Under consideration for publication in Knowledge and Information Systems

Shared-Memory and Shared-Nothing Stochastic Gradient Descent Algorithms for Matrix Completion

Faraz Makari[1], Christina Teflioudi[1], Rainer Gemulla[1], Peter Haas[2] and Yannis Sismanis[2]

[1]Max-Planck-Institut für Informatik, Saarbrücken, Germany; {fmakari,chteflio,rgemulla}@mpi-inf.mpg.de
[2]IBM Almaden Research Center, San Jose, CA, USA; {phaas, syannis}@us.ibm.com

Abstract. We provide parallel algorithms for large-scale matrix completion on problems with millions of rows, millions of columns, and billions of revealed entries. We focus on in-memory algorithms that run either in a shared-memory environment on a powerful compute node or in a shared-nothing environment on a small cluster of commodity nodes; even very large problems can be handled effectively in these settings. Our ASGD, DSGD-MR, DSGD++, and CSGD algorithms are novel variants of the popular stochastic gradient descent (SGD) algorithm, with the latter three algorithms based on a new "stratified SGD" approach. All of the algorithms are cache-friendly and exploit thread-level parallelism, in-memory processing, and asynchronous communication. We investigate the performance of both new and existing algorithms via a theoretical complexity analysis and a set of large-scale experiments. The results show that CSGD is more scalable, and up to 60% faster, than the best-performing alternative method in the shared-memory setting. DSGD++ is superior in terms of overall runtime, memory consumption, and scalability in the shared-nothing setting. For example, DSGD++ can solve a difficult matrix completion problem on a high-variance matrix with 10M rows, 1M columns, and 10B revealed entries in around 40 minutes on 16 compute nodes. In general, algorithms based on stochastic gradient descent appear to perform better than algorithms based on alternating minimizations, such as the PALS and DALS alternating least-squares algorithms.

Keywords: parallel and distributed matrix completion; low-rank matrix factorization; stochastic gradient descent; recommender systems

1. Introduction

Low-rank matrix completion techniques have recently received significant attention in the data mining community. In particular, they have been successfully applied in

Received Feb 15, 2013
Revised Jun 18, 2013
Accepted Jul 17, 2013

the context of collaborative filtering in recommender systems (Chen et al., 2012; Das et al., 2010; Hu et al., 2008; Koren et al., 2009; Mackey et al., 2011; Niu et al., 2011; Recht and Ré, 2013; Yu et al., 2012; Zhou et al., 2008). At its heart, matrix completion is a variant of low-rank matrix factorization in which the input matrix is only partially observed and the observations are potentially noisy. In the setting of recommender systems, matrix rows correspond to users or customers, columns to items such as movies or musical pieces, and entries to feedback provided by users for items (e.g., explicit feedback in the form of numerical ratings and time of rating, or implicit feedback such as page views). Matrix completion is an effective tool for analyzing such dyadic data in that it discovers and quantifies the interactions between users and items. The idea is to posit $r \geq 1$ features for each user and for each item, where the rank r is a small number, usually less than 100. The features are usually latent in that they do not have explicit semantic interpretations; feature values are learned from the revealed matrix entries. In the simplest case, which we focus on in this paper, each missing matrix entry is estimated as the inner product of feature vectors for the corresponding user and item. Thus a user is predicted to rate an item highly if features that are important to the customer (i.e., having a large absolute value) match with the features of the item (large value of equal sign). In general, the estimation formula can be a complex function of the features as well as of other data such as the time stamp of the rating, user bias, implicit feedback, and so on.

Large applications can involve matrices with millions of rows, millions of columns, and billions of entries. For example, Netflix—a company that offers movies for rental and streaming and employs low-rank matrix completion in their recommendation engine—gathered more than five billion ratings for more than 80k movies from its more than 20M customers (Amatriain and Basilico, 2012; Bennett and Lanning, 2007). Similarly, Yahoo Music! collected billions of user ratings for musical pieces (Dror et al., 2012). At such massive scales, algorithms for matrix completion must be parallelized to achieve reasonable performance (Das et al., 2007; Das et al., 2010; Liu et al., 2010; Mackey et al., 2011; Niu et al., 2011; Recht and Ré, 2013; Zhou et al., 2008; Yu et al., 2012).

Stochastic gradient descent (SGD) is an iterative optimization algorithm that has been shown, in a sequential setting, to be very effective for matrix completion (Koren et al., 2009). Unfortunately, the generic SGD algorithm is not embarrassingly parallel and hence cannot directly scale to large data. We can, however, exploit the special structure of the matrix-completion problem to obtain a "stratified" version of SGD (SSGD) that can be parallelized, allowing scalability to extremely large problems.

In this paper, we apply the SSGD approach in three different computational settings and propose three different algorithms:[1] CSGD for parallel processing on a single powerful machine with multiple shared-memory processors, DSGD++ for small clusters of shared-nothing commodity nodes, and DSGD-MR for shared-nothing MapReduce processing environments. In addition to SSGD-based algorithms, we experiment with an asynchronous version of SGD (ASGD) for shared-nothing environments. Our new algorithms are cache-friendly and designed to exploit thread-level parallelism, in-memory processing, and asynchronous communication. See Tables 3 and 4 for a summary of new and existing algorithms for matrix completion.

Our focus throughout is on *in-memory* algorithms. Such algorithms are applicable when the problem instance, i.e., the data and feature vectors, can fit in the main memory of a single machine (in the shared-memory setting) or in the aggregate memory of a small cluster of commodity nodes (in the shared-nothing setting). This in-memory

[1] Some of the material in this paper originally appeared in (Gemulla, Nijkamp, Haas and Sismanis, 2011) and (Teflioudi et al., 2012).

assumption holds for a wide range of real-world matrix completion problems. Consider, for example, an extremely large hypothetical problem instance in which the input matrix has 20M rows, 1M columns, and 1% of the entries are revealed. (By comparison, data in (Bennett and Lanning, 2007; Dror et al., 2012) imply that 0.3% of Netflix entries and 0.04% of Yahoo! Music entries are known, and the number of rows and columns for the Netflix problem is smaller than for the hypothetical problem.) If 100 features are used per row and per column (i.e., a rank-100 factorization) and each entry is a 64-bit number, then the total data and model size is approximately 1.5TB. High-end parallel machines routinely ship with multiple terabytes of RAM, and shared-nothing clusters can have even higher aggregate memory capacities. We also assume throughout (without loss of generality) that the number of rows exceeds the number of columns, so that algorithms in shared-nothing environments move mostly column-factor data between the nodes.

In the shared-nothing setting, there have been almost no studies of in-memory matrix completion algorithms based on programming models, such as MPI (MPI, 2013), that allow asynchronous communication among processors. Similarly, the possibilities for exploiting multithreading have not been investigated. In a multithreaded shared-nothing architecture, different processing nodes do not share memory, but threads at the same processing node can share memory. We therefore provide a complexity analysis of shared-nothing algorithms that illuminates the various performance trade-offs and provides guidance in applying the algorithms to specific problems. This analysis shows that, under our assumptions, the key performance-determining characteristics with respect to the input data are the rank of the factorization and the ratio of the number of revealed entries to the number of columns. The analysis also indicates that DSGD++ is the most effective algorithm in terms of scalability and memory consumption.

We also report the results of an extensive set of experiments on both real-word and synthetic datasets of varying sizes. These experiments both confirm our theoretical results and provide a comprehensive comparison of new and existing algorithms for matrix completion. In particular, we compare algorithms based on SGD both to each other and to algorithms based on alternating-minimization approaches, such as alternating least-squares and cyclic coordinate descent. With respect to this comparison, the key experimental results are:

- For the shared-memory scenario, CSGD improves the runtime over the best-performing alternative approach by up to 60%.
- On large datasets and a shared-nothing scenario, DSGD++ outperforms competing methods in terms of overall runtime, memory consumption, and scalability. For example, DSGD++ can solve a difficult matrix completion problem on a high-variance matrix with 10M rows, 1M columns, and 10B revealed entries in around 40 minutes on 16 compute nodes.

The remainder of this paper is organized as follows. Sec. 2 reviews the matrix completion problem. In Sec. 3, we review sequential SGD, and introduce our new parallel shared-nothing and shared-memory SGD algorithms. Sec. 4 surveys prior matrix completion algorithms that are based not on SGD, but on alternating-minimization ideas. Sec. 5 contains a comparative complexity analysis of shared-nothing algorithms for matrix completion. Our experimental findings are summarized in Sec. 6, and we conclude the paper in Sec. 7.

2. The Matrix Completion Problem

To gain understanding about applications of matrix completion, consider the "Netflix problem" (Bennett and Lanning, 2007) of recommending movies to customers. Netflix is a company that offers tens of thousands of movies for rental. The company has more than 20M customers, each of whom can provide feedback about their personal taste by rating movies with 1 to 5 stars. The feedback can be represented in matrix form, for example

$$\begin{array}{c} \\ \textit{Alice} \\ \textit{Bob} \\ \textit{Charlie} \end{array} \begin{array}{c} \begin{array}{ccc} \textit{Avatar} & \textit{The Matrix} & \textit{Up} \end{array} \\ \left(\begin{array}{ccc} ? & 4 & 2 \\ 3 & 2 & ? \\ 5 & ? & 3 \end{array} \right) \end{array}.$$

Each entry may contain additional data, e.g., the date of rating or other forms of feedback such as click history. The goal of the completion is to predict missing entries (denoted by "?"), so that entries with a high predicted rating can then be recommended to users for viewing. This matrix-completion approach to recommender systems has been successfully applied in practice; see (Koren et al., 2009) for an excellent discussion of the underlying intuition.

Table 1 summarizes the notation used throughout this paper. Denote by the *training set* $\Omega = \{\,\omega_1, \ldots, \omega_N\,\}$ the set of revealed entries in $m \times n$ input matrix $\boldsymbol{V}$, where $\omega_k = (i_k, j_k)$, $k \in [1, N]$, $i_k \in [1, m]$, and $j_k \in [1, n]$. In what follows, we assume without loss of generality that $m \geq n$. Let N_{i*} and N_{*j} denote the number of revealed entries in row i and column j, respectively. Finally, denote by $r \ll \min(m, n)$ a rank parameter. Our goal is to find an $m \times r$ row-factor matrix $\boldsymbol{L}$ and an $r \times n$ column-factor matrix $\boldsymbol{R}$ such that $\boldsymbol{V} \approx \boldsymbol{LR}$, i.e., we aim to approximate $\boldsymbol{V}$ by the low-rank matrix $\boldsymbol{LR}$. The approximation is governed by an application-dependent loss function $L(\boldsymbol{L}, \boldsymbol{R})$ that measures the difference between the revealed entries in $\boldsymbol{V}$ and the corresponding entries in $\boldsymbol{LR}$. (We suppress the dependence on $\boldsymbol{V}$ for brevity.) The matrix completion problem is to find the factor matrices that give rise to the smallest loss, i.e.,

$$(\boldsymbol{L}^*, \boldsymbol{R}^*) = \operatorname*{argmin}_{\boldsymbol{L}, \boldsymbol{R}} L(\boldsymbol{L}, \boldsymbol{R}). \tag{1}$$

The matrix $\boldsymbol{L}^*\boldsymbol{R}^*$ is a "completed version" of $\boldsymbol{V}$, and each unrevealed entry $\boldsymbol{V}_{ij}$ is predicted by $[\boldsymbol{L}^*\boldsymbol{R}^*]_{ij}$.

The loss function L may also incorporate user biases, implicit feedback, temporal effects, and confidence levels, as well as regularization terms to prevent over-fitting. The most basic loss is the squared loss $L_{\text{Sl}}(\boldsymbol{L}, \boldsymbol{R}) = \sum_{(i,j)\in\Omega} (\boldsymbol{V}_{ij} - [\boldsymbol{LR}]_{ij})^2$. Table 2 summarizes other popular loss functions. L_{L2} incorporates L2 regularization and is closely related to the problem of minimizing the nuclear norm of the reconstructed matrix (Recht and Ré, 2013). L_{L2w} incorporates weighted L2 regularization (Zhou et al., 2008), in which the amount of regularization depends on the number of revealed entries. This particular loss function was a key ingredient in the best performing solutions of both the Netflix competition and the 2011 KDD-Cup (Koren et al., 2009; Chen et al., 2012; Zhou et al., 2008).

Our formulation of the matrix completion problem is motivated by its application in data-mining settings where a fixed set of training data and a loss function are given, and the goal is to compute loss-minimizing factor matrices as efficiently as possible. Candes and Recht (2009) discuss the theoretical foundations of the basic minimization problem. There is also a large body of literature that assumes a "true" underlying $\boldsymbol{V}$ matrix together with a stochastic process that generates from $\boldsymbol{V}$ the observed training data. The goal is then to statistically infer the true $\boldsymbol{V}$ matrix from the training data, where

Table 1. Notation

Symbol	Description
$\boldsymbol{V}$	Data matrix
m, n	Number of rows & columns of $\boldsymbol{V}$
Ω	Set of revealed entries in $\boldsymbol{V}$
N	Number of revealed entries in $\boldsymbol{V}$
N_{i*}	Number of revealed entries in row i of $\boldsymbol{V}$
N_{*j}	Number of revealed entries in column j of $\boldsymbol{V}$
r	Rank of the factorization
$\boldsymbol{L}, \boldsymbol{R}$	Factor matrices
$\boldsymbol{E}$	Residual matrix
w	Number of compute nodes
t	Number of threads per compute node
p	Total number of threads
b	Number of row/column blocks (SSGD)
T	Repetition parameter (CCD++)
s	Number of shufflers (Jellyfish)

Table 2. Popular loss functions for matrix completion

Loss	Definition	Local loss
L_{Sl}	$\sum_{(i,j)\in\Omega}(\boldsymbol{V}_{ij} - [\boldsymbol{LR}]_{ij})^2$	$(\boldsymbol{V}_{ij} - [\boldsymbol{LR}]_{ij})^2$
L_{L2}	$L_{\text{Sl}} + \lambda\Big(\sum_{ik} \boldsymbol{L}_{ik}^2 + \sum_{kj} \boldsymbol{R}_{kj}^2\Big)$	$(\boldsymbol{V}_{ij} - [\boldsymbol{LR}]_{ij})^2 + \lambda \sum_k (N_{i*}^{-1}\boldsymbol{L}_{ik}^2 + N_{*j}^{-1}\boldsymbol{R}_{kj}^2)$
L_{L2w}	$L_{\text{Sl}} + \lambda\Big(\sum_{ik} N_{i*}\boldsymbol{L}_{ik}^2 + \sum_{kj} N_{*j}\boldsymbol{R}_{kj}^2\Big)$	$(\boldsymbol{V}_{ij} - [\boldsymbol{LR}]_{ij})^2 + \lambda \sum_k (\boldsymbol{L}_{ik}^2 + \boldsymbol{R}_{kj}^2)$

the inference algorithm may exploit knowledge about the stochastic process. In one such model, the entries to be revealed are selected randomly and uniformly from the set of all $\boldsymbol{V}$ entries. Many algorithms and supporting theory have been developed for this specific setting; see, e.g., (Mackey et al., 2011).

