

# NOMAD: Non-locking, stOchastic Multi-machine algorithm for Asynchronous and Decentralized matrix completion

Hyokun Yun  
Purdue University  
yun3@purdue.edu

Hsiang-Fu Yu  
University of Texas, Austin  
rofuyu@cs.utexas.edu

Cho-Jui Hsieh  
University of Texas, Austin  
cjhsieh@cs.utexas.edu

S V N Vishwanathan  
Purdue University  
vishy@stat.purdue.edu

Inderjit Dhillon  
University of Texas, Austin  
inderjit@cs.utexas.edu

## ABSTRACT

We develop an efficient parallel distributed algorithm for matrix completion, named NOMAD (Non-locking, stOchastic Multi-machine algorithm for Asynchronous and Decentralized matrix completion). NOMAD is a decentralized algorithm with non-blocking communication between processors. One of the key features of NOMAD is that the ownership of a variable is asynchronously transferred between processors in a decentralized fashion. As a consequence it is a lock-free parallel algorithm. In spite of being asynchronous, the variable updates of NOMAD are serializable, that is, there is an equivalent update ordering in a serial implementation. NOMAD outperforms synchronous algorithms which require explicit bulk synchronization after every iteration: our extensive empirical evaluation shows that not only does our algorithm perform well in distributed setting on commodity hardware, but also outperforms state-of-the-art algorithms on a HPC cluster both in multi-core and distributed memory settings.

## 1. INTRODUCTION

The aim of this paper is to develop an efficient parallel distributed algorithm for matrix completion. We are specifically interested in solving large industrial scale matrix completion problems on commodity hardware with limited computing power, memory, and interconnect speed, such as the ones found in data centers. The widespread availability of cloud computing platforms such as Amazon Web Services (AWS) make the deployment of such systems feasible.

However, existing algorithms for matrix completion are designed for conventional high performance computing (HPC) platforms. In order to deploy them on commodity hardware we need to employ a large number of machines, which increases inter-machine communication. Since the network bandwidth in data centers is significantly lower and less-

reliable than the high-speed interconnects typically found in HPC hardware, this can often have disastrous consequences in terms of convergence speed or the quality of the solution.

In this paper, we present NOMAD (Non-locking, stOchastic Multi-machine algorithm for Asynchronous and Decentralized matrix completion), a new parallel algorithm for matrix completion with the following properties:

- Non-blocking communication: Processors exchange messages in an asynchronous fashion [6], and there is no bulk synchronization.
- Decentralized: Processors are symmetric to each other, and each processor does the same amount of computation and communication.
- Lock free: Using an *owner computes* paradigm, we completely eliminate the need for locking variables.
- Fully asynchronous computation: Because of the lock free nature of our algorithm, the variable updates in individual processors are fully asynchronous.
- Serializability: There is an equivalent update ordering in a serial implementation. In our algorithm *stale* parameters are never used and this empirically leads to faster convergence [17].

Our extensive empirical evaluation shows that not only does our algorithm perform well in distributed setting on commodity hardware, but also outperforms state-of-the-art algorithms on a HPC cluster both in multi-core and distributed memory settings. We show that our algorithm is significantly better than existing multi-core and multi-machine algorithms for the matrix completion problem.

This paper is organized as follows: Section 2 establishes some notation and introduces the matrix completion problem formally. Section 3 is devoted to describing NOMAD. We contrast NOMAD with existing work in Section 4. In Section 5 we present extensive empirical comparison of NOMAD with various existing algorithms. Section 6 concludes the paper with a discussion.

## 2. BACKGROUND

Let  $A \in \mathbb{R}^{m \times n}$  be a rating matrix, where  $m$  denotes the number of users and  $n$  the number of items. Typically  $m \gg n$ , although the algorithms we consider in this paper do not depend on such an assumption. Furthermore, let

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivs 3.0 Unported License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc-nd/3.0/>. Obtain permission prior to any use beyond those covered by the license. Contact copyright holder by emailing [info@vldb.org](mailto:info@vldb.org). Articles from this volume were invited to present their results at the 40th International Conference on Very Large Data Bases, September 1st - 5th 2014, Hangzhou, China. *Proceedings of the VLDB Endowment*, Vol. 7, No. 11  
Copyright 2014 VLDB Endowment 2150-8097/14/07.

$\Omega \subseteq \{1 \dots m\} \times \{1, \dots, n\}$  denote the observed entries of  $A$ , that is,  $(i, j) \in \Omega$  implies that user  $i$  gave item  $j$  a rating of  $A_{ij}$ . The goal here is to predict accurately the unobserved ratings. For convenience, we define  $\Omega_i$  to be the set of items rated by the  $i$ -th user, i.e.,  $\Omega_i := \{j : (i, j) \in \Omega\}$ . Analogously  $\bar{\Omega}_j := \{i : (i, j) \in \Omega\}$  is the set of users who have rated item  $j$ . Also, let  $\mathbf{a}_i^\top$  denote the  $i$ -th row of  $A$ .

One popular model for matrix completion finds matrices  $W \in \mathbb{R}^{m \times k}$  and  $H \in \mathbb{R}^{n \times k}$ , with  $k \ll \min(m, n)$ , such that  $A \approx WH^\top$ . One way to understand this model is to realize that each row  $\mathbf{w}_i^\top \in \mathbb{R}^k$  of  $W$  can be thought of as a  $k$ -dimensional embedding of the user. Analogously, each row  $\mathbf{h}_j^\top \in \mathbb{R}^k$  of  $H$  is an embedding of the item in the same  $k$ -dimensional space. In order to predict the  $(i, j)$ -th entry of  $A$  we simply use  $\langle \mathbf{w}_i, \mathbf{h}_j \rangle$ , where  $\langle \cdot, \cdot \rangle$  denotes the Euclidean inner product of two vectors. The goodness of fit of the model is measured by a loss function. While our optimization algorithm can work with an arbitrary separable loss, for ease of exposition we will only discuss the square loss:  $\frac{1}{2}(A_{ij} - \langle \mathbf{w}_i, \mathbf{h}_j \rangle)^2$ . Furthermore, we need to enforce regularization to prevent over-fitting, and to predict well on the unknown entries of  $A$ . Again, a variety of regularizers can be handled by our algorithm, but we will only focus on the following weighted square norm-regularization in this paper:  $\frac{\lambda}{2} \sum_{i=1}^m |\Omega_i| \cdot \|\mathbf{w}_i\|^2 + \frac{\lambda}{2} \sum_{j=1}^n |\bar{\Omega}_j| \cdot \|\mathbf{h}_j\|^2$ , where  $\lambda > 0$  is a regularization parameter. Here,  $|\cdot|$  denotes the cardinality of a set, and  $\|\cdot\|^2$  is the  $L_2$  norm of a vector. Putting everything together yields the following objective function:

$$\min_{\substack{W \in \mathbb{R}^{m \times k} \\ H \in \mathbb{R}^{n \times k}}} J(W, H) := \frac{1}{2} \sum_{(i,j) \in \Omega} (A_{ij} - \langle \mathbf{w}_i, \mathbf{h}_j \rangle)^2 + \frac{\lambda}{2} \left( \sum_{i=1}^m |\Omega_i| \cdot \|\mathbf{w}_i\|^2 + \sum_{j=1}^n |\bar{\Omega}_j| \cdot \|\mathbf{h}_j\|^2 \right). \quad (1)$$

This can be further simplified and written as

$$J(W, H) = \frac{1}{2} \sum_{(i,j) \in \Omega} \{ (A_{ij} - \langle \mathbf{w}_i, \mathbf{h}_j \rangle)^2 + \lambda (\|\mathbf{w}_i\|^2 + \|\mathbf{h}_j\|^2) \}.$$

In the above equations,  $\lambda > 0$  is a scalar which trades off the loss function with the regularizer.

In the sequel we will let  $w_{il}$  and  $h_{il}$  for  $1 \leq l \leq k$  denote the  $l$ -th coordinate of the column vectors  $\mathbf{w}_i$  and  $\mathbf{h}_j$ , respectively. Furthermore,  $H_{\Omega_i}$  (resp.  $W_{\bar{\Omega}_j}$ ) will be used to denote the sub-matrix of  $H$  (resp.  $W$ ) formed by collecting rows corresponding to  $\Omega_i$  (resp.  $\bar{\Omega}_j$ ).

Note the following property of the above objective function (1): If we fix  $H$  then the problem decomposes to  $m$  independent convex optimization problems, each of which has the following form:

$$\min_{\mathbf{w}_i \in \mathbb{R}^k} J_i(\mathbf{w}_i) = \frac{1}{2} \sum_{j \in \Omega_i} (A_{ij} - \langle \mathbf{w}_i, \mathbf{h}_j \rangle)^2 + \lambda \|\mathbf{w}_i\|^2. \quad (2)$$

Analogously, if we fix  $W$  then (1) decomposes into  $n$  independent convex optimization problems, each of which has the following form:

$$\min_{\mathbf{h}_j \in \mathbb{R}^k} \bar{J}_j(\mathbf{h}_j) = \frac{1}{2} \sum_{i \in \bar{\Omega}_j} (A_{ij} - \langle \mathbf{w}_i, \mathbf{h}_j \rangle)^2 + \lambda \|\mathbf{h}_j\|^2.$$

The gradient and Hessian of  $J_i(\mathbf{w})$  can be easily computed:

$$\nabla J_i(\mathbf{w}_i) = M\mathbf{w}_i - \mathbf{b}, \text{ and } \nabla^2 J_i(\mathbf{w}_i) = M,$$

where we have defined  $M := H_{\Omega_i}^\top H_{\Omega_i} + \lambda I$  and  $\mathbf{b} := H^\top \mathbf{a}_i$ .

