Nomadic Computing for Big Data Analytics

Hsiang-Fu Yu, University of Texas at Austin
Cho-Jui Hsieh, University of California, Davis
Hyokun Yun, Amazon.com
S.V.N. Vishwanathan, University of California, Santa Cruz
Inderjit Dhillon, University of Texas at Austin

Analyzing the massive datasets of today's applications will require scalable and sophisticated machine-learning methods. NOMAD, a novel nomadic framework, combines two common approaches: stochastic optimization and distributed computing.

Today's applications often contain datasets that are too big to fit in a single computer's main memory. Analyzing these massive datasets will require scalable and sophisticated machine-learning methods. Two commonly used approaches are stochastic optimization and inference algorithms,[1] which process one data point at a time; and distributed computing based on the MapReduce framework,[2] where the computation proceeds in iterations, with a master processor distributing the computation to slaves at each iteration. Although stochastic optimization and inference algorithms are effective for large-scale machine learning, they are inherently sequential. On the other hand, MapReduce-based algorithms suffer from the curse of the last reducer, in that the slaves must wait for the slowest processor to finish before moving on to the next computational iteration.

In this article, we describe NOMAD, a novel nomadic framework that combines stochastic optimization's and distributed computing's advantages without incurring their drawbacks. NOMAD is an acronym for Nonlocking, stOchastic Multimachine framework for Asynchronous and Decentralized computation. We show that many modern machine-learning problems have a double separability property, meaning the objective function decomposes into a sum over two different variables. We use two concrete problems to illustrate our framework: matrix completion for recommender systems and latent Dirichlet allocation for topic modeling.

MATRIX COMPLETION

In applications such as recommender systems, gene–disease interactions in bioinformatics, and link predictions in social-network analysis, we observe incomplete

0018-9162/16/$33.00 © 2016 IEEE

interactions between two different kinds of entities. For instance, when users are interacting with movies, the interactions might be implicit (the user watched the movie) or explicit (the user reviewed and rated the movie). Given the observed interactions, we must infer the unobserved interactions. This challenge has great practical significance because it often underlies the systems that e-commerce websites use to recommend ads, products, news articles, and movies to users.[3]

Mathematically, the problem can be formulated as follows. Let $A \in R^{m \times n}$ be an interaction matrix, where m denotes the number of users and n the number of items. Typically $m \gg n$. Furthermore, let $\Omega \in [1, \ldots, m] \times [1, \ldots, n]$ denote the observed entries of A, that is, $(i, j) \in \Omega$ implies that the interaction between user i and item j has a value of A_{ij}. The goal is to accurately predict the unobserved entries of A. For convenience, we define Ω_i as the set of interactions observed for the ith user, that is, $\Omega_i := \{j: (i,j) \in \Omega\}$. Analogously, $\bar{\Omega}_j := \{i: (i,j) \in \Omega\}$ is the set of users who have interacted with item j. Also, let $\boldsymbol{a}_i^{\mathrm{T}}$ denote the ith row of A.

One popular model for matrix completion finds matrices $W \in R^{m \times k}$ and $H \in R^{n \times k}$ with $k \ll \min(m,n)$, such that $A \approx WH^{\mathrm{T}}$.[3] One way to understand this model is to think of each row $\boldsymbol{w}_i^{\mathrm{T}} \in R^k$ of W as a k-dimensional embedding of the user. Analogously, each row $\boldsymbol{h}_j^{\mathrm{T}} \in R^k$ of H is an embedding of the item in the same k-dimensional space. To predict the (i,j)th entry of A, we simply use $\langle \boldsymbol{w}_i, \boldsymbol{h}_j \rangle$, where $\langle \cdot, \cdot \rangle$ denotes the Euclidean inner product of two vectors. The model's goodness of fit is measured by a loss function, typically given by $1/2(A_{ij} - \langle \boldsymbol{w}_i, \boldsymbol{h}_j \rangle)^2$. Furthermore, we need to enforce regularization to prevent overfitting and to properly predict the unknown A entries. For instance, a popular regularizer is

$$\frac{\lambda}{2}\sum_{i=1}^{m}\left|\Omega_i\right| \cdot \left\|\boldsymbol{w}_i\right\|^2 + \frac{\lambda}{2}\sum_{j=1}^{n}\left|\bar{\Omega}_j\right| \cdot \left\|\boldsymbol{h}_j\right\|^2,$$

where $\lambda > 0$ is a tunable parameter. Here, $|\cdot|$ denotes the cardinality of a set, and $\|\cdot\|$ is the L_2 norm of a vector. Putting everything together yields the following empirical risk-minimization problem:

$$\min_{W,H} J(W,H) := \frac{1}{2}\left\{\sum_{(i,j)\in\Omega}\left(A_{ij} - \left\langle \boldsymbol{w}_i, \boldsymbol{h}_j \right\rangle\right)^2 + \text{regularization on } W \text{ and } H\right\}.$$

Stochastic optimization

Stochastic gradient descent (SGD) is a popular stochastic optimization technique,[1] which replaces $J(W,H)$ by the instantaneous approximation

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

The gradient of this objective function is easily computed as

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

and

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

and is used to update the parameters

$$\boldsymbol{w}_{i_t} \leftarrow \boldsymbol{w}_{i_t} - s_t\left[-(A_{ij} - \left\langle \boldsymbol{w}_i, \boldsymbol{h}_j \right\rangle)\boldsymbol{h}_{j_t} + \lambda \boldsymbol{w}_{i_t}\right] \quad (1)$$

$$\boldsymbol{h}_{j_t} \leftarrow \boldsymbol{h}_{j_t} - s_t\left[-(A_{ij} - \left\langle \boldsymbol{w}_i, \boldsymbol{h}_j \right\rangle)\boldsymbol{w}_{i_t} + \lambda \boldsymbol{h}_{j_t}\right], \quad (2)$$

where s_t is the learning rate for the tth update. Thus, stochastic optimization requires sampling a random index (i_t, j_t) uniformly from the set of nonzero indices Ω, and performing updates 1 and 2.

