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

Hyokyun Yun
Purdue University
West Lafayette, IN
United States
yun3@purdue.edu

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

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

S V N Vishwanathan
Purdue University
West Lafayette, IN
United States
vishy@stat.purdue.edu

Inderjit Dhillon
University of Texas, Austin
Austin, TX
United States
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). In our algorithm, the ownership of a variable is asynchronously transferred between processors in a decentralized fashion. Due to its lock-free and asynchronous nature, 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.

Unfortunately, machine learning problems such as matrix completion cannot be partitioned into independent subproblems; the solution of one subproblem depends on that of other subproblems. Another challenge is that machine learning algorithms are typically iterative; to continue making progress on one subproblem, parameter updates from other subproblems have to be repeatedly received [24]. Therefore, communication is essential when the problem is distributed across processors [27].

In order to reduce the cost of communication, many parallel machine learning algorithms are based on a shared-memory architecture. For example, the first version of the popular GraphLab toolkit used a shared-memory architecture [18]. Similarly, the randomized (block) coordinate descent methods of Richtarik and Takac [21] and the Hogwild! algorithm of Recht et al. [20] assume that every parallel processing unit can access every component of the parameter simultaneously. Related work on delayed stochastic gradient descent algorithm by Langford et al. [17] assumes that the stochastic gradients are computed with a delay by individual processors. Even in a single machine, however, the cost of communication between processors is not free; a well-known hardware issue called cache ping-pong [13] and the lack of locality can hamper the scalability of these algorithms [31].

In the distributed-memory setting, algorithms that bulk synchronize their state after every iteration are popular [8, 27]. This is partly because of the widespread availability of the MapReduce framework [9], and its open source implementation Hadoop [1]. Recent enhancements such as Spark [29] and MLBase [15] are also readily available for the practitioner to implement these iterative algorithms. However, bulk synchronization after every iteration suffers from the *curse of the last reducer* [4, 25]. What this means is that a vast majority of the computation finishes quickly, but a small fraction takes disproportionately long time. This is particularly true for many real-world datasets which suffer from skew; for instance, a small fraction of the users may rate many movies or a small fraction of popular movies may be rated by a large number of users [25]. Moreover, bulk synchronization requires a master-slave architecture,

wherein the master collects the messages from all the slaves and redistributes messages back to the slaves. This means that if either one of the slaves or the master nodes fail then the whole system fails catastrophically.

In this paper, we present NOMAD (Non-locking, stOchastic Multi-machine algorithm for Asynchronous and Decentralized matrix completion), which completely avoids bulk synchronization. Instead, communication is non-blocking, and machines exchange messages in an asynchronous fashion [6]. Moreover, the pattern of communication is also decentralized; each machine is symmetric to each other and does the same amount of computation and communication. 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 as well as existing multi-machine algorithms for the matrix completion problem.

This paper is organized as follows: Section 2 introduces notation and briefly surveys related work. Section 3 is devoted to describing NOMAD, and in Section 4 we present extensive empirical comparison of NOMAD with various existing algorithms. Section 5 concludes the paper with a discussion and provides pointers to future work.

2. BACKGROUND AND RELATED WORK

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 $\Omega \subseteq \{1 \dots m\} \times \{1, \dots, n\}$ 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 Ω_i to be the set of items rated by the i-th user, given by $\Omega_i := \{j : (i,j) \in \Omega\}$. 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 $\mathbf{w}_i \in \mathbb{R}^k$ of W can be thought of as a k-dimensional embedding of the user. Analogously, each row $\mathbf{h}_j \in \mathbb{R}^k$ 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: $\frac{\lambda}{2}\sum_{i=1}^{m} |\Omega_i| \cdot \|\mathbf{w}_i\|^2 + \frac{\lambda}{2}\sum_{j=1}^{n} |\bar{\Omega}_j| \cdot \|\mathbf{h}_i\|^2$, 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:

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

This can be further simplified and written as

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

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 $1 \le l \le k$ denote the l-th coordinate of the column vector $\mathbf{w}_i$ and $\mathbf{h}_j$, respectively. Furthermore, H_{Ω_i} (resp. $W_{\bar{\Omega}_j}$) will be used to denote the sub-matrix of H (resp. W) formed by collecting rows corresponding to Ω_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 a series of well known algorithms for matrix completion, 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:

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

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:

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

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

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

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

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

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}, ..., w_{1k}, w_{21}, ..., w_{2k}, ..., w_{m1}, ..., w_{mk}, h_{11}, ..., h_{1k}, h_{21}, ..., h_{2k}, ..., h_{n1}, ..., h_{nk}, w_{11}, ..., w_{1k}, and so on, then this recovers Cyclic Coordinate Descent (CCD) [14]. On the other hand, the update sequence w_{11}, ..., w_{m1}, h_{11}, ..., h_{n1}, w_{12}, ..., w_{m2}, h_{12}, ..., h_{n2} and so on, recovers the CCD++ algorithm of Yu et al. [28]. The CCD++ updates can be performed more efficiently than the CCD updates by maintaining a residual matrix [28].

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

$$\theta \leftarrow \theta - s_t \cdot \nabla_\theta f(\theta), \tag{7}$$

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

$$\theta \leftarrow \theta - s_t \cdot \nabla_\theta f_i(\theta). \tag{8}$$

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, 22]. The above update also enjoys desirable properties in terms of sample complexity, and hence is widely used in machine learning [7, 23].

When applied to matrix completion (1), the SGD updates require sampling a random index (i, j) uniformly from the set of nonzero indicies Ω, computing

$$\nabla_{\mathbf{w}_i} F_t(\theta) = \begin{cases} (A_{ij} - \langle \mathbf{w}_i, \mathbf{h}_{j_t} \rangle)\mathbf{h}_{j_t} + \lambda \mathbf{w}_i, & \text{for } i = i_n, \\ 0, & \text{for } i \neq i_n, \end{cases}$$

$$\nabla_{\mathbf{h}_j} F_t(\theta) = \begin{cases} (A_{ij} - \mathbf{w}_{i_t}^T \mathbf{h}_j)\mathbf{w}_{i_t} + \lambda \mathbf{h}_j, & \text{for } j = j_n, \\ 0, & \text{for } j \neq j_t. \end{cases}$$

and performing the update

$$\mathbf{w}_{i_t} \leftarrow \mathbf{w}_{i_t} - s_t \cdot \left[(A_{i_t j_t} - \mathbf{w}_{i_t}\mathbf{h}_{j_t})\mathbf{h}_{j_t} + \lambda \mathbf{w}_{i_t}\right], \tag{9}$$

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

2.4 Parallelization

In order to describe parallel algorithms for matrix completion, we find it very instructive to 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 $j \in \Omega_i$.

For parallelization, Yu et al. [28] propose the following scheme: the l-th column of W and H are copied to all machines, and each machine uses this information to update the l-th coordinate of the $\mathbf{w}_i$'s they own using (6). The updates are then synchronized, and then the iteration proceeds. A similar scheme, which updates all the coordinates of W using the ALS update (3) was proposed by Zhou et al. [30]. However, both these schemes require bulk synchronization after every iteration.

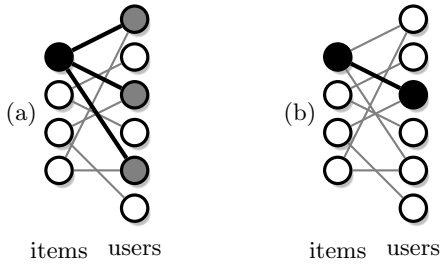


Figure 1: Illustration of updates used in matrix completion. Three algorithms are shown here: (a) alternating least squares and coordinate descent, (b) stochastic gradient descent. Black indicates that the value of the node is being updated, gray indicates that the value of the node is being read. White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

Low et al. [19] proposed an alternative asynchronous approach to parallelization, based on the GraphLab framework. The variables $\mathbf{h}_j$ are distributed across multiple machines, and whenever $\mathbf{w}_i$ is updated, the values of all the $\mathbf{h}_j$ for $j \in \Omega_i$ are retrieved. This is done by read-locking the variables in the respective machines. This means that there is no bulk-synchronization, but one needs to acquire read-locks on the network which are often expensive. Furthermore, a popular user who has rated many items will require read locks on a number of items, leading to delays in updating such nodes. PowerGraph aims to solve this problem by replicating popular users on multiple nodes [12], but at the added expense of synchronizing the copies.

