Parallelizing Big Data Machine Learning Algorithms with 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 investigates a novel approach to parallelization of machine learning algorithms using model rotation as an effective parallel computation model. We identify the importance of model rotation owing to its ability to shift the latest model updates to a neighboring computation, thereby guaranteeing model consistency which is hard to achieve in other computation models. We distinguish computation model, programming interface and implementation as design principles, and give new optimizations for the model rotation approach to parameter updates using Intel multi-/many-core architectures. Our pipeline and time control optimizations further allow us to obtain fine-grained parallelism and load balance which are necessary to achieve reliable scaling results. We show our solution to be effective in two representative algorithms: Collapsed Gibbs Sampling (CGS) for Latent Dirichlet Allocation and Stochastic Gradient Descent (SGD) for matrix factorization, and give better performance than previous work by us and others. Our model rotation solution leads to a general approach for parallelizing machine learning algorithms efficiently.**

***Keywords*-machine learning; big data; big model; model rotation; pipelining; time control**

I. Introduction

Machine learning has been successfully applied in such diverse application domains as text mining [1], recommender systems [2], tag prediction of images [3], videos and music [4], speech recognition [5], and bioinformatics [6]. These big data applications may involve matrices with billions of entries for training data and model parameters. For example, recommender systems help more than a billion people search among millions of items at Amazon or Facebook[1] to find those that are most relevant to them. Innovative distributed algorithm design is necessary to allow us to scale to these constantly growing datasets.

However, the growth of data size makes it hard to employ many machine learning algorithms that scale to our needs. A huge amount of effort has been invested in parallelizing machine learning algorithms, and yet much of the literature deals with application or framework-based approach (e.g. MPI, MapReduce/Iterative MapReduce, Graph/BSP, Parameter Server, and Multicore/GPU approaches). It remains unclear what is the best approach to parallelization. To bridge the gap, we investigate a systematic approach based on "model rotation", which effectively expresses kernel computation characteristics and synchronization mechanisms for a scalable solution. We believe that understanding of kernel computations in big data analytics will foster both system and algorithm innovations, subsequently leading to tools that are useful for many people.

Our approach involves two computation concepts: iterative computation and model rotation. The former applies a computation (e.g. Map task) or function repeatedly, using output from one iteration as the input of the next iteration. Iterative computation has the advantage of solving complex problems using simple functions. The computation can stop when it converges or meets an applications approximation criteria. Well-known examples include Expectation-Maximization (EM), as well as deep learning and graph algorithms. We identify the importance of iterative computation for machine learning and data analytics and our initial framework Twister [7] caches invariant training (or input) data in memory, thereby supporting iterative algorithms effectively.

In this paper, we focus on addressing those difficult problems previously mentioned, when "Big Model" parameters exceed the computation capacity of a single node machine. This implies that common practices such as Broadcast, AllReduce, Scatter-Gather-AllGather and asynchronous Point-to-Point communication are less attractive in dealing with such obstacles. Instead we apply an alternative solution involving fine-grained synchronization mechanisms in handling data consistency vs. scaling, and observe that model rotation shifts the latest model updates to a neighboring computation, which guarantees data consistency among distributed workers. This method can scale to large-scale problems based on the observation that some machine learning algorithms can execute out of order while still converging, which we leverage through time control to sample subsets of the training data for model updates.

We design and implement a novel approach that enables model rotation pipelining within a machine via multithreading (or Intels DAAL) and across machines via our framework Harp (Hadoop plugin) [8], with the details of the parallelization process for machine learning algorithms defined in Figure 1. We apply our model rotation solution to

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

two algorithms: (i) Markov Chain Monte Carlo Type algorithm Collapse Gibbs Sampling (CGS) for Latent Dirichlet Allocation (LDA), commonly used for topic modeling in text mining; (ii) Gradient Optimization Type algorithm Stochastic Gradient Descent (SGD) for matrix factorization widely applied in recommendation systems.

Our experiments have successfully run on different datasets (29.9 billion tokens for CGS and 16 billion cells for SGD) on 30x60, 60x30 or 90x20 nodes and threads (a total of 1800 threads) on Intel Xeon/Haswell architectures. We compare Harp with state-of-the-art model rotation implementations (Petuum for CGS and NOMAD for SGD) by running them side-by-side on the same cluster. The results show that our solution can achieve similar or faster model convergence speed. Worthy of note is that the size of CGS model parameters used for our CGS/LDA experiment (76.2 million documents, 1 million vocabulary, 29.9 billion tokens, 10k topics) is the largest in the text mining literature that we are aware of to date.

The rest of this paper is organized as follows. Section 2 illustrate some machine learning algorithms. Section 3 talks about the computation model. Section 4 describes the model rotation programming interface. Section 5 explains the improvement on model rotation implementation. Section 6 gives algorithms examples of using model rotation. Section 7 shows the experiment results. Section 8 describes the related work. Section 9 gives the conclusion.

II. Model Rotation Design Principles

To parallelize big data machine learning applications, the model rotation solution is designed under the guidance of several principles. These principles include four aspects (see Fig. 1):

1) What kind of algorithm should be used for the big data machine learning application?: Before parallelizing an application with model rotation, it is important to identify which algorithm to use. A big data machine learning application can be parallelized with different algorithms. In this paper, Gibbs Sampling is selected to solve LDA application in topic modeling and Stochastic Gradient Descent is chosen to solve matrix factorization in recommendation systems (see Section III). LDA can also be implemented by other algorithms such as Collapsed Variational Bayesian and Matrix factorization can be implemented by Cyclic Coordinate Descent.

