2012 IEEE 12th International Conference on Data Mining

Distributed Matrix Completion

Christina Teflioudi, Faraz Makari, Rainer Gemulla
Max-Planck-Institut für Informatik
Saarbrücken, Germany
{chteflio,fmakari,rgemulla}@mpi-inf.mpg.de

***Abstract*—We discuss parallel and distributed 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 on a small cluster of commodity nodes; even very large problems can be handled effectively in such a setup. Our DALS, ASGD, and DSGD++ algorithms are novel variants of the popular alternating least squares and stochastic gradient descent algorithms; they exploit thread-level parallelism, in-memory processing, and asynchronous communication. We provide some guidance on the asymptotic performance of each algorithm and investigate the performance of both our algorithms and previously proposed MapReduce algorithms in large-scale experiments. We found that DSGD++ outperforms competing methods in terms of overall runtime, memory consumption, and scalability. Using DSGD++, we can factor a matrix with 10B entries on 16 compute nodes in around 40 minutes.**

***Keywords*-parallel and distributed matrix factorization; stochastic gradient descent; ALS; recommender systems**

I. Introduction

Low-rank matrix completion techniques have recently received significant attention in the data mining community; they have been successfully applied in the context of collaborative filtering in recommender systems [1]–[9]. At its heart, matrix completion is a variant of low-rank matrix factorization in which the input matrix is only partially observed and potentially noisy. In the setting of recommender systems, matrix rows correspond to users or customers, columns to items, such as movies or music 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.

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 [10].[1] Similarly, Yahoo Music! collected billions of user ratings for musical pieces [11]. At such massive scales, parallel and distributed algorithms for matrix factorization are essential to achieve reasonable performance [2], [3], [6]–[9], [12], [13].

In this paper, we study parallel and distributed algorithms for large-scale matrix completion. We focus on in-memory algorithms that run on a small cluster of commodity nodes; even very large factorization tasks can be handled effectively in such a setup. In contrast, most existing distributed algorithms for matrix factorization are designed for MapReduce [2], [3], [9], [13]. Compared to our setting, MapReduce algorithms have multiple drawbacks: (1) they need to repeatedly read the input data from disk into memory, (2) they are limited to the MapReduce programming model, and (3) they may suffer from runtime overheads of popular implementations such as Hadoop (which has been designed for much larger clusters and different workloads). These drawbacks have been recognized by the community: The next version[2] of Hadoop will allow applications to use different programming models seamlessly (e.g., MPI[3]). Most Hadoop clusters, as well as the machines offered by most cloud providers, have limited per-node computing capability (say, 8 cores) and memory (say, 64GB). The algorithms discussed in this paper are designed to run in such a setting and can readily be integrated into an existing data mining infrastructure.

Popular algorithms for large-scale matrix completion are based on alternating least-squares (ALS) [9] or stochastic gradient descent (SGD) [5]. We review previous parallel (i.e., shared memory) and MapReduce versions of these algorithms. In general, parallel algorithms are a good fit when the input data is small enough to fit into main memory of a single machine with a sufficiently large number of CPUs. MapReduce algorithms can handle much larger datasets by leveraging the aggregate memory and CPUs of multiple compute nodes. As mentioned above, however, these algorithms induce a severe performance penalty (even when implemented in our fast in-memory MapReduce engine). In this paper, we propose a set of distributed (i.e., shared nothing) algorithms that are faster, more scalable, and less memory-intensive than existing MapReduce algorithms. In particular, our DALS algorithm is a scalable variant of ALS that benefits from thread-level parallelism to speed up processing and reduce the memory footprint; it is closely related to the distributed ALS algorithm of [9]. For SGD, we propose the asynchronous ASGD algorithm, which is inspired by recent work on distributed LDA [15], and the DSGD++ algorithm, which is based on the MapReduce algorithm of [3]. Both ASGD and DSGD++ are designed to exploit thread-level parallelism, in-memory processing, and asynchronous communication.

[1]http://techblog.netflix.com/2012/04/netflix-recommendations-beyond-5-stars.html

[2]http://developer.yahoo.com/blogs/hadoop/posts/2011/02/mapreduce-nextgen/

[3]http://www.mcs.anl.gov/research/projects/mpi/index.htm

The main challenges in distributed matrix completion are to partition the data effectively across the compute cluster and to minimize the amount of communication between different nodes. We provide the key metrics of the input data that play a crucial role in the performance of distributed matrix completion: the rank of the factorization (for DALS) and the ratio of the number of revealed entries and the number of columns (for ASGD and DSGD++), by both asymptotic analysis and experimental study. Finally, we compare all parallel and distributed algorithms in an extensive set of experiments on both real-word and synthetic datasets of varying sizes. On large datasets, we found that DSGD++ outperforms competing methods in terms of overall runtime, memory consumption, and scalability. For example, DSGD++ can factor a matrix with 10B entries on 16 compute nodes in around 40 minutes.

The remainder of this paper is organized as follows: Sec. II reviews the matrix completion problem. In Secs. III, IV, V, we discuss sequential, parallel, and distributed algorithms, respectively. Our experimental findings are summarized in Sec. VI. We conclude the paper in Sec. VII.

II. The Matrix Completion Problem

To gain understanding about applications of matrix completions, consider the "Netflix problem" [10] 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 a feedback matrix such as

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

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 "?"); entries with a high predicted rating are then recommended to users for viewing. This matrix-completion approach to recommender systems has been successfully applied in practice; see [5] for an excellent discussion of the underlying intuition.

Denote by the *training set* $Z = \{ z_1, \dots, z_N \}$ the set of observed entries in $\boldsymbol{V}$, where $z_k = (i_k, j_k)$, $k \in [1, N]$, $i_k \in [1, m]$, 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 observed 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 $\boldsymbol{W}^*$ and an $r \times n$ column-factor matrix $\boldsymbol{H}^*$ such that $\boldsymbol{V} \approx \boldsymbol{W}^*\boldsymbol{H}^*$, i.e., we aim to approximate $\boldsymbol{V}$ by the low-rank matrix $\boldsymbol{W}^*\boldsymbol{H}^*$. The approximation is governed by an application-dependent loss function $L(\boldsymbol{W}, \boldsymbol{H})$; 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.,

$$(\boldsymbol{W}^*, \boldsymbol{H}^*) = \operatorname*{argmin}_{\boldsymbol{W}, \boldsymbol{H}} L(\boldsymbol{W}, \boldsymbol{H}).$$

We can now view the matrix $\boldsymbol{W}^*\boldsymbol{H}^*$ as a "completed version" of $\boldsymbol{V}$: unobserved entry $\boldsymbol{V}_{ij}$ is predicted by $[\boldsymbol{W}^*\boldsymbol{H}^*]_{ij}$.

The loss function L measures the difference between the observed entries in $\boldsymbol{V}$ and the corresponding entries in $\boldsymbol{W}\boldsymbol{H}$, but may also incorporate regularization terms, biases, implicit feedback, temporal effects, or confidence levels. Thus our formulation of the matrix completion problem is driven by its application in data mining; see for example [16] for a treatment of its theoretical foundations. The most basic loss is the squared loss $L_{\text{Sl}}(\boldsymbol{W}, \boldsymbol{H}) = \sum_{(i,j) \in Z} (\boldsymbol{V}_{ij} - [\boldsymbol{W}\boldsymbol{H}]_{ij})^2$. Table I 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 [8]. L_{L2w} incorporates weighted L2 regularization [9], in which the amount of regularization depends on the number of observed entries. This particular loss was a key ingredient in the best performing solutions of both the Netflix competition and the KDD-Cup 2011 [1], [5], [9].

