Parallelizing Big Data Machine Learning Applications with Dynamic Model Rotation

Bingjing Zhang, Bo Peng, Judy Qiu
School of Informatics and Computing
Indiana University
Bloomington, IN, USA
Email: {zhangbj, pengb, xqiu}@indiana.edu

***Abstract*—This paper proposes dynamic model rotation as a novel approach to parallelize big data machine learning applications. We rotate different model parts in a ring topology among inter-node workers and dynamically schedule intra-node parallel model updates and fine-grained load balance with time control. The advantages of dynamic model rotation come from maximizing the effectiveness of model updates for algorithm convergence while minimizing the overhead of communication for scaling. We formulate a solution using computation model, programming interface and implementation as design principles and apply it to two representative applications: Latent Dirichlet Allocation using Collapsed Gibbs Sampling algorithm and Matrix Factorization using Stochastic Gradient Descent algorithm. The performance results on an Intel Haswell cluster with a total of 1800 threads show that our solution achieves faster model convergence speed and more reliable scalability than previous work by ourselves and others.**

***Keywords*-machine learning; big data; big model; dynamic model rotation;**

I. 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, "dynamic model rotation", which improves the efficiency of model convergence speed and provides reliable scalability. This solution involves fine-grained synchronization mechanisms in handling model updates. In the context of this paper, "model rotation" is a parallel computation model that guarantees the latest model updates via rotation of different model parts in a static ring topology. We expand "model rotation" to "dynamic model rotation" which dynamically schedules parallel model updates with fine-grained load balance control. We design a programming interface using Harp's MapCollective [6] communication API. Further optimizations include reducing the communication overhead through pipelining and time control to coordinate when the model rotates.

We investigate two core algorithms: Collapsed Gibbs Sampling (CGS) for LDA and Stochastic Gradient Descent (SGD) for Matrix Factorization (MF). Doing so enables us to reveal three important features of model updates: the ability to use stale parameters; an exchangeable update order; and randomly selected update order. The experiments run with different big datasets on a total of 1800 threads in an Intel Haswell cluster. We compare our CGS and SGD implementations under dynamic model rotation with state-of-the-art model rotation implementations (Petuum for CGS and NOMAD for SGD) running side-by-side on the same cluster, and our solution matches or exceeds their model convergence speed.

The rest of this paper is organized as follows. Section 2 discusses the design guideline of our dynamic model rotation solution. Section 3 explores the features of model update in machine learning algorithms and the advantages of our solution. Section 4 describes the programming interface and the implementation. The experiment results are shown in Section 5, while Section 6 describes the related work. Section 7 gives the conclusion.

II. Design Guidelines of Dynamic Model Rotation

Our dynamic model rotation solution is designed with regards to the following four questions:

a. *What kind of algorithm should be used for the big data machine learning application?*

In this paper, we focus on two applications: LDA in topic modeling and MF for recommendation systems. LDA can

[1] https://code.facebook.com/posts/861999383875667/recommending-items-to-more-than-a-billion-people/

[2] http://www.mpi-forum.org/

be solved by CGS algorithm [7] or Collapsed Variational Bayesian (CVB) [8] while MF can be solved by SGD algorithm [9] or Cyclic Coordinate Descent (CCD) [10]. Related literature shows that CGS scales better than CVB [1] and SGD outperforms CCD [11] for parallelization. For this reason, we use CGS for LDA and SGD for MF (see Section III).

b. *Which computation model is suitable for the algorithm?*

We define "computation model" as the pattern of parallel algorithm execution steps and synchronization strategies. By comparing the efficiency of computation models on a moderate cluster, both computation and communication time prove to be important factors for the overall execution time of big data machine learning applications. This paper shows that the proposed dynamic model rotation has advantages compared with other solutions (see Section III).

c. *How is the programming interface designed for the computation model?*

Programming interfaces are utilized by parallelization tools to express the parallel algorithm under a computation model. We implement dynamic model rotation using the MapCollective programming interface because it is easy to maintain and provides high performance (see Section IV).

d. *How is the programming interface implemented?*

A programming interface can be applied in different mechanisms, two of which are pipeline and time control for dynamic model rotation (see Section IV).

III. Algorithms and Computation Models

In this section, we observe three important features of model updates in machine learning algorithms, including CGS for LDA and SGD for MF, whose model update formula is not of the summation form introduced in [12]. For the algorithm parallelization, by discussing the efficiency of different computation models, we highlight the benefits of using dynamic model rotation.

A. Algorithms

Many machine learning algorithms are built on iterative computation. In general, iterative algorithms can be formulated as

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

Here 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 fixed number of iterations).

We use CGS for LDA and SGD for MF as examples to show the nature of model update dependency.

