGD implementation called Hogwild [11], and 2D grid partitioning methods [10,12,13] are discussed. Proposed load balancing methods are explained in Chapter 4. Another proposed method, LASGD, is described in Chapter 5. We explain our common experiment setup that is used to compare different methods, datasets, and tools to measure the bottleneck and performance of the methods in Chapter 6. And lastly, experiment results and related discussions are given in Chapter 7.

Chapter 2

Matrix Completion with Stochastic Gradient Descent

Given an $m \times n$ matrix $V = (v_{ij})$, matrix completion is defined as the task of reconstructing the matrix $\hat{V} = (\hat{v}_{ij})$, where m denotes the number of users, n denotes the number of items, v_{ij} is a true rating value of corresponding user-item pair, $\hat{v}_{ij}$ is the estimate rating value in reconstructed matrix $\hat{V}$. v_{ij} also refers to a nonzero entry of the rating matrix. True ratings are either given by the users explicitly or inferred by the system from the actions or behaviours of the users. The loss function for the problem is usually defined as the sum of squared errors which yield to the following minimization :

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

Since the aim is to estimate unseen ratings as correctly as possible; a portion of data, known as the test set, is set aside and used only to evaluate the quality of the solution. A common evaluation metric is the root squared mean error (RMSE) calculated on the test set as follows:

$$RMSE = \sqrt{\frac{1}{\mid V \mid} \sum_{v_{ij} \in V} (v_{ij} - \hat{v}_{ij})^2}.$$

Low-rank matrix completion methods solve this problem by finding two low-rank matrices, P and Q, such that their matrix multiplication gives reconstructed matrix, $\hat{V}$. A row of $P_{m \times f}$ and a row of $Q_{n \times f}$ represent a user and an item respectively as a latent factor vector of size f. The latent factor size, f, is much smaller than n and m and tuned to get better root mean squared errors. With this setup, any rating in the matrix can be estimated by calculating the dot product of its corresponding user vector, p_i, and item vector, q_i^T. Figure 2.1 shows this relationship between rating, user and item matrices.

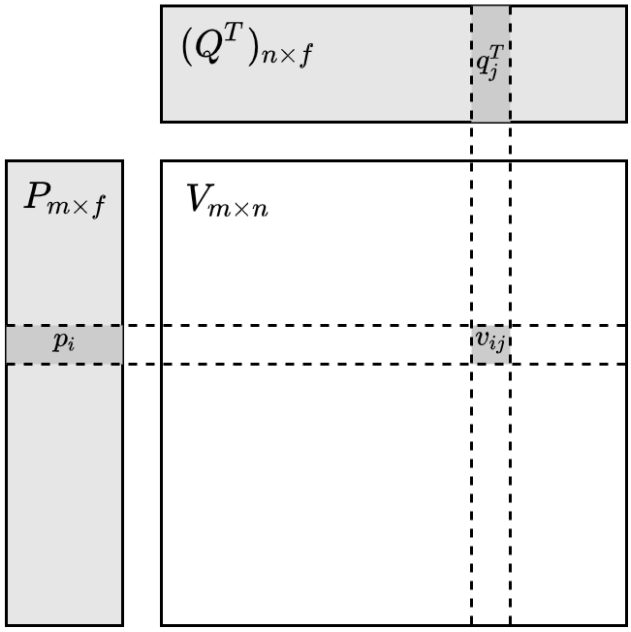


Figure 2.1: Rating matrix V and latent factor matrices P and Q

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 its gradient. The learning rate hyperparameter, γ, controls how much the variables are changed at each step. It can be

Table 2.1: The Notation Table

Symbol	Description
V	Rating matrix (Also denotes the set of nonzero ratings)
m, n	Number of rows and columns of V matrix
v_{ij}	Nonzero entry at row i and column j
$\hat{v}_{ij}$	Estimate for v_{ij}
r_i, c_j	i^{th} row and j^{th} column of V matrix.
	Also denotes the set of nonzeros in r_i and c_j
P, Q	Latent mactor matrices
p_i, q_j	i^{th} and j^{th} row of P-matrix 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 V matrix
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	α^{th} row slice, β^{th} 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 respected set

a constant value determined before the execution or dynamic to the current iteration number or previous gradient value. The variables are continuously updated until a stopping criterion is satisfied. Some implementations stop updating after a defined number of iterations. Others continue doing updates until the change in the gradients becomes smaller than a defined value or a desired test error level is reached.

Loss function for low rank matrix completion problem is can be written as

$$error = \sum_{i,j \in V} (v_{ij} - p_i q_j^T)^2 + \lambda(\|P\|^2 + \|Q\|^2)$$

Here, λ 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 iteration, a random nonzero is selected and P-matrix and Q-matrix rows are updated for this nonzero only as shown in Algorithm 1. Since other nonzero elements are not taken into account, updates of SGD may not be directed to the point of convergence but studies show that the performance gain of quick updates surpasses the error caused by noisy gradient and 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 that SGD can give better results due to the fact that it can make many more updates in the given time.

Algorithm 1: SGD Algorithm

Input: $V_{m \times n}$, f, γ, λ, *iterCount*
Initialize $P_{m \times f}$ and $Q_{n \times f}$;
repeat *iterCount* **times**
  forall $v_{ij} \in V$ *in* $\langle V \rangle$ *order* **do**
    $e_{ij} \leftarrow v_{ij} - p_i q_j^T$;
    $p_i \leftarrow p_i + \gamma(e_{ij} q_j - \lambda p_i)$;
    $q_j \leftarrow q_j + \gamma(e_{ij} p_i - \lambda q_j)$;
  end
end

One fundamental property of the algorithm is the random selection of nonzeros

in the updating step. It is a theoretical constraint [18] 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 [18]. 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 training set.
- The sequence of nonzero updates should be randomized.

Chapter 3

Literature Survey and Discussions

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

3.1 Simple-Parallel SGD

The main problem when threads do concurrent updates on the same data without any synchronization is data conflicts. These data conflicts are read-after-write and write-after-write hazards. A read-after-write data hazard occurs when a thread reads old data before another thread updates it with the new result(s). A write-after-write hazard occurs when a thread overwrites data written by another thread.

In parallel SGD, read-after-write hazard causes some of the modifications to P-matrix and Q-matrix rows to be disregarded in the gradient calculation step.

Suppose that thread a reads P-matrix row p_i just before thread b updates that row to p_i^*. Now, thread a performs these calculations based on p_i so the contribution of p_i^* to the calculations will be lost. It degrades the convergence rate of the SGD algorithm since some updates are not taken into the account.

The other hazard, write-after-write, occurs after P-matrix and Q-matrix rows are updated and then some threads overwrite those rows based on their values before the update. It means that updates incurred by some ratings do not contribute to the final P-matrix and Q-matrix values. Both data problems cause threads to do extra computations to reach the convergence. To avoid this, Simple-Parallel SGD locks the rows of the P and Q matrices before doing 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 problem can occur so Simple-SGD solves both data hazard problems.

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 Simple-Parallel SGD 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. Note that the parts of Π ($V_t \in \Pi$) are mutually nonzero disjoint and their union gives the whole set of nonzeros of V.

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. Simple-Parallel SGD 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 utilizing $\langle V^* \rangle$ and Simple-Parallel SGD utilizing $\langle \Pi \rangle$ 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: Simple-Parallel SGD Algorithm

```
Input: ⟨Π⟩, Π, f, γ, λ, iterCount
Initialize P_{m×f} and Q_{n×f} ;
for each thread t in parallel do
    for t ← 0 to iterCount do
        forall v_ij of V_t in ⟨V_t⟩ order do
            Lock P- and Q-matrix rows, p_i and q_j;
            Update P- and Q-matrix rows, p_i and q_j;
            Unlock P- and Q-matrix rows, p_i and q_j;
        end
    end
end
```

Although this naive parallelization leads to an inefficient algorithm, we believe it provides some insights when the performances of other methods in the following sections are analyzed against the performance Simple-Parallel SGD.

Figure 3.1 displays the variation of the throughput performance of the Simple-Parallel SGD algorithm with an increasing number of threads on 10 SGD test instances given in Table 6.1. The throughput values are computed as the number of nonzero updates per second. As seen in Figure 3.1, Simple-Parallel SGD does not perform better than the sequential algorithm until 8 cores. Simple-Parallel SGD runs are slightly faster than the sequential algorithm on 8 cores and we see the benefit of parallelism after 8 cores. However, the speedup is far from the ideal speedup and we investigate the bottlenecks and potential improvements by using the vTune profiling tool as discussed in the following two paragraphs.

Table 3.1 reports the average total spin time and overhead time as percentage values. The spin time is defined as the time spent on 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. Figure 3.2 shows the lock wait probability 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-matrix or Q-matrix row when it is locked by another

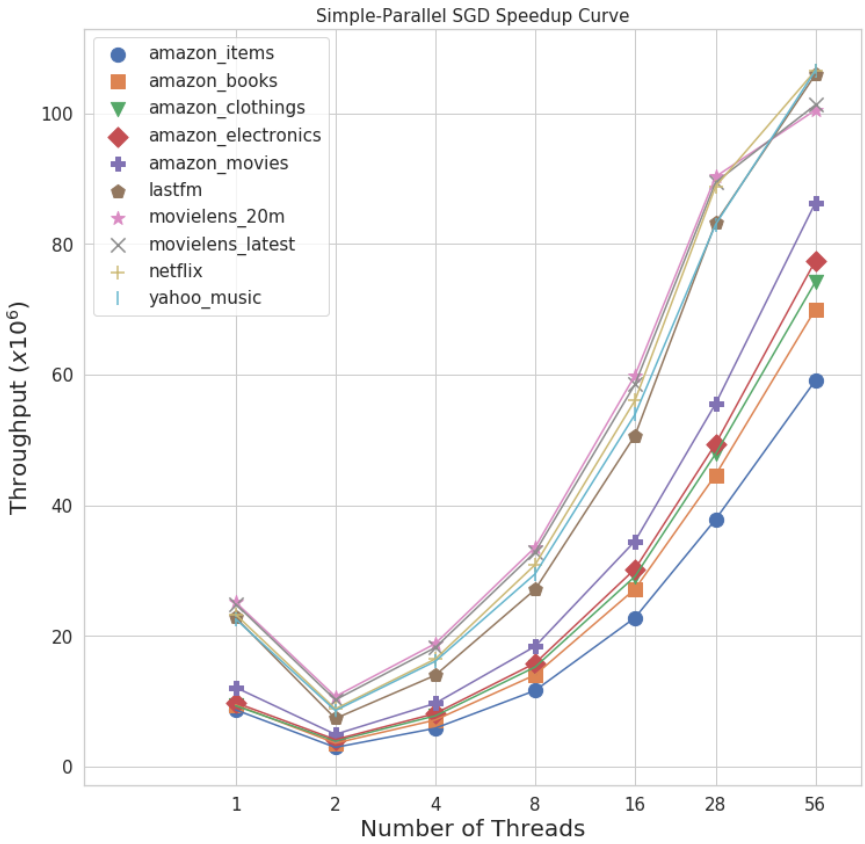


Figure 3.1: Simple-Parallel SGD throughput of processed nonzero as a function of the number of threads

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 Figure 3.2 and Table 3.1, lock waits are rare and vary between 0 and 0.04% depending on the density of the dataset. When we consider this result and the poor speedup performance in Figure 3.1 together, we can conclude that around 75-90% of CPU time is wasted to prevent data hazards 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. The next section covers a lock-free parallel SGD implementation called Hogwild [11] which shows that we can safely drop synchronization in favor of performance.

Table 3.1: 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_20m	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

3.2 Hogwild: Asynchronous SGD

Hogwild [11] states that possible data hazards that occur in parallel SGD are rare and SGD can be implemented without any locking mechanism. Hogwild theoretically proves that even though data hazards make SGD to converge slower, 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 algorithm and the Simple-Parallel SGD algorithm is the absence of locks on P-matrix and Q-matrix rows.

The main problem with this approach is the underutilization of cache due to cache coherence between private caches. When a nonzero is processed in the sequential algorithm, its corresponding P-matrix 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-matrix and/or Q-matrix rows concurrently.

