# Parallelizing Big Data Machine Learning Applications with Model Rotation

Bingjing ZHANG, Bo PENG and Judy QIU
*School of Informatics and Computing, Indiana University, Bloomington, IN, USA*

**Abstract.** This paper proposes model rotation as a general approach to parallelize big data machine learning applications. To solve the big model problem in parallelization, we distribute the model parameters to inter-node workers and rotate different model parts in a ring topology. The advantage of model rotation comes from maximizing the effect of parallel model updates for algorithm convergence while minimizing the overhead of communication. We formulate a solution using computation models, programming interfaces, and system implementations as design principles and derive a machine learning framework with three algorithms built on top of it: Latent Dirichlet Allocation using Collapsed Gibbs Sampling, Matrix Factorization using Stochastic Gradient Descent and Cyclic Coordinate Descent. The performance results on an Intel Haswell cluster with max 60 nodes show that our solution achieves faster model convergence speed and higher scalability than previous work by others.

**Keywords.** machine learning, big data, big model, parallelization, model rotation

## 1. Introduction

Machine learning applications such as Latent Dirichlet Allocation (LDA) and Matrix Factorization (MF) have been successfully applied on big data within various domains. For example, Tencent uses LDA for search engines and online advertising [1] while Facebook[1] uses MF to recommend items to more than one billion people. With tens of billions of data entries and billions of model parameters, these applications can help data scientists to gain a better understanding of big data.

However, the growth of data size and model size makes it hard to deploy these machine learning applications in a way that scales to our needs. A huge amount of effort has been invested in parallelization of these applications, and yet much of the literature deals with a framework-based approach using tools such as MPI[2], (Iterative) MapReduce [2][3], Graph/BSP [4], and Parameter Server [5]. At this stage, it remains unclear what is the best approach to parallelization.

To bridge the gap, we propose a systematic approach, "model rotation", which improves the efficiency of model convergence speed and provides high scalability. This solution involves fine-grained synchronization mechanisms in handling parallel model

[1] `https://code.facebook.com/posts/861999383875667/recommending-items-to-more-than-a-billion-people`
[2] `http://www.mpi-forum.org`

updates. In the context of this paper, "model rotation" is a generalized parallel computation model that performs parallel model parameter computation via rotation of different model parts in a ring topology. We provide programming interfaces for model rotation in Harp MapCollective framework [6]. Further optimizations include pipelining to reduce the communication overhead and dynamically controlling the time point of the model rotation.

We investigate three core algorithms: Collapsed Gibbs Sampling (CGS) for LDA [7], Stochastic Gradient Descent (SGD) for MF [8] and Cyclic Coordinate Descent (CCD) for MF [9]. The experiments run with different datasets on up to a maximum of 60 nodes and 1200 threads in an Intel Haswell cluster. We compare our CGS, SGD and CCD implementations under model rotation-based framework with state-of-the-art implementations (Petuum for CGS, NOMAD for SGD and CCD++ for CCD) running side-by-side on the same cluster. The results show that our solution exceeds their model convergence speeds.

The rest of this paper is organized as follows. Section 2 explores the features of model update in machine learning algorithms and the advantages of model rotation. Section 3 describes the programming interface and the implementation. The experiment results are shown in Section 4, while Section 5 describes the related work. Section 6 gives the conclusion.

## 2. Algorithms and Computation Models

In this section, we observe three important features of model updates in machine learning algorithms (CGS for LDA, SGD and CCD for MF) whose model update formula is not of the summation form introduced in [10]. For the algorithm parallelization, by discussing the efficiency of different computation models, we highlight the benefits of using model rotation.

### *2.1. Algorithms*

Many machine learning algorithms are built on iterative computation, which can be formulated as

$$A^t = F(D, A^{t-1}) \tag{1}$$

In this equation, $D$ is the observed dataset, $A$ is model parameters to learn, and $F$ is the model update function. The algorithm keeps updating model $A$ until convergence (by reaching a stop criterion or a fixed number of iterations).

We use CGS for LDA, and SGD or CCD for MF as examples to show the nature of model update dependency. The sequential pseudo code of the three algorithms is listed in Table 1. LDA is a generative modeling technique using latent topics. CGS algorithm learns the model parameters by going through the tokens in a collection of documents $D$ and computing the topic assignment $Z_{ij}$ on each token $X_{ij} = w$ by sampling from a multinomial distribution of a conditional probability of $Z_{ij}$:

$$p\left(Z_{ij} = k \mid Z^{\neg ij}, X_{ij}, \alpha, \beta\right) \propto \frac{N_{wk}^{\neg ij} + \beta}{\sum_w N_{wk}^{\neg ij} + V\beta} \left(M_{kj}^{\neg ij} + \alpha\right) \tag{2}$$

**Table 1.** Sequential Pseudo Code of Machine Learning Algorithm Examples

CGS Algorithm for LDA

**Input:** training data $X$, the number of topics $K$, hyperparameters $\alpha, \beta$
**Output:** topic assignment matrix $Z$, topic-document matrix $M$, word-topic matrix $N$