We will now present three well known optimization strategies for solving (1), which essentially differ in only two characteristics namely, the sequence in which updates to the variables in  $W$  and  $H$  are carried out, and the level of approximation in the update.

## 2.1 Alternating Least Squares

A simple version of the Alternating Least Squares (ALS) algorithm updates variables as follows:  $\mathbf{w}_1, \mathbf{w}_2, \dots, \mathbf{w}_m, \mathbf{h}_1, \mathbf{h}_2, \dots, \mathbf{h}_n, \mathbf{w}_1, \dots$  and so on. Updates to  $\mathbf{w}_i$  are computed by solving (2) which is in fact a least squares problem, and thus the following Newton update gives us:

$$\mathbf{w}_i \leftarrow \mathbf{w}_i - [\nabla^2 J_i(\mathbf{w}_i)]^{-1} \nabla J_i(\mathbf{w}_i), \quad (3)$$

which can be rewritten using  $M$  and  $\mathbf{b}$  as  $\mathbf{w}_i \leftarrow M^{-1}\mathbf{b}$ . Updates to  $\mathbf{h}_j$ 's are analogous.

## 2.2 Coordinate Descent

The ALS update involves formation of the Hessian and its inversion. In order to reduce the computational complexity, one can replace the Hessian by its diagonal approximation:

$$\mathbf{w}_i \leftarrow \mathbf{w}_i - [\text{diag}(\nabla^2 J_i(\mathbf{w}_i))]^{-1} \nabla J_i(\mathbf{w}_i), \quad (4)$$

which can be rewritten using  $M$  and  $\mathbf{b}$  as

$$\mathbf{w}_i \leftarrow \mathbf{w}_i - \text{diag}(M)^{-1} [M\mathbf{w}_i - \mathbf{b}]. \quad (5)$$

If we update one component of  $\mathbf{w}_i$  at a time, the update (5) can be written as:

$$w_{il} \leftarrow w_{il} - \frac{\langle \mathbf{m}_l, \mathbf{w}_i \rangle - b_l}{m_{ll}}, \quad (6)$$

where  $\mathbf{m}_l$  is  $l$ -th row of matrix  $M$ ,  $b_l$  is  $l$ -th component of  $\mathbf{b}$  and  $m_{ll}$  is the  $l$ -th coordinate of  $\mathbf{m}_l$ .

If we choose the update sequence  $w_{11}, \dots, w_{1k}, w_{21}, \dots, w_{2k}, \dots, w_{m1}, \dots, w_{mk}, h_{11}, \dots, h_{1k}, h_{21}, \dots, h_{2k}, \dots, h_{n1}, \dots, h_{nk}, w_{11}, \dots, w_{1k}$ , and so on, then this recovers Cyclic Coordinate Descent (CCD) [15]. On the other hand, the update sequence  $w_{11}, \dots, w_{m1}, h_{11}, \dots, h_{n1}, w_{12}, \dots, w_{m2}, h_{12}, \dots, h_{n2}$  and so on, recovers the CCD++ algorithm of Yu et al. [26]. The CCD++ updates can be performed more efficiently than the CCD updates by maintaining a residual matrix [26].

## 2.3 Stochastic Gradient Descent

The stochastic gradient descent (SGD) algorithm for matrix completion can be motivated from its more classical version, gradient descent. Given an objective function  $f(\theta) = \frac{1}{m} \sum_{i=1}^m f_i(\theta)$ , the gradient descent update is

$$\theta \leftarrow \theta - s_t \cdot \nabla_\theta f(\theta), \quad (7)$$

where  $t$  denotes the iteration number and  $\{s_t\}$  is a sequence of step sizes. The stochastic gradient descent update replaces  $\nabla_\theta f(\theta)$  by its unbiased estimate  $\nabla_\theta f_i(\theta)$ , which yields

$$\theta \leftarrow \theta - s_t \cdot \nabla_\theta f_i(\theta). \quad (8)$$

It is significantly cheaper to evaluate  $\nabla_\theta f_i(\theta)$  as compared to  $\nabla_\theta f(\theta)$ . One can show that for sufficiently large  $t$ , the above updates will converge to a fixed point of  $f$  [16, 21]. The above update also enjoys desirable properties in terms of sample complexity, and hence is widely used in machine learning [7, 22].

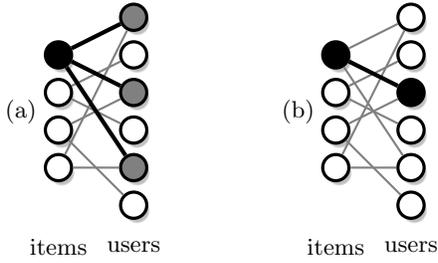


Figure 1: Illustration of updates used in matrix completion. Three algorithms are shown here: (a) alternating least squares and coordinate descent, (b) stochastic gradient descent. Black indicates that the value of the node is being updated, gray indicates that the value of the node is being read. White nodes are neither being read nor updated.

For the matrix completion problem, note that for a fixed  $(i, j)$  pair, the gradient of (1) can be written as

$$\nabla_{\mathbf{w}_i} J(W, H) = (A_{ij} - \langle \mathbf{w}_i, \mathbf{h}_j \rangle) \mathbf{h}_j + \lambda \mathbf{w}_i \text{ and}$$

$$\nabla_{\mathbf{h}_j} J(W, H) = (A_{ij} - \langle \mathbf{w}_i, \mathbf{h}_j \rangle) \mathbf{w}_i + \lambda \mathbf{h}_j,$$

Therefore the SGD updates require sampling a random index  $(i_t, j_t)$  uniformly from the set of nonzero indices  $\Omega$ , and performing the update

$$\mathbf{w}_{i_t} \leftarrow \mathbf{w}_{i_t} - s_t \cdot [(A_{i_t j_t} - \mathbf{w}_{i_t} \mathbf{h}_{j_t}) \mathbf{h}_{j_t} + \lambda \mathbf{w}_{i_t}] \text{ and} \quad (9)$$

$$\mathbf{h}_{j_t} \leftarrow \mathbf{h}_{j_t} - s_t \cdot [(A_{i_t j_t} - \mathbf{w}_{i_t} \mathbf{h}_{j_t}) \mathbf{w}_{i_t} + \lambda \mathbf{h}_{j_t}]. \quad (10)$$

### 3. NOMAD

In NOMAD, we use an optimization scheme based on SGD. In order to justify this choice, we find it instructive to first understand the updates performed by ALS, coordinate descent, and SGD on a bipartite graph which is constructed as follows: the  $i$ -th user node corresponds to  $\mathbf{w}_i$ , the  $j$ -th item node corresponds to  $\mathbf{h}_j$ , and an edge  $(i, j)$  indicates that user  $i$  has rated item  $j$  (see Figure 1). Both the ALS update (3) and coordinate descent update (6) for  $\mathbf{w}_i$  require us to access the values of  $\mathbf{h}_j$  for all  $j \in \Omega_i$ . This is shown in Figure 1 (a), where the black node corresponds to  $\mathbf{w}_i$ , while the gray nodes correspond to  $\mathbf{h}_j$  for  $j \in \Omega_i$ . On the other hand, the SGD update to  $\mathbf{w}_i$  (9) only requires us to retrieve the value of  $\mathbf{h}_j$  for a single random  $j \in \Omega_i$  (Figure 1 (b)). What this means is that in contrast to ALS or CCD, multiple SGD updates can be carried out simultaneously in parallel, without interfering with each other. Put another way, SGD has higher potential for finer-grained parallelism than other approaches, and therefore we use it as our optimization scheme in NOMAD.

#### 3.1 Description

For now, we will denote each parallel computing unit as a *worker*; in a shared memory setting a worker is a thread and in a distributed memory architecture a worker is a machine. This abstraction allows us to present NOMAD in a unified manner. Of course, NOMAD can be used in a hybrid setting where there are multiple threads spread across multiple machines, and this will be discussed in Section 3.4.

In NOMAD, the users  $\{1, 2, \dots, m\}$  are split into  $p$  disjoint sets  $I_1, I_2, \dots, I_p$  which are of approximately equal size<sup>1</sup>. This induces a partition of the rows of the ratings matrix  $A$ . The  $q$ -th worker stores  $n$  sets of indices  $\bar{\Omega}_j^{(q)}$ , for  $j \in \{1, \dots, n\}$ , which are defined as

$$\bar{\Omega}_j^{(q)} := \{(i, j) \in \bar{\Omega}_j; i \in I_q\},$$

as well as the corresponding values of  $A$ . Note that once the data is partitioned and distributed to the workers, it is never moved during the execution of the algorithm.