1) CGS for LDA: 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} (see Fig. 1):

Input: training data X, the number of topics K, hyperparameters α, β
Output: topic assignment matrix Z, topic-document matrix M, word-topic matrix N

```
Initialize M, N to zeros
for document j ∈ [1, D] do
  for token position i in document j do
    Z_ij = k ~ Mult(1/K)
    M_kj += 1; N_wk += 1
  end for
end for
repeat
  for document j ∈ [1, D] do
    for token position i in document j do
      M_kj −= 1; N_wk −= 1
      Z_ij = k' ~ p(Z_ij = k|rest) {sample a new topic
      according to Eq. 2 using SparseLDA [13]}
      M_k'j += 1; N_wk' += 1
    end for
  end for
until convergence
```

Figure 1. CGS Algorithm for LDA

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

Here 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. α and β 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.

2) SGD for MF: 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 square error and a regularizer (see Fig. 2). 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 $H_{*j'}$.

Parallelization of the 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 par-

Input: training matrix V, the number of features K, regularization parameter λ, learning rate ϵ
Output: model matrix W and H
Initialize W, H to *UniformReal*$(0, 1/\sqrt{K})$
repeat
for random $V_{ij} \in V$ **do**
{use L_2 regularization}
$error = W_{i*}H_{*j} - V_{ij}$
$W_{i*} = W_{i*} - \epsilon(error \cdot H_{*j}^{\intercal} + \lambda W_{i*})$
$H_{*j} = H_{*j} - \epsilon(error \cdot W_{i*}^{\intercal} + \lambda H_{*j})$
end for
until convergence

Figure 2. SGD Algorithm for MF

allelization lies in the computation dependencies inside F, which are between the data entries and the model parameter or among the model parameters. If F is in a "summation form", such algorithms can be easily parallelized through the first category [12].

However in large-scale learning applications, the algorithms picking random examples in model update perform asymptotically better than the algorithms with the summation form [14]. 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 across iterations of model parameter updates. Thus when the dataset is partitioned to N 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 some features which can maintain the algorithm correctness and improve the parallel performance.

I. ***The algorithms can converge even when the consistency of a model are not guaranteed to some extent.***

Algorithms can work on model A with an older version i, as

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

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 rate, they normally don't make the algorithm diverge. If F only accesses and updates one of the disjointed parts of the whole model parameters ($A_{p'}$), there is a chance to find an arrangement on the order of model updates that allows independent model parts be processed in parallel while keeping the dependencies.

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

Fig. 1 provides one update order by document, but other orders 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 in its nature supports random scanning on model parameters [15]. In SGD, a random selection on model parameters for updating is done through randomly selecting examples from the dataset.

B. Computation Models

A detailed description of computation models can be found in previous work [16]. 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 the model convergence speed.

Computation models using the 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 "allgather" and "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 usage 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 [17].

The computation model with the latest model parameters provides more effective model updates. It is commonly performed through "model rotation". The model rotation-based computation model 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.

Both CGS and SGD can be implemented by the above-mentioned computation models because of Feature I and II of model updates [18][19][20][21][22]. Although model rotation performs better [19][22], it may result in high synchronization overhead in these algorithms due to the dataset skew 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.

As a result, we propose dynamic model rotation utilizing features II and III and control the order of model updates without affecting the algorithm convergence:

- At the start of each iteration, the ring topology rank is randomly generated, which formulates the order of inter-node model rotation.
- Within an iteration, intra-node multi-threading is dynamically scheduled to avoid conflicts of concurrent model updates.
- Fine grained load balance is achieved by controlling computation workload and synchronization point.

IV. Programming Interface and Implementation

Here we introduce dynamic model rotation using the Harp MapCollective programming interface. Harp [6] works as a plug-in on top of Hadoop MapReduce system and provides collective communication operations to synchronize Map tasks. In addition, we show how these mechanisms are applied to CGS for LDA and SGD for MF.

A. Data and Model

The structure of the 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 is 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 matching row or column of training data) is cached with the data, leaving the other to be rotated. We abstract the model for rotation as a distributed data structure organized as partitions and indexed with partition IDs. Each partition holds a row/column's related model parameter vector. A partition can be expressed as an array if the vector is dense, or as a key-value pair if sparse.