Initialize $M, N$ to zeros
**for** document $j \in [1, D]$ **do**
  **for** token position $i$ in document $j$ **do**
    $Z_{ij} = k \sim Mult(\frac{1}{K})$
    $M_{kj}$ += 1; $N_{wk}$ += 1
  **end for**
**end for**
**repeat**
  **for** document $j \in [1, D]$ **do**
    **for** token position $i$ in document $j$ **do**
      $M_{kj}$ −= 1; $N_{wk}$ −= 1
      $Z_{ij} = k' \sim p(Z_{ij} = k | rest)$ {sample a new topic by Eq. (2) using SparseLDA [11]}
      $M_{k'j}$ += 1; $N_{wk'}$ += 1
    **end for**
  **end for**
**until** convergence

SGD Algorithm for MF

**Input:** training matrix $V$, the number of features $K$, regularization parameter $\lambda$, learning rate $\varepsilon$
**Output:** row related model matrix $W$ and column related model matrix $H$

Initialize $W, H$ to $UniformReal(0, 1/\sqrt{K})$
**repeat**
  **for random** $V_{ij} \in V$ **do**
    {$L_2$ regularization}
    $error = W_{i*} H_{*j} - V_{ij}$
    $W_{i*} = W_{i*} - \varepsilon(error \cdot H_{*j}^{\mathsf{T}} + \lambda W_{i*})$
    $H_{*j} = H_{*j} - \varepsilon(error \cdot W_{i*}^{\mathsf{T}} + \lambda H_{*j})$
  **end for**
**until** convergence

CCD Algorithm for MF

**Input:** training matrix $V$, the number of features $K$, regularization parameter $\lambda$
**Output:** row related model matrix $W$ and column related model matrix $H$

Initialize $W, H$ to $UniformReal(0, 1/\sqrt{K})$
Initialize residual matrix $R$ to $V - WH$
**repeat**
  **for** $V_{*j} \in V$ **do**
    **for** $k = 1$ **to** $K$ **do**
      $s^* = \frac{\sum_{i \in V_{*j}} (R_{ij} + H_{kj} W_{ik}) W_{ik}}{\sum_{i \in V_{*j}} (\lambda + W_{ik}^2)}$
      $R_{ij} = R_{ij} - (s^* - H_{kj}) W_{ik}$
      $H_{kj} = s^*$
    **end for**
  **end for**
  **for** $V_{i*} \in V$ **do**
    **for** $k = 1$ **to** $K$ **do**
      $z^* = \frac{\sum_{j \in V_{i*}} (R_{ij} + W_{ik} H_{kj}) H_{kj}}{\sum_{j \in V_{i*}} (\lambda + H_{kj}^2)}$
      $R_{ij} = R_{ij} - (z^* - W_{ik}) H_{kj}$
      $W_{ik} = z^*$
    **end for**
  **end for**
**until** convergence

In this equation, superscript $\neg ij$ means that the corresponding token is excluded. $V$ is the vocabulary size. $N_{wk}$ is the current token count of the word $w$ assigned to topic $k$ in $K$ topics, and $M_{kj}$ is the current token count of the topic $k$ assigned in the document $j$. $\alpha$ and $\beta$ are hyperparameters. The model includes $Z$, $N$, $M$ and $\sum_w N_{wk}$. When $X_{ij} = w$ is computed, some elements in the related row $N_{w*}$ and column $M_{*j}$ are updated. Therefore dependencies exist among different tokens when accessing or updating $N$ and $M$ model matrices.

MF decomposes a $m \times n$ matrix $V$ (dataset) to a $m \times K$ matrix $W$ (model) and a $K \times n$

matrix $H$ (model). SGD algorithm learns the model parameters by optimizing the object loss function composed by a squared error and a regularizer ($L_2$ regularization is used here). When an element $V_{ij}$ is computed, the related row vector $W_{i*}$ and column vector $H_{*j}$ are updated. The gradient calculation of the next random element $V_{i'j'}$ depends on the previous updates in $W_{i'*}$ and $H_{*j'}$. CCD also solves MF application. But unlike SGD, the model update order firstly goes through all the rows in $W$ and then the columns in $H$, or all the columns in $H$ first and then rows in $W$. The model update inside each row of $W$ or column of $H$ goes through feature by feature.

Parallelization of these iterative algorithms can be done by utilizing either the parallelism inside different components of model update function $F$ or the parallelism among multiple invocations of $F$. For the first category, the difficulty of parallelization lies in the computation dependencies inside $F$, which are either between the data and the model or among the model parameters. If $F$ is in a "summation form", such algorithms can be easily parallelized through the first category [10]. However, in large-scale learning applications, the algorithms picking random examples in model update perform asymptotically better than the algorithms with the summation form [12]. In this paper, we focus on this type of algorithm with the second category of parallelism where the difficulty of parallelization lies in the dependencies between iterative updates of a model parameter. Thus when the dataset is partitioned to $P$ parts, model updates in this kind of algorithm only use one part of data entries $D_p$ as

$$A^t = F(D_p, A^{t-1}) \tag{3}$$

Obtaining the exact $A^{t-1}$ is not feasible in parallelization. It is challenging to parallelize different invocations of $F$. However, these algorithms have certain features which can maintain the algorithm correctness and improve the parallel performance.

**I.** ***The algorithms can converge even when the consistency of a model is not guaranteed to some extent.*** Algorithms can work on model $A$ with an older version $i$ when $i$ is within bounds [13], as shown in

$$A^t = F(D_p, A^{t-i}) \tag{4}$$

By using a different version of $A$, Feature I breaks the dependency across iterations.