2) Which computation model is suitable for the algorithm?: We use the concept "computation model" to describe a parallel algorithm without associating any particular parallel execution environment. One algorithm can be parallelized by using different computation models. We present model rotation based computation model has advantages compared with other computation models (see Section III).

3) How is the programming interface designed for the computation model?: Programming interfaces are provided by parallelization tools to express the parallel algorithm in a computation model. Some tools are only allowed to express one computation model. We express the model rotation-based computation model using the MapCollective programming interface [1] (see Section IV).

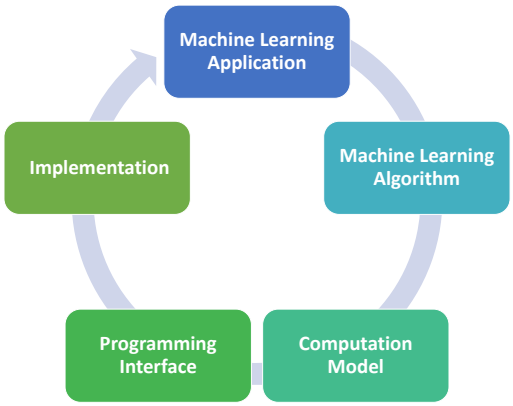


Figure 1. A solution for big data machine learning application includes decisions on algorithms, computation models, programming interfaces, and implementation.

4) How the programming interface is implemented?: A programming interface can be implemented in different mechanisms. Each tool provides a specific implementation for a programming interface. In this paper, we optimize model rotation through pipelining and time control two mechanisms (see Section IV).

III. Algorithms and Computation Models

Many machine learning algorithms are designed as an iterative computation in which model parameters are iteratively updated according to the data entries and the current model parameter values. In this section, we use Gibbs Sampling and Stochastic Gradient Descent two examples to show how the machine learning algorithms work in the sequential code. For the algorithm parallelization, we summarize four computation models. Through discussing how to map algorithms to computation models and how to map computation models to parallel execution environments, we highlight the benefits of using a model rotation-based computation model.

A. Algorithms

Many machine learning algorithms are built on iterative computation, such as EM, MCMC and Gradient Optimization which are the workhorses for the data analysis applications.

In general, iterative algorithms can be formulated as

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

Here, D is the dataset and A is model parameters, F is the model update function. The algorithm keeps updating

model A until convergence (by reaching a stop criterion or fixed number of iterations).

Taking K-means as an example. $x_i \in D$ is one data point, $C_{1..K}$ is K centroids and $Z_i \in 1..K$ is the membership of each data point. F works in two steps as $Z_i^t = \arg\min_k L(x_i, C_k^{t-1})$, and $C_k^t = \frac{1}{n}\sum_{x_i \in D} \mathbb{1}(Z_i^t = k) * x_i$. Here, L is distance function and $\mathbb{1}$ as indicator function. C and Z are two parts of the model and update sequences on them have a bipartite dependency relationship, in which Z_i can be updated in parallel when C_k^{t-1} all ready but C_k must wait for the output of $Z|Z_i = k$.

The dependencies exist inside the model structure and cross the boundary of iterations make iterative algorithms difficult to be parallelized.

According a general IID (independent and identically distributed) assumption on dataset, the data points in D are normally independent and be partitioned among all nodes. When dataset and model are large and partitioned to N nodes, model A_p on node p now is updated by

$$A_p^t = F(D, A^{t-1}) = F(D_p, A^{t-1}) \tag{2}$$

To access distributed A^{t-1} is difficult and different approaches exist that lead to different computation models.

Since iterative algorithms have an interesting feature that they converges even the consistency of model are not guaranteed to some extent, one popular approach tries to make the algorithm working in asynchronous mode by giving up the model consistency and even breaking the dependency constrains between iterations. It can works on model A with older version i, as

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

Another approach tries to find an arrangement on the order of updates to make independent parts running in parallel while keeping all the dependency constrains. When the model updates only need inputs from local D_p and local A_p^{t-1}, it should be much beneficial for efficiency since there are no network data transfer necessary. Some kind of algorithms can match this requirement by adjusting the order of their model updates.

$$A_p^t = F(D_p, A_p^{t-1}) \tag{4}$$

1) Collapsed Gibbs Sampling (CGS): This algorithm goes through all the tokens in a collection of documents and computes the topic assignment on each token $X_{ij} = w$ by sampling from a multinomial distribution of a conditional probability of Z_{ij} (see Fig. 2):

$$p\left(Z_{ij} = k \mid Z^{\neg ij}, X_{ij}, \alpha, \beta\right) \propto \frac{N_{wk}^{\neg ij} + \beta}{\sum_w N_{wk}^{\neg ij} + V\beta}\left(M_{kj}^{\neg ij} + \alpha\right) \tag{5}$$