Recall that there are two types of parameters in matrix completion: user parameters  $\mathbf{w}_i$ 's, and item parameters  $\mathbf{h}_j$ 's. In NOMAD,  $\mathbf{w}_i$ 's are partitioned according to  $I_1, I_2, \dots, I_p$ , that is, the  $q$ -th worker stores and updates  $\mathbf{w}_i$  for  $i \in I_q$ . The variables in  $W$  are partitioned at the beginning, and never move across workers during the execution of the algorithm. On the other hand, the  $\mathbf{h}_j$ 's are split randomly into  $p$  partitions at the beginning, and their ownership changes as the algorithm progresses. At each point of time an  $\mathbf{h}_j$  variable resides in one and only worker, and it moves to another worker after it is processed, independent of other item variables. Hence these are *nomadic* variables<sup>2</sup>.

Processing an item variable  $\mathbf{h}_j$  at the  $q$ -th worker entails executing SGD updates (9) and (10) on the ratings in the set  $\bar{\Omega}_j^{(q)}$ . Note that these updates only require access to  $\mathbf{h}_j$  and  $\mathbf{w}_i$  for  $i \in I_q$ ; since  $I_q$ 's are disjoint, each  $\mathbf{w}_i$  variable in the set is accessed by only one worker. This is why the communication of  $\mathbf{w}_i$  variables is not necessary. On the other hand,  $\mathbf{h}_j$  is updated only by the worker that currently owns it, so there is no need for a lock; this is the popular *owner-computes* rule in parallel computing. See Figure 2.

We now formally define the NOMAD algorithm (see Algorithm 1 for detailed pseudo-code). Each worker  $q$  maintains its own concurrent queue, `queue`[ $q$ ], which contains a list of items it has to process. Each element of the list consists of the index of the item  $j$  ( $1 \leq j \leq n$ ), and a corresponding  $k$ -dimensional parameter vector  $\mathbf{h}_j$ ; this pair is denoted as  $(j, \mathbf{h}_j)$ . Each worker  $q$  pops a  $(j, \mathbf{h}_j)$  pair from its own queue, `queue`[ $q$ ], and runs stochastic gradient descent update on  $\bar{\Omega}_j^{(q)}$ , which is the set of ratings on item  $j$  locally stored in worker  $q$  (line 16 to 21). This changes values of  $\mathbf{w}_i$  for  $i \in I_q$  and  $\mathbf{h}_j$ . After all the updates on item  $j$  are done, a uniformly random worker  $q'$  is sampled (line 22) and the updated  $(j, \mathbf{h}_j)$  pair is pushed into the queue of that worker,  $q'$  (line 23). Note that this is the only time where a worker communicates with another worker. Also note that the nature of this communication is asynchronous and non-blocking. Furthermore, as long as there are items in the queue, the computations are completely asynchronous and decentralized. Moreover, all workers are symmetric, that is, there is no designated master or slave.

#### 3.2 Complexity Analysis

First, we consider the case when the problem is distributed across  $p$  workers, and study how the space and time complexity behaves as a function of  $p$ . Each worker has to store

<sup>1</sup>An alternative strategy is to split the users such that each set has approximately the same number of ratings.

<sup>2</sup>Due to symmetry in the formulation of the matrix completion problem, one can also make the  $\mathbf{w}_i$ 's nomadic and partition the  $\mathbf{h}_j$ 's. Since usually the number of users is much larger than the number of items, this leads to more communication and therefore we make the  $\mathbf{h}_j$  variables nomadic.

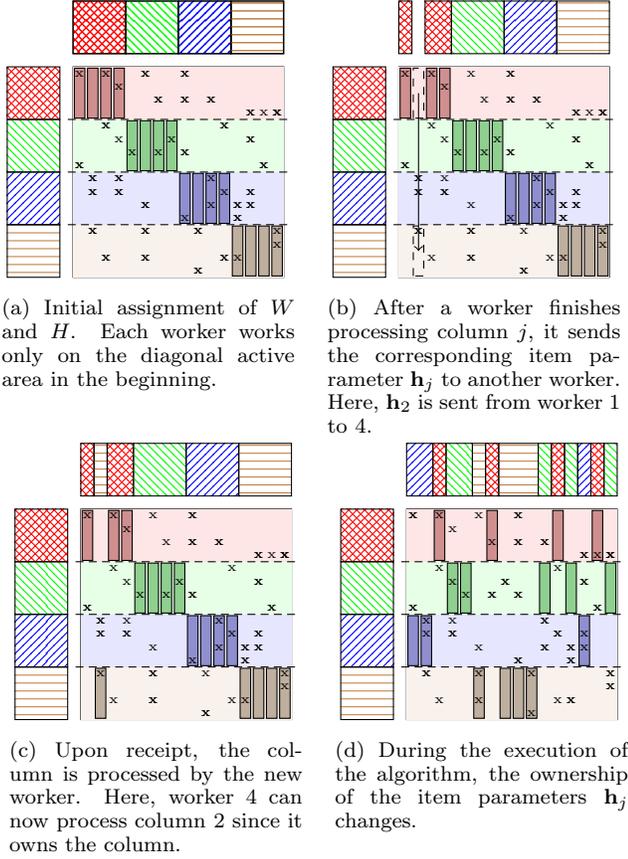


Figure 2: Illustration of the NOMAD algorithm

---

**Algorithm 1** the basic NOMAD algorithm

---

```

1:  $\lambda$ : regularization parameter
2:  $\{s_t\}$ : step size sequence
3: // initialize parameters
4:  $w_{il} \sim \text{UniformReal}\left(0, \frac{1}{\sqrt{k}}\right)$  for  $1 \leq i \leq m, 1 \leq l \leq k$ 
5:  $h_{jl} \sim \text{UniformReal}\left(0, \frac{1}{\sqrt{k}}\right)$  for  $1 \leq j \leq n, 1 \leq l \leq k$ 
6: // initialize queues
7: for  $j \in \{1, 2, \dots, n\}$  do
8:    $q \sim \text{UniformDiscrete}\{1, 2, \dots, p\}$ 
9:    $\text{queue}[q].\text{push}(j, \mathbf{h}_j)$ 
10: end for
11: // start  $p$  workers
12: Parallel Foreach  $q \in \{1, 2, \dots, p\}$ 
13:   while stop signal is not yet received do
14:     if  $\text{queue}[q]$  not empty then
15:        $(j, \mathbf{h}_j) \leftarrow \text{queue}[q].\text{pop}()$ 
16:       for  $(i, j) \in \bar{\Omega}_j^{(q)}$  do
17:         // SGD update
18:          $t \leftarrow$  number of updates on  $(i, j)$ 
19:          $\mathbf{w}_i \leftarrow \mathbf{w}_i - s_t \cdot [(A_{ij} - \mathbf{w}_i \mathbf{h}_j) \mathbf{h}_j + \lambda \mathbf{w}_i]$ 
20:          $\mathbf{h}_j \leftarrow \mathbf{h}_j - s_t \cdot [(A_{ij} - \mathbf{w}_i \mathbf{h}_j) \mathbf{w}_j + \lambda \mathbf{h}_j]$ 
21:       end for
22:        $q' \sim \text{UniformDiscrete}\{1, 2, \dots, p\}$ 
23:        $\text{queue}[q'].\text{push}(j, \mathbf{h}_j)$ 
24:     end if
25:   end while
26: Parallel End

```

---

$1/p$  fraction of the  $m$  user parameters, and approximately  $1/p$  fraction of the  $n$  item parameters. Furthermore, each worker also stores approximately  $1/p$  fraction of the  $|\Omega|$  ratings. Since storing a row of  $W$  or  $H$  requires  $O(k)$  space the space complexity per worker is  $O((mk + nk + |\Omega|)/p)$ . As for time complexity, we find it useful to use the following assumptions: performing the SGD updates in line 16 to 21 takes  $a \cdot k$  time and communicating a  $(j, \mathbf{h}_j)$  to another worker takes  $c \cdot k$  time, where  $a$  and  $c$  are hardware dependent constants. On the average, each  $(j, \mathbf{h}_j)$  pair contains  $O(|\Omega|/np)$  non-zero entries. Therefore when a  $(j, \mathbf{h}_j)$  pair is popped from  $\text{queue}[q]$  in line 15 of Algorithm 1, on the average it takes  $a \cdot (|\Omega|k/np)$  time to process the pair. Since computation and communication can be done in parallel, as long as  $a \cdot (|\Omega|k/np)$  is higher than  $c \cdot k$  a worker thread is always busy and NOMAD scales linearly.

Suppose that  $|\Omega|$  is fixed but the number of workers  $p$  increases; that is, we take a fixed size dataset and distribute it across  $p$  workers. As expected, for a large enough value of  $p$  (which is determined by hardware dependent constants  $a$  and  $b$ ) the cost of communication will overwhelm the cost of processing an item, thus leading to slowdown.

On the other hand, suppose the work per worker is fixed, that is,  $|\Omega|$  increases and the number of workers  $p$  increases proportionally. The average time  $a \cdot (|\Omega|k/np)$  to process an item remains constant, and NOMAD scales linearly.