Following [17], we say 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 observed elements of $\boldsymbol{V}$, i.e., $L(\boldsymbol{W}, \boldsymbol{H}) = \sum_{(i,j) \in Z} L_{ij}(\boldsymbol{W}_{i*}, \boldsymbol{H}_{*j})$, where $\boldsymbol{W}_{i*}$ and $\boldsymbol{H}_{*j}$ refer to the i-th row of $\boldsymbol{W}$ and j-th column of $\boldsymbol{H}$, respectively. In this paper, we focus on loss functions that admit a summation form; this includes all of the loss functions given in Table I, in which we also show the corresponding local losses.

III. Sequential Algorithms

We focus on algorithms based on gradient descent (e.g., GD, L-BFGS, SGD) and alternating projections (ALS), which have been shown to perform best in our setting [3], [7]–[9].[4] All algorithms start with some initial point $(\boldsymbol{W}_0, \boldsymbol{H}_0)$ and iteratively improve it. Under appropriate conditions [14], they asymptotically converge to a local minimum or stationary point of L. We both summarize the basic algorithms and discuss practical considerations.

A. Alternating Least Squares

ALS [9] alternates between optimizing for $\boldsymbol{W}$ given $\boldsymbol{H}$, and optimizing for $\boldsymbol{H}$ given $\boldsymbol{W}$. For L_{Sl}, this amounts to computing the least squares solutions to the following systems of linear equations

$$\text{Compute } \boldsymbol{W}_{n+1}\text{: } (\forall i)\ \underline{\boldsymbol{W}_{i*}} \boldsymbol{H}_n^{(i)} = \boldsymbol{V}_{i*}, \tag{1}$$

$$\text{Compute } \boldsymbol{H}_{n+1}\text{: } (\forall j)\ \boldsymbol{W}_{n+1}^{(j)} \underline{\boldsymbol{H}_{*j}} = \boldsymbol{V}_{*j}, \tag{2}$$

where the unknown variable is underlined, $\boldsymbol{V}_{i*}$ (resp. $\boldsymbol{V}_{*j}$) denotes the revealed entries in row i (column j), and $\boldsymbol{H}_n^{(i)}$ (resp. $\boldsymbol{W}_{n+1}^{(j)}$) refers to the corresponding columns of $\boldsymbol{H}_n$ (rows of $\boldsymbol{W}_{n+1}$). Note that the sparse matrix $\boldsymbol{V}$ is accessed by row in Eq. (1) and by column in Eq. (2). For this reason, ALS implementations need to store two sparse representations of $\boldsymbol{V}$ in memory: one in row-major and one in column-major order. We refer to the application of (1) or (2) as a $\boldsymbol{W}$-epoch or $\boldsymbol{H}$-epoch, respectively; each epoch requires a single pass over the data. We handle loss functions L_{L2} and L_{L2w} as in [9].

[4]We exclude algorithms that require that the revealed entries in $\boldsymbol{V}$ are distributed uniformly from our discussion. Many other algorithms have been developed for this setting; see for example [6].

TABLE I: Popular loss functions for matrix completion

Loss function	Definition	Local loss
L_{Sl}	$\sum_{(i,j)\in Z}(\boldsymbol{V}_{ij} - [\boldsymbol{WH}]_{ij})^2$	$(\boldsymbol{V}_{ij} - [\boldsymbol{WH}]_{ij})^2$
L_{L2}	$L_{\mathrm{Sl}} + \lambda\left(\sum_{ik}\boldsymbol{W}_{ik}^2 + \sum_{kj}\boldsymbol{H}_{kj}^2\right)$	$(\boldsymbol{V}_{ij} - [\boldsymbol{WH}]_{ij})^2 + \lambda\sum_k(N_{i*}^{-1}\boldsymbol{W}_{ik}^2 + N_{*j}^{-1}\boldsymbol{H}_{kj}^2)$
L_{L2w}	$L_{\mathrm{Sl}} + \lambda\left(\sum_{ik}N_{i*}\boldsymbol{W}_{ik}^2 + \sum_{kj}N_{*j}\boldsymbol{H}_{kj}^2\right)$	$(\boldsymbol{V}_{ij} - [\boldsymbol{WH}]_{ij})^2 + \lambda\sum_k(\boldsymbol{W}_{ik}^2 + \boldsymbol{H}_{kj}^2)$

Under our running assumption that $m \geq n$, an ALS epoch then has time complexity $O(Nr^2 + mr^3)$.

B. Gradient-Based Methods

For brevity, we will write $L(\theta)$ and $L'(\theta)$, where $\theta = (\boldsymbol{W}, \boldsymbol{H})$, to denote the loss function and its gradient. Denote by $\nabla_{\boldsymbol{W}} L$ (resp. $\nabla_{\boldsymbol{H}} L$) the $m \times r$ (resp. $r \times n$) matrix of the partial derivatives of L w.r.t. to the entries in $\boldsymbol{W}$ (resp. $\boldsymbol{H}$). Then $L' = (\nabla_{\boldsymbol{W}} L, \nabla_{\boldsymbol{H}} L)$. For example, $[\nabla_{\boldsymbol{W}} L_{\mathrm{Sl}}]_{ik} = -2\sum_{(i,j)\in Z_{i*}} \boldsymbol{H}_{kj}(\boldsymbol{V}_{ij} - [\boldsymbol{WH}]_{ij})^2$, where Z_{i*} denotes the set of observed entries in row $\boldsymbol{V}_{i*}$.

Gradient descent. 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 [18].

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

$$\theta_{n+1} = \theta_n - \epsilon_n \hat{L}'(\theta). \tag{3}$$

We obtain a noisy gradient estimate by scaling up just one of the *local gradients* [5], i.e., $\hat{L}'(\theta) = NL'_{ij}(\theta)$ for some $(i,j) \in Z$. 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 $\boldsymbol{V}_{ij}$, $\boldsymbol{W}_{i*}$ and $\boldsymbol{H}_{*j}$. Therefore, we need only update a *single row* $\boldsymbol{W}_{i*}$ and a *single column* $\boldsymbol{H}_{*j}$ in each SGD step.

Step size sequence. A simple adaptive method for selecting the step size worked extremely well in our experiments (but does not guarantee asymptotic convergence). 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 we can compute the current loss after every epoch, we employ a heuristic called *bold driver* [19]. Starting from an initial step size ϵ_0, we (1) increase the step size by a small percentage (say, 5%) whenever the loss decreased after one epoch, and (2) drastically decrease the step size (say, by 50%) if the loss increased. Within each epoch, the step size remains fixed. To obtain ϵ_0, we try different step sizes on a small sample (say, 0.1%) of Z and pick the one that works best.

