GASGD: Stochastic Gradient Descent for Distributed Asynchronous Matrix Completion via Graph Partitioning.[*]

Fabio Petroni
Department of Computer Control and Management Engineering Antonio Ruberti,
Sapienza University of Rome
petroni@dis.uniroma1.it

Leonardo Querzoni
Department of Computer Control and Management Engineering Antonio Ruberti,
Sapienza University of Rome
querzoni@dis.uniroma1.it

ABSTRACT

Matrix completion latent factors models are known to be an effective method to build recommender systems. Currently, stochastic gradient descent (SGD) is considered one of the best latent factor-based algorithm for matrix completion. In this paper we discuss GASGD, a distributed asynchronous variant of SGD for large-scale matrix completion, that (i) leverages data partitioning schemes based on graph partitioning techniques, (ii) exploits specific characteristics of the input data and (iii) introduces an explicit parameter to tune synchronization frequency among the computing nodes. We empirically show how, thanks to these features, GASGD achieves a fast convergence rate incurring in smaller communication cost with respect to current asynchronous distributed SGD implementations.

Categories and Subject Descriptors

G.4 [**Mathematics of Computing**]: Mathematical Software — *Parallel and vector implementations*

Keywords

Recommender system; Matrix completion; Stochastic gradient descent; Distributed computing; Graph partitioning

1. INTRODUCTION

Many of todays internet businesses strongly base their success in the ability to provide personalized user experiences. This trend, pioneered by e-commerce companies like Amazon [8], has spread to many different sectors. As of today, personalized user recommendations are commonly offered by internet radio services, social networks, media sharing platforms (e.g. YouTube) and movie rental services.

[*]This work has been partially supported by the TENACE PRIN Project (n. 20103P34XC) funded by the Italian Ministry of Education, University and Research and by the academic project n. C26A133HZY funded by the University of Rome "La Sapienza"

RecSys'14, October 6–10, 2014, Foster City, Silicon Valley, CA, USA.
Copyright 2014 ACM 978-1-4503-2668-1/14/10 ...$15.00.
http://dx.doi.org/10.1145/2645710.2645725 .

Building user recommendations can be abstracted as a *matrix completion problem*; given a set of users and a set of available items (e.g. songs, movies, tweets, etc.), these can be mapped to rows and columns of a *rating matrix* whose values represent preferences expressed by users on items. Building recommendations boils down to estimate, as precisely as possible, all missing values in this matrix.

Recent experiences in this field [4, 2] has promoted algorithms based on *latent factors* as the best approach to the matrix completion problem. These algorithms map users and items to a latent feature space and define their affinity as the product of their latent feature vectors. The basic idea is that latent features, albeit non clearly definable, contain useful information on user and item similarity and can thus be exploited to predict missing ratings. *Stochastic Gradient Descent* (SGD) [7, 12] is considered, as of today, the best latent factor based algorithm for matrix completion in all settings where the rating matrix is huge (e.g. millions of users with hundreds of thousands items and billions of available ratings). The SGD success stems from the availability of efficient parallel implementations [16, 9, 10] that make it possible to efficiently exploit modern multi-processor and multi-core computing architectures. However, as the problem size grows to massive scale the memory occupation of such algorithms become the limiting factor: the cost of computing infrastructures skyrockets, while the performance tend to grow non linearly due to more frequent contention on data shared among running processes. Thereby, the research trend is now focussed on *distributed SGD* implementations whose aim is to make the analysis of huge data sets feasible on possibly large clusters of cheap commodity machines. The shift toward distributed implementations is a key move to make recommendation algorithms a service easily and cheaply available through cloud platforms.

Recent proposals [13, 5] represent important steps in the right direction, but still suffer from some limitations that hinder their performance. Firstly, all these solutions divide the input dataset among the available SGD instances leveraging, at best, randomized partitioning approaches; partitioning is performed by subdividing the rating matrix in blocks of the same size; however, no attention is payed to the amount of data (user preferences) in each block, even if this value drives the real load of the SGD instance that will compute on that matrix block; as a consequence, this "blind" approach to partitioning, albeit easily applicable, can cause skewed load on the computing nodes. Furthermore, it is inefficient as it does not appropriately exploit some intrinsic characteristics of the input data. Secondly, all these solutions employ a *bulk synchronous processing* approach where shared data values are concurrently updated on local copies and then periodically resynchronized. The synchronization frequency is a fundamental parameter for these solutions as it regulates a crucial tradeoff between the convergence rate of the algorithm toward the solution and the communication cost incurred during its functioning. To the best of our knowledge, none of the existing works in this field explored and leveraged this tradeoff.

Our proposal is to address the aforementioned problems with three distinct contributions. On one side we mitigate the load imbalance among SGD instances by proposing an input slicing solution based on graph partitioning algorithms (contribution 1). This approach looks at the input data as a graph where each node represents either a user or an item and each edge represents a rating expressed by the former on the latter. The graph is then greedily partitioned edge by edge thus providing a substantially more balanced load among the available computing nodes. A better balanced load has a positive impact on the overall efficiency of the system as less loaded processes will not wait idly while more loaded processes delay the bulk data synchronization phase (assuming homogeneous computing nodes).

On the other side we attack the efficiency problem with two distinct solutions:

1. By properly leveraging known characteristics of the input dataset, i.e. the fact that it is bipartite (users can only express ratings on items) and with skewed degree distributions (i.e. the user base is usually order of magnitude larger than the item base and the distribution of user ratings expressed on items follows a power-law) we show how to reduce the number of shared data values among SGD instances (contribution 2).