The SGD update (9) for $\mathbf{w}_i$ only require us to access the value of $\mathbf{h}_j$ for a single random $j \in \Omega_i$ (Figure 1 (b)), and therefore leads to finer-grained parallelism. The Distributed Stochastic Gradient Descent (DSGD) of Gemulla et al. [11] exploits this property of SGD to partition the nodes of the bipartite graph and distributes them across multiple machines. Each machine picks a random edge from the subgraph it owns and performs SGD updates. Periodically, a bulk synchronization step is performed to re-partition and re-distribute the nodes of the bipartite graph.

More concretely, suppose p machines are available, then, one can partition the indices of users $\{1, 2, \ldots, m\}$ into mutually exclusive sets $I_1, I_2, \ldots, I_p$. Similarly, the indices of items can be partitioned into $J_1, J_2, \ldots, J_p$. Now, define

$$\Omega^{(q)} := \{(i, j) \in \Omega; i \in I_q, j \in J_q\}, \quad 1 \leq q \leq p, \tag{11}$$

and suppose that each machine runs SGD updates (9) and (10) independently, but the sampling of a random user-item pair (i, j) is constrained; machine q samples only within $\Omega^{(q)}$. Note that by construction, $\Omega^{(q)}$'s are disjoint and therefore it is impossible for two machines q and q' with $q \neq q'$ to sample the same user or item. A bulk synchronization step redistributes the sets $J_1, J_2, \ldots, J_p$ and corresponding rows of H, which in turn means that the $\Omega^{(q)}$ of each machine changes, and the iteration proceeds (see Figure 2)

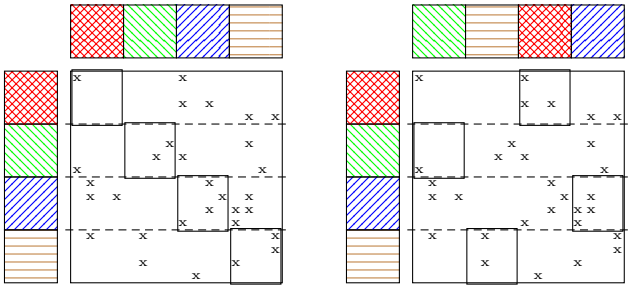


Figure 2: Illustration of DSGD algorithm with 4 machines. Initially W and H are partitioned as shown on the left. Each machine runs SGD on its active area as indicated. After each machine completes processing data points in its own active area, the columns of item parameters H^T are exchanged randomly, and the active area changes. This process is repeated for each iteration.

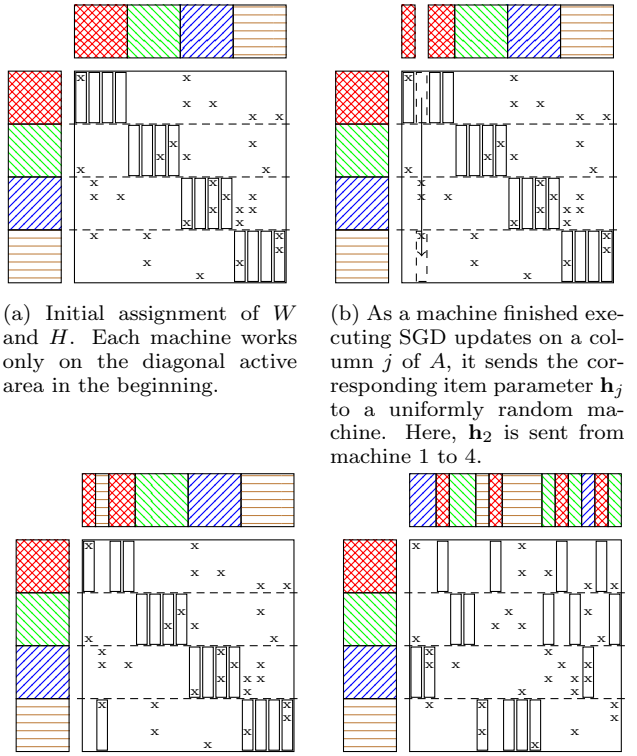
DSGD uses the CPU and the network in a sequential manner. When the network is active during bulk synchronization, no computation can be done. On the other hand, when the SGD updates are being performed, no network communication is possible. DSGD++ is an algorithm proposed by Teflioudi et al. [26] to remedy this situation. Given p machines, the items are partitioned into $J_1, J_2, \ldots, J_{2p}$, while the parameters corresponding to p of these partitions is being updated the other p partitions are transmitted across the network. Another attempt to remedy the same problem is proposed by Zhuang et al. [31]. Given p threads their FPSGD** algorithm, which only works in a shared memory setting, 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. This avoids thread stalling and keeps the CPU busy all the time. However, it is not clear how to extend this architecture to a distributed setting. Furthermore, as we will see in Section 4.2, NOMAD outperforms FPSGD** on a shared memory architecture, and comprehensively outperforms DSGD in a distributed memory setting.

3. NOMAD

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 in many machines (see Section 3.3).

To achieve a high level of parallelism, NOMAD splits the parameters, and hence the work, into much finer level of granularity as compared to previous approaches. The users $\{1, 2, \ldots, m\}$ are split into p disjoint sets I_1,I_2,...,I_p, as in DSGD. The rating matrix A for each user set I_q is partitioned into the level of individual items; that is, the rating matrix A is divided into $p \times n$ blocks, where each block, denoted by $\bar{\Omega}_j^{(q)}$, is defined as follows:

$$\bar{\Omega}_j^{(q)} := \left\{(i,j) \in \bar{\Omega}_j; i \in I_q\right\}, \quad 1 \le j \le n, 1 \le q \le p. \quad (12)$$



(a) Initial assignment of W and H. Each machine works only on the diagonal active area in the beginning.

(b) As a machine finished executing SGD updates on a column j of A, it sends the corresponding item parameter $\mathbf{h}_j$ to a uniformly random machine. Here, $\mathbf{h}_2$ is sent from machine 1 to 4.

(c) Upon receipt, the corresponding column of A becomes actively sampled by SGD. In this example, now machine 4 owns column 2 and it is the only machine that will touch column 2 of rating matrix A.

(d) At each point of time, each item parameter $\mathbf{h}_j$ is located in a random machine.

Figure 3: Illustration of the NOMAD algorithm

The $\mathbf{w}_i$ variables are *native* to a worker and are not communicated while the $\mathbf{h}_j$ variables move between workers, and are therefore *nomadic* variables. This distribution is effective for the real-life setting where the number of users typically far exceeds the number of items. A worker which owns the variable $\mathbf{h}_j$ executes 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 is accessed by only one worker, while the owner-computes paradigm ensures the same for $\mathbf{h}_j$'s.

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 ($1 \le j \le n$), and a corresponding k-dimensional parameter vector $\mathbf{h}_j$; this pair is denoted as $(j, \mathbf{h}_j)$. Each worker q tries to pop a $(j, \mathbf{h}_j)$ pair from its own queue, `queue`$[q]$. If the pop succeeds, it runs stochastic gradient descent update on $\bar{\Omega}_j^{(q)}$, which is the set of ratings of 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, and also that the nature of this communication is asynchronous and non-blocking.