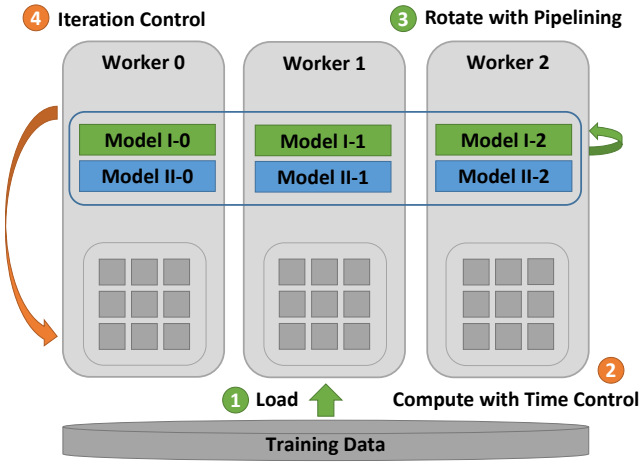


Figure 3. Parallel Execution Steps of Dynamic Model Rotation

B. 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. For local computation inside each Map task, we simply express the model rotation in multi-threading through a programming interface of "schedule-execute". A scheduler employs a user-defined function to maintain a dynamic order of model parameter updates.

Under the MapCollective interfaces, programming model rotation requires just one API. Since the local computation only needs to process the model obtained during the rotation without considering the parallel model updates from other tasks, the code of a parallel machine learning algorithm can be modularized as a process of obtaining model partitions, performing computation and updating.

C. Pipelining

The model rotation operation is defined as a non-blocking call so that the efficiency of model rotation can be optimized through pipelining. Taking the matrix structured data as an example, we divide the model for rotation into two sets and evenly distribute them across all the workers. We call these two model splits Model I and Model II (see Fig. 3). The pipelined model rotation is conducted in the following way (see Fig. 4): all the workers compute Model I with its related data. Then they start to shift Model I and at the same time they compute Model II. When the computation on Model II is completed, it starts to shift. All workers wait for the completion of corresponding model rotations, and then begin computing model updates again. Therefore

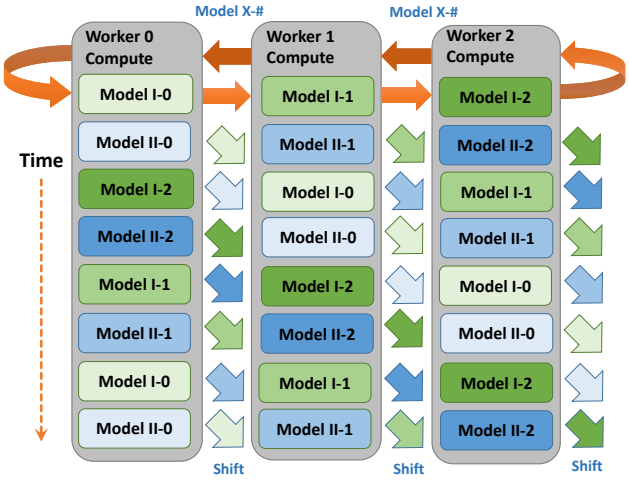


Figure 4. Pipelined Model Rotation Operation

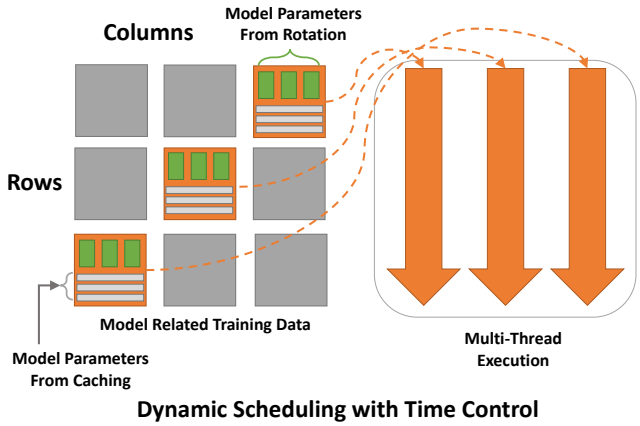


Figure 5. Local Computation with Time Control

```
Initialize time_limit
for i = 0 to number_of_iterations do
  for j = 0 to number_of_workers do
    for k = 0 to number_of_model_splits do
      model_split = rotation.get()
      compute(data, model_split, time_limit)
      rotation.submit(model_split)
    end for
  end for
  Update time_limit
end for
```

Figure 6. A General Structure of Pseudo Code Performing Dynamic Model Rotation on Each Map Task in the Harp MapCollective Programming Interface

the communication is overlapped with the computation. This pipelining mechanism works at the distributed data structure level where each time a chunk of model parameters are computed and communicated. In experiments, communicating model parameters in large batches is more efficient than flooding the network with small messages [23].

