Fast and Robust Parallel SGD Matrix Factorization

Jinoh Oh, Wook-Shin Han,* and
Hwanjo Yu*
Pohang University of Science and Technology
(POSTECH), Pohang, South Korea
{kurin, wshan, hwanjoyu}@postech.ac.kr

Xiaoqian Jiang
University of California at San Diego (UCSD),
California, USA
x1jiang@ucsd.edu

ABSTRACT

Matrix factorization is one of the fundamental techniques for analyzing latent relationship between two entities. Especially, it is used for recommendation for its high accuracy. Efficient parallel SGD matrix factorization algorithms have been developed for large matrices to speed up the convergence of factorization. However, most of them are designed for a shared-memory environment thus fail to factorize a large matrix that is too big to fit in memory, and their performances are also unreliable when the matrix is skewed.

This paper proposes a fast and robust parallel SGD matrix factorization algorithm, called MLGF-MF, which is robust to skewed matrices and runs efficiently on block-storage devices (e.g., SSD disks) as well as shared-memory. MLGF-MF uses Multi-Level Grid File (MLGF) for partitioning the matrix and minimizes the cost for scheduling parallel SGD updates on the partitioned regions by exploiting partial match queries processing. Thereby, MLGF-MF produces reliable results efficiently even on skewed matrices. MLGF-MF is designed with asynchronous I/O permeated in the algorithm such that CPU keeps executing without waiting for I/O to complete. Thereby, MLGF-MF overlaps the CPU and I/O processing, which eventually offsets the I/O cost and maximizes the CPU utility. Recent flash SSD disks support high performance parallel I/O, thus are appropriate for executing the asynchronous I/O.

From our extensive evaluations, MLGF-MF significantly outperforms (or converges faster than) the state-of-the-art algorithms in both shared-memory and block-storage environments. In addition, the outputs of MLGF-MF is significantly more robust to skewed matrices. Our implementation of MLGF-MF is available at http://dm.postech.ac.kr/MLGF-MF as executable files.

Categories and Subject Descriptors

H.4.0 [**Information Systems Applications**]: General—*Matrix factorization*

General Terms

Algorithms, Performance

---

*Corresponding author

Keywords

Matrix factorization; Stochastic gradient descent

1. INTRODUCTION

Matrix factorization is one of the fundamental techniques for analyzing latent relationship between two entities. For example, the latent relationship between documents and keywords can be analyzed for topic detection and document clustering [21]. The latent relationship between users and products can be analyzed for recommendation [1, 11]. Especially, it is popularly used for recommendation for its high accuracy.

Efficient algorithms for matrix factorization have been developed including alternating least square (ALS) [9, 24], coordinate descent [8, 22], and stochastic gradient descent (SGD) [26, 5]. Among those, SGD has gained much attention, as the winners of two extensive competitions, KDDCup 2011 [3] and Netflix comptition [1], used SGD-based matrix-factorization algorithms [1, 25].

To further speed up the convergence of matrix factorization for large matrix, parallel or distributed SGD algorithms have been developed [5, 16, 23, 25, 26]. They first partition the dataset (i.e., a matrix) into several subsets (i.e., sub-matrices) and simultaneously perform SGD updates on each subset in parallel or in a distributed manner. However, (1) most of the algorithms are designed with an assumption that the dataset fully resides in memory, thus they fail to factorize a large matrix that is too big to fit in memory. (2) Also, their performances are unreliable when the matrix is skewed (i.e., the numbers of non-zero entries in the sub-matrices are highly different.) A rating matrix is easily skewed in practice for two reasons: a) old user tend to have more ratings than new users; b) some items often have far more ratings than others. As Figure 1 shows, they partition the matrix into grids. Thus, when the matrix is skewed, a sub-matrix (or a grid cell) with scarce non-zero entries is updated multiple times while a sub-matrix with many non-zero entries is updated once, which could cause a bias on the SGD output.

An SGD algorithm for block-storage devices (e.g., SSD disks) has been developed based on GraphChi [12]. However, the SGD algorithm on GraphChi has two major limitations: First, GraphChi aligns a dataset by the row and column indices of entries to process the dataset via sequential scan. Thus, the order of SGD updates is pre-determined accordingly, which makes the convergence slow. A random order of SGD updating typically produces a better result with a faster convergence [5, 25]. Second, the SGD on GraphChi does not fully utilize the CPU and I/O resources. In a block-storage device, the speed of I/O is far slower than that of memory. Thus, in GraphChi, CPU often winds up waiting for I/O to complete, which ends up wasting CPU resources and slower convergence. Section 3 discusses limitations of existing SGD algorithms in detail.

This paper proposes a fast and robust parallel SGD matrix factorization algorithm, named MLGF-MF, which is robust to skewed matrices and runs efficiently on a block-storage device as well as shared-memory. MLGF-MF minimizes the cost for scheduling parallel SGD updates by developing efficient scheduling operators. MLGF-MF also supports the random order update while being efficient for block-storage devices. As discussed above, the random order of SGD updates is important for fast convergence.

The key idea of MLGF-MF is two folded: First, MLGF-MF uses Multi-Level Grid File (MLGF) instead of typical grid structures for partitioning a matrix. MLGF [20] is a multi-dimensional index, where a region having entries more than the pre-specified capacity is dynamically partitioned. By using MLGF, MLGF-MF contains a similar number of entries in each region, which makes its performance more robust to a skewed matrix. MLGF-MF also efficiently supports partial match query processing, which is important to efficiently execute our new scheduling operators (detailed in Section 4.2).

Second, since the speed of I/O is slower than that of memory, in a disk-based algorithm, it is important to overlap the CPU and I/O processings in order to offset the I/O cost and maximize the CPU utility. MLGF-MF is designed with asynchronous I/O permeated in the algorithm such that CPU keeps executing without waiting for I/O to complete. Thereby, MLGF-MF overlaps the processings of CPU and I/O, which leads to offsetting the I/O cost and improving the CPU utility. Recent flash SSD disks support high performance parallel I/O thus are appropriate for executing the asynchronous I/O.

From our extensive evaluations, MLGF-MF significantly outperforms (or converges faster than) the state-of-the-art algorithms in both shared-memory and block-storage environments. In addition, the outputs of MLGF-MF is significantly more robust for skewed matrices. Our implementation of MLGF-MF is available at http://dm.postech.ac.kr/MLGF-MF as executable files.

The remainder of this paper is organized as follows. In Section 2, we briefly introduce the preliminary for matrix factorization via SGD. We provide the details of the state-of-the art methods, and discuss their limitations in Section 3. In Section 4, we propose MLGF-MF. Specifically, we introduce the MLGF and propose MLGF-MF in Section 4.1. We propose an efficient scheduling operators and discuss the optimal splitting strategy in Section 4.2. In Section 4.3, we introduce efficient I/O model for MLGF-MF. We report the evaluation result in Section 5, and conclude our paper in Section 6.