Algorithm 1 the basic NOMAD algorithm

1: λ: 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 $1 \le i \le m, 1 \le l \le k$
5: $h_{jl} \sim \text{UniformReal}\left(0, \frac{1}{\sqrt{k}}\right)$ for $1 \le j \le n, 1 \le l \le k$
6: `// initialize queues`
7: **for** $j \in \{1, 2, \ldots, n\}$ **do**
8: $q \sim \text{UniformDiscrete}\{1, 2, \ldots, p\}$
9: `queue`$[q]$`.push`$((j, \mathbf{h}_j))$
10: **end for**
11: `// start` p `workers`
12: **Parallel Foreach** $q \in \{1, 2, \ldots, p\}$
13: **while** stop signal is not yet received **do**
14: **if** `queue`$[q]$ not empty **then**
15: $(j, \mathbf{h}_j) \leftarrow$ `queue`$[q]$`.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: $q' \sim \text{UniformDiscrete}\{1, 2, \ldots, p\}$
23: `queue`$[q']$`.push`$((j, \mathbf{h}_j))$
24: **end if**
25: **end while**
26: **Parallel End**

3.1 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 $1/p$ 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 `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 processors 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 processor is fixed, that is, $|\Omega|$ increases and the number of processors p increases proportionally. In this case, the average time $a \cdot (|\Omega| k/np)$ to process an item remains constant, and NOMAD continues to scale linearly. In contrast, bulk synchronization based algorithms will slow down because the communication cost of a synchronous algorithm grows as a function of p; they have to wait for the slowest worker.

3.2 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 machines might process updates at different rates dues 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 who has fewer items in its queue to process. To do this, a payload carrying information about the size of the `queue`$[q]$ is added to the messages that the workers send each other. The overhead of the payload 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.

3.3 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. Inter-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 an item variable more than once was empirically found to not improve convergence, and hence is not used in our algorithm.

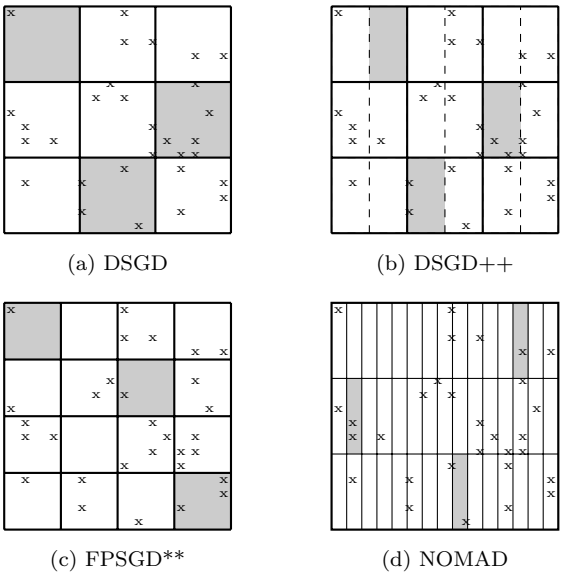
3.4 Implementation Details

Multi-threaded MPI was used for inter-machine communication. Instead of communicating single $(j, \mathbf{h}_j)$ pairs, we follow the strategy of [24], 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 among threads in NOMAD, it is important for each thread to use thread-local cache as much as possible to exploit memory locality. Therefore, memory assignments in each thread are carefully aligned with cache lines such that threads do not interfere with each other.

3.5 Comparison with other Algorithms



(a) DSGD (b) DSGD++

(c) FPSGD** (d) NOMAD

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

We now discuss how NOMAD is related to existing algorithms in terms of the data partitioning strategy. Recall that 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 1×2 sub-blocks (Figure 4 (a) and (b)). On the other hand, FPSGD** splits A into $p' \times p'$ blocks with $p' > p$, such that there always exists an available block which does not conflict with an assigned block of any other worker (Figure 4 (c)); the increased amount of flexibility in assignment of jobs to workers leads to a better parallelization performance. Indeed, it is common in parallel computing that a finer granularity in defining parallel tasks leads to better potential parallelism. In this respect, $p \times n$ partitioning of NOMAD (Figure 4 (d)) is the best one can achieve in the distributed memory setting.

4. EXPERIMENTS

In this section, we evaluate the empirical performance of NOMAD with extensive experiments. For the distributed memory experiments we compare NOMAD with DSGD [26], and CCD++ [28]. For the single machine multi-threaded experiments we pitch NOMAD against FPSGD** [31] (which is shown to outperform DSGD in single machine experiments) as well as CCD++. Our experiments are designed to answer the following questions:

- How does NOMAD scale with the number of cores on a single machine? (Section 4.2)
- How does NOMAD scale as a fixed size dataset is distributed across multiple machines? (Section 4.3)
- How does NOMAD perform on a commodity hardware cluster? (Section 4.4)
- How does NOMAD scale when both the size of the data as well as the number of machines grow? (Section 4.5)

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. For our experiments all algorithms were initialized by the same distribution, and the default setting provided by the authors of FPSGD** and CCD++ was used. For DSGD, which we had to implement ourselves, we closely followed the recommendations of Gemulla et al. [11] and [26], and in some cases made improvements based on our experience.

Two comparators are notably missing from our empirical evaluation: DSGD++ [26] and GraphLab [19]. Unfortunately the code for DSGD++ is not publicly available (personal communication with the authors), and is non-trivial to implement. On the other hand, in spite of our best efforts, the quality of results produced by the collaborative filtering toolbox of GraphLab is considerably worse than the other methods. We are communicating with the developers, and will include GraphLab results in the camera ready version if available.

4.1 Experimental Setup

For all experiments, except the ones in Section 4.5, we will work with three benchmark datasets namely Netflix, Yahoo! Music, and Hugewiki (see Table 1 for more details). 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 λ for each dataset as reported by Yu et al. [28], and shown in Table 2; 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 use the same initial parameters; we set each entry of W and H by independently sampling an uniformly random variable in the range $(0, \frac{1}{\sqrt{k}})$ [28, 31].

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

$$\sqrt{\frac{\sum_{(i,j)\in\Omega^{\text{test}}} \left(A_{ij} - \langle \mathbf{w}_i, \mathbf{h}_j \rangle\right)^2}{|\Omega^{\text{test}}|}}, \tag{13}$$

where Ω^{test} denotes the ratings in the test set.

All experiments, except the ones reported in Section 4.4, 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 4.2), 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 4.4 we used `m1.xlarge` instances of Amazon Web Services, which are equipped with 15GB of RAM and four cores. Here, we

utilized all four cores in each machine; NOMAD uses 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.

For a fair comparison both CCD++ and DSGD were tuned for optimal performance on our hardware. For FPSGD** experiments, we used the LibMF library [31]. Since FPSGD** uses single precision arithmetic, the experiments in Section 4.2 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 4.4 where we used a 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

$$s_t = \frac{\alpha}{1 + \beta \cdot t^{1.5}}, \tag{14}$$

where t is the number of SGD updates that were performed on a particular user-item pair (i, j). DSGD, on the other hand, uses an alternative strategy called bold-driver [11]; here, the step size is adapted by monitoring the change of the objective function. In DSGD, this heuristic is reported to show the best performance among many other step-size schedules, and therefore used in our experiments.

Parameters used in our experiments are summarized in Table 2. The scripts required for reproducing the experiments are readily available for download from `https://sites.google.com/site/hyokunyun/software`.

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

Table 1: Dataset Details

Name	k	λ	α	β
Netflix	100	0.05	0.012	0.05
Yahoo! Music	100	1.00	0.00075	0.01
Hugewiki	100	0.01	0.001	0

Table 2: Dimensionality parameter k, regularization parameter λ (1) and step-size schedule parameters α, β (14)

Section	Exception
Section 4.2	• run on `largemem` queue (32 cores, 1TB RAM) • single precision floating point used
Section 4.4	• run on `m1.xlarge` (4 cores, 15GB RAM) • compiled with `gcc` • `MPICH2` for MPI implementation
Section 4.5	• Synthetic datasets

Table 3: Exceptions to each experiment

4.2 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** and CCD++ on the Netflix, Yahoo! Music, and Hugewiki datasets[1](Figure 5). On Netflix dataset (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 the Yahoo! Music dataset (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. [31] 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, 7, and 8). Figure 6 shows how test RMSE changes as a function of the number of updates. 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 3.1 for mathematical analysis). This effect was more strongly observed on Yahoo! Music dataset 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.