**II.** ***The update order of the model parameters is exchangeable.*** Although different update orders can lead to different convergence rates, they normally don't make the algorithm diverge. If $F$ only accesses and updates one of the disjointed parts of the model parameters ($A_{p'}$), there is a chance of finding an arrangement on the order of model updates that allows independent model parts to be processed in parallel while keeping the dependencies.

$$A^t_{p'} = F(D_p, A^{t-1}_{p'}) \tag{5}$$

In CGS, we provide one update order by document, but other orders such as updating by words are also correct since CGS allows order exchange. Two tokens of different words in different documents can be trained in parallel since there is no update conflict in model matrices $N$ and $M$.

**III.** ***The model parameters for update can be randomly selected.*** CGS by its nature supports random scanning on model parameters [14]. In SGD, a random selection on

model parameters for updating is done through randomly selecting examples from the dataset. In CCD, the selection of rows or columns is also random.

### *2.2. Computation Models*

A detailed description of computation models can be found in previous work [15]. Our summary of related work helps to define computation models based on two properties. One is whether the computation model uses synchronous or asynchronous algorithms for parallelization. Another looks at whether the model parameters used in computation are the latest or stale. Both the synchronization strategies and the model consistency can impact model convergence speed.

Computation models using stale model parameters can be easily applied based on Feature I of model updates. However, it does come with certain performance issues such as less effective model updates. When a synchronized algorithm is applied, the computation model can be implemented via "allreduce". By doing so, the routing can be optimized while each worker retains a full copy of the model. For big models, this causes high memory usage and can result in failure to scale for applications. Another method is allowing each worker to only fetch the model parameters related to the local training data. This saves memory but offers less opportunity for routing optimization. When an asynchronous algorithm is used, the computation model reduces the synchronization overhead. However, since each worker directly communicates large numbers of model updates, routing among the workers cannot be optimized. Without synchronization barriers, this computation model does not aim for complete synchronization of model copies on all the workers. As such, the model convergence speed is affected by the real network speed. To solve this problem, Q. Ho et al. combine asynchronous algorithms and synchronized algorithms into one computation model to guarantee the model convergence and improve the speed [13].

The computation model with the latest model parameters provides more effective model updates. It is commonly performed through "model rotation". This method shows many advantages. Unlike computation models using stale model parameters, there is no additional local copy for model parameters fetched during the synchronization, meaning the memory usage is low. Plus in a distributed environment, the communication only happens between two neighboring workers so that the routing can be easily optimized.

CGS, SGD and CCD can all be implemented by the above-mentioned computation models because of Features I and II of model updates [16][17][18][19][20][21][22]. Although model rotation performs better [17][18][19][22], it may result in high synchronization overhead in these algorithms due to the dataset being skewed and the unbalanced workload of each worker [23][24]. Therefore the completion of an iteration has to wait for the slowest worker. If the straggler acts up, the cost of synchronization becomes even higher. In the implementation of our model rotation framework, we take these problems into consideration and minimize the overhead.

## 3. Programming Interface and Implementation

We introduce our model rotation solution based on the Harp MapCollective framework. Harp [6] works on top of Hadoop MapReduce system and provides collective commu-

nication operations to synchronize Map tasks. The following demonstrates how model rotation is applied to three algorithms: CGS for LDA, SGD and CCD for MF.

### *3.1. Data and Model*

The structure of the training data can be generalized as a tensor. For example, the dataset in CGS is a document-word matrix. In SGD, the dataset is explicitly expressed as a matrix. When it is applied to recommendation systems, each row of the matrix represents a user and each column an item; thus every element represents the rating of a user to an item. In these matrix structured training data, a row has a row-related model parameter vector as does a column. For quickly visiting data entries and related model parameters, indices are built on the row IDs and column IDs. Based on the model settings, the number of elements per vector can be very large. As a result, both row-related and column-related model structures might be large matrices.

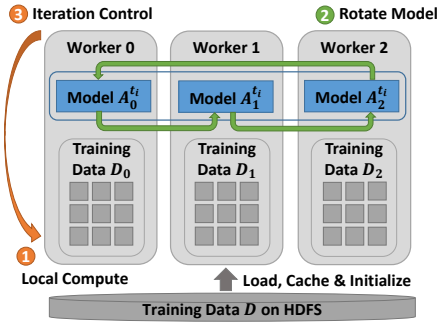
In CGS and SGD, the model update function allows for the data to be split by rows or columns so that one model (with regards to the matching row or column of training data) is cached with the data, leaving the other to be rotated. For CCD, the model update function requires both the row-related model matrix $W$ and the column-related model matrix $H$ to be rotated. We abstract the model for rotation as a distributed data structure organized as partitions and indexed with partition IDs. A partition can be expressed as an array if the vector is dense, or as a key-value pair if sparse. In CGS, each partition holds a word's topic counts. In SGD, each partition holds a column's related model parameter vector. In CCD, each partition holds a feature dimension's parameters of all the rows or columns.

### *3.2. Operation API*

We express model rotation as a collective communication. The operation takes the model part owned by a process and performs the rotation. By default, the operation sends the model partitions to the next neighbor and receives the model partitions from the last neighbor in a predefined ring topology of workers. An advanced option is that we can dynamically define the ring topology before performing the model rotation. Thus programming model rotation requires just one API. For local computation inside each worker, we simply program it through an interface of "schedule-update". A scheduler employs a user-defined function to maintain a dynamic order of model parameter updates and avoid the update conflict. Since the local computation only needs to process the model obtained during the rotation without considering the parallel model updates from other workers, the code of a parallel machine learning algorithm can be modularized as a process of performing computation and rotating model partitions (see Figure 1).