NOMAD: A novel parallel scheme for matrix completion

Note that SGD updates 1 and 2 only require us to read $\boldsymbol{w}_i$, $\boldsymbol{h}_j$, and A_{ij} for some $(i,j) \in \Omega$, and to update $\boldsymbol{w}_i$ and $\boldsymbol{h}_j$ (see Figure 1). As a result, we can simultaneously perform multiple SGD updates in parallel. The updates will not interfere with one other as long as we ensure they are not reading or writing the same $\boldsymbol{w}_i$ and $\boldsymbol{h}_j$ values. This observation forms the basis of NOMAD.

We refer to a parallel computing unit as a *worker*. In a shared memory setting, a worker is a thread; in a distributed memory architecture, it is a machine. This abstraction allows us to present NOMAD in a unified manner. Of course, NOMAD can also be used in a hybrid setting where multiple threads are spread across multiple machines. The users $\{1, \ldots, m\}$ are split into p disjoint sets $I_1, I_2, \ldots, I_p$, which are of approximately equal size. (An alternative strategy is to split the users such that each set has approximately the same number of ratings.) This induces a partition of the rows of the ratings matrix A. The qth worker stores n sets of indices $\bar{\Omega}_j^{(q)}$ for $j \in \{1, \ldots, n\}$, 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 algorithm's execution.

Recall that there are two types of

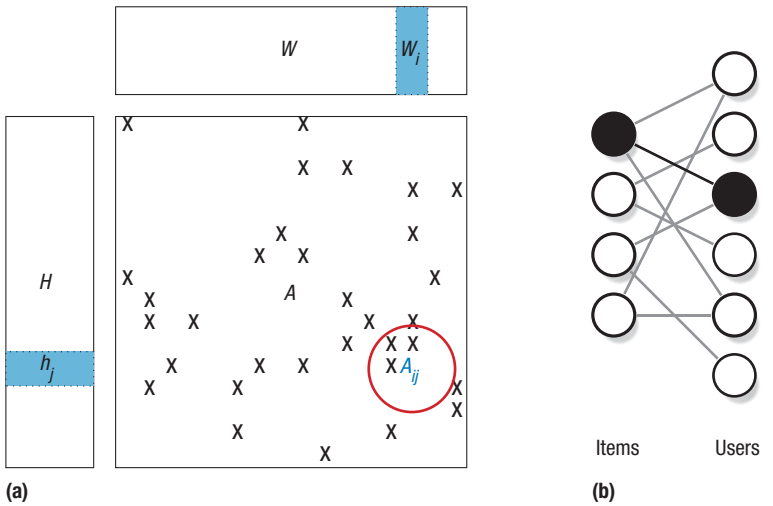

FIGURE 1. Access graph for stochastic optimization of the matrix-completion objective function. (a) Updating the parameters $\boldsymbol{w}_i$ and $\boldsymbol{h}_j$ requires access to $\boldsymbol{w}_i$, $\boldsymbol{h}_j$, and A_{ij}. (b) The same access pattern represented graphically. Black indicates that the node value is being updated, gray that the node value is being read, and white that the nodes are being neither updated nor read.

parameters in matrix completion: $\boldsymbol{w}_i$ user parameters and $\boldsymbol{h}_j$ item parameters. In NOMAD, $\boldsymbol{w}_i$ parameters are partitioned according to $I_1, I_2, \ldots, I_p$, that is, the qth worker stores and updates $\boldsymbol{w}_i$ for $i \in I_q$. The variables in W are partitioned at the beginning, and never move across workers during the algorithm's execution. On the other hand, $\boldsymbol{h}_j$ parameters are split randomly into p partitions at the beginning, and their ownership changes as the algorithm progresses. The $\boldsymbol{h}_j$ variables are nomadic: at each time point, one $\boldsymbol{h}_j$ variable resides in only one worker, moving to another worker after it is processed, independent of other item variables. (Due to symmetry in the formulation of the matrix-completion problem, one can also make $\boldsymbol{w}_i$ nomadic and partition $\boldsymbol{h}_j$. Because the number of users is usually much larger than the number of items, this leads to more communication; therefore, we made the $\boldsymbol{h}_j$ variables nomadic.)

Processing an item variable $\boldsymbol{h}_j$ at the qth worker entails executing SGD updates 1 and 2 on the ratings in the set $\overline{\Omega}_i^{(q)}$. These updates require access to only $\boldsymbol{h}_j$ and $\boldsymbol{w}_i$ for $i \in I_q$; because I_q are disjoint sets, only one worker accesses each $\boldsymbol{w}_i$ variable. This is why the communication of $\boldsymbol{w}_i$ variables is not necessary. On the other hand, $\boldsymbol{h}_j$ is updated only by the worker that currently owns it, so no lock is needed; this is parallel computing's popular owner-computes rule (see Figure 2).