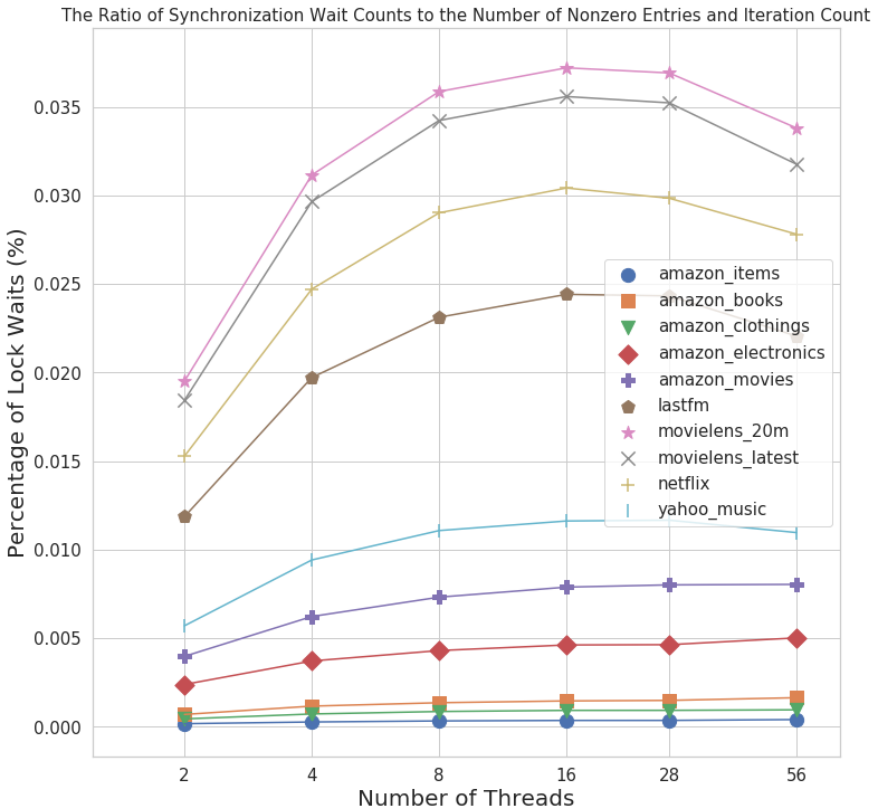


Figure 3.2: The wait probability of a nonzero in a single iteration

In multi-cores 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 the same probability to access a particular row or column in the Hogwild algorithm, the chance that a cache line invalidated increases as more threads are used. Although this data migration doesn't affect the algorithm in terms of correctness, it degrades the performance since accessing P-matrix and Q-matrix rows take longer times.

Algorithm 3: Hogwild Algorithm

Input: $\langle\Pi\rangle$, Π, f, γ, λ, *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**
      Update P- and Q-matrix rows, p_i and q_j;
    end
  end
end

3.3 2D Grid Partitioning

Another approach to decrease or eliminate data migrations other than locking P-matrix and Q-matrix rows is to enforce computing units to process on regions of the matrix that do not conflict with each other. 2D grid partitioning [10] is one of the popular approaches to divide the rating matrix into 2D blocks that do not interfere with each other. A 2D block is an intersection of a row slice and a column slice on the rating matrix. Although this scheme is originally proposed for distributed-memory SGD (DSGD) in [10], we utilize it for shared-memory parallelization. A set of blocks is called stratum if their row or column slices do not intersect. Since nonzeros in a singe stratum do not share any P-matrix or 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 obvious $T\times T$ 2D grid partitioning originally proposed for distributed-memory systems [10], there are $(T+1)\times(T+1)$ [12], $T\times 2T$ [19], and $T\times kT$ [13] partitioning schemes in the literature. The motivation of latter partitioning methods is mainly to provide better block scheduling to reduce threads' idle time in shared-memory systems.

In this work, we consider $T \times T$ partitioning where each row slice is assigned to a distinct thread. Figure 3.3 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 assigned to be processed by thread A, B, C, and D respectively. The numbers show different strata so that $\{A_k, B_k, C_k, D_k\}$ denote the set of blocks constituting stratum k for $k = 1, 2, 3, 4$. At each epoch, the block constituting the k stratum are processed concurrently in the k^{th} time period/subepoch.

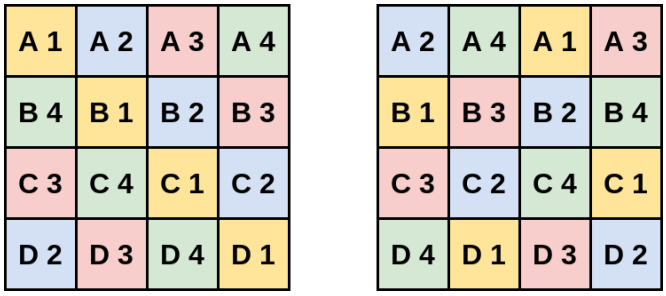


Figure 3.3: Two different block scheduling for 4×4 partitioning

The motivation behind row-vise partitioning of nonzeros among threads 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. It means that 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]. An example to such block scheduling where threads work on different row slices is given in Figure 3.4.

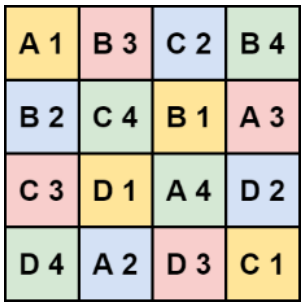


Figure 3.4: A block scheduling example for 4×4 partitioning. Threads can work on different row slices.

The random row and column permutation is assumed to attain balance on the nonzero counts of the 2D blocks.

A thread needs to wait for other threads after processing a block to avoid updates on the same P-matrix and Q-matrix rows before continuing on the next stratum. 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. 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, 20] make this block assignment decision in the run time. 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.

The main approach currently used is to adopt random 2D grid partitioning. In this method, the rows and columns of the rating matrix 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 and columns differ by at most 1.

Chapter 4

Load Balancing in 2D Grid Partitioning

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 block scheduling mechanism 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 if blocks contain unevenly partitioned nonzeros.
- **Asynchronous block scheduling** solves thread underutilization by keeping threads busy most of the time. The problem is that threads work on blocks that share P-matrix or Q-matrix rows. As Hogwild shows, this situation doesn't cause a convergence problem but working on the same P-matrix or Q-matrix rows creates data migrations that slow the execution due to cache coherence delays.
- **Standalone scheduler** assigns safe blocks to threads when they complete their work. Threads work on blocks that do not share any P-matrix or Q-matrix row at any time so it solves previous problems at the cost of

one extra thread that does the scheduling job. The main issue with this approach is that 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.

The random partitioning is a simple and fast technique and it creates fairly balanced blocks if nonzeros are distributed evenly on rows and columns. However, it scales poorly with an increasing number of threads. Also, nonzero distribution among rows and columns in real datasets usually follow the power law. It requires more advanced techniques to get balanced blocks on scale-free datasets when the number of threads is large.

In this section, we propose three load balancing algorithms based on bin-packing (BP) with a fixed number of bins [21]. 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.

Here 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: A relatively large number of sparse rows/columns has the potential of correcting the imbalance incurred by the assignment of dense rows/columns at the initial steps.

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

4.1 Independent Bin-packing Algorithm ($\mathbf{BP}_{r+c}$)

In this scheme, 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, 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. After each row/column-to-bin assignment, the weight of the respective bin is incremented by the degree of the assigned row/column.

The assignment of an V-matrix nonzero v_{ij} to a 2D-block is induced by the assignment of row i and column j to the bins in row and column assignment phases. That is, the assignment of row-i to bin B^r_α and column-j to bin B^c_β induces the assignment of nonzero v_{ij} to 2D-block $V_{\alpha,\beta}$. In other words, block $V_{\alpha,\beta}$ contains the V-matrix nonzeros at the intersection of rows and columns assigned to bins B^r_α and B^c_β respectively.

The details of the algorithm are given in Algorithm 5. The running time of the algorithm is $O(mlogm + nlogn + (m+n)logT + |V|)$. $|\cdot|$ denotes the number of nonzeros in the respected set. The first two terms come from sorting of the rows and columns according to their degrees. The third term comes from the selection of the best bin for each row and column assuming a binary-heap priority-queue implementation is used. The last term comes from computing the block assignment for each nonzero.

This method does not restrict blocks to have the same number of rows. As can be seen in Figure 4.1, block A1 has more rows and columns compared to block D1. Our algorithm allows such patterns as long as nonzero load balance among blocks is ensured. As a matter of fact, such patterns often arise in real sparse datasets.

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

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

Algorithm 5: BP_{r+c} Algorithm

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

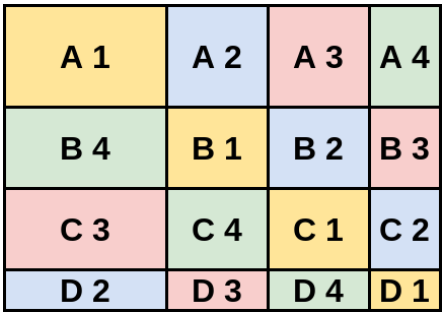


Figure 4.1: Example 4×4 partitioning achieved from proposed BP_{r+c} algorithm

4.2 Row/Column Assignment Respecting Bin Packing ($\text{BP}_{r\rightarrow c}$ and $\text{BP}_{c\rightarrow r}$)

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. In this work, we propose two BP-based algorithms that utilize the assignment information obtained in the first phase for the assignment in the second phase. Here and hereafter, these two algorithms will be referred to as $\text{BP}_{r\rightarrow c}$ and $\text{BP}_{c\rightarrow r}$. $\text{BP}_{r\rightarrow c}$ refers to performing row assignment in the first phase and column assignment in the second phase. $\text{BP}_{c\rightarrow r}$ refers to performing column assignment in the first phase and row assignment in the second phase. Here we describe only $\text{BP}_{r\rightarrow c}$ where a similar discussion holds for $\text{BP}_{c\rightarrow r}$ in a dual manner.

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

In the second phase, $\text{BP}_{r\rightarrow c}$ maintains a $T\times T$ 2D bin array, where $B_{\alpha,\beta}$ denotes the bin corresponding to the 2D-block $V_{\alpha,\beta}$. Maintaining this 2D bin array, necessities the utilization of row assignment information obtained in the first phase for computing the distribution of nonzeros in a particular column among the row slices obtained in the first phase. That is, we compute $|c_{j,\alpha}|$ which denotes the number of nonzeros in c_j that reside in R_α as follows:

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

At the beginning of the second phase, the weights of the $T \times T$ bins are initialized to zero.

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

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

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

According to the cost function defined in equation 4.1, the cost of assigning a column j to column slice C_β can computed as follows:

$$\begin{aligned} cost(c_j, C_\beta) &= Cost(\Pi_{new}) - Cost(\Pi_{cur}) \\ &= \sum_{\alpha=1}^{T} (((|B_{\alpha,\beta}| + |c_{j,\alpha}|)^2) - \sum_{\alpha=1}^{T} |B_{\alpha,\beta}|^2 \\ &= \sum_{\alpha=1}^{T} \left(|c_{j,\alpha}|^2 + 2|B_{\alpha,\beta}| \cdot |c_{j,\alpha}| \right) \end{aligned} \tag{4.1}$$

Here Π_{cur} denotes the nonzero-to-bin distribution obtained by the previous column-to-slice assignments, whereas Π_{new} denotes the nonzero-to-bin distribution to be obtained by assigning c_j to C_β in Π_{cur}. this assignment cost function 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.

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 column-to-slice 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 Equation 4.1.

The details of $\text{BP}_{r\to c}$ is given in Algorithm 6. The complexity of the algorithm is $O(mlogm+nlogn+mlogT+|V|+nT^2)$. The first two terms come from sorting the rows and columns according to their degrees. The third term comes from the selection of the best bin for each row in the first phase. The fourth term $|V|$ comes from computing all $c_{j,\beta}$ values since this requires scanning all nonzero of the V matrix. The last term comes from computing Equation 4.1 which takes $\theta(T)$ time, for each of the n column and for each of the T column slices.

Algorithm 6: $\text{BP}_{r\to c}$ 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**
    $B_{\alpha,\beta} \leftarrow \emptyset$
  end
end
Sort columns in descending order
for $j \leftarrow 1$ **to** n **do**
  for $\beta \leftarrow 1$ **to** T **do**
    $Cost_{j,\beta} \leftarrow \sum_{\alpha=1}^{T}(|B_{\alpha,\beta}| + |c_{j,\alpha}|)^2 - |B_{\alpha,\beta}|^2$
  end
  Find min s.t. $Cost_{j,min} \leq Cost_{j,\beta}$, $\beta = 1, 2, \ldots T$
  for $\alpha \leftarrow 1$ **to** T **do**
    $B_{\alpha,min} \leftarrow B_{\alpha,min} \cup c_{j,min}$
  end
  $map(c_j) \leftarrow min$
end
forall $v_{ij} \in V$ **do**
  $\alpha \leftarrow map(r_i)$
  $\beta \leftarrow map(c_j)$
  $V_{\alpha,\beta} \leftarrow V_{\alpha,\beta} \cup v_{ij}$
end

Chapter 5

Locality Aware Stochastic Gradient Descent (LASGD)

Two definitions are given to express the motivation and the main points of the proposed LASGD algorithm. These definitions assume there is a given random nonzero update sequence.

Definition 1. Valid SGD Update Sequence

For a given nonzero update sequence, there can be different nonzero update sequences such that the update order of P-matrix and Q-matrix rows are the same as the given original sequence. Each one of such sequences is defined as a Valid SGD Update Sequence. Since the order of updates on P-matrix and Q-matrix rows are exactly the same, latent factor matrices P and Q are the same as the original update sequence produces. That is, any valid SGD update sequence can be used in the SGD process instead of the original one.

Example 1. Consider a simple rating matrix shown on Figure 5.1. There are 2 rows with x and y indices and 2 columns with z and w indices. The matrix contains only 3 nonzeros. The numbers on nonzeros show their nonzero index and their original update sequence, $\langle v_1, v_2, v_3 \rangle$. So, first v_1 is updated then v_2 is updated and finally v_3 is updated in the original sequence. In Figure 5.2, a different valid SGD update sequence, $\langle v_2, v_1, v_3 \rangle$, is given.