D. Dynamic Scheduling and Time Control

Timers are used to control the point at which we perform model rotation. To describe the mechanism of time control, we again use matrix structured training data. Assuming each worker caches rows of data and row-related model parameters (see Fig. 5) and obtains column-related model parameters through rotation, it then selects related training data to perform local computation. Through the "schedule-execute" programming interface, we split the data and the model into small blocks which the scheduler randomly selects for model update while avoiding the model update conflicts on the same row or column. (see Fig. 5). 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 (see Fig. 3). Because the computation is load balanced with the same length of time, the synchronization overhead is reduced.

As such the semantics of "iteration" change when using time control. All model partitions are still rotated for one round per iteration, but only a partial training dataset is processed in one iteration. Based on Feature III, this mechanism does not affect the level of model convergence ultimately achieved in CGS and SGD. Since these two algorithms spend more time on computation than communication, we further tune the time setting to keep an appropriate amount of data entries being trained thereby making sure computation and communication can fully overlap. Our experiments show that the time control mechanism enhances the efficiency of model rotation and improves the model convergence speed.

E. Algorithm Parallelization

For algorithm parallelization with dynamic model rotation, Fig. 6 shows a common structure of code in a Map task using MapCollective interfaces. We further provide the implementation details of CGS and SGD with dynamic model rotation.

1) CGS: The model in CGS has four data structures. The first two are document-topic matrix and word-topic matrix. Because training data is split by document, the document-topic matrix is partitioned by documents while the word-topic matrix is rotated among processes. Next is the topic assignment on each token, which is stored with the training data. Lastly we have total number of tokens on each topic. Since this is a small array with length equal to the number of topics, where all the elements are required in the local computation, we simply synchronize it with

Table I
TRAINING DATASETS

CGS Dataset	Number of Documents	Vocabulary	Number of Tokens	Number of Topics	Model for Rotation
clueweb	76.2M	1.0M	29.9B	10K	Word-Topic Matrix (Initial Size 17.1GB)
SGD Dataset	Number of Rows	Number of Columns	Number of Cells	Number of Features	Model for Rotation
clueweb	76.2M	1.0M	16.0B	2K	Column Model Matrix (16.0GB)

"allreduce" operation. 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 keep an upper bound and a lower bound for the tokens trained in the time period. When the number of tokens trained crosses the bounds, we increase or decrease the time setting.

2) 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. Since the computation time used to train each data entry does not change across iterations, we only tune the time limit to a specific value. We estimate the ratio of computation and communication cost during the first iteration, then set the time limit to a value which meets the minimum requirement for overlapping.

V. EXPERIMENTS

Here we test the efficiency of dynamic model rotation and compare our solution with other designs.

A. Training Dataset and Model Parameter Settings

Two datasets are used in the test: one is for CGS and another for SGD (see Table. I). Both are generated from a subset of the "ClueWeb09" dataset[3]. In CGS the model for rotation theoretically has 10 billion model parameters, but the real number after initialization is approximately 2 billion [23]. Similarly, SGD rotates about 2 billion model parameters.

B. Comparison of Implementations

We compare four implementations in the experiments. First is CGS implemented with Harp (with time control vs. without time control) compared to CGS implemented with Petuum LDA[4]. Then we compare SGD implemented with Harp (with time control vs. without time control) to SGD implemented with NOMAD[5]. The concept of model rotation is applied in all these implementations. We analyze the differences in the design, programming interfaces and implementations and show how they influence the model convergence speed. Note that Petuum LDA and NOMAD are both implemented in C++11 while Harp CGS and Harp SGD are implemented in Java 8. Petuum LDA uses Open MPI for multi-processes and POSIX threads for multi-threading and ZeroMQ 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 for inter-node processes and Intel Thread Building Blocks 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.

C. Parallel Execution Environment

Our experiments are conducted on a 128 node Intel Haswell cluster (Juliet) at Indiana University. Among them, 32 nodes each have two 18-core Xeon E5-2699 v3 processors (72-thread total) and 96 nodes each have two 12-core Xeon E5-2670 v3 processors (48-thread total). All the nodes have 128 GB memory and are connected by QDR InfiniBand. For our tests, JVM memory is set to "-Xmx120000m -Xms120000m -Xss4m -Xmn30000m" and IPoIB is used for communication.

The implementations are tested on three configurations. One is 30 Xeon E5-2699 nodes each with 60 threads (30x60). Another is 60 Xeon E5-2670 nodes each with 30 threads (60x30). The third is a heterogeneous environment which uses 30 Xeon E5-2699 nodes and 60 Xeon E5-2670 nodes to form a cluster of 90 nodes each with 20 threads (90x20). All three configurations have the same parallelism of 1800 threads in total. The same dataset and model settings are preserved across configurations, so that different implementations of model rotation can be compared.

