HarpLDA+: Optimizing Latent Dirichlet Allocation for Parallel Efficiency

Bo Peng[1] Bingjing Zhang[1] Langshi Chen[1] Mihai Avram[1] Robert Henschel[2] Craig Stewart[2]
Shaojuan Zhu[3] Emily Mccallum[3] Lisa Smith[3] Tom Zahniser[3] Jon Omer [3] Judy Qiu[1]
[1]School of Informatics and Computing, Indiana University
[2]UITS, Indiana University
[3]Intel Corporation
{pengb, zhangbj, lc37, mavram, xqiu}@indiana.edu
{henschel, stewart}@iu.edu
{shaojuan.zhu, emily.l.mccallum, lisa.m.smith, tom.zahniser, jon.omer}@intel.com

***Abstract*—We present HarpLDA+, a sychronized trainer for Latent Dirichlet Allocation(LDA). With focusing on the parallel efficiency aspects of the system design, two mechanisms, dynamic scheduling and timer control, are proposed respectively to reduce the synchronization overhead in shared memory and distributed environment. Experiments on diverse datasets and parallelism settings show that HarpLDA+ outperforms some state-of-the-art systems. Our method is efficient and scalable.**

I. Introduction

Latent Dirichlet Allocation (LDA) [1] is a widely used machine learning technique in topic modeling and data analysis. LDA training are iterative algorithms, starting from a randomly initialized model(parameters to learn) and iteratively computing and updating the model until it converges. It is an irregular computation with a model size that can be huge and changes as one iterates to convergence. Meanwhile, parallel workers need to synchronize the model. State-of-the-art LDA trainers are implemented to handle billions of documents, hundreds of billion tokens, millions of topics and millions of unique tokens. However, the pros and cons of different approaches in the existing tools are often hard to explain because many trade-offs between effectiveness and efficiency of model updates and implementation details impact the performance of LDA training systems. One of the popular trade-offs is to decrease the time complexity of the computation by introducing approximations. Another widely used one is to reduce the sychronization overhead by using an asynchronous design working on stale model. In this paper, we propose a different approach to design a high performance trainer. Our main contributions can be summarized as follows:

- Review state-of-the-art LDA training systems and summarize their design features. Analyze learning algorithms for parallel efficiency.
- Propose new mechanisms to reduce overheads in a synchronized system, dynamic scheduling for shared memory system and Timer Control for distributed system.
- Implement HarpLDA+ based on Hadoop and demonstrate excellent performance and scalability.
- Summarize our system design approach and its implications for other machine learning algorithms.

The outline of this paper is as follows: Section II introduces the background of the LDA algorithm and related work, while Section III analyzes the architecture and parallel efficiency of existing solutions. Section IV describes our system design and implementation details of HarpLDA+ and Section V presents experimental results coupled with a performance analysis. Finally, Section VI draws conclusions and discusses future work.

II. LDA Algorithm and Related Work

A. LDA with Collapsed Gibbs Sampling

LDA is a topic modeling technique to discover latent structures inside data. When data is represented as a collection of documents, where each document is a bag of words, LDA models each document as a mixture of latent topics and each topic as a multinomial distribution over words.

Many algorithms have been proposed to estimate the parameters for the LDA model. CGS (Collapsed Gibbs Sampling) [2], a Markov chain Monte Carlo (MCMC) algorithm, is widely adopted for large scale LDA training. In the MCMC framework, samples can be drawn according to the unknown posterior distribution by a carefully designed transition function that visits the whole parameter space. Gibbs sampling is one such design that visits the parameter space from one dimension to the other. For each iteration, it fixes all the states of other dimensions and only updates the current visiting one.

In CGS, each training data point or token is assigned to a random topic denoted as z_{ij} at initialization. Then it begins to reassign topics to each token $x_{ij} = w$ by sampling from a multinomial distribution of a conditional probability of z_{ij} as shown below:

$$p\left(z_{ij} = k \mid z^{\neg ij}, x, \alpha, \beta\right) \propto \frac{N_{wk}^{\neg ij} + \beta}{\sum_w N_{wk}^{\neg ij} + V\beta} \left(M_{kj}^{\neg ij} + \alpha\right) \tag{1}$$

Here superscript $\neg ij$ indicates that the corresponding token is excluded. V is the vocabulary size, N_{wk} is the token count

of word w assigned to topic k in K topics, and M_{kj} is the token count of topic k assigned in document j. The matrices Z_{ij}, N_{wk} and M_{kj}, form the model to be learned. Hyper-parameters α and β control the topic density in the final model. The model gradually converges during the process of iterative sampling.