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 7, while varying the number of cores. If NOMAD exhibits linear scaling in terms of the speed it processes ratings, the average throughput should remain constant[2]. On Netflix dataset (left), the average throughput indeed remains almost constant as the number of cores changes. On Yahoo! Music and Hugewiki datasets (center and right), 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 and are investigating this.

Now we study how much speed-up NOMAD can achieve by increasing the number of cores. In Figure 8, 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

[1]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.

[2]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.

we increase the number of cores. This is indeed the case for Netflix and Hugewiki datasets. In the case of Yahoo! Music dataset we observe that the speed of convergence actually 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.

4.3 Scaling as a Fixed Dataset is Distributed Across Processors

In this subsection, we use 4 computation threads per each machine. For the first experiment we fix the number of machines to 32 (64 for hugewiki), and compare the performance of NOMAD with DSGD and CCD++ on the Netflix, Yahoo! Music, and Hugewiki datasets (Figure 9). On Netflix and Hugewiki datasets, NOMAD converges much faster than its competitors; not only initial convergence is faster, it also discovers a better quality solution. On Yahoo! Music dataset, three methods perform almost the same to each other. This is because the cost of network communication relative to the size of the data is much higher for the Yahoo! Music dataset; while Netflix and Hugewiki datasets 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 dataset 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, $\bar{\Omega}_j^{(q)}$. 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.

For the second experiment we varied the number of machines from 1 to 32, and plot the scaling behavior of NOMAD (Figures 10, 11 and 12). Figure 10 shows how test RMSE decreases as a function of the number of updates. Again, if NOMAD scales linearly the average throughput has to remain constant. On the Netflix dataset (left) convergence is mildly slower with two or four machines. However, as we increase the number of machines the speed of convergence improves. On Yahoo! Music (center), we uniformly 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 4.2. On the Hugewiki dataset, however, we do not see any notable difference between configurations.

In Figure 11 we plot the average throughput (the number of updates per machine per core per second) as a function of the number of machines. On the Yahoo! Music dataset 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 $\mathbf{w}_i$'s within the block decrease, and on Netflix dataset this makes a meaningful difference: indeed, there are only 480,189 users in Netflix dataset 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 $\mathbf{w}_i$ 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 12, 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.

4.4 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[3].

Since NOMAD dedicates two threads for network communication, on each machine only two cores are available for computation. In contrast, synchronous algorithms such as DSGD and CCD++ which separate computation and communication can utilize all four cores for computation. In spite of this disadvantage, Figure 13 shows that NOMAD outperforms DSGD and CCD++ in this setting as well. In this plot, we fixed the number of machines to 32; on Netflix and Hugewiki datasets, NOMAD converges more rapidly to a better solution. Recall that on Yahoo! Music dataset, all three algorithms performed very similarly on a HPC cluster in Section 4.3. 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 dataset, 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 4.3, 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, 11, and 12; due to page constraints, the plots for this experiment can be found in the Appendix C.

4.5 Scaling as both Dataset Size and Number of Machines Grows

In previous sections (Section 4.3 and Section 4.4), 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 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

[3] `http://epamcloud.blogspot.com/2013/03/testing-amazon-ec2-network-speed.html`

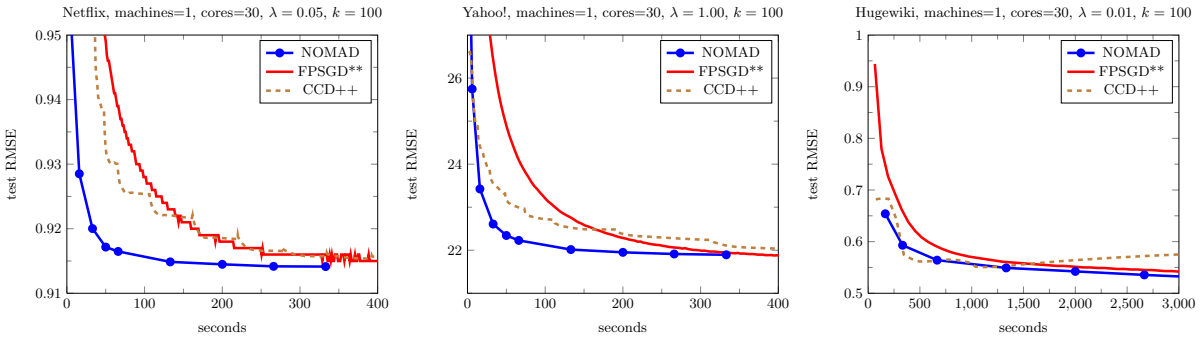


Figure 5: Comparison of NOMAD and FPSGD** on a single-machine with 30 computation cores. All algorithms are initialized with the same parameter values.

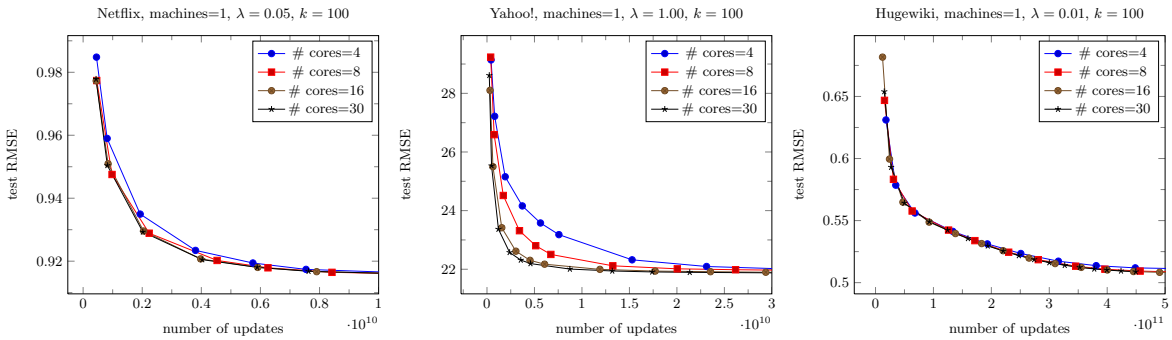


Figure 6: Test RMSE of NOMAD as a function of the number of updates, when the number of cores is varied.

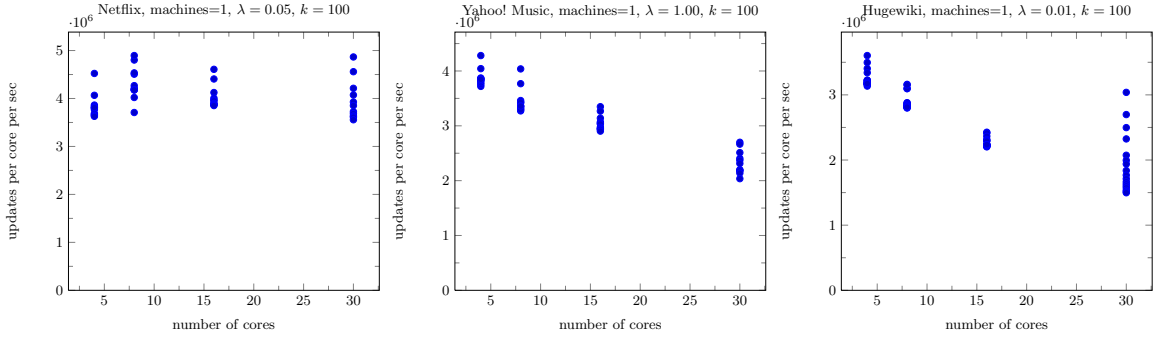


Figure 7: Number of updates of NOMAD per core per second as a function of the number of cores.