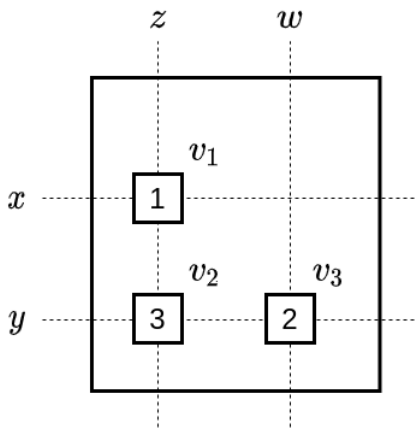


Figure 5.1: A simple rating matrix consisting of three ratings

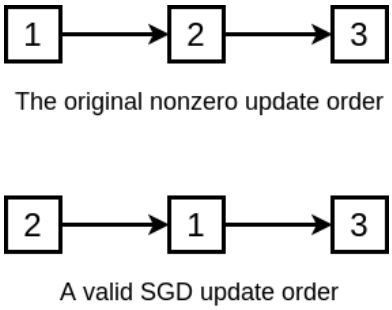


Figure 5.2: The original and another valid SGD update sequence

All update calculations on P-matrix and Q-matrix rows are shown to prove that $\langle v_1, v_2, v_3 \rangle$ and $\langle v_2, v_1, v_3 \rangle$ update sequences modify P-matrix and Q-matrix rows in exactly the same way. We encapsulate sgd calculation for P-matrix 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-matrix and Q-matrix rows as arguments in order to complete a single rating update. In the calculations, $p_x^{(t)}$ shows the state of x^{th} row of P-matrix after the row is updated t times. Equation 1 shows the updates on P-matrix and Q-matrix rows if the original update sequence, $\langle v_1, v_2, v_3 \rangle$, and the alternative update sequence, $\langle v_2, v_1, v_3 \rangle$ is used. Although latent matrix rows can be updated in different orders, P-matrix rows and as well as Q-matrix rows are exactly the same after all of three nonzero updates. It shows that $\langle v_2, v_1, v_3 \rangle$ is a valid SGD update sequence. In fact, it is the only valid SGD update sequence other than the original one.

$$
\begin{aligned}
&\langle v_1, v_2, v_3 \rangle && \langle v_2, v_1, v_3 \rangle \\
p_x^{(1)} &= sgd_P(p_x^{(0)}, q_z^{(0)}, v_1) & p_y^{(1)} &= sgd_P(p_y^{(0)}, q_w^{(0)}, v_2) \\
q_z^{(1)} &= sgd_Q(p_x^{(0)}, q_z^{(0)}, v_1) & q_w^{(1)} &= sgd_Q(p_y^{(0)}, q_w^{(0)}, v_2) \\
p_y^{(1)} &= sgd_P(p_y^{(0)}, q_w^{(0)}, v_2) & p_x^{(1)} &= sgd_P(p_x^{(0)}, q_z^{(0)}, v_1) \\
q_w^{(1)} &= sgd_Q(p_y^{(0)}, q_w^{(0)}, v_2) & q_z^{(1)} &= sgd_Q(p_x^{(0)}, q_z^{(0)}, v_1) \\
p_y^{(2)} &= sgd_P(p_y^{(1)}, q_z^{(1)}, v_3) & p_y^{(2)} &= sgd_P(p_y^{(1)}, q_z^{(1)}, v_3) \\
q_z^{(2)} &= sgd_Q(p_y^{(1)}, q_z^{(1)}, v_3) & q_z^{(2)} &= sgd_Q(p_y^{(1)}, q_z^{(1)}, v_3)
\end{aligned} \tag{5.1}
$$

Definition 2. Nonzero Update Graph

Nonzero Update Graph is a directed acyclic graph where each node represents a nonzero of the rating matrix. There is a row edge from node a to node b only if the following three conditions are met:

(i) Nonzeros represented by node a and node b reside in the same row.

(ii) Nonzero represented by node a appears before the nonzero of node b in the nonzero update sequence.

(iii) In the row that contains the nonzeros corresponding to a and b, there is no other nonzero between them with an update order that falls between the update orders of a and b.

This construction logic applies to column edges as well. Nonzero update graph satisfies the following properties:

- Each node has at most 2 incoming edges.
- Each node has at most 2 outgoing edges.
- Two nodes sharing the same incoming neighbor cannot have an edge between them.

The graph construction rules and the properties imply that a nonzero update graph is a sparse graph. The total number of edges in a nonzero update graph is upper bounded by 2 times the number of nonzeros.

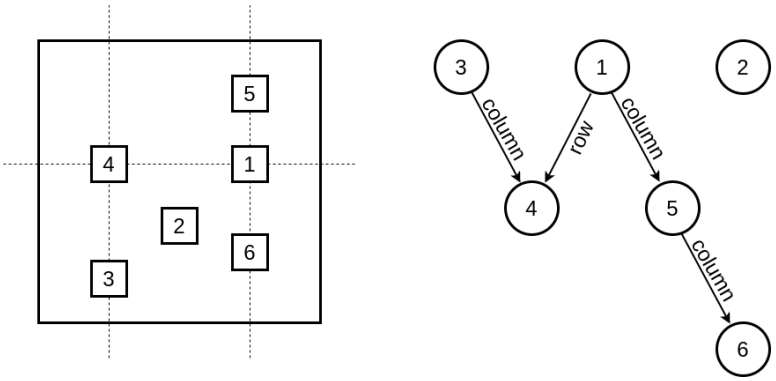


Figure 5.3: A sample rating matrix and its corresponding nonzero update graph. The numbers inside the nonzeros and nodes show the update order.

Example 2. Figure 5.3 shows an example rating matrix and its corresponding nonzero update graph. The original update sequence is given as $\langle v_1, v_2, v_3, v_4, v_5, v_6 \rangle$. Nonzero 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. Also, note that there is no edge between node 1 and 6 because of rule (*iii*) although they reside on the same column.

LASGD algorithm is based on the following observation about this graph: Any topological order of the nonzero update graph gives a valid SGD update sequence according to Definition 1. In other words, a nonzero update sequence generated from a topological order of the nonzero update graph can be used in SGD updates instead of the original sequence and the same result ($\hat{V}$) should be achieved.

Example 3. Figure 5.4 shows three different valid SGD update sequence for the example graph in Figure 5.3. There can be many other update sequences as well.

LASGD algorithm tries to pick a valid update sequence such that the machine can use memory hardware more effectively. Although LASGD is inherently an

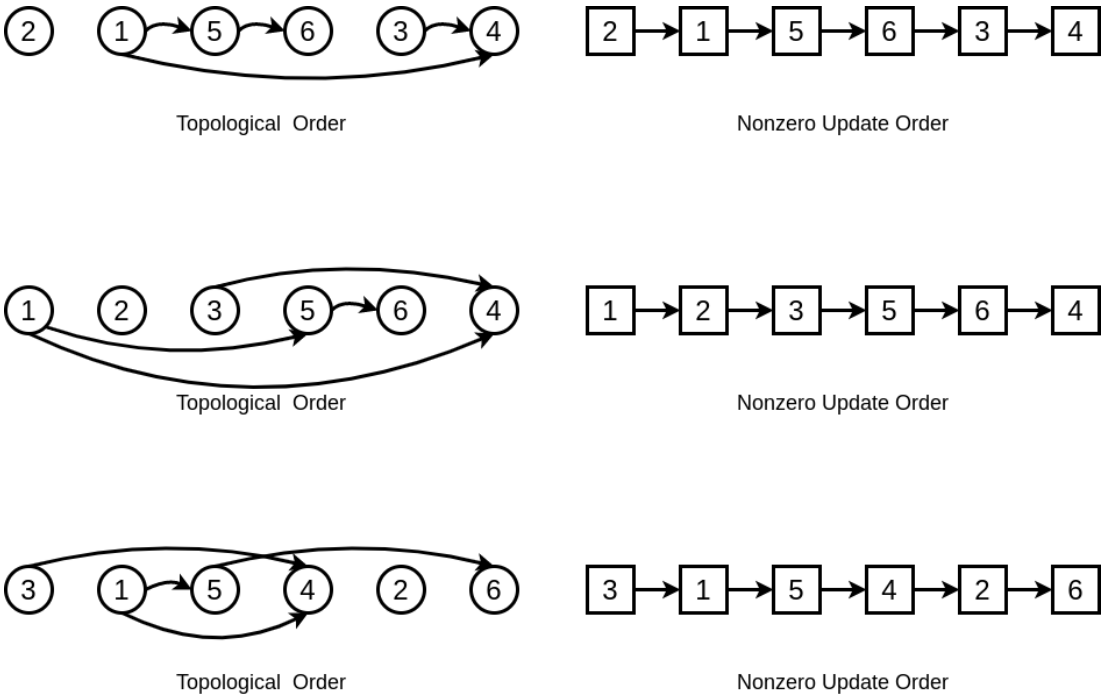


Figure 5.4: Three possible nonzero update sequences for the graph given in Example 2

optimization over sequential SGD algorithm, it can be adapted for parallel algorithms as well. The next section will describe LASGD in detail on the sequential case. After that, we'll discuss how it can be extended to cover parallel SGD methods that use grid-based partitioning.

5.1 LASGD on a single thread

The first step of LASGD is to create a nonzero update graph for V matrix and a given update sequence $\langle V \rangle$. Initially, the modified update sequence is empty. Nodes with no incoming edges create a set of candidate nonzeros for the next nonzero in the modified update sequence. When a nonzero is selected from this set, it is appended to the end of the update sequence being constructed and the corresponding node of the graph and outgoing edges of that node are removed from the graph. Neighbor nodes that do not have other incoming edges apart from edges that just removed are inserted to candidate nonzero set. This procedure is

repeated until the graph becomes empty. This selection obeys a topological order so the modified sequence we get at the end of the algorithm should be a valid SGD update sequence.

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

- **The corresponding P-matrix 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-matrix row or a 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.

- **The corresponding P-matrix or Q-matrix rows are close to recently used rows in DRAM** in DRAM, row buffer acts as a bridge between DRAM cells and memory bus. Accessing data that is already in the DRAM row buffer is faster than accessing . This is known as row buffer locality. According to our micro-benchmarks, 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-matrix or Q-matrix rows are not found in the cache, LASGD prefers nonzeros whose latent factors are close to the last used memory location. As it is indicated previously, these decisions are taken in the preprocessing step by modeling core caches and DRAM. Row buffer locality and our related micro-benchmark are discussed in Appendix A.

- **The corresponding P-matrix or Q-matrix rows are encountered a first time in ordering step** It is directly related to the previous point. If P-matrix 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 to the recently used DRAM location. By doing so, SGD can update the rows more quickly.

As Algorithm 7 explains, a score is given to each of the nonzeros in the candidate set to track listed conditions on nonzeros. At each step, the nonzero with the best score is placed into the SGD update sequence. After each selection, the candidate set and scores of nonzeros are updated.

Algorithm 7: LASGD Algorithm

```
Input: V_{m×n}, ⟨V⟩
Construct the nonzero update graph G = (U, E) from ⟨V⟩ ;
S ← the set of all nodes with no incoming edge ;
⟨V*⟩ ← empty list of the modified update sequence ;
forall v_ij in S do
    Calculate score_ij ;
end
while U is not empty do ;
    Insert v_ij to ⟨V*⟩ such that score_ij is maximum ;
    u ← the node of v_ij in the graph ;
    forall node v : edge (u, v) ∈ E do
        Remove edge (u, v) from the graph ;
        if v has no other incoming edges then
            Insert v into S ;
    end
    Remove node u from the graph ;
    Remove v_ij from S
    forall v_ij in S do
        Update score_ij ;
    end
    SGD(V_{m×n}, ⟨V*⟩)
```

5.2 LASGD on multiple threads

The main problem encountered when adapting LASGD to parallel methods is that the actual update sequence isn't known beforehand. For two nonzeros assigned to different threads, one can be processed earlier in one execution and the other

can be processed earlier in another execution depending on the execution speed of instructions of assigned cores. This situation makes it difficult to construct a nonzero update graph since we cannot determine nodes connected with edges and the direction of the edges.

We can resolve this problem by using 2D grid-based parallel methods in which the nonzero update sequence is clear in block level instead of using methods like Hogwild whose nonzero update sequence is more chaotic. If nonzero update graph generation rules are analyzed carefully, it can be understood that LASGD doesn't need to know the nonzero update sequence completely. the relative update order between independent nonzeros, nonzeros that do not share a P-matrix or a Q-matrix row, is insignificant for the algorithm since corresponding nodes in the graph are not connected at all. Grid-based methods with synchronous block scheduling ensure that threads work on independent blocks at all times. If cores are homogeneous and blocks are sufficiently balanced, the statement is expected to be true for grid-based methods with asynchronous scheduling as well. If we go over Figure 3.3, the nonzeros in block A1 are independent of nonzeros that are updated in the same time interval (nonzeros in B1, C1, and D1 blocks).