Although CGS generally requires a large number of iterations to converge, it is memory efficient and therefore salable for large models. In this paper, we focus on the LDA trainers under the umbrella of the CGS algorithm.

B. Related Work on Parallel LDA-CGS

Gibbs sampling in LDA-CGS is a strictly sequential process. AD-LDA (Approximate Distributed LDA)[3] proposed to relax the requirement of sequential sampling of topic assignments based on the observation that the dependence between the update of one topic assignment $z_{i,j}$ and the update of any other topic assignment $z_{i',j'}$ is weak. In AD-LDA, the distributed approach is to *partition* the training data for different workers, run local CGS training and *synchronize* the model by merging back to a single and consistent set of N_{wk}. PLDA [4], implemented the AD-LDA algorithm in both MPI and MapReduce, where the Allreduce operation is used for synchronization.

A synchronized algorithm that requires global synchronization at each iteration sometimes may not seem feasible or efficient; Therefore, an *asynchronous* solution becomes the alternative choice. Async-LDA [5] extended AD-LDA to an asynchronous solution by a gossip protocol. [6][7] created the first production level LDA trainer called Yahoo!-LDA. The mechanism is an asynchronous reconciliation of the model, one word at a time for all samplers. Furthermore, [8] introduced Parameter Server as a general framework that scaled to thousands of servers. Another progression was presented by [9]. It proposed a "mixed" approach SSP (Stale Synchronous Parallel), which is a parameter server that can limit the maximum age of the staleness.

Some researchers have investigated synchronized algorithms. For instance, [10] proposed a novel data partitioning scheme to *avoid memory access conflicts* on GPUs. The basic idea is to partition the training data into blocks, where all samplers start from the diagonal blocks and then shift to the right neighbor all together. In contrast, [11][12] extended this idea to a general machine learning framework, Petuum Strads, where parameters of the ML program were partitioned for different workers. As the all-to-all communication observed in the asynchronous trainers is hard to optimize, [13] HarpLDA adopted a similar synchronized design and proposed collective communication operators which achieved better performance. Finally, [14] introduced F+Nomad-LDA based on idea of NOMAD[15], in which each variable (one row of N_{wk}) becomes the basic unit to be scheduled, and the ownership of a variable is asynchronously transferred between workers in a decentralized fashion.

Other research involves optimizations on the sampling algorithm. According to equation (1), a naive implementation involves drawing a sample from a discrete distribution which contains two steps: first calculate the probability of each event as $p(z_{ij} = k), k \in K$, secondly generate a random number uniformly from $[0 - 1)$ and search linearly along the array of the probabilities, stopping when the accumulation of probability mass is greater than or equal to the random number. The time complexity for this is $\mathcal{O}(K)$. [16] SparseLDA decomposed numerator of equation (1) into three parts: $\alpha\beta$, $\beta * M_{kj}^{\neg ij}$ and $N_{wk}^{\neg ij}(M_{kj}^{\neg ij} + \alpha)$. The first part is a constant; while the second part is non-zero only when M_{kj} is non-zero, and the third part is non-zero only when N_{wk} is non-zero. Both the probability calculation and search part can benefit from utilizing the characteristics of this sparseness pertaining to the model. When using this feature, the computation time complexity drops to $\mathcal{O}(K_d + K_w)$, equivalent to the average non-zero items number in column of M_{kj} and row of N_{wk}, which are typically much smaller than K. F+Nomad-LDA [14] provides an optimization on the search part by using a $\mathcal{O}(logK)$ binary tree search instead of a $\mathcal{O}(K)$ linear search by a tree data structure. Baesed on Alias Table which allows us to draw subsequent samples from the same distribution in $\mathcal{O}(1)$ time, Alias-LDA [17] uses another sampling algorithm called Metropolis Hasting (MH) to draw each sample correctly from the stale alias table and achieves $\mathcal{O}(K_d)$ complexity. [18] LightLDA extends the Alias-LDA idea by decomposing equation (1) into two parts and alternating the proposals into a cycle proposal, thus achieving $\mathcal{O}(1)$ complexity. [19] WarpLDA introduces a more aggressive approach based on the idea of MH to delay all the updates after sampling one pass of Z, by drawing the proposals for all tokens before computing any acceptance rates.