Training point schedule. Common schedules for an SGD epoch are: process Z sequentially in some fixed order (SEQ), sample with replacement from Z (WR), and sample without replacement from Z (WOR). In practice, WOR often outperforms WR in terms of number of epochs to convergence;[5] SEQ requires yet more epochs and also 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 suggest to prefetch the data needed in SGD step $n+2$ in steps n and $n-1$ into the CPU cache (e.g., using gcc's `__builtin_prefetch` macro). We refer to SGD with prefetching as SGD++; see the algorithm in Fig. 1. In our experiments, SGD++ was up to 15% faster than SGD (see Sec. VI-C). If sufficient main memory is available, an alternative approach to SGD++ is to shuffle a copy of the data in a parallel thread [8].

```
Require: A training set $Z$, initial values $\boldsymbol{W}_0$ and $\boldsymbol{H}_0$
  while not converged do  /* step */
    Prefetch $(i_{n+2}, j_{n+2}) \in Z$ for next but one step
    Prefetch $\boldsymbol{V}_{i_{n+1}j_{n+1}}$, $\boldsymbol{W}_{i_{n+1}*}$, $\boldsymbol{H}_{*j_{n+1}}$ for next step
    $\boldsymbol{W}'_{i_n*} \leftarrow \boldsymbol{W}_{i_n*} - \epsilon_n N \nabla_{\boldsymbol{W}_{i_n*}} L_{ij}(\boldsymbol{W}, \boldsymbol{H})$
    $\boldsymbol{H}_{*j_n} \leftarrow \boldsymbol{H}_{*j_n} - \epsilon_n N \nabla_{\boldsymbol{H}_{*j_n}} L_{ij}(\boldsymbol{W}, \boldsymbol{H})$
    $\boldsymbol{W}_{i_n*} \leftarrow \boldsymbol{W}'_{i_n*}$
  end while
```

Fig. 1: The SGD++ Algorithm for Matrix Factorization

C. Discussion

Within each epoch, SGD performs many quick-and-dirty steps whereas GD or L-BFGS perform a single careful step. For large matrices, the increased number of SGD steps leads to much faster convergence [3]. For this reason, we focus on SGD in our ongoing discussion. Since 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, $r \leq 50$).

IV. Parallel Algorithms

Even on only moderately large matrices, both SGD and ALS may need a significant amount of time to converge. Fortunately, both algorithms are easy to parallelize effectively. We describe shared-memory algorithms that run on t concurrent threads.

A. Parallel ALS

Parallel ALS (PALS) is based on the observation that ALS solves multiple independent least-squares problems. For

[5] We use "convergence" to refer to running an algorithm until some convergence criterion is met, and "asymptotical convergence" for the actual limit.

example, when updating $\boldsymbol{W}$ according to Eq. (1), a least-squares problem is solved for every row $\boldsymbol{W}_{i*}$; these problems are independent because the inputs $\boldsymbol{V}$ and $\boldsymbol{H}_n$ remain fixed. Thus, we partition the rows of $\boldsymbol{W}$ evenly among the threads, so that each thread solves m/t least-squares problems. After all threads have finished, we proceed to updating $\boldsymbol{H}$ in the same way. We found that PALS achieves almost linear speed-up w.r.t. ALS.

B. Parallel SGD

To obtain a parallel version of SGD (PSGD), we follow [8] and partition the training point schedule evenly among the threads, i.e., each thread runs N/t SGD steps per epoch. To avoid concurrent parameter updates, we lock row i of $\boldsymbol{W}$ and column j of $\boldsymbol{H}$ before processing training point (i, j). Since there are usually significantly more rows and columns than available threads (i.e., $m, n \gg t$), it is unlikely that overlapping rows and columns are processed by multiple threads at the same time so that there is little lock contention. Niu et al. [7] also experimented with a lock-free version of PSGD, in which inconsistent updates are allowed, but found virtually no difference in running time for matrix completion problems.

V. Distributed Algorithms

Distributed algorithms are designed for large-scale completion problems in which parallel algorithms are too slow to converge and/or the input matrix and factors do not fit into main memory of a single node. Denote by w the number of compute nodes and by t the number of cores per node; the total number of cores is thus given by $p = wt$. The main challenge in distributed matrix completion is to effectively manage the communication between the compute nodes.

All distributed algorithms partition the matrices $\boldsymbol{V}$, $\boldsymbol{W}$ and $\boldsymbol{H}$ across the compute nodes. To do so, we divide $\boldsymbol{V}$ into $m_b \times n_b$ blocks—where the values of m_b and n_b depend on the particular algorithm—such that each block is an $(m/m_b) \times (n/n_b)$ matrix. We shuffle the rows and columns of the data matrix before blocking; thus each block contains $N/m_b n_b$ training points in expectation. We write $\boldsymbol{V}^{b_i b_j}$ to refer to the block (b_i, b_j) of $\boldsymbol{V}$. The factor matrices are blocked conformingly, i.e., $\boldsymbol{W}$ is blocked $m_b \times 1$ and $\boldsymbol{H}$ is blocked $1 \times n_b$.

A. Distributed ALS

We describe a distributed ALS algorithm (DALS) similar to the one of Zhou et al. [9]. The algorithm assumes 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 $\boldsymbol{W}$ and $\boldsymbol{H}$.

Data partitioning. Similar to ALS, the data matrix needs to be stored twice: once in row-major (denoted $\boldsymbol{V}_{\mathrm{r}}$) and once in column-major order (denoted $\boldsymbol{V}_{\mathrm{c}}$). DALS uses a $w \times 1$ blocking for $\boldsymbol{V}_{\mathrm{r}}$ and a $1 \times w$ blocking for $\boldsymbol{V}_{\mathrm{c}}$. Factor matrix $\boldsymbol{W}$ is blocked conformingly to $\boldsymbol{V}_{\mathrm{r}}$ (i.e., $w \times 1$), $\boldsymbol{H}$ is blocked conformingly to $\boldsymbol{V}_{\mathrm{c}}$ (i.e., $1 \times w$). Node k stores blocks $\boldsymbol{V}_{\mathrm{r}}^{k1}$, $\boldsymbol{V}_{\mathrm{c}}^{1k}$, $\boldsymbol{W}^{k1}$, and $\boldsymbol{H}^{1k}$. This memory layout is illustrated in Fig. 2a. Colors indicate whether the data is node-local (green), subject to communication (yellow), or temporary (red).

Algorithm. DALS extends the main idea of PALS to a distributed setting, i.e., every node updates a part of a factor matrix in parallel. To update $\boldsymbol{W}$ according to (1), the k-th node updates $\boldsymbol{W}^{k1}$—i.e., its local block of $\boldsymbol{W}$—using PALS with t threads. This requires access to $\boldsymbol{V}_{\mathrm{r}}^{k1}$ and the *entire* $\boldsymbol{H}$ matrix. Note that $\boldsymbol{V}^{k1}$ is stored locally at node k, but $\boldsymbol{H}$ is not. Therefore, DALS creates a local copy of $\boldsymbol{H}$ on each node by broadcasting blocks $\boldsymbol{H}^{11}, \ldots \boldsymbol{H}^{1k}$. Since the least-squares problems solved by ALS are expensive, the incurred communication cost was negligible in our experiments. The update of $\boldsymbol{H}$ according to (2) is done along the same lines; node k updates $\boldsymbol{H}^{1k}$ using $\boldsymbol{V}_{\mathrm{c}}^{1k}$ and a copy of $\boldsymbol{W}$. Note that DALS differs from the algorithm of [9] only in that it uses multiple threads instead of multiple processes on each node. This allows us to share factor matrices among threads, which greatly reduces memory consumption.