2. PRELIMINARY: MATRIX FACTORIZATION VIA STOCHASTIC GRADIENT DESCENT(SGD)

Let $R \in \mathbb{R}^{m \times n}$ be a target matrix (e.g., rating matrix), where m is the number of row entities (e.g., users), and n is the number of column entities (e.g, items). Let r_{ij} denote (i, j)-th entry of matrix R which presents the relationship between the corresponding row and column entities (e.g., the rating of the user i on item j). The goal of matrix factorization is to find two latent factor matrices $U \in \mathbb{R}^{k \times m}$ and $V \in \mathbb{R}^{k \times n}$ which accurately approximate the original matrix R ($U^T V \approx R$) with lower dimensionality $k << \min(m, n)$. This problem is formulated as the following optimization problem.

$$\min_{U,V} \mathcal{J}(U,V) = \sum_{(i,j)\in\Theta(R)} \left\{ ||r_{ij} - u_i^T v_j||_2 + \lambda_u||u_i^T||_2 + \lambda_v||v_j||_2 \right\}$$
(1)

where $\Theta(R)$ is a set of index pairs for non-zero entries in the target matrix R; u_i^T is the transpose of the i-th column vector of matrix U; v_j is the j-th column vector of matrix V; $|| \cdot ||_2$ is L_2 norm; λ_u and λ_v are the regularizers which are non-negative scalar values used to avoid the overfitting problem.

Gradient descent [14] is one of the fundamental techniques to solve the optimization problem which finds a local minimum of function $\mathcal{J}(U,V)$ by the gradient of the function. The key idea of gradient descent is iteratively updating the learning parameters (i.e., U and V) by the amount of the product of gradient (i.e., $\nabla \mathcal{J}(U,V)$) and pre-specified learning rate η. More specifically, each parameter is updated as follows.

$$U \leftarrow U - \eta\nabla_U\mathcal{J}(U,V)$$
(2)
$$V \leftarrow V - \eta\nabla_V\mathcal{J}(U,V)$$
(3)

However, calculating $\nabla_U\mathcal{J}(U,V)$ and $\nabla_V\mathcal{J}(U,V)$ is computationally expensive.

For the computational reason, stochastic gradient descent (SGD) [1, 25] is widely used instead of traditional gradient descent [23, 25]. The key idea of SGD is to replace the gradient by an unbiased estimation of the gradient, which is computationally much cheaper [10, 18]. Specifically, SGD approximates $\nabla \mathcal{J}(U,V)$ by the gradient of a randomly selected (i, j) pair (i.e., $\nabla_{u_i,v_j}\mathcal{J}(U,V)$) which yields the following update rules.

$$u_i \leftarrow u_i + \eta\left((r_{ij} - u_i^T v_j)v_j - \lambda_u u_i\right)$$
(4)
$$v_j \leftarrow v_j + \eta\left((r_{ij} - u_i^T v_j)u_i - \lambda_v v_j\right)$$
(5)

Note that, the gradient of randomly sampled (i, j) pair is an unbiased estimator of $\nabla\mathcal{J}(U,V)$ because the expected value of the gradient of randomly sampled (i, j) pair is the same as the true value of $\nabla\mathcal{J}(U,V)$.

3. EXISTING SGD METHODS AND THEIR LIMITATIONS

3.1 Parallel or distributed SGD for matrix factorization

As the size of a dataset is larger, the required time for the convergence via SGD becomes longer. To speed up the convergence, several parallel or distributed algorithms for SGD have been proposed [5, 16, 23, 25, 26]. The key ideas of the algorithms are similar: They partition the dataset $\Theta(R)$ into several subsets and update the parameters (i.e., U and V) for each subset in parallel. The algorithms are distinguished from each others according to the data partitioning strategy and the SGD update scheduling algorithm.

PSGD is the one of early works for parallel SGD computation [26]. PSGD first generates several subsets via uniform random sampling and runs SGD on each subset independently and in parallel. The outputs of SGD update on each subset are averaged at the end. Zinkevich et al. theoretically proved that, even with this simple process, the SGD would eventually converges [26]. However, PSGD exhibits slow convergence in practice [5], and it would incur heavy I/O cost if it is directly extended to block-storage device because it accesses data objects in random order. Similarly, [16] suffers from the same problem.

Gemulla et al. proposed a distributed (or parallel) SGD algorithm, DSGD, based on the concept of interchangeability [5]. Specifically, a pair of data objects such that the order of SGD updates does not affect on the final outcome is referred to as an interchangeable pair. Given two entries r_{ij} and r_{xy}, SGD update on r_{ij} will only
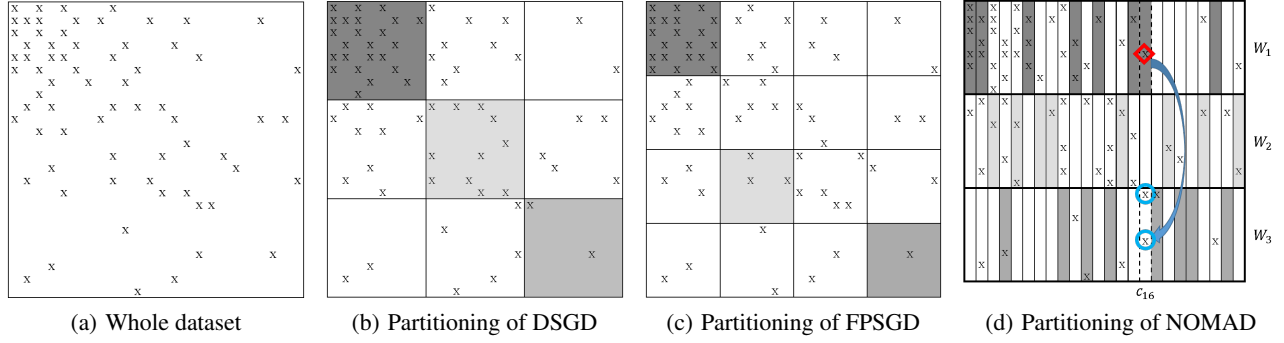
|(a) Whole dataset | (b) Partitioning of DSGD | (c) Partitioning of FPSGD | (d) Partitioning of NOMAD |

Figure 1: Difference in partitioning strategies of DSGD, FPSGD, NOMAD

affect on u_i and v_j vectors, and SGD update on r_{xy} will only affect on u_x and v_y vectors. If $i \neq x$ and $j \neq y$, updating order on r_{ij} and r_{xy} does not make any difference on the result. This concept is naturally extended to interchangeable blocks. Two blocks X and Y are interchangeable, if any points $p_x \in X$ and $p_y \in Y$ are interchangeable.

To find a set of interchangeable blocks, DSGD partitions the dataset into a p-by-p grid, i.e., $R = \{R^{ij} | 1 \leq i \leq p, 1 \leq j \leq p\}$ where R^{ij} is the (i, j)-th grid cell. A maximal set of mutually interchangeable blocks, which is called a *stratum*, can be easily obtained via a random permutation. Suppose that $\sigma(\cdot) = f : \mathbb{N} \to \mathbb{N}$ is a random permutation from $[1, p]$ to $[1, p]$. Then, a set of block $R' = \{R^{i\sigma(i)} | 1 \leq i \leq p\}$ is a maximal set of mutually interchangeable blocks. Based on this retrieved stratum, DSGD runs p SGD updates in parallel. Algorithm 1 describes DSGD.