III. Parallel Design Principles

A. Parallel Efficiency

Parallelizing a sequential algorithm inevitably introduces overhead. *Communication overhead* comes from the additional cost of moving data around the parallel workers. In a shared memory system, this overhead is generally ignored with the assumption of a uniform memory access cost. The notion of overlapping communication with computation is a key design choice for high performance distributed systems. In this case, asynchronous communication and pipelining are two standard solutions. *Synchronization overhead* comes from the additional cost of coordinating parallel workers that reach the same state in order to finish a task together. Asynchronous trainers, such as ones using a parameter server, try to reduce this type of overhead by avoiding a global consensus, relaxing the consistency of the model and working in an independent fashion. Synchronized trainers, however, will face the issue of *load imbalance*, which is a major source of synchronization overhead. Imbalanced

Trainer	Sampler	Sampling Time Complexity	Intra-node Design	Inter-node		Model
				Design	Comm	
PLDA	PlainLDA	$\mathcal{O}(K)$	Allreduce	Allreduce	collective	stale
Yahoo!LDA	SparseLDA	$\mathcal{O}(K_d + K_w)$	Allreduce	Asynchronous	async	stale
StradsLDA	SparseLDA	$\mathcal{O}(K_d + K_w)$	Allreduce	Rotation	async	stale
LightLDA	MH	$\mathcal{O}(1)$	Asynchronous	Asynchronous	async	stale
F+NomadLDA	F+Tree	$\mathcal{O}(\log K_d + \log K_w)$	Rotation	Rotation	async	latest
WarpLDA	MH	$\mathcal{O}(1)$	DelayUpdates	Rotation	collective	stale
HarpLDA+	SparseLDA	$\mathcal{O}(K_d + K_w)$	Rotation	Rotation	collective	latest

Table I: System Architectures of LDA Trainers. *AllReduce*, works on stale model and do synchronization on model replicas. *Asynchronous*, works on local replicas and synchronizes them through a group of parameter servers in a best effort. *Rotation*, works on distributed model partitions and the model partitions should 'rotate' among the workers while at the same time keeping model updates conflict free.

workloads lead some workers waiting for the other busy workers within the synchronization operation, degrading the parallel efficiency in the system.

Some data partitioning algorithms have been proposed that aim to improve load balancing for LDA training. For example, random permutations on the document usually give good results. Some algorithms partition the word-topic model, whereas randomized algorithms do not perform as good as greedy algorithms [13][19] since the word frequency follows the power-law distribution. Unfortunately, even optimal partitioning algorithms cannot completely solve the load imbalance problem. Sampling algorithms may perform differently on the same number of tokens with different distributions. In practice, variations of node performance and stragglers are not uncommon even in homogeneous HPC clusters.

Parallel efficiency can be meaured by *Speedup*, which is defined as parallel performance over original sequential performance in parallel processing. When generalize *Amdahls law* with overhead incurred due to parallelizing, we have:

$$Speedup = \frac{T_{original}}{T_{enhanced}} = \frac{1}{f + s + \frac{1-f}{P}} \leq \frac{P}{1-f} \qquad (2)$$

With P workers, f is the serial portion that cannot be parallelized, s is the portion of the overhead time introduced by parallelism. In parallel CGS, f is small, the overhead time s becomes the major issue to pursue a good parallel efficient system.

B. System Architectures

Machine learning algorithms can generally tolerate some kind of staleness in the model. Using stale models in computation can degrade the convergence rate but potentially boost the system efficiency because it relaxes the constrains for system design. The trade-off between effectiveness and efficiency is critical for the parallelization of these algorithms. For example, the sum of topic count $\sum_w N_{wk}$ in the denominator of equation (1) is hard to keep strict consistency in a parallel setting. Using locks on data being frequently accessed will give poor performance. A typical solution is to remove the locks and keep using a local copy of the model partition. Furthermore, synchronizing the model at the end of each epoch is good enough as the deviations are small. But the decision of whether to use stale values of N_{wk} and M_{kj} in the numerator of equation (1) is more sensitive.