The resultant algorithm has the following attractive properties:

- *Nonblocking communication*. Processors exchange messages asynchronously,[4] and there is no bulk synchronization.
- *Decentralized*. Workers are symmetric, and each worker does the same amount of computation and communication.
- *Lock free*. Owing to the owner-computes paradigm, the need for locking variables is completely eliminated.
- *Fully asynchronous computation*. Because the algorithm is lock free, the variable updates in individual processors are fully asynchronous.
- *Serializability*. There is an equivalent update ordering in a serial implementation. Stale parameters are never used, which empirically leads to faster convergence.[5]

We performed two experiments to demonstrate NOMAD's performance. For the first experiment on shared memory experiments, we pitted NOMAD against FPSGD**[6] (which is shown to outperform DSGD in single machine experiments) as well as CCD++. For the second experiment on distributed memory, we compared NOMAD with DSGD,[7] DSGD++,[8] and CCD++.[9] DSGD and CCD++ are synchronous algorithms, and DSGD++ and FPSGD** are variants of asynchronous SGD.

We worked with three benchmark datasets: Netflix, Yahoo! Music, and Hugewiki (see Table 1). We used the same training and test dataset partition for all algorithms in every experiment. Because our goal was to compare optimization algorithms, we did very minimal parameter tuning. For instance, we used the same regularization parameter λ for each dataset as reported by Hsiang-Fu Yu and his colleagues.[9] By default, we used $k = 100$ for the dimension of the latent space. We initialized all algorithms with the same initial parameters: each W and H entry was set by independently sampling a uniformly random variable in the range

$$\left(0, \frac{1}{\sqrt{k}}\right).^{6,9}$$

We compared solvers in terms of root mean square error (RMSE) on the test set, defined as

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

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

We ran all experiments at the

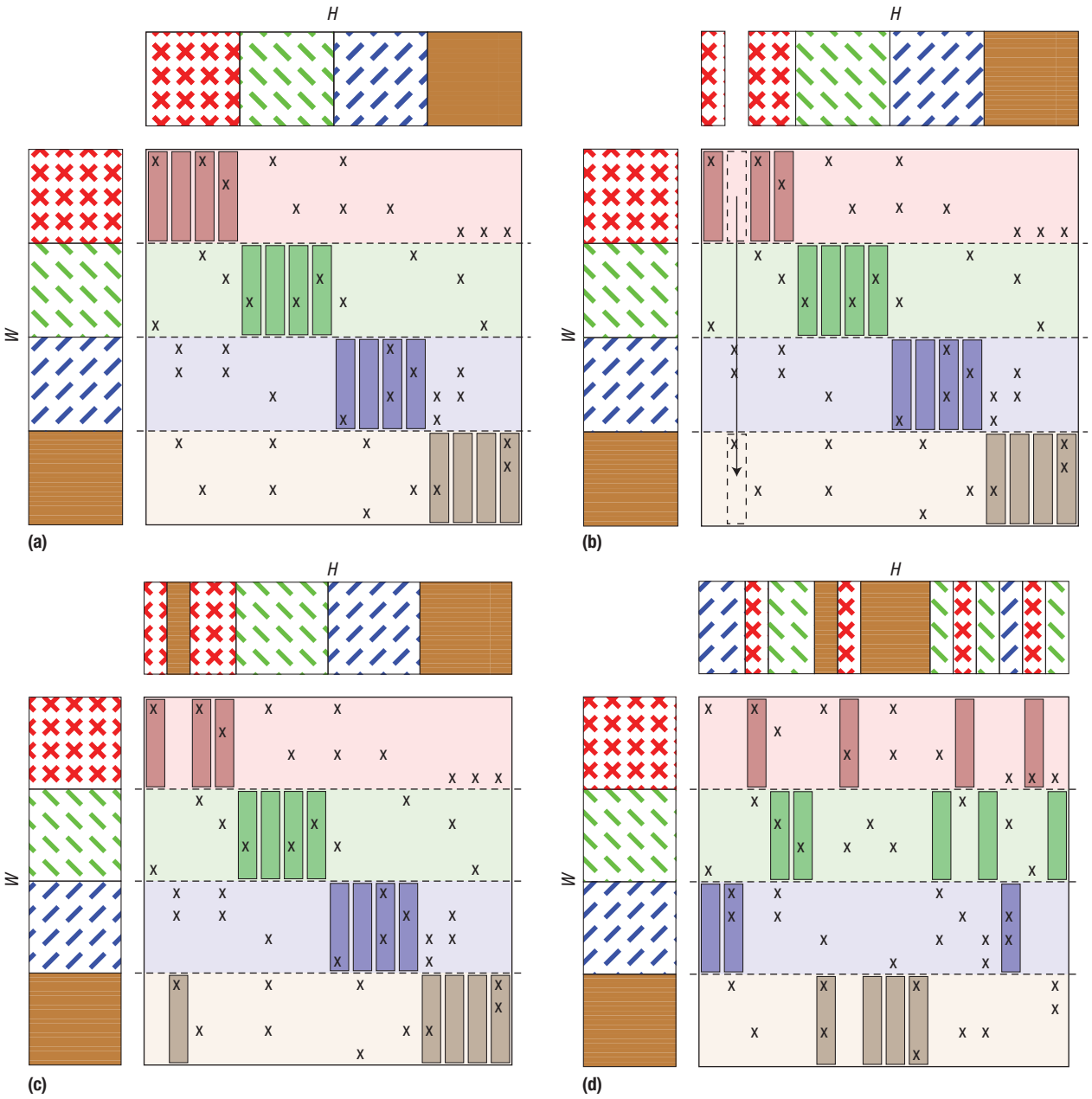

FIGURE 2. The NOMAD algorithm. Each x denotes an observed entry in the interaction matrix *A*. The ownership of data and variables is shown by different colors. Small rectangles in the middle denote each machine's active regions. (a) Initial assignment of matrices *W* and *H*. Each worker processes only the diagonal active area. (b) Once a worker finishes processing column *j*, it sends the corresponding item parameter $\boldsymbol{h}_j$ to another worker. Here, $\boldsymbol{h}_2$ is sent from worker 1 to worker 4. (c) Upon receipt, the column is processed by the new worker. Here, worker 4 can now process column 3 because it owns the column. (d) During the algorithm's execution, ownership of the $\boldsymbol{h}_j$ item parameters changes.