D. Model Convergence Speed

To measure the model convergence speed, we provide details of experiment settings here. In CGS, the hyper-parameters α and β are both fixed to 0.01. The model convergence speed is evaluated with respect to model likelihood which is a value calculated from the trained word-topic model matrix. In SGD, λ is fixed to 0.01 and ϵ to

[3] http://lemurproject.org/clueweb09.php/
[4] https://github.com/petuum/strads/tree/master/apps/lda_release
[5] http://bikestra.github.io/

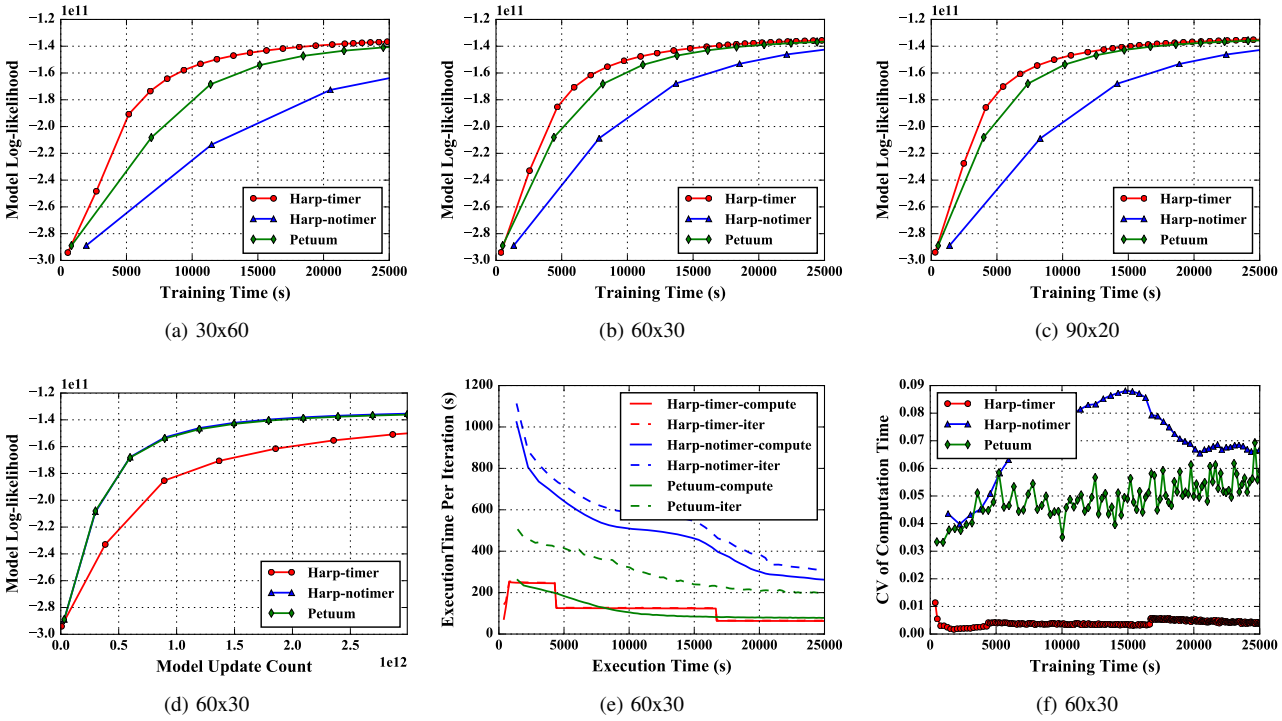


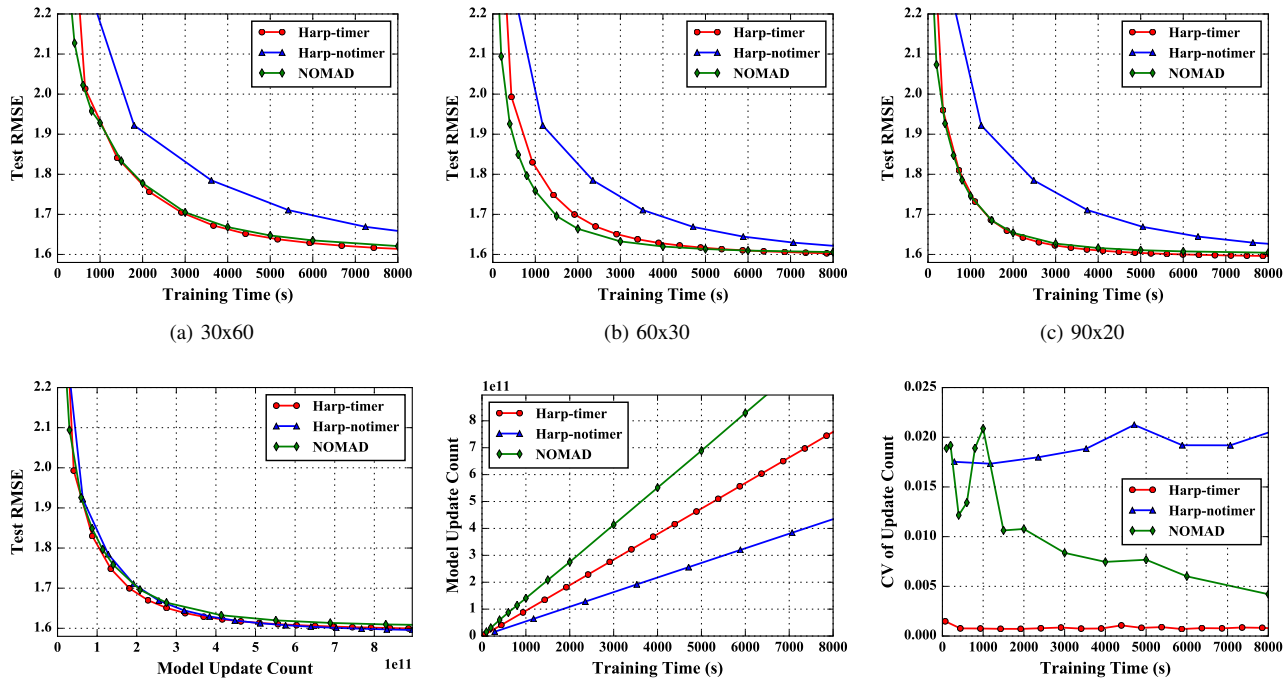
Figure 7. Performance Results on CGS (a) Model Likelihood vs. Training Time on 30x60 (b) Model Likelihood vs. Training Time on 60x30 (c) Model Likelihood vs. Training Time on 90x20 (d) Model Likelihood vs. Model Update Counts on 60x30 (e) The Iteration Execution Time and the Average Computation Time per Iteration on 60x30 (f) The Coefficient of Variation (CV) of All the Workers' Iteration Computation Time on 60x30

0.001. The model convergence speed is evaluated by the value of Root Mean Square Error (RMSE) calculated on a test dataset of 300 million data entries, which is from the matrix V separated from the training dataset.

For time control of both algorithms implemented with Harp, we set the computation time for the first iteration to 1000ms on 30x60, 500ms on 60x30 and 333ms on 90x20 so that the total computation time for each worker is 60s. We enable time tuning so that the timer settings can be adjusted for later iterations.

The performance results of CGS are presented in Fig. 7. Through examining the model likelihood achieved by the training time, the results on three configurations show that Harp implementation with time control consistently outperforms Petuum and Harp with no time control in terms of model convergence speed (see Fig. 7a, 7b, 7c). Petuum is the second fastest while Harp with no time control is the slowest. Though our previous work has shown that using collective communication operation for model rotation can result in faster communication speed than Petuum [23], when the computation load per node becomes high, the