B. Asynchronous SGD

Algorithms based on SGD are harder to distribute than ALS because the SGD steps of Eq. (3) are not independent. In general, we cannot simply partition the data and factors across the cluster such that (1) each pair of nodes works on disjoint pieces of $\boldsymbol{W}$ and $\boldsymbol{H}$ (so that nodes can run independently) and (2) each node processes roughly the same amount of data (so that distributed processing is effective). To see this, assume to the contrary that there exists such a partitioning and that some node k is responsible for training points $Z_k \subseteq Z$. Further suppose that $(i, j) \in Z_k$. Since we use the local gradients L'_{ij} as estimates of the gradient L', an SGD step on (i, j) will update $\boldsymbol{W}_{i*}$ and $\boldsymbol{H}_{*j}$. 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 Z_k, i.e., $(i, j) \in Z_k \implies Z_{i*} \cup Z_{*j} \subseteq Z_k$. Thus, Z_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 such that we obtain a $w \times w$ block-diagonal matrix with a balanced number of revealed entries in each block on the diagonal. This is not possible in general; in fact, 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 (this section) and distributed SGD (next sections).

Suppose that $\boldsymbol{V}$ is blocked $w \times 1$, $\boldsymbol{W}$ is blocked conformingly $w \times 1$, and $\boldsymbol{H}$ is blocked $1 \times w$. At each node k, we store blocks $\boldsymbol{V}^{k1}$, $\boldsymbol{W}^{k1}$, and $\boldsymbol{H}^{1k}$. Note that this particular blocking ensures that each update of $\boldsymbol{W}$ is node-local, whereas updates of $\boldsymbol{H}$ are either local or remote. In the following, we refer to $\boldsymbol{H}_{*j}$ as the *master copy* of column j; the node that stores $\boldsymbol{H}_{*j}$ is referred to as *master node*. A naive way to implement distributed SGD is as follows: When node k processes training point (i, j), it locks column j at its master node, fetches $\boldsymbol{H}_{*j}$, updates $\boldsymbol{W}_{i*}$ and $\boldsymbol{H}_{*j}$ locally according to (3), sends the new value of $\boldsymbol{H}_{*j}$ back to the master, and unlocks. This *synchronous algorithm* is clearly impractical, because the SGD

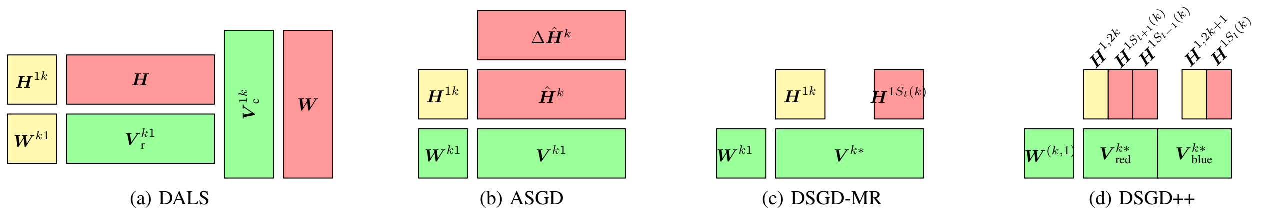


Fig. 2: Memory layout used on node k by the distributed algorithms ($t = 1$). Node-local data is shown in green, master copies in yellow, and temporary data in red.

steps of (3) are inexpensive so that most of the time is spent in communicating columns of $\boldsymbol{H}$.

Our asynchronous SGD algorithm (ASGD) avoids this problem by storing a *working copy* $\hat{\boldsymbol{H}}_{*j}^k$ of column $\boldsymbol{H}_{*j}$ 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 $\hat{\boldsymbol{H}}_{*j}^k$ 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. [20] proposed to average 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* $\Delta\hat{\boldsymbol{H}}_{*j}^k$ to the master, where $\Delta\hat{\boldsymbol{H}}_{*j}^k$ is given by the sum of updates to $\hat{\boldsymbol{H}}_{*j}$ since the last averaging. Whenever a master node has received all update vectors $\Delta\hat{\boldsymbol{H}}_{*j}^1, \ldots, \Delta\hat{\boldsymbol{H}}_{*j}^w$, 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. 2b.

In contrast to SGD, updates to a column of $\hat{\boldsymbol{H}}_{*j}$ 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 [21]. In contrast to general asynchronous SGD, we average only a subset of the parameters (i.e., $\boldsymbol{H}$ but not $\boldsymbol{W}$); this idea is motivated by the work on distributed LDA in [15]. 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 to 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, and run PSGD using t threads instead of SGD on each node. An additional thread takes care of averaging; this thread has low CPU utilization since the time to compute the update vectors is swamped by communication costs.

C. Distributed SGD on MapReduce

Recall the discussion at the beginning of Sec. V-B, where we argued that distributing SGD is hard because, in general, we cannot write $\boldsymbol{V}$ as a $w \times w$ block-diagonal matrix with a balanced number of revealed entries in the diagonal blocks.

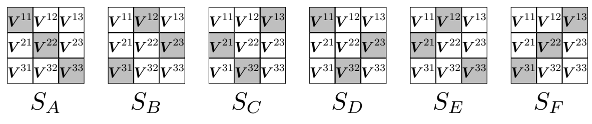


Fig. 3: Strata used by DSGD-MR for a 3×3 blocking of $\boldsymbol{V}$

The key idea of the MapReduce SGD algorithm (DSGD-MR) of [3] is to induce such a blocking by ignoring some of the entries in $\boldsymbol{V}$ in a principled way.

DSGD-MR. In more detail, suppose that $\boldsymbol{V}$ is blocked $w \times w$; node k stores blocks $\boldsymbol{V}^{k1}, \ldots, \boldsymbol{V}^{kw}$, $\boldsymbol{W}^{k1}$, and $\boldsymbol{H}^{1k}$. This layout is illustrated in Fig. 2c, where $\boldsymbol{V}^{k*}$ refers to the k-th row of blocks of $\boldsymbol{V}$. Now observe that when running SGD on some block $\boldsymbol{V}^{b_i b_j}$, we need only access the matrices $\boldsymbol{W}^{b_i 1}$ and $\boldsymbol{H}^{1 b_j}$. Thus we can independently process all the blocks on the main diagonal (i.e., $\boldsymbol{V}^{11}, \ldots, \boldsymbol{V}^{ww}$): node k processes block $\boldsymbol{V}^{kk}$, the required blocks $\boldsymbol{W}^{k1}$ and $\boldsymbol{H}^{1k}$ are node-local. In general, we say that two different blocks $\boldsymbol{V}^{b_i b_j}$ and $\boldsymbol{V}^{b_i' b_j'}$ are *interchangeable* whenever $b_i \neq b_i'$ and $b_j \neq b_j'$, i.e., they share neither rows nor columns. We call a set of w pairwise interchangeable blocks a *stratum*, the set of all strata is denoted by $\mathscr{S}$; see Fig. 3 for an example.