In order to better present HarpLDA+'s design, we summarize the features and parallel architectures of current CGS trainers in a Model-Centric view (See Table I).

IV. HarpLDA+: Design and Implementation

HarpLDA+ builds upon Harp[1], which is a Java collective communication library released as a plugin for Hadoop. While our previous work of LDA trainer optimizes communications in different architectures, HarpLDA+ focuses on the Rotation architecture and reducing the synchronization overhead.

A. Programming Model Based on Collective Communication

Using collective communication within a Rotation design is easy to program. For each iteration, all workers concurrently sample on a local training data partition with a local model split without conflicts in model updates. Afterwards, a call to a collective communication operator 'rotate' is made, in order to do global scheduling. (see Algorithm 1)

A concrete scheduling strategy is encapsulated inside the 'rotate' operator. So long as each model split is owned by only one worker, the scheduling strategy guarantees to be conflict free. For instance, when a rotate call returns, all the workers can continue sampling concurrently without causing conflicts when updating the model. A default strategy shifts the model splits to their neighbor nodes (see Fig. 1a). Selecting the neighbor on a random permutation of the node list is also easy to implement. Furthermore, a priority based scheduler and work load based scheduler can be implemented in this framework without losing the simplicity of the programming model.

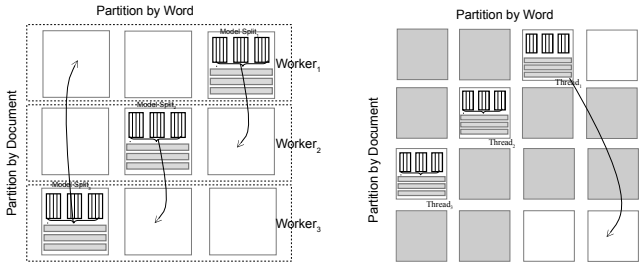
Algorithm 1 presents a general framework for scheduling, where multi-threading and distributed parallelism can adopt the same procedure. We can however improve it to reduce

[1]https://dsc-spidal.github.io/harp/

Algorithm 1: HarpLDA+ Parallel Pseudo Code

input : training data X, P workers, model A^0, number of iterations T
output: A^T

```
1 parallel for worker p ∈ [1, P] do
2     for t = 1 to T do
          // initialize: model A^{t_0} is A^{t-1}
3         for i = 1 to P do
              // update local model split by
                 sampling on local training
                 data
4             A_{p'}^{t_i} = Sampling(X_p, A_{p'}^{t_{i-1}})
              // synchronization to exchange
                 model splits
5             rotate(A_{p'}^{t_i})
```



(a) model rotation (b) dynamic scheduling

Figure 1: Model Rotation Framework and Dynamic scheduling in Shared Memory

synchronization overhead, leveraging the computation characteristics in these two different environments.

B. Dynamic Scheduling in Shared Memory Systems

Dynamic scheduling provides a low cost solution to remove synchronization overhead in a shared memory system. To keep the faster workers busy, it creates a more spare workload in the beginning and dynamically selects an unoccupied one to feed into the first finished worker. This is an effective solution, which also occurs in parallel matrix factorization design [20].

As in Fig. 1b, training data is partitioned into blocks, with the row partition using a random permutation of document id and the column partition using a greedy algorithm based on word frequency. Indexes are constructed during the initialization phase in order to build the map from word id to the related documents appearing in each block. In this case, the minimal unit for scheduling is a block. Furthermore, the partition number is larger than the thread number, which means that there are always spare rows and columns when one working thread finishes its current task. In this example, thread 1 finishes its work in the first place, then the scheduler can select a new block randomly from the 'free' blocks, which are the white blocks in the figure. Because thread 2 and thread 3 are still working, the rows and columns are occupied accordingly as denoted by the gray blocks. In a shared memory system, the scheduler does not move data but instead assigns data addresses of the selected free blocks to the idle threads. The wait time of the working threads are bounded by the overhead of the scheduler. In case the thread number is P and the splits number is L, a $L \times L$ matrix maintains a two level status: free, or finished. The scheduler can randomly select a free block by scanning the matrix with time complexity of no more than $\mathcal{O}(L^2)$. The larger the L, the lesser the conflicts and wait time, but the more overhead introduced by the scheduler itself. Thus, there is a trade-off. By experimentation, we found that $L = \sqrt{2}P$ is a good choice in most cases.