### *3.3. Pipelining and Dynamic Rotation Control*

The model rotation operation is implemented as a non-blocking call so that the efficiency of model rotation can be optimized through pipelining. We divide the distributed model parameters on all the workers into two sets $A_{*a}$ and $A_{*b}$ (see Figure 2a). The pipelined model rotation is conducted in the following way: all the workers first compute Model $A_{*a}$ with related local data. Then they start to shift $A_{*a}$, and at the same time they compute Model $A_{*b}$. When the computation on Model $A_{*b}$ is completed, it starts to shift.




**Input:** $P$ workers, data $D$, model $A^0$, the number of iterations $T$
**Output:** $A^t$

1. **parallel for** worker $p \in [1,P]$ **do**
2. **for** $t = 1$ **to** $T$ **do**
3. **for** $i = 1$ **to** $P$ **do**
4. $A_{p'}^{t_i} = F(D_p, A_{p'}^{t_{i-1}})$
5. rotate $A_{p'}^{t_i}$
6. **end for**
7. **end for**
8. **end parallel for**

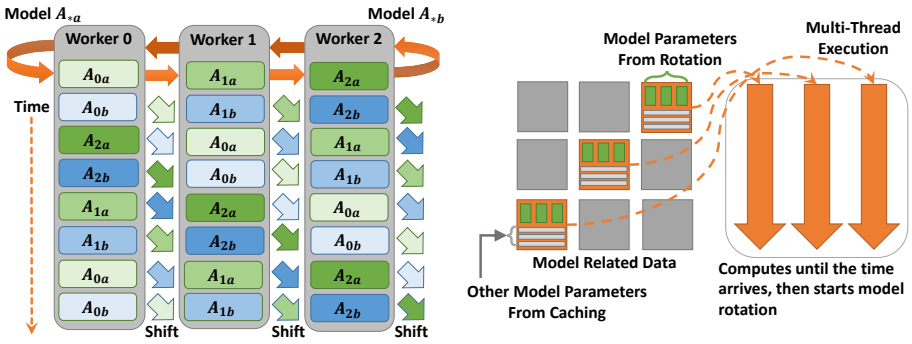
**Figure 1.** Execution Flow and Pseudo Code of Model Rotation

All workers wait for the completion of corresponding model rotations and then begin computing model updates again. Therefore the communication is overlapped with the computation. In this pipelining mechanism, each time a chunk of model parameters is computed and communicated. In experiments, communicating model parameters in large batches is more efficient than flooding the network with small messages [23].

Utilizing Features II and III in model update, we also allow dynamic control on the invocation of model rotation based on the time spent on computation (see Figure 2b). In CGS for LDA and SGD for MF, assuming each worker caches rows of data and row-related model parameters and obtains column-related model parameters through rotation, it then selects related training data to perform local computation. We split the data and model into small blocks which the scheduler dynamically selects for model update while avoiding the model update conflicts on the same row or column. Once a block is processed by a thread, it reports the status back to the scheduler. Then the scheduler searches another free block and dispatches to an idle thread. We set a timer to oversee the training progress. When the designated time arrives, the scheduler stops dispatching new blocks, and the execution ends. Because the computation is load balanced with the same length of time, the synchronization overhead is reduced. In the iterations of model rotations, the time length is further adjusted based on the amount of data items processed. But in CCD, the algorithm dependency does not allow us to dynamically control the model rotation. The reason is that in each iteration of CCD, each model parameter is only updated once. Using dynamic control results in incomplete model updates in each iteration, which reduces the model convergence speed.

### *3.4. Algorithm Parallelization*

- **CGS:** When parallelizing CGS with model rotation, the training data is split by document. As a result, the document-topic model matrix is partitioned by documents while the word-topic model matrix is rotated among processes. Documents are partitioned into blocks on each worker. Inside each block, inverted index is used to group tokens by word. The word-topic matrix owned by the worker is also split into blocks. By selecting a document block ID and a word block ID, we can train a small set of data and update the related model parameters. Because the computation time per token changes as the model converges [23], the amount of tokens which can be trained during a time period grows larger. As a result, we




(a) Pipelining (b) Dynamic Rotation Control

**Figure 2.** Model Rotation Mechanisms

keep an upper bound and a lower bound for the amount of tokens trained between two invocations of model rotation.

- **SGD:** Both $W$ and $H$ are model matrices. Assuming $n < m$, then $V$ is regrouped by rows, $W$ is partitioned with $V$, and $H$ is the model for rotation. The ring topology for rotation is randomized per iteration for accelerating the model convergence. For pipelining and load balancing, we estimate the ratio of computation and communication cost, and determine the time point to perform model rotation.
- **CCD:** Both $W$ and $H$ are model matrices. Because model update on a row of $W$ needs all the related data items on the same row and model update on a column of $H$ needs all the related data items on the column, the training data is duplicated so that one is regrouped by rows and another is regrouped by columns among workers. However, the model update on each feature dimension is still independent. Then we split $W$ and $H$ by features and distribute them across workers. Both $W$ and $H$ are rotated for parallel updating of each feature vector in $W$ and $H$.