University of Texas at Austin using the Stampede Cluster, a Linux cluster in which each node is outfitted with two Intel Xeon E5 (Sandy Bridge) processors and an Intel Xeon Phi Coprocessor (Many Integrated Core Architecture). We used the nodes in the normal queue, which were equipped with 32 GBytes of RAM and 16 cores (only 4 of the 16 cores were used for computation). MVAPICH2 software handled intermachine communication.

SGD methods' convergence speed depends on the step-size schedule chosen. The schedule we used for NOMAD was

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

where t is the number of SGD updates

TABLE 1. Dataset details.

Dataset	Rows	Columns	Nonzeros
Netflix[3]	2,649,429	17,770	99,072,112
Yahoo! Music[10]	1,999,990	624,961	252,800,275
Hugewiki	50,082,603	39,780	2,736,496,604

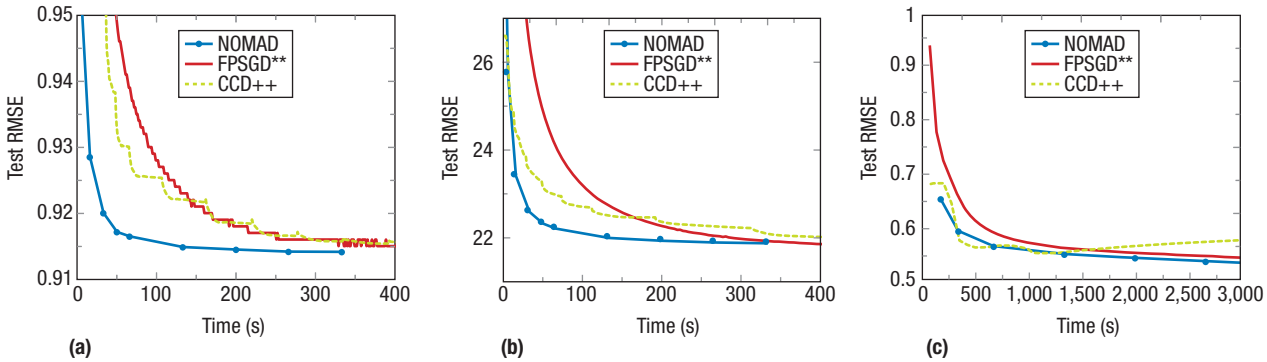

FIGURE 3. Comparison of NOMAD, FPSGD**, and CCD++ stochastic gradient descent on a single machine with 30 computation cores for three datasets: (a) Netflix ($\lambda = 0.05$, $k = 100$), (b) Yahoo! Music ($\lambda = 1.00$, $k = 100$), and (c) Hugewiki ($\lambda = 0.01$, $k = 100$). RMSE: root mean square error.

performed on a particular user–item pair (i,j). DSGD and DSGD++, on the other hand, use an alternative strategy called bold driver that adapts step size by monitoring the change of the objective function.[7]

Shared memory experiment

For the shared memory experiment (see Figure 3), we fixed the number of cores to 30 and compared the performance of NOMAD with CCD++ and FPSGD** (because the current FPSGD** implementation in LibMF only reports CPU execution time, we divided this by the number of threads and used this as a proxy for wall-clock time). On Netflix (Figure 3a), NOMAD not only converged to a slightly better quality solution than did the other methods (NOMAD RMSE was 0.914, versus 0.916 for FPSGD** and CCD++), but also rapidly reduced the RMSE from the beginning. On Yahoo! Music (Figure 3b), NOMAD converged to a slightly worse solution than FPSGD** (RMSE 21.894 versus 21.853, respectively), but as with Netflix, the initial convergence was more rapid. On Hugewiki (Figure 3c), the difference in RMSE was smaller but NOMAD still outperformed the other two methods. CCD++'s initial speed on Hugewiki was comparable to NOMAD, but its quality in terms of test RMSE started to deteriorate after 1,500.

Distributed memory experiment

For the distributed memory experiment (see Figure 4), we used four computation threads per machine, fixed the number of machines to 32 (64 for Hugewiki), and compared the performance of NOMAD with DSGD, DSGD++, and CCD++. On Netflix and Hugewiki, NOMAD converged much faster than its competitors; not only was its initial convergence faster, but it also discovered a better quality solution. On Yahoo! Music, all four methods performed similarly 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 nonzero ratings per item, respectively, Yahoo! Music has only 404 ratings per item. Therefore, when Yahoo! Music was divided equally across 32 machines, each item only had an average of 10 ratings per machine. Hence, the cost of sending and receiving item parameter vector $\boldsymbol{h}_j$ for one item j across the network was higher than for executing SGD updates on the ratings of the item locally stored within the machine, $\overline{\Omega}_j^{(q)}$. Consequently, the cost of network communication dominated the algorithms' overall execution time, meaning that there was little difference in their convergence speeds.