Because of this observation, LASGD can create a nonzero update graph by assuming there is some relative order between nonzeros processed in the same time interval as long as the update order in a single block is protected. No matter what order is assumed between blocks in the same time interval, there is only one nonzero update sequence that can be generated. Therefore, the rating matrix is partitioned with a grid partitioning method in order to apply LASGD when multiple threads are used. Then, an arbitrarily order is assumed to exist between blocks updated at the same time interval. After that, the same LASGD procedure for a single thread is applied. Any load balancing method can be used in the grid partitioning step. We used the $\text{BP}_{r\rightarrow c}$ algorithm since it generates better-balanced blocks (as shown in Table 7.1) than alternative methods do.

Chapter 6

Experiment Setup and Datasets

We use dual Intel Xeon Platinum 8280 CPUs having 28 physical cores each. 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.

All implementations in the experiments are written in C++11, OpenMP [22] library is used for parallelization. Result figures are plotted with the help of Matplotlib library [23]. Some papers we investigated do not offer an open implementation or they are written for distributed systems only but still applicable to shared memory systems. Therefore, we needed to code those methods. In fact, we have coded all compared method from scratch even if an implementation has already existed because of the following reasons:

- Implementations may have extra optimizations or bottlenecks that are not directly related to the method. This would be advantageous or disadvantageous for some methods.
- Parameters are defined differently in the papers, it is not easy to set the same latent factor size, learning rate, and regularization parameters without modifying existing code.
- Although the main SGD update operations are the same, there are slight

changes like weight initialization and stopping criteria that prevent consistent results between implementations.

- Several implementations do extra computations like baseline bias calculations and some do not calculate them. Although they can give the same result by setting bias weights to 0, they may still do calculations which makes fair comparison difficult to do.

- Out intend 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.

We prefer using coordinate list (COO) sparse matrix format to store the rating matrix as it is more flexible for permuting nonzero elements. The COO format is known to have an expensive lookup time but SGD doesn't need to access a specific cell in the rating matrix.

P-matrix and Q-matrix are initialized uniformly at random between 0 and 1. All evaluation error results are calculated with the RMSE metric on a test set which is around 20% of all nonzero ratings. the latent factor size f is chosen as 16.

Code Profilings are done with Intel's vTune Profile tool [24].

6.1 Datasets

Several datasets with different attributes and domains are used to preserve the fairness of our experiments because some algorithms may favor datasets having a certain pattern. The statistics for selected datasets are reported in Table 6.1. Row count (m), col count (n), nonzero count, ($\mid V \mid$), density values and row/column degree information of V matrix are reported. We use the following definition of density.

$$d = \frac{\mid V \mid}{m \times n}$$

Although original datasets are in different formats, we processed them to be in the matrix market exchange format.

The selected datasets are as follows: amazon_item_dedup, amazon_books, amazon_clothing_shoes_and_jewelry, amazon_movies_tv, lastfm, movielens_20M, movielens_latest, netflix, yahoo_music. The main categories are shopping, movies, and music. Netflix and yahoo datasets are used in competitions of Netflix Challenge and KDD cup respectively. Beside them, we tried to use less commonly known datasets like amazon-electronics [15]. Although they have extra attributes like timestamp, category that would be useful for prediction, only rating values, user-id values, and item id values are considered in the experiments.

We permute the update sequence of nonzeros to achieve two stochastic properties mentioned in Section 2 instead of selection with replacement. 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, row and column indices of nonzeros are also changed. In the raw data files, rows and columns are arranged with respect to the number of ratings they have. For example, The row with the most number of ratings is the first row and the density of rows decreases from top to bottom. We find out that such patterns affect the performance of algorithms differently and we don't want methods to rely on a particular distribution. If a method needs a special matrix layout to work or perform better then we expect the method itself to preprocess the matrix. So, we swapped indices of rows and columns with random valid indices to break the pattern as shown in figure 6.1.

6.1.1 Matrix Nonzero Distribution

We can learn more about the characteristics of the datasets by inspecting their row and column degree distributions. Figure 6.1.1 shows row and column degree

Table 6.1: Dataset Properties

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

min: Minimum, max: Maximum, cv: Coefficient of Variance

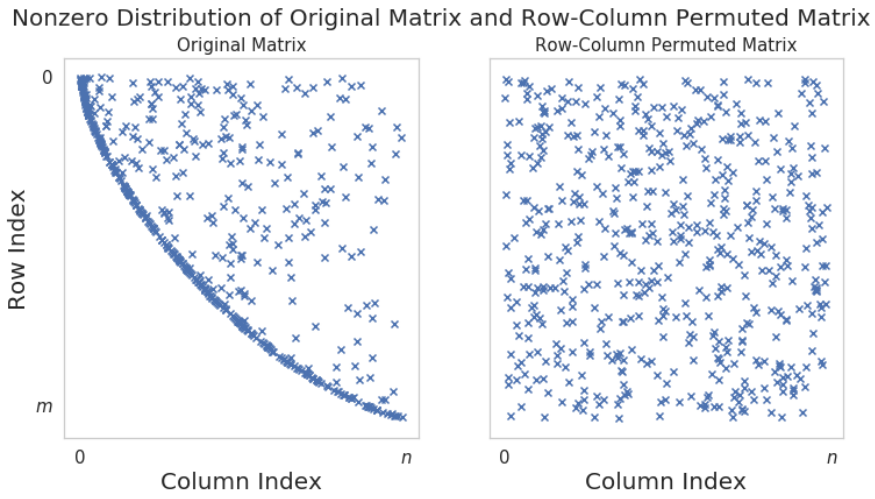
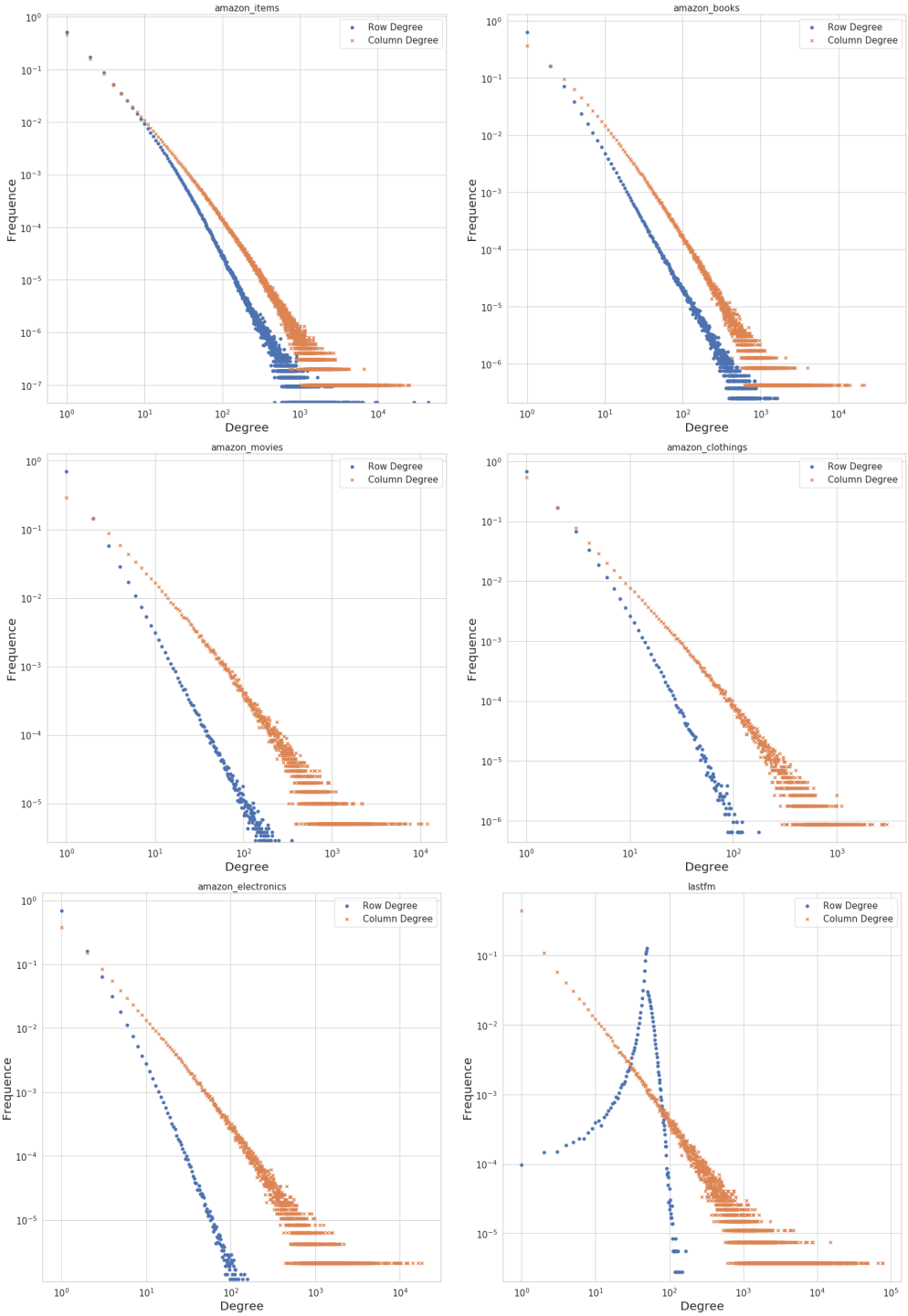


Figure 6.1: A comparison between original data and index shuffled data on the distribution of nonzeros across the matrix, only a small fraction of data depicted for the sake of clarity.

distributions of ten datasets when both axes are *log*-scaled. As seen on the figure, both row and column degree distributions follow a linear pattern for most of the datasets. This is a well-known property of scale-free networks. Scale-free networks [25] are graphs such that their degree distribution follows a power-law. In other words, if we denote k as the degree of a node and $P(k)$ as the frequency of nodes having k number of edges, there is a power-law relationship between k and $P(k)$ with some constant γ. This power-law relation becomes a linear relation when *log* function is applied to both sides. That is why linear patterns show up in the figure.

$$P(k) \sim k^{-\gamma}$$


amazon_items
amazon_books
amazon_movies
amazon_clothings
amazon_electronics
lastfm
Row Degree
Column Degree
Frequence
Degree

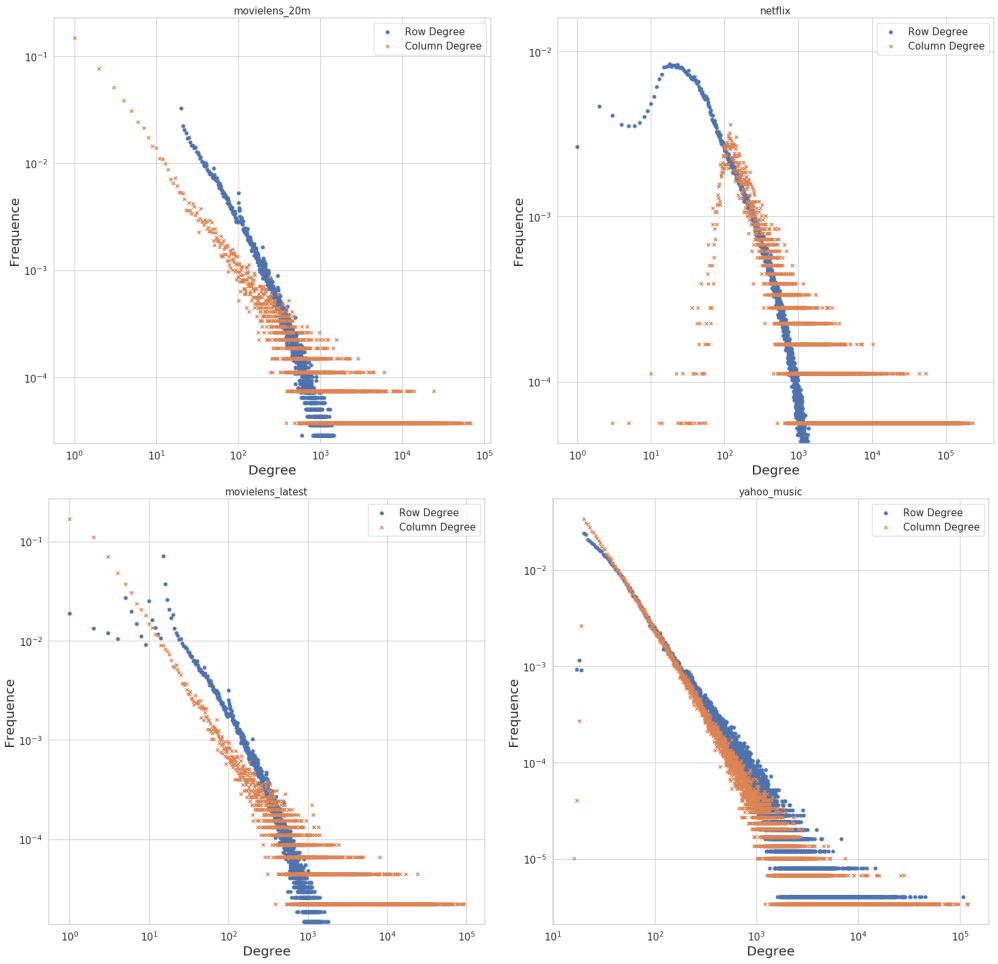


Figure 6.2: Row-column degree distribution of individual datasets

What this relation means for the rating matrices is that most of the rows and columns are very sparse and few rows and columns contain most of the ratings. This "rich-get-richer" phenomenon causes nonzero distribution of V matrix rows and columns to be skewed. Skewed rows and columns cause load balance problems in the random partitioning algorithm as discussed in Chapter 4.