Algorithm 1: DSGD for matrix factorization

Input : Rating Matrix R, the number of iteration T, and the number of concurrent executions p

Output: Factor matrices U, V

1 Initialize U, V;
2 Block R / U / V into $p \times p$ / $p \times 1$ / $1 \times p$ grid;
3 **for** $t = 1, \ldots, T$ **do**
4   Pick step size η;
5   **for** $s = 1, \ldots, p$ **do**
6   Generate a random permutation σ';
7   Pick p blocks $\{R^{1\sigma'(1)}, \ldots, R^{p\sigma'(p)}\}$ to form a stratum;
8   **for** $b = 1, \ldots, p$ **do in parallel**
9   Run SGD Updating on the block $R^{b\sigma'(b)}$ with step size η;

Zhuang et al. pointed out that DSGD does not fully utilize the CPU resource [25]: Suppose that DSGD is performed in a p-by-p grid setting with p parallel threads. If a thread finish its job earlier, there is no interchangeable block except the just updated block by the thread. Thus it should wait for other threads. To remedy this problem, FPSGD is developed based on two key ideas: 1) using at least $(p + 1)$-by-$(p + 1)$ grid and 2) developing a non-locking scheduling algorithm. By $(p + 1)$-by-$(p + 1)$ grid, there is always an or additional *free blocks*, which is a block not being updated by any other threads but interchangeable to all blocks being updated. Then, the proposed non-locking scheduling algorithm assigns a new block having the smallest number of updates among

free blocks to the early terminated thread. Algorithm 2 describes FPSGD.

Algorithm 2: FPSGD for matrix factorization

Input : Rating Matrix R, the number of update T, the number of concurrent executions p, the size of grid cell $p' > p$

Output: Factor matrices U, V

1 Initialize U, V;
2 Grid R into a set B with at least $p' \times p'$ blocks;
3 Sort each block by user (or item) indices;
4 $totNumUpdate := 0$;
5 **for** $s = 1, \ldots, p$ **do in parallel**
6   **while** $totNumUpdate < T$ **do**
7          $b_s = $ getJob();
8   Run SGD update on block b_s;
9   putJob(b_s);
10          $totNumUpdate + +$;

11 **Procedure** getJob()
12      $b_x = $ NULL ;
13   **forall the** b *in* B **do**
14   **if** b *is non-free* **then**
15   continue ;
16   **else if** $b.numUpdate \leq b_x.numUpdate$ **then**
17              $b_x = b$;
18   **return** b_x;

19 **Procedure** putJob(b)
20   b.numUpdate++;
21   **return**;

Limitation of DSGD and FPSGD: DSGD and FPSGD produce unreliable results when the matrix is skewed. Suppose that we have a dataset of Figure 1(a) and the number of concurrent executions is three. Then, DSGD and FPSGD will partition the dataset as Figure 1(b) and 1(c) respectively. In these figures, there is a clear difference on the number of non-zero entries in each block. In Figure 1(b), the darkest block has 24 non-zero entries while the second darkest block has only two non-zero entries. Similarly, in Figure 1(c), the number of entries in the darkest block is 20 while the second darkest block has only one entry. Due to this difference, DSGD could significantly waste the CPU resource. In FPSGD, the number of SGD updates for each block varies, which could make a bias and eventually make the convergence slow.

Second, the grid structure becomes inefficient when it is applied to block-storage device (or disk) [15, 20]. It has significant space overhead for storing the directories, which becomes significantly larger as the data size grows [20]. It also has non-negligible cost for maintaining the directories if the data is updated incrementally, because when there is a overflowing page, a split of a region affects other entries in the directory.

Yun et al. also pointed out that DSGD suffers from two problems [23]: 1) the waste of CPU and network resources due to bulk synchronization, which is the synchronization after every iteration, and 2) the curse of the last reducer, as all machines have to wait for the slowest machine to finish. NOMAD, a recently proposed parallel (or distributed) SGD algorithm [23], adopts asynchronous communication and computation to avoid bulk synchronization, and decentralized lock-free scheduling. Specifically, NOMAD is executed as follows.

1. The row indices are partitioned and assigned to the workers in advance, and original matrix R is also partitioned and assigned accordingly. The *ownerships of update* of row vectors are assigned to the workers according to the partition, and are never changed during the execution of the algorithm.

2. *Ownerships of update* for columns are initially distributed to the workers and exchanged among workers later.

3. During SGD updating, each worker updates the parameters (i.e., u_i and v_j) only for the rows and columns that the worker has the *ownership of update*.

4. After the pre-determined number of updates, workers hand over their *ownerships of update* for columns to each other via asynchronous message passing.

Figure 1(d) illustrates how NOMAD works. In this example, there are three workers (W_1, W_2, W_3). The matrix is row-wise partitioned into three rows, and the *ownership of update* for each row is assigned to each worker. The *ownerships of update* for columns (i.e., shaded slices) are distributed to the workers non-consecutively. Each worker performs SGD updating on the entries of which each worker has the *ownerships of update* for both row and column indices. For example, let W_1 initially has the *ownership of update* for column c_{16}. W_1 performs SGD update for the red diamond data object because it has the *ownerships of update* for both row and column indices. After a few updates, suppose W_1 hands over the *ownership of update* for column c_{16} to W_3. Then, W_1 does not perform SGD update on column c_{16} anymore. Instead, W_3 starts performing SGD update on the blue circled data objects.

Limitation of NOMAD: NOMAD also suffers from similar limitations. First, NOMAD is mainly designed for distributed environment and not I/O-efficient for block-storage device. As shown in the figure, the *ownerships of update* for columns are allocated non-consecutively to each worker. However, access on data entries having non-consecutive column indices incurs more I/O cost. In addition, NOMAD cannot be naturally extended to block-storage environment because the data storage scheme for each worker is not clearly designed.

NOMAD produces unreliable results for a skewed matrix. As shown in Figure 1(d), there is a clear difference on the number of entries assigned to the workers in NOMAD. For example, W_1 has 35, while W_2 and W_3 have 25 and 9 non-zero entries. Accordingly, the number of currently updated entries also varies; W_1 updates 16 entries while W_2 and W_3 update only 8 and 3 entries. Due to this

difference, the entries in W_1 are less updated than those in W_3, which result in a bias in the final output. As discussed in [23], this could be resolved via random permutation, but the random permutation is not applicable for block-storage device since it produces too much I/O cost, and such time-consuming batch process is seldom used in real-applications.

3.2 Disk-based SGD method and its limitation

To the best of our knowledge, the SGD implementation in GraphChi [12] is the only implementation of parallel SGD for block-storage device. In this implementation, SGD updating is parallelized via parallel sliding windows (PSW), which consists of three stages: it 1) loads a subgraph from disk, 2) updates the factor matrices corresponding to the subgraph, and 3) writes the updated values to disk. To support PSW framework, the matrix R is row-wise partitioned into several execution *intervals*. Then, every entry corresponding to the *interval* is stored in a shard file, and the order of entries in a shard file is sorted by the column indices.

The SGD implementation on GraphChi has two major limitations: First, in GraphChi, the data accessing order is pre-determined and not changed over, which makes it converge slowly. A random order of SGD updating is important for fast convergence [5, 25], thus most other SGD algorithms (e.g., DSGD, FPSGD, NOMAD) are designed with random order of SGD updates. Second, GraphChi does not fully utilize CPU and I/O resources [7]. The I/O speed of block-storage device is far slower than that of memory. Thus, in a disk-based algorithm, it is important to overlap the CPU and I/O processings in order to offset the I/O cost and maximize the CPU utility. However, GraphChi does not overlap CPU and I/O processings, thus the CPU resources are wasted during I/O processing.

4. MULTI-LEVEL GRID FILE BASED MATRIX FACTORIZATION (MLGF-MF)

In this section, we detail MLGF-MF, a novel parallel SGD algorithm for matrix factorization which is efficient for block-storage environment as well as for shared-memory and robust to a skewed matrix. Specifically, we 1) introduce a data structure which is more robust to a skewed matrix and provide a formal description of algorithm in Section 4.1, 2) develop efficient operators for scheduling parallel SGD update based on partial match query and provide an optimal region splitting strategy to support an efficient partial match query processing in Section 4.2, and 3) provide an efficient I/O model for SGD-based matrix factorization via asynchronous I/O in Section 4.3.