slow computation speed in Harp heavily affects the model convergence speed. The performance of all three implementations does not drop when the number of nodes increases, but improves with less threads per node. For example, the results of 60x30 configuration (refer to Fig. 7b) show that when the model likelihood achieves -1.38×10^{11} and the difference of values between successive iterations drops to 5.00×10^{8}, Harp with time control has a $1.65\times$ speedup over Harp with no time control and is also 15% faster than Petuum. When the model likelihood achieves -1.36×10^{11} and the difference drops to 2.00×10^{8}, Harp with time control has a $1.55\times$ speedup over Harp with no time control and is 14% faster than Petuum.

To understand why Harp with time control performs better than Petuum, it is sufficient to analyze the results of 60x30 configuration as similar trends exist on 30x60 and 90x20. Fig. 7d shows the model likelihood achieved by different implementations based on the number of model update counts. By using the same computation model, Harp achieves the same model likelihood as Petuum. When time control is applied, Harp generates a different model update order across iterations, which may affect the model convergence rate by the number of model update counts. Here we observe that the model convergence rate achieved with time control is not as high as the other two. However, when time control is used, the performance of pipelined model rotation is improved. Fig. 7e shows that the average computation time per iteration in Harp with time control is nearly identical with the execution time per iteration, which implies communication is completely hidden by computation. Therefore communication in dynamic model rotation



(a) 30x60 (b) 60x30 (c) 90x20

(d) 60x30 (e) 60x30 (f) 60x30

Figure 8. Performance Results on SGD (a) RMSE vs. Training Time on 30x60 (b) RMSE vs. Training Time on 60x30 (c) RMSE vs. Training Time on 90x20 (d) RMSE vs. Model Update Count on 60x30 (e) Model Update Count vs. Traning Time on 60x30 (f) The Coefficient of Variation (CV) of All the Workers' Iteration Computation Time on 60x30