In this paper, we focus on loss functions that admit a summation form; we say, following (Chu et al., 2006), that a loss function is in *summation form* if it is written as a sum of *local losses* L_{ij} that occur at only the revealed entries of $\boldsymbol{V}$, i.e., $L(\boldsymbol{L}, \boldsymbol{R}) = \sum_{(i,j)\in\Omega} L_{ij}(\boldsymbol{L}_{i*}, \boldsymbol{R}_{*j})$, where $\boldsymbol{L}_{i*}$ and $\boldsymbol{R}_{*j}$ refer to the i-th row of $\boldsymbol{L}$ and j-th column of $\boldsymbol{R}$, respectively. Table 2 shows examples of loss functions in summation form together with the corresponding local losses. We refer to the gradient of a local loss as a *local gradient*; by the linearity of the differentiation operator, the gradient of a loss function having summation form can be represented as a sum of local gradients: $L'(\boldsymbol{L}, \boldsymbol{R}) = \sum_{(i,j)\in\Omega} L'_{ij}(\boldsymbol{L}_{i*}, \boldsymbol{R}_{*j})$.

3. Stochastic Gradient Descent Algorithms

In the following sections, we present shared-nothing and shared-memory algorithms based on stochastic gradient descent; see Table 1 for an overview of our notation.

We first describe the basic SGD algorithm and then discuss various parallelization strategies. For brevity, we write $L(\theta)$ and $L'(\theta)$, where $\theta = (\boldsymbol{L}, \boldsymbol{R})$, to denote the loss function and its gradient. Denote by $\nabla_{\boldsymbol{L}} L$ (resp. $\nabla_{\boldsymbol{R}} L$) the $m \times r$ (resp. $r \times n$) matrix of the partial derivatives of L w.r.t. to the entries in $\boldsymbol{L}$ (resp. $\boldsymbol{R}$). Then $L' = (\nabla_{\boldsymbol{L}} L, \nabla_{\boldsymbol{R}} L)$.

For example, $[\nabla_{\boldsymbol{L}} L_{\mathrm{Sl}}]_{ik} = -2 \sum_{j:(i,j)\in\Omega_{i*}} \boldsymbol{R}_{kj}(\boldsymbol{V}_{ij} - [\boldsymbol{LR}]_{ij})$, where Ω_{i*} denotes the set of revealed entries in row $\boldsymbol{V}_{i*}$.

3.1. Stochastic Gradient Descent (SGD)

Various gradient-based methods have been explored in the context of matrix completion. Perhaps the simplest algorithm is gradient descent (GD), which iteratively takes small steps in the direction of the negative gradient:

$$\theta_{n+1} = \theta_n - \epsilon_n L'(\theta),$$

where n denotes the step number and $\{ \epsilon_n \}$ is a sequence of decreasing step sizes. Under appropriate conditions, GD has a linear rate of convergence; better rates can be obtained by using a quasi-Newton method, such as L-BFGS-B (Byrd et al., 1995).

Stochastic gradient descent (SGD) is based on GD, but uses a noisy estimate $\hat{L}'(\theta)$ of the gradient $L'(\theta)$. SGD iterates the stochastic difference equation

$$\theta_{n+1} = \theta_n - \epsilon_n \hat{L}'(\theta), \tag{2}$$

The gradient estimate is obtained by scaling up just one of the local gradients, i.e., $\hat{L}'(\theta) = NL'_{ij}(\theta)$ for some $(i,j) \in \Omega$. The choice of training point (i,j) varies from step to step according to a *training point schedule*; see below. Note that the local gradients at point (i,j) depend only on $\boldsymbol{V}_{ij}$, $\boldsymbol{L}_{i*}$ and $\boldsymbol{R}_{*j}$. Therefore, only a single row $\boldsymbol{L}_{i*}$ and a single column $\boldsymbol{R}_{*j}$ are updated during each SGD step.

Thus for each pass over the training data, SGD performs many quick-and-dirty steps whereas gradient descent (or a quasi-Newton method such as L-BFGS) performs a single careful step. For large matrices, the increased number of SGD steps leads to much faster convergence (Gemulla, Nijkamp, Haas and Sismanis, 2011). Moreover, the noisy estimation of the descent direction helps keep the algorithm from getting stuck at a local minimum, especially during the early stages of the descent. Further details of SGD are as follows.

Step size sequence. All of our SGD implementations use a simple adaptive method for selecting the step size that has worked extremely well in our experiments, even though guarantees of asymptotic convergence[2] have not been formally established. We refer to one GD step or a sequence of N SGD steps as an *epoch*; an epoch roughly corresponds to a single pass over the data. Exploiting the fact that the current loss can be computed after every epoch, we employ a heuristic called *bold driver* (Battiti, 1989). Bold driver starts from an initial step size ϵ_0. After each epoch, the algorithm increases the step size by a small percentage (5%) if the loss has decreased during the epoch, and drastically decreases the step size (by 50%) if the loss has increased. Within each epoch, the step size remains fixed. The initial step size ϵ_0 is obtained by trying different step sizes on a small sample (say, 0.1%) of Ω and picking the one that works best.

Training point schedule. Common schedules for an SGD epoch are: process Ω sequentially in some fixed order (SEQ), sample with replacement from Ω (WR), and sample without replacement from Ω (WOR). In practice, WOR often outperforms WR in terms of number of epochs to convergence; SEQ requires even more epochs than WR and

[2] "Convergence" refers to running an algorithm until some convergence criterion is met; "asymptotic convergence" means that the algorithm converges to the true solution as the runtime increases to $+\infty$.

```
**Require:** Incomplete matrix $\boldsymbol{V}$, initial values $\boldsymbol{L}$ and $\boldsymbol{R}$
  **while** not converged **do**  /* epoch */
    Create random permutation $\Pi$ of $\{ 1, \ldots, N \}$  /* WOR schedule */
    **for** $n = 1, 2, \ldots, N$ **do**  /* step */
      Prefetch indexes $(i^{\Pi}_{n+2}, j^{\Pi}_{n+2}) \in \Omega$ for next but one step
      Prefetch data $\boldsymbol{V}_{i^{\Pi}_{n+1} j^{\Pi}_{n+1}}$, $\boldsymbol{L}_{i^{\Pi}_{n+1} *}$, $\boldsymbol{R}_{* j^{\Pi}_{n+1}}$ for next step
      $\boldsymbol{L}'_{i^{\Pi}_n *} \leftarrow \boldsymbol{L}_{i^{\Pi}_n *} - \epsilon_n N \nabla_{\boldsymbol{L}_{i^{\Pi}_n *}} L_{i^{\Pi}_n j^{\Pi}_n}(\boldsymbol{L}, \boldsymbol{R})$
      $\boldsymbol{R}_{* j^{\Pi}_n} \leftarrow \boldsymbol{R}_{* j^{\Pi}_n} - \epsilon_n N \nabla_{\boldsymbol{R}_{* j^{\Pi}_n}} L_{i^{\Pi}_n j^{\Pi}_n}(\boldsymbol{L}, \boldsymbol{R})$
      $\boldsymbol{L}_{i^{\Pi}_n *} \leftarrow \boldsymbol{L}'_{i^{\Pi}_n *}$
    **end for**
  **end while**
```

Fig. 1. The SGD++ algorithm for matrix completion

converges to an inferior solution. Nevertheless, SEQ epochs are significantly faster than WR or WOR epochs, because they have better memory locality. To reduce this performance gap, we make use of the WOR schedule but employ latency-hiding techniques. Specifically, we prefetch the required data into the CPU cache before it is accessed by the SGD algorithm (e.g., using gcc's `__builtin_prefetch` macro). In the beginning of each epoch, we precompute and store a permutation Π of $\{ 1, \ldots, N \}$ that indicates the order in which training points are to be processed. In the n-th step, the SGD algorithm accesses the values $\boldsymbol{V}_{i^{\Pi}_n j^{\Pi}_n}$, $\boldsymbol{L}_{i^{\Pi}_n *}$, and $\boldsymbol{R}_{* j^{\Pi}_n}$, whose common index value (i^{Π}_n, j^{Π}_n) is determined from the $\Pi(n)$-th entry of Ω. We access Π and then prefetch the index value (i^{Π}_n, j^{Π}_n) during SGD step $n-2$ (so that it is in the CPU cache at step $n-1$), and the values $\boldsymbol{V}_{i^{\Pi}_n j^{\Pi}_n}$, $\boldsymbol{L}_{i^{\Pi}_n *}$, and $\boldsymbol{R}_{* j^{\Pi}_n}$ in SGD step $n-1$ (so that they are in the CPU cache at step n). Note that Π itself is accessed sequentially so that no explicit prefetching is needed. We refer to SGD with prefetching as SGD++; see Fig. 1. In our experiments, SGD++ was up to 15% faster than SGD (Teflioudi et al., 2012). We discuss alternative, cache-conscious approaches in Sec. 3.3.2.

3.2. SGD Methods for Shared-Nothing Environments

Even on only moderately large matrices, sequential methods may need an unacceptably large amount of time to converge, so that parallel versions of SGD are needed for scalability to massive data. The key challenge is that the SGD steps of Eq. (2) are not independent. In particular, steps that process training points that lie on the same row i (column j) both read and modify the corresponding row $\boldsymbol{L}_{i*}$ (column $\boldsymbol{R}_{*j}$) of the factor matrix.

In this section, we describe SGD algorithms for shared-nothing parallel processing environments. Such algorithms are designed for large-scale completion problems in which the input and factor matrices do not fit into main memory of a single node. We defer the discussion of parallel shared-memory algorithms, which target problems of intermediate sizes, to Sec. 3.3. Denote by w the number of compute nodes and by t the number of threads per node; the total number of threads is thus given by $p = wt$. We assume throughout that t is no larger than the number of (physical or virtual) cores available at each compute node. The main challenge in a shared-nothing environment is to effectively manage the communication between the compute nodes.

Ideally, we would like to parallelize the SGD algorithm by partitioning the data and factors across the cluster such that (1) the nodes work on mutually disjoint pieces of $\boldsymbol{L}$ and of $\boldsymbol{R}$ so that they can run independently and (2) each node processes roughly the same amount of data so that the workload is balanced. In general, however, these goals are not achievable simultaneously. To see this, assume to the contrary that there exists such a partitioning and that some node k is responsible for training points $\Omega_k \subseteq \Omega$. Further suppose that $(i,j) \in \Omega_k$. Since SGD uses the local gradients L'_{ij} as estimates of the gradient L', an SGD step on (i,j) will update $\boldsymbol{L}_{i*}$ and $\boldsymbol{R}_{*j}$. Since by assumption these parameters are not updated by any other node, all of the training points in row i and column j must also be in Ω_k, i.e., $(i,j) \in \Omega_k \implies \Omega_{i*} \cup \Omega_{*j} \subseteq \Omega_k$. Thus, Ω_k forms a submatrix of $\boldsymbol{V}$ that contains *all* revealed entries of any of its rows or columns. We can form w balanced partitions if and only if the rows and columns of $\boldsymbol{V}$ can be permuted to obtain a $w \times w$ block-diagonal matrix with a balanced number of revealed entries in each diagonal block. This is not possible in general; indeed, most (or even all) revealed entries usually concentrate in a single block.

In what follows, we discuss two different approaches to circumvent this problem: asynchronous SGD and stratified SGD. With the exception of the DSGD-MR algorithm discussed in Sec. 3.2.3, we assume that nodes can directly communicate with each other asynchronously, using a protocol such as MPI. All approaches partition the data and factor matrices into a carefully chosen set of blocks; we refer to such a partitioning as a *blocking*.

3.2.1. Asynchronous SGD (ASGD)

Suppose that $\boldsymbol{V}$ is blocked $w \times 1$, $\boldsymbol{L}$ is blocked conformingly $w \times 1$, and $\boldsymbol{R}$ is blocked $1 \times w$. At each node k, we store blocks $\boldsymbol{V}^{k*}$, $\boldsymbol{L}^k$, and $\boldsymbol{R}^k$. Note that this particular blocking ensures that each update of $\boldsymbol{L}$ is node-local, whereas updates of $\boldsymbol{R}$ are either local or remote. In the following, we refer to $\boldsymbol{R}_{*j}$ as the *master copy* of column j; the node that stores $\boldsymbol{R}_{*j}$ is referred to as *master node*. A naive way to parallelize SGD is as follows: When node k processes training point (i,j), it locks column j at its master node, fetches $\boldsymbol{R}_{*j}$, updates $\boldsymbol{L}_{i*}$ and $\boldsymbol{R}_{*j}$ locally according to (2), sends the new value of $\boldsymbol{R}_{*j}$ back to the master, and unlocks. This *synchronous algorithm* is clearly impractical, because the SGD steps of (2) are inexpensive so that most of the time is spent in communicating columns of $\boldsymbol{R}$.