2. By tuning the frequency used by SGD instances to start the bulk synchronization phase during the computation we show how to leverage the tradeoff between communication cost and algorithm convergence rate to improve system performance without affecting the overall quality of the final result (contribution 3).

All our three contributions nicely integrate with the current state of the art as they don't require modifications to existing asynchronous distributed SGD algorithms, but rather complement them. In particular, contributions 1 and 2 are integrated in a *input partitioner* that must be used to preprocess input data before running the SGD algorithm, while contribution 3 is provided as an empirical evaluation on well known publicly-available data traces.

In the following we will first introduce the reader to the the matrix completion problem (Section 2) and then detail the SGD algorithm with its parallel and distributed variants (Section 3). Our novel input partitioner included in GASGD is then described in Section 4, followed by an experimental evaluation (Section 5) that backups our claims. Section 5 also contains a description of contribution 3. Finally, an analysis of the related work (Section 6), and some final remarks (Section 7) will conclude this paper.

2. THE MATRIX COMPLETION PROBLEM

We consider a system constituted by $U = (u_1, \cdots, u_n)$ users and $X = (x_1, \cdots, x_m)$ items. Items represent a general abstraction that can be case by case instantiated as news, tweets, shopping items, movies, songs, etc. Users can rate items with values from a predefined range. Without loss of generality, here we assume that ratings are represented with real numbers. By collecting user ratings it is possible to build a $n \times m$ rating matrix R that is usually a sparse matrix as each user rates a small subset of the available items. Denote by $O \subseteq \{1, ..., n\} \times \{1, ..., m\}$ the set of indices identifying observed entries in R; $(i, j) \in O$ implies that user u_i rated item x_j with vote r_{ij}. The *training set* is defined as $T = \{r_{ij} : (i, j) \in O\}$. The goal of a *matrix completion algorithm* is to predict missing entries $\widehat{r}_{ij}$ in R using ratings contained in the training set. As demonstrated by the Netflix competition [2] and the KDD-Cup 2011 [4], collaborative approaches (i.e., *collaborative filtering* techniques [11]) based on latent factors are today considered the best models for large-scale recommender systems.

Denote by $k \ll min(n, m)$ a rank parameter. These kind of solutions aim at finding an $n \times k$ row-factor matrix P^* (*user vectors matrix*) and an $k \times m$ column-factor matrix Q^* (*item vectors matrix*) such that $R \approx P^* Q^*$. The approximation is performed by minimizing an application dependent error function $L(P, Q)$, that measures the quality of the reconstruction. We call p_i, the i-th row of P, the k-dimensional vector of user u_i and q_j, the j-th column of Q, the k-dimensional vector of item x_j. Unobserved entries $\widehat{r}_{ij} \in R, (i, j) \notin O$ are predicted by multiplying the corresponding user and item vectors $p_i q_j$. There exists a wide range of objective functions for matrix completion and factorization. The most used error function is the regularized squared loss [7, 12, 13, 16]:

$$L(P, Q) = \sum_{(i,j) \in P} (r_{ij} - p_i q_j)^2 + \lambda(||P||_F^2 + ||Q||_F^2) \qquad (1)$$

where $||\cdot||_F$ is the Frobenius norm and $\lambda \geq 0$ is a regularization coefficient used to avoid overfitting. In accordance with [13], we observe that Equation 1 is in *summation form* as it is expressed as a sum of *local losses* L_{ij} for each element in R:

$$L_{ij}(P, Q) = (r_{ij} - p_i q_j)^2 + \lambda \sum_{k=1}^{r} (P_{ik}^2 + Q_{kj}^2) \qquad (2)$$

The most popular techniques to minimize the objective function are *Alternating Least Squares* (ALS) and *Stochastic Gradient Descent* (SGD). ALS [15] alternates between keeping P and Q fixed. The idea is that, although both these values are unknown, when the item vectors are fixed, the system can recompute the user vectors by solving a *least-squares* problem (that can be solved optimally), and vice versa. In this paper we will focus on SGD [7, 12], since it has been shown that it performs better on large-scale data [13]. SGD, in fact, was the approach chosen by the top three solutions of KDD-Cup 2011 [4].

3. STOCHASTIC GRADIENT DESCENT

SGD works by iteratively updating current estimations of P and Q with values proportional to the negative of the gradient of the error function. The term *stochastic* means that P and Q are updated, at each iteration, by a small step for each given training case toward the average gradient descent. For each observed entry $(i, j) \in O$, the objective function is expressed by Equation 2 and the variables are updated proportionally to the sub-gradient over p_i and q_j:

$$p_i \leftarrow p_i + \mu(\varepsilon_{ij} q_j - \lambda p_i) \tag{3}$$

$$q_j \leftarrow q_j + \mu(\varepsilon_{ij} p_i - \lambda q_j) \tag{4}$$

where $\varepsilon_{ij} = r_{ij} - p_i q_j$ is the error between the real and predicted ratings for the (i, j) entry, and μ is the learning rate. Therefore, in each SGD step only the involved user and item vectors are updated.

The algorithm proceeds performing several iterations through the available ratings until a convergence criterion is met. We refer to a single iteration over the data as an *epoch*.