It is convenient to view a stratum as a bijective map from a node k (or row-block index) to a column-block index $b_j = S(k)$; e.g., $S_B(1) = 2$, $S_B(2) = 3$, and $S_B(3) = 1$ in the example of Fig. 3. DSGD-MR now 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: node k processes block $\boldsymbol{V}^{kS(k)}$. Continuing the example with $S = S_B$, node 1 processes block $\boldsymbol{V}^{12}$, node 2 block $\boldsymbol{V}^{23}$, and node 3 block $\boldsymbol{V}^{31}$. By construction, the required blocks $\boldsymbol{V}^{kS(k)}$ and $\boldsymbol{W}^{k1}$ are node-local; block $\boldsymbol{H}^{1S(k)}$ is fetched from node $S(k)$ before processing and stored back afterwards. Thus only blocks of $\boldsymbol{H}$ are communicated by DSGD-MR. In what follows, we refer to processing a single stratum as a *subepoch* and to a sequence of w subepochs as an *epoch*. Note that an epoch roughly corresponds to processing N training points: each block contains N/w^2 entries in expectation, we process w blocks per subepoch, and there are w 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 DSGD-MR. Formally, a stratum schedule is a sequence $S_1, S_2, \ldots$ of strata from $\mathscr{S}$; DSGD-MR processes stratum S_l in the l-th subepoch.

It can be shown that DSGD-MR asymptotically converges to a stationary point of L under certain conditions on the stratum schedule [3]. E.g., when the step size is fixed, a schedule must satisfy the property that every training point is processed equally often in the long run. In what follows, we focus on only the first epoch of DSGD-MR. The simplest correct schedule processes blocks in sequential order on each node (SEQ), i.e., we set $S_l(k) = 1 + (k + l - 2 \mod w)$. In the example above, this corresponds to the sequence (S_A, S_B, S_C). An alternative is to select a random stratum from $\mathscr{S}$ in each subepoch (WR); e.g., strata (S_A, S_E, S_C). Finally, we may select strata randomly but ensure that every block is processed exactly once per epoch (WOR); e.g., (S_B, S_A, S_C) or (S_F, S_E, S_D). In our experiments, we found that WOR achieves best results since (1) every training point is processed in every epoch and (2) the order of blocks is randomized as much as possible.

D. DSGD++

DSGD-MR has been developed for a MapReduce environment, in which nodes cannot communicate directly and data is stored on disk. In our setting, we can improve on DSGD-MR in multiple ways. Our DSGD++ algorithm uses a novel data partitioning and stratum schedule, but also improves on DSGD-MR by exploiting direct memory access and multi-threading.

Direct fetches. Denote by S_l the stratum used in the l-th subepoch and by $S_l^{-1}(b_j)$ the node which updates block $\boldsymbol{H}^{1b_j}$ in subepoch l. When running subepoch l, we communicate the blocks of $\boldsymbol{H}$ directly between the nodes in DSGD++ (i.e., we avoid storing back the result as in DSGD-MR). In more detail, node k fetches the block $\boldsymbol{H}^{kS_l(k)}$ directly from node $S_{l-1}^{-1}(S_l(k))$, which processed it in the previous subepoch.

Overlapping subepochs. Node k starts processing block $\boldsymbol{V}^{kS_l(k)}$ as soon as $\boldsymbol{H}^{kS_l(k)}$ has been received. This allows us to overlap subepochs and thus compensate for varying runtimes across the nodes: When node k starts processing its block for subepoch l, some other nodes might still be running subepoch $l - 1$.

Asynchronous communication. Observe that DSGD-MR is separated into a communication phase (receive next block of $\boldsymbol{H}$) and a computation phase (process block of $\boldsymbol{V}$). In our setting, we overlay communication and computation as follows: We use a $w \times 2w$ blocking of $\boldsymbol{V}$ instead of a $w \times w$ blocking. In each epoch, we conceptually partition $\boldsymbol{V}$ (and conformingly $\boldsymbol{H}$) at random into two matrices $\boldsymbol{V}_{\text{red}}$ and $\boldsymbol{V}_{\text{blue}}$, each consisting of w of the $2w$ column blocks. We then alternate 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 $\boldsymbol{H}$ (cf. Fig. 2d). This is exploited as follows: Suppose that some node k runs subepoch l (say, blue). Node k now simultaneously fetches the block of $\boldsymbol{H}$ needed in the $(l+1)$-th subepoch (red) from the node that processed it in the $(l-1)$-th subepoch (also red). Thus communication and computation are overlayed.

Multithreading. We can exploit thread-level parallelism by using a $p \times 2p$ blocking of $\boldsymbol{V}$ (instead of $w \times 2w$). Each node then stores $2tp$ blocks of $\boldsymbol{V}$, t blocks of $\boldsymbol{W}$, and $2t$ blocks of $\boldsymbol{H}$. 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*). Only if the block is stored on some other node, we need to actually communicate data (*remote fetch*).

Locality-aware scheduling. A consequence of the distinction between local and remote fetches is that different stratum schedules have different communication cost, depending on the (expected) number of local fetches. SEQ is significantly more communication-efficient than WR/WOR since in every subepoch, only a single remote fetch occurs per node (the other $t - 1$ fetches are all local). However, the increased randomization of WOR leads to better convergence properties. We propose a locality-aware schedule (LA-WOR) that strikes a middleground between SEQ and WOR: The idea is to randomly group the column blocks of $\boldsymbol{V}_{\text{red}}$ (and independently $\boldsymbol{V}_{\text{blue}}$) into w groups. We then use a WOR schedule ($w \times w$) across groups and another WOR schedule ($t \times t$) within each group. For example, when $w = 2$ and $t = 2$, we may obtain the following schedule for $\boldsymbol{V}_{\text{red}}$:

$$\begin{array}{c} \\ S_1 \\ S_3 \\ S_5 \\ S_7 \end{array} \begin{array}{c} \text{Node 1} \mid \text{Node 2} \\ \begin{pmatrix} \mathbf{1} & \mathbf{4} & | & 3 & 2 \\ \mathbf{4} & \mathbf{1} & | & 2 & 3 \\ 2 & 3 & | & \mathbf{1} & \mathbf{4} \\ 3 & 2 & | & \mathbf{4} & \mathbf{1} \end{pmatrix} \end{array},$$

where rows correspond to red subepochs, columns to threads, and entries to blocks of $\boldsymbol{H}_{\text{red}}$. The first group (blocks 1 and 4) is printed bold, the vertical line indicates node boundaries. In this example, there are 10 local fetches (2+4+0+4) and 6 remote fetches.

E. Discussion

In what follows, we discuss the asymptotic performance of the distributed methods and issues arising in practice. The key question we try to answer is under which conditions distributed processing is effective. To simplify analysis, we assume that (1) computation and communication are not overlayed, (2) each revealed entry of $\boldsymbol{V}$ and each entry of $\boldsymbol{W}$ and $\boldsymbol{H}$ requires $O(1)$ words of memory and $O(1)$ time to communicate, (4) 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. DALS stores two copies of the input matrix as well as a copy of the entire factor matrices in memory; the total memory consumption is $O(N/w + mr)$ words. For ASGD, we only need to store entirely the smaller factor matrix $\boldsymbol{H}$, which gives $O((N+mr)/w+nr)$ words. Finally, DSGD++ fully partitions the factor matrices and thus requires $O((N + mr)/w)$ words. We conclude that DSGD++ is most memory-efficient, followed by ASGD, and then DALS.