Our asynchronous SGD algorithm (ASGD) avoids this problem by storing a *working copy* $\hat{\boldsymbol{R}}^k_{*j}$ of column $\boldsymbol{R}_{*j}$ at each node k. Initially, all the working copies agree with their corresponding masters. We now run SGD on each node as above, but update the working copy $\hat{\boldsymbol{R}}^k_{*j}$ instead of the master when processing (i,j); this avoids synchronous communication. However, the working copies still need to be coordinated to ensure correctness. In the context of perceptron training, McDonald et al. (McDonald et al., 2010) proposed averaging the working copies once after every epoch. In our setting, however, nodes can communicate continuously, which allows us to improve on this approach by also averaging during each epoch: From time to time, each node sends its *update vector* $\Delta\hat{\boldsymbol{R}}^k_{*j}$ to the master, where $\Delta\hat{\boldsymbol{R}}^k_{*j}$ is given by the sum of updates to $\hat{\boldsymbol{R}}_{*j}$ since the last averaging. Whenever a master node has received all update vectors $\Delta\hat{\boldsymbol{R}}^1_{*j}, \ldots, \Delta\hat{\boldsymbol{R}}^w_{*j}$, it adds their average to the master copy and broadcasts the result. Each node k then updates its working copy and integrates all local changes that have been accumulated meanwhile. The memory layout of ASGD is shown in Fig. 4a in Sec. 4. In the figure, $\hat{\boldsymbol{R}}^k = \left(\hat{\boldsymbol{R}}^k_{*1} \mid \hat{\boldsymbol{R}}^k_{*2} \mid \cdots \mid \hat{\boldsymbol{R}}^k_{*n}\right)$ and similarly for $\Delta\hat{\boldsymbol{R}}^k$.

In contrast to SGD, updates to a column of $\hat{\boldsymbol{R}}_{*j}$ at some node are not immediately seen by other nodes. When the delay between updating a column and broadcasting the update is bounded, asynchronous SGD converges to a stationary point of L (Tsitsiklis et al., 1986). In contrast to general asynchronous SGD, we average only a subset of the parameters, i.e., $\boldsymbol{R}$ but not $\boldsymbol{L}$; this idea is motivated by the distributed LDA method for text mining (Smola and Narayanamurthy, 2010). In our actual implementation, we send update vectors both continuously during and once after every epoch. This ensures that updates are communicated as often as possible and that the local copies agree with the master after every epoch. The latter property also allows us to apply the bold driver heuristic for step size selection. Moreover, we ensure that averaging is non-blocking. Finally, instead of running ordinary sequential SGD on each node, we run a multithreaded version of SGD called PSGD, which is described in Sec. 3.3.1 below. Besides the threads used for PSGD, an additional thread takes care of averaging. This latter thread has low CPU utilization since the time to compute the update vectors is swamped by communication costs.

3.2.2. Stratified SGD (SSGD)

An alternative approach to parallelizing SGD uses a stratification technique to avoid inconsistent updates; we refer to this general technique as stratified SGD (SSGD). SSGD provides the basis for our shared-nothing DSGD-MR and DSGD++ algorithms, as well as our shared-memory CSGD algorithm.

Recall the discussion at the beginning of Sec. 3.2, where we argued that parallelizing SGD is hard because, in general, $\boldsymbol{V}$ cannot be written as a $w \times w$ block-diagonal matrix with a balanced number of revealed entries in the diagonal blocks. The key idea behind stratified SGD is to induce parallelism-friendly blockings, called strata, by ignoring some of the entries. The algorithm moves from stratum to stratum in such a way that all training points are sampled correctly and the algorithm converges.

In more detail, suppose that $\boldsymbol{V}$ is blocked $b \times b$, where b is chosen to be greater than or equal to the number of available processing units (nodes or threads); for simplicity, we describe the algorithm for the case in which b exactly equals the number of processing units. The corresponding factor matrices are blocked conformingly:

$$\begin{array}{c} \\ \boldsymbol{L}^1 \\ \boldsymbol{L}^2 \\ \vdots \\ \boldsymbol{L}^b \end{array} \begin{array}{c} \begin{array}{cccc} \boldsymbol{R}^1 & \boldsymbol{R}^2 & \cdots & \boldsymbol{R}^b \end{array} \\ \begin{pmatrix} \boldsymbol{V}^{11} & \boldsymbol{V}^{12} & \cdots & \boldsymbol{V}^{1b} \\ \boldsymbol{V}^{21} & \boldsymbol{V}^{22} & \cdots & \boldsymbol{V}^{2b} \\ \vdots & \vdots & \ddots & \vdots \\ \boldsymbol{V}^{b1} & \boldsymbol{V}^{b2} & \cdots & \boldsymbol{V}^{bb} \end{pmatrix} \end{array}.$$

Rows and columns are randomly shuffled prior to blocking, so that each block contains N/b^2 training points in expectation. Now observe that when SGD runs on some block $\boldsymbol{V}^{ij}$, it accesses only the matrices $\boldsymbol{L}^i$ and $\boldsymbol{R}^j$. Thus SGD can, for example, be run independently and in parallel on each of the blocks on the main diagonal (i.e., $\boldsymbol{V}^{11}, \ldots, \boldsymbol{V}^{bb}$); the SGD instances refer to disjoint parts of the factor matrices and will hence yield the same result as processing the main diagonal using sequential SGD. In general, we say that two different blocks $\boldsymbol{V}^{ij}$ and $\boldsymbol{V}^{i'j'}$ are *interchangeable* whenever $i \neq i'$ and $j \neq j'$, i.e., they share neither rows nor columns. We call a set of b pairwise interchangeable blocks a *stratum*, the set of all strata is denoted by $\mathscr{S}$; see Fig. 3 for an example. By the arguments above, the blocks of a stratum can be processed independently and in parallel since they do not share any common rows or columns.

```
**Require:** Incomplete matrix $\boldsymbol{V}$, initial values $\boldsymbol{L}$ and $\boldsymbol{R}$, blocking parameter $b$
  Block $\boldsymbol{V}$ / $\boldsymbol{L}$ / $\boldsymbol{R}$ into $b \times b$ / $b \times 1$ / $1 \times b$ blocks
  while not converged do   /* epoch */
    Pick step size $\epsilon$
    for $s = 1, \ldots, b$ do   /* subepoch */
      Pick $w$ blocks $\{\boldsymbol{V}^{1j_1}, \ldots, \boldsymbol{V}^{bj_b}\}$ to form a stratum
      for $k = 1, \ldots, b$ do   /* in parallel */
        Run SGD on the training points in $\boldsymbol{V}^{kj_k}$ with step size $\epsilon$
      end for
    end for
  end while
```

Fig. 2. The generic SSGD algorithm for matrix completion

It is convenient to view a stratum as a bijective map from a row-block index k to a column-block index $j = S(k)$; here k corresponds to a processing unit such as a node or thread. For example, $S_B(1) = 2$, $S_B(2) = 3$, and $S_B(3) = 1$ in the example of Fig. 3. The complete SSGD algorithm is outlined in Fig. 2. SSGD repeatedly selects and processes a stratum $S \in \mathscr{S}$; the selection is based on a *stratum schedule* (see below). Stratum S is processed in parallel: processing unit k processes block $\boldsymbol{V}^{kS(k)}$. Continuing the example with $S = S_B$, unit 1 processes block $\boldsymbol{V}^{12}$, unit 2 processes block $\boldsymbol{V}^{23}$, and unit 3 processes block $\boldsymbol{V}^{31}$. In what follows, we refer to the processing of a single stratum as a *subepoch* and to a sequence of b subepochs as an *epoch*.[3] Note that an epoch roughly corresponds to processing N training points: each block contains N/b^2 entries in expectation, we process b blocks per subepoch, and there are b subepochs per epoch.

Stratum schedule. Just as the training point schedule of SGD influences its convergence in practice, the stratum schedule influences the convergence properties of SSGD. Formally, a stratum schedule is a (possibly random) sequence $S_1, S_2, \ldots$ of strata from $\mathscr{S}$; SSGD processes stratum S_l in the l-th subepoch. It can be shown that SSGD asymptotically converges to a stationary point of L under some natural conditions on the stratum schedule (Gemulla, Nijkamp, Haas and Sismanis, 2011; Gemulla, Haas, Nijkamp and Sismanis, 2011). For example, a schedule must satisfy the property that every training point is processed equally often in the long run.

When processing blocks during a typical epoch of SSGD, the simplest correct schedule processes blocks in sequential order on each node (SEQ), i.e., we set $S_l(k) = 1 + (k + l - 2 \mod b)$. In the example above, this corresponds to using the stratum schedule (S_A, S_B, S_C) for every epoch. An alternative is to randomly select a stratum from $\mathscr{S}$ in each subepoch (WR); e.g., the schedule (S_A, S_E, S_A) might be selected for a given epoch. Finally, we may select strata randomly but ensure that every block is processed exactly once per epoch (WOR); e.g., possible schedules for an epoch include all permutations of (S_A, S_B, S_C) and all permutations of (S_D, S_E, S_F); at each epoch, one of these 12 schedules is selected at random. In our experiments, we found that WOR achieves the best results because it randomizes the order of blocks as much as possible while ensuring that every training point is processed in every epoch.

[3] It is sometimes desirable to choose a value of b exceeding the number of processing units p. In this case, each stratum consists of p ($< b$) interchangeable blocks, so that p blocks are processed per subepoch and an epoch comprises b^2/p subepochs.

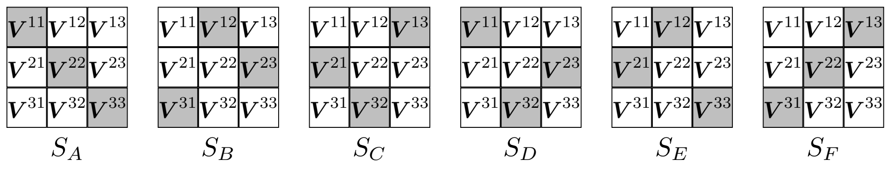

V^{11} V^{12} V^{13}
V^{21} V^{22} V^{23}
V^{31} V^{32} V^{33}
S_A
S_B
S_C
S_D
S_E
S_F

Fig. 3. Strata used by SSGD for a 3×3 blocking of $\boldsymbol{V}$

Specific algorithms for different settings can be obtained by specializing the generic SSGD algorithm. In the following sections, we describe DSGD-MR, an algorithm tailored to the shared-nothing MapReduce framework, and DSGD++, an in-memory algorithm for shared-nothing architectures in which nodes or threads can communicate directly. In Sec. 3.3.2 we show how the SSGD idea can be applied in a shared-memory environment.

3.2.3. SGD on MapReduce (DSGD-MR)

In MapReduce, data is partitioned into physical files that are stored in a distributed file system and read into memory as needed. Processing is divided into a map phase and reduce phase; files are written out to disk after each processing step to guard against data loss if a node fails. The w compute nodes do not communicate directly, but rather indirectly through reading and writing of files.

DSGD-MR applies SSGD in the MapReduce environment. It makes use of a stratification based on a $w \times w$ blocking of $\boldsymbol{V}$ (i.e., $b = w$), where node k stores blocks $\boldsymbol{V}^{k1}$,...,$\boldsymbol{V}^{kw}$, $\boldsymbol{L}^k$, and $\boldsymbol{R}^k$. This layout is illustrated in Fig. 4b,[4] where $\boldsymbol{V}^{k*}$ refers to the k-th row of blocks of $\boldsymbol{V}$. To process a stratum, DSGD-MR launches a single map-only job consisting of b map tasks. When processing a stratum S_l, the k-th map task ($1 \leq k \leq b$) processes block $\boldsymbol{V}^{kS_l(k)}$. The required blocks $\boldsymbol{V}^{kS_l(k)}$ and $\boldsymbol{L}^k$ are node-local and read from disk, and block $\boldsymbol{R}^{S_l(k)}$ is fetched from node $S_l(k)$ before processing and stored back afterward. Thus only blocks of $\boldsymbol{R}$ are communicated by DSGD-MR.

3.2.4. DSGD++

The MapReduce environment is limited in the sense that nodes cannot communicate directly and data is stored on disk. The DSGD++ algorithm, which we discuss next, is an in-memory algorithm that can run on a small cluster of commodity nodes. It uses a novel data partitioning and stratum schedule, and improves on DSGD-MR by exploiting asynchronous communication, as well as direct memory access and multithreading.

Direct fetches. Denote by S_l the stratum used in the l-th subepoch and by $S_l^{-1}(j)$ the node that updates block $\boldsymbol{R}^j$ in subepoch l. When running subepoch l, DSGD++ communicates the blocks of $\boldsymbol{R}$ directly between the nodes, i.e., the algorithm avoids writing the result back to disk as in DSGD-MR. In more detail, node k fetches the block $\boldsymbol{R}^{S_l(k)}$ directly from node $S_{l-1}^{-1}(S_l(k))$, which processed this block in the previous subepoch. A similar approach, but in the context of the Spark cluster-computing framework, has

[4] To facilitate comparison with other algorithms, we assume in Fig. 4b that input data and factor matrices are stored in main memory instead of in a distributed file system. Such an approach is also followed in the DSGD-MR implementation used in our experimental study. See (Gemulla, Haas, Nijkamp and Sismanis, 2011) for details on a disk-based Hadoop implementation.

been explored in the Sparkler system (Li et al., 2013); this system focuses on issues orthogonal to those of this paper, such as cluster elasticity, fault-tolerance and ease of programming.