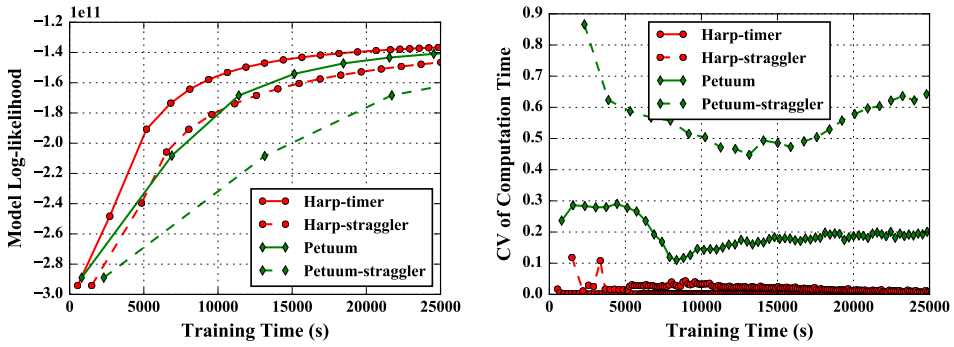
incurs no additional overhead. In contrast, both Petuum and Harp with no time control have high synchronization overhead per iteration. Although pipelining is applied in both implementations, the unbalanced computation workload on each worker makes overlapping of communication within computation difficult. Despite the fact that Petuum's computation is highly optimized, its parameter level messaging pipelining causes higher communication overhead compared to Harp with no time control. Fig. 7f illustrates the variation of computation time in all the workers' at each iteration. Harp with time control shows much lower variation, demonstrating more balanced workload than other implementations.

In SGD, Figs. 8a, 8b, and 8c show the performance results on 30x60, 60x30 and 90x20. We observe that the model convergence speed increases with the number of nodes. Harp with time control performs better than with no time control. NOMAD is fast at the beginning but its speed becomes similar to or slower than that of Harp with time control as the model converges. Let's take the results of 60x30 (refer to Figs. 8b) as an example. When the RMSE value is 1.61 and the difference between iterations drops to 5.00×10^{-4}, Harp with time control is 1% slower than NOMAD but has a 1.69× speedup over Harp with no time control. However, when the RMSE value is 1.60 and the difference drops to 1.00×10^{-4}, Harp with time control is 57% faster than NOMAD and still has a 1.47× speedup over Harp with no time control.

The reason for NOMAD's unstable model convergence speed is its randomized model parameter shifting mechanism. The results of Fig. 8d show that Harp with time control samples subsets of training data per iteration but still achieves the same RMSE value as Harp without time control. NOMAD is slightly slower because the destination of model parameter shifting is randomly selected. As a result, a model parameter may map to the same training data partitions across two successive training steps, causing less effective model update. This problem is illustrated in Fig. 8e. Although NOMAD trains more elements than Harp within the same training time, it does not converge effectively. Randomized model parameter shifting may lead to unoptimized communication in routing. Fig. 8f shows the variation of model update counts from all the workers at a particular time in training. In this figure, Harp with time control shows very little variation in the model update count on each worker.

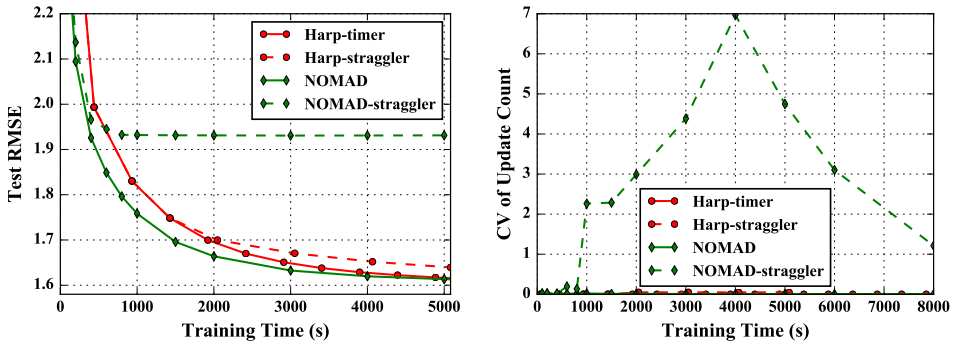
E. Reliability

The experiments show that using time control generates reliable performance results. In Fig. 9a, when one node becomes a straggler (around 10 times slower in computation) on CGS in 30x60 configuration, Harp with time control can



(a) CGS 30x60

(b) CGS 30x60



(c) SGD 60x30

(d) SGD 60x30

Figure 9. (a) CGS Model Likelihood vs. Training Time on 30x60 when a straggler exists (b) The Coefficient of Variation (CV) of Iteration Computation Time of All the Workers in CGS on 30x60 when a straggler exists (c) SGD RMSE vs. Training Time on 60x30 when a straggler exists (h) The Coefficient of Variation (CV) of Model Update Counts of All the Workers in SGD on 60x30 when a straggler exists

maintain model convergence speed while Petuum becomes much slower. Fig. 9b shows the variation of computation times of all the workers in each iteration. Even when a straggler exists, the performance of Harp with time control does not change much while Petuum becomes very unstable.