Finally, we discuss the communication complexity of NOMAD. For this discussion we focus on a single item parameter  $\mathbf{h}_j$  which consists of  $O(k)$  numbers. In order to be processed by all the  $p$  workers once, it needs to be communicated  $p$  times. This requires  $O(kp)$  communication per item. There are  $n$  items, and if we make a simplifying assumption that during the execution of NOMAD each item is processed a constant  $c$  number of times by each processor, then the total communication complexity is  $O(nkp)$ .

### 3.3 Dynamic Load Balancing

As different workers have different number of ratings per item, the speed at which a worker processes a set of ratings  $\bar{\Omega}_j^{(q)}$  for an item  $j$  also varies among workers. Furthermore, in the distributed memory setting different workers might process updates at different rates due to differences in hardware and system load. NOMAD can handle this by dynamically balancing the workload of workers: in line 22 of Algorithm 1, instead of sampling the recipient of a message uniformly at random we can preferentially select a worker which has fewer items in its queue to process. To do this, a payload carrying information about the size of the  $\text{queue}[q]$  is added to the messages that the workers send each other. The overhead of passing the payload information is just a single integer per message. This scheme allows us to dynamically load balance, and ensures that a slower worker will receive smaller amount of work compared to others.

### 3.4 Hybrid Architecture

In a hybrid architecture we have multiple threads on a single machine as well as multiple machines distributed across the network. In this case, we make two improvements to the basic algorithm. First, in order to amortize the communication costs we reserve two additional threads per machine for sending and receiving  $(j, \mathbf{h}_j)$  pairs over the network. Intra-machine communication is much cheaper than machine-to-machine communication, since the former does not involve

a network hop. Therefore, whenever a machine receives a  $(j, \mathbf{h}_j)$  pair, it circulates the pair among all of its threads before sending the pair over the network. This is done by uniformly sampling a random permutation whose size equals to the number of worker threads, and sending the item variable to each thread according to this permutation. Circulating a variable more than once was found to not improve convergence, and hence is not used in our algorithm.

### 3.5 Implementation Details

Multi-threaded MPI was used for inter-machine communication. Instead of communicating single  $(j, \mathbf{h}_j)$  pairs, we follow the strategy of [23], and accumulate a fixed number of pairs (e.g., 100) before transmitting them over the network.

NOMAD can be implemented with lock-free data structures since the only interaction between threads is via operations on the queue. We used the concurrent queue provided by Intel Thread Building Blocks (TBB) [3]. Although technically not lock-free, the TBB concurrent queue nevertheless scales almost linearly with the number of threads.

Since there is very minimal sharing of memory across threads in NOMAD, by making memory assignments in each thread carefully aligned with cache lines we can exploit memory locality and avoid cache ping-pong. This results in near linear scaling for the multi-threaded setting.

## 4. RELATED WORK

### 4.1 Map-Reduce and Friends

Since many machine learning algorithms are iterative in nature, a popular strategy to distribute them across multiple machines is to use bulk synchronization after every iteration. Typically, one partitions the data into chunks that are distributed to the workers at the beginning. A master communicates the current parameters which are used to perform computations on the slaves. The slaves return the solutions during the bulk synchronization step, which are used by the master to update the parameters. The popularity of this strategy is partly thanks to the widespread availability of Hadoop [1], an open source implementation of the MapReduce framework [9].

All three optimization schemes for matrix completion namely ALS, CCD++, and SGD, can be parallelized using a bulk synchronization strategy. This is relatively simple for ALS [27] and CCD++ [26], but a bit more involved for SGD [12, 18]. Suppose  $p$  machines are available. Then, the Distributed Stochastic Gradient Descent (DSGD) algorithm of Gemulla et al. [12] partitions the indices of users  $\{1, 2, \dots, m\}$  into mutually exclusive sets  $I_1, I_2, \dots, I_p$  and the indices of items into  $J_1, J_2, \dots, J_p$ . Now, define

$$\Omega^{(q)} := \{(i, j) \in \Omega; i \in I_q, j \in J_q\}, \quad 1 \leq q \leq p,$$

and suppose that each machine runs SGD updates (9) and (10) independently, but machine  $q$  samples  $(i, j)$  pairs only from  $\Omega^{(q)}$ . By construction,  $\Omega^{(q)}$ 's are disjoint and hence these updates can be run in parallel. A similar observation was also made by Recht and Ré [18]. A bulk synchronization step redistributes the sets  $J_1, J_2, \dots, J_p$  and corresponding rows of  $H$ , which in turn changes the  $\Omega^{(q)}$  processed by each machine, and the iteration proceeds (see Figure 3)

Unfortunately, bulk synchronization based algorithms have two major drawbacks: First, the communication and computation steps are done in sequence. What this means is

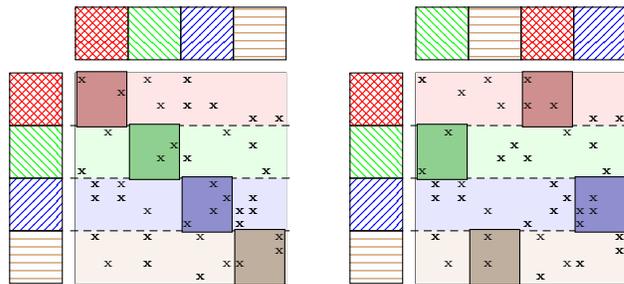


Figure 3: Illustration of DSGD algorithm with 4 workers. Initially  $W$  and  $H$  are partitioned as shown on the left. Each worker runs SGD on its active area as indicated. After each worker completes processing data points in its own active area, the columns of item parameters  $H^T$  are exchanged randomly, and the active area changes. This process is repeated for each iteration.

that when the CPU is busy, the network is idle and vice versa. The second issue is that they suffer from what is widely known as the *the curse of last reducer* [4, 24]. In other words, all machines have to wait for the slowest machine to finish before proceeding to the next iteration. Zhuang et al. [28] report that DSGD suffers from this problem even in the shared memory setting.

DSGD++ is an algorithm proposed by Teflioudi et al. [25] to address the first issue discussed above. Instead of using  $p$  partitions, DSGD++ uses  $2p$  partitions. While the  $p$  workers are processing  $p$  partitions, the other  $p$  partitions are sent over the network. This keeps both the network and CPU busy simultaneously. However, DSGD++ also suffers from the curse of the last reducer.

Another attempt to alleviate the problems of bulk synchronization in the shared memory setting is the FPSGD\*\* algorithm of Zhuang et al. [28]; given  $p$  threads, FPSGD\*\* partitions the parameters into more than  $p$  sets, and uses a task manager thread to distribute the partitions. When a thread finishes updating one partition, it requests for another partition from the task manager. It is unclear how to extend this idea to the distributed memory setting.

In NOMAD we sidestep all the drawbacks of bulk synchronization. Like DSGD++ we also simultaneously keep the network and CPU busy. On the other hand, like FPSGD\*\* we effectively load balance between the threads. To understand why NOMAD enjoys both these benefits, it is instructive to contrast the data partitioning schemes underlying DSGD, DSGD++, FPSGD\*\*, and NOMAD (see Figure 4). Given  $p$  number of workers, DSGD divides the rating matrix  $A$  into  $p \times p$  number of blocks; DSGD++ improves upon DSGD by further dividing each block to  $1 \times 2$  sub-blocks (Figure 4 (a) and (b)). On the other hand, FPSGD\*\* splits  $A$  into  $p' \times p'$  blocks with  $p' > p$  (Figure 4 (c)), while NOMAD uses  $p \times n$  blocks (Figure 4 (d)). In terms of communication there is no difference between various partitioning schemes; all of them require  $O(nkp)$  communication for each item to be processed a constant  $c$  number of times. However, having smaller blocks means that NOMAD has much more flexibility in assigning blocks to processors, and hence better ability to exploit parallelism. Because NOMAD operates at the level of individual item parameters,  $\mathbf{h}_j$ , it can dynamically load balance by assigning fewer columns to a slower

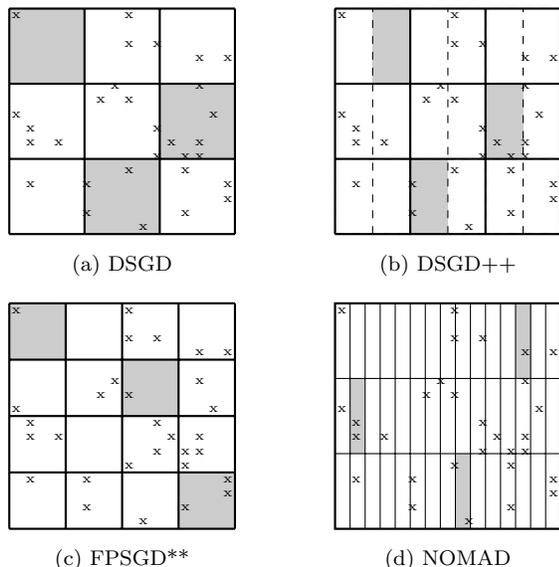


Figure 4: Comparison of data partitioning schemes between algorithms. Example active area of stochastic gradient sampling is marked as gray.

worker. A pleasant side effect of such a fine grained partitioning coupled with the lock free nature of updates is that one does not require sophisticated scheduling algorithms to achieve good performance. Consequently, NOMAD outperforms DSGD, DSGD++, and FPSGD\*\*.