Overlapping subepochs. Node k starts processing block $\boldsymbol{V}^{kS_l(k)}$ as soon as $\boldsymbol{R}^{S_l(k)}$ has been received. This strategy allows for overlapping of subepochs. For example, assume that all nodes are working on a stratum S_A as in Fig. 3 and then nodes 1 and 2 finish their jobs, but node 3, which is slower, is still working on block $\boldsymbol{V}^{33}$. Rather than forcing both nodes 1 and 2 to wait for node 3 to finish, node 1 can immediately move to S_B and start working on block $\boldsymbol{V}^{12}$ (because node 2 is finished and can send $\boldsymbol{R}^2$ to node 1). Node 2 must wait until node 3 is finished before moving to S_B and starting to process block $\boldsymbol{V}^{23}$, since it cannot start processing until it receives $\boldsymbol{R}^3$ from node 3. Note that this protocol avoids inconsistent updates, e.g., on $\boldsymbol{R}^3$. In this way DSGD++ gracefully handles varying processing speeds across the nodes.

Asynchronous communication. Observe that DSGD-MR is separated into a communication phase (receive next block of $\boldsymbol{R}$) and a computation phase (process block of $\boldsymbol{V}$). DSGD++ overlays communication and computation based on a $w \times 2w$ blocking of $\boldsymbol{V}$ instead of a $w \times w$ blocking. In each epoch, DSGD++ conceptually partitions $\boldsymbol{V}$ (and conformingly $\boldsymbol{R}$) at random into two matrices $\boldsymbol{V}_{\text{red}}$ and $\boldsymbol{V}_{\text{blue}}$, each consisting of w of the $2w$ column blocks. The algorithm then alternates between running a subepoch on $\boldsymbol{V}_{\text{red}}$ and $\boldsymbol{V}_{\text{blue}}$. This approach ensures that the red and blue subepochs work on disjoint blocks of $\boldsymbol{R}$ (cf. Fig. 4c). This is exploited as follows: Suppose that some node k runs subepoch l (say, blue). Node k now simultaneously fetches the block of $\boldsymbol{R}$ needed in the $(l+1)$-st subepoch (red) from the node that processed it in the $(l-1)$-st subepoch (also red). Thus communication and computation are overlaid.

Multithreading. Given w compute nodes and t threads per node, DSGD++ exploits thread-level parallelism by using a $p \times 2p$ blocking of $\boldsymbol{V}$ (instead of $w \times 2w$), where $p = wt$. Each node then stores $2tp$ blocks of $\boldsymbol{V}$, t blocks of $\boldsymbol{L}$, and $2t$ blocks of $\boldsymbol{R}$, and processes t blocks of $\boldsymbol{V}$ in t parallel threads during a subepoch. In contrast to using multiple independent processes per node, multithreading allows us to share memory between the threads. In more detail, when the block required in subepoch l (say, blue) has been processed at the same node in subepoch $l-2$ (also blue), no communication cost is incurred (*local fetch*). Data only needs to be communicated if the block is stored on some other node (*remote fetch*).

Locality-aware scheduling. A consequence of the distinction between local and remote fetches is that different stratum schedules have different communication costs, depending on the (expected) number of local fetches. SEQ is significantly more communication-efficient than WR/WOR because, in every subepoch, only a single remote fetch occurs per node and the other $t-1$ fetches are all local. However, the increased randomization of WOR leads to better convergence properties. DSGD++ uses a locality-aware schedule (LA-WOR) that strikes a compromise between the efficiency of SEQ and the desirable randomness of WOR. The key idea is to apply the stratification idea twice: After $\boldsymbol{V}_{\text{red}}$ and $\boldsymbol{V}_{\text{blue}}$ have been determined at the start of an epoch, randomly group the tw column blocks of $\boldsymbol{V}_{\text{red}}$ (and independently $\boldsymbol{V}_{\text{blue}}$) into w groups. Similarly, group the tw row blocks of $\boldsymbol{V}_{\text{red}}$ by the node at which they are stored. We thus obtain a $w \times w$ *coarse-grained blocking* of $\boldsymbol{V}_{\text{red}}$. Each of the coarse-grained blocks is then further broken up into $t \times t$ *fine-grained blocks*. When $w = t = 2$, for example, suppose that $\boldsymbol{V}^{\text{red}}$ consists of $\boldsymbol{V}^{*1}, \dots, \boldsymbol{V}^{*4}$ and that row blocks 1 and 2 (resp., 3 and 4) of $\boldsymbol{V}$ are stored at node 1 (resp., node 2). Then one possible blocking is:

$$\begin{array}{r} \text{Node 1} \\ \text{Node 2} \end{array} \left(\begin{array}{cc} \begin{pmatrix} \boldsymbol{V}^{11} & \boldsymbol{V}^{14} \\ \boldsymbol{V}^{21} & \boldsymbol{V}^{24} \end{pmatrix} & \begin{pmatrix} \boldsymbol{V}^{12} & \boldsymbol{V}^{13} \\ \boldsymbol{V}^{22} & \boldsymbol{V}^{23} \end{pmatrix} \\ \begin{pmatrix} \boldsymbol{V}^{31} & \boldsymbol{V}^{34} \\ \boldsymbol{V}^{41} & \boldsymbol{V}^{44} \end{pmatrix} & \begin{pmatrix} \boldsymbol{V}^{32} & \boldsymbol{V}^{33} \\ \boldsymbol{V}^{42} & \boldsymbol{V}^{43} \end{pmatrix} \end{array} \right)$$

Here the column blocks $\boldsymbol{V}^{*1}$ and $\boldsymbol{V}^{*4}$ have been grouped together, as have $\boldsymbol{V}^{*2}$ and $\boldsymbol{V}^{*3}$. To process a red stratum (a blue stratum is processed similarly), we use a $w \times w$ WOR schedule over the coarse-grained blocking (to distribute across nodes), and another $t \times t$ WOR schedule selected independently for each coarse-grained block (to parallelize across threads). In this way, whenever a coarse-grained block is processed at a node, the corresponding fine-grained blocks of $\boldsymbol{R}$ have to be communicated only once. Continuing the example, we might obtain the following LA-WOR schedule for $\boldsymbol{V}_{\text{red}}$:

$$\begin{array}{r} \\ \text{Node 1, thread 1} \\ \text{Node 1, thread 2} \\ \text{Node 2, thread 1} \\ \text{Node 2, thread 2} \end{array} \begin{array}{c} \begin{array}{cccc} S_1 & S_3 & S_5 & S_7 \end{array} \\ \left(\begin{array}{cc|cc} \boldsymbol{V}^{11} & \boldsymbol{V}^{14} & \boldsymbol{V}^{13} & \boldsymbol{V}^{12} \\ \boldsymbol{V}^{24} & \boldsymbol{V}^{21} & \boldsymbol{V}^{22} & \boldsymbol{V}^{23} \\ \hline \boldsymbol{V}^{32} & \boldsymbol{V}^{33} & \boldsymbol{V}^{31} & \boldsymbol{V}^{34} \\ \boldsymbol{V}^{43} & \boldsymbol{V}^{42} & \boldsymbol{V}^{44} & \boldsymbol{V}^{41} \end{array} \right) \end{array},$$

where the column entries for a stratum S_i correspond to the blocks of $\boldsymbol{V}$ that comprise the stratum, and the strata $S_1, S_3, \dots, S_7$ are displayed in the order that they are processed during a given sequence of red subepochs (hence the odd-numbered stratum indexes). As before, we only need to communicate blocks of $\boldsymbol{R}$. Assume that in this example blocks $\boldsymbol{R}^1$ and $\boldsymbol{R}^2$ are initially stored on node 1, while blocks $\boldsymbol{R}^3$ and $\boldsymbol{R}^4$ are at node 2. Denote by thread(i, j) the jth thread at node i. Stratum S_1 is processed first. To process $\boldsymbol{V}^{11}$, thread$(1, 1)$ fetches $\boldsymbol{R}^1$, a local fetch, whereas thread$(1, 2)$ fetches $\boldsymbol{R}^4$ to process $\boldsymbol{V}^{24}$, a remote fetch. Similarly, thread$(2, 1)$ and thread$(2, 2)$ need to fetch $\boldsymbol{R}^2$ and $\boldsymbol{R}^3$, resulting in one remote and one local fetch. Next, S_3 is processed, which requires the fetching of $\boldsymbol{R}^4$ and $\boldsymbol{R}^1$ by thread$(1, 1)$ and thread$(1, 2)$ and of $\boldsymbol{R}^3$ and $\boldsymbol{R}^2$ by thread$(2, 1)$ and thread$(2, 2)$. Because these blocks of $\boldsymbol{R}$ were all fetched during the processing of S_1, they are now local, so that no remote fetches are required. Overall, the processing of S_1–S_7 incurs 10 local fetches and only 6 remote fetches.

3.3. SGD Methods for Shared-Memory Environments

We now turn to methods for shared-memory parallel SGD on a single, powerful compute node. Shared-memory algorithms target matrices of intermediate sizes, for which both data and factor matrices fit into the memory of single compute node. In this setting, our objectives are (1) to parallelize SGD across multiple threads, and (2) to make SGD cache-conscious so that memory accesses do not become a bottleneck. As before, denote by t the number of available threads.

3.3.1. Parallel SGD (PSGD)

As discussed previously, if two SGD steps process training points that lie in the same row i (resp., column j), then both of these steps read and update row $\boldsymbol{L}_{i*}$ (resp., column

$\boldsymbol{R}_{*j}$) of the factor matrix. Perhaps the simplest way to parallelize SGD is to partition the training point schedule evenly among the t threads, i.e., each thread runs N/t SGD steps per epoch. To avoid concurrent parameter updates, we lock row i of $\boldsymbol{L}$ and column j of $\boldsymbol{R}$ before processing training point (i, j). This lock-based approach works well when the number t of threads is small (say, $t \leq 8$), but both locking and random memory accesses impede scalability to large numbers of threads. Niu et al. (Niu et al., 2011) experimented with a lock-free algorithm, henceforth denoted PSGD, in which no locks are obtained so that inconsistent updates may occur. Since there are usually significantly more rows and columns than available threads (i.e., $m, n \gg t$), it is unlikely that a given row or column is processed by multiple threads at the same time; we thus expect few inconsistent updates. Niu et al. found virtually no difference between lock-based and lock-free parallel SGD in terms of running time and quality (for matrix completion problems). On the whole, our experiments validate these findings, except that we observed a small performance improvement (of up to 9%) of PSGD over the lock-based approach when using a large number of threads ($t \geq 16$).

3.3.2. Cache-Conscious Parallel SGD (CSGD and Jellyfish)

A key disadvantage of PSGD is that its memory access pattern is non-sequential. Recall from Sec. 3.1 that SGD processes training points in random order. When N is large, accesses to training points often result in cache misses, which hinders scalability to a large number of threads due to high consumption of memory bandwidth. This effect is especially pronounced in large machines based on non-uniform memory access designs (NUMA), where a cache miss can lead to expensive accesses to non-local memory. In this section, we propose a new cache-conscious parallel SGD algorithm, termed CSGD, which alleviates this problem by localizing memory accesses.

The main idea behind CSGD is to run (a variant of) SSGD with a suitably chosen, fine-grained blocking of the input matrix. In particular, the algorithm uses a blocking parameter $b \gg t$ such that each block $\boldsymbol{V}^{ij}$, as well as the corresponding factor matrices $\boldsymbol{L}^i$ and $\boldsymbol{R}^j$, fit into the L2 cache of a single core. The training points within a block are laid out in consecutive memory locations; if a block is processed using SGD, we expect few cache misses when accessing the data and factor matrices because of higher locality. To obtain a parallel algorithm, a direct application of the SSGD algorithm would select, in each subepoch, t interchangeable blocks, process each block on a separate thread using SGD, synchronize, and proceed to the next subepoch. Although such an approach is feasible and guarantees correctness, the large number of subepochs (b^2/t), and hence synchronization points, severely limit performance in practice. An alternative procedure is to simply omit synchronization, akin to the PSGD algorithm described above. In particular, we partition matrix $\boldsymbol{V}$ evenly across the t threads such that each row of blocks is assigned to exactly one core. This ensures that there will be no inconsistent updates on the row-factor matrix $\boldsymbol{L}$. Each thread then independently processes its blocks in WOR order; inconsistent updates on $\boldsymbol{R}$ may occur, but since $b \gg t$, we expect this to happen rarely.

An alternative approach to CSGD is the Jellyfish algorithm (Recht and Ré, 2013). As with SSGD, Jellyfish uses a $t \times t$ blocking of the input matrix. In contrast to SSGD and CSGD, Jellyfish changes the blocking of $\boldsymbol{V}$ from epoch to epoch by reshuffling the entire data set. Once the data is shuffled, it is accessed in a sequential fashion so that prefetching is very effective. To speed up data shuffling, Jellyfish maintains s copies of the data, where $s \geq 2$ is a small number. While one copy is being processed, $s - 1$ parallel shuffle threads reorganize the data in the remaining $s - 1$ copies. To process a copy, Jellyfish makes use of t parallel threads, avoiding fine-grained locking in a manner similar to SSGD. As compared to CSGD, Jellyfish's disadvantages are that (1) it is memory intensive because multiple copies of the input matrix need to be maintained and (2) parallel shuffling may also lead to memory bottlenecks.

CSGD can be seen as a combination of Jellyfish (which is cache conscious), lock-free SGD (which allows a few inconsistent updates), and SSGD (which uses stratification, i.e., a fixed blocking). Our experiments in Sec. 6 indicate that CSGD is currently the most scalable and best-performing matrix completion algorithm in the shared-memory setting (up to 60% faster than its closest competitor).

4. Matrix Completion via Alternating Minimizations

We now review some popular algorithms for large-scale matrix completion that are based on the idea of "alternating minimizations". See Tables 3 and 4 for a summary of the parallel algorithms considered in this paper.