LATENT DIRICHLET ALLOCATION

Topic models, including the popular latent Dirichlet allocation (LDA),[11] allow us to aggregate vocabulary from a document corpus to form latent topics. Learning meaningful topic models with massive document collections that contain millions of documents and billions of tokens is challenging for two reasons. First, it requires dealing with a large number of topics (typically on the order of thousands). Second, it requires

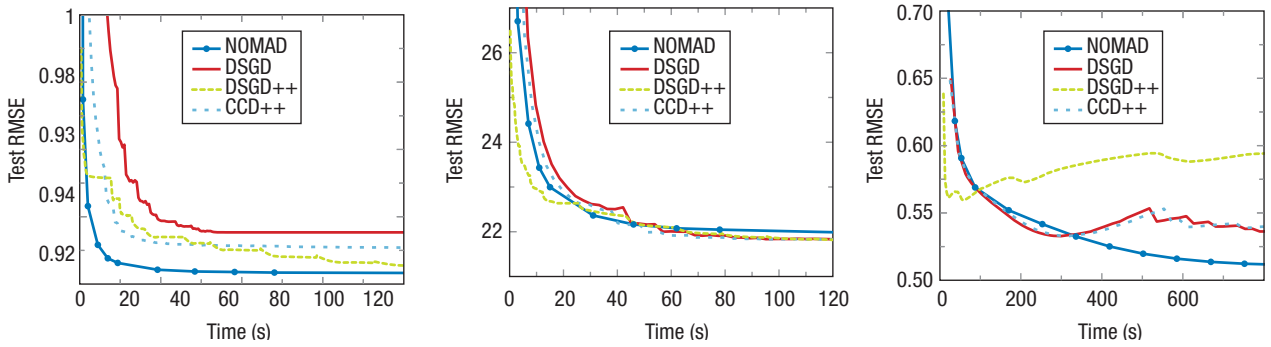

FIGURE 4. Comparison of NOMAD, DSGD, DSGD++, and CCD++ on a high-performance computing cluster with four computing threads and two communication threads per machine for three datasets: (a) Netflix (machines = 32, $\lambda = 0.05$, $k = 100$), (b) Yahoo! Music (machines = 32, $\lambda = 1.00$, $k = 100$), and (c) Hugewiki (machines = 64, $\lambda = 0.01$, $k = 100$).

a scalable, efficient way to distribute the computation across multiple machines. In this article, we focus on distributed computation and refer readers to Hsiang-Fu Yu and colleagues' work for details on handling large numbers of topics.[12]

We begin by very briefly reviewing LDA. Suppose we are given I documents, denoted as $d_1, d_2, \ldots, d_I$, and let J denote the number of words in the vocabulary. Moreover, let n_i denote the number of words in a document d_i. Let w_j denote the jth word in the vocabulary and $w_{i,j}$ denote the jth word in the ith document. Assume that the documents are generated by sampling from T topics denoted as $\varphi_1, \varphi_2, \ldots, \varphi_T$; a topic is simply a J-dimensional multinomial distribution over words. Each document includes some proportion of the topics. These proportions are latent, and we use the T-dimensional probability vector θ_i to denote the topic distribution for a document d_i. Moreover, let $z_{i,j}$ denote the latent topic from which $w_{i,j}$ was drawn. Let α and β be hyperparameters of the Dirichlet distribution. The LDA generative process can be described as follows:

1. Draw T topics $\varphi_k \sim Direchlet(\beta)$, $k = 1, \ldots, T$.
2. For each document $d_i \in \{d_1, d_2, \ldots, d_I\}$:
   Draw $\theta_i \sim Direchlet(\beta)$.
   For $j = 1, \ldots, n_i$
      Draw $z_{i,j} \sim Discrete(\theta_i)$.
      Draw $w_{i,j} \sim Discrete(\varphi_{z_{i,i}})$.

Inference

The inference task for LDA is to characterize the posterior distribution $\Pr(\varphi_i, \theta_k, z_{i,j} | w_{i,j})$. In the Bayesian setting, we want an efficient way to draw samples from this posterior distribution. Collapsed Gibbs sampling (CGS) is a popular LDA inference scheme.[13] Define

$$n_{z,i,w} := \sum_{j=1}^{n_i} I(z_{i,j} = z \text{ and } w_{i,j} = w),$$

$$n_{z,i,*} = \sum_w n_{z,i,w},\ n_{z,*,w} = \sum_i n_{z,i,w},$$

$$\text{and } n_{z,*,*} = \sum_{i,w} n_{z,i,w},$$

where I(·) is the indicator function. The update rule for CGS can be written as follows:

1. Decrease $n_{z_{i,j},i,*}$, $n_{z_{i,j},*,w_{i,j}}$, and $n_{z_{i,j},*,*}$ by 1.
2. Resample $z_{i,j}$ according to

$$\Pr(z_{i,j} \mid w_{i,j}, \alpha, \beta) \propto \frac{(n_{z_{i,j},i,*} + \alpha)(n_{z_{i,j},*,w_{i,j}} + \beta)}{n_{z_{i,j},*,*} + J \cdot \beta}. \quad (3)$$

3. Increase $n_{z_{i,j},i,*}$, $n_{z_{i,j},*,w_{i,j}}$, and $n_{z_{i,j},*,*}$ by 1.

Although we focus on CGS, there are many other LDA inference techniques such as collapsed variational Bayes, stochastic variational Bayes, or expectation maximization, all of which follow a very similar update pattern.[14] The parallel framework we describe also applies to this wider class of inference techniques.