The row degree distribution of lastfm dataset does not follow the mentioned power-law pattern. As Figure 6.1.1 show, its degree distribution resembles the bell shape. Other than this exception, all datasets follow scale-free power-law degree distribution with γ varying between 0.36 and 3.07.

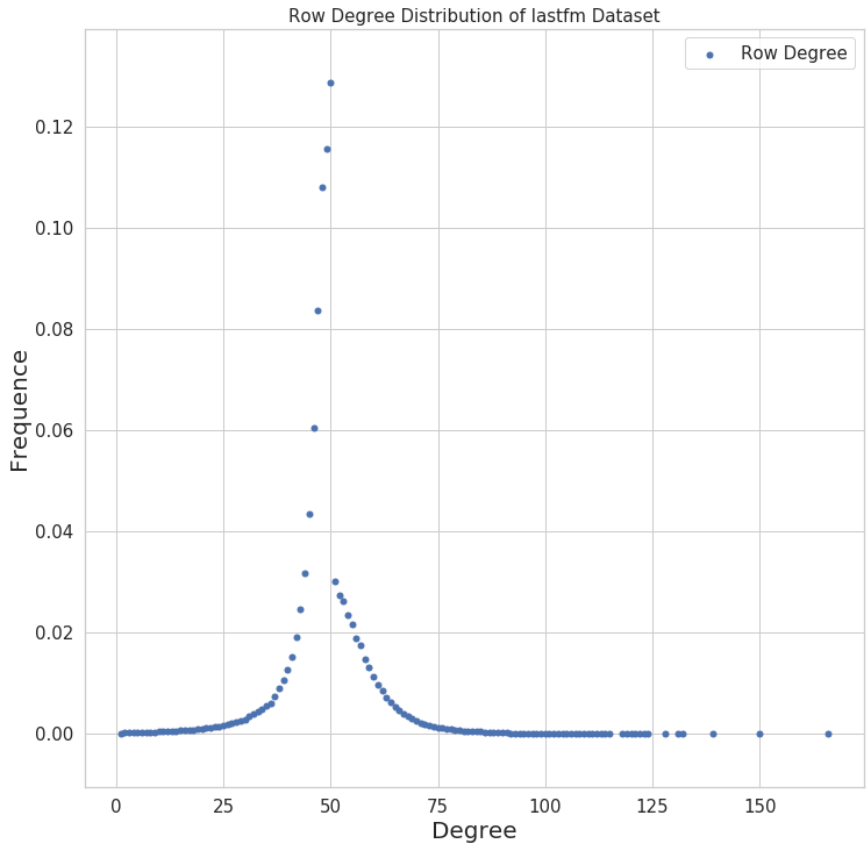


Figure 6.3: Row degree distribution of lastfm dataset does not follow power law

6.2 Experiment Parameters

The learning rate, regularization parameter, and iteration count values used in our experiments are listed in Table 6.2 with their corresponding RMSE values. RMSE values we got seems to be consistent with the results of other studies [12, 26], although the learning rate, regularization parameter, and iteration count differ due to the implementation differences.

Table 6.2: Parameters and RMSE Values for Each Dataset

Dataset	Learning Rate	Regularization	Iteration Count	RMSE
amazon_item_dedup	0.0030	0.10	30	1.22
amazon_books	0.0030	0.10	40	1.10
amazon_clothing	0.0030	0.10	40	1.31
amazon_electronics	0.0030	0.10	30	1.38
amazon_movies	0.0030	0.10	30	1.18
lastfm	0.0010	0.20	40	1.92
movielens_20m	0.0030	0.01	80	0.79
movielens_latest	0.0030	0.01	80	0.78
netflix	0.0030	0.10	60	0.84
yahoo_music	0.0001	0.05	60	23.30

6.3 Implementation Detail: Contiguous Memory

The hardware of memory structures is optimized for linear accesses. In 2D grid partitioning methods, the corresponding P-matrix and Q-matrix rows of nonzeros in a single 2D-block should be contiguous in the main memory in order to utilize memory better. An example of contiguous and non-contiguous memory layouts is given in Figure 6.3 assuming that the rating matrix is partitioned into 4 threads. The color of a nonzero represents the thread the nonzero is assigned and the numbers show the update order of the nonzeros.

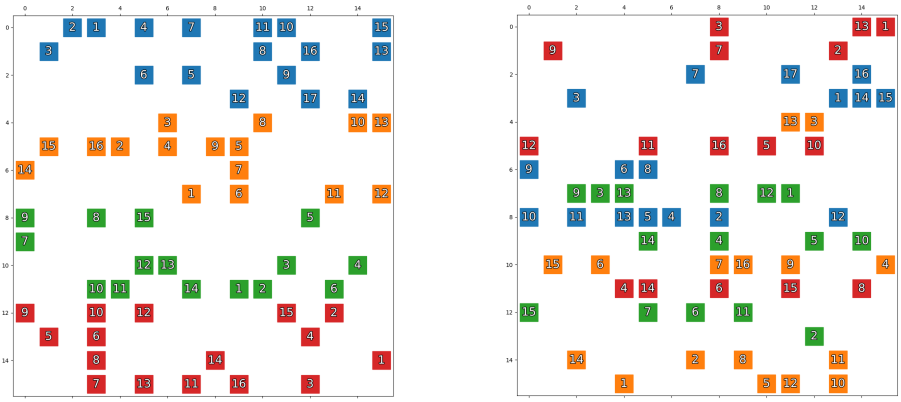


Figure 6.4: Example of contiguous memory layout (left) and non-contiguous memory layout (right) when a 2D grid partitioning method is used.

Placing P-matrix and Q-matrix rows of nonzeros in a 2D block close to each other allow the program to utilize spatial cache locality and row buffer locality better. Depending on the factor size, contiguous memory layout increases the probability of CPU cache hit and/or the probability of row buffer hit while processing nonzeros in a 2D block.

If P-matrix and Q-matrix are not aligned to cacheline size or a row of P and Q matrices is smaller than a cacheline, a false sharing problem can occur in non-contiguous memory layouts. False sharing is a data conflict between threads over a single cacheline such that a thread modifies a part of the cacheline and another thread modifies another part of the same cacheline. Although they don't share any real data, data migration still occurs because they update the same cacheline. Contiguous memory layout solves this problem completely since nonzeros processed in the same subepoch by different threads are placed into different cachelines.

For these reasons, we use contiguous memory layout in all 2D grid-based implementations. Given a partition Π, it is always possible to arrange the rating matrix by moving rows and columns of the rating matrix as shown in figure 6.3.

Chapter 7

Results

Table 7.1 shows how good random permutation, BP_{r+c}, $\text{BP}_{r\to c}$ and $\text{BP}_{c\to r}$ algorithms balances the block partitions using datasets mentioned in Section 5. As the table shows, the proposed algorithms are superior compared to the random permutation method especially when the number of blocks becomes large. Another observation is that matrices that our algorithms partition better by a huge margin are dense matrices whose number of rows and columns is low compared to the number of nonzeros. In sparse matrices, random permutation also generates fairly balanced blocks. It starts to perform worse as the dataset gets denser and the number of partitions increases.

It may seem counter-intuitive that independent row and column partitioning (BP_{r+c}) gives balanced blocks. The algorithm assumes that nonzero distribution in a column is independent of the number of nonzeros in the column. It looks a strong assumption but table 7.1 shows that it provides a significant improvement over the random partitioning method and it also achieves a rather good balance for a small number of threads.

Figure 7.1 and 7.3 compares proposed SGD algorithms (BP_{r+c}, $\text{BP}_{r\to c}$, LASGD) and existing algorithms (Simple-Parallel SGD, Hogwild, Grid with random permutation) as the number of threads increases. Figure 7.2 analyzes the

Table 7.1: Comparing bin packing based load balancing algorithms against random permutation algorithm. Values indicate the ratio between the number of nonzeros of the block with largest number of nonzeros and the block with least number of nonzeros

	4×4				16×16				64×64			
Dataset	RP	BP_{r+c}	$BP_{r \to c}$	$BP_{c \to r}$	RP	BP_{r+c}	$BP_{r \to c}$	$BP_{c \to r}$	RP	BP_{r+c}	$BP_{r \to c}$	$BP_{c \to r}$
amazon_item_dedup	1.013	1.001	1.000	1.000	1.059	1.009	1.000	1.000	1.172	1.048	1.000	1.000
amazon_books	1.037	1.003	1.000	1.000	1.112	1.018	1.000	1.000	1.361	1.096	1.000	1.000
amazon_clothing	1.015	1.003	1.000	1.000	1.083	1.034	1.000	1.000	1.368	1.184	1.001	1.001
amazon_electronics	1.038	1.003	1.000	1.000	1.175	1.029	1.000	1.000	1.639	1.156	1.001	1.001
amazon_movies	1.067	1.004	1.000	1.000	1.227	1.031	1.000	1.000	1.834	1.189	1.000	1.001
lastfm	1.083	1.001	1.000	1.000	1.490	1.021	1.000	1.000	2.366	1.108	1.000	1.001
movielens_20m	1.071	1.001	1.000	1.000	1.443	1.015	1.000	1.000	3.902	1.088	1.000	1.002
movielens_latest	1.143	1.002	1.000	1.000	1.449	1.015	1.000	1.000	3.330	1.085	1.000	1.000
netflix	1.127	1.001	1.000	1.000	1.491	1.006	1.000	1.000	2.543	1.046	1.001	1.000
yahoo_music	1.101	1.001	1.000	1.000	1.336	1.009	1.000	1.000	1.855	1.059	1.000	1.000

RP denotes the random permutation method. BP_{r+c} denotes independent bin-packing method. BP_{r+c} denotes row then column assignment bin packing method. BP_{c+r} denotes column then row assignment bin packing method.

sequential case between LASGD and sequential SGD. Note that, Hogwild and other grid-based parallel methods are reduced to sequential SGD when only one thread is used.

Experiment results show that the proposed methods perform better compared to Hogwild and grid approach with the random permutation partitioning. The fact the performance difference between proposed and existing methods increases as the number of threads increases indicates our methods are more scalable. On 56 threads, the LASGD algorithm gets more than 3 times more throughput than the Hogwild method and gets around 1.5 times more throughput than the random permutation method.

Another observation is about the performance scaling of Hogwild. The throughput of Hogwild doesn't change between 28 threads and 56 threads. When code profiling results and the architecture of the experiment 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 data migrations between L1 or L2 caches become migrations between L3 caches when 56 threads are used. Since data migration between L3 caches is a lot more costly than data migration between L1 or L2 caches, almost no speedup is achieved. We can see this problem doesn't occur in grid partitioning methods because data migration is avoided due to threads running on independent blocks.

When the BP_{r+c} algorithm is compared with the $\text{BP}_{r\rightarrow c}$ algorithm, $\text{BP}_{r\rightarrow c}$ performed slightly better, which is consistent with Table 7.1. However, the preprocessing time of $\text{BP}_{r\rightarrow c}$ is higher so BP_{r+c} can be preferable for some dataset since its grid partitioning is simple and fast.

Figure 7.2 shows there is significant improvement gained with Sequential LASGD with amazon datasets. It still performs better than sequential SGD on movielens, netflix, lastfm, and yahoo but the difference is not much.

When speedup curves for different datasets are examined separately, it becomes easier to understand why some ideas work better for some datasets. The speedup

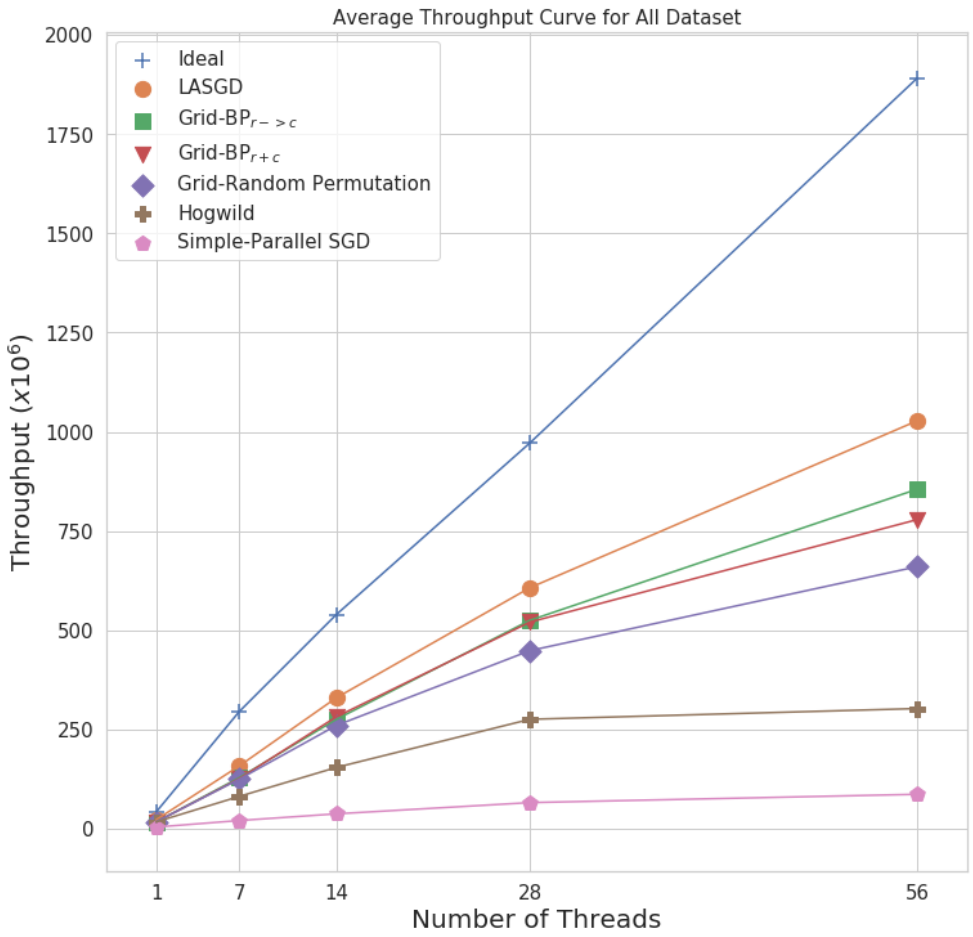


Figure 7.1: Comparison of proposed SGD algorithms against existing algorithms in terms of average throughput values with respect to increasing number of threads

curves for netflix and movielens dataset almost reach the ideal speedup. On this dataset, there is an immense improvement gained through proposed load-balanced algorithms over using the random permutation method. It is seen from Table 7.1 that netflix and movielens dataset suffer most from the imbalance between blocks so it is logical to see that better load balancing methods like BP$_{r+c}$ and BP$_{r\to c}$ affect the performance extremely.