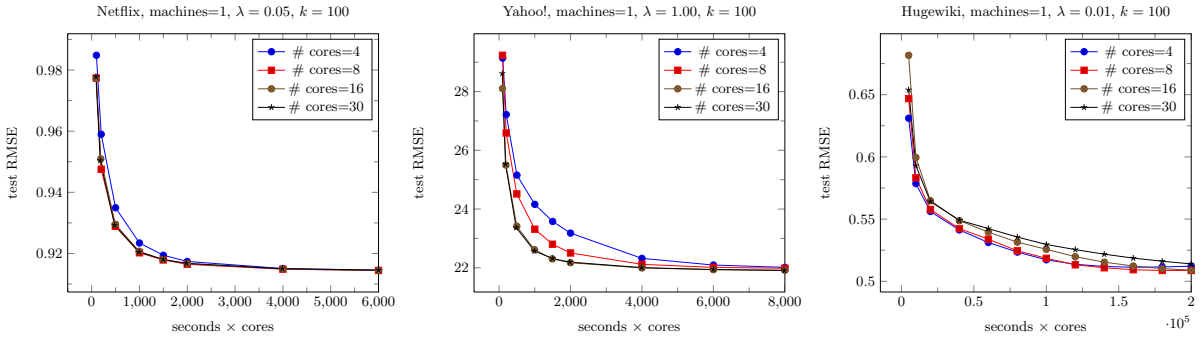


Figure 8: Test RMSE of NOMAD as a function of computation time (time in seconds × the number of cores), when the number of cores is varied.

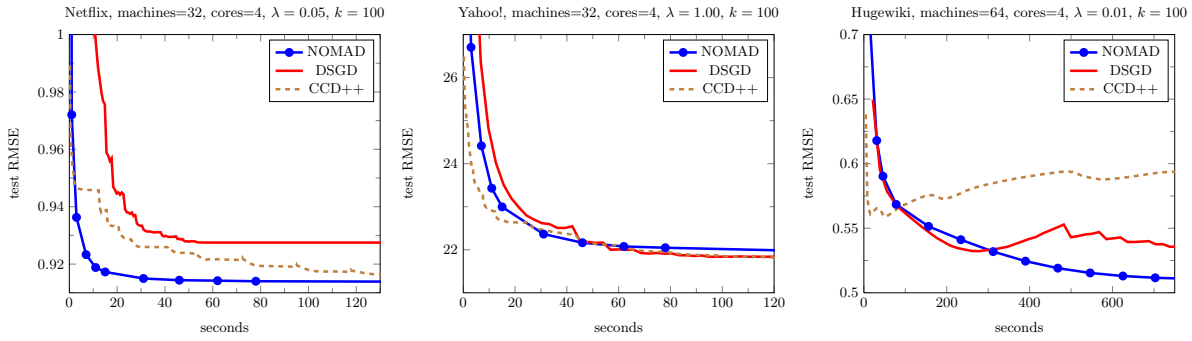


Figure 9: Comparison of NOMAD, DSGD and CCD++ on a HPC cluster.

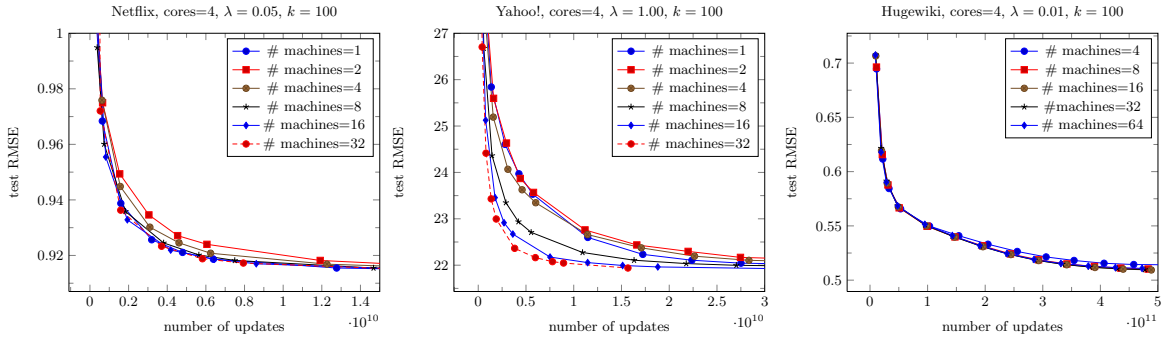


Figure 10: Test RMSE of NOMAD as a function of the number of updates on a HPC cluster, when the number of machines is varied.

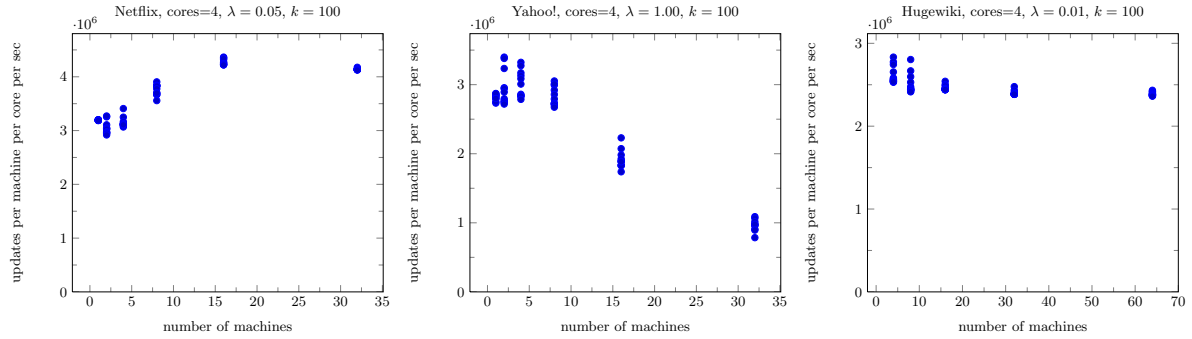


Figure 11: Number of updates of NOMAD per machine per core per second as a function of the number of machines, on a HPC cluster.

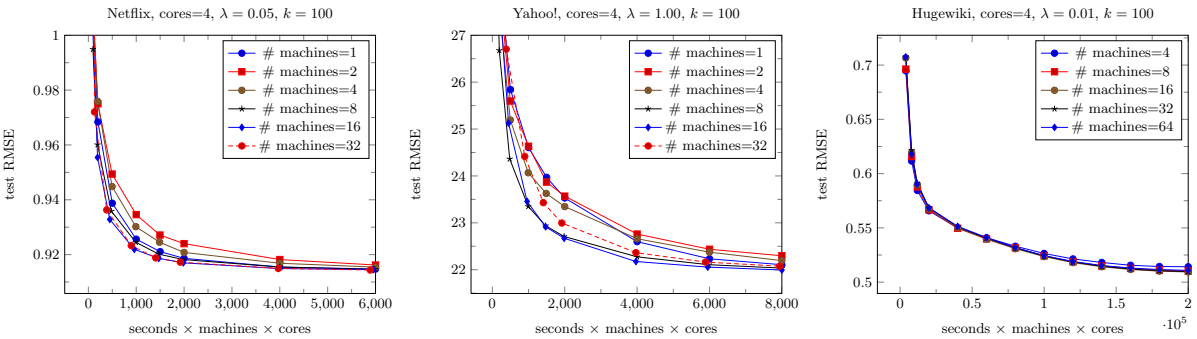


Figure 12: Test RMSE of NOMAD as a function of computation time (time in seconds × the number of machines × the number of cores per each machine) on a HPC cluster, when the number of machines is varied.