NOMAD applies dynamic load balancing through preferentially selecting a worker which has fewer items in its queue to send out, but it can only identify the straggler that is slow in communication, not computation. For example, on 60x30, if the straggler is selected for parameter destination, it holds the parameters and is slow to send them out. In this case, the model stops converging (refer to Fig. 9c) and variation of the model update counts on each worker becomes large (refer to Fig. 9d). When Harp with time control has a straggler, the algorithm still continues to converge.

VI. Related Work

Much initial work on machine learning algorithms deploys one computation model and a single programming interface. Mahout[6] [12], Spark Machine Learning Library[7] and Graph-based tools such as PowerGraph[8] [4] are three such examples. All these implementations 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 Sections II and III, these solutions are not efficient for solving LDA and MF applications. Previous research on CGS algorithm also demonstrates that implementations using computation models with stale model parameters do not converge as fast as solutions using model rotation [16].

Model rotation has been applied before in machine learning. In LDA, F. Yan et al. implement CGS on a GPU [27]. In MF, NOMAD [11] and DSGD++ [22] use model rotation for SGD in a distributed environment while LIBMF [24] applies it to SGD on a single node through dynamic scheduling. Another work, Petuum STRADS [19][28], 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[9], instead using "allgather" operation to

[6] http://mahout.apache.org/

[7] https://spark.apache.org/docs/latest/mllib-guide.html

[8] https://github.com/turi-code/PowerGraph/tree/master/toolkits

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

collect model matrices W and H without using model rotation. 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. To our best knowledge, there exist no current research efforts that use "dynamic model rotation".

VII. Conclusion

To solve big model problems in machine learning applications such as LDA and MF, this paper focuses on the CGS and SGD algorithms and gives a full solution using dynamic 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 dynamic 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 to improve the efficiency of model rotation and timers to dynamically control model rotation. With these steps, we achieve reliable scalability and faster model convergence speed compared with related work.

In the future we can apply our solution to applications other than LDA and MF. We do not claim that our dynamic model rotation solution is the silver bullet for all the machine learning applications and all the parallel execution environments. Future research will investigate providing templates for performance to guide developers to parallelize different machine learning applications based on data, algorithm and hardware.

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] D. Blei, A. Ng, and M. Jordan, "Latent Dirichlet Allocation," *JMLR*, 2003.
[9] Y. Koren *et al.*, "Matrix Factorization Techniques for Recommender Systems," *Computer*, 2009.
[10] H.-F. Yu *et al.*, "Scalable Coordinate Descent Approaches to Parallel Matrix Factorization for Recommender Systems," in *ICDM*, 2012.
[11] H. Yun *et al.*, "NOMAD: Non-locking, stOchastic Multi-machine algorithm for Asynchronous and Decentralized matrix completion," *VLDB*, 2014.
[12] C.-T. Chu *et al.*, "Map-Reduce for Machine Learning on Multicore," in *NIPS*, 2007.
[13] L. Yao, D. Mimno, and A. McCallum, "Efficient Methods for Topic Model Inference on Streaming Document Collections," in *KDD*, 2009.
[14] L. Bottou, "Large-Scale Machine Learning with Stochastic Gradient Descent," in *COMPSTAT*, 2010.
[15] R. A. Levine and G. Casella, "Optimizing Random Scan Gibbs Samplers," *JMVA*, 2006.
[16] B. Zhang, B. Peng, and J. Qiu, "Model-Centric Computation Abstractions in Machine Learning Applications," in *BeyondMR*, 2016.
[17] Q. Ho *et al.*, "More Effective Distributed ML via a Stale Synchronous Parallel Parameter Server," in *NIPS*, 2013.
[18] D. Newman *et al.*, "Distributed Algorithms for Topic Models," *JMLR*, 2009.
[19] S. Lee *et al.*, "On Model Parallelization and Scheduling Strategies for Distributed Machine Learning," in *NIPS*, 2014.
[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] J. K. Kim *et al.*, "STRADS: A Distributed Framework for Scheduled Model Parallel Machine Learning," in *EuroSys*, 2016.