Another observation is that the proposed load balancing methods favor dense datasets more than the sparse ones while the memory access improving method (LASGD) performs much better on sparse datasets. Since there are fewer P-matrix and Q-matrix rows compared to the total number of nonzeros in a denser

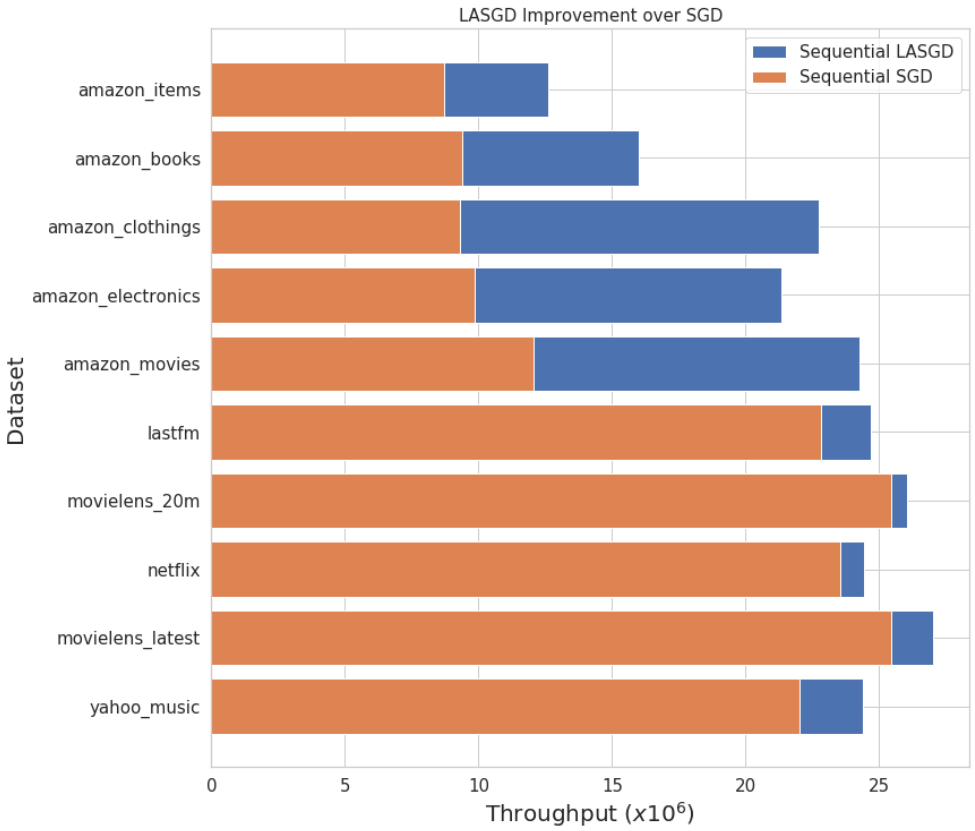
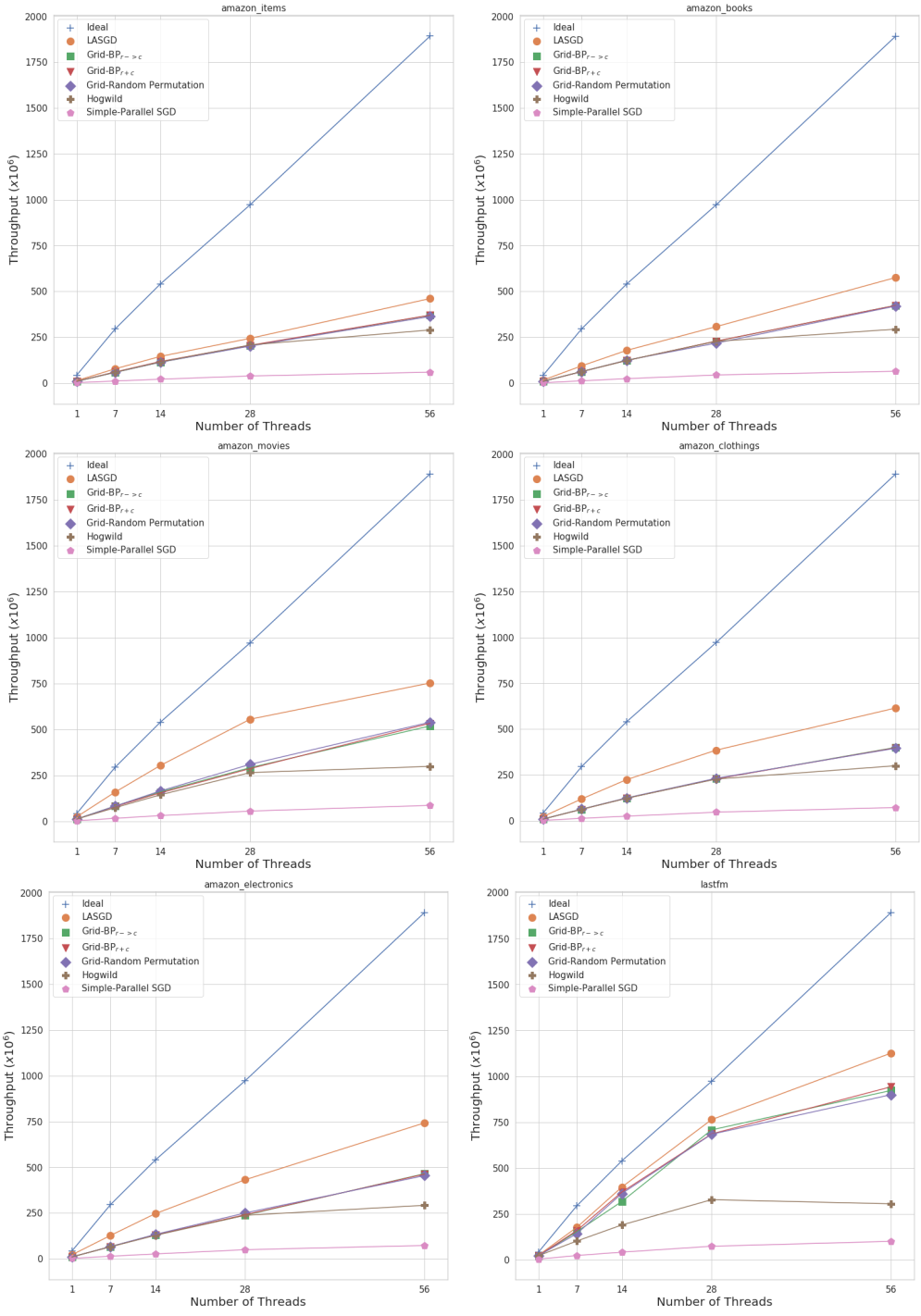


Figure 7.2: Sequential LASGD results compared to Sequential SGD results

dataset like movielens_20m, movielens_latest, and netflix, it is possible that load balance becomes the main bottleneck. On the other hands, sparse datasets like all amazon matrices contain more P-matrix and Q-matrix rows relative to the number of nonzeros so a load imbalance is hard to occur and doesn't affect the performance as much as memory access patterns to P-matrix and Q-matrix rows.


amazon_items
amazon_books
amazon_movies
amazon_clothings
amazon_electronics
lastfm
Ideal
LASGD
Grid-BP$_{r->c}$
Grid-BP$_{r+c}$
Grid-Random Permutation
Hogwild
Simple-Parallel SGD
Throughput ($x10^6$)
Number of Threads
0
250
500
750
1000
1250
1500
1750
2000
1
7
14
28
56

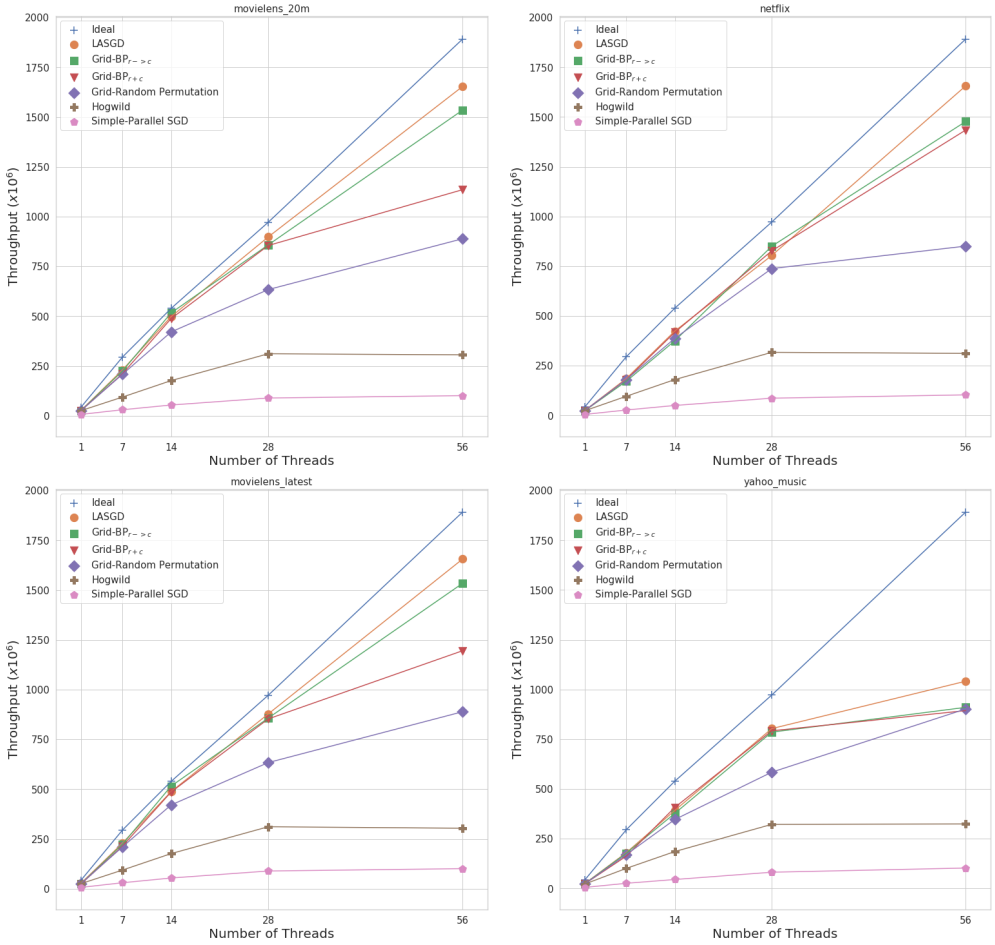


Figure 7.3: Throughput scalability curve with increasing number of threads for individual datasets