The SGD algorithm is, by its nature, inherently sequential; however, sequential implementations are usually considered poorly scalable as the time to convergence for large-scale problems may quickly grow to significant amounts. Parallel versions of SGD have been designed to overcome this problem by sharing the computation among multiple processing cores working on shared memory. A straightforward approach [10] to manage concurrent updates of shared variables is to lock, before processing training point $(i, j) \in O$, both row p_i and column q_j. However, lock-based approaches are known to adversely affect concurrency and, in the end, limit the scalability of the algorithm. HogWild [9] proposed a lock-free version assuming an highly sparse rating matrix. Recently, Zhuang et al. [16] proposed FPSGD, a fast parallel SGD implementation for shared memory systems. They tackled similar problems to our: (i) how to reduce the cost that the algorithm faces when retrieving data and (ii) how to ensure that all the threads will process the same amount of data. The algorithm partitions the training data into several blocks (more than the available threads) and uses a task manager to distribute the load among the threads. Beside these recent improvements, parallel SGD algorithms are hardly applicable to large-scale datasets, since the time-to-convergence may be too slow or, simply, the input data may not fit into the main memory of a single computer. Storing training data on disk is inconvenient because the two-dimensional nature of the rating matrix R will force non-sequential I/O making disk-based SGD approaches unpractical and poorly performant, although technically feasible. These problems recently motivated several research efforts toward distributed versions of SGD.

Distributed SGD Algorithms.

This kind of algorithms [13, 3, 1] are designed for very large scale instances of the matrix completion problem.

DSGD [5] exploits the fact that some blocks of the rating matrix R are mutually independent (i.e., they share neither any row nor any column) so that the corresponding user and item vectors can be updated concurrently. For each epoch, several sequences of independent blocks of equal size (that constitute a *stratum*) are selected to cover the entire available data set. Then the algorithm proceeds by elaborating each stratum sequentially, assigning each block to a different computing node, until all the input data have been processed; at that point a new epoch starts. It is important to notice that with this solution a processing node must have complete knowledge of the rating matrix R in order to decompose it in blocks and strata, but this is unfeasible if the input data does not fit in the main memory of a single machine.

To cope with such massive scale, current solutions distribute the rating matrix R among the set of computing nodes C, so that each of them only owns a slice of the input data. A problem with such approach is that, in general, the input partitioner is forced to assign ratings expressed by a single user (resp. received by a single item) to different computing nodes, in order to maintain the load in the system balanced. Thereby, user and item vectors must be concurrently updated, during the SGD procedure, by multiple nodes. A common solution is to replicate vectors on all the nodes that will work on them, forcing synchronization among the replicas via message exchange.

The main challenge faced by distributed SGD algorithms is thus how to effectively partition the data across multiple processing node, in such a way that: (i) all the computing nodes are approximately fed with the same load and (ii) the communication between the computing nodes is minimized (iii) without compromising the quality of the result.

Given that each node has a local view of the vectors it works on, the algorithm needs to keep vector replicas on different nodes from diverging. This is achieved in two possible ways: either by maintaining replicas always in synch by leveraging a locking scheme (*synchronous* approach), or by letting nodes concurrently update their local copies and then periodically resynchronizing diverging vectors (*asynchronous* approach). Synchronous distributed SGD algorithms all employ some form of locking to maintain vector copies synchronized. This approach is however inefficient, because computing nodes spend most of their time in retrieving and delivering vector updates in the network, or waiting for lock to be released. The strictly sequential computation order imposed by this locking approach on shared vector updates negatively impacts the performance of such solutions.

Distributed Asynchronous SGD.

Differently from synchronous algorithms, in distributed asynchronous SGD (ASGD) algorithms computing nodes are allowed to concurrently work on shared user/item vectors, that can therefore deviate inconsistently during the computation. The system defines for each vector a unique *master copy* and $s \in [0, |C|)$ *working copies*. In the following we will refer to the node that store the master copy of a vector as *master node*. Each computing node updates only the local working copy of p_i and q_j[1] while processing training point $(i, j) \in O$. The synchronization between working copies and master is performed periodically according to the *Bulk Synchronous Processing* (BSP) model [13]. Initially all the vector copies are synchronized with their corresponding masters. The synchronization procedure is then repeated f times during each epoch. We refer to the parameter f as the *synchronization frequency* of the algorithm.

In this study we considered the following ASGD algorithm. At the beginning of an epoch, each node shuffles the subset data that it owns and divides it in f folds of equal size. Consequently, each epoch is divided in f steps. Each step y consists of: (i) a computation phase, where each node updates the local working copy of user and item vectors using data in the y-th fold; (ii) a global message transmission phase, where each node sends all the vector working copies that have been updated in the previous phase to the corresponding masters; (iii) a barrier synchronization, where each

[1]Also the vector master node updates a local working copy.

master node collects all the vector working copies, compute the new vector values, and sends back the result. New vector values, computed in phase (iii), are obtained by performing a weighted average of the received working copies. The weight is given by the number of training points in the y-th fold that lead to the update of the vector (i.e., the number of ratings in the y-th fold expressed by the user or received by the item). In [14] an exhaustive theoretical study of the convergence of ASGD is presented.

Current ASGD state-of-the-art implementations synchronize vector working copies either continuously during the epoch ($f = \frac{|T|}{|C|}$, each fold contains a single rating) or once after every epoch ($f = 1$) [13]. Obviously, the first approach ($f = \frac{|T|}{|C|}$) guarantees a *convergence rate*, defined as the speed at which the objective function approaches its lower limit with respect to the epochs passing, almost equal to the centralized version of SGD, since every update is from time to time notified to the entire system. This, however, comes at the cost of a huge number of messages that have to be exchanged in the system (i.e. large *communication cost*). Conversely, the second alternative ($f = 1$) incurs in the minimum communication cost, but its convergence rate can be quite slow. In Section 5 we will report on our empirical study of f that shows the tradeoff between convergence rate and communication cost, and how this can be leveraged to improve the efficiency of ASGD algorithms.

Another problem of current implementations is how to balance the load among the computing nodes. A known problem of the BSP approach, namely the *curse of the last reducer* [1], is that the slowest machine determines the runtime of each step. This shortcoming can be mitigated, assuming homogeneous computing resources, by perfectly balancing the load among the available nodes.