4.1 Multi-Level Grid File (MLGF)

Multi-Level Grid File (MLGF) is a multi-dimensional indexing [20], which is distinguished from a grid file in that, 1) hash values are used to represent regions, and 2) a region is dynamically partitioned via a hash function (e.g., msb hash function) if it has more data entries than a pre-specified capacity.

Figure 2 illustrates how MLGF is built. Assume that the maximum capacity of a region is four, and there is only one region for whole space at initial t_0, and the region addresses the data block B_0. Accordingly, D_0 has only one entry having "-" hash value, which matches any value. At time t_1 (Figure 2(a)), a data object is inserted, and the region exceeds its maximum capacity. The region is partitioned, and data objects are distributed to each partitioned region accordingly. For simplicity, we assume the region is vertically partitioned. After partitioning, there are two blocks B_0, B_1, each of which has three and two data objects, respectively. Directory D_0

(a) at time t_1	(b) at time t_2	(c) at time t_3	(d) at time t_4
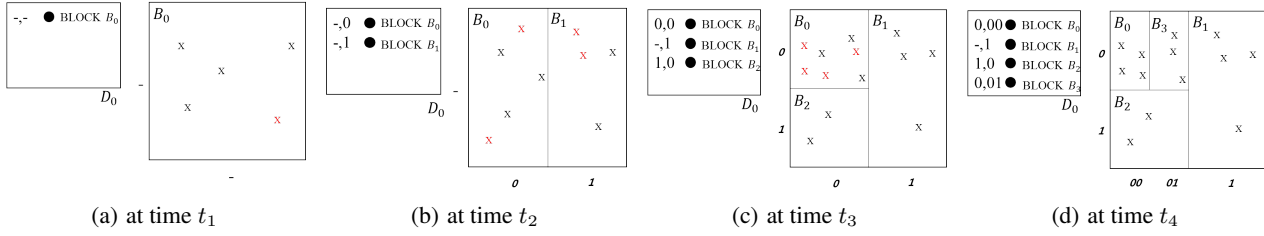
Figure 2: Example of MLGF partitioning. D_0 **is a directory of MLGF;** B_i **is a data block; Red "x" is an updated data object; Italic numbers around the block are hash values; Capacity of a data block is four.**

is also changed accordingly, and has two entries $(-, 0)$ and $(-, 1)$ which address block B_0 and B_1 respectively. At time t_2 (Figure 2(b)), four data objects are inserted. While B_1 does not exceed the capacity thus it is not partitioned, B_0 is partitioned again because it exceeds the maximum capacity. Suppose, in this time, block B_0 is horizontally partitioned. Then, directory D_0 has three entries, $(0, 0)$, $(-, 1)$, and $(1, 0)$, which address block B_0, B_1, and B_2, respectively. Similar tasks are repeated for each update, and the final partitions and the directory will be as shown in Figure 2(d).

Note that, in MLGF, the directory is organized hierarchically, and the entries of a directory can address other directories. Figure 3 shows an example of MLGF directory and the corresponding block partitioning for the data set in Figure 1(a). In this figure, the entries of D_1 in Figure 3(a) address other directories hierarchically. For example, the hash value of the first entry in D_1 is $(00, 00)$, and this entry addresses another directory D_2. In D_2, most entries address data blocks, while the sixth entry addresses another directory.

We choose MLGF as our underlying structure for matrix factorization for following reasons: 1) In MLGF, a region is dynamically partitioned based on the number of entries and thus each region ends up having similar number of entries. Thus, parallel executions of matrix factorization become more robust to a skewed dataset. 2) MLGF efficiently supports block-storage devices [20]. Specifically, in MLGF, a region partitioning does not affect on other entries in the directory, and it supports an efficient partial match query processing, which is an important operation for scheduling algorithm. 3) Other alternatives such as R-tree [6] and Kd-tree [2] are not suitable for our problem because R-tree is designed for spatial objects having sizes thus its operations are more expensive than MLGF, and Kd-tree does not account for block-storage devices [4].

Our parallel matrix factorization algorithm, MLGF-MF, is performed based on the partitioned regions of MLGF. Specifically, our scheduling algorithm outputs an MLGF region (Algorithm 4), and once an MLGF region is assigned to a thread, the thread reads all data objects from the region, and updates two factor matrices U and V accordingly. Algorithm 3 formally describes MLGF-MF.

4.2 Scheduling Operators for MLGF-MF

In this subsection, we introduce efficient scheduling operators for MLGF-MF. The main task of a scheduler is assigning a new interchangeable block to a thread. To support this, a scheduler should be aware of which blocks are interchangeable and non-interchangeable against the blocks currently being updated.

In grid structure, it is easy to find which blocks are interchangeable and non-interchangeable, as the sizes of the blocks are the same. An interchangeable pair refers to a pair of entries such that the order of SGD updates does not affect on the final outcome [5]. In grid structure, an interchangeable pair can be interpreted as a pair having different row and column indices, thus a set of non-
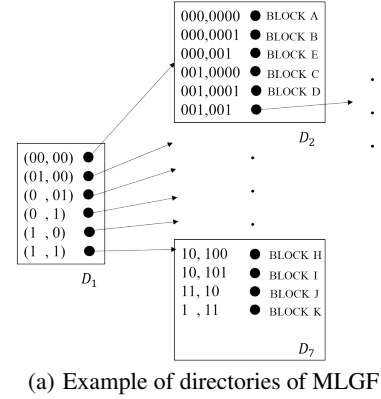
(a) Example of directories of MLGF
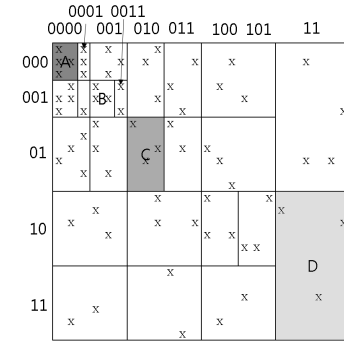
(b) Example of corresponding block partitioning

Figure 3: Example of an index and the corresponding block partitioning of MLGF

interchangeable blocks for a certain block R^{ij} can be retrieved by $\{R^{i'j'} | i' = i \text{ or } j' = j\}$.
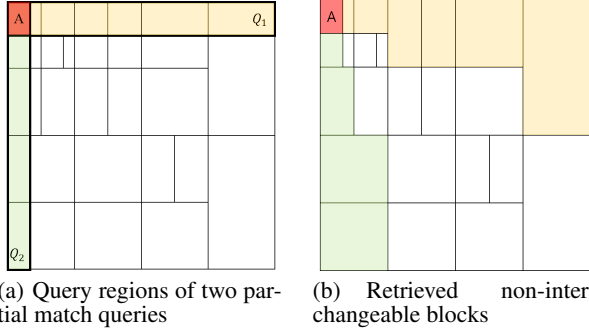
However, in MLGF, identifying non-interchangeable blocks seems not intuitive, as the size of each region is different from each other. A straightforward way to retrieve all the non-interchangeable blocks is traversing the whole MLGF index and testing all regions, which is time-consuming. We exploit *partial match query processing*, which is extensively studied in database community [17, 19, 20], to efficiently retrieve non-interchangeable blocks. A partial match query in MLGF is processed as follows. Given a query region, we starts examining from entries of the root directory of MLGF, to see whether they intersect with the query region. If an entry intersects with the query region, we recursively examine the directory addressed by the entry. However, if an entry is not intersecting with the query region, we do not further examine the directory addressed by the entry. Due to the compact directory structure, par-