Input: training data X, the number of topics K, hyperparamters 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. 5}
      M_k'j += 1; N_wk' += 1
    end for
  end for
until convergence
```

Figure 2. The Sequential Algorithm of CGS

Here superscript $\neg ij$ means that the corresponding token is excluded. V is the vocabulary size. N_{wk} is the token count of a word w assigned to topic k in K topics, and M_{kj} is the token count of a topic k assigned in document j. α, β are hyperparameters. The process of probability calculation can be optimized by SparseLDA [2] in which the time complexity is not based on the total number of topics K but depends on the non-zero token counts of a document j and a word w. As the model converges, the sizes of M and N shrink, resulting the decrease of the computation time complexity.

2) Stochastic Gradient Descent (SGD): In matrix factorization, we use SGD decomposes a $m \times n$ matrix V to a $m \times K$ matrix W and a $K \times n$ matrix H (see Fig. 3). When an element V_{ij} is computed, the related row W_{i*} and column H_{*j} are updated. The gradient calculation of the next element $V_{i'j'}$ depends on the previous updates in $W_{i'*}$ and $H_{*j'}$.

B. Computation Models

The detailed description of computation models can be found in previous work [3]. We 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 policies and the model consistency can impact the model convergence speed (see Table I). There are two computation models using the synchronized algorithm and the latest model parameters. One is through model

Input: training matrix V, the number of features K, hyper-paramters λ, ϵ
Output: model matrix W and H
Initialize W, H to *UniformReal*$(0, 1/\sqrt{K})$
repeat
for $V_{ij} \in V$ **do**
{use L2 norm to calculate the gradients}
$error = W_{i*}H_{*j} - V_{ij}$
$W_{i*} = W_{i*} - \epsilon(error \cdot H_{*j}^{\intercal} + \lambda W_{i*})$
$H_{*j} = H_{*j} - \epsilon(error \cdot W_{i*}^{\intercal} + \lambda H_{*j})$
end for
until convergence

Figure 3. The Sequential Algorithm of SGD

blocking, and another is through model rotation. The third computation model uses synchronized algorithm with the stale model parameters while the fourth uses a synchronous algorithm with the stale model parameters.

Table I
COMPUTATION MODELS

	The Latest Model	The Stale Model
Synchronized Algorithm	A (model blocking) B (model rotation)	C
Asynchronous Algorithm	N/A	D

Though both sequential algorithms of CGS and SGD require that the model parameter update has to depend on the current values of the model parameters, many practices show that these algorithms still work when the model parameters used in the computation are not the most updated [4][5][HogWild]. Therefore both CGS and SGD can be implemented by any of the four computation models. But some algorithms such as Expectation-Maximization type algorithms rely on synchronization barriers so that they can only be implemented by computation models with synchronized algorithms but not with asynchronous algorithms. For CGS and SGD, it is better to use computation models with the latest model parameters rather than those using the stale model parameters. The reason is that these computation models match the design of the original sequential algorithms and result in effective model parameter updates with the latest model parameters.

When mapping computation models to parallel execution environments, we focus our discussion on two types of environments. One is the distributed environment where workers are processes and communicate through network interfaces. Another is the multi-thread environment where workers are CPU threads that communicate through shared memory. Then we go though each computation model and discuss how they can be implemented in the parallel execution environments. The computation model with the synchronized algorithm and the latest model parameters can be performed through model blocking or model rotation. However model blocking is seldom used due to the high locking cost involved. In contrast, model rotation is applied in many implementations where it can occur as communication among processes on a ring topology, or synchronization between threads. Instead of using a static or predefined ordering, model rotation can be implemented through dynamic scheduling which provides a dynamic ordering to avoid conflicts between model parameter updates.

When the stale model parameters are used, the computation model with the synchronized algorithm 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, it can cause high memory usage and make applications fail to scale. Another way is to let each worker only fetch the model parameters related to the local training data. This method saves memory usage but has less opportunity for routing optimization. When changing to the asynchronous algorithm, the computation model reduces the model synchronization overhead. However, since each worker directly communicates large numbers of model parameters, the routing among the workers cannot be optimized. Without synchronization barriers, this computation model does not aim for complete model synchronization so that the model convergence speed is affected by the real network speed.

Thus the model rotation 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.

IV. PROGRAMMING INTERFACE AND IMPLEMENTATION

These advantages of model rotation-based computation model inspire us to use it in expressing the parallel machine learning algorithms based on this computation model. In this section, we introduce our programming interface for model rotation under the MapCollective programming interface. As opposed to MapReduce which uses "shuffle" operation to synchronize intermediate data from Map to Reduce tasks, MapCollective synchronizes Map tasks through collective communication operation APIs [1]. Since the computation load on each node is unbalanced, a straggler may harm the efficiency of the model rotation. We further introduce two mechanisms pipelining and time control to implement the model rotation programming interface. Finally, we use CGS and SGD as two agorithm examples under the model rotation solution.

A. Data Abstraction and Model Abstraction

Training data entries describe the relationships among entities. Thus the structure of the data can be generalized as

a multidimensional array. Assuming there are two entities, the data can be expressed as a matrix and each data entry can be expressed as a cell in the matrix. For example, the data in CGS is a document-word matrix. In SGD, the data 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 a movie, thus every element represents the rating of a user to a movie.

In this relational data model, each entity has a related model parameter vector. Back to the matrix structured training data, a row has a row-related model parameter vector as does a column. Based on the model settings, the number of elements per vector can be very large. As a result both row-related and column-related models might be large matrices. To avoid synchronizing two model matrices at the same time, the data are split by rows or by columns so that one model is stored in local with the data and only the other is required to be rotated.

We abstract the model matrix for rotation as a distributed dataset. The dataset is 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 array if the vector is dense, or as a key-value pair if sparse.

B. Operation API

We express model rotation as a collective communication operation among Map tasks. The operation takes the model portion owned by the processes and performs the rotation. In 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. The advanced option allows us to remap model partitions to workers by exchanging the model partitions on each worker. For local computation inside each Map task, We simply express the model rotation-based computation model in multi-threading through a programming interface of "schedule-execute". A scheduler employs a user-defined function to maintain the ordering of model parameter updates in model rotation dynamically and keep the workload balance on each threads.

Under the MapCollective interfaces, programming model rotation is very simple. Only one API is used to manage the model rotation. 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 getting model partitions, performing computation and updating (see Fig. 4).

C. Pipelining

Pipelining is one mechanism we use to optimize the model rotation implementation. Taking the matrix structured data as an example, and assuming that the column-related model matrix is the one for model rotation, we divide it into two