The problem is that, in general, the input ratings are not uniformly distributed in R. A common approach to input data partition is to grid the rating matrix R in $|C|$ blocks and then assign each block to a different computing node [13]. This partitioning approach clearly cannot guarantee a balanced number of ratings for each block, thus can possibly cause strong load imbalance among computing nodes. The authors of [13] proposed to mitigate this problem by applying some random permutations of columns and rows. While this approach improve the load balancing aspect, it still lead to non negligible skews in the rating distributions (See Section 5 for an empirical evaluation of this aspect). Furthermore, a second problem of the grid partitioning approach is that the communication cost between the computing nodes is not considered as a factor to be minimized. Matrix blocking, in fact, is performed without considering the relationships connecting users with items in R. As a consequence, the number of replicas for each user/item vector can possibly grow to the number C of available nodes.

In the next section we introduce a novel input splitter based on a graph partitioning approach that aims at solving both these problems.

4. INPUT PARTITIONER

Our partitioner treats input data as a graph, where users and items represent nodes and ratings represent edges. It works by assigning each edge in exactly one of C partitions (i.e. subgraphs of the input graph), each managed by a different computing node. As a baseline to our graph partitioning scheme we follow [6, 1]. In particular, we perform a balanced $|C|$-way vertex-cut partitioning of the graph, that aims both at minimizing the number of shared vertices and at balancing the load among the computing nodes.

The motivation behind the vertex-cut approach, with respect to the classical edge-cut one, is that SGD stores and exchanges data (i.e., user and item vectors) that are associated with vertices rather than edges. Moreover, one characteristic of real preference datasets is their skewed power-law degree distribution: most vertices have relatively few connections while a few have many. Figure 1 shows this characteristics from three popular datasets, namely MovieLens, Netflix and Yahoo! (see Section 5). It has been shown [6] that the vertex-cut approach performs better than edge-cut in these scenarios. Intuitively, by replicating the few very high degree vertices it is possible to quickly partition the graph, while maintaining a balanced load among the available computing nodes.

Given a weighted undirected graph $G = (V, E)$ that represents a rating matrix R, where $V = U \cup X$ and $e_{i,j} \in E$ iff $(i, j) \in O$, the input partitioner puts each edge $e_{i,j}$, together with the vertices v_i and v_j it connects, in one of the C available nodes. At the end of this operation, each edge will be stored exactly in one node, while some vertices will be replicated in more than one node. Note that, if a vertex resides in a single node (i.e. it has only one replica) no synchronization is needed for him in ASGD. However, in the general case, the storage overhead and the communication cost incurred by ASGD at runtime depend on the vertex replication factor, defined ad the average number of vertex replicas (i.e., $RF = \#replicas/\#vertices$). The goal of the input partitioner is thus twofold: (i) minimizing the *replication factor* (RF) and (ii) balance as much as possible the edge load among the available computing nodes. This objective is formalized by the *balanced Ω-way vertex-cut* problem [6], where Ω is the number of partitions (in our case $|C|$). Each edge $e_{i,j} \in E$ is assigned to a single node $A(e) \in C$. Each vertex v is then replicated on the set of nodes $A(v) \subseteq C$ that contain its adjacent edges. The balanced $|C|$-way vertex-cut objective function is defined as follows:

$$\min \frac{1}{|V|} \sum_{v \in V} |A(v)| \tag{5}$$

$$s.t. \quad \max_{c \in C} |\{e_{i,j} \in E | A(e) = c\}| < \sigma \frac{|E|}{|C|} \tag{6}$$

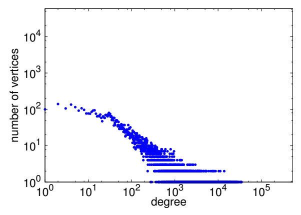
where $\sigma \geq 1$ is a small constant that defines the system tolerance to load imbalance.

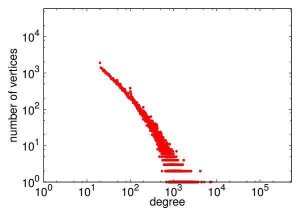
For each vertex v with multiple replicas, one of the replicas is randomly selected as the one which maintains the master vector associated with the vertex. All remaining replicas of v maintain a working copy of the vector.

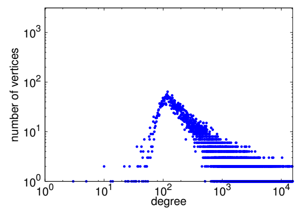
Greedy Vertex-Cut Streaming Algorithm.

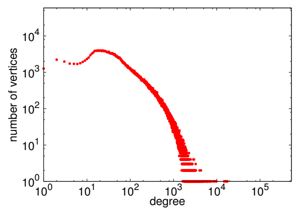
Since we are here assuming that the input data doesn't fit in main memory, the algorithm used to partition it among the available nodes must work in a streaming fashion, requiring only a single pass over the input data. A limitation of this approach is that the assignment decision taken on an input element (i.e. an edge) can only be based on previously analyzed data and cannot be later changed.