Algorithm 3: MLGF-MF

Input : Rating Matrix R, the number of update T, the number of concurrent execution p

Output: Factor matrices U, V

1 Initialize U, V;
2 Construct MLGF index I_{MLGF} based on matrix R;
3 Initialize leafList via traversal on I_{MLGF};
4 $totNumUpdate := 0$;
5 **for** $s = 1, \ldots, p$ **do in parallel**
6   **while** $totNumUpdate < T$ **do**
7   b_s, blockList$_s$ = getJob();
8   Run SGD update on block b_s;
9   putJob(b_s, blockList$_s$);
10       $totNumUpdate + +$;

(a) Query regions of two partial match queries

(b) Retrieved non-interchangeable blocks

Figure 4: Example of retrieving non-interchangeable blocks via partial match queries in MLGF

tial match query is more efficient processed in MLGF than in grid structure [20].

Non-interchangeable blocks are retrieved by processing partial match queries. For example, assume that SGD updates are executed on region A in Figure 4, which is specified by hash values (h_x, h_y). Then, two partial match queries are generated on regions $Q_1 = (h_x, -)$ and $Q_2 = (-, h_y)$. By processing Q_1 and Q_2, we retrieve all regions intersecting with Q_1 or Q_2 which are actually regions sharing either the row or column indices with block A (Figure 4(b)).

In practice, non-interchangeable blocks for a certain block b are repeatedly retrieved for each iteration. Thus, we retrieve non-interchangeable blocks once in the beginning and use it for the following iterations. Algorithm 4 describes our scheduling operators, i.e., getJob() and putJob().

Region splitting policy: The cost of partial match query processing is proportional to the number of regions intersecting with the query region. If the number of intersecting region increases, we have to examine more directories. Lee et al. [13] proposed the optimal splitting policy for a given query set on a two-dimensional space as follows.

THEOREM 1. *[13] Suppose that l query regions of varying sizes $q_i(x) \times q_i(y)$ $(i = 1, \ldots, l)$ are given in a two-dimensional domain space partitioned into page regions of varying sizes, where $q_i(x)$ is the length of a query region q_i along x-axis, and the record density of each query region is d_i. Then, the optimal interval ratio of page regions that minimize the total number of the page re-*

Algorithm 4: Scheduling operators of MLGF-MF

1 **Procedure** getJob()
2   b_x = NULL ;
3   **forall the** b *in* leafList **do**
4   **if** $b.lock > 0$ **then**
5   continue ;
6   **else if** $b.numUpdate < b_x.numUpdate$ **then**
7           $b_x = b$;
8   blockList$_x$ = getNonInterchangeable(b_x);
9   **forall the** b *in* blockList$_x$ **do**
10       $b.lock = b.lock + 1$;
11   **return** b_x, blockList$_x$;
12 **Procedure** putJob(b, blockList)
13    $b.numUpdate++$;
14   **forall the** b *in* blockList **do**
15       $b.lock = b.lock - 1$;
16   **return**;
17 **Procedure** getNonInterchangeable(b)
18   q_1, q_2 = generateTwoPartialQuery(b);
19    $L_1 = I_{MLGF}.partialQuery(q_1)$;
20    $L_2 = I_{MLGF}.partialQuery(q_2)$;
21   **return** $L_1 \cup L_2$;

gions intersecting with the query regions is given by $p(x) : p(y) = \sum_{i=1}^{l} q_i(x)\sqrt{d_i} : \sum_{i=1}^{l} q_i(y)\sqrt{d_i}$.

We apply the optimal region splitting policy to MLGF-MF. In MLGF-MF, we generate two partial queries for each region r_i with sizes $r_i(x) \times n$ and $m \times r_i(y)$, where m and n are the size of rows and columns of R. Then, the optimal interval ratio of page region is

$$p(x) : p(y) = \sum_{i=1}^{l} (r_i(x) + m)\sqrt{d_i} : \sum_{i=i}^{l} (r_i(y) + n)\sqrt{d_i}$$

Suppose each region r_i already follows the optimal ratio (i.e. $r_i(x) = \gamma_i \cdot p(x)$ and $r_i(y) = \gamma_i \cdot p(y)$, where γ_i is the scaling factor for r_i). Then,

$$
\begin{aligned}
& p(x) : p(y) \\
= \ & \sum_{i=1}^{l}(\gamma_i \cdot p(x) + m)\sqrt{d_i} : \sum_{i=i}^{l}(\gamma_i \cdot p(y) + n)\sqrt{d_i} \\
= \ & (p(x)Z + mD) : (p(y)Z + nD)
\end{aligned}
$$

where $Z = \sum_{i=1}^{l} \gamma_i \sqrt{d_i}$ and $D = \sum_{i=1}^{l} \sqrt{d_i}$. From the above ratio, we can obtain

$$
\begin{aligned}
p(x) \cdot (p(y)Z + nD) &= p(y) \cdot (p(x)Z + mD) \\
p(x)p(y)Z + p(x)nD &= p(x)p(y)Z + p(y)mD \\
p(x)nD &= p(y)mD \\
p(x) : p(y) &= m : n
\end{aligned}
$$

Therefore, the splitting direction for overflowing regions follows the ratio of $m : n$, which is the ratio of the original matrix R.

4.3 Efficient I/O Model for MLGF-MF

As the I/O speed of block-storage device is far slower than that of memory, it is important to overlap the CPU and I/O processings in a disk-based algorithm, in order to offset the I/O cost and maximize

Algorithm 5: MLGF-MF with asynchronous I/O

Input : Rating Matrix R, the number of update T, the number of concurrent execution p

Output: Factor matrices U, V

1 Initialize U, V;

2 Construct MLGF index I_{MLGF} based on matrix R;

3 Initialize leafList via traversal on I_{MLGF}.
   $totNumUpdate := 0$;

4 **for** $s = 1, \ldots, p$ **do in parallel**

5 | b_s, blockList$_s$ = getJob();

6 | IssueReadAndCallback (b_s, blockList, callbackSGD);

7 | $totNumUpdate++$;

8 Wait until the completion of all callback threads ;

9 **return**;

10 **Procedure** callbackSGD (*b*, blockList)

11 | // This function is called after the completion of read I/O on the block b ;

12 | **if** $totNumUpdate < T$ **then**

13 | | b_f, blockList$_f$ = getJob();

14 | | IssueReadAndCallback (b_f, blockList$_f$, callbackSGD);

15 | | $totNumUpdate++$;

16 | Run SGD update on block b ;

17 | putJob (*b*, blockList) ;

18 | **return**;

---

the CPU utility. MLGF-MF is designed with asynchronous I/O permeated in the algorithm such that CPU keeps executing without waiting for I/O to complete. Specifically, for each update on a block, we first find a future block b_f that will be updated just after the SGD update on the current block. We issue an asynchronous I/O request for future block b_f. After that, we return to the current block, and continue to perform SGD update on the current block. After completing the SGD updates, putJob() is called for postprocessing. When the I/O request for b_f is completed, another callback thread starts to processes SGD updates, and so on. In this way, we overlap the I/O processing and CPU processing, which offsets the I/O cost and maximizes the CPU utility. Algorithm 5 describes MLGF-MF with asynchronous I/O.