```
protected void mapCollective(KeyValReader reader) {
  Map data = createData(reader);
  Table model = createModel();
  Rotation rotation = createRotation(model);
  for i = 0 to number_of_iterations do
    for j = 0 to number_of_workers do
      Table model = rotation.get();
      compute(data, model);
      rotation.submit(model);
    end for
  end for
}
```

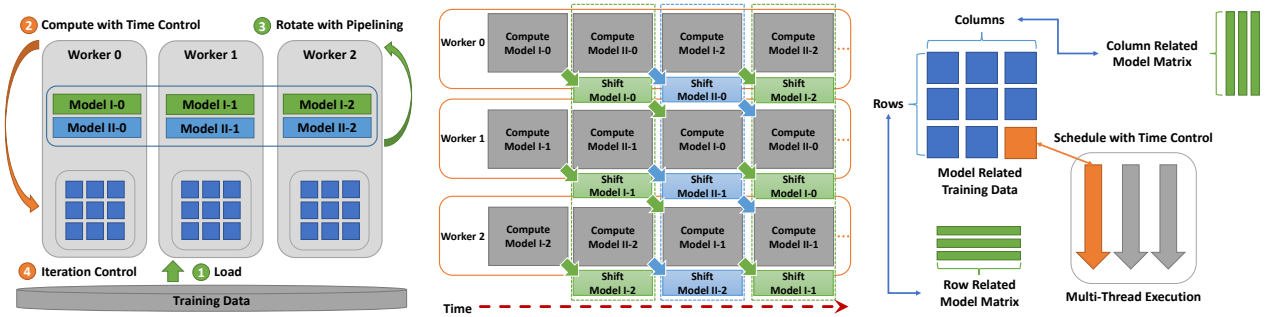
Figure 4. Model Rotation Programming Interface

sets and evenly distribute them across all the workers. We call these two model splits Model I and Model II (see Fig. 5a). The pipelined model rotation is conducted in the following way (see Fig. 5b): 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, the workers wait for the completion of Model I rotation, and then start to compute Model I again when the rotation is finished. Thus the communication is overlapped with the computation. This pipelining mechanism works at the dataset 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 [6].

D. Time Control

We introduce another mechanism called time control to improve the efficiency of the model rotation. Since a straggler may slow down the whole execution of the model rotation, we use a timer to control the computation time and balance the computation load on each worker.

To describe the mechanism of time control, we take matrix structured training data as an example again. Assuming each worker caches rows of data and row-related model parameters and owns partitions of column-related model parameters for rotation, the local computation is performed on the data related to the column model owned by the worker. Through the "schedule-execute" programming interface, we split the data and the model into small blocks and allow the scheduler to dispatch them to threads (see Fig. 5c). The scheduler randomly selects blocks but avoids computation 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 dispatches another free block to the thread available. We set a timer to oversee the training's progress. When the appointed time arrives, the scheduler stops dispatching new blocks and the execution ends (see Fig. 5a).



(a) Parallel Computation using Model Rotation (b) Model Rotation with Pipelining (c) Computation with Time Control

Figure 5. Model Rotation

Thus the semantics of the "iteration" change when using time control. All the model partitions are still rotated for one round per iteration, but only a partial training dataset is processed in one iteration. However, evidence suggests this mechanism does not affect the level of model convergence finally achieved in CGS and SGD. The reason is that in these algorithms, a model parameter update only depends on an element of the training data matrix. As a result, all the model parameters can still be updated in one iteration. In addition, we tune the time setting to keeps the total amount of data entries trained within bounds and make sure the time used in computation can hide the communication time. This improves the efficiency of model rotation and further improves the model convergence speed.

E. Algorithm Examples

1) CGS: The training tokens are split according to the document and distributed to workers. There are also four models. The first one is the topic assignment on each token, which is stored with the training data. The next two are document-topic matrix and word-topic matrix. Because the training data is split by document, the document-topic matrix is partitioned with the documents while the word-topic matrix is rotated between workers. The last one is the token count sum on each topic. This is a small array with length equal to the number of topics, where all the elements are required in the local computation. As a result, we simply synchronize it with "allreduce" operation. We partition the documents on each worker into blocks. Inside each block, inversed indexing is used to group tokens by the word. The words' topic counts owned by the worker are also split into blocks. Thus 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, the amount of tokens which can be trained during a fixed time limit is growing larger. As a result, for time tuning, we keep an upper bound and a lower bound for the tokens trained in the time limit.