Computation/communication trade-off. The overall performance of the distributed methods crucially depends on the relationship between computation and communication cost. We say that distributed processing is *effective* if the computational cost (reduced linearly by distributed processing) dominates the communication cost (increased). Under our assumptions, DALS requires $O(mr^3/w)$ time for computation and $O(mr)$ time for communication. 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. For DSGD-MR, $O(Nr/w)$ time is required for computation and $O(nr)$ time for communication (since we only communicate $\boldsymbol{H}$). 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 the key to determine how well we can distribute SGD. To see this, rewrite the computational cost to $O(\bar{N}nr/w)$ 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. Finally, observe that DSGD++ and ASGD do not satisfy assumption (1) above. Nevertheless, the analysis carries over to DSGD++ directly and to ASGD under the additional assumption that we average working copies at least once per epoch.

Practical considerations. The asymptotic analysis above may say little about the relative performance of the algorithms in practice, because (1) data size and number of nodes are finite, and (2) distributed processing may affect the number of epochs required to converge. In what follows, we assume that the data is small enough to fit into the memory of a single machine. Regarding (1), DALS significantly outperformed PALS in all our experiments, which agrees with our analysis. Similarly, we found that the relative time for a DSGD-MR or ASGD epoch compared to PSGD depends on $\bar{N}$, which also agrees with the above discussion. In contrast, DSGD++ is effective even for small values of $\bar{N}$, since computation and communication are overlayed. Regarding (2), DALS and ALS perform identically, since they solve exactly the same least-squares problems. Asymptotically, the same holds for the SGD-based approaches. In practice, however, ASGD, DSGD-MR, and DSGD++ may need more epochs to converge than PSGD: For DSGD-MR and DSGD++, stratification reduces the amount of randomness in training point selection, and the asynchronous averaging of ASGD introduces delay in broadcasting updates. Therefore, the time per epoch is not the sole indicator for overall performance; the effect on convergence properties needs to be taken into account as well. Our experiments below give more insight into the relative performance of each of the algorithms.

VI. Experimental Study

We compared all algorithms in an extensive experimental study 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). 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 less epochs to converge (and thus less scans of the input data), and *faster* if it needs less total time.

A. Overview of Results

Unless stated otherwise, all algorithms converged to a solution of similar loss (within 2% of each other). 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. In fact, DSGD++ was the only method that outperformed PSGD in some of our experiments with moderately-sized data (and it did so by a large margin). ASGD was faster than DSGD++ in one experiment with few nodes (and large $\bar{N}$), but its performance degrades significantly as more nodes are added. Thus ASGD, and to a lesser extent also DSGD-MR, was much more sensitive to communication overhead than DSGD++ and DALS. Finally, DALS was more data-efficient but less compute-efficient than the SGD-based methods. In terms of total time, it was competitive only when the rank r was small.

TABLE II: 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}	-

TABLE III: SGD step size sequence (Netflix, $r = 50$)

	Bold driver	Standard(1)	Standard(0.6)
Epochs	40	36	42
Loss (x10^7)	**7.936**	9.267	8.469

B. Experimental Setup

We implemented SGD, SGD++, ALS, PSGD, PALS, DSGD-MR, DSGD++, ASGD, and DALS in C++. All distributed algorithms used MPICH2[6] for communication. We used the GNU scientific library for solving the least-squares problems of ALS.[7] We ran our experiments on an 16-node cluster; each node was equipped with an Intel Xeon 2.40GHz processor with 8 cores and 48GB of RAM.

Statistics of the datasets used in our experiments are summarized in Table II. We used two real-world datasets: The Netflix dataset consists of roughly 99M ratings (1–5) of 480k Netflix users for 18k movies; it takes 2.2GB to store in main memory. The dataset of Track 1 of the KDD-Cup 2011 (termed KDD) consists of approximately 253M ratings of 1M Yahoo! Music users for 625k musical pieces; it takes 5.5 GB in memory. Note that Netflix and KDD differ significantly in the value of $\bar{N}$ (large for Netflix, small for KDD). For both datasets, we used the official validation sets and focus on L_{L2w} because it performs best in practice [1], [5], [9]. We did not tune the regularization parameter for varying choices of rank r but used the values given in Table II throughout.

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 $\boldsymbol{W}^*_{m\times 50}$ and $\boldsymbol{H}^*_{50\times n}$ with entries sampled independently from the Normal$(0, 10)$ distribution. We then obtained the data matrix

[6]http://www.mcs.anl.gov/mpi/mpich/

[7]In our experiments, GSL was significantly faster than LAPACK and, in contrast to LAPACK, also supports multi-threading.

TABLE IV: 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**

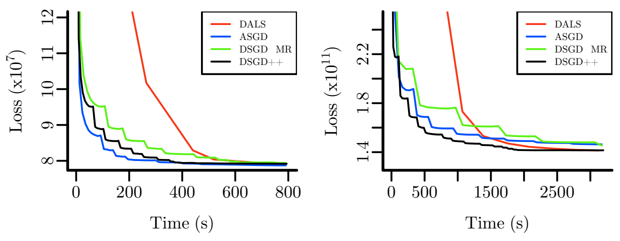
by sampling N random entries from $\boldsymbol{W}^*\boldsymbol{H}^*$ and adding Normal$(0, 1)$ noise. Note that the resulting datasets are very structured. We use them here to test the scalability of the various algorithms; the matrices can potentially be factored much more efficiently by exploiting their structure directly. To judge the impact of the shape of the data matrix on distributed processing, 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 factor Syn1B-rect (550M) than to factor 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.

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 [9]. The starting points $\boldsymbol{W}_0$ and $\boldsymbol{H}_0$ 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. 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. Unless stated otherwise, we used the bold driver heuristic for step size selection, which was thus fully automatic. 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 SGD++. For each algorithm, we declared convergence as soon as it reached a point within 2% of the overall best solution.

C. Sequential and Parallel Algorithms

We start with a discussion of sequential algorithms, which form a baseline for the parallel and distributed methods.

SGD step size sequence (Table III). In this experiment, we compared the performance of various step size sequences for SGD on the Netflix data for $r = 50$. In Table III, Standard(α) refers to a sequence of form ϵ_0/n^α, where n denotes the epoch and α is a parameter that controls the rate of decay; such sequences are commonly used in stochastic approximation. For this experiment only, we declared SGD as converged if its improvement in loss after one epoch falls below 0.1%. For SGD with the bold driver heuristic, we checked for convergence only in epochs following a drop in step size. We found that the bold driver heuristic significantly outperformed



(a) Netflix, $r = 100$ (b) KDD, $r = 100$

Fig. 4: Distributed algorithms on real-world datasets (4x8)

the standard sequences, even though it does not guarantee asymptotic convergence. For example, on Netflix, all step size sequences converged in roughly the same number of epochs, but the bold driver sequence converged to a significantly better factorization.