Table 3. Overview of shared-nothing methods (see Table 1 for notation)

Method	**Partitioning**	**Memory consumption/node**	**Epochs/it.**	**Time/it.**	**Communication/node/it.**
DALS (Zhou et al., 2008)	$\boldsymbol{V}$ & $\boldsymbol{L}$ by rows $\boldsymbol{V}$ & $\boldsymbol{R}$ by columns	$2\boldsymbol{V}/w + \boldsymbol{L} + \boldsymbol{R}$	2	$O(p^{-1}[Nr^2 + (m+n)r^3])$	$O([m+n]r)$
DCCD++ (Yu et al., 2012)	$\boldsymbol{E}$ & $\boldsymbol{L}$ by rows $\boldsymbol{E}$ & $\boldsymbol{R}$ by columns	$(2\boldsymbol{E} + \boldsymbol{L} + \boldsymbol{R})/w$	$2r(T+1)$	$O(p^{-1}TNr)$	$O(T[m+n]r)$
ASGD (new)	$\boldsymbol{V}$ & $\boldsymbol{L}$ by rows	$(\boldsymbol{V} + \boldsymbol{L} + \boldsymbol{R})/w + 2\boldsymbol{R}$	1	$O(p^{-1}Nr + nr)^*$	$O(nr)^*$
DSGD-MR (new)	$\boldsymbol{V}$ blocked, $\boldsymbol{L}$ by row, $\boldsymbol{R}$ by column	$(\boldsymbol{V} + \boldsymbol{L} + 2\boldsymbol{R})/w$	1	$O(p^{-1}Nr)$	$O(nr)$
DSGD++ (new)	$\boldsymbol{V}$ blocked rect., $\boldsymbol{L}$ by row, $\boldsymbol{R}$ by column	$(\boldsymbol{V} + \boldsymbol{L} + 2.5\boldsymbol{R})/w$	1	$O(p^{-1}Nr)$	$O(nr)$

*Assuming asynchronous averaging is performed a constant number of times per iteration.

Table 4. Overview of shared-memory methods (see Table 1 for notation)

Method	Partitioning	Memory consumption	Epochs/iteration	Time/iteration
PALS (Zhou et al., 2008)	$\boldsymbol{V}$ & $\boldsymbol{L}$ by rows, $\boldsymbol{V}$ & $\boldsymbol{R}$ by columns	$2\boldsymbol{V} + \boldsymbol{L} + \boldsymbol{R}$	2	$O(t^{-1}[Nr^2 + (m+n)r^3])$
PCCD++ (Yu et al., 2012)	$\boldsymbol{E}$ & $\boldsymbol{L}$ by rows, $\boldsymbol{E}$ & $\boldsymbol{R}$ by columns	$2\boldsymbol{E} + \boldsymbol{L} + \boldsymbol{R}$	$2r(T+1)$	$O(t^{-1}TNr)$
PSGD (Niu et al., 2011)	$\boldsymbol{V}$ & $\boldsymbol{L}$ by rows	$\boldsymbol{V} + \boldsymbol{L} + \boldsymbol{R}$	1	$O(t^{-1}Nr)$
Jellyfish (Recht and Ré, 2013)	$\boldsymbol{V}$ blocked, $\boldsymbol{L}$ by rows, $\boldsymbol{R}$ by columns	$s\boldsymbol{V} + \boldsymbol{L} + \boldsymbol{R}$	1*	$O(t^{-1}Nr)^*$
CSGD (new)	$\boldsymbol{V}$ blocked, $\boldsymbol{L}$ by rows, $\boldsymbol{R}$ by columns	$\boldsymbol{V} + \boldsymbol{L} + \boldsymbol{R}$	1	$O(t^{-1}Nr)$

*Excluding data shuffling processes.

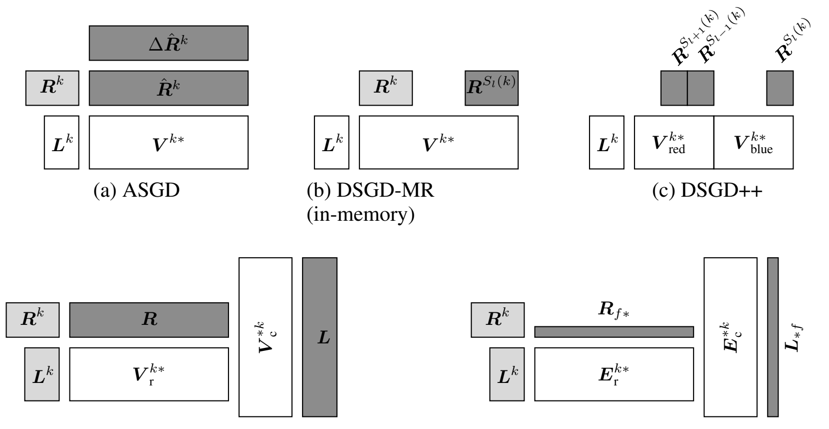


Fig. 4. Memory layout used on node k by the shared-nothing algorithms ($t = 1$). Node-local data is shown in white, master copies in light gray, and temporary data in dark gray.

4.1. Alternating Least Squares (ALS)

ALS alternates between optimizing for $\boldsymbol{L}$ given $\boldsymbol{R}$, and optimizing for $\boldsymbol{R}$ given $\boldsymbol{L}$. For the loss function L_{Sl}, this amounts to solving a set of least squares problems: one for each row of $\boldsymbol{L}$ and one for each column of $\boldsymbol{R}$. During the computation of least squares solutions, matrix $\boldsymbol{V}$ is accessed by row when updating $\boldsymbol{L}$, and by column when updating $\boldsymbol{R}$. For this reason, ALS implementations need to store two (sparse) representations of $\boldsymbol{V}$ in memory: one in row-major order (denoted $\boldsymbol{V}_{\text{r}}$) and one in column-major order (denoted $\boldsymbol{V}_{\text{c}}$). Loss functions L_{L2} and L_{L2w} can also be handled (Zhou et al., 2008). Under our running assumption that $m \geq n$, an ALS epoch has time complexity $O(Nr^2 + mr^3)$.

A shared-memory parallel algorithm for ALS, denoted PALS, is based on the observation that the foregoing least-squares problems can be solved independently (Zhou et al., 2008). E.g., an update to a row of $\boldsymbol{L}$ does not affect other rows of $\boldsymbol{L}$, so that the processing of rows of $\boldsymbol{L}$ can be partitioned evenly among the available threads. Processing of columns of $\boldsymbol{R}$ can similarly be partitioned among threads.

Following (Zhou et al., 2008), we obtain a shared-nothing variant of ALS, denoted DALS, as follows. Assume that each node has sufficient memory to store a fraction of $2/w$ of the entries of $\boldsymbol{V}$, as well as a full copy of the factor matrices $\boldsymbol{L}$ and $\boldsymbol{R}$. DALS uses a $w \times 1$ blocking for $\boldsymbol{V}_{\text{r}}$ and a $1 \times w$ blocking for $\boldsymbol{V}_{\text{c}}$; factor matrices are blocked conformingly. Node k stores blocks $\boldsymbol{V}^{k*}_{\text{r}}$, $\boldsymbol{V}^{*k}_{\text{c}}$, $\boldsymbol{L}^k$, and $\boldsymbol{R}^k$. This memory layout is illustrated in Fig. 4d. To update $\boldsymbol{L}^k$ (at node k), DALS requires access to $\boldsymbol{V}^{k*}_{\text{r}}$ and the *entire* $\boldsymbol{R}$ matrix. Note that $\boldsymbol{V}^{k*}_{\text{r}}$ and $\boldsymbol{L}^k$ are stored locally at node k, but $\boldsymbol{R}$ is not. Therefore, DALS creates a local copy of $\boldsymbol{R}$ on each node by broadcasting blocks $\boldsymbol{R}^1, \ldots \boldsymbol{R}^k$. DALS differs from the algorithm of (Zhou et al., 2008) only in that it uses multiple threads (which share the same memory space and variables) instead of multiple processes (each with its own address space) on each node, which allows DALS to reduce memory consumption.

4.2. Cyclic Coordinate Descent (CCD++)

Cyclic coordinate descent (CCD) can also be seen as an alternating minimization method: It optimizes a *single* entry of one of the factor matrices at a time while keeping all other entries fixed. This leads to a much simpler minimization problem than that of ALS. Practical variants of CCD adopt the approach of "hierarchical" ALS (Cichocki and Phan, 2009) in that they do not operate on the original input matrix but on the residual matrix $\boldsymbol{E}$ whose entries are $\boldsymbol{V}_{ij} - [\boldsymbol{L}\boldsymbol{R}]_{ij}$ for $(i, j) \in \Omega$.

Recently, a version of CCD, termed CCD++, has been proposed for matrix completion problems (Yu et al., 2012). Similar to ALS, CCD++ stores two copies of the residual matrix $\boldsymbol{E}$: one in row-major order (to update $\boldsymbol{L}$), denoted $\boldsymbol{E}_{\mathrm{r}}$, and one in column-major order (to update $\boldsymbol{R}$), denoted $\boldsymbol{E}_{\mathrm{c}}$. CCD++ employs a feature-wise sequence of updates, i.e., each iteration loops over all features $f \in [1, r]$. For each feature f, the algorithm executes T update operations, where T is an automatically tuned parameter that is independent of the data size, and each operation updates the fth feature-vector of $\boldsymbol{L}$ (i.e., $\boldsymbol{L}_{*f}$) and then the fth feature-vector of $\boldsymbol{R}$ (i.e., $\boldsymbol{R}_{f*}$). Finally, both copies of the residual matrix are updated and the algorithm continues with feature $f + 1$. For each feature, the residual matrix is scanned $2T$ times to update the corresponding feature vectors of $\boldsymbol{L}$ and $\boldsymbol{R}$, and twice to update $\boldsymbol{E}_{\mathrm{r}}$ and $\boldsymbol{E}_{\mathrm{c}}$. An iteration, i.e., the processing of all features, therefore consists of $2r(T + 1)$ epochs and has overall time complexity of $O(TNr)$. Overall, our experiments, as well as results in (Yu et al., 2012), indicate that CCD++ is computationally less expensive than ALS and can handle larger ranks efficiently.

Parallel versions of CCD++ (Yu et al., 2012) are based on ideas similar to those used for parallelizing ALS; in particular, they make use of a similar partitioning of the factor matrices. We refer to the shared-memory variant as PCCD++ and to the shared-nothing variant as DCCD++. In contrast to DALS, DCCD++ requires only one feature vector to be broadcast and stored in each epoch (instead of an entire factor matrix as in DALS); see Fig. 4e. Since communication is needed every time a feature vector is being updated, i.e., $2Tr$ times per iteration, synchronization costs between the nodes can potentially be substantial.

4.3. Alternating Minimizations Versus SGD

Before embarking on a comprehensive comparison of algorithms, we make the following general remarks concerning the alternating-minimizations and SGD approaches. Because ALS needs to solve a large number of linear least squares problems, it is generally much more expensive than SGD. This computational overhead is acceptable, however, when the rank of the factorization is sufficiently small (say, $r \leq 50$). An advantage of both ALS and CCD++ over SGD is that the former methods are parameter-free, whereas SGD methods make use of a step size sequence. Our experiments suggest, however, that SGD is the method of choice when the step size sequence is chosen judiciously, e.g., using the bold driver method of Sec. 3.1. Moreover, both ALS and CCD++ are more memory-intensive than SGD since they need to store the data matrix twice. Finally, SGD-based methods apply to a wide range of loss functions, whereas ALS and CCD++ target quadratic loss functions.

5. Comparison of Methods I: Complexity Analysis

In this section, we begin our comparison of the various matrix completion algorithms by providing a theoretical complexity analysis of shared-nothing methods. (We do not present a theoretical analysis of shared-memory algorithms here because the results are sensitive to assumptions about the memory architecture and cache behavior, which can vary widely.) Empirical results are presented in Sec. 6.

A key objective of this section is to identify conditions under which distributed processing is effective. To simplify the analysis, we assume that

1. computation and communication are not overlaid,
2. each revealed entry of $\boldsymbol{V}$ and each entry of $\boldsymbol{L}$ and $\boldsymbol{R}$ requires $O(1)$ words of memory and $O(1)$ time to communicate,
3. the number t of threads per node is constant,
4. $r, n \leq m$, and
5. $N = O(mr)$.

We give bounds on the memory consumption per node as well as computation and communication time per node and epoch.

Memory consumption per node. DALS stores in memory two partitions of the input matrix (one partition from $\boldsymbol{V}_r$ and one from $\boldsymbol{V}_c$) as well as both factor matrices; the total memory consumption is $O(N/w + mr)$ words. (DALS also stores a row-partition of $\boldsymbol{L}$ and a column-partition of $\boldsymbol{R}$, but these are negligible with respect to the other data.) DCCD++ stores two partitions of the residual matrix, two feature vectors (i.e., a row of $\boldsymbol{R}$ and a column of $\boldsymbol{L}$), a row-partition of $\boldsymbol{L}$, and a column-partition of $\boldsymbol{R}$. Therefore the total memory consumption is $O((N + mr)/w + m)$ words. ASGD stores, in addition to a partition of the input matrix and partitions of $\boldsymbol{L}$ and $\boldsymbol{R}$, the smaller factor matrix $\boldsymbol{R}$ in its entirety, for a total consumption of $O((N + mr)/w + nr)$ words. Finally, DSGD++ fully partitions the factor matrices and thus requires $O((N + mr)/w)$ words in total. We conclude that DSGD++ is most memory-efficient, followed by DCCD++ and ASGD, and then DALS.

Computation/communication trade-off. The overall performance of the shared-nothing methods crucially depends on the relationship between computation and communication cost. We say that a distributed-processing algorithm is *effective* if computation costs dominate communication costs, because the former costs are linearly reduced by distributed processing, whereas the latter costs are increased. Under our assumptions, DALS requires $O(mr^3/w)$ time for computation and $O(mr)$ time for communication per epoch. As $m, r, w \to \infty$, computation dominates communication if the rank of the factorization is sufficiently large: $r^2 = \omega(w)$. In practice, we often have $r > w$ so that we expect DALS to be effective. Similarly, for updating and communicating a single factor, DCCD++ has computation cost $O(mr/w)$ and communication cost $O(m)$. Thus DCCD++ is effective when $r = \omega(w)$, i.e., under tighter conditions than DALS.