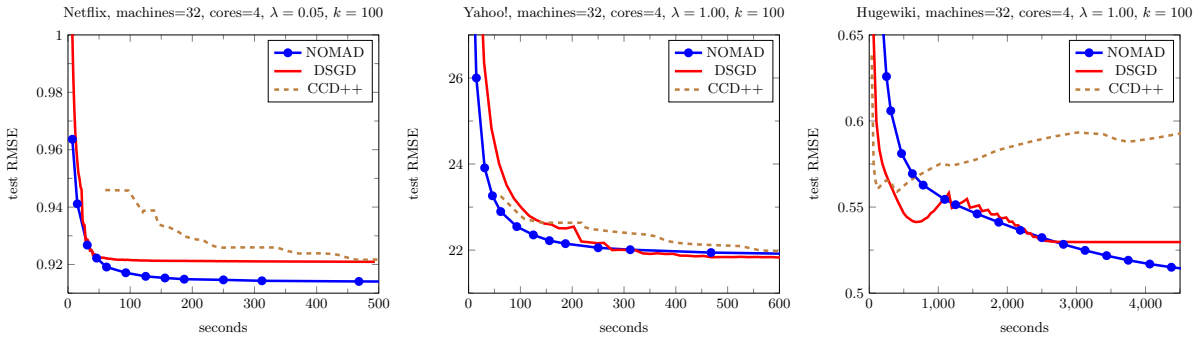


Figure 13: Comparison of NOMAD, DSGD and CCD++ on a commodity hardware cluster.

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 which resemble characteristics of 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 number of machines (480,189 $\times$ the number of machines[4]). 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 isometric 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 14 shows that the comparative advantage of NOMAD against DSGD and CCD++ increases as we grow the scale of the problem. NOMAD clearly outperforms DSGD on all configurations; DSGD is very competitive on the small scale, but as the size of the problem grows NOMAD shows better scaling behavior.

5. CONCLUSION AND FUTURE WORK

In this paper, we argued that asynchronous and decentralized algorithms like NOMAD should outperform synchronous algorithms on large-scale systems, and showed NOMAD's empirical advantage over other competitive methods via extensive experiments.

Several interesting questions remain to be answered, however. First, how can we extend the idea of using *nomadic* variables to other statistical machine learning models? Currently, we are working on applying NOMAD on at least two more important problems, regularized risk minimization and latent Dirichlet allocation (LDA). Second, we observed that partitioning the rating matrix A into smaller blocks increases the cost of communication, but improves the quality of each update. How can we balance this trade-off to maximize the performance?

[4]480,189 is the number of users in the Netflix dataset who have at least one rating.

References

[1] Apache hadoop, 2009. http://hadoop.apache.org/core/.

[2] Graphlab datasets, 2013. http://graphlab.org/downloads/datasets/.

[3] Intel thread building blocks, 2013. https://www.threadingbuildingblocks.org/.

[4] A. Agarwal, O. Chapelle, M. Dudík, and J. Langford. A reliable effective terascale linear learning system. *CoRR*, abs/1110.4198, 2011.

[5] R. M. Bell and Y. Koren. Lessons from the netflix prize challenge. *SIGKDD Explorations*, 9(2):75–79, 2007. URL `http://doi.acm.org/10.1145/1345448.1345465`.

[6] D. P. Bertsekas and J. N. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods*. Athena Scientific, 1997.

[7] L. Bottou and O. Bousquet. The tradeoffs of large-scale learning. *Optimization for Machine Learning*, page 351, 2011.

[8] C.-T. Chu, S. K. Kim, Y.-A. Lin, Y. Yu, G. Bradski, A. Y. Ng, and K. Olukotun. Map-reduce for machine learning on multicore. In B. Schölkopf, J. Platt, and T. Hofmann, editors, *Advances in Neural Information Processing Systems 19*, pages 281–288. MIT Press, 2006.

[9] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *CACM*, 51(1):107–113, 2008. URL `http://doi.acm.org/10.1145/1327452.1327492`.

[10] 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.

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

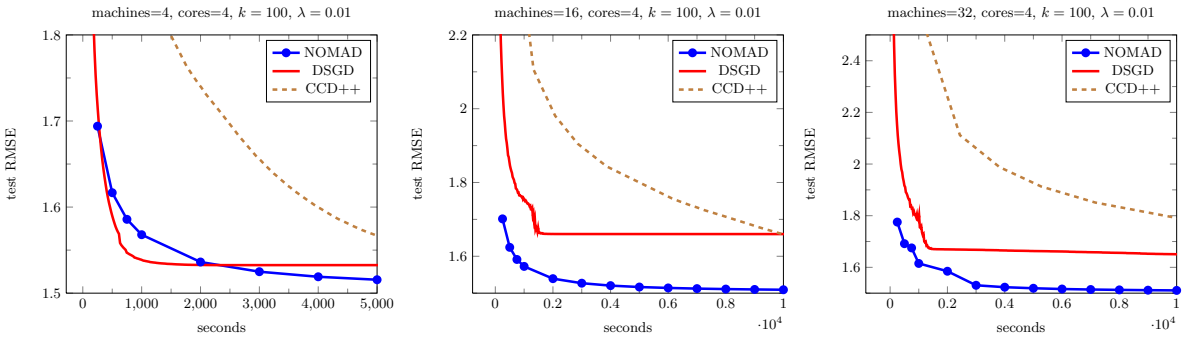


Figure 14: Comparison of algorithms when both dataset size and the number of machines grows. Left: 4 machines, middle: 16 machines, right: 32 machines

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

[13] J. L. Hennessy and D. A. Patterson. *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufman, 4 edition, 2009.

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

[15] T. Kraska, A. Talwalkar, J. C. Duchi, R. Griffith, M. J. Franklin, and M. I. Jordan. Mlbase: A distributed machine-learning system. In *CIDR*, 2013.

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

[17] J. Langford, A. J. Smola, and M. Zinkevich. Slow learners are fast. arXiv:0911.0491, 2009. URL `http://arxiv.org/abs/0911.0491`.

[18] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. GraphLab: A new parallel framework for machine learning. In *Conference on Uncertainty in Artificial Intelligence*, 2010. URL `https://select.cs.cmu.edu/code/graphlab/index.html`.

[19] 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 *PVLDB*, 2012.

[20] B. Recht, C. Re, S. Wright, and F. Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In P. Bartlett, F. Pereira, R. Zemel, J. Shawe-Taylor, and K. Weinberger, editors, *Advances in Neural Information Processing Systems 24*, pages 693–701, 2011. URL `http://books.nips.cc/nips24.html`.

[21] P. Richtarik and M. Takac. Distributed coordinate descent method for learning with big data. Technical report, 2013. URL `"http://arxiv.org/abs/1310.2059"`.

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

[23] S. Shalev-Schwartz and N. Srebro. SVM optimization: Inverse dependence on training set size. In W. Cohen, A. McCallum, and S. Roweis, editors, *Proceedings of the 25th Annual International Conference on Machine Learning (ICML 2008)*. Omnipress, 2008.

[24] A. J. Smola and S. Narayanamurthy. An architecture for parallel topic models. In *Very Large Databases (VLDB)*, 2010.

[25] S. Suri and S. Vassilvitskii. Counting triangles and the curse of the last reducer. In S. Srinivasan, K. Ramamritham, A. Kumar, M. P. Ravindra, E. Bertino, and R. Kumar, editors, *Conference on World Wide Web*, pages 607–614. ACM, 2011. URL `http://doi.acm.org/10.1145/1963405.1963491`.

[26] C. Teflioudi, F. Makari, and R. Gemulla. Distributed matrix completion. In *Data Mining (ICDM), 2012 IEEE 12th International Conference on*, pages 655–664. IEEE, 2012.

[27] C. H. Teo, S. V. N. Vishwanthan, A. J. Smola, and Q. V. Le. Bundle methods for regularized risk minimization. *Journal of Machine Learning Research*, 11: 311–365, January 2010.