2) SGD: Both W and H are two model matrices. Assuming $n < m$, then V is regrouped by rows, W is partitioned with V, and H is the model for rotation. Since the computation load in each iteration does not change like what in CGS, we only tune the time limit to a specific value. By measuring the computation time and communication time at the first iteration, we estimate the ratio of computation cost and communication cost, then set the time limit to a value which meets the minimum requirement to overlap communication time with computation time.

V. Experiments

In this section, we test the efficiency of our model rotation approach using CGS and SGD. For each algorithm, we check the effectiveness of using pipeline and time control, and compare our implementations with other using different designs. We show that our solution has reliable scalability.

A. Training Dataset and Model Parameter Settings

Two datasets are used in the test. One is for CGS and another is for SGD (see Table. II). Both datasets are generated from a subset of the "ClueWeb09" dataset[2]. The model parameter settings result in similar sizes of the models for rotation. In CGS, theoretically the setting can bring the maximum 10 billion parameters to the model for rotation, but the real number is approximately 2 billion. In SGD, the model for rotation is dense with also approximately 2 billion model parameters.

B. Comparison Implementations

Four implementations are used in the tests. In CGS, we compare Harp CGS implementation with and without time control and Petuum LDA. In SGD, we use Harp SGD implementation with and without time control and NOMAD.

We focus on the comparison among different implementations using model rotation. The purpose is to show how the different programming interfaces and the related implementations can affect the model convergence speed. There are

[2]A collection of web pages, http://lemurproject.org/clueweb09.php/

Table II
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)

subtle differences in these implementations which can affect the experiment observation. Petuum LDA and NOMAD both are implemented in C++11 while Harp CGS/SGD are implemented in Java 8. Petuum LDA uses Open MPI for multi-processes and POSIX threads for multi-threading and ZeroMQ for communication. Though Petuum uses model rotation-based computation model in the distributed environment, it uses the computation model with asynchronous algorithm and stale model parameters in multi-threading, causing small difference compared with model rotation in model convergence. NOMAD uses MPICH2 for multi-process, Intel Thread Building Blocks for mult-threading and MPI_Send/MPI_Recv for communication. In NOMAD, the destination of each parameter shifting is randomly selected without following any ring topology.

C. Parallel Execution Environment

The Juliet cluster at Indiana University contains 32 nodes each with two 18-core 36-thread Xeon E5-2699 v3 Intel Haswell processors and 96 nodes each with two 12-core 24-thread Xeon E5-2670 v3 Intel Haswell processors. All the nodes have 128GB memory and are connected with Infiniband. During the test, JVM memory is set to "-Xmx120000m -Xms120000m -Xss4m -Xmn30000m" and IPoIB is used for communication.

The implementations are tested in three scales. 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 one 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 the three parallel execution environments have the same parallelism with 1800 threads in total. With using the same dataset and the same model settings, the data and model size for computation and communication per worker does not change when the scale changes. In this way, the scalability of model rotation can be presented.

D. CGS Performance Results

The performance results are presented in Fig. 6. We first describe the model convergence speed on different scales. And then we use the results on 60x30 to analyze the efficiency of model rotation. We also provide other detailed experiment settings here. The hyper-parameters α and β are both fixed to 0.01 during the whole execution without any tuning. For Harp with time control, we set the computation time after each model shifting to 1000ms on 30x60, 500ms on 60x30 and 333ms on 90x20 for the first iteration. Thus the total computation time for each worker at the first iteration is 60s. We enable time tuning so that the timer settings can be adjusted for the later iterations according to the current training progress.