Nomadic inference for LDA

Figure 5 illustrates the access pattern for a CGS update. For simplicity, we use $\mathbf{n}_w$ to denote the T-dimensional vector of counts corresponding to the word w, $\mathbf{n}_d$ to denote the count vector corresponding to the dth document, and $\mathbf{n}_t$ to denote the vector corresponding counts for each T topic in the entire corpus. Analogous to matrix completion, the key observation in CGS is that to perform an update for an occurrence of word w (a hyperedge in Figure 5) in the dth document, we need to access counts $z_{i,j}$, $\mathbf{n}_w$, $\mathbf{n}_d$, and $\mathbf{n}_t$. Notice that the access pattern of CGS without access to $\mathbf{n}_t$ is identical to the access pattern of matrix completion (see Figure 1). Thus, the same nomadic scheme for matrix completion can be applied to parallelize CGS for LDA: partition the data across machines, distribute the static variables $\{\mathbf{n}_d\}$, and let $\{\mathbf{n}_w\}$ be nomadic variables.

However, a crucial difference is that CGS updates need to read and update two entries of $\mathbf{n}_t$ (the summation node in Figure 5). Each element of $\mathbf{n}_t$ is a large value (it contains the total number of words from the whole corpus, which are assigned to each topic), and each update changes its value by at most one.[15] Therefore, the relative change of $\mathbf{n}_t$ in a short period of time is often negligible. This allows us to design a special nomadic scheme to keep $\mathbf{n}_t$ in sync across workers (see Figure 6). Other than the global nomadic $\mathbf{n}_t$, which travels across workers, we will

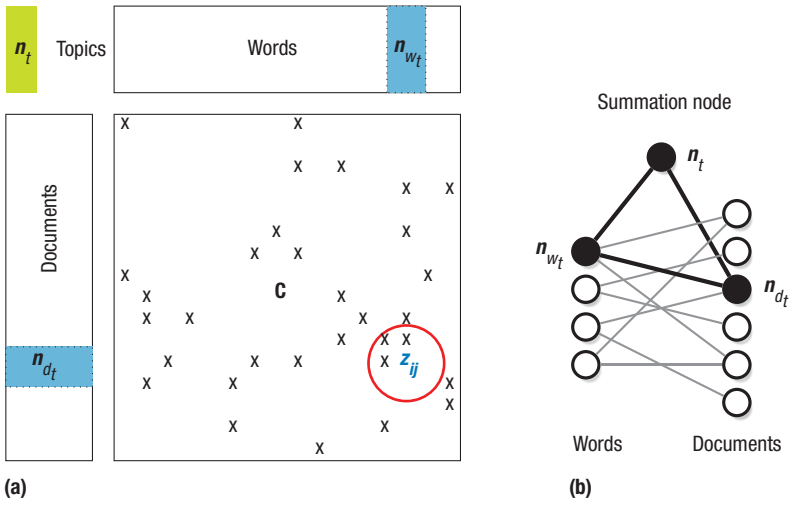

FIGURE 5. Access graph for collapsed Gibbs sampling inference for latent Dirichlet allocation. (a) Updating counts n_{w_t} and n_{d_t} requires access to n_{w_t}, n_{d_t}, n_t, and $z_{i,j}$. (b) The same access pattern represented by a hypergraph. Black indicates that the node's value is being updated, gray that the node's value is being read, and white that the nodes are being neither read nor updated.

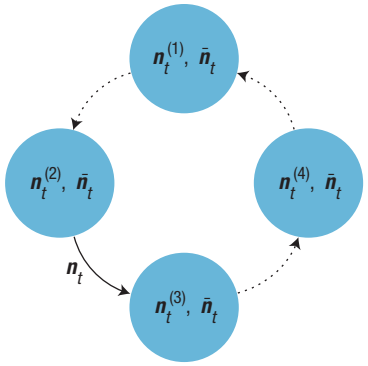

FIGURE 6. The global sum n_t is a nomadic variable. Whenever a worker receives n_t, it updates the global copy with the change in its local working copy $n_t^{(l)}$, copies the updated value to its local working copy, stores a local snapshot $\bar{n}_t$, and passes it on to the next worker. The process is illustrated here with four processors.

have two copies of n_t in each worker: $n_t^{(l)}$ and $\bar{n}_t \cdot n_t^{(l)}$ is a local working copy for n_t. The lth worker always reads and writes to $n_t^{(l)}$. On the other hand, $\bar{n}_t$ is a snapshot of n_t from the last time the global n_t visited this worker. Owing to the additivity of the count vector n_t, we can easily compute the change in $\bar{n}_t$ as $n_t^{(l)} - \bar{n}_t$. Whenever n_t arrives, the worker can perform the operations shown in Figure 6 to update the global n_t and copy its value to $n_t^{(l)}$ and $\bar{n}_t$.

Experiments

As described in Table 2, we worked with three large real-world datasets: PubMed, Amazon, and UMBC (WebBase Corpus processed by ebiquity research group at University of Maryland, Baltimore County; http://ebiquity.umbc.edu/blogger/2013/05/01/).[12] The experiments were conducted on Maverick (https://portal.tacc.utexas.edu/user-guides/maverick), a parallel platform at the Texas Advanced Computing Center (TACC). Each node contained 20 Intel Xeon E5-2680 CPUs and 256 GBytes of memory. Each job could run on at most 32 nodes (640 cores) for 12 hours maximum. We set the hyperparameters to $\alpha = 50/T$ and $\beta = 0.01$, where $T = 1{,}024$ is the number of topics.