In our implementation, we also use a thread pool, and the number of concurrent callback threads is strictly restricted not to exceed the maximum number of concurrent threads p.

5. EXPERIMENT

This section extensively evaluates the empirical performance of MLGF-MF. Our experiments are designed to verify that (1) MLGF-MF is efficient for both shared-memory and block-storage environments, and (2) MLGF-MF is robust to skewed datasets. In addition, we investigate the bottleneck of MLGF-MF for block-storage environment, the scalability of MLGF-MF, and the performance of MLGF-MF on varying sizes of pages.

5.1 Experimental setting

Dataset description: Throughout our experiments, we evaluate MLGF-MF and competitors for three benchmark datasets: 1) NetFlix [1], 2) Yahoo! Music [3], 3) HugeWiki [23] (see Table 1 for more details). We follow the official partitioning for training and testing

sets for NetFlix and Yahoo! Music. For HugeWiki, there is no known partitioning for training and testing sets, so we randomly extract 1% of dataset for testing, and use the others for training. In addition, to evaluate the robustness of the methods, we generated a skewed version of NetFlix by reordering user IDs and item IDs in an decreasing order according to the number of ratings, which makes the matrix dense in the upper-left side and scarce in the lower-right side. This dataset is denoted as NetFlix$_{freq}$.

We follow the SGD parameter setting of Zhuang et al. [25] for NetFlix and Yahoo! Music. For HugeWiki, we follow the setting of Yun et al. [23] except we decrease the latent space size $k = 40$, which further improves the SGD performance. Our settings for SGD update parameters are reported in Table 1. For initializing the latent matrices, we set each entry of U and V via uniform random sampling in the range $[0, 0.1]$. We set the maximum number of concurrent threads p to be 8. And, by default, FPSGD uses $2p$-by-$2p$ grid.

Dataset name	Netflix	Yahoo! Music	Hugewiki
m	480,189	1,000,990	50,082,603
n	17,770	624,961	39,780
# Training	99,072,112	252,800,275	3,411,259,583
# Test	1,408,395	4,003,960	34,458,060
k	40	100	40
λ_u	0.05	1	0.01
λ_v	0.05	1	0.01
η	0.002	0.0001	0.001

Table 1: Dataset details and SGD update parameter settings

Competitors: We evaluate the performance of MLGF-MF for both shared-memory and block-storage device environments. We use the state-of-the-art parallel SGD matrix factorization algorithms as our competitors. Specifically, we use FPSGD [25] and NO-MAD [23] for shared-memory environment, and GraphChi [12] for block-storage device environment. We use publicly available source codes from authors of FPSGD, NOMAD, and GraphChi.

As stated in [23], the convergence speed of SGD methods varies according to the choice of stepsize, and each proposed method uses a different stepsize. FPSGD uses a constant stepsize while NO-MAD dynamically adjusts the stepsize, η_t, as follows.

$$\eta_t = \frac{\eta_0}{1 + \beta \cdot t^{1.5}} \qquad (6)$$

where t is the number of SGD updates that were performed on a particular user-item pair (i, j); η_0 is an initial stepsize; and β is a decaying factor. In GraphChi, stepsize is adjusted by $\eta_t = \beta \cdot \eta_{t-1}$. For fair evaluation, we use a constant stepsize for all the four methods, i.e., the three competitors and MLGF-MF. The use of dynamical stepsize such as that of Equation 6 may improve the convergence speed of the methods, but its effects are common for all methods.

For fair evaluation, we also disable SSE instruction set, which reduces the CPU time by concurrently performing floating point multiplication and addition; NOMAD and GraphChi do not support the technique. In addition, all experiments are performed using single precision arithmetic because FPSGD uses single precision arithmetic.

Experimental environment: All experiments are conducted in a machine equipped by Intel Core i7 4700@3.4GHz (8 cores), 24GB RAM, and Ubuntu 14.04 trusty. By default, the page size is set to

| (a) NetFlix | (b) Yahoo! Music | (c) NetFlix$_{freq}$ |
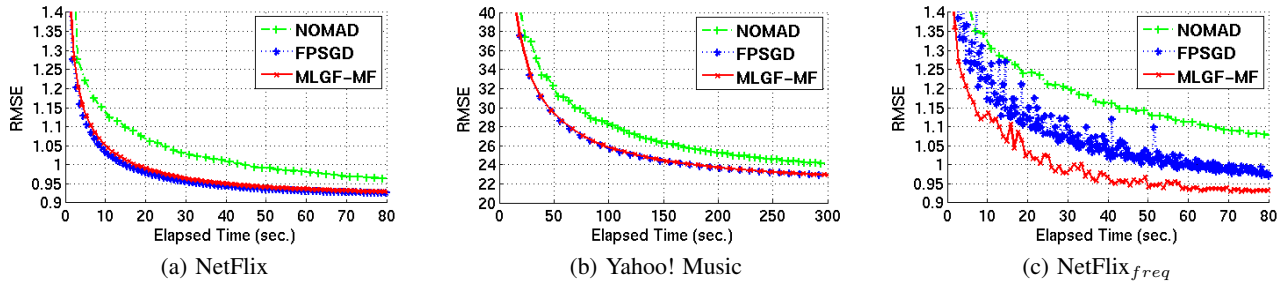
Figure 5: Comparison on convergence speed in shared-memory device for varying datasets: NetFlix, Yahoo! Music, and NetFlix$_{freq}$. X-axis: Elapsed time (sec.); Y-axis: RMSE.

| (a) NetFlix, RevuAhn | (b) Yahoo! Music, RevuAhn | (c) HugeWiki, RevuAhn |
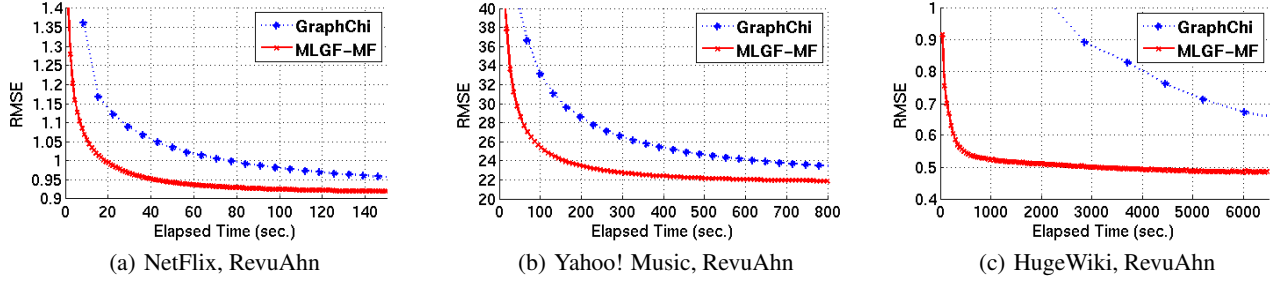
Figure 6: Comparison on convergence speed in block-storage device for varying datasets: NetFlix, Yahoo! Music, and HugeWiki. X-axis: Elapsed time (sec.); Y-axis: RMSE.

Name	RevuAhn	S850PRO
Volume	2 TB	128 GB
Interface	PCI-E	SATA3
Sequential Read (MB/s)	1442	511.8
512k Random Read (MB/s)	733.3	365.9
4k Random Read (MB/s)	22.86	37.11
4k Random Read with Queue depth 32 (MB/s)	342.7	401.3

Table 2: I/O performance of block-storage devices (SSD disks)

be 1 MByte, but we also report the performance of MLGF-MF for varying sizes of pages from 512 KByte to 4 MByte in Section 5.2.5.