Through examining the model likelihood achieved by the training time, the results on three scales all show that Harp with time control has the fastest model convergence speed compared with Petuum and Harp with no time control (see Fig. 6a, 6b, 6c). At the same time, Petuum is the second and Harp with no time control is the slowest. As time elapses, the model likelihood achieved by Petuum gets close to the value achieved by Harp with time control. We take the results on 60x30 (see Fig. 6b) as an example. When the model likelihood achieves -1.41×10^{11} and its change is within 1.00×10^{9}, Harp with time control has a $1.16\times$ speedup over Petuum and a $1.74\times$ speedup over Harp with no time control. When the model likelihood achieves -1.38×10^{11} and its change is within 5.00×10^{8}, Harp with time control has $1.15\times$ speedup over Petuum and a $1.65\times$ speedup over Harp with no time control. When the model likelihood achieves -1.36×10^{11} and its change is within 2.00×10^{8}, Harp with time control is $1.14\times$ faster than Petuum and $1.55\times$ faster than Harp with no time control.

To understand why Harp implementation with pipelining and time control performs better than Petuum, we still look into the results on 60x30 without repeating the similar results achieved on 30x60 and 90x20. Fig. 6d shows the model likelihood achieved by different implementations according to the number of iterations. Because of using the same computation model, Harp achieves the same model likelihood as Petuum. When time control is applied, Harp only training partial tokens per iteration so that the model likelihood achieved per iteration is not as high as the other two. However, when time control is used, the efficiency of pipelining in model rotation is improved. Fig. 6e shows the trend of iteration execution time as the time elapses. The computation time per iteration in Harp with time control is almost overlapped with iteration execution time, which means communication time is completely hidden inside of

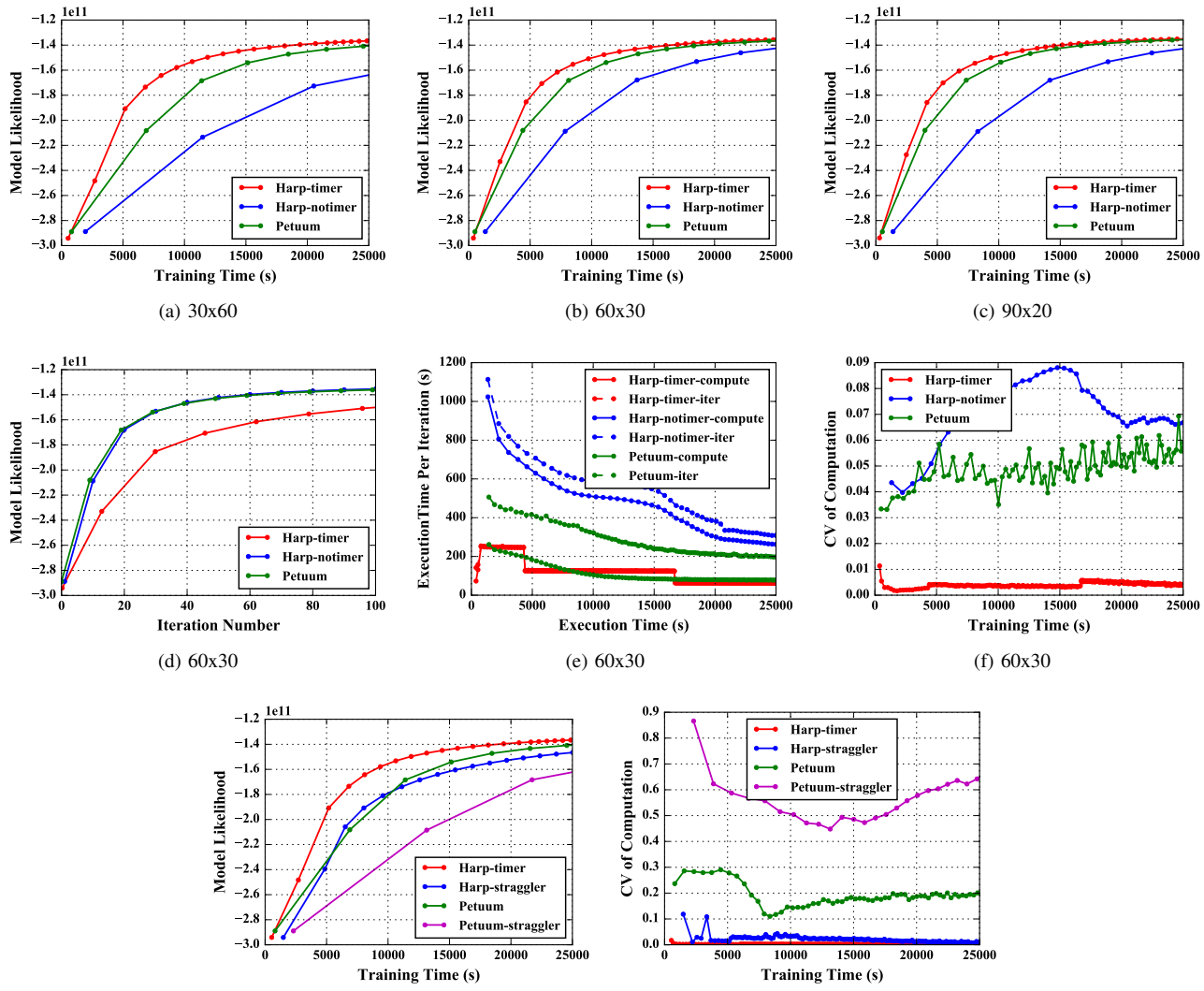


Figure 6. Performance Results on CGS (a) Model Likelihood vs. Training Time on 30x60 (b) Model Likelihood vs. Training Time on 60x30 (c) Model Likelihood vs. Training Time on 90x20 (d) Model Likelihood vs. Iteration Number on 60x30 (e) The Iteration Execution Time and the Computation Time per Iteration on 60x30 (f) The Coefficient of Variation of All the Workers' Iteration Computation Time on 60x30 (g) Model Likelihood vs. Training Time on 30x60 when a straggler exists (h) The Coefficient of Variation of Iteration Computation Time of All the Workers on 30x60 when a straggler exists

the computation time. Thus there is no additional overhead for model rotation. In contrast, both Petuum and Harp with no time control have high communication overhead in each iteration. Though pipelining is applied in both implementations, the uneven computation load on each worker makes overlapping the computation time and the communication time be difficult. Though Petuum's computation is highly optimized, its parameter level messaging pipelining is not efficient, even causing higher communication overhead compared with Harp with no time control. Fig. 6f gives a detailed look of the variation of the computation times from all the workers on each iteration. Harp with time control shows much lower variation than other implementations.

Using time control also makes the performance results more reliable. Fig. 6g show that on 30x60 when one node becomes a straggler (10 times slower in computation), Harp implementation with time control can maintain model convergence speed while Petuum becomes much slower. Fig. 6h shows the variation of the computation times from all the workers on each iteration. Even when one straggler exists, the variation of the Harp implementation with time control does not change much. But the variation of Petuum becomes very unstable.

E. SGD Performance Results

we present the performance results in Fig. 7. We firstly show the model convergence speed on the three scales. Then

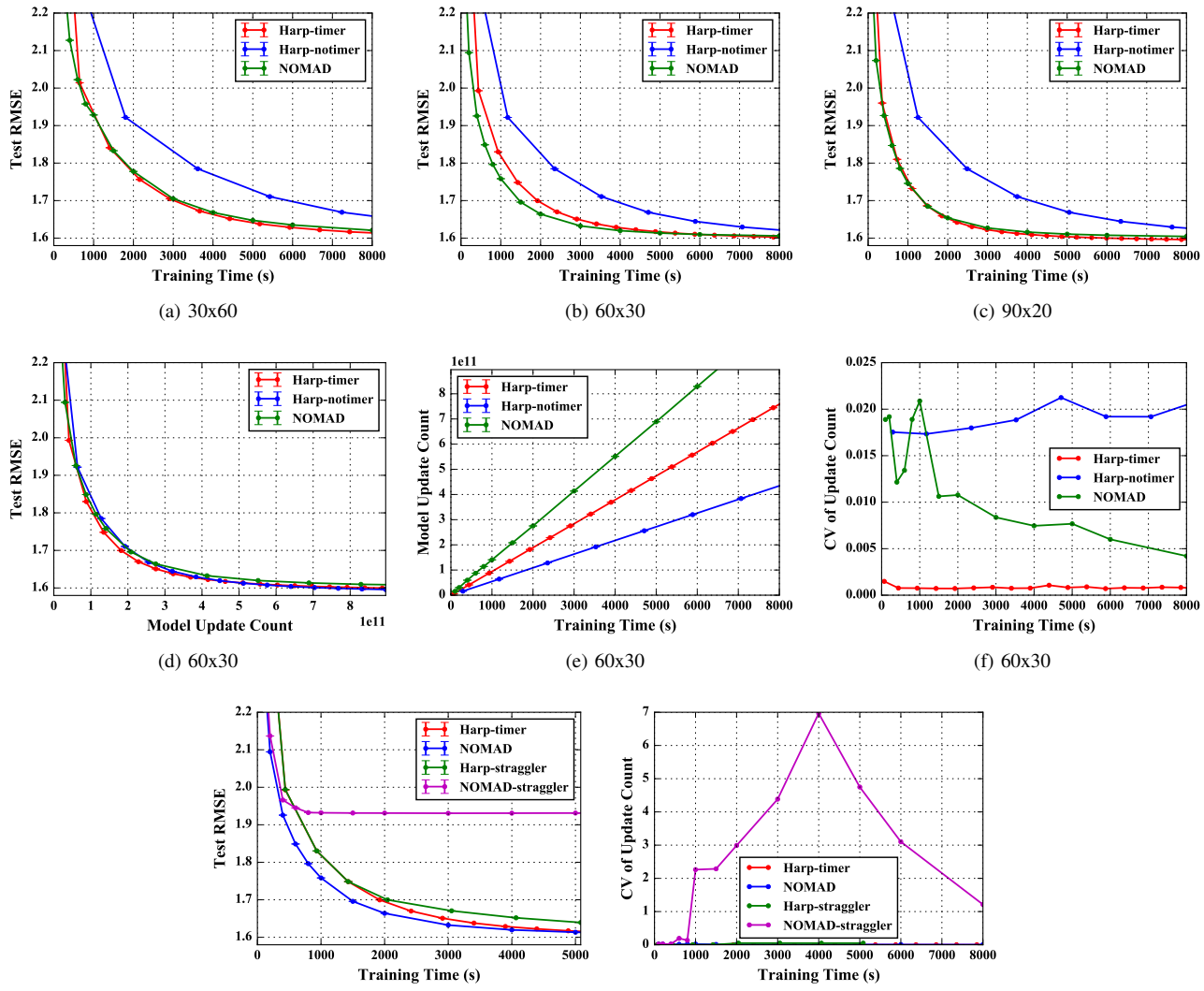


Figure 7. Performance Results on SGD (a) RMSE vs. Training Time on 30x60 (b) RMSE vs. Training Time on 60x30 (c) RMSE vs. Training Time on 90x20 (d) RMSE vs. Model Update Count on 60x30 (e) Model Update Count vs. Traning Time on 60x30 (f) The Coefficient of Variation of All the Workers' Iteration Computation Time on 60x30 (g) Model Likelihood vs. Training Time on 60x30 when a straggler exists (h) The Coefficient of Variation of Model Update Counts of All the Workers on 60x30 when a straggler exists