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

[29] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *HotCloud 2010*, June 2010.

[30] Y. Zhou, D. Wilkinson, R. Schreiber, and R. Pan. Large-scale parallel collaborative filtering for the netflix prize. In *Proceedings of the 4th international conference on Algorithmic Aspects in Information and Management*, pages 337–348, 2008.

[31] Y. Zhuang, W.-S. Chin, Y.-C. Juan, and C.-J. Lin. A fast parallel sgd for matrix factorization in shared memory systems. In *Proceedings of the 7th ACM conference on Recommender systems*, pages 249–256. ACM, 2013.

APPENDIX

A. EFFECT OF THE REGULARIZATION PARAMETER

In this subsection, we study the convergence behavior of NOMAD as we change the regularization parameter λ (Figure 15). Note that in Netflix data (left), for non-optimal choices of the regularization parameter the test RMSE increases from the initial solution as the model overfits or underfits to the training data. While NOMAD reliably converges in all cases, on Netflix dataset the convergence is notably faster with higher values of λ; this is expected because regularization smooths the objective function and makes the optimization problem easier to solve. On other datasets, the speed of convergence was not very sensitive to the selection of the regularization parameter.

B. EFFECT OF THE LATENT DIMENSION

In this subsection, we study the convergence behavior of NOMAD as we change the dimensionality parameter k (Figure 16). In general, the convergence is faster for smaller values of k as the computational cost of SGD updates (9) and (10) is linear to k. On the other hand, the model gets richer with higher values of k, as its parameter space expands; it becomes capable of picking up weaker signals in the data, with the risk of overfitting. This is observed in Figure 16 with Netflix (left) and Yahoo! Music (right) datasets. In Hugewiki dataset, however, small values of k were sufficient to fit the training data, and test RMSE suffers from overfitting with higher values of k. Nonetheless, NOMAD reliably converged in all cases.

C. SCALING ON COMMODITY HARDWARE

In this section, we augment Section 4.4 by providing actual plots of the experiment. We increase the number of machines from 1 to 32, and plot how the convergence of NOMAD is affected by the number of machines. As in Section 4.4, we used `m1.xlarge` machines from Amazon Web Servies (AWS) which have quad-core Intel Xeon E5430 CPU and 15GB of RAM per each.

The overall pattern is identical to what was found in Figure 10, 11 and 12 of Section 4.3. Figure 17 shows how the test RMSE decreases as a function of the number of updates. As in Figure 10, the speed of convergence is faster with larger number of machines as the updated information is more frequently exchanged. Figure 18 shows the number of updates performed per second in each computation core of each machine; NOMAD exhibits linear scaling on Netflix and Hugewiki datasets, but slows down on Yahoo! Music dataset due to extreme sparsity of the data. Figure 19 compares the convergence speed of different settings when the same amount of computational power is given to each; on every dataset we observe linear to super-linear scaling up to 32 machines.

D. COMPARISON OF ALGORITHMS FOR DIFFERENT VALUES OF THE REGULARIZATION PARAMETER

In this section, we augment experiments in Section 4.3 by comparing the performance of NOMAD, CCD++, and DSGD on different values of the regularization parameter λ. Figure 20 shows the result of the experiment. As NOMAD and DSGD are both stochastic gradient descent methods, they behave similarly to each other when the regularization parameter is changed. On the other hand, CCD++, which decreases the objective function more greedily, behaves a differently.

For small values of λ, CCD++ seems to overfit to the model due to its greedy strategy; it generally converges to a worse solution than others. For high values of λ, however, the strategy of CCD++ is advantageous and it shows rapid initial convergence. Note that in all cases, NOMAD is competitive with the better of the other two algorithms.

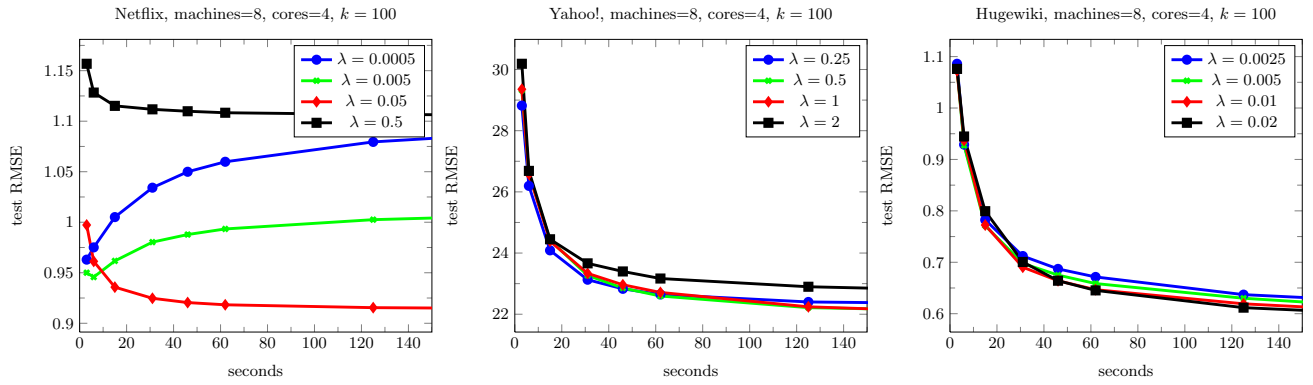


Figure 15: Convergence behavior of NOMAD when the regularization parameter λ is varied.

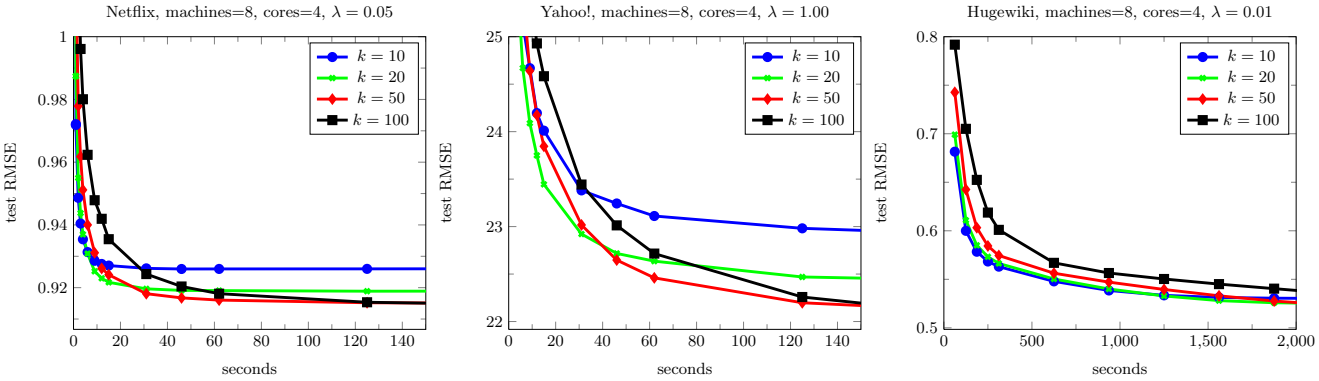


Figure 16: Convergence behavior of NOMAD when the latent dimension k is varied.