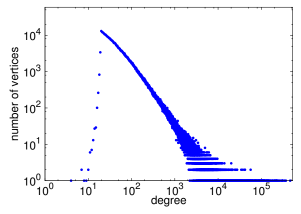
A simple solution is given by the *hashing* algorithm that pseudo-randomly assigns each edge to a partition. This solu-


(a) MovieLens - items


(b) MovieLens - users


(c) Netflix - items


(d) Netflix - users


(e) Yahoo! - items

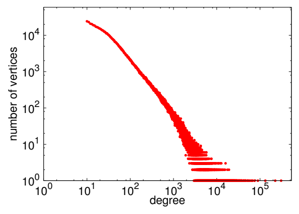
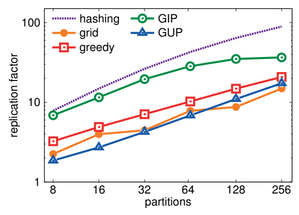
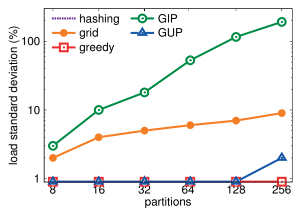

(f) Yahoo! - users

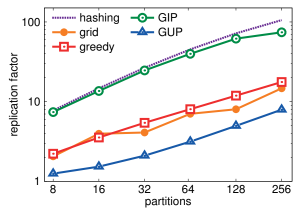
Figure 1: The items and users degree distributions in the considered datasets (log-log scale).



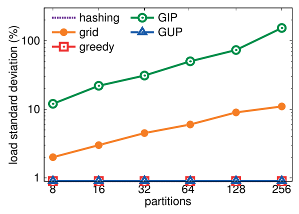
(a) MovieLens - *RF*



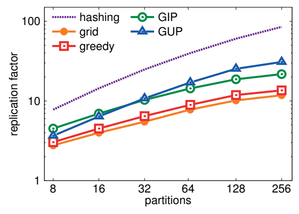
(b) MovieLens - RSD



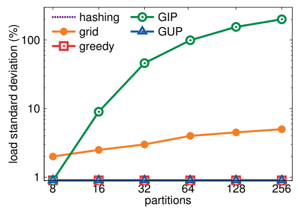
(c) Netflix - *RF*



(d) Netflix - RSD



(e) Yahoo! - *RF*



(f) Yahoo! - RSD

Figure 2: Replication factor and load RSD achieved by the various partitioning schemes (log-log scale).

tion achieves nearly perfect load balance on large graphs, but provides results with very high replication factors. A better solution is represented by a greedy approach [6, 1] that breaks the randomness of the hashing solution by maintaining some global status information. In particular, the system stores the set of computing nodes $A(v)$ to which each already observed vertex v has been assigned[2]. The algorithm, when processing edge $e_{i,j} \in E$ operates as follows:

Case 1 If neither u_i nor x_j have been assigned to a node, then $e_{i,j}$ is assigned to the least loaded node $c_l \in C$. c_l becomes the master node of u_i and x_j.

Case 2 If only one of the two vertices has been already assigned (without loss of generality assume that u_i is the assigned vertex) then $e_{i,j}$ is placed in the least loaded $c_l \in A(u_i)$. c_l becomes the master node of x_j.

Case 3 If $A(u_i) \cap A(x_j) \neq \emptyset$, then the edge $e_{i,j}$ is assigned to the least loaded node $c_l \in A(u_i) \cap A(x_j)$.

Case 4 If $A(u_i) \neq \emptyset$, $A(x_j) \neq \emptyset$ and $A(u_i) \cap A(x_j) = \emptyset$, then $e_{i,j}$ is assigned to the least loaded node $c_l \in A(u_i) \cup A(x_j)$. Without loss of generality assume $c_l \in A(u_i)$, then c_l becomes a working node for x_j.

We implemented the above procedure in a parallel fashion. In particular, the input data is randomly divided between the computing threads/nodes. All the threads/nodes have access to a shared key-value storage that maintains as key the id of the vertex v and as value $A(v)$. To manage concurrent access we define fine-grained locks, one for each entry.

[2]The memory footprint needed to run such algorithm is orders of magnitude smaller than the input data.

The novelty of our solution lies on the idea to exploit the bipartite nature of the considered graph. A bipartite graph is a graph where the vertex set V can be split in two disjoint subsets $W1$ and $W2$ such that all edges connect vertices placed in distinct subsets: if $\{v, w\} \in E$, either $v \in W1$ and $w \in W2$ or $v \in W2$ and $w \in W1$. Such graphs are natural in the area of recommender systems, where vertices represent distinct classes of objects, e.g. users and items.

Bipartite Aware Greedy Algorithm.

Our algorithm exploits the fact that in real word datasets the size of the two sets constituting the bipartite graph are often significantly skewed: one of the two sets is much bigger that the other. If this is the case, by perfectly splitting the bigger set it is possible to achieve an average replication factor smaller than the one obtained through the greedy approach (although we will show in Section 5 that this is not always enough to achieve a smaller communication cost).

It is therefore possible to identify two dual approaches:

- *item partitioned* strategy. Vectors for all items $x_j \in X$ always reside in a single node while vectors for users $u_i \in U$ are replicated as necessary.

- *user partitioned* strategy. Vectors for all users $u_i \in U$ always reside in a single node while vectors for items $x_j \in X$ are replicated as necessary.

Dataset	MovieLens	Netflix	Yahoo!
n	69878	480189	1000990
m	10677	17770	624961
$\lvert E\rvert$	10000054	100480507	252800275
k	50	50	50
λ	0.05	0.05	1
μ	0.015	0.015	0.0001

Table 1: Statistics and parameters for each dataset.

To achieve this result we modify *Case 4* of the greedy algorithm, that is the only case where replicas are generated. Assume without loss of generality a *user partitioned* strategy, so $|A(u_i)| = 1, \forall u_i \in U$.

Case 4 If $A(u_i) \notin A(x_j)$ then $e_{i,j}$ is assigned to $A(u_i)$, and the unique node in $A(u_i)$ creates a working copy for x_j.