## 4. Experiments

### *4.1. Training Dataset and Model Parameter Settings*

Four datasets are used in the experiments. For LDA and MF applications, each has one small dataset and one big dataset. The datasets and related training parameters are presented in Table 2. Two big datasets are generated from the same source "ClueWeb09"[3], while two small datasets are generated from Wikipedia[4]. With different types of datasets, we show the effectiveness of our model rotation solution on the model convergence speed. In LDA, the model convergence speed is evaluated with respect to the model likelihood, which is a value calculated from the trained word-topic model matrix. In MF, the model convergence speed is evaluated by the value of Root Mean Square Error (RMSE) calculated on the test dataset, which is from the original matrix $V$ but separated from the

[3] `http://lemurproject.org/clueweb09.php`
[4] `https://www.wikipedia.org`

**Table 2.** Training Datasets

| **LDA Dataset** | **Documents** | **Words** | **Tokens** | **CGS Parameters** | |
|---|---|---|---|---|---|
| clueweb1 | 76163963 | 999933 | 29911407874 | $K = 10000$ $\alpha = 0.01$ $\beta = 0.01$ | |
| enwiki | 3775554 | 1000000 | 1107903672 | | |
| **MF Dataset** | **Rows** | **Columns** | **Non-Zero Elements** | **SGD Parameters** | **CCD Parameters** |
| clueweb2 | 76163963 | 999933 | 15997649665 | $K = 2000$<br>$\lambda = 0.01$<br>$\varepsilon = 0.001$ | $K = 120$<br>$\lambda = 0.1$ |
| hugewiki | 50082603 | 39780 | 3101135701 | $K = 1000$<br>$\lambda = 0.01$<br>$\varepsilon = 0.004$ | |

**Table 3.** Test Plan

| **Dataset** | **Node** | |
|---|---|---|
| | **Xeon E5 2699 v3 (each uses 30 Threads)** | **Xeon E5 2670 v3 (each uses 20 Threads)** |
| clueweb1 | Harp CGS vs. Petuum (on 30) | Harp CGS vs. Petuum (on 30/45/60) |
| enwiki | Harp CGS vs. Petuum (on 10) | |
| clueweb2 | Harp SGD vs. NOMAD (on 30)<br>Harp CCD vs. CCD++ (on 30) | Harp SGD vs. NOMAD (on 30/45/60)<br>Harp CCD vs. CCD++ (on 60) |
| hugewiki | Harp SGD vs. NOMAD (on 10)<br>Harp CCD vs. CCD++ (on 10) | |

training dataset. In addition, we examine the scalability of implementations through the tests on the big datasets.

### *4.2. Comparison of Implementations*

We compare six implementations in the experiments. First is CGS implemented with Harp compared to CGS implemented with Petuum LDA[5]. Then we compare SGD implemented with Harp to SGD implemented with NOMAD[6]. Later we compare CCD implemented with Harp to CCD++[7]. The three compared implementations are the fastest open-source implementations we surveyed from the related work. Note that Petuum LDA, NOMAD and CCD++ are all implemented in C++11 while Harp CGS and Harp SGD are implemented in Java 8, so it is quite a challenge for us to exceed them in performance. Petuum LDA uses Open MPI[8] for multi-processes, POSIX threads for multi-threading, and ZeroMQ[9] for communication. Although Petuum uses model rotation at an inter-node level, intra-node multi-threading is deployed with asynchronous algorithms and stale model parameters. NOMAD uses MPICH2[10] for inter-node processes and In-

[5] `https://github.com/petuum/strads/tree/master/apps/lda_release`
[6] `http://bikestra.github.io`
[7] `http://www.cs.utexas.edu/~rofuyu/libpmf`
[8] `https://www.open-mpi.org`
[9] `http://zeromq.org`
[10] `http://www.mpich.org`

tel Thread Building Blocks[11] for multi-threading. In NOMAD, MPI_Send/MPI_Recv are communication operations, but the destination of model shifting is randomly selected without following a ring topology. CCD++ also uses MPICH2 for inter-node processes with collective communication operations and OpenMP[12] for multi-threading.

### *4.3. Parallel Execution Environment*

Experiments are conducted on a 128-node Intel Haswell cluster at Indiana University. Among them, 32 nodes each have two 18-core Xeon E5-2699 v3 processors (36 cores in total), and 96 nodes each have two 12-core Xeon E5-2670 v3 processors (24 cores in total). All the nodes have 128 GB memory and are connected by QDR InfiniBand. For our tests, JVM memory is set to "-Xmx120000m -Xms120000m", and IPoIB is used for communication. The implementations are tested on two types of machines separately (see Table 3). We focus on the convergence speed and scaling on the test configurations in which computation time is still the main part of the whole execution. For the small datasets, we use 10 Xeon E5-2699 v3 nodes each with 30 threads, while the big datasets are undertaken by 30 Xeon E5-2699 v3 nodes each with 30 threads and 60 Xeon E5-2670 nodes each with 20 threads to compare the model convergence speed among different implementations. We further examine the scalability with 30, 45, and 60 Xeon E5-2670 v3 nodes each with 20 threads.