We compared our algorithm, F+NOMAD LDA (the prefix F+ denotes the fast F+LDA sampling algorithm proposed in previous work[12]), against a state-of-the-art distributed LDA inference approach: Yahoo! LDA,[15] which is claimed to outperform other open source implementations such as AD-LDA and PLDA. Yahoo! LDA is a disk-based implementation that assumes that the latent variables associated with tokens in the documents are streamed from the disk at each iteration. In addition to running the disk-based Yahoo! LDA, denoted as Yahoo! LDA(D), we also ran our algorithm on the `tmpfs` file system that resides on RAM for the intermediate storage used by Yahoo! LDA. Thus, we eliminated the cost of disk I/O and could fairly compare Yahoo! LDA(D)'s code with our own code, which does not stream data from disk; we used Yahoo! LDA(M) to denote this version. We used the same training likelihood routine to evaluate each model's quality (see Equation 2 in Alexander Smola and Shravan Narayanamurthy's article[15]).

Multicore experiments. Both F+NOMAD LDA and Yahoo! LDA support parallel computation on a single machine with multiple cores. We conducted experiments on two datasets: PubMed and Amazon (see Figure 7). F+NOMAD LDA outperformed both the memory and disk versions of Yahoo! LDA and achieved a better quality solution in the same amount of time (see Figures 7a and 7b). Given a desired log-likelihood level, F+NOMAD LDA was approximately four times faster than Yahoo! LDA.

We then turned our attention to F+NOMAD LDA scaling as a function of the number of cores (see Figure 7c). As the number of cores increased, convergence speed became faster.

Distributed memory experiments. We compared the performance of

TABLE 2. Data statistics.

Dataset	No. of documents (I)	No. of vocabulary in the corpus (J)	No. of word tokens
PubMed	8,200,000	141,043	737,869,083
Amazon	29,907,995	1,682,527	1,499,602,431
University of Maryland, Baltimore County	40,559,164	2,881,476	1,483,145,192

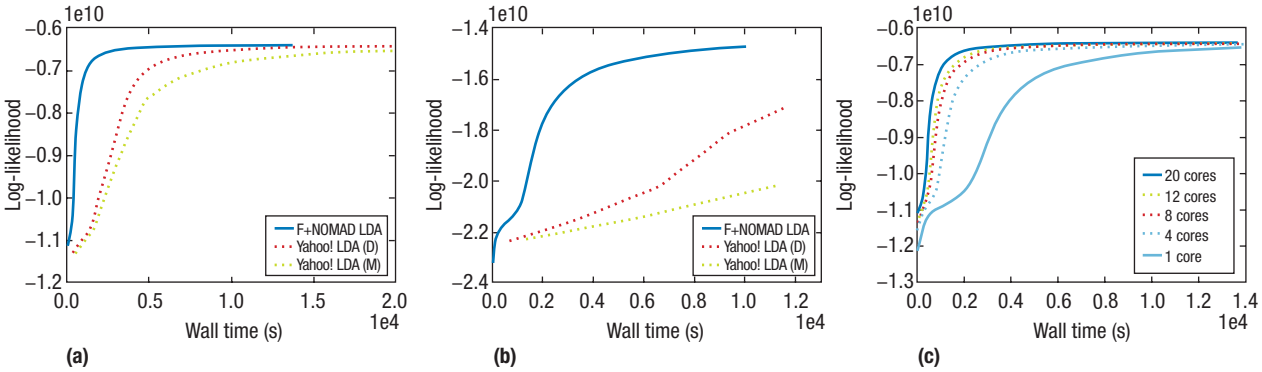

FIGURE 7. Multicore experiment comparing F+NOMAD and Yahoo! (data [D] and memory [M] versions) latent Dirichlet allocation (LDA) using 20 cores on a single machine for two datasets: (a) PubMed and (b) Amazon. (c) F+NOMAD LDA's scaling performance as a function of number of cores.

F+NOMAD LDA and Yahoo! LDA on two huge datasets, Amazon and UMBC, in a distributed memory setting. We set the number of machines to 32 and the number of cores per machine to 20. F+NOMAD LDA outperformed both the memory and disk versions of Yahoo! LDA and obtained a significantly better quality solution (in terms of log-likelihood) within the same wall-clock time (see Figure 8).

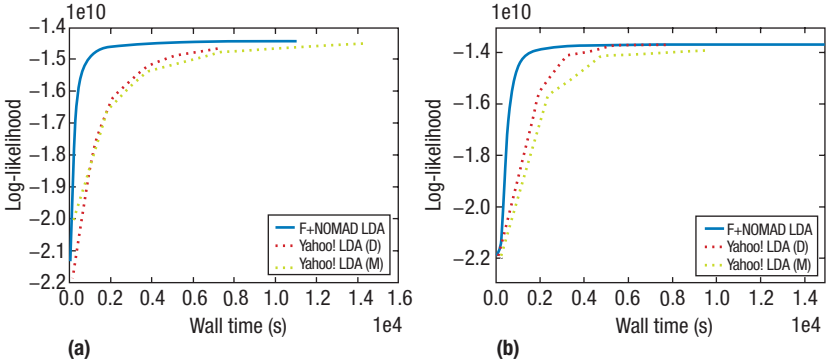

FIGURE 8. Distributed memory experiment comparing F+NOMAD LDA and Yahoo! (D) and (M) LDA on 32 machines with 20 cores per machine for two datasets: (a) Amazon and (b) UMBC.