5. EVALUATION

In this section we report the results of the experimental evaluation we conducted on a prototype implementation of GASGD. The goal of this evaluation was to show how the GASGD characteristics provide a solution able to reach high quality results in an efficient manner.

Experimental Settings and Test Datasets.

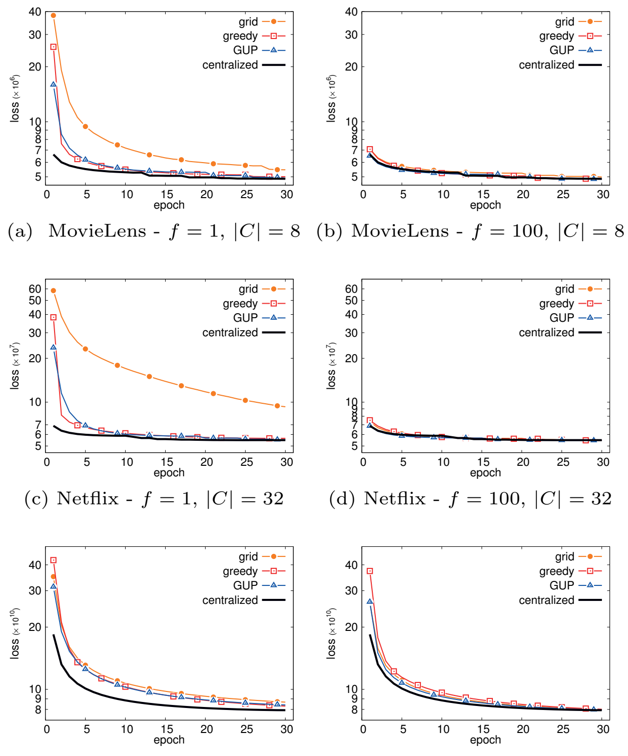
We used three datasets for the experiments: MovieLens 10M[3], Netflix [2] and Yahoo! [4]. Dataset statistics as well as simulation parameters are synthesized in Table 1. The reported learning rate μ only represents the starting value, as we adopted an adaptive mechanism called *bold driver* [13] to automatically vary the parameter during the computation. Furthermore, in our implementation each computing node shuffles the training points it analyzes before each epoch.

We implemented five different input partitioning scheme: *grid*, *hashing*, *greedy*, *greedy - item partitioned* (*GIP*), *greedy - user partitioner* (*GUP*). The *grid* partitioner, commonly used by DSGD (Section 3), simply shuffles rows and columns of the rating matrix R, and then divides it in $|C|$ identical blocks. The other four schemes are based on graph partitioning techniques (Section 4). As a baseline for comparison we used the solution obtained by running SGD on a single machine. For this centralized approach we employed the parallel-SGD algorithm (with locks) described in Section 3.

Partitioning quality.

We begin our experimental analysis by studying the quality of the various partitioning schemes, in terms of achieved replication factor (average number of copies for each vector) and load balancing (expressed as the *Relative Standard Deviation* (RSD) of the load among the nodes). Figure 2 reports the obtained results. As expected *GUP* achieves the smallest *RF* in those scenarios where the number of users is much bigger than the number of items (MovieLens and Netflix), sporting, moreover, a perfect load balance among the computing nodes. However, in the Yahoo! dataset users and items are comparable in number and the classical *greedy* solution outperforms its *user partitioned* variant. Interestingly, the *grid* solution always outperforms the *greedy* approach in *RF*, even if its load balancing performance are inferior. Intuitively, by shuffling rows and columns of R and then grouping them in blocks, *grid* achieves the same objective of *greedy*, that is to replicate users or items with high degree, but does so in a more effective way, since it has complete knowledge of the rating matrix R while *greedy* processes the training set in a streaming fashion.

[3]http://grouplens.org/datasets/movielens



(a) MovieLens - $f = 1$, $|C| = 8$ (b) MovieLens - $f = 100$, $|C| = 8$

(c) Netflix - $f = 1$, $|C| = 32$ (d) Netflix - $f = 100$, $|C| = 32$

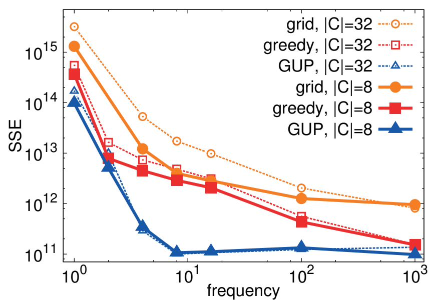
(e) Yahoo! - $f = 1$, $|C| = 4$ (f) Yahoo! - $f = 100$, $|C| = 4$

Figure 3: Algorithms convergence rate (loss vs epochs), with $f = 1$ and $f = 100$ (y-axis in log-scale).

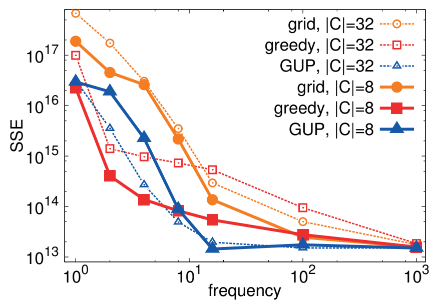
Overall SGD performance.