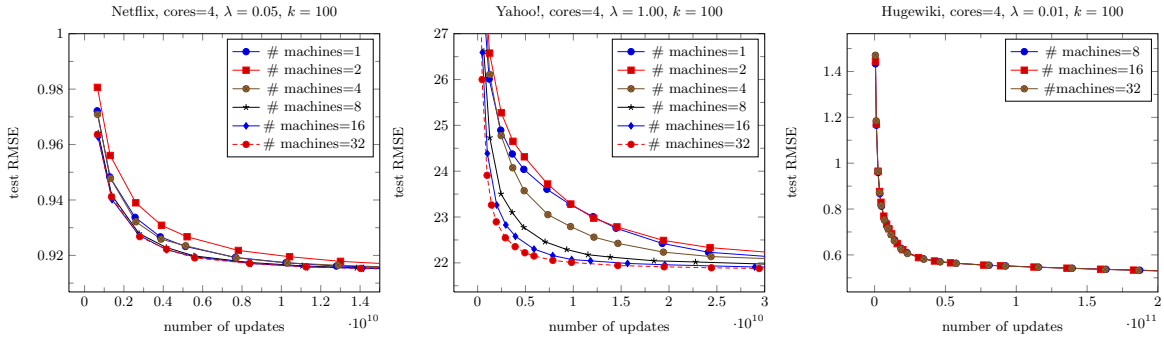


Figure 17: Test RMSE of NOMAD as a function of the number of updates on a commodity hardware cluster, when the number of machines is varied.

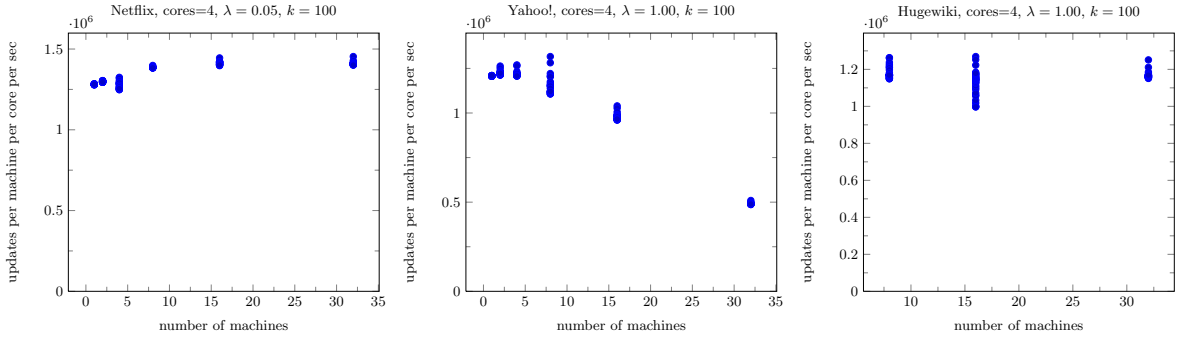


Figure 18: Number of updates of NOMAD per machine per core per second as a function of the number of machines, on a commodity hardware cluster.

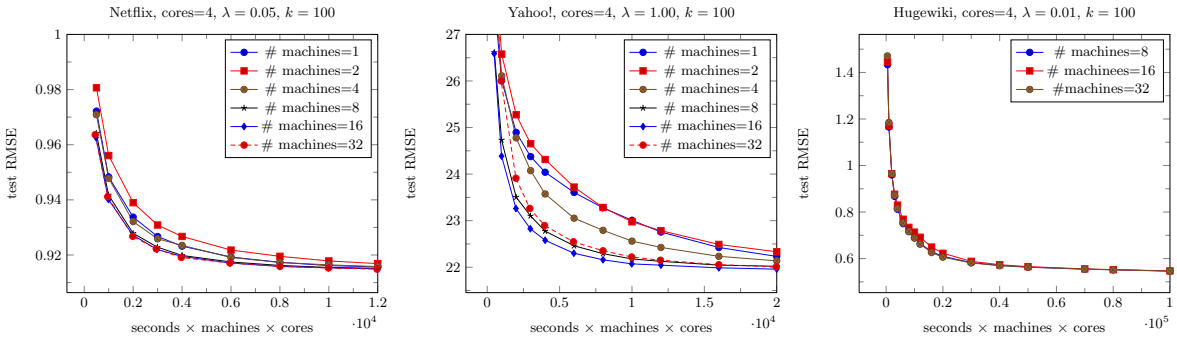


Figure 19: Test RMSE of NOMAD as a function of computation time (time in seconds × the number of machines × the number of cores per each machine) on a commodity hardware cluster, when the number of machines is varied.

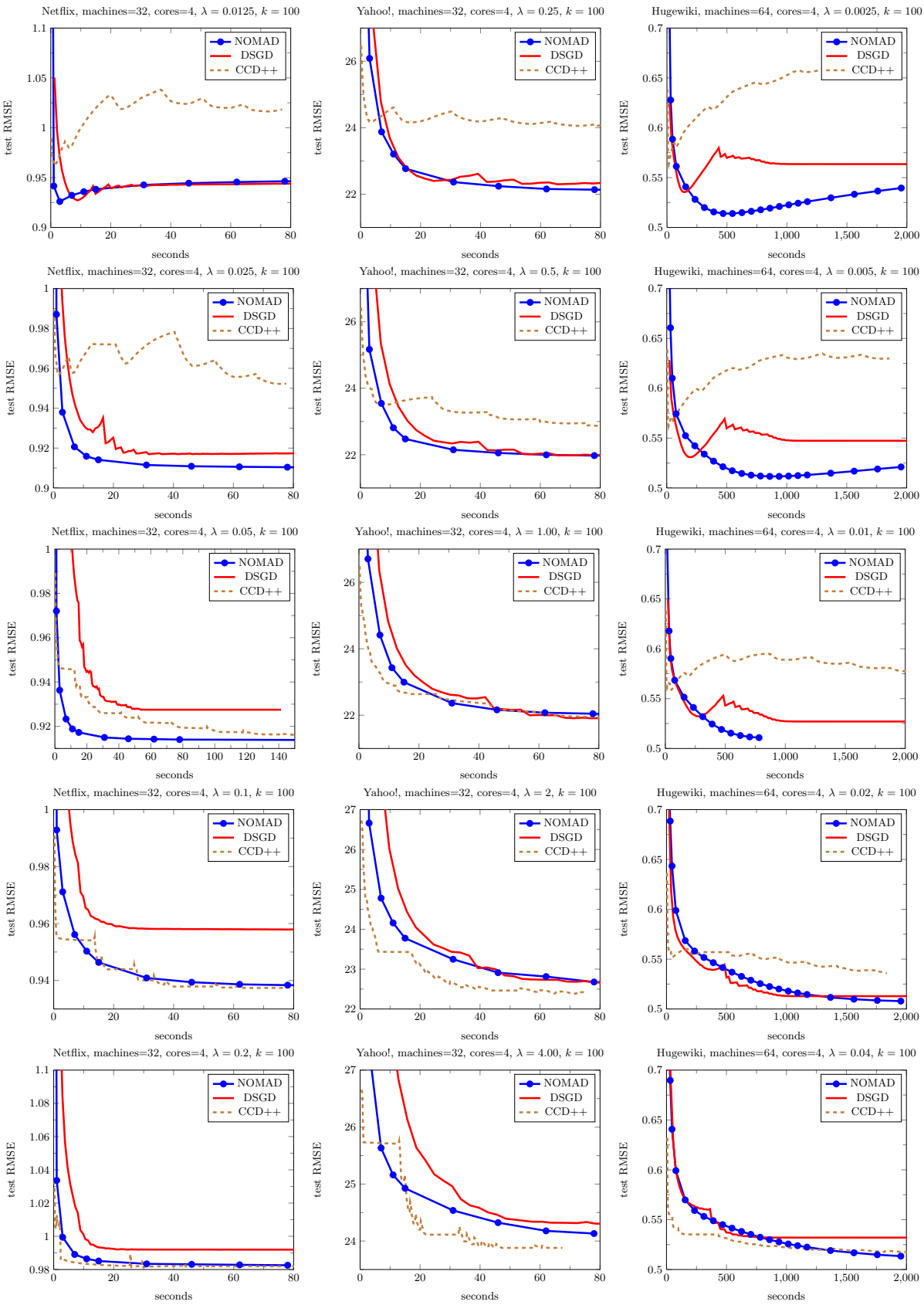


Figure 20: Comparison of NOMAD, DSGD and CCD++ on a HPC cluster when the regularization paramter λ is varied. The value of λ increases from top to bottom.