we use the results on 60x30 as an example to explain the efficiency of using model rotation. Here are the detailed settings in SGD algorithm. For the training hyperparameters, λ is set to 0.01 and ϵ is set to 0.001. Parameter tuning is disabled during the execution. For the timer settings, we use the same values at what we use in CGS experiments. The model convergence speed is evaluated by the RMSE value calculated by the trained W, H model matrices on the test dataset.

The performance results on 30x60, 60x30 and 90x20 are presented in Fig. 7a, 7b, and 7c. What we observe in these experiments are similar: Harp with time control performs better than the one with no time control. NOMAD is fast at the beginning. However, when the model is about to converge, its speed reduces and results similar model convergence speed and even slower than the speed of the Harp implementation with time control. We take the results on 60x30 (see Figs. 7b) as an example. When the RMSE value achieves 1.62 and its change is within 1.00×10^{-3}, Harp with time control is 14% slower than NOMAD but provides a $1.77\times$ speedup over Harp with no time control. When the RMSE value achieves 1.61 and its change is within 5.00×10^{-4}, Harp with time control is 1% slower than NOMAD but a $1.69\times$ speedup over Harp with no time control. However, when the RMSE value achieves 1.60 and its change is within 1.00×10^{-4}, Harp with time control

provides a $1.57\times$ speedup over NOMAD and a $1.47\times$ speedup over Harp with no time control.

The reason causing the issue of NOMAD's instable model convergence speed is its random model parameter shifting mechanism. We take a detailed look with using the results on 60x30. Fig. 7d shows that though using time control does not train all the elements in one iteration but it still achieves the same RMSE value as it without time control when the same number of model updates is performed. NOMAD is even slightly slower because the destination of model parameter shifting is randomly selected. As a result, a model parameter may go to the same training data partitions two successive training steps, causing model update be not effective. This problem is also shown in Fig. 7e. NOMAD can train more elements than Harp using the same amount of training time. But this does not provide effective contribution as the model update in Harp does. Random model parameter shifting may also cause unoptimized routing. Fig. 7f shows the variation of the model update counts from all the workers at a training time point. In this figure, Harp with time control shows very little difference of the model update count on each worker.

Similar to Petuum, NOMAD is also easily affected by stragglers. When all the parameters chose to go to the straggler, it devours all the parameters. In this case, the model stops converging (see Fig. 7g) and variation of the model update counts on each worker becomes large (see Fig. 7h).

VI. Related Work

Initially many researchers attempted to implement machine learning algorithm through a single solution. Mahout[7], Spark Machine Learning Library and Graph-based tools such as PowerGraph [8] are such examples. All these implementations follow the computation model with the synchronized algorithm and stale model parameters. Parameter Server [9] is a type of solution which follows the computation model with asynchronous algorithm and stale model. They put a programming interface which allows each worker to "push" or "pull" model parameters for local computation. Other implementations such as Yahoo! LDA [10][11] do not provide programming interface as Parameter Server but still follow the same computation model. All these implementations have two issues. One of them is that due to the big data problem, the training data on each worker may relate to many model parameters. Therefore the local model size can be very large which results in high memory usage and high communication cost. Another is that using the local model may break the model consistency in the original sequential algorithm so that the model convergence speed is decreased.