We use two block-storage devices (SSD disks), i.e., RevuAhn Drive X (denoted as RevuAhn) and Samsung SSD 850 PRO 128GB (denoted as S850PRO). I/O performances of two devices are compared in Table 2.

5.2 Evaluation results

5.2.1 Comparison on convergence speed for shared-memory environment

We evaluate the performance of MLGF-MF in shared-memory environment by comparing its convergence speed with that of FPSGD and NOMAD. Figure 5 shows the RMSE change of each method on three datasets, NetFlix, Yahoo! Music, and NetFlix$_{freq}$. HugeWiki dataset is excluded here due to the lack of memory. The lower RMSE for the same execution time is the better performance or faster convergence.

For NetFlix and Yahoo! Music, NOMAD shows the slowest convergence, and MLGF-MF and FPSGD show comparable performance. NOMAD suffers from the memory discontinuity caused by non-consecutive access on column indices. For NetFlix$_{freq}$ (a skewed dataset), as we expected, the convergence speed of FPSGD and NOMAD becomes noticeably slow compared to that of MLGF-
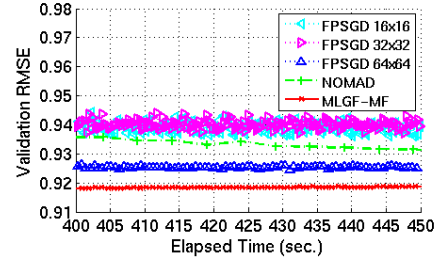
Figure 7: RMSE of methods for NetFlix$_{freq}$

MF. For example, RMSE of NOMAD for NetFlix$_{freq}$ around 20 sec. is 1.25 while it slightly overs 1.05 for NetFlix dataset. Similarly, RMSE of FPSGD around 20 sec. is around 1.1 - 1.15 while it is under 1 in NetFlix. On the other hand, RMSE of MLGF-MF around 20 sec. is close to 1 for NetFlix$_{freq}$, which is similar to that for NetFlix.

RMSE of FPSGD and NOMAD at convergence is higher than that of MLGF-MF for a skewed matrix. Figure 7 shows that RMSEs of FPSGD and NOMAD do not reach to 0.919 which is the state-of-the-art RMSE for NetFlix dataset [25].

We also tried FPSGD with various grid settings. For example, even with 64-by-64 grid having 4,096 cells, FPSGD does not reach to 0.919 of RMSE, while MLGF-MF having around 1,400 regions reaches to 0.919 within 150 seconds. From this evaluation, we observe that increasing the granularity of grid does not improve the convergence quality for a skewed dataset.

These results show that MLGF-MF achieves better convergence quality at faster speed than other methods when the matrix is skewed.
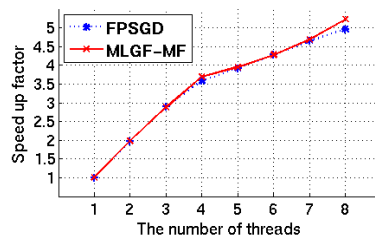
5.2.2 Comparison on convergence speed for block-storage environment

We evaluate the performance of MLGF-MF for block-storage device. Figure 6 shows the RMSE change of two methods, MLGF-
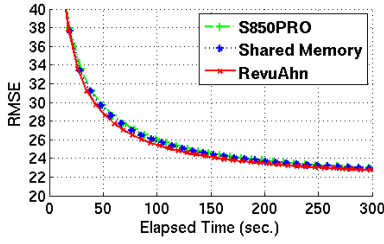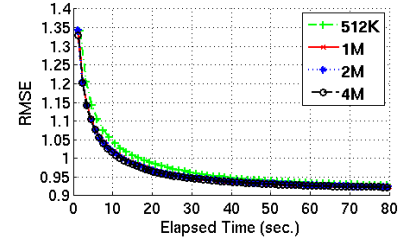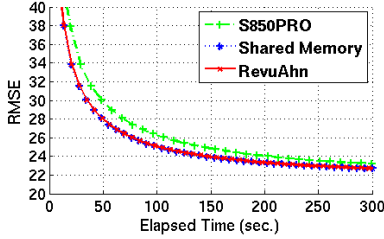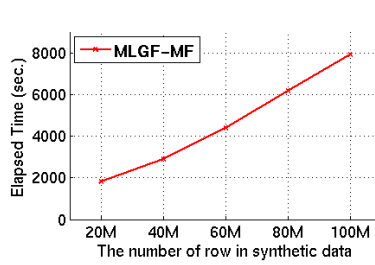
(a) Without SSE instruction

(b) With SSE instruction

Figure 8: Comparison on the convergence speed of MLGF-MF for Yahoo! Music dataset on shared-memory, RevuAhn, S850PRO.

(a) Varying thread size, NetFlix, shared-memory

(b) Varying data size, Synthetic dataset, RevuAhn

Figure 9: Investigation on scalability of MLGF-MF.

(a) Convergence speed, NetFlix, RevuAhn
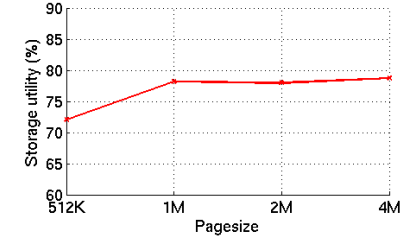
(b) Storage utility, NetFlix

Figure 10: Performance sensitivity of MLGF-MF for varying page size.

MF and GraphChi, for various datasets on RevuAhn. For all datasets, MLGF-MF significantly outperforms GraphChi. For example, for NetFlix dataset, to achieve 0.95 of RMSE, MLGF-MF takes around 40 seconds while GraphChi requires more than 140 seconds to achieve the same level of RMSE. For larger datasets, Yahoo! Music and HugeWiki, the results are similar. MLGF-MF shows much faster convergence speed than GraphChi.

We also evaluate them for another slower block-storage device S850PRO. For all the datasets, the evaluation results are similar to those on RevuAhn, and thus we report the results in our website due to the page limitation.

These results show that MLGF-MF demonstrates much faster convergence speed than the state-of-the-art SGD algorithm for block-storage environment.

5.2.3 Performance bottleneck of MLGF-MF

We investigate the performance bottleneck of MLGF-MF using Yahoo! Music dataset. Figure 8 shows the convergence speeds of MLGF-MF on shared memory, on RevuAhn, and on S850PRO. We observe that the performance difference among them is very marginal, because (1) the asynchronous I/O offsets the I/O cost of SSD disks, and (2) the CPU cost linearly increases as the dimensionality k of latent matrices U and V increases; As shown in the updating rules (Equation 4 and 5), the required number of floating computation depends on the size of dimensionality of u_i and v_j.

We repeat the experiments with SSE instruction set. Figure 8(b) reports the convergence speeds of three different settings with SSE instruction set. By using SSE instruction set, the CPU cost is reduced, and the CPU computation is not a bottleneck for S850PRO. However, MLGF-MF on RevuAhn still shows almost the same result as MLGF-MF on shared-memory.

These experiments show that the performance barrier of MLGF-MF by the I/O cost is not very high because the I/O cost is offset by the asynchronous I/O, and depending on the k value, CPU cost could be higher than I/O cost.