For DSGD-MR, $O(Nr/w)$ time is required for computation and $O(nr)$ time for communication, since we only communicate $\boldsymbol{R}$. Denote by $\bar{N} = N/n$ the average number of revealed entries per column of $\boldsymbol{V}$; $\bar{N}$ measures the amount of work per column and is key to determining how well we can parallelize SGD. To see this, rewrite the computational cost as $O(\bar{N}nr/w)$ and observe that computation dominates communication only for large values of $\bar{N}$, i.e., $\bar{N} = \omega(w)$. Thus, we expect DSGD-MR to be effective when the data matrix is not too sparse or has few columns. The same conclusions hold for DSGD++ and ASGD, even though they do not satisfy Assumption 1 above. Indeed, the analysis for DSGD-MR carries over to DSGD++ directly and to ASGD under the additional assumption that working copies are averaged at least once per epoch.

Theoretical analysis and actual performance. To what extent can our asymptotic complexity analysis predict the actual relative performance of the algorithms? There are several causes for concern. One issue is that the data size and number of nodes are in fact finite. Even when the data was small enough to fit on a single machine, however, a number of conclusions from our theoretical analysis were confirmed by our experiments. For example, DALS significantly outperformed PALS for problems of a given size (confirming that DALS is "effective" as defined above), and the relative time for a DSGD-MR or ASGD epoch compared to that of a PSGD epoch depends on $\bar{N}$. Our theoretical analysis diverged somewhat from our empirical results in the case of DSGD++. This latter algorithm proved effective even for small values of $\bar{N}$ because—unlike the assumption that we used for the complexity analysis—computation and communication are in fact overlaid.

Another potential inaccuracy in our complexity analysis is that distribution of processing among nodes may affect the number of epochs required by the algorithms to converge. This is not an issue for the algorithms based on alternating minimizations: DALS and ALS, as well as DCCD++ and CCD++, perform identically, since they solve the exact same least-squares problems. In contrast, the DSGD-MR, and DSGD++, and ASGD algorithms may need more epochs to converge than PSGD. The reasons are that the stratification used in DSGD-MR and DSGD++ reduces the randomness in training point selection, and the asynchronous averaging used in ASGD introduces delay in broadcasting updates. Therefore, the processing time per epoch is not by itself a reliable indicator of overall performance; convergence-rate effects must be taken into account as well. The experiments in the following section give further insight into the performance of the algorithms.

6. Comparison of Methods II: Experimental Study

We conducted an experimental study and compared all algorithms along the following dimensions: the time per epoch (excluding loss computation), the number of epochs required to converge, and the total time to converge (including loss computations). Recall that an epoch corresponds roughly to a single pass over the input data, so that the number of epochs reflects the number of data scans. When comparing two algorithms A and B in an experiment, we say that A is more *compute-efficient* than B if it needs less time per epoch, more *data-efficient* if needs fewer epochs to converge, and *faster* if it needs less total time.

6.1. Overview of Results

In the shared-memory setting, CSGD outperformed all alternative methods on both real and synthetic datasets: It was up to 15.6x faster than PALS, up to 2.5x faster than PSGD, and up to 5.7x faster than PCCD++. CSGD also showed better scalability than PSGD, Jellyfish, and PCCD++ in that we could use more parallel threads before hitting the memory bandwidth. PALS was the most data-efficient but the least compute-efficient method, whereas PCCD++ was least data-efficient but most compute-efficient. The SGD-based approaches lay in between.

Table 5. Summary of datasets

	m	n	N	$\bar{N}$	Size	L	λ
Netflix	480k	18k	99M	5.5k	2.2GB	L_{L2w}	0.05
KDD	1M	625k	253M	0.4k	5.6GB	L_{L2w}	1
Syn1B-rect	10M	1M	1B	1k	22.3GB	L_{Sl}	-
Syn1B-sq	3.4M	3M	1B	0.3k	22.3GB	L_{Sl}	-
Syn10B	10M	1M	10B	10k	223.5GB	L_{Sl}	-

In the shared-nothing setting, DSGD++ was the best-performing method on our large-scale experiments in all configurations. It was up to 12.8x faster than DALS, up to 5x times faster than DSGD-MR, and up to 12.3x faster than ASGD. Indeed, DSGD++ even outperformed PSGD, by a large margin, in some of our experiments with moderately-sized data for which both algorithms were applicable. DSGD++ was the only shared-nothing method to outperform PSGD, which testifies to its high communication efficiency. ASGD was faster than DSGD++ in two experiments with few nodes and large $\bar{N}$, but its performance degraded significantly as more nodes were added. Thus ASGD, and to a lesser extent also DSGD-MR, was much more sensitive to communication overhead than DSGD++ and DALS. Finally, as with PALS, the DALS algorithm was the most data-efficient but the least compute-efficient method. In terms of total time, it was competitive only when the rank r was small.

6.2. Experimental Setup

Implementation. We implemented SGD, SGD++, ALS, PSGD, PALS, CSGD, DSGD-MR, DSGD++, ASGD, and DALS in C++. For CCD++ and PCCD++, we used the C++ implementation provided by the authors of (Yu et al., 2012); the code for DCCD++ was not available to us. For Jellyfish, we used the C++ implementation of (Recht and Ré, 2013), but incorporated the bold driver heuristic for step size selection to ensure a fair comparison. In all our experiments, we used $s = 3$ shuffle threads (we did not see significant differences for other choices of s). For CSGD, we chose the blocking parameter b such that one partition of the data and the corresponding factors barely fit into the cache available for one core. For communication, all shared-nothing algorithms used the MPICH2 implementation of MPI. We used the GNU scientific library for solving the least-squares problems of ALS; in our experiments, GSL was significantly faster than LAPACK and, in contrast to LAPACK, also supports multithreading.

Hardware. We used two different hardware configurations in our experiments: a compute cluster comprising 16 nodes for the shared-nothing setting and a single powerful high-memory server for the shared-memory setting. Each node in the compute cluster had 48GB of main memory and an Intel Xeon 2.40GHz processor with 8 cores. The high-memory server had 512GB of main memory and was equipped with 4 Intel Xeon 2.40GHz processors with 10 cores each (40 in total).

Real-world datasets. We used two real-world datasets: Netflix and KDD. The Netflix dataset, which occupies 2.2GB of main memory, consists of roughly 99M ratings of 480k Netflix users for 18k movies; rating values range from 1 to 5. The KDD dataset, which occupies 5.5 GB of main memory, corresponds to that of Track 1 of the 2011 KDD Cup 2011 and consists of approximately 253M ratings of 1M Yahoo! Music users for 625k musical pieces. Netflix and KDD differ significantly in the value of $\bar{N}$ (large for Netflix, small for KDD). Detailed statistics for these datasets, as well as for the synthetic datasets described below, are summarized in Table 5. For both real-world datasets, we used the official validation sets and focused on L_{L2w} because it performs best in practice (Zhou et al., 2008; Koren et al., 2009; Chen et al., 2012). We did not tune the regularization parameter for varying choices of rank r but used the values given in Table 5 throughout.

Synthetic datasets. For our large-scale experiments, we generated three synthetic datasets that differ in the choice of m, n, and N. We generated each dataset by first creating two rank-50 matrices $\boldsymbol{L}^*_{m\times 50}$ and $\boldsymbol{R}^*_{50\times n}$ with entries sampled independently from the Normal$(0, 10)$ distribution. We then obtained the data matrix by sampling N random entries from $\boldsymbol{L}^*\boldsymbol{R}^*$ and adding Normal$(0, 1)$ noise. Note that the resulting datasets are very structured. We use them here to test the scalability of the various algorithms; the matrices can potentially be factored much more efficiently by exploiting their structure directly. To judge the impact of the shape of the data matrix, we generated two large datasets with 1B revealed entries and identical sparsity: Syn1B-rect is a tall rectangular matrix (high $\bar{N}$, easier to distribute), Syn1B-sq is a square matrix (low $\bar{N}$, harder to distribute). Note that we need to learn more parameters to complete Syn1B-rect (550M) than to complete Syn1B-sq (320M). We also generated a very large dataset with 10B entries (Syn10B) to explore the scalability of each method; Syn10B is significantly larger than the main memory of each individual machine.

Methodology. For all datasets, we centered the input matrix around its mean. To investigate the impact of the factorization rank, we experimented with ranks $r = 50$ and $r = 100$; in practice, values of up to $r = 1000$ can be beneficial (Zhou et al., 2008). The starting points $\boldsymbol{L}_0$ and $\boldsymbol{R}_0$ were chosen by taking i.i.d. samples from the Uniform$(-0.5, 0.5)$ distribution; the same starting point was used for each algorithm to ensure a fair comparison. Note that, for a given initial point $(\boldsymbol{L}_0, \boldsymbol{R}_0)$ with $\boldsymbol{R}_0 \neq \boldsymbol{0}$, CCD++ needs to compute the residual matrix once at the beginning of the algorithm. In our experiments, the time required for computing the residual matrix was negligible (always less than 0.05% of the total time) and we do not include this overhead in our experimental results. For all SGD-based algorithms, we selected the initial step size based on a small sample of the data (1M entries): 0.0125 for Netflix ($r = 50$), 0.025 for Netflix ($r = 100$), 0.00125 for KDD ($r = 50$, $r = 100$) and 0.000625 for Syn1B and Syn10B. Throughout, we used the bold driver heuristic for step size selection with a step-size-increase factor of 5% and a step-size-decrease factor of 50%; step size selection was thus fully automatic. See (Teflioudi et al., 2012) for an experimental comparison of the bold driver heuristic with other popular step size sequences. We used the WOR training point schedule and the WOR stratum schedule throughout our experiments unless stated otherwise, and ran a truncated version of SGD that clipped the entries in the factor matrices to $[-100, 100]$ after every SGD step. Also, unless stated otherwise, all SGD-based algorithms make use of prefetching as in the SGD++ algorithm of Sec. 3.1. For each algorithm, we declared convergence as soon as it reached a point within 2% of the overall best solution.

We focus on experimental results for the shared-memory and shared-nothing settings. The performance of various sequential algorithms has been studied in (Teflioudi et al., 2012).

6.3. Shared-Memory Algorithms

In order to evaluate the shared-memory methods, we performed experiments on the Netflix, KDD, Syn1B-rect, and Syn1B-sq datasets on our high-memory server. We ran experiments using 8, 16, and 32 threads and refer to these setups as H8, H16, and H32, respectively.

We initially experimented with PSGD both with and without locking. We found that in settings with a large number of threads (H16 and H32), lock-free PSGD was slightly more compute-efficient than PSGD with locking (e.g., 9% for KDD, $r = 100$) without any degradation in the quality of the solution. We presume that this speedup originates from the reduced synchronization costs of lock-free PSGD, which we used throughout our experiments.

Netflix, KDD (Fig. 5). Fig. 5b and 5d show the time until convergence on Netflix and KDD, $r = 100$, for various methods and setups. All approaches led to a similar overall loss (all within 1% of each other); Fig. 5a and 5c show examples of the progress of the methods over time. On Netflix, PCCD++ and PALS found a slightly better solution (1% lower loss) than the SGD-based approaches; on KDD, results were almost identical.

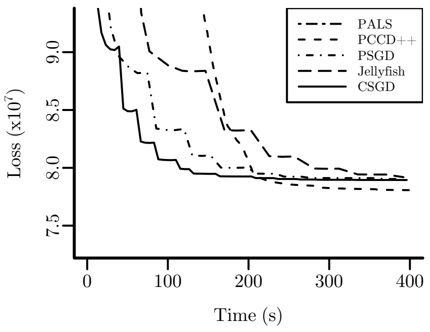
Across all setups and datasets, PALS was the slowest and CSGD the fastest method. The compute-efficiency of PALS was very low so that its epochs were significantly slower than that of alternative methods. In particular, on Netflix, PALS performed only 3 epochs within 400s and was not able to reach the vicinity of the points obtained by the other methods (for this reason, the PALS curve is not visible in Fig. 5a). PCCD++ performed better than PALS, and it outperformed PSGD on the Netflix dataset (but not on KDD). Jellyfish was slower than PSGD throughout; potentially because worker and shuffle threads compete for memory bandwidth and CPU cache. For this reason, we excluded Jellyfish from our larger-scale experiments. Finally, CSGD was faster than its closest competitors (PCCD++ on Netflix, PSGD on KDD); its cache-conscious blocking thus appears to be effective.[5]

The overall runtime of all algorithms improved significantly as we moved from 8 to 16 threads (1.6x–1.7x speedup on Netflix, 1.5x–1.9x on KDD). When we increased the number of threads further to $t = 32$, all methods still benefited on Netflix, but only PALS and CSGD gave good speedups on KDD (1.7–1.9x, versus $\leq$1.3x speedup for other methods). In particular, less CPU-intensive methods such as PCCD++, PSGD and Jellyfish hit the memory bandwidth on KDD. An exception is CSGD (1.7x speedup from H16 to H32), which avoids the memory bottleneck due to better cache utilization. Overall, PALS and CSGD scaled best w.r.t. the number of threads.