Model rotation is applied in a great deal of previous research work. NOMAD [12] and DSGD++ [5] implement model rotation-based computation model as independent applications, which does not provide any programming interface. Besides they manage model rotation as point-to-point messaging but not in a collective way. Another work, Petuum STRADS [13], tries to include model rotation in a general parallelism solution called "model parallelism" through "schedule-update-aggregate" interfaces. algorithms implemented by this framework often use model rotation but not with clear specification. The vague abstraction results in inconsistency between computation models and algorithm implementations in Petuum STRADS. For example, Petuum CGS implementation uses model rotation. But its Cyclic Coordinate Descent (CCD) implementation uses "allgather" operation to collect model matrices without any rotation. The interfaces of Petuum STRADS may still work at the model parameter level but does not manage the model as a whole dataset, resulting in inefficiency in performance. Despite of these shortcomings, Petuum CGS and NOMAD are still the fastest implementations we know among open-source implementations of the two algorithms.

VII. Conclusion

For the big model problem in big data machine learning applications, this paper gives a full solution including computation model selection, programming interface design, and implementation improvements. Comparing four computation models, we conclude model rotation-based computation model is suitable for machine learning applications with big data and big model. Next, our MapCollective programming interface is more convenient than other implementations' parameter-level APIs. Finally the two mechanisms of pipelining and time control in implementation can help to improve the model convergence speed.

In the future we can apply our solution to algorithms other than CGS and SGD. We do not claim that our model rotation solution is the silver bullet for all machine learning applications and all parallel execution environments. For example, when the number of parallel worker keeps increasing, each worker may only need to handle a small amount of the data and related model parameters. As a result, other computation models may be more suitable depending on the property of data and the configuration of the parallel execution environment. As such in potential future, a research direction is to provide configurations to parallelization solutions to guide developers in understanding the applicability of each solution.

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] B. Zhang, Y. Ruan, and J. Qiu, "Harp: Collective Communication on Hadoop," in *IC2E*, 2015.

[2] L. Yao, D. Mimno, and A. McCallum, "Efficient Methods for Topic Model Inference on Streaming Document Collections," in *KDD*, 2009.
[3] B. Zhang, B. Peng, and J. Qiu, "Model-Centric Computation Abstractions in Machine Learning Applications," in *BeyondMR*, 2016.
[4] D. Newman *et al.*, "Distributed Algorithms for Topic Models," *The Journal of Machine Learning Research*, vol. 10, pp. 1801–1828, 2009.
[5] C. Teflioudi, F. Makari, and R. Gemulla, "Distributed matrix completion," in *Data Mining (ICDM), 2012 IEEE 12th International Conference on*. IEEE, 2012, pp. 655–664.
[6] B. Zhang, B. Peng, and J. Qiu, "High Performance LDA through Collective Model Communication Optimization," in *ICCS*, 2016.
[7] C.-T. Chu *et al.*, "Map-Reduce for Machine Learning on Multicore," in *NIPS*, 2007.
[8] J. E. Gonzalez *et al.*, "Powergraph: Distributed graph-parallel computation on natural graphs," in *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, 2012, pp. 17–30.
[9] M. Li *et al.*, "Scaling Distributed Machine Learning with the Parameter Server," in *OSDI*, 2014.
[10] A. Smola and S. Narayanamurthy, "An Architecture for Parallel Topic Models," *VLDB*, vol. 3, no. 1-2, pp. 703–710, 2010.
[11] A. Ahmed *et al.*, "Scalable Inference in Latent Variable Models," in *WSDM*, 2012.
[12] H. Yun *et al.*, "NOMAD: Non-locking, stOchastic Multi-machine algorithm for Asynchronous and Decentralized matrix completion," *Proceedings of the VLDB Endowment*, vol. 7, no. 11, pp. 975–986, 2014.
[13] S. Lee *et al.*, "On Model Parallelization and Scheduling Strategies for Distributed Machine Learning," in *NIPS*, 2014.
[14] J. Qiu and B. Zhang, "Mammoth Data in the Cloud: Clustering Social Images," *Cloud Computing and Big Data*, vol. 23, p. 231, 2013.
[15] B. Zhang and J. Qiu, "High Performance Clustering of Social Images in a Map-Collective Programming Model," in *SoCC*, 2013.