5.2.4 Evaluation on scalability of MLGF-MF

We investigate the scalability of MLGF-MF with respect to the size of threads. Specifically, we evaluate the speedup factor (the time using single threads / the time using multiple threads) on NetFlix dataset, and compare it to that of FPSGD. Both MLGF-MF and FPSGD show ideal speed up until using four threads (Figure 9(a)), however, the both slows down after four threads.

We also evaluate the scalability of MLGF-MF with respect to the size of dataset. For evaluation, we generate a synthetic dataset by following the synthetic dataset generation procedure in DSGD [5]. Specifically, we fix the size of column (i.e., the number of items) to one million, and increase the number of users from 20 millions to 100 millions with the interval of 20 million. Accordingly, the number of non-zero entries in the matrix increases from 2 billions to 10 billions with the interval of 2 billions. As a result, we generate 5 datasets having the 20 million - 100 million of users, and the size of synthesized matrix is 54G - 274G byte in raw text format. For these datasets, the time for factorizing matrices by MLGF-MF increases in linear scale and does not explode (Figure 9(b)).

5.2.5 Effect of page size

In block-storage environment, according to the page size, the performance may vary. We investigate the performance of MLGF-MF for various sizes of pages. Figure 10(a) shows that the performance change due to the change of page size is very marginal.

Though there is a slight difference on the performance, it is mainly due to the storage utility. For example, in NetFlix, MLGF-MF with 4MByte page size shows the best performance. It is mainly because the storage utility is the best for 4 MByte page size (Figure 10(b)). On the contrary, the storage utility for 512K is the worst, thus the convergence speed is slower than that of 4 MByte. However, the difference is very marginal.

These results show that the convergence speed and storage utilization of MLGF-MF is not sensitive with respect to page size.

6. CONCLUSION

This paper proposes a fast and robust parallel SGD algorithm for matrix factorization, called MLGF-MF, which is robust to skewed matrices, and runs efficiently for both shared-memory and block-storage (e.g., SSD disks) environments. Existing parallel or distributed SGD algorithms produce too much I/O cost and produce unreliable results when the matrix is skewed. To reduce the I/O cost and produce robust results, MLGF-MF 1) exploits MLGF, a multi-dimensional index complementing the grid structure, and 2) adopts the asynchronous I/O which fully utilizes the CPU resources and block-storage devices. From our extensive evaluations, we conclude that MLGF-MF significantly outperforms (or converges faster than) the state-of-the-art SGD algorithms in both shared-memory and block-storage environments, and it is more robust to skewed matrices or achieves higher quality of convergence at faster speed.

7. ACKNOWLEDGEMENTS

8. REFERENCES

[1] R. M. Bell and Y. Koren. Lessons from the netflix prize challenge. *SIGKDD Explor. Newsl.*, 9(2):75–79, Dec. 2007.

[2] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, Sept. 1975.

[3] G. Dror, N. Koenigstein, Y. Koren, and M. Weimer. The Yahoo! Music Dataset and KDD-Cup'11. *JMLR Workshop and Conference Proceedings*, 18:3–18, 2012.

[4] V. Gaede and O. Günther. Multidimensional access methods. *ACM Comput. Surv.*, 30(2):170–231, June 1998.

[5] 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*, KDD '11, pages 69–77. ACM, 2011.

[6] A. Guttman. R-trees: A dynamic index structure for spatial searching. *SIGMOD Rec.*, 14(2):47–57, June 1984.

[7] W.-S. Han, S. Lee, K. Park, J.-H. Lee, M.-S. Kim, J. Kim, and H. Yu. Turbograph: A fast parallel graph engine handling billion-scale graphs in a single pc. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '13, pages 77–85. ACM, 2013.

[8] C.-J. Hsieh and I. S. Dhillon. Fast coordinate descent methods with variable selection for non-negative matrix factorization. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '11, pages 1064–1072. ACM, 2011.

[9] Y. Hu, Y. Koren, and C. Volinsky. Collaborative filtering for implicit feedback datasets. In *Proceedings of the 2008 Eighth IEEE International Conference on Data Mining*, ICDM '08, pages 263–272. IEEE Computer Society, 2008.

[10] J. Kiefer and J. Wolfowitz. Stochastic estimation of the maximum of a regression function. *The Annals of Mathematical Statistics*, 23:462–466, 1952.

[11] Y. Koren. Factorization meets the neighborhood: A multifaceted collaborative filtering model. In *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '08, pages 426–434. ACM, 2008.

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

[13] J.-H. Lee, Y.-K. Lee, K.-Y. Whang, and I.-Y. Song. A region splitting strategy for physical database design of multidimensional file organizations. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, VLDB '97, pages 416–425. Morgan Kaufmann Publishers Inc., 1997.

[14] J. Nocedal and S. J. Wright. *Numerical Optimization*. Springer, New York, 2nd edition, 2006.

[15] A. Papadopoulos, Y. Manolopoulos, Y. Theodoridis, and V. Tsotras. Grid file (and family). In *Encyclopedia of Database Systems*, pages 1279–1282. Springer US, 2009.

[16] 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 24*, pages 693–701. Curran Associates, Inc., 2011.

[17] R. L. Rivest. Partial-match retrieval algorithms. *SIAM Journal on Computing*, 5(1):19–50, 1976.

[18] H. Robbins and S. Monro. A Stochastic Approximation Method. *The Annals of Mathematical Statistics*, 22(3):400–407, 1951.

[19] J. T. Robinson. The k-d-b-tree: A search structure for large multidimensional dynamic indexes. In *Proceedings of the 1981 ACM SIGMOD International Conference on Management of Data*, SIGMOD '81, pages 10–18. ACM, 1981.

[20] K. Y. Whang, S. W. Kim, and G. Wiederhold. Dynamic maintenance of data distribution for selectivity estimation. *The VLDB Journal*, 3(1):29–51, Jan. 1994.

[21] W. Xu, X. Liu, and Y. Gong. Document clustering based on non-negative matrix factorization. In *Proceedings of the 26th Annual International ACM SIGIR Conference on Research and Development in Informaion Retrieval*, SIGIR '03, pages 267–273. ACM, 2003.

[22] H.-F. Yu, C.-J. Hsieh, S. Si, and I. Dhillon. Scalable coordinate descent approaches to parallel matrix factorization for recommender systems. In *Proceedings of the 2012 IEEE 12th International Conference on Data Mining*, ICDM '12, pages 765–774. IEEE Computer Society, 2012.

[23] H. Yun, H.-F. Yu, C.-J. Hsieh, S. Vishwanathan, and I. S. Dhillon. Nomad: Non-locking, stochastic multi-machine algorithm for asynchronous and decentralized matrix completion. In *International Conference on Very Large Data Bases (VLDB)*, sep 2014.

[24] Y. Zhou, D. Wilkinson, R. Schreiber, and R. Pan. Large-scale parallel collaborative filtering for the netflix prize. In *Proceedings of the 4th International Conference on Algorithmic Aspects in Information and Management*, AAIM '08, pages 337–348. Springer-Verlag, 2008.

[25] 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*, RecSys '13, pages 249–256. ACM, 2013.

[26] M. Zinkevich, M. Weimer, L. Li, and A. J. Smola. Parallelized stochastic gradient descent. In J. Lafferty, C. Williams, J. Shawe-Taylor, R. Zemel, and A. Culotta, editors, *Advances in Neural Information Processing Systems 23*, pages 2595–2603. Curran Associates, Inc., 2010.