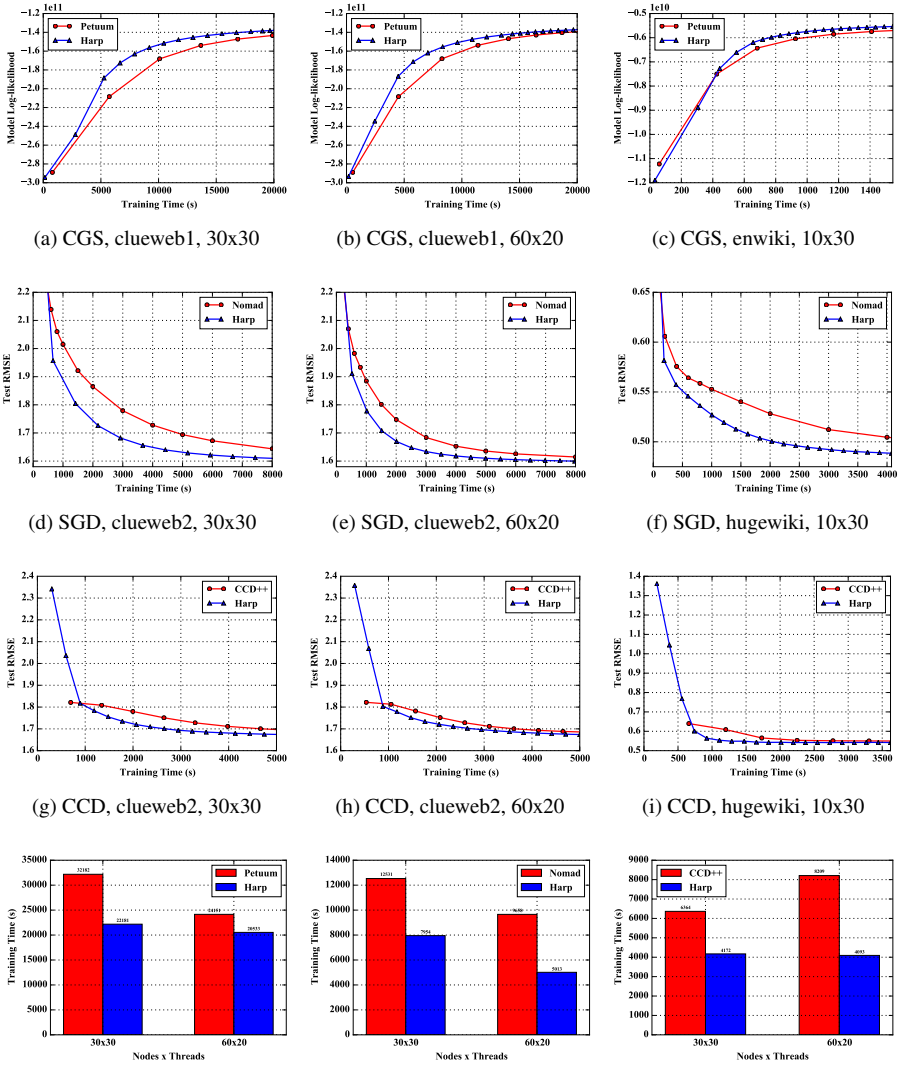
### *4.4. Model Convergence Speed*

In CGS, through examination of the model likelihood achieved by the training time, the results on two different datasets all show that Harp consistently outperforms Petuum. We test Harp CGS and Petuum on "clueweb1" with $30 \times 30$ and $60 \times 20$ two configurations (see Figure 3a and Figure 3b). Both results show that Harp CGS converges faster than Petuum. We also test "enwiki" on $10 \times 30$ and the result is the same (see Figure 3c). Concerning the convergence speed on the same dataset with different configurations, we observe that the fewer the number of cores used and the more computation per core, the faster Harp runs compared to Petuum. When the scale goes up, the difference in the convergence speed reduces. With $30 \times 30$ Xeon E5-2699 v3 nodes, Harp is 45% faster than Petuum while with $60 \times 20$ Xeon E5-2670 v3 nodes, Harp is 18% faster than Petuum when the model likelihood converges to $-1.37 \times 10^{11}$ (see Figure 3j).

In SGD, Harp SGD also converges faster than NOMAD. On "clueweb2" (see Figure 3d and Figure 3e), with $30 \times 30$ Xeon E5-2699 v3 nodes, Harp is 58% faster, and with $60 \times 20$ Xeon E5-2670 v3 nodes, Harp is 93% faster when the test RMSE value converges to 1.61 (see Figure 3k). The difference in the convergence speed increases because the random shifting mechanism in NOMAD becomes unstable when the scale goes up. We also test the small dataset "hugewiki" on $10 \times 30$ Xeon E5-2699 v3 nodes, and the result remains that Harp SGD is faster than NOMAD (see Figure 3f).