## 4.2 Asynchronous Algorithms

There is growing interest in designing machine learning algorithms that do not perform bulk synchronization. See, for instance, the randomized (block) coordinate descent methods of Richtarik and Takac [20] and the Hogwild! algorithm of Recht et al. [19]. A relatively new approach to asynchronous parallelism is to use a so-called parameter server. A parameter server is either a single machine or a distributed set of machines which caches the current values of the parameters. Workers store local copies of the parameters and perform updates on them, and periodically synchronize their local copies with the parameter server. The parameter server receives updates from all workers, aggregates them, and communicates them back to the workers. The earliest work on a parameter server, that we are aware of, is due to Smola and Narayanamurthy [23], who propose using a parameter server for collapsed Gibbs sampling in Latent Dirichlet Allocation. PowerGraph [13], upon which the latest version of the GraphLab toolkit is based, is also essentially based on the idea of a parameter server. However, the difference in case of PowerGraph is that the responsibility of parameters is distributed across multiple machines, but at the added expense of synchronizing the copies.

Very roughly speaking, the asynchronously parallel version of the ALS algorithm in GraphLab works as follows:  $\mathbf{w}_i$  and  $\mathbf{h}_j$  variables are distributed across multiple machines, and whenever  $\mathbf{w}_i$  is being updated with equation (3), the values of  $\mathbf{h}_j$ 's for  $j \in \Omega_i$  are retrieved across the network and read-locked until the update is finished. GraphLab provides functionality such as network communication and a distributed locking mechanism to implement this. However,

frequently acquiring read-locks over the network can be expensive. In particular, a popular user who has rated many items will require read locks on a large number of items, and this will lead to vast amount of communication and delays in updates on those items. GraphLab provides a complex job scheduler which attempts to minimize this cost, but then the efficiency of parallelization depends on the difficulty of the scheduling problem and the effectiveness of the scheduler.

In our empirical evaluation NOMAD performs significantly better than GraphLab. The reasons are not hard to see. First, because of the lock free nature of NOMAD, we completely avoid acquiring expensive network locks. Second, we use SGD which allows us to exploit finer grained parallelism as compared to ALS, and also leads to faster convergence. In fact, the GraphLab framework is not well suited for SGD (personal communication with the developers of GraphLab). Finally, because of the finer grained data partitioning scheme used in NOMAD, unlike GraphLab whose performance heavily depends on the underlying scheduling algorithms, we do not require a complicated scheduling mechanism.

## 4.3 Numerical Linear Algebra

The concepts of asynchronous and non-blocking updates have also been studied in numerical linear algebra. To avoid the load balancing problem and to reduce processor idle time, asynchronous numerical methods were first proposed over four decades ago by Chazan and Miranker [8]. Given an operator  $\mathcal{H} : \mathbb{R}^m \rightarrow \mathbb{R}^m$ , to find the fixed point solution  $x^*$  such that  $\mathcal{H}(x^*) = x^*$ , a standard Gauss-Seidel-type procedure performs the update  $x_i = (\mathcal{H}(x))_i$  sequentially (or randomly). Using the asynchronous procedure, each computational node asynchronously conducts updates on each variable (or a subset)  $x_i^{\text{new}} = (\mathcal{H}(x))_i$  and then overwrites  $x_i$  in common memory by  $x_i^{\text{new}}$ . Theory and applications of this asynchronous method have been widely studied (see the literature review of Frommer and Szyld [11] and the seminal textbook by Bertsekas and Tsitsiklis [6]). The concept of this asynchronous fixed-point update is very closely related to the Hogwild algorithm of Recht et al. [19] or the so-called Asynchronous SGD (ASGD) method proposed by Teflioudi et al. [25]. Unfortunately, such algorithms are *non-serializable*, that is, there may not exist an equivalent update ordering in a serial implementation. In contrast, our NOMAD algorithm is not only asynchronous but also serializable, and therefore achieves faster convergence in practice.

On the other hand, non-blocking communication has also been proposed to accelerate iterative solvers in a distributed setting. For example, Hoefer et al. [14] presented a distributed conjugate gradient implementation with non-blocking collective MPI operations for solving linear systems. However, this algorithm still requires synchronization at each CG iteration, so it is very different from our NOMAD algorithm.

## 4.4 Discussion

We remark that among algorithms we have discussed so far, NOMAD is the only distributed-memory algorithm which is both asynchronous and lock-free. Other parallel versions of SGD such as DSGD and DSGD++ are lock-free, but not fully asynchronous; therefore, the cost of synchronization increases as the number of machines grows [28]. On the other hand, the GraphLab implementation of ALS [17] is asynchronous but not lock-free, and therefore depends on

a complex job scheduler to reduce the side-effect of using locks.

## 5. EXPERIMENTS

In this section, we evaluate the empirical performance of NOMAD with extensive experiments. For the distributed memory experiments we compare NOMAD with DSGD [12], DSGD++ [25] and CCD++ [26]. We also compare against GraphLab, but the quality of results produced by GraphLab are significantly worse than the other methods, and therefore the plots for this experiment are delegated to Appendix F. For the shared memory experiments we pitch NOMAD against FPSGD\*\* [28] (which is shown to outperform DSGD in single machine experiments) as well as CCD++. Our experiments are designed to answer the following:

- How does NOMAD scale with the number of cores on a single machine? (Section 5.2)
- How does NOMAD scale as a fixed size dataset is distributed across multiple machines? (Section 5.3)
- How does NOMAD perform on a commodity hardware cluster? (Section 5.4)
- How does NOMAD scale when both the size of the data as well as the number of machines grow? (Section 5.5)

Since the objective function (1) is non-convex, different optimizers will converge to different solutions. Factors which affect the quality of the final solution include 1) initialization strategy, 2) the sequence in which the ratings are accessed, and 3) the step size decay schedule. It is clearly not feasible to consider the combinatorial effect of all these factors on each algorithm. However, we believe that the overall trend of our results is not affected by these factors.

### 5.1 Experimental Setup

Publicly available code for FPSGD\*\*<sup>3</sup> and CCD++<sup>4</sup> was used in our experiments. For DSGD and DSGD++, which we had to implement ourselves because the code is not publicly available, we closely followed the recommendations of Gemulla et al. [12] and Teffioudi et al. [25], and in some cases made improvements based on our experience. For a fair comparison all competing algorithms were tuned for optimal performance on our hardware. The code and scripts required for reproducing the experiments are readily available for download from <http://bikestra.github.io/>. Parameters used in our experiments are summarized in Table 1.

Table 1: Dimensionality parameter  $k$ , regularization parameter  $\lambda$  (1) and step-size schedule parameters  $\alpha, \beta$  (11)

Name	$k$	$\lambda$	$\alpha$	$\beta$
Netflix	100	0.05	0.012	0.01
Yahoo! Music	100	1.00	0.00075	0.05
Hugewiki	100	0.01	0.001	0

For all experiments, except the ones in Section 5.5, we will work with three benchmark datasets namely Netflix, Yahoo! Music, and Hugewiki (see Table 2 for more details).

<sup>3</sup><http://www.csie.ntu.edu.tw/~cjlin/libmf/>

<sup>4</sup><http://www.cs.utexas.edu/~rofyu/libpmf/>

Table 2: Dataset Details

Name	Rows	Columns	Non-zeros
Netflix [5]	2,649,429	17,770	99,072,112
Yahoo! Music [10]	1,999,990	624,961	252,800,275
Hugewiki [2]	50,082,603	39,780	2,736,496,604

The same training and test dataset partition is used consistently for all algorithms in every experiment. Since our goal is to compare optimization algorithms, we do very minimal parameter tuning. For instance, we used the same regularization parameter  $\lambda$  for each dataset as reported by Yu et al. [26], and shown in Table 1; we study the effect of the regularization parameter on the convergence of NOMAD in Appendix A. By default we use  $k = 100$  for the dimension of the latent space; we study how the dimension of the latent space affects convergence of NOMAD in Appendix B. All algorithms were initialized with the same initial parameters; we set each entry of  $W$  and  $H$  by independently sampling a uniformly random variable in the range  $(0, \frac{1}{\sqrt{k}})$  [26, 28].

We compare solvers in terms of Root Mean Square Error (RMSE) on the test set, which is defined as:

$$\sqrt{\frac{\sum_{(i,j) \in \Omega^{\text{test}}} (A_{ij} - \langle \mathbf{w}_i, \mathbf{h}_j \rangle)^2}{|\Omega^{\text{test}}|}},$$

where  $\Omega^{\text{test}}$  denotes the ratings in the test set.

All experiments, except the ones reported in Section 5.4, are run using the Stampede Cluster at University of Texas, a Linux cluster where each node is outfitted with 2 Intel Xeon E5 (Sandy Bridge) processors and an Intel Xeon Phi Coprocessor (MIC Architecture). For single-machine experiments (Section 5.2), we used nodes in the `largemem` queue which are equipped with 1TB of RAM and 32 cores. For all other experiments, we used the nodes in the `normal` queue which are equipped with 32 GB of RAM and 16 cores (only 4 out of the 16 cores were used for computation). Inter-machine communication on this system is handled by MVAPICH2.