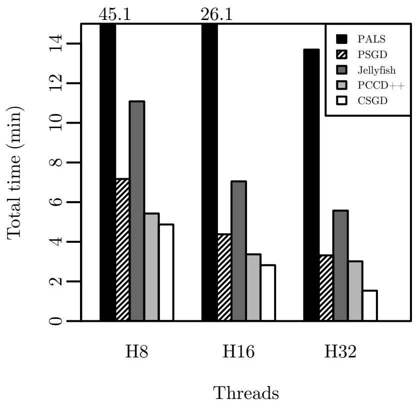
PCCD++ was the least data-efficient, and PALS the most data-efficient method. In more detail, PCCD++ required over 600 times more epochs than PALS to converge (e.g., 4856 vs. 8 epochs on KDD). Nevertheless, PCCD++ was consistently faster than PALS in terms of overall runtime. Recall that PCCD++ scans the input matrix $2r(T + 1)$ times per iteration, i.e., for processing one feature. It thus requires many epochs, but each epoch is inexpensive ($O(N)$ time). In contrast, PALS executes only a few epochs, but each epoch is expensive, taking $O(Nr^2 + mr^3)$ time. Similarly, PSGD and CSGD needed more epochs than PALS but fewer than PCCD++ to converge (e.g., 13 epochs for PSGD on KDD); the compute efficiencies of PSGD and CSGD lie between those of PALS and PCCD++.

Overall, we found that CSGD was the best-performing method on both datasets; CSGD on H16 was faster than any other method on H32. On Netflix, PCCD++ was closest (and slightly better in terms of quality); on KDD, PSGD was closest.

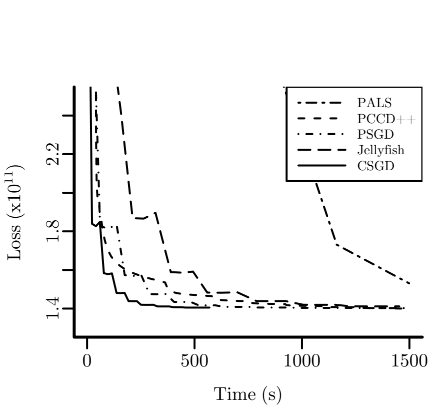
[5] Note that our results w.r.t. the relative performance of PSGD and PCCD++ differ from the ones in (Yu et al., 2012), presumably due to our use of the bold driver heuristic.



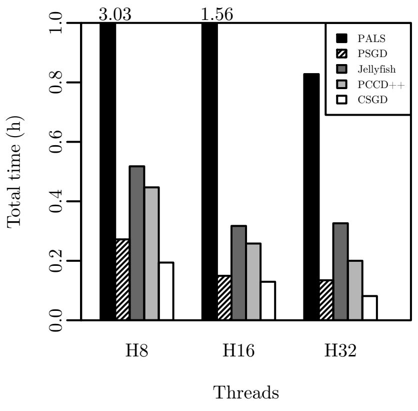
(a) Netflix on H32



(b) Netflix on H8, H16, and H32



(c) KDD on H32

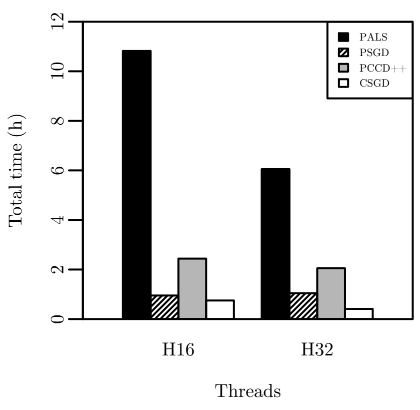


(d) KDD on H8, H16, and H32

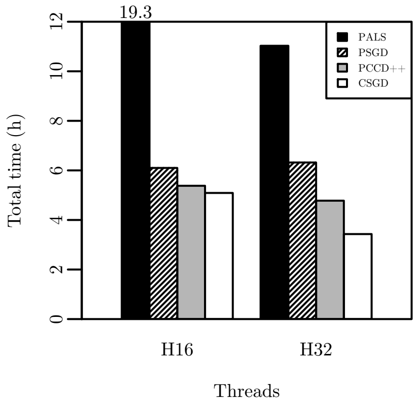
Fig. 5. Performance of shared-memory algorithms on real datasets, $r = 100$

Syn1B-sq (Fig. 6a). On the larger Syn1B-sq dataset, CSGD was again the best-performing method. Using 16 (resp., 32) threads, it was 1.3x (resp., 2.5x) faster than PSGD, the second-best-performing method (0.75h vs. 0.95h for H16; 0.41h vs. 1.04h for H32). As with KDD, PSGD was significantly faster than PCCD++. The scalability behavior of the various algortihms was also similar to that for KDD: When moving from 16 to 32 threads, PALS and PCCD++ achieved 1.8x and 1.18x speedup, respectively, PSGD became slower, and CSGD achieved a 1.9x speedup. As before, CSGD with 16 threads was faster than any other method with 32 threads.

Syn1B-rect (Figs. 6b and 7a). On Syn1B-rect, all methods were slower than on Syn1B-sq, presumably because there are more factors to learn. One striking result is that PALS did not converge to an acceptable solution, its loss being four orders of magnitude greater than all other methods. We therefore report the running time of PALS until its loss

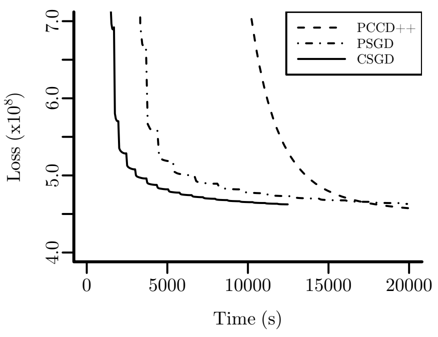


(a) Square matrix, 1B entries (Syn1B-sq)

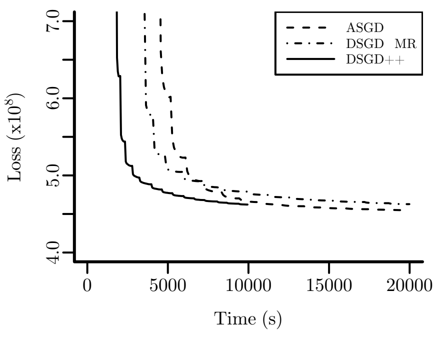


(b) Rectangular matrix, 1B entries (Syn1B-rect)

Fig. 6. Performance of shared-memory algorithms on synthetic datasets



(a) Syn1B-rect on H32



(b) Syn1B-rect on 4x8

Fig. 7. Comparison of shared-memory and shared-nothing methods on Syn1B-rect (PALS and DALS converge to a different point)

changed by less then 0.1% in two consecutive epochs. (Such erratic behavior of PALS was not observed on any other dataset.) As with Netflix, when moving from 16 to 32 threads, only PALS and CSGD yielded good speedups. Specifically, the speedups for PCCD++ and CSGD were 1.12x and 1.5x, respectively, and PSGD actually slowed down. Also as with Netflix, PCCD++ was somewhat faster than PSGD. In general, on Syn1B-rect, PCCD++ behaved better relative to alternative methods than it did on Syn1B-sq. A potential reason for this behavior is that the SGD-based methods are less data-efficient on Syn1B-rect. In particular, we observed that the SGD-based methods required only a small number of epochs to move to the vicinity of the solution, but converged slowly afterward. Nevertheless, CSGD was the fastest method. It achieved a 25% (57%) speedup in compute-efficiency over PSGD and ran 16.5% (45%) faster than PSGD on H16 (H32).

Overall, CSGD was the fastest method both on KDD and our synthetic datasets. PSGD was the second-fastest method on KDD and Syn1B-sq, while PCCD++ was the second-fastest method on Syn1B-rect and, on this dataset, was competitive with

Table 6. Impact of stratum schedules on DSGD++ (2x8)

	Netflix, $r = 100$			KDD, $r = 100$		
	SEQ	WOR	LA-WOR	SEQ	WOR	LA-WOR
Time/ep. (s)	**10.47**	11.5	10.61	**32.13**	40.8	32.62
Epochs	200	**65**	106	88	**62**	69
Total time (s)	2426	**861**	1300	3750	3182	**2976**

CSGD. PCCD++ was consistently faster (up to 3.5x) than PALS. On the other hand, being compute-intensive and slow, PALS was one of the most scalable methods; its performance always improved by adding more threads.

6.4. Shared-Nothing Algorithms

For the shared-nothing setting, we experimented with both real (Netflix, KDD) and synthetic datasets and used between 2 and 16 compute nodes with 8 threads each. We write $w \times t$ to refer to a setup with w compute nodes, each running t threads.

DSGD++ stratum schedule (Table 6). Recall that the DSGD++ stratum schedule affects both the time per epoch (governed by the relative number of local and remote fetches) and the number of epochs to convergence (governed by the degree of randomization). In Table 6, we compare the performance of DSGD++ with the SEQ, WOR, and LA-WOR schedules for the 2x8 setup. First, observe that WOR is more data-efficient than LA-WOR, which in turn is more data-efficient than SEQ. As with the SGD training point schedule, more randomness leads to better data-efficiency. Regarding compute-efficiency, we found that all three approaches performed similarly on Netflix, because $\bar{N}$ is large and thus communication costs are relatively small. In such a setting, WOR is the method of choice. On KDD, where $\bar{N}$ is small so that communication becomes significant, LA-WOR outperformed both WOR and SEQ. These results are in line with the analysis in Sec. 5, and we therefore recommend the WOR schedule for datasets with large $\bar{N}$ and LA-WOR for those with small $\bar{N}$.

Netflix, KDD. Both of our real-world datasets are only moderately large so that distributed processing may not be necessary. In fact, CSGD can factor these datasets in a few minutes, e.g., 2min for Netflix and 5min for KDD on H32. For experiments with these datasets in a shared-nothing environment, see (Teflioudi et al., 2012), where it was found that ASGD (2x8) and DSGD++ (4x8) were the fastest algorithms on Netflix, and DSGD++ was the fastest algorithm on KDD (2x8 and 4x8).

Large-scale experiments (Fig. 8). For our large-scale experiments in the shared-nothing setting, we used the Syn1B-rect, Syn1B-sq, and Syn10B datasets, and varied the number of nodes (2–16) and threads per node (2–8). When running DSGD++, we used the LA-WOR schedule for Syn1B-sq and Syn1B-rect, and the WOR schedule for Syn10B. The results are summarized in Fig. 8. The plots show the total time to convergence required by each of the shared-nothing methods and, if possible, the corresponding shared-memory baselines. Note that the available memory was insufficient to run PALS for all datasets (since it requires two copies of the data in memory) and to run PSGD for Syn10B. In the remainder of this section, we describe the results in more detail for each of the three datasets.

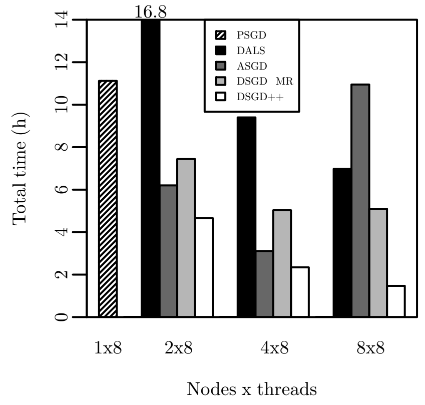
Syn1B-rect (Figs. 7b and 8a). We applied the shared-nothing algorithms to Syn1B-rect on the 1x8, 2x8, 4x8, and 8x8 setups. Syn1B-rect has a high value of $\bar{N}$, so that distributed processing should be relatively effective, at least for smaller numbers of nodes. DSGD-MR confirmed this expectation by performing better than the PSGD baseline in all cases, though the communication and synchronization overheads of DSGD-MR caused its performance to deteriorate slightly beyond 4 nodes. ASGD also performed better than the baseline on 2 and 4 nodes but, in contrast to DSGD-MR, did not behave gracefully beyond 4 nodes. Specifically, at 8 nodes the ASGD runtime became significantly longer than for PSGD. Even though ASGD was more compute-efficient on 8 nodes, the increased synchronization overhead led to increased delays between parameter updates, which drastically reduced data efficiency.

DSGD++ performed best in all setups. It was 2.4x faster than PSGD on 2 nodes and 4.7x faster on 4 nodes. Speedup was superlinear, presumably due to larger overall cache sizes. The communication overheads of DSGD++ did not adversely affect performance because of asynchronous communication and use of the LA-WOR schedule. On 8 nodes, the overhead of communication starts to become visible (7.5x speedup). Nevertheless, DSGD++ was able to factor Syn1B-rect in 88 minutes on 8 nodes; its closest competitor was ASGD, which required 186 minutes on 4 nodes. Even though we used different hardware for our shared-memory and shared-nothing experiments, our results suggest that on large datasets, DSGD++ is competitive with the best-performing shared-memory methods running a similar number of threads (see Fig. 7).

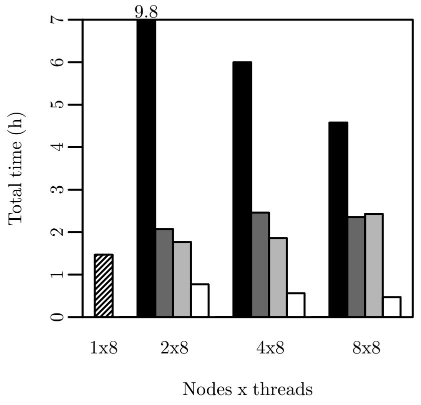
Our final observation is that, even though Syn1B-rect is inherently amenable to distributed processing because of its high $\bar{N}$ value, DALS did not converge to an acceptable solution. Moreover, DALS exhibited sublinear speedup. This sublinearity indicates that the time required to broadcast the factor matrices becomes significant as the number of nodes increases. The poor performance of DALS is reminiscent of the shared-memory experiments (Sec. 6.3), where the ALS-based algorithm (in that case PALS) also had convergence problems.