To assess the quality of the solution obtained from the asynchronous distributed SGD variants, we compare the convergence curves of the objective function expressed by Equation 1. Note that here our goal was not to assess the absolute solution quality achieved by the algorithm (i.e., RMSE), but rather to show how fast the solutions provided by the distributed solutions approach the centralized one. Figure 3 shows the loss curves for the various algorithms, applying two different synchronization frequencies: $f = 1$ and $f = 100$. In general, by increasing f the ASGD curves tend to reproduce the centralized SGD trend, while with $f = 1$ the distributed convergence rates are quite far from the centralized one. This was expected, since with $f = 1$ all the vector copies are free to diverge during the computation of an epoch while with $f = 100$ they are frequently synchronized, closely reproducing the behavior of a centralized implementation. The worst performance, with small synchronization frequencies, is provided by the *grid* solution: by shuffling R, in fact, *grid* uniformly distributes the ratings of an entity (i.e. user or item) among the available nodes, thereby favoring the divergence of vector copies.

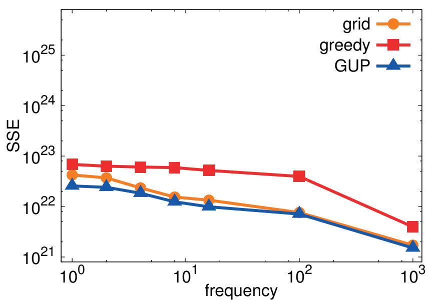
To synthesize these results we used an aggregate value for the convergence curves difference: the *sum of squared*



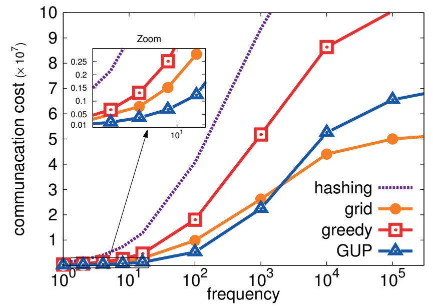
(a) MovieLens - SSE



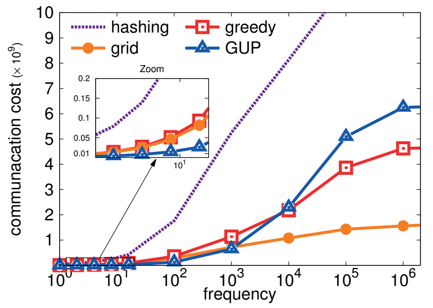
(b) Netflix - SSE



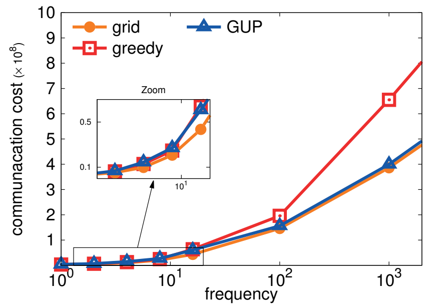
(c) Yahoo! -SSE, $|C| = 4$



(d) MovieLens - CC, $|C| = 8$



(e) Netflix - CC, $|C| = 32$



(f) Yahoo! - CC, $|C| = 4$

Figure 4: SSE between ASGD variants and SGD convergence curves (log-log scale), in the top half. Communication cost varying synchronization frequency (x-axis in log scale), in the bottom part.

errors (SSE) between the ASGD curve points and the centralized ones (limited to the first 30 epochs). The top half of Figure 4 (Figures 4a, 4b and 4c) reports the obtained SSE values varying the synch frequency f. In datasets with significantly more users than items (MovieLens and Netflix) the *GUP* strategy convergence curve quickly approaches the centralized one (respectively with $f = 8$ and $f = 16$) while other solutions achieve the same result with larger frequencies. Furthermore, while increasing the number of nodes degrades the performance of both *greedy* and *grid* (dashed lines in the figures), the *GUP* solution is not affected by this problem; it actually sports a performance improvement to its convergence rate in the Netflix dataset (Figure 4b). Also in the Yahoo! dataset the *GUP* approach outperforms the other variants in convergence rate (Figure 4c).

Communication Cost.

The *GUP* loss curve is the one more quickly approaching the centralized SGD curve for the smallest value of f. Now we will show how, at that frequency f, the *GUP* solution is also the least expensive with respect to communication cost. We express the communication cost (CC) as the total number of messages that the system exchanges in an epoch. This cost depends on three factors: the number $|C|$ of processing nodes, the synchronization frequency f and the replication factor (RF). We remark that RF is a weighted average of the *replication factor* of item (RF_X) and user (RF_U) vectors.

More formally, each node $c \in C$ owns a slice T^c of the training set T, where $|T^c| = \frac{|T|}{|C|}$ assuming perfect load balance. At the beginning of each epoch all computing nodes shuffle the content of their slice T^c and cut it in f folds such that $T^c = \{K_1^c \cup ... \cup K_f^c\}$. Each fold K_y^c contains roughly $\frac{|T^c|}{f}$ ratings. We define $U_y^c = \{u_i \in U : \exists r_{ij} \in K_y^c\}$ (resp. $X_y^c = \{x_j \in X : \exists r_{ij} \in K_y^c\}$) the set of users (resp. items) with at least a rating in the fold. We denote with $U_y = \{U_y^1 \cup ... \cup U_y^{|C|}\}$ (resp. $X_y = \{X_y^1 \cup ... \cup X_y^{|C|}\}$) the set of users (resp. items) with at least one rating in the y-th fold on any computing node. The communication cost for an epoch is then defined as:

$$CC \approx \sum_{c \in C} \sum_{y=1}^{f} (|U_y^c| + |X_y^c|) + \sum_{y=1}^{f} \Big(\sum_{u \in U_y} |A(u)| + \sum_{x \in X_y} |A(x)| \Big) \quad (7)$$

The first part of Equation 7 considers messages that the working copies send to the master, while the second one considers messages that the vector master copies send to all the replicas after the update. For the sake of simplicity the equation does not consider the fact that working and master copies overlap on master nodes, and so they don't need a messages exchange.