For the commodity hardware experiments in Section 5.4 we used `m1.xlarge` instances of Amazon Web Services, which are equipped with 15GB of RAM and four cores. We utilized all four cores in each machine; NOMAD and DSGD++ use two cores for computation and two cores for network communication, while DSGD and CCD++ use all four cores for both computation and communication. Inter-machine communication on this system is handled by MPICH2.

Since FPSGD\*\* uses single precision arithmetic, the experiments in Section 5.2 are performed using single precision arithmetic, while all other experiments use double precision arithmetic. All algorithms are compiled with Intel C++ compiler, with the exception of experiments in Section 5.4 where we used `gcc` which is the only compiler toolchain available on the commodity hardware cluster. For ready reference, exceptions to the experimental settings specific to each section are summarized in Table 3.

The convergence speed of stochastic gradient descent methods depends on the choice of the step size schedule. The schedule we used for NOMAD is

$$s_t = \frac{\alpha}{1 + \beta \cdot t^{1.5}}, \quad (11)$$

where  $t$  is the number of SGD updates that were performed on a particular user-item pair  $(i, j)$ . DSGD and DSGD++,

Table 3: Exceptions to each experiment

Section	Exception
Section 5.2	<ul style="list-style-type: none"> <li>run on <code>largemem</code> queue (32 cores, 1TB RAM)</li> <li>single precision floating point used</li> </ul>
Section 5.4	<ul style="list-style-type: none"> <li>run on <code>m1.xlarge</code> (4 cores, 15GB RAM)</li> <li>compiled with <code>gcc</code></li> <li>MPICH2 for MPI implementation</li> </ul>
Section 5.5	<ul style="list-style-type: none"> <li>Synthetic datasets</li> </ul>

on the other hand, use an alternative strategy called bold-driver [12]; here, the step size is adapted by monitoring the change of the objective function.

## 5.2 Scaling in Number of Cores

For the first experiment we fixed the number of cores to 30, and compared the performance of NOMAD vs FPSGD\*\*<sup>5</sup> and CCD++ (Figure 5). On Netflix (left) NOMAD not only converges to a slightly better quality solution (RMSE 0.914 vs 0.916 of others), but is also able to reduce the RMSE rapidly right from the beginning. On Yahoo! Music (middle), NOMAD converges to a slightly worse solution than FPSGD\*\* (RMSE 21.894 vs 21.853) but as in the case of Netflix, the initial convergence is more rapid. On Hugewiki, the difference is smaller but NOMAD still outperforms. The initial speed of CCD++ on Hugewiki is comparable to NOMAD, but the quality of the solution starts to deteriorate in the middle. Note that the performance of CCD++ here is better than what was reported in Zhuang et al. [28] since they used double-precision floating point arithmetic for CCD++. In other experiments (not reported here) we varied the number of cores and found that the relative difference in performance between NOMAD, FPSGD\*\* and CCD++ are very similar to that observed in Figure 5.

For the second experiment we varied the number of cores from 4 to 30, and plot the scaling behavior of NOMAD (Figures 6 and 7). Figure 6 (left) shows how test RMSE changes as a function of the number of updates on Yahoo! Music. Interestingly, as we increased the number of cores, the test RMSE decreased faster. We believe this is because when we increase the number of cores, the rating matrix  $A$  is partitioned into smaller blocks; recall that we split  $A$  into  $p \times n$  blocks, where  $p$  is the number of parallel workers. Therefore, the communication between workers becomes more frequent, and each SGD update is based on fresher information (see also Section 3.2 for mathematical analysis). This effect was more strongly observed on Yahoo! Music than others, since Yahoo! Music has much larger number of items (624,961 vs. 17,770 of Netflix and 39,780 of Hugewiki) and therefore more amount of communication is needed to circulate the new information to all workers. Results for other datasets are provided in Figure 18 in Appendix D.

On the other hand, to assess the efficiency of computation we define *average throughput* as the average number of ratings processed per core per second, and plot it for each dataset in Figure 6 (right), while varying the number of cores. If NOMAD exhibits linear scaling in terms of the speed it processes ratings, the average throughput should re-

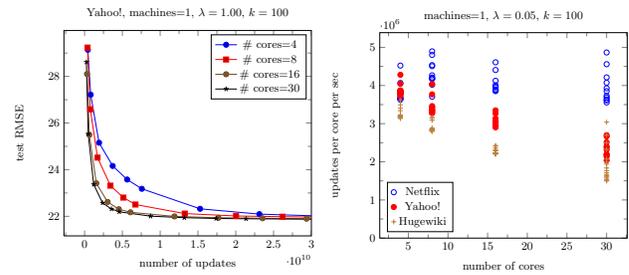


Figure 6: Left: Test RMSE of NOMAD as a function of the number of updates on Yahoo! Music, when the number of cores is varied. Right: Number of updates of NOMAD per core per second as a function of the number of cores.

main constant<sup>6</sup>. On Netflix, the average throughput indeed remains almost constant as the number of cores changes. On Yahoo! Music and Hugewiki, the throughput decreases to about 50% as the number of cores is increased to 30. We believe this is mainly due to cache locality effects.

Now we study how much speed-up NOMAD can achieve by increasing the number of cores. In Figure 7, we set  $y$ -axis to be test RMSE and  $x$ -axis to be the total CPU time expended which is given by the number of seconds elapsed multiplied by the number of cores. We plot the convergence curves by setting the # cores=4, 8, 16, and 30. If the curves overlap, then this shows that we achieve linear speed up as we increase the number of cores. This is indeed the case for Netflix and Hugewiki. In the case of Yahoo! Music we observe that the speed of convergence increases as the number of cores increases. This, we believe, is again due to the decrease in the block size which leads to faster convergence.

## 5.3 Scaling as a Fixed Dataset is Distributed Across Workers

In this subsection, we use 4 computation threads per machine. For the first experiment we fix the number of machines to 32 (64 for hugewiki), and compare the performance of NOMAD with DSGD, DSGD++ and CCD++ (Figure 8). On Netflix and Hugewiki, NOMAD converges much faster than its competitors; not only is initial convergence faster, but also it discovers a better quality solution. On Yahoo! Music, all four methods perform similarly. This is because the cost of network communication relative to the size of the data is much higher for Yahoo! Music; while Netflix and Hugewiki have 5,575 and 68,635 non-zero ratings per each item respectively, Yahoo! Music has only 404 ratings per item. Therefore, when Yahoo! Music is divided equally across 32 machines, each item has only 10 ratings on average per each machine. Hence the cost of sending and receiving item parameter vector  $\mathbf{h}_j$  for one item  $j$  across the network is higher than that of executing SGD updates on the ratings of the item locally stored within the machine,  $\Omega_j^{(q)}$ . As a consequence, the cost of network communication dominates the overall execution time of all algorithms, and little difference in convergence speed is found between them.

<sup>5</sup>Since the current implementation of FPSGD\*\* in LibMF only reports CPU execution time, we divide this by the number of threads and use this as a proxy for wall clock time.

<sup>6</sup>Note that since we use single-precision floating point arithmetic in this section to match the implementation of FPSGD\*\*, the throughput of NOMAD is about 50% higher than that in other experiments.

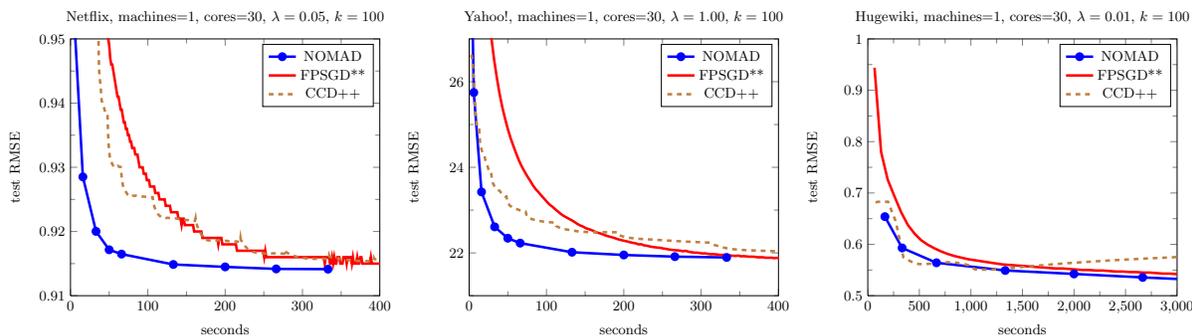


Figure 5: Comparison of NOMAD, FPSGD\*\*, and CCD++ on a single-machine with 30 computation cores.

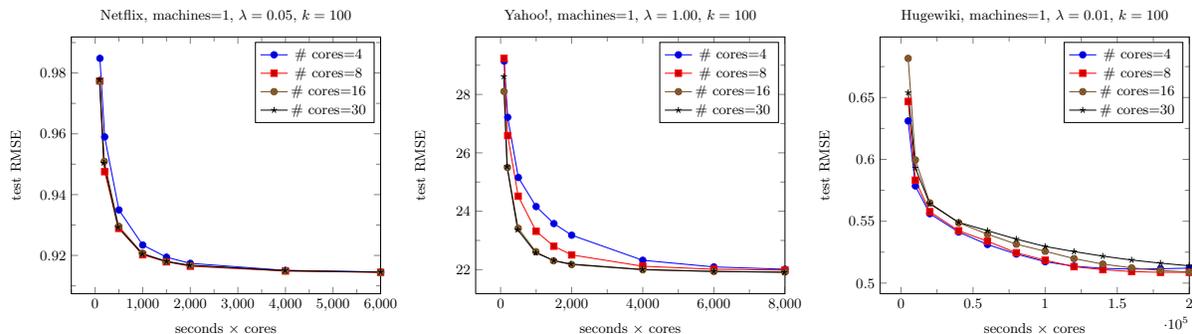


Figure 7: Test RMSE of NOMAD as a function of computation time (time in seconds  $\times$  the number of cores), when the number of cores is varied.