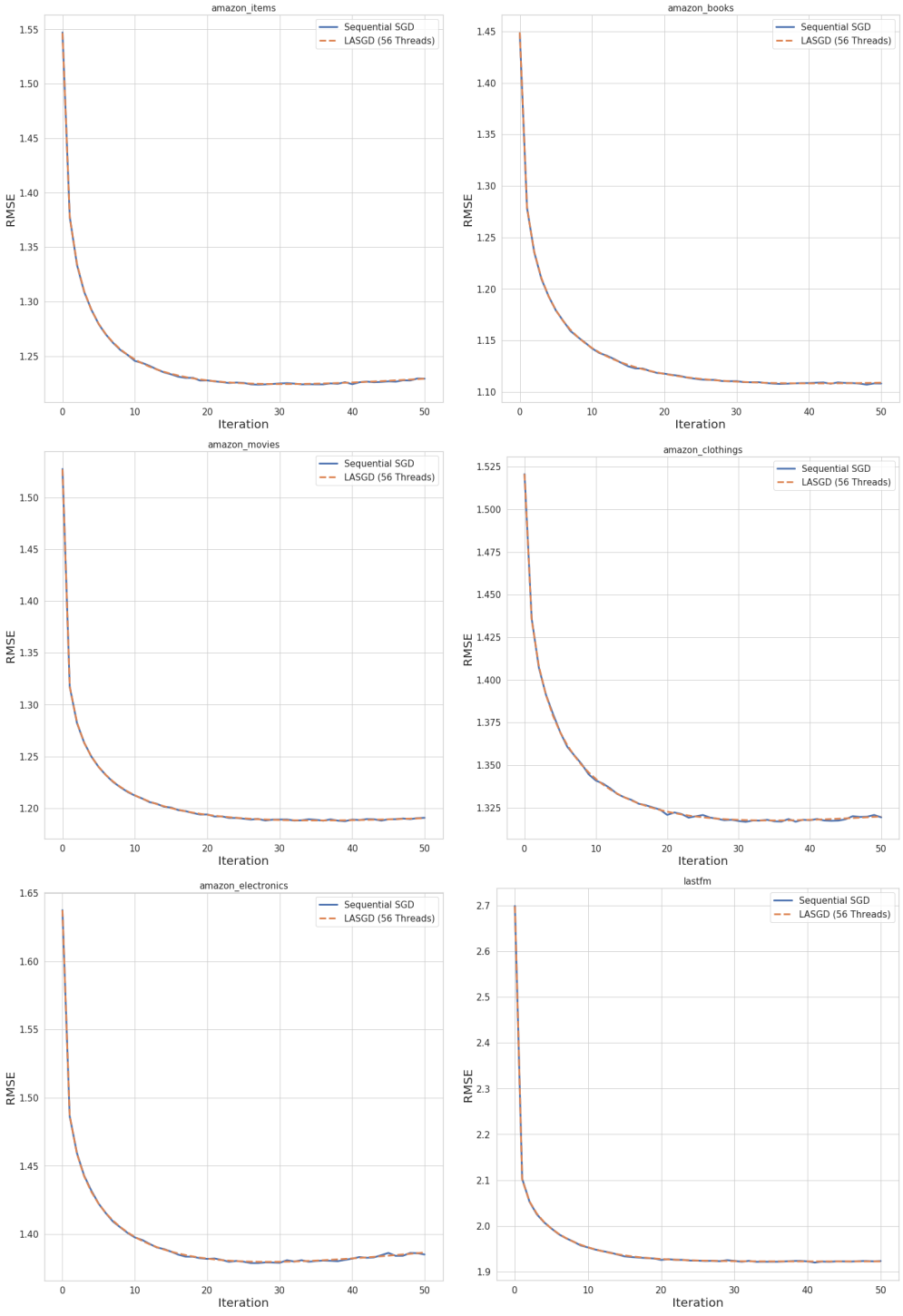
7.1 Convergence Discussion

Many studies [10, 12, 27, 28] show the convergence of their methods with plots that one axis shows the wall clock time and the other axis shows RMSE values. In their approach, the underlying reason for better convergence of a method is not clear. It might be converged faster because it did more SGD updates per second or it found more accurate P- and Q-matrices per update. To clarify the reasons behind better convergence, we show updates per second values and RMSE per iteration values separately. Here and hereafter, convergence rate means the

change of RMSE values over iteration numbers.

Figure 7.4 shows the convergence rate of the sequential SGD algorithm and LASGD algorithm. As shown in the figure, convergence lines are almost completely overlapping. Therefore, LASGD doesn't degrade the convergence rate of SGD according to the plot.

Since LASGD algorithm uses a proposed load balancing method ($\mathrm{BP}_{r \to c}$), the figure indirectly compares the convergence rate of proposed load balancing methods. LASGD doesn't affect the convergence rate of SGD so we can say that proposed load balancing methods also have no negative effect on the convergence rate.


amazon_items
amazon_books
amazon_movies
amazon_clothings
amazon_electronics
lastfm
Sequential SGD
LASGD (56 Threads)
RMSE
Iteration

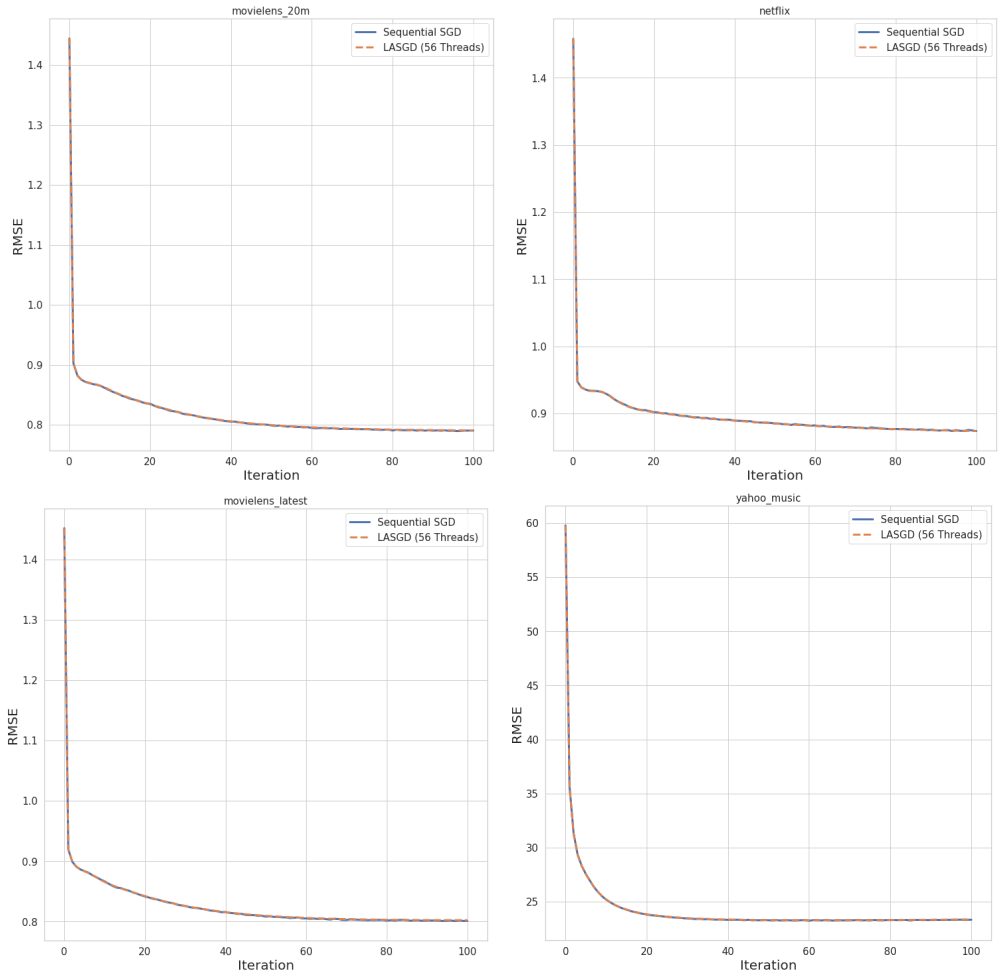


Figure 7.4: Convergence plots of Sequential SGD and LASGD for each dataset

7.2 Discussion about the Factor Size

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 smaller f values are used, the produced model (P and Q matrices) is not strong enough to capture the 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 can be good. Therefore, f should be picked in such a way that it should give good

results on unseen data.

In our experiments, the f value equals to 16. We use the same f value for all datasets. The reasons for our decision on this factor size are as follows:

- Using the same f value allows us to compare the performance results of different datasets.
- The $f = 16$ value gives reasonable RMSE results as shown in table 6.2. We could get better RMSE values by selecting a different factor size value for each dataset. However, we want to use a single f value for all datasets so 16 seems to be a good parameter value that works for every dataset.
- We use 4-byte float type for each element in the factor matrices. So a row of P or Q matrices becomes 64-bytes which is also the size of a cacheline. Setting a row of latent factor matrices equal to a cacheline makes our analysis of LASGD algorithm simpler.

Here, we discuss how the performances of proposed algorithms are affected in terms of the factor size. The proposed load balancing algorithms (BP_{r+c}, $\text{BP}_{r\rightarrow c}$, $\text{BP}_{c\rightarrow r}$) reduces the threads' idle time, their optimization is not directly related to the factor size. We get similar performance speedup results compared to the 2D grid partitioning with the random permutation method when a different f value is used.

The LASGD algorithm improves the latency of reading/writing latent factor rows. So, the value f affects the performance of LASGD. If larger f values are used, it will be difficult for the algorithm to apply the same cache locality optimizations and row buffer locality optimization because the algorithm will place fewer P-matrix and Q-matrix rows to the caches and to the active row. As a result, the larger f values reduce the performance gain of LASGD algorithm whereas the smaller f values increase the performance dually. That is, we can fit more P-matrix and Q-matrix rows to the caches and to the active row. Also, using f values smaller than 16 means that we can fit more than one row into a single

cacheline. This is also a performance increasing factor for LASGD algorithm although the algorithm should be updated accordingly to cover this case.

Chapter 8

Conclusion and Future Works

In the load balancing methods, we tried to balance all blocks as evenly as possible. However, it is enough to load balance blocks that are processed in the same time interval. That is, blocks on different time intervals can have a different number of nonzeros without hurting runtime performance. For the future work, We are planning to enhance the update order selection process of LASGD to gain even more performance since it has more room for improvements due to this relaxed load balance constraint.

We proposed algorithms to address load balancing issues in grid-based parallel SGD methods and proposed LASGD to address the memory underutilization problem. We explained how these ideas can be merged into a single shared memory parallel SGD algorithm that performs more effective than currently used methods.

Appendix A

Row Buffer Locality Micro-benchmark

We will give some background information on DRAM and row buffers before explaining our benchmark setup and results. Modern DRAM architectures are consist of many storage organization layers like DIMMs, ranks, banks and arrays. Each bank contains arrays where data is stored. A row buffer placed in every bank which acts as a data bridge between data cells and the memory bus. When a read command is issued on DRAM, the corresponding DRAM row is fetched from the data cells and stored on the row buffer. This is known as a row activation and the activated row is called open row or open page. After row activation step, the row buffer can send requested word of the row with the help of a column decoder. Even if a single cache line of data is requested, the whole row which can be as large as 8 KB should be fetched into the row buffer. The write operation is similar. That is, updated words are first written into row buffer. Then, they are transferred to data cells.

There are two main policies to manage the row buffer: open-page and closed-page policies. In open page policy, after a page (DRAM row) is fetched into the row buffer, it stays in the buffer until a request to another row is made. If another request to a word in the same row, it can be transmitted to CPU with

a low latency since it is already in the row buffer [29, 30]. This is known as row buffer hit. If the requested word is not in the open row, open row should be written to DRAM cells before requested row is fetched into the row buffer. This is known as row buffer miss. Because of the extra data communication between row buffer and DRAM cells, the latency is longer if row buffer miss occurs. In closed-page policy however, open row is written back to data cells immediately after it is used. Closed page policy can provide better latency in the case of row buffer misses because row buffer have sent the open row before hand so it doesn't need to send open row to DRAM cells while CPU is waiting.

As explained in previous two paragraphs row buffers can be considered as a cache for the recently used row. We benchmark the code given in Appendix B to develop a better understanding on the performance behaviour of our system regarding row buffer hits.

The profiled code is basically consist of two nested loops. In the inner loop, the program reads an index from an array of size N. Then the program reads the data from the data array according to the index. It does a simple calculation and then it writes to the same memory location. By changing the index array, we can adjust the DRAM access pattern of the program. For example if $\langle 1, 2, \cdots N \rangle$ is used as the index array, the program visit data array in linear fashion. Each location of the data array contains a predefined structure called chunk. A chunk is 64 byte structure containing a single integer value. The reason for using a 64 byte structure is that a chunk exactly fits into a cacheline if data array is aligned to 64 bytes. We can negate the effect of L1, L2 and L3 cache hits/misses and focus on the row buffer hits/misses. Outer loop is used to iterate the inner loop several times to get smoother results. The array size N is large enough that the array does not fit into the caches so each data in the array is fetched from the DRAM.

The index array configurations used in the experiment are listed below:

- **Complete Linear :** Processes items from 1 to N.

- **Complete Random :** Index array is shuffled so the code visits the elements of the data array completely random.
- **Shuffled Blocks In-Block Linear $< k >$:** Index array is split into blocks with k elements. All blocks are shuffled. Within each block, items are processed in linear order.
- **Shuffled Blocks In-Block Random $< k >$:** Index array is split into blocks with k elements. All blocks are shuffled. Within each block, items are processed randomly.

Note that the complete linear configuration can be represented by the shuffled blocks in-block linear configuration with N block size. Similarly, the complete random configuration can be represented as shuffled blocks in-block random configuration with 1 block size. We decided to show them separately to emphasize those special cases. Example index arrays for the mentioned configurations are shown in Figure A. In the example, the array size (N) is 12 and the block size (k) is 4.