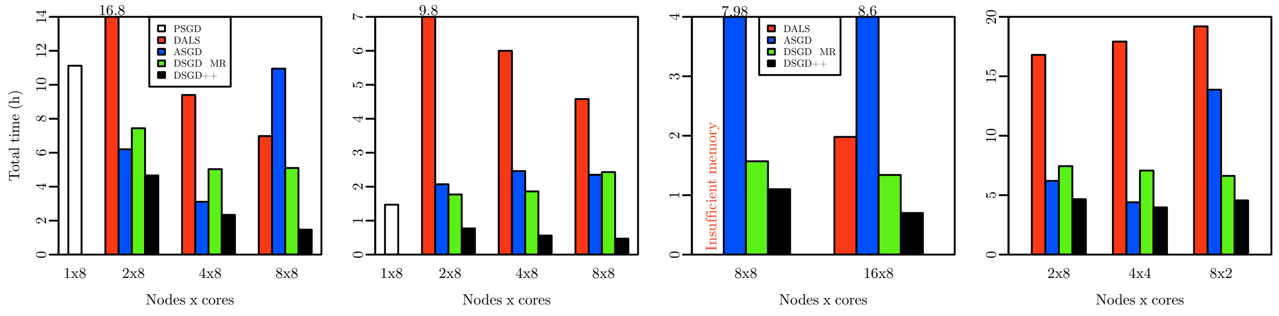
SGD, SGD++, ALS. We found that SGD++ is up to 13% more compute-efficient than SGD (7.4 vs. 8.4min for KDD, $r = 100$) so that prefetching is beneficial. When compared to ALS, SGD++ needs up to 5 times more epochs to converge (e.g., 31 vs. 6 epochs for Netflix, $r = 50$). However, SGD++ epochs are more compute-efficient so that SGD++ was faster overall. This effect is strongest when r is high; e.g., SGD++ is $\approx$5.5x faster for Netflix, $r = 100$ (52.8 vs. 290min) but only $\approx$1.1x faster for $r = 50$ (58 vs. 66min).

PSGD, PALS. We compared PSGD with PALS on a single node using 8 threads each (i.e., $w = 1$, $t = 8$). PALS was 7–8x faster than ALS and is equally data-efficient. Similarly, PSGD was 5x–6.8x faster than SGD; the speed-up was less than for ALS because memory latency became a bottleneck. On the KDD data (where the larger number of rows and columns slowed down ALS), PSGD converged significantly faster (e.g., 21 vs. 162min, $r = 100$).

D. Distributed Algorithms on Real Datasets

We ran the distributed algorithms on the Netflix and KDD datasets using 2 and 4 compute nodes, each running 8 threads; these setups are referred to as 2x8 and 4x8, respectively.

DSGD++ stratum schedule (Table IV). Recall that the DSGD++ stratum schedule affects both the time per epoch (governed by the fraction of local and remote fetches) and the number of epochs to convergence (governed by the amount of randomization). In Table IV, 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. Similarly to 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 (since $\bar{N}$ is large and thus communication costs are comparably 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 match our analysis of Sec. V-E; we use the WOR schedule for a large $\bar{N}$ and LA-WOR for small $\bar{N}$.



(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. 5: Performance of distributed algorithms on synthetic datasets

Netflix (Fig. 4a). Even though the Netflix dataset is only moderately large, we found that distributed processing can speed-up the factorization significantly; e.g., DALS provided 22x–29x speed-up on 4x8 (290 vs. 10min, $r = 100$). First, we observed that DSGD++ was up to 2.3x faster than DSGD-MR and $\approx$3x (4.6x–6.1x) more compute-efficient than PSGD on 2 nodes (4 nodes). These superlinear speed-ups arise because DSGD++ is more cache-friendly. However, DSGD++ was also less data-efficient than PSGD so that DSGD++ was only 1.7x–2.9x faster on 4 nodes overall. Second, ASGD was less compute-efficient than DSGD++ (12.6 vs. 5.9s, $r = 50$), the latter due to the cost of averaging of working copies. The data-efficiency of ASGD dropped significantly as we increased the number of nodes (25 epochs on 2x8, 60 epochs on 4x8, $r = 50$). This happens because more working copies need to be averaged when we use more nodes; the delay of propagating updates thus increases. For example, we ran on average 65 rounds of averaging per epoch on 2x8, but only 15 rounds on 4x8 (less time per epoch, more nodes to synchronize). Nevertheless, ASGD outperformed all other methods on 2 nodes and $r = 100$. Finally, DALS was the fastest method for $r = 50$ but was outperformed by DSGD++ for higher ranks (5.8 vs. 10.7 min, 4x8, $r = 100$).

KDD (Fig. 4b). For the KDD dataset, DALS provided similar speed-up as on the Netflix data and was the overall fastest method for $r = 50$ (with 4 nodes). In contrast, the compute-efficiency of DSGD-MR and ASGD was penalized since $\bar{N}$ is small for KDD. This effect was most pronounced for ASGD (51min on 2 nodes, 125min on 4 nodes, $r = 100$). DSGD++ remained unaffected since communication and computation are overlayed. In all cases, the distributed SGD-based methods were less data-efficient than PSGD, which was faster overall.[8]

E. Large-Scale Experiments

For our large-scale experiments, we used the Syn1B-rect, Syn1B-sq, and Syn10B datasets and varied the number of compute nodes (2–16) and threads per node (2–8). Our results are summarized in Fig. 5. The plots show the total time to convergence required by each of the distributed methods and, if possible, the parallel 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. For DSGD++, we employed the LA-WOR schedule on Syn1B-sq and Syn1B-rect, and the WOR schedule on Syn10B.

Syn1B-rect (Fig. 5a). In our first experiment, we used Syn1B-rect on the 1x8, 2x8, 4x8, and 8x8 setups. First, we observed that DALS did not converge to an acceptable solution (4 orders of magnitude larger loss than all other methods); we did not see this erratic behavior on any other dataset. In Fig. 5a, we thus give the running time of DALS until its loss changed by less then 0.1% in two consecutive epochs. In any case, DALS was 44% faster per epoch on 4x8 than on 2x8, and 25% faster on 8x8 than on 4x8. This sublinear speed-up indicates that the time required to broadcast the factor matrices becomes significant as we increase the number of nodes. Second, DSGD-MR performed better than the PSGD baseline in all cases (1.5x speed-up on 2 nodes, 2.2x on 4 nodes), but the communication and synchronization penalties of DSGD-MR let its performance deteriorate slightly as we went beyond 4 nodes. The former matches our expectation: Syn1B-rect has a high value of $\bar{N}$ and is thus "easy" to distribute. Third, ASGD performed better on 2 and 4 nodes (1.8x and 3.6x faster then PSGD, resp.), but the runtime on ASGD increased significantly when we moved to 8 nodes. Even though ASGD was more compute-efficient on 8 nodes (90s vs. 57s per epoch), the increased synchronization overhead and consequently increased delay between parameter updates drastically reduced data efficiency (110 vs. 625 epochs). In contrast to DSGD-MR, ASGD thus does not behave gracefully if too many nodes are used. Finally, DSGD++ performed best in all setups. It was 2.4x faster than PSGD on 2 nodes and 4.7x faster on 4 nodes. Similar to our experiments on real datasets, we saw these superlinear speed-ups due to better cache locality. The increased communication overhead did not affect performance because of asynchronous communication and our use of the LA-WOR schedule. On 8 nodes, the overhead of communication starts to be visible (7.5x speed-up). Nevertheless, DSGD++ was able to factor Syn1B-rect in 88 minutes (8 nodes); its closest competitor was ASGD with 186 minutes (4 nodes).