For the second experiment we varied the number of machines from 1 to 32, and plot the scaling behavior of NOMAD (Figures 10 and 9). Figure 10 (left) shows how test RMSE decreases as a function of the number of updates on Yahoo! Music. Again, if NOMAD scales linearly the average throughput has to remain constant; here we observe improvement in convergence speed when 8 or more machines are used. This is again the effect of smaller block sizes which was discussed in Section 5.2. On Netflix, a similar effect was present but was less significant; on Hugewiki we did not see any notable difference between configurations (see Figure 19 in Appendix D).

In Figure 10 (right) we plot the average throughput (the number of updates per machine per core per second) as a function of the number of machines. On Yahoo! Music the average throughput goes down as we increase the number of machines, because as mentioned above, each item has a small number of ratings. On Hugewiki we observe almost linear scaling, and on Netflix the average throughput even improves as we increase the number of machines; we believe this is because of cache locality effects. As we partition users into smaller and smaller blocks, the probability of cache miss on user parameters  $w_i$ 's within the block decrease, and on Netflix this makes a meaningful difference: indeed, there are only 480,189 users in Netflix who have at least one rating. When this is equally divided into 32 machines, each machine contains only 11,722 active users on average. Therefore the  $w_i$  variables only take 11MB of memory, which is smaller than the size of L3 cache (20MB) of the machine we used and therefore leads to increase in the number of updates per machine per core per second.

Now we study how much speed-up NOMAD can achieve by increasing the number of machines. In Figure 9, we set  $y$ -axis to be test RMSE and  $x$ -axis to be the number of seconds elapsed multiplied by the total number of cores used in the configuration. Again, all lines will coincide with each other if NOMAD shows linear scaling. On Netflix, with 2 and 4 machines we observe mild slowdown, but with more than 4 machines NOMAD exhibits super-linear scaling. On Yahoo! Music we observe super-linear scaling with respect to the speed of a single machine on all configurations, but the highest speedup is seen with 16 machines. On Hugewiki, linear scaling is observed in every configuration.

## 5.4 Scaling on Commodity Hardware

In this subsection, we want to analyze the scaling behavior of NOMAD on commodity hardware. Using Amazon Web Services (AWS), we set up a computing cluster that consists of 32 machines; each machine is of type `m1.xlarge` and equipped with quad-core Intel Xeon E5430 CPU and 15GB of RAM. Network bandwidth among these machines is reported to be approximately 1Gb/s<sup>7</sup>.

Since NOMAD and DSGD++ dedicates two threads for network communication, on each machine only two cores are available for computation<sup>8</sup>. In contrast, bulk synchronization algorithms such as DSGD and CCD++ which separate

<sup>7</sup><http://epamcloud.blogspot.com/2013/03/testing-amazon-ec2-network-speed.html>

<sup>8</sup>Since network communication is not computation-intensive, for DSGD++ we used four computation threads instead of two and got better results; thus we report results with four computation threads for DSGD++.

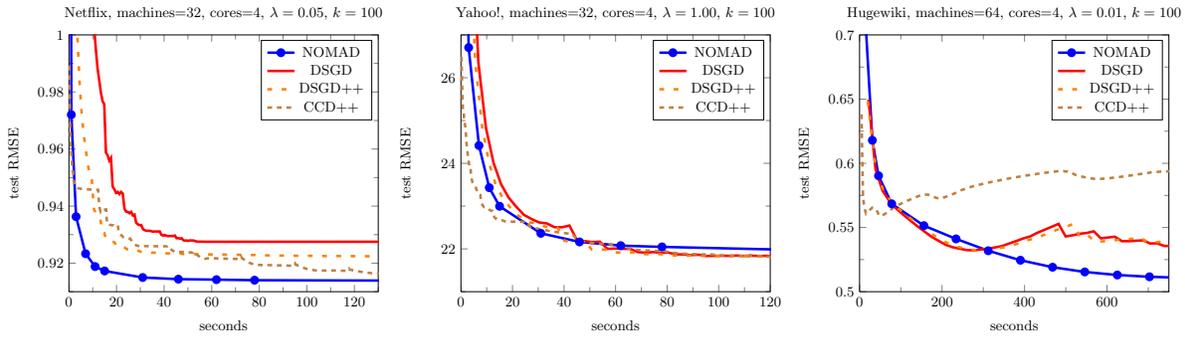


Figure 8: Comparison of NOMAD, DSGD, DSGD++, and CCD++ on a HPC cluster.

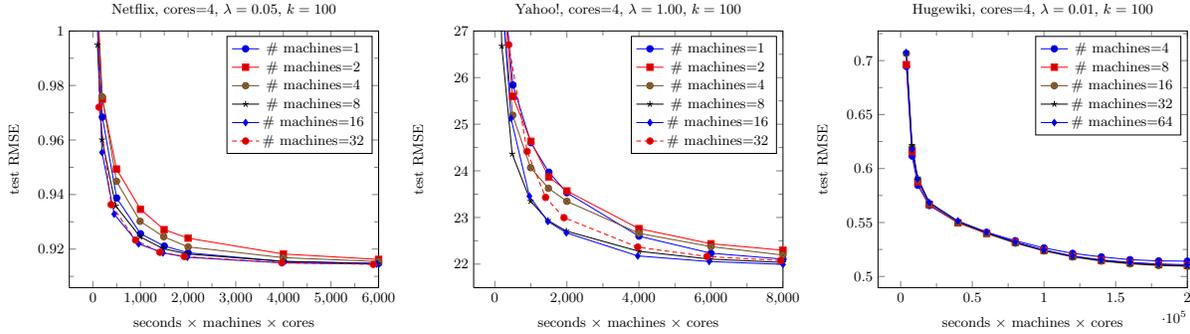


Figure 9: Test RMSE of NOMAD as a function of computation time (time in seconds  $\times$  the number of machines  $\times$  the number of cores per each machine) on a HPC cluster, when the number of machines is varied.

computation and communication can utilize all four cores for computation. In spite of this disadvantage, Figure 11 shows that NOMAD outperforms all other algorithms in this setting as well. In this plot, we fixed the number of machines to 32; on Netflix and Hugewiki, NOMAD converges more rapidly to a better solution. Recall that on Yahoo! Music, all four algorithms performed very similarly on a HPC cluster in Section 5.3. However, on commodity hardware NOMAD outperforms the other algorithms. This shows that the efficiency of network communication plays a very important role in commodity hardware clusters where the communication is relatively slow. On Hugewiki, however, the number of columns is very small compared to the number of ratings and thus network communication plays smaller role in this dataset compared to others. Therefore, initial convergence of DSGD is a bit faster than NOMAD as it uses all four cores on computation while NOMAD uses only two. Still, the overall convergence speed is similar and NOMAD finds a better quality solution.

As in Section 5.3, we increased the number of machines from 1 to 32, and studied the scaling behavior of NOMAD. The overall trend was identical to what we observed in Figure 10 and 9; due to page constraints, the plots for this experiment can be found in the Appendix C.

### 5.5 Scaling as both Dataset Size and Number of Machines Grows

In previous sections (Section 5.3 and Section 5.4), we studied the scalability of algorithms by partitioning a fixed amount of data into increasing number of machines. In real-world applications of collaborative filtering, however, the

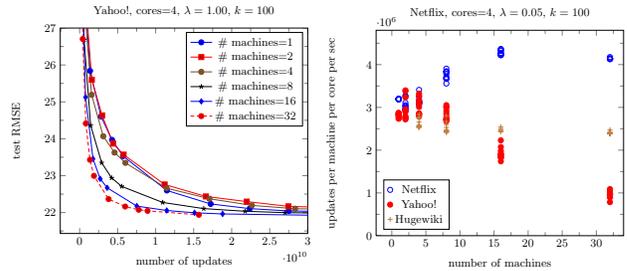


Figure 10: Results on HPC cluster when the number of machines is varied. Left: Test RMSE of NOMAD as a function of the number of updates on Netflix and Yahoo! Music. Right: Number of updates of NOMAD per machine per core per second as a function of the number of machines.

size of the data should grow over time as new users are added to the system. Therefore, to match the increased amount of data with equivalent amount of physical memory and computational power, the number of machines should increase as well. The aim of this section is to compare the scaling behavior of NOMAD and that of other algorithms in this realistic scenario.