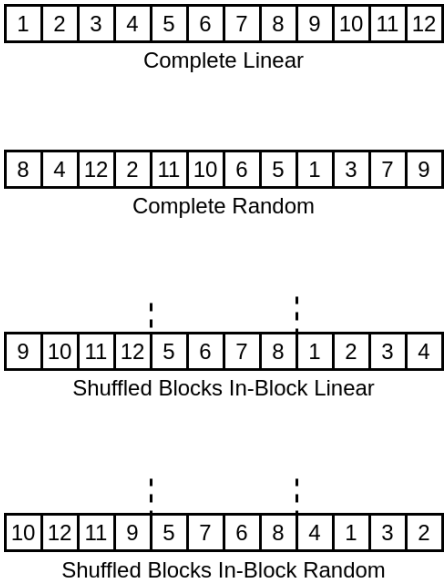


Figure A.1: Example index array of experiment configurations for index array with 12 elements and 4-size blocks. The blocks are divided with dashed lines.

A.1 Benchmark Results

Table A.1: Prefetching Disabled In-Block Linear Benchmark Results

Configuration	Throughput ($\times 10^6$)	Bandwidth (%)	Latency (%)	Latency (cycles)
Complete Random	55.07	58.72	38.99	322.60
In-Block Linear 2	58.32	61.69	36.39	306.20
In-Block Linear 4	60.11	61.74	36.50	288.70
In-Block Linear 8	67.56	70.32	27.97	274.60
In-Block Linear 16	72.15	74.06	24.02	265.50
In-Block Linear 32	88.38	93.21	6.23	243.00
In-Block Linear 64	88.89	93.71	5.96	232.50
In-Block Linear 128	88.75	92.13	6.06	225.50
In-Block Linear 256	88.64	92.93	6.36	232.10
In-Block Linear 512	88.65	93.35	5.56	231.20
In-Block Linear 1024	89.25	93.69	5.51	229.60
Complete Linear	89.14	82.33	5.23	224.70

Table A.1 and A.2 show benchmark results of in-block linear and in-block random configurations respectively with various block sizes. The throughput column shows the number of millions item processed per second. The bandwidth column shows percentage of time CPU was stalled on the main memory with high DRAM utilization. The latency (%) column shows the percentage of time CPU was stalled due to the latency of the main memory (DRAM) and the latency (cycles) column shows the latency of DRAM in average CPU cycles.

As seen on the tables, accessing memory cachelines in linear fashion is significantly faster than random access. Also, clustering items into a set blocks and

Table A.2: Prefetching Disabled In-Block Random Benchmark Results

Configuration	Throughput ($\times 10^6$)	Bandwidth (%)	Latency (%)	Latency (cycles)
Complete Random	55.07	58.72	38.99	322.60
In-Block Random 2	58.12	61.77	36.60	311.10
In-Block Random 4	59.77	61.90	36.99	286.00
In-Block Random 8	67.54	70.43	28.26	278.90
In-Block Random 16	73.24	75.39	22.26	270.20
In-Block Random 32	88.17	92.86	6.47	237.60
In-Block Random 64	88.48	92.69	5.96	233.20
In-Block Random 128	86.37	92.78	5.59	233.90
In-Block Random 256	84.50	93.64	5.12	239.40
In-Block Random 512	83.78	93.90	5.00	233.70
In-Block Random 1024	83.69	93.84	5.22	250.50
Complete Linear	89.14	82.33	5.23	224.70

requesting them block by block is also efficient way of accessing data in DRAM. As the block size is increased the latency of DRAM accesses is decreased and consequently we get better throughput values. However, this performance gain is stopped at block size of 32. In other words, using blocks larger than 32 does not achieve more performance than the configuration using blocks with 32 size. Since 1 block is 64 bytes, a block of size 32 is equivalent to 2 kilobytes. These results are consistent with our row buffer locality explanations. That is, accessing the data on the open row has lower latency. The fact that the performance gain stops at 2 KB implies the row size is 2 KB.

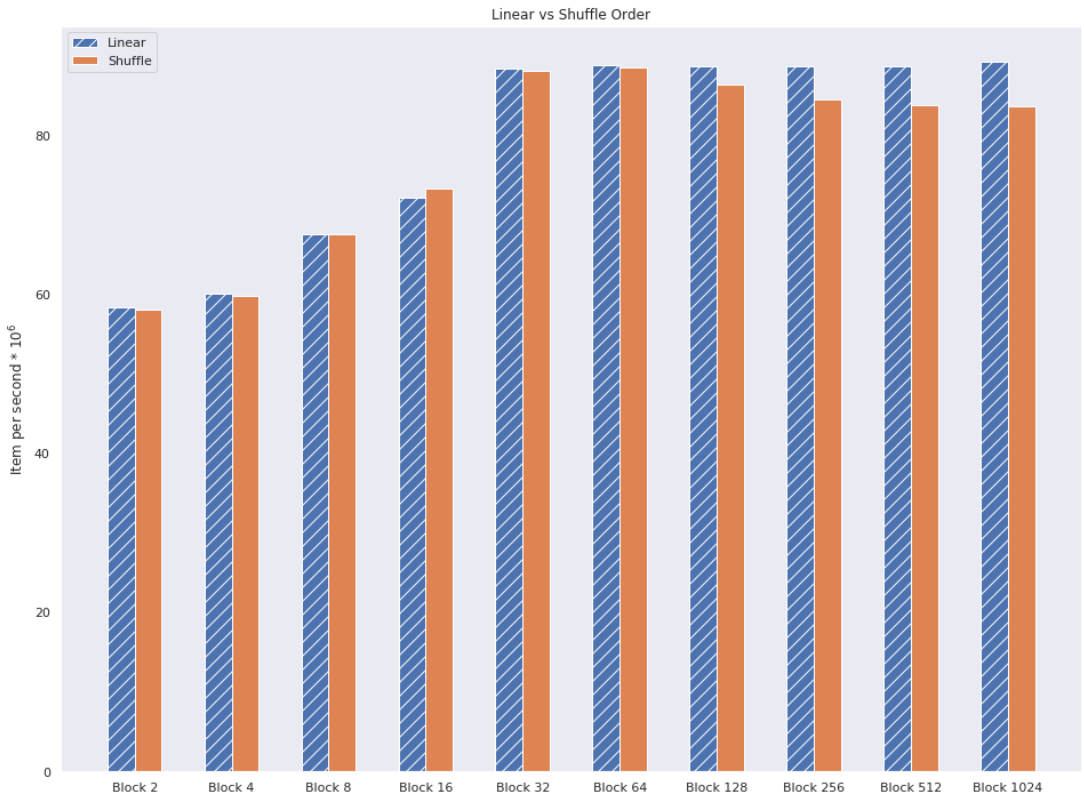


Figure A.2: Comparison between in-block linear order configuration and in-block random order configuration.

As we said earlier, consecutive accesses to 2 KB memory window rather than accessing data in a completely random fashion can utilize memory better. Figure A.1 shows that the statement is true even if we access items in the block in random order. The figure compares throughput of in-block linear order configurations against in-block random order configurations with different block size values. In general, there is no distinguishable difference between two approach. However, the throughput seems to start decreasing slightly after block size 32. This result also supports our inference that row buffer size is 2 KB because in configurations where block size is larger than 32 items (or 2 KB), DRAM starts to access different DRAM rows while the program is processing a single block.

We use the outcome of this experiment in the LASGD algorithm. LASGD

algorithm can change the memory access pattern of SGD. LASGD utilize DRAM better by arranging the rating matrix in such a way that latent factor read/write requests exploits DRAM row buffer locality. The detailed discussion is given in Chapter 5.

Appendix B

Code

All code is hosted on https://github.com/Shathra/parallel-sgd-on-multicore-arch

B.1 Row Buffer Locality Benchmark Code

```
struct chunk
{
    int val;
    int pad[15];
};
```

Main loop :

```
for(int i = 0; i < iteration; i++)
{
        for(int j = 0; j < N; j++)
        {
                int idx = idx_arr[j];
                arr[idx].val *= current;
```

```
                current = arr[idx].val;
        }
}
```

Bibliography

[1] O. Levy and Y. Goldberg, "Neural word embedding as implicit matrix factorization," in *Advances in neural information processing systems*, pp. 2177–2185, 2014.

[2] J. Lu, D. Wu, M. Mao, W. Wang, and G. Zhang, "Recommender system application developments: a survey," *Decision Support Systems*, vol. 74, pp. 12–32, 2015.

[3] H. Kim and H. Park, "Sparse non-negative matrix factorizations via alternating non-negativity-constrained least squares for microarray data analysis," *Bioinformatics*, vol. 23, no. 12, pp. 1495–1502, 2007.

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

[5] R. M. Bell, Y. Koren, and C. Volinsky, "All together now: A perspective on the netflix prize," *Chance*, vol. 23, no. 1, pp. 24–29, 2010.

[6] R. M. Bell and Y. Koren, "Lessons from the netflix prize challenge.," *SiGKDD Explorations*, vol. 9, no. 2, pp. 75–79, 2007.

[7] E. J. Candès and B. Recht, "Exact matrix completion via convex optimization," *Foundations of Computational mathematics*, vol. 9, no. 6, p. 717, 2009.

[8] P. Jain and P. Netrapalli, "Fast exact matrix completion with finite samples," in *Conference on Learning Theory*, pp. 1007–1034, 2015.

[9] J. Langford, A. Smola, and M. Zinkevich, "Slow learners are fast," *arXiv preprint arXiv:0911.0491*, 2009.

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

[11] B. Recht, C. Re, S. Wright, and F. Niu, "Hogwild: A lock-free approach to parallelizing stochastic gradient descent," in *Advances in neural information processing systems*, pp. 693–701, 2011.

[12] Y. Zhuang, W.-S. Chin, Y.-C. Juan, and C.-J. Lin, "A fast parallel sgd for matrix factorization in shared memory systems," in *Proceedings of the 7th ACM conference on Recommender systems*, pp. 249–256, ACM, 2013.

[13] H. Yun, H.-F. Yu, C.-J. Hsieh, S. Vishwanathan, and I. Dhillon, "Nomad: Non-locking, stochastic multi-machine algorithm for asynchronous and decentralized matrix completion," *Proceedings of the VLDB Endowment*, vol. 7, no. 11, pp. 975–986, 2014.

[14] F. Petroni and L. Querzoni, "Gasgd: stochastic gradient descent for distributed asynchronous matrix completion via graph partitioning.," in *Proceedings of the 8th ACM Conference on Recommender systems*, pp. 241–248, ACM, 2014.

[15] H. Lakkaraju, J. McAuley, and 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, and P. Lamere, "The million song dataset," in *Proceedings of the 12th International Conference on Music Information Retrieval (ISMIR 2011)*, 2011.

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

[18] J. Kiefer, J. Wolfowitz, *et al.*, "Stochastic estimation of the maximum of a regression function," *The Annals of Mathematical Statistics*, vol. 23, no. 3, pp. 462–466, 1952.

[19] F. Makari, C. Teflioudi, R. Gemulla, P. Haas, and Y. Sismanis, "Shared-memory and shared-nothing stochastic gradient descent algorithms for matrix completion," *Knowledge and Information Systems*, vol. 42, no. 3, pp. 493–523, 2015.

[20] Y. Yu, D. Wen, Y. Zhang, X. Wang, W. Zhang, and X. Lin, "Efficient matrix factorization on heterogeneous cpu-gpu systems," *arXiv preprint arXiv:2006.15980*, 2020.

[21] E. Horowitz and S. Sahni, *Fundamentals of computer algorithms*. Computer Science Press, 1978.

[22] L. Dagum and R. Menon, "Openmp: An industry-standard api for shared-memory programming," *Computing in Science & Engineering*, no. 1, pp. 46–55, 1998.

[23] J. D. Hunter, "Matplotlib: A 2d graphics environment," *Computing in Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007.

[24] J. Reinders, "Vtune performance analyzer essentials," *Intel Press*, 2005.

[25] A.-L. Barabási and R. Albert, "Emergence of scaling in random networks," *science*, vol. 286, no. 5439, pp. 509–512, 1999.

[26] J. Gaillard and J.-M. Renders, "Time-sensitive collaborative filtering through adaptive matrix completion," in *European Conference on Information Retrieval*, pp. 327–332, Springer, 2015.

[27] H.-F. Yu, C.-J. Hsieh, S. Si, and I. Dhillon, "Scalable coordinate descent approaches to parallel matrix factorization for recommender systems," in *2012 IEEE 12th International Conference on Data Mining*, pp. 765–774, IEEE, 2012.

[28] J. Oh, W.-S. Han, H. Yu, and X. Jiang, "Fast and robust parallel sgd matrix factorization," in *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 865–874, ACM, 2015.

[29] K. Sudan, N. Chatterjee, D. Nellans, M. Awasthi, R. Balasubramonian, and A. Davis, "Micro-pages: increasing dram efficiency with locality-aware data placement," *ACM SIGARCH Computer Architecture News*, vol. 38, no. 1, pp. 219–230, 2010.

[30] H. Yoon, J. Meza, R. Ausavarungnirun, R. A. Harding, and O. Mutlu, "Row buffer locality aware caching policies for hybrid memories," in *2012 IEEE 30th International Conference on Computer Design (ICCD)*, pp. 337–344, IEEE, 2012.