[8]This drop might be related to the specifics of the KDD dataset and our choice of loss. For example, when using L_{L2}, we found that DSGD++ achieved a 3x overall speed-up on 4 nodes, $r = 50$, when compared to PSGD.

Syn1B-sq (Fig. 5a). On Syn1B-sq, all methods were faster than on Syn1B-rect since there were less factors to learn. First, we observed that DALS (now working correctly) was consistently slower than the PSGD baseline. As before, increasing the number of nodes leads to sublinear speed-up due to increased communication overhead. Second, neither ASGD nor DSGD-MR are able to improve on the PSGD baseline. In fact, Syn1B-sq is our "hard" dataset (low $\bar{N}$) so that communication overheads dominate potential gains due to distributed processing. 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 less SGD updates are run per column and time unit (but, as before, the time per epoch decreased and the number of epochs increased). Finally, DSGD++ was the only method that was able to improve upon PSGD. It achieved speed-ups of 1.6x (2 nodes), 2.3x (4 nodes), and 3.5x (8 nodes). As expected, the speed-ups are lower than the ones for Syn1B-rect but nevertheless significant. This indicates that DSGD++ is the only SGD-based method that can handle matrices with low $\bar{N}$ gracefully.

Syn10B (Fig. 5c). We could not run experiments on Syn10B using four or less nodes due to insufficient aggregate memory. We thus only show results for 8x8 and 16x8. First, note that we cannot run DALS even on 8 nodes since the available memory is insufficient to store the two copies of data matrix and a full copy of the factor matrix simultaneously (DALS would require at least 11 nodes). On 16 nodes, DALS took 2h to converge; the increased density of the Syn10B matrix simplifies the completion problem so that only 11 epochs were needed to converge. Second, all 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. Third, DSGD-MR took 1.6h on 8x8 and 1.34h on 16x8. Thus DSGD-MR is faster on 8 nodes than DALS on 16 nodes. Next, 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++ is faster on 8 nodes than any other method on 16 nodes, and is almost twice as fast on 16 nodes than its closest competitor (DSGD-MR).

Impact of distributed processing (Fig. 5d). In our final experiment on Syn1B-rect, we investigate the behavior of the various algorithms as we increase the number of nodes while keeping the overall number of processors constant (2x8, 4x4, 8x2). Since the computational cost is identical in each setup, this allows 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 discussion above).

VII. Conclusion

We proposed the distributed DALS, ASGD, and DSGD++ algorithms for in-memory matrix completion on a small cluster of commodity nodes. Our algorithms exploit thread-level parallelism, in-memory processing, and asynchronous communication, and scale to matrices with millions of rows, millions of columns, and billions of entries. In our large-scale experiments, DSGD++ performed best. It consistently outperformed alternative approaches with respect to speed, scalability, and memory footprint.

References

[1] P.-L. Chen et al, "A linear ensemble of individual and blended models for music rating prediction," in *KDDCup 2011 Workshop*, 2011.

[2] S. Das, Y. Sismanis, K. S. Beyer, R. Gemulla, P. J. Haas, and J. McPherson, "Ricardo: Integrating R and Hadoop," in *SIGMOD*, 2010, pp. 987–998.

[3] R. Gemulla, E. Nijkamp, P. J. Haas, and Y. Sismanis, "Large-scale matrix factorization with distributed stochastic gradient descent," in *SIGKDD*, 2011, pp. 69–77. [Online]. Available: http://dl.acm.org/citation.cfm?id=2020426

[4] Y. Hu, Y. Koren, and C. Volinsky, "Collaborative filtering for implicit feedback datasets," in *ICDM*, 2008, pp. 263–272.

[5] Y. Koren, R. Bell, and C. Volinsky, "Matrix factorization techniques for recommender systems," *IEEE Computer*, vol. 42, no. 8, pp. 30–37, 2009.

[6] L. Mackey, A. Talwalkar, and M. Jordan, "Divide-and-conquer matrix factorization," in *NIPS*, 2011.

[7] F. Niu, B. Recht, C. Ré, and S. J. Wright, "Hogwild!: A lock-free approach to parallelizing stochastic gradient descent," in *NIPS*, 2011.

[8] B. Recht and C. Ré, "Parallel stochastic gradient algorithms for Large-Scale matrix completion," *Optimization Online*, 2011. [Online]. Available: http://www.optimization-online.org/DB_HTML/2011/04/3012.html

[9] Y. Zhou, D. Wilkinson, R. Schreiber, and R. Pan, "Large-scale parallel collaborative filtering for the Netflix Prize," in *AAIM*, 2008, pp. 337–348.

[10] J. Bennett and S. Lanning, "The Netflix prize," in *KDD Cup and Workshop*, 2007.

[11] G. Dror, N. Koenigstein, Y. Koren, and M. Weimer, "The Yahoo! Music Dataset and KDD-Cup'11," in *KDDCup 2011 Workshop*, 2011.

[12] A. S. Das, M. Datar, A. Garg, and S. Rajaram, "Google news personalization: scalable online collaborative filtering," in *WWW*, 2007, pp. 271–280.

[13] C. Liu, H.-c. Yang, J. Fan, L.-W. He, and Y.-M. Wang, "Distributed nonnegative matrix factorization for web-scale dyadic data analysis on mapreduce," in *WWW*, 2010, pp. 681–690.

[14] H. J. Kushner and G. Yin, *Stochastic Approximation and Recursive Algorithms and Applications*, 2nd ed. Springer, 2003.

[15] A. Smola and S. Narayanamurthy, "An architecture for parallel topic models," *PVLDB*, vol. 3, no. 1-2, pp. 703–710, 2010.

[16] E. J. Cands and B. Recht, "Exact matrix completion via convex optimization." *Foundations of Computational Mathematics*, vol. 9, no. 6, pp. 717–772, 2009. [Online]. Available: http://dblp.uni-trier.de/db/journals/focm/focm9.html#CandesR09

[17] C. T. Chu, S. K. Kim, Y. A. Lin, Y. Y. Yu, G. Bradski, A. Y. Ng, and K. Olukotun, "Map-reduce for machine learning on multicore," in *NIPS*, 2006, pp. 281–288.

[18] R. H. Byrd, P. Lu, J. Nocedal, and C. Zhu, "A limited memory algorithm for bound constrained optimization," *SIAM J. Sci. Comput.*, vol. 16, no. 5, pp. 1190–1208, 1995.

[19] R. Battiti, "Accelerated backpropagation learning: Two optimization methods," *Complex Systems*, vol. 3, pp. 331–342, 1989.

[20] R. McDonald, K. Hall, and G. Mann, "Distributed training strategies for the structured perceptron," in *Human Language Technologies*, 2010, pp. 456–464. [Online]. Available: http://www.aclweb.org/anthology-new/N/N10/N10-1069.bib

[21] J. Tsitsiklis, D. Bertsekas, and M. Athans, "Distributed asynchronous deterministic and stochastic gradient optimization algorithms," *IEEE Transactions on Automatic Control*, vol. 31, no. 9, pp. 803– 812, 1986.