In CCD, we again test the model convergence speed on "clueweb2" and "hugewiki" datasets (see Figure 3g, Figure 3h and Figure 3i). The results show that Harp CCD also has comparable performance with CCD++. Note that CCD++ uses a different model update order, so that the convergence rate based on the same number of model update

[11] `https://www.threadingbuildingblocks.org`
[12] `http://openmp.org/wp`




(a) CGS, clueweb1, 30x30 (b) CGS, clueweb1, 60x20 (c) CGS, enwiki, 10x30

(d) SGD, clueweb2, 30x30 (e) SGD, clueweb2, 60x20 (f) SGD, hugewiki, 10x30

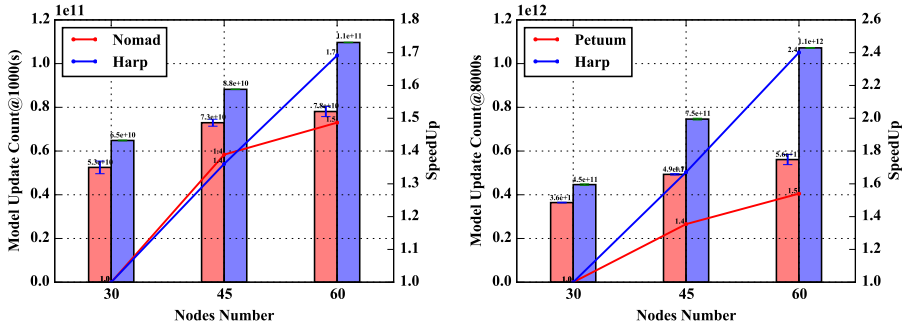
(g) CCD, clueweb2, 30x30 (h) CCD, clueweb2, 60x20 (i) CCD, hugewiki, 10x30

(j) CGS, clueweb1 (k) SGD, clueweb2 (l) CCD, clueweb2

**Figure 3.** Model Convergence Speed Comparison on CGS, SGD and CCD

count is different with Harp CCD. However, the tests on "clueweb2" reveal that with $30 \times 30$ Xeon E5-2670 v3 nodes, Harp CCD is 53% faster than CCD++ and with $60 \times 20$ Xeon E5-2699 v3 nodes Harp CCD is 101% faster than CCD++ when the test RMSE converges to 1.68 (see Figure 3l).

All the convergence charts demonstrate that Harp implementation is faster than the related implementations on the same algorithm. Though the percentage of speedup may vary as we change the convergence point and the scale, the overall difference is still significant in the whole convergence process.




(a) SGD, clueweb2 (b) CGS, clueweb1

**Figure 4.** Scaling of SGD and CGS on 30, 45, 60 Xeon E5-2699 v3 Nodes

### *4.5. Scalability*

We further evaluate the scalability of Harp implementations by focusing on the comparison between different implementations of CGS and SGD. We do not include CCD because regardless of whether we use Harp or CCD++, both $W$ and $H$ model matrices are required to be synchronized rather than only rotating $H$ in SGD. There is more communication time than computation time per iteration, hence the iteration time does not decrease much when the scale goes up, resulting in low scalability.

Since the convergence speed is not linear as the execution time passes, we use throughput on the number of training data items processed to evaluate the scalability of CGS and SGD implementations. In SGD, the time complexity of processing a training data item is $\mathcal{O}(K)$ for all the elements in $V$; as such this metric is suitable for evaluating the performance of SGD implementations on different scales. Figure 4a shows that the throughput of Harp at 1000s achieves 1.7× speedup when scaling from 30 to 60 nodes. Meanwhile, NOMAD only achieves 1.5× speedup. The throughput of NOMAD on three scales are all lower than Harp. The situation on CGS is a little different. Since the time complexity of sampling each token is $\mathcal{O}(\sum_k \mathbb{1}(N_{wk} \neq 0) + \sum_k \mathbb{1}(N_{kj} \neq 0))$, the time complexity decreases during the convergence. As a result, the throughput on the number of tokens processed grows higher and higher as the execution time passes. When the dynamic rotation control is applied, Harp can sample more tokens as the execution progresses, which results in super linear speedup. Figure 4b shows the throughput of Harp and Petuum at 8000s. Harp has higher throughput than Petuum on the three scales.

## 5. Related Work

Much initial work on machine learning algorithms deploys one computation model and a single programming interface. Mahout[13] [10], Spark Machine Learning Library[14], and Graph-based tools such as PowerGraph[15] [4] are three such examples. All these imple-

[13] `http://mahout.apache.org`
[14] `https://spark.apache.org/docs/latest/mllib-guide.html`
[15] `https://github.com/turi-code/PowerGraph/tree/master/toolkits`

mentations are based on synchronized algorithms. Meanwhile, Parameter Server solutions [5][25][26] use asynchronous algorithms in which a programming interface allows each worker to "push" or "pull" model parameters for local computation. As mentioned in Section II, these solutions are not efficient for solving LDA and MF applications because they use computation models with stale model parameters which do not converge as fast as solutions using model rotation [15].

Model rotation has been applied before in machine learning. In LDA, F. Yan et al. implement CGS on a GPU [27]. In MF, DSGD++ [22] and NOMAD [28] use model rotation in SGD for MF in a distributed environment while LIBMF [24] applies it to SGD on a single node through dynamic scheduling. Another work, Petuum STRADS [17][18][19], supplies a general parallelism solution called "model parallelism" through "schedule-update-aggregate" interfaces. This framework implements CGS for LDA using model rotation but not CCD for MF[16]. Instead it uses "allgather" operation to collect model matrices $W$ and $H$ without using model rotation. Thus Petuum CCD cannot be applied to big model applications due to the memory constraint. The interfaces of Petuum STRADS operate at the model parameter level but not in a collective way, resulting in communication inefficiency. Despite these shortcomings, Petuum LDA and NOMAD are still among the fastest implementations we know among open-source implementations of the two algorithms.