Syn1B-sq (Fig. 8b). On Syn1B-sq, all methods were faster than on Syn1B-rect since there were fewer factors to learn. Our other observations were as follows. First, we observed that DALS (now working correctly) was consistently slower than the PSGD baseline. As before, increasing the number of nodes led to sublinear speedup due to increased communication overhead. Second, neither ASGD nor DSGD-MR were able to improve on the PSGD baseline. Indeed, Syn1B-sq is our "hard" dataset (low $\bar{N}$) so that communication overheads dominate potential gains due to use of multiple processing nodes. Note that ASGD behaves more gracefully than on Syn1B-rect for a large number of nodes. We conjecture that the low value of $\bar{N}$ decreases the effect of delayed parameter updates since fewer SGD updates are run per column and time unit (but, as before, the time per epoch decreased and the number of epochs increased). Our final observation was that DSGD++ was the only method that was able to improve upon PSGD. It achieved speedups of 1.6x (2 nodes), 2.3x (4 nodes), and 3.5x (8 nodes). As expected, the speedups are lower than the ones for Syn1B-rect but nevertheless significant. Overall, our results indicate that DSGD++ is the only SGD-based method that can handle matrices with low $\bar{N}$ gracefully.

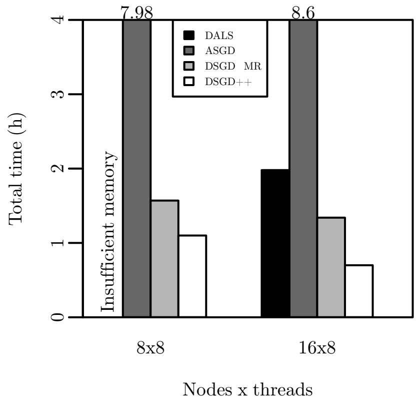
Syn10B (Fig. 8c). We could not run experiments on Syn10B using four or fewer nodes due to insufficient aggregate memory. We therefore show results only for 8x8 and 16x8. Even in this setting, we could not run DALS on 8 nodes because the available memory is insufficient to store the required two copies of data matrix and full copy of the factor matrix simultaneously (DALS requires at least 11 nodes to do this). On 16 nodes, DALS



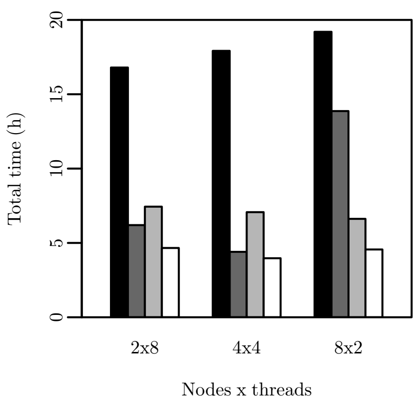
(a) Rectangular matrix, 1B entries (Syn1B-rect)



(b) Square matrix, 1B entries (Syn1B-sq)



(c) Rectangular matrix, 10B entries (Syn10B)



(d) Overhead of distributed processing (Syn1B-rect)

Fig. 8. Performance of shared-nothing algorithms on synthetic datasets

took 2h to converge; the increased density of the Syn10B matrix simplifies the completion problem so that only 11 epochs were needed. In contrast to DALS, all of the SGD-based methods were able to run on both 8x8 and 16x8. Since these methods store the data matrix only once and also fully partition the factor matrices, they are more memory efficient and can thus be used on smaller clusters. Strikingly, DSGD-MR was faster on 8 nodes than DALS on 16 nodes. ASGD did not converge to a satisfactory point in this experiment (2 orders of magnitude off the best loss) and was much slower than all other methods (as before, we declared convergence when the loss reduced by less than 0.1%). This is another indication that ASGD is not robust enough for larger clusters. Finally, DSGD++ required 1.1h on 8 nodes and 0.7h on 16 nodes. Thus DSGD++ was faster on 8 nodes than any other method on 16 nodes, and was almost twice as fast on 16 nodes as its closest competitor (DSGD-MR).

Impact of distributed processing (Fig. 8d). In our final experiment on Syn1B-rect, we investigated the behavior of the various algorithms as we increased the number of nodes while keeping the overall number of threads constant (2x8, 4x4, 8x2). Since the number of threads is identical in each setup, this allowed us to directly measure the impact of distributed processing. We observed that all approaches except ASGD handle the increased cluster size gracefully. The runtime of DALS increases slightly (increased cost of broadcasting), while the runtime of DSGD++ and DSGD-MR decrease slightly (more cache per thread). Thus even when less powerful compute nodes are available, these methods perform well. In contrast, the runtime of ASGD again increases sharply as we go beyond 4 nodes; see the discussion above.

7. Conclusion

We have provided parallel SGD algorithms for in-memory matrix completion in three different settings: the CSGD algorithm for shared-memory architectures, the ASGD and DSGD++ algorithms for shared-nothing architectures with MPI-like communication, and the DSGD-MR algorithm for the shared-nothing MapReduce framework. Our algorithms exploit thread-level parallelism, in-memory processing, asynchronous communication, and are cache-conscious. In experiments with data of intermediate and large sizes, the CSGD and DSGD++ algorithms, which are based on a novel stratification approach, performed best in the shared-memory and shared-nothing settings, respectively. Both algorithms scale to matrices with millions of rows, millions of columns, and billions of entries and consistently outperformed alternative approaches with respect to speed, scalability, and memory footprint.

References

Amatriain, X. and Basilico, J. (2012). Netflix recommendations: Beyond the 5 stars (part 1). `http://techblog.netflix.com/2012/04/netflix-recommendations-beyond-5-stars.html`.

Battiti, R. (1989). Accelerated backpropagation learning: Two optimization methods, *Complex Systems* **3**: 331–342.

Bennett, J. and Lanning, S. (2007). The Netflix prize, *Proc. of the KDD Cup Workshop*, pp. 3–6.

Byrd, R. H., Lu, P., Nocedal, J. and Zhu, C. (1995). A limited memory algorithm for bound constrained optimization, *SIAM Journal on Scientific Computing* **16**(5): 1190–1208.

Candes, E. J. and Recht, B. (2009). Exact matrix completion via convex optimization, *Foundations of Computational Mathematics* **9**(6): 717–772.

Chen, P.-L., Tsai, C.-T., Chen, Y.-N., Chou, K.-C., Li, C.-L., Tsai, C.-H., Wu, K.-W., Chou, Y.-C., Li, C.-Y., Lin, W.-S., Yu, S.-H., Chiu, R.-B., Lin, C.-Y., Wang, C.-C., Wang, P.-W., Su, W.-L., Wu, C.-H., Kuo, T.-T., McKenzie, T., Chang, Y.-H., Ferng, C.-S., Ni, C.-M., Lin, H.-T., Lin, C.-J. and Lin, S.-D. (2012). A linear ensemble of individual and blended models for music rating prediction, *Journal of Machine Learning Research: Proceedings Track* **18**: 21–60.

Chu, C. T., Kim, S. K., Lin, Y. A., Yu, Y. Y., Bradski, G., Ng, A. Y. and Olukotun, K. (2006). Map-Reduce for machine learning on multicore, *Advances in Neural Information Processing Systems (NIPS)*, pp. 281–288.

Cichocki, A. and Phan, A. H. (2009). Fast local algorithms for large scale nonnegative matrix and tensor factorizations, *IEICE Transactions* **92-A**(3): 708–721.

Das, A. S., Datar, M., Garg, A. and Rajaram, S. (2007). Google news personalization: Scalable online collaborative filtering, *Proc. of the Intl. Conf. on World Wide Web (WWW)*, pp. 271–280.

Das, S., Sismanis, Y., Beyer, K. S., Gemulla, R., Haas, P. J. and McPherson, J. (2010). Ricardo: Integrating R and Hadoop, *in* A. K. Elmagarmid and D. Agrawal (eds), *Proc. of the ACM Intl. Conf. on Management of Data (SIGMOD)*, ACM, pp. 987–998.

Dror, G., Koenigstein, N., Koren, Y. and Weimer, M. (2012). The Yahoo! Music dataset and KDD-Cup '11, *Journal of Machine Learning Research: Proceedings Track* **18**: 8–18.

Gemulla, R., Haas, P. J., Nijkamp, E. and Sismanis, Y. (2011). Large-scale matrix factorization with distributed stochastic gradient descent, *Technical Report RJ10481*, IBM Almaden Research Center, San Jose, CA. `http://researcher.watson.ibm.com/researcher/files/us-phaas/rj10482Updated.pdf`.

Gemulla, R., Nijkamp, E., Haas, P. J. and Sismanis, Y. (2011). Large-scale matrix factorization with distributed stochastic gradient descent, *Proc. of the ACM Intl. Conf. on Knowledge Discovery and Data Mining (SIGKDD)*, pp. 69–77.

Hu, Y., Koren, Y. and Volinsky, C. (2008). Collaborative filtering for implicit feedback datasets, *Proc. of the IEEE Intl. Conf. on Data Mining (ICDM)*, pp. 263–272.

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

Li, B., Tata, S. and Sismanis, Y. (2013). Sparkler: Supporting large-scale matrix factorization, *Proc. of the Intl. Conf. on Extending Database Technology (EDBT)*, pp. 625–636.

Liu, C., Yang, H.-c., Fan, J., He, L.-W. and Wang, Y.-M. (2010). Distributed nonnegative matrix factorization for web-scale dyadic data analysis on mapreduce, *Proc. of the Intl. Conf. on World Wide Web (WWW)*, New York, NY, USA, pp. 681–690.

Mackey, L., Talwalkar, A. and Jordan, M. (2011). Divide-and-conquer matrix factorization, *Advances in Neural Information Processing Systems (NIPS)*, pp. 1134–1142.

McDonald, R., Hall, K. and Mann, G. (2010). Distributed training strategies for the structured perceptron, *Human Language Technologies*, pp. 456–464.

MPI (2013). Message Passing Interface Forum. `http://www.mpi-forum.org`.

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

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

Smola, A. and Narayanamurthy, S. (2010). An architecture for parallel topic models, *Proc. of the VLDB Endowment* **3**(1–2): 703–710.

Teflioudi, C., Makari, F. and Gemulla, R. (2012). Distributed matrix completion, *Proc. of the IEEE Intl. Conf. on Data Mining (ICDM)*, pp. 655–664.

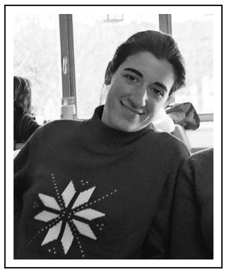
Tsitsiklis, J., Bertsekas, D. and Athans, M. (1986). Distributed asynchronous deterministic and stochastic gradient optimization algorithms, *IEEE Transactions on Automatic Control* **31**(9): 803–812.

Yu, H.-F., Hsieh, C.-J., Si, S. and Dhillon, I. S. (2012). Scalable coordinate descent approaches to parallel matrix factorization for recommender systems, *Proc. of the IEEE Intl. Conf. on Data Mining (ICDM)*, pp. 765–774.

Zhou, Y., Wilkinson, D., Schreiber, R. and Pan, R. (2008). Large-scale parallel collaborative filtering for the Netflix Prize, *Proc. of the Intl. Conf. on Algorithmic Aspects in Information and Management (AAIM)*, Vol. 5034, pp. 337–348.

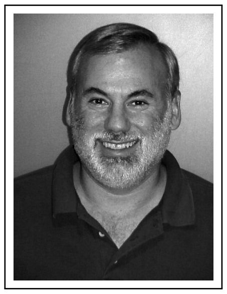
Author Biographies

Faraz Makari received both his B.S. and M.S. in Computer Science in 2008 and 2010 from the University of Saarland, Germany, respectively. He is currently a Ph.D. student in Computer Science at the Max Planck Institute for Informatics in Saarbrücken, Germany. His research interests mainly include efficient algorithms for large-scale matrix factorization and linear programming.



Christina Teflioudi received a Diploma in Electrical and Computer Engineering in 2007 from Aristotle University of Thessaloniki, Greece, and a M.S. in Computer Science in 2011 from the University of Saarland, Germany. She is currently pursuing a Ph.D. in Computer Science in Max Planck Institute for Informatics in Saarbrücken, Germany. She is mainly interested in data mining and in particular in matrix factorization and inductive logic programming.

Rainer Gemulla received a PhD in Computer Science in 2008 from Technische Universität Dresden, Germany, and has been a research group leader at the Max Planck Institute for Informatics at Saarbrücken, Germany, since 2011. He has been elected a junior fellow of the German Informatics Society (GI) in 2013. His research focuses on data analysis and data mining techniques for efficiently extracting useful information from large, complex data collections. He has published more than 20 conference publications and journal articles, holds multiple patents, and served on more than 35 program committees of international conferences, workshops, and journals.



Peter J. Haas received a PhD in Operations Research in 1985 from Stanford University, and has been a Research Staff Member at the IBM Almaden Research Center since 1987. He is also a Consulting Professor in the Department of Management Science and Engineering at Stanford University, teaching and pursuing research in stochastic modeling and simulation. At IBM, his work on the application of statistical and Monte Carlo techniques to problems in information management and knowledge discovery has been incorporated into several IBM products, and he has been awarded over 20 U.S. and international patents. He is a former president of the INFORMS Simulation Society currently serves on the editorial boards of *Operations Research*, *The VLDB Journal*, and *ACM Transactions on Modeling and Computer Simulation*. He has published over 120 conference publications, journal articles, and books. In 2007, he received the ACM SIGMOD Test-of-Time Award for his 1997 work on interactive exploration of massive datasets, and was recently designated an IBM Research Master Inventor.

Yannis Sismanis received a PhD degree in Computer Science from the University of Maryland, College Park in 2004, and he is a Research Staff Member at IBM Research - Almaden since then. His research focuses on massively parallel processing for big data, real-time business intelligence, structured and unstructured analysis, and information retrieval. He has published more than 25 papers in peer-reviewed journals, conferences, and workshops and holds numerous patents. Since 2004, he has served on more than 30 program committees of international conferences and workshops.

Correspondence and offprint requests to: Faraz Makari, Max Planck Institute for Computer Science, Saarbrücken, Germany. Email: fmakari@mpi-inf.mpg.de