The bottom half of Figure 4 (Figures 4d, 4e and 4f) shows the CC incurred by the various solutions varying f. To better understand the charts we show what happens to CC for the extreme values of f: $f = 1$, single synchronization at the end of each epoch, and $f = \frac{|T|}{|C|}$, vectors are synchronized after each observation.

$$f = 1 \rightarrow \quad CC \approx 2(|U|RF_U + |X|RF_X) = 2|V|RF \quad (8)$$

$$f = \frac{|T|}{|C|} \rightarrow \quad CC \approx |T|(2 + RF_U + RF_X) \quad (9)$$

By looking at (8), it is clear that, at low frequencies, the solution with the smallest RF achieves the lowest CC (i.e. *GUP* in MovieLens and Netflix, *grid* in Yahoo!). The situation changes by increasing f, since, for instance, in the

Netflix dataset (Figure 4e) the *GUP* solution gradually becomes the most expensive. This behavior is explained by Equation 9. At high frequencies, in fact, the *CC* basically depends on the sum of RF_U and RF_X. The *GUP* strategy, by partitioning the user set, is forced to highly replicate all items. This behavior has a small impact on *RF*, as the number of users is larger than the number of items; however, *GUP* sports large RF_X and thus incurs highly expensive synchronization phases for high sync frequencies.

The positive aspect highlighted by these tests is that it's not necessary to massively increase f in order to achieve a convergence rate similar to a centralized implementation. A dozen synchronizations per epoch, in fact, are enough for *GUP* to reach this goal in both the MovieLens and Netflix datasets (Figures 4a and 4b), and at such frequency it is the most efficient solution (Figures 4d and 4e). For the Yahoo! case, instead, a slightly higher frequency is required for *GUP* to reach the centralized convergence rate (Figure 4c). At that frequency, *GUP* is just a bit more expensive in CC than *grid* (Figure 4f), but with the advantage of (i) an almost perfect load balance and (ii) the streaming implementation, that makes it applicable to huge input data (i.e. exceeding the memory capacity of a single node).

6. RELATED WORK

The scalability of collaborative filtering algorithms is a research topic that was recently interested by a huge growth of contributions from the scientific community [13, 5, 16, 9, 10]; we already discussed most of them in Section 3 as they represent a cornerstone for the work proposed in this paper, so this section is devoted to a short description of a few further works that are connected to this paper.

The vertex-cut distributed graph placement problem is addressed by PowerGraph [6], a distributed graph-parallel computation paradigm, in which each vertex in parallel executes some software, respecting the Gather-Apply-Scatter (GAS) programming model. We already described their greedy partitioning algorithm in Section 4.

In the context of graph factorization, [1] proposes an asynchronous SGD version where the global explicit synchronization phase is eliminated. Vector copies, indeed, are synchronized continuously (a thread loops in local memory) in an asynchronous fashion and independently. We think that also in this case it's possible to tune the frequency of the synchronization, in a fine-grained fashion, and we leave this idea as part of our future work.

7. CONCLUSIONS

In this paper we described three distinct contributions aimed at improving the efficiency and scalability of Asynchronous distributed SGD algorithms. In particular, we proposed a novel input slicing solution based on graph partitioning approach that mitigates the load imbalance among SGD instances (i.e. better scalability), while, at the same time, providing lower communication costs during the algorithm execution (i.e. better efficiency) by exploiting specific characteristics of the training dataset. The paper also introduces a synchronization frequency parameter driving a tradeoff that can be accurately leveraged to further improve the algorithm efficiency.

8. REFERENCES

[1] A. Ahmed, N. Shervashidze, S. Narayanamurthy, V. Josifovski, and A. J. Smola. Distributed large-scale natural graph factorization. In *Proceedings of the 22nd international conference on World Wide Web*, 2013.

[2] J. Bennett and S. Lanning. The netflix prize. In *Proceedings of KDD cup and workshop*, 2007.

[3] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, M. Ranzato, A. W. Senior, P. A. Tucker, et al. Large scale distributed deep networks. In *Conference on Neural Information Processing Systems*, 2012.

[4] G. Dror, N. Koenigstein, Y. Koren, and M. Weimer. The yahoo! music dataset and kdd-cup'11. *Journal of Machine Learning Research-Proceedings Track*, 18:8–18, 2012.

[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*, 2011.

[6] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[7] Y. Koren, R. Bell, and C. Volinsky. Matrix factorization techniques for recommender systems. *Computer*, 42(8):30–37, 2009.

[8] J. Mangalindan. Amazon's recommendation secret. `http://tech.fortune.cnn.com/2012/07/30/amazon-5` CNN Money, 2012.

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

[10] B. Recht and C. Ré. Parallel stochastic gradient algorithms for large-scale matrix completion. *Mathematical Programming Computation*, 5(2):201–226, 2013.

[11] X. Su and T. M. Khoshgoftaar. A survey of collaborative filtering techniques. *Advances in Artificial Intelligence*, 2009:4, 2009.

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

[13] C. Teflioudi, F. Makari, and R. Gemulla. Distributed matrix completion. In *International Conference on Data Mining*, 2012.

[14] J. N. Tsitsiklis, D. P. Bertsekas, M. Athans, et al. Distributed asynchronous deterministic and stochastic gradient optimization algorithms. *IEEE transactions on automatic control*, 31(9):803–812, 1986.

[15] Y. Zhou, D. Wilkinson, R. Schreiber, and R. Pan. Large-scale parallel collaborative filtering for the netflix prize. In *Algorithmic Aspects in Information and Management*, pages 337–348. Springer, 2008.

[16] 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*, 2013.