CCD++ uses a parallelization method on CCD different from either Petuum CCD or our model rotation implementation. CCD++ allows parallelization on updating different elements in a single feature vector of $W$ and $H$. In such a way, only one feature vector in $W$ and one feature vector in $H$ are "allgathered". Thus there is no memory constraint in CCD++ compared with Petuum CCD.

## 6. Conclusion

To solve big model problems in machine learning applications such as LDA and MF, this paper focuses on three algorithms, CGS, SGD and CCD, and gives a full solution using model rotation, which includes computation model innovation, programming interface design, and implementation improvements. For the algorithms without the "summation form", we identify three important features in the model update mechanism and conclude that model rotation is more efficient compared to other computation models. We design the model rotation API with MapCollective programming interface which is more convenient than parameter-level APIs of other implementations. Finally, we use pipelining and dynamic rotation control to improve the efficiency of model rotation in CGS and SGD. With these steps, we achieve higher scalability and faster model convergence speed compared with related work. In the future, we can apply our model rotation solution to other large-scale learning applications with big models. Future research will also investigate providing templates for performance to guide developers to parallelize different machine learning applications based on data, algorithm, and hardware.

[16] `https://github.com/petuum/strads/tree/master/apps`

## Acknowledgment

We gratefully acknowledge support from Intel Parallel Computing Center (IPCC) Grant, NSF 1443054 CIF21 DIBBs 1443054 Grant, and NSF OCI 1149432 CAREER Grant. We appreciate the system support offered by FutureSystems.

## References

[1] Y. Wang *et al.*, "Peacock: Learning Long-Tail Topic Features for Industrial Applications," *ACM TIST*, 2015.
[2] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *CACM*, 2008.
[3] J. Ekanayake *et al.*, "Twister: A Runtime for Iterative MapReduce," in *HPDC*, 2010.
[4] J. E. Gonzalez *et al.*, "Powergraph: Distributed Graph-Parallel Computation on Natural Graphs," in *OSDI*, 2012.
[5] M. Li *et al.*, "Scaling Distributed Machine Learning with the Parameter Server," in *OSDI*, 2014.
[6] B. Zhang, Y. Ruan, and J. Qiu, "Harp: Collective Communication on Hadoop," in *IC2E*, 2015.
[7] T. L. Griffiths and M. Steyvers, "Finding Scientific Topics," *PNAS*, 2004.
[8] Y. Koren *et al.*, "Matrix Factorization Techniques for Recommender Systems," *Computer*, 2009.
[9] H.-F. Yu *et al.*, "Scalable Coordinate Descent Approaches to Parallel Matrix Factorization for Recommender Systems," in *ICDM*, 2012.
[10] C.-T. Chu *et al.*, "Map-Reduce for Machine Learning on Multicore," in *NIPS*, 2007.
[11] L. Yao, D. Mimno, and A. McCallum, "Efficient Methods for Topic Model Inference on Streaming Document Collections," in *KDD*, 2009.
[12] L. Bottou, "Large-Scale Machine Learning with Stochastic Gradient Descent," in *COMPSTAT*, 2010.
[13] Q. Ho *et al.*, "More Effective Distributed ML via a Stale Synchronous Parallel Parameter Server," in *NIPS*, 2013.
[14] R. Levine and G. Casella, "Optimizing Random Scan Gibbs Samplers," *JMVA*, 2006.
[15] B. Zhang, B. Peng, and J. Qiu, "Model-Centric Computation Abstractions in Machine Learning Applications," in *BeyondMR*, 2016.
[16] D. Newman *et al.*, "Distributed Algorithms for Topic Models," *JMLR*, 2009.
[17] S. Lee *et al.*, "On Model Parallelization and Scheduling Strategies for Distributed Machine Learning," in *NIPS*, 2014.
[18] E. P. Xing *et al.*, "Petuum: A New Platform for Distributed Machine Learning on Big Data," *IEEE Transactions on Big Data*, 2015.
[19] J. K. Kim *et al.*, "STRADS: A Distributed Framework for Scheduled Model Parallel Machine Learning," in *EuroSys*, 2016.
[20] B. Recht *et al.*, "HOGWILD!: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent," in *NIPS*, 2011.
[21] R. Gemulla *et al.*, "Large-Scale Matrix Factorization with Distributed Stochastic Gradient Descent," in *SIGKDD*, 2011.
[22] C. Teflioudi, F. Makari, and R. Gemulla, "Distributed Matrix Completion," in *ICDM*, 2012.
[23] B. Zhang, B. Peng, and J. Qiu, "High Performance LDA through Collective Model Communication Optimization," in *ICCS*, 2016.
[24] Y. Zhuang *et al.*, "A Fast Parallel SGD for Matrix Factorization in Shared Memory Systems," in *RecSys*, 2013.
[25] A. Smola and S. Narayanamurthy, "An Architecture for Parallel Topic Models," *VLDB*, 2010.
[26] A. Ahmed *et al.*, "Scalable Inference in Latent Variable Models," in *WSDM*, 2012.
[27] F. Yan, N. Xu, and Y. Qi, "Parallel Inference for Latent Dirichlet Allocation on Graphics Processing Units," in *NIPS*, 2009.
[28] H. Yun *et al.*, "NOMAD: Non-locking, stOchastic Multi-machine algorithm for Asynchronous and Decentralized matrix completion," *VLDB*, 2014.