Through our experiments, we showed that our novel NOMAD framework can tackle matrix completion and inference in LAD. We are actively working to extend NOMAD to other machine-learning problems, and have obtained encouraging results for training support vector machines and logistic regression. Despite our framework's many advantages, it is currently not fault tolerant—if one worker fails, there is no way to recover. How to design a mechanism to increase the fault tolerance of NOMAD framework is an ongoing research challenge. Another research challenge will be to find a way to minimize nomadic variables' movement based on the data distribution. We hope to collaborate with computer systems researchers to explore these interesting and active research areas.

ACKNOWLEDGMENTS

We thank the Texas Advanced Computing Center for providing infrastructure and timely support for our experiments. This research was supported by National Science Foundation grants IIS-1546452 and IIS-1546459.

REFERENCES

1. L. Bottou and O. Bousquet, "The Tradeoffs of Large Scale Learning," *Proc. Advances in Neural Information Processing Systems 20* (NIPS 07), 2007; http://leon.bottou.org/publications/pdf/nips-2007.pdf.

ABOUT THE AUTHORS

HSIANG-FU YU is a PhD candidate in the Computer Science Department at the University of Texas at Austin (UT Austin). His research interests include large-scale machine learning and data mining. Yu received an MS degree in computer science from National Taiwan University. Contact him at rofuyu@cs.utexas.edu.

CHO-JUI HSIEH is an assistant professor in the Department of Computer Science and Statistics at the University of California, Davis. His research interests include new algorithms and optimization techniques for large-scale machine-learning problems. Hsieh received a PhD in computer science from UT Austin. Contact him at chohsieh@ucdavis.edu.

HYOKUN YUN is a machine-learning scientist at Amazon.com. His research interests include multimodal representation learning, stochastic optimization, and distributed computing. Yun received a PhD in statistics from Purdue University. Contact him at yunhyoku@amazon.com.

S.V.N. VISHWANATHAN is a professor of computer science at the University of California, Santa Cruz. His research interests include machine learning, large-scale distributed optimization, and personalization. Vishwanathan received a PhD in machine learning from the Indian Institute of Science. Contact him at vishy@ucsc.edu.

INDERJIT DHILLON is the Gottesman Family Centennial Professor of Computer Science and Mathematics at UT Austin, as well as director of the Institute for Computational Engineering and Sciences Center for Big Data Analytics. His research interests include big data, machine learning, network analysis, linear algebra, and optimization. Dhillon received a PhD in computer science from the University of California, Berkeley. He is a Fellow of IEEE, SIAM, and ACM. Contact him at inderjit@cs.utexas.edu.

2. J. Dean and S. Chemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Comm. ACM 50th Anniversary Issue 1958–2008*, vol. 51, no. 1, 2008, pp. 107-113.
3. R.M. Bell and Y. Koren, "Lessons from the Netflix Prize Challenge," *ACM SIGKDD Explorations Newsletter*, vol. 9, no. 2, 2007, pp. 75–79.
4. D.P. Bertsekas and J.N. Tsitsiklis, *Parallel and Distributed Computation: Numerical Methods*, Athena Scientific, 1997.
5. Y. Low et al., "Distributed Graphlab: A Framework for Machine Learning and Data Mining in the Cloud," *J. Proc. VLDB Endowment*, vol. 5, no. 8, 2012, pp. 716–727.
6. Y. Zhuang et al., "A Fast Parallel SGD for Matrix Factorization in Shared Memory Systems," *Proc. 7th ACM Conf. Recommender Systems* (RecSys 13), 2013, pp. 249–256.
7. R. Gemulla et al., "Large-Scale Matrix Factorization with Distributed Stochastic Gradient Descent," *Proc. 17th ACM SIGKDD Int'l Conf. Knowledge Discovery and Data Mining* (KDD 11), 2011, pp. 69–77.
8. C. Teflioudi, F. Makari, and R. Gemulla, "Distributed Matrix Completion," *Proc. IEEE 12th Int'l Conf. Data Mining* (ICDM 12), 2012, pp. 655–664.
9. H.-F. Yu et al., "Scalable Coordinate Descent Approaches to Parallel Matrix Factorization for Recommender Systems," *Proc. IEEE 12th Int'l Conf. Data Mining* (ICDM 12), 2012, pp. 765–774.
10. G. Dror et al., "The Yahoo! Music Dataset and KDD-Cup'11," *J. Machine Learning Research: Conf. and Workshop Proc.*, vol. 18, 2012, pp. 3–18.
11. D. Blei, A. Ng, and M. Jordan, "Latent Dirichlet Allocation," *J. Machine Learning Research*, vol. 3, 2003, pp. 993–1022.
12. H.-F. Yu et al., "A Scalable Asynchronous Distributed Algorithm for Topic Modeling," *Proc. Int'l World Wide Web Conf.* (WWW 15), 2015, pp. 1340–1350.
13. T. Griffiths and M. Steyvers, "Finding Scientific Topics," *PNAS*, vol. 101, suppl. 1, 2004, pp. 5228–5235.
14. A. Asuncion et al., "On Smoothing and Inference for Topic Models," *Proc. 25th Conf. Uncertainty in Artificial Intelligence* (UAI 09), 2009, pp. 27–34.
15. A.J. Smola and S. Narayanamurthy, "An Architecture for Parallel Topic Models," *J. Proc. VLDB Endowment*, vol. 3, no. 1, 2010, pp. 703–710.