To simulate such a situation, we generated synthetic datasets whose characteristics resemble real data; the number of ratings for each user and each item is sampled from the corresponding empirical distribution of the Netflix data. As we increase the number of machines from 4 to 32, we fixed the number of items to be the same to that of Netflix (17,770), and increased the number of users to be proportional to the

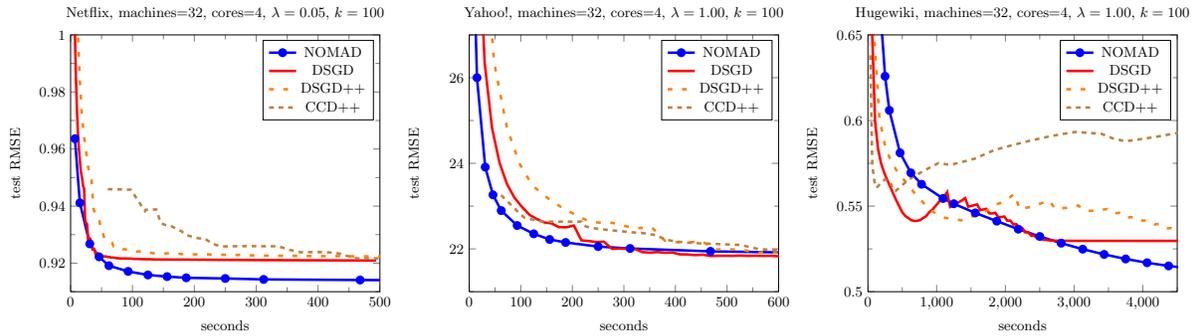


Figure 11: Comparison of NOMAD, DSGD, DSGD++, and CCD++ on a commodity hardware cluster.

number of machines ( $480,189 \times$  the number of machines<sup>9</sup>). Therefore, the expected number of ratings in each dataset is proportional to the number of machines ( $99,072,112 \times$  the number of machines) as well.

Conditioned on the number of ratings for each user and item, the nonzero locations are sampled uniformly at random. Ground-truth user parameters  $\mathbf{w}_i$ 's and item parameters  $\mathbf{h}_j$ 's are generated from 100-dimensional standard isotropic Gaussian distribution, and for each rating  $A_{ij}$ , Gaussian noise with mean zero and standard deviation 0.1 is added to the “true” rating  $\langle \mathbf{w}_i, \mathbf{h}_j \rangle$ .

Figure 12 shows that the comparative advantage of NOMAD against DSGD, DSGD++ and CCD++ increases as we grow the scale of the problem. NOMAD clearly outperforms other methods on all configurations; DSGD++ is very competitive on the small scale, but as the size of the problem grows NOMAD shows better scaling behavior.

## 6. CONCLUSION AND FUTURE WORK

From our experimental study we conclude that

- On a single machine, NOMAD shows near-linear scaling up to 30 threads.
- When a fixed size dataset is distributed across multiple machines, NOMAD shows near-linear scaling up to 32 machines.
- Both in shared-memory and distributed-memory setting, NOMAD exhibits superior performance against state-of-the-art competitors; in commodity hardware cluster, the comparative advantage is more conspicuous.
- When both the size of the data as well as the number of machines grow, the scaling behavior of NOMAD is much nicer than its competitors.

Although we only discussed the matrix completion problem in this paper, it is worth noting that the idea of NOMAD is more widely applicable. Specifically, ideas discussed in this paper can be easily adapted as long as the objective function can be written as

$$f(W, H) = \sum_{i,j \in \Omega} f_{ij}(\mathbf{w}_i, \mathbf{h}_j).$$

As part of our ongoing work we are investigating ways to rewrite Support Vector Machines (SVMs), binary logistic

<sup>9</sup>480,189 is the number of users in Netflix who have at least one rating.

regression as a saddle point problem which have the above structure.

Inference in Latent Dirichlet Allocation (LDA) using a collapsed Gibbs sampler has a similar structure as the stochastic gradient descent updates for matrix factorization. An additional complication in LDA is that the variables need to be normalized. We are also investigating how the NOMAD framework can be used for LDA.

## 7. ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their constructive comments. We thank the Texas Advanced Computing Center at University of Texas and the Research Computing group at Purdue University for providing infrastructure and timely support for our experiments. Computing experiments on commodity hardware were made possible by an AWS in Education Machine Learning Research Grant Award. This material is partially based upon work supported by the National Science Foundation under grant nos. IIS-1219015 and CCF-1117055.

## References

- Apache Hadoop, 2009. <http://hadoop.apache.org/core/>.
- Graphlab datasets, 2013. <http://graphlab.org/downloads/datasets/>.
- Intel thread building blocks, 2013. <https://www.threadingbuildingblocks.org/>.
- A. Agarwal, O. Chapelle, M. Dudík, and J. Langford. A reliable effective terascale linear learning system. *CoRR*, abs/1110.4198, 2011.
- R. M. Bell and Y. Koren. Lessons from the netflix prize challenge. *SIGKDD Explorations*, 9(2):75–79, 2007.
- D. P. Bertsekas and J. N. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods*. Athena Scientific, 1997.
- L. Bottou and O. Bousquet. The tradeoffs of large-scale learning. *Optimization for Machine Learning*, pages 351–368, 2011.
- D. Chazan and W. Miranker. Chaotic relaxation. *Linear Algebra and its Applications*, 2:199–222, 1969.

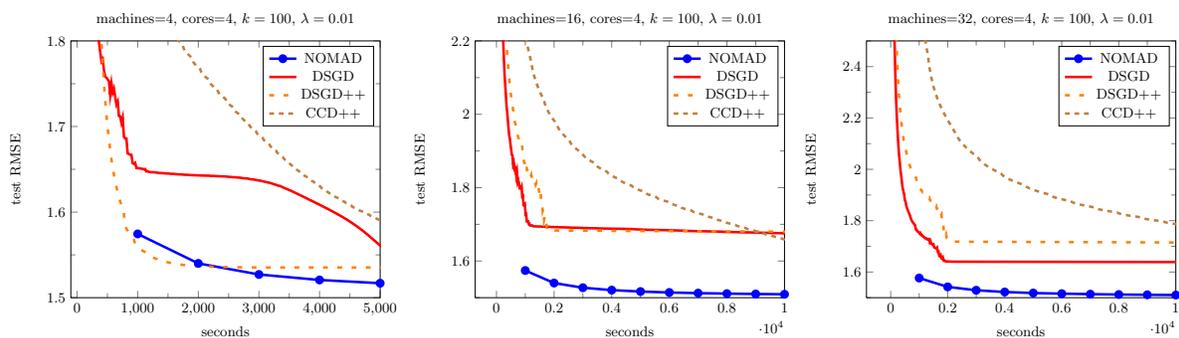


Figure 12: Comparison of algorithms when both dataset size and the number of machines grows. Left: 4 machines, middle: 16 machines, right: 32 machines

J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

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

A. Frommer and D. B. Szyld. On asynchronous iterations. *Journal of Computational and Applied Mathematics*, 123: 201–216, 2000.

R. Gemulla, E. Nijkamp, P. J. Haas, and Y. Sismanis. Large-scale matrix factorization with distributed stochastic gradient descent. In *Proceedings of the conference on Knowledge Discovery and Data Mining*, pages 69–77, 2011.

J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, pages 17–30, 2012.

T. Hoefer, P. Gottschling, W. Rehm, and A. Lumsdaine. Optimizing a conjugate gradient solver with non blocking operators. *Parallel Computing*, 2007.

C. J. Hsieh and I. S. Dhillon. Fast coordinate descent methods with variable selection for non-negative matrix factorization. In *Proceedings of the conference on Knowledge Discovery and Data Mining*, pages 1064–1072, August 2011.

H. Kushner and D. Clark. *Stochastic Approximation Methods for Constrained and Unconstrained Systems*, volume 26 of *Applied Mathematical Sciences*. Springer, New York, 1978.

Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Distributed graphlab: A framework for machine learning and data mining in the cloud. In *Proceedings of the International Conference on Very Large Data Bases*, pages 716–727, 2012.

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

B. Recht, C. Re, S. Wright, and F. Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Proceedings of the conference on Advances in Neural Information Processing Systems*, pages 693–701, 2011.

P. Richtarik and M. Takac. Distributed coordinate descent method for learning with big data. 2013. URL "<http://arxiv.org/abs/1310.2059>".

H. E. Robbins and S. Monro. A stochastic approximation method. *Annals of Mathematical Statistics*, 22:400–407, 1951.

S. Shalev-Schwartz and N. Srebro. SVM optimization: Inverse dependence on training set size. In *Proceedings of the International Conference on Machine Learning*, pages 928–935, 2008.

A. J. Smola and S. Narayanamurthy. An architecture for parallel topic models. In *Proceedings of the International Conference on Very Large Data Bases*, pages 703–710, 2010.

S. Suri and S. Vassilvitskii. Counting triangles and the curse of the last reducer. In *Proceedings of the International Conference on the World Wide Web*, pages 607–614, 2011.

C. Teflioudi, F. Makari, and R. Gemulla. Distributed matrix completion. In *Proceedings of the International Conference on Data Mining*, pages 655–664, 2012.

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

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

Y. Zhuang, W.-S. Chin, Y.-C. Juan, and C.-J. Lin. A fast parallel SGD for matrix factorization in shared memory systems. In *Proceedings of the ACM conference on Recommender systems*, pages 249–256, 2013.