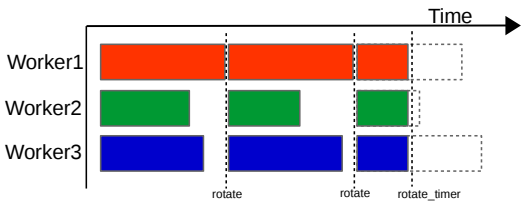
C. Pipelining and Timer Control in Distributed Systems

In distributed systems, the cost of data movement cannot be omitted, thus the dynamic scheduling approach does not work anymore. As in Fig. 1a, each worker holds a static row partition of the training data and corresponding document related model. Only the word-topic model partitions move among the workers. To reduce the synchronization overhead, the first step is to reduce the overhead of the communication inside the rotate operator. Pipelining is a broadly used technique to solve this kind of problem, by overlapping I/O threads with computing threads. First, each block is split further into two slices horizontally, and the inner loop of algorithm 1 is modified as a loop on each slice. Consequently, the original rotate call becomes two rotate calls on each slice. As long as the communication time is less than the computing time spent on one slice, the pipeline will be effective in removing the overhead from communication.

Another overhead of a rotate call is the time to wait for all workers to finish their computation. Due to load imbalance, the slowest worker will force all other workers to wait for it until it finishes its computation.

To solve this problem, we first discuss the sampling order of the Gibbs Sampling Algorithms. We note that LDA trainers can use two common scan orders: random scan and deterministic scan. For a Gibbs sampler, the usual deterministic-scan order proceeds by updating first x_1, then x_2, then $x_3, \ldots x_d$ and back to x_1, visiting the state space X by a sequential order. Another random scan version usually proceeds at each iteration by choosing i uniformly from $1, 2, ..., d$. [21] demonstrating that the order really matters for the convergence rate of different models, although due to the benefits of locality in hardware, a deterministic scan is commonly used. This is the situation in current LDA trainers, in which sampling occurs over document or over word on Z, via deterministic scan. Generally, the order with

a better memory cache hit rate gives a better performance. For large datasets with $V << D$, word order is better. It is hard to achieve good performance with a pure random scan due to the cache miss issues. However, HarpLDA+ uses a quasi-random order. The dynamic scheduler picks a block uniformly from the free block list. While inside the block, we still keep the word order during sampling. Although no significant performance difference is observed for the different sampling orders in LDA-CGS, we found that the random order provide a natural solution for load balancing.



(a) Timer Control

Figure 2: Timer to Control the Synchronization Point

See Fig. 2, wait time is the region between the end of computation and the start of synchronization, also known as the time point to call rotate operation. If we adjust the synchronization point ahead of when the computation finishes, the gap of wait time can be closed. Under the deterministic scan order, the adjustment is harder due to the housekeeping work needed and the original scan order lost. For a random scan, this adjustment does not change the property of the uniform random selection of blocks. The third rotate call demonstrates this mechanism in Fig. 2.

We further propose a simple solution for the LDA-CGS trainer. Each sampler just works for the same period of time and then the samplers do synchronization all together. They all use a *timer* to *control* the synchronization point other than waiting until all the blocks to finish. Because the model size shrinks and the computation time drops during the process of convergence, we've designed an auto-tuning mechanism to set the value of the timer for each iteration in HarpLDA+.

First, the timer works best when the communication can be fully overlapped by computation, where the computation time or the number of the training data points being processed should have a lower bound L. Secondly, we make sure that all workers stop at the same time before any of them finishes. This implies an upper bound H. L and H are set as input parameters. In normal cases, $L = 40\%$,$H = 80\%$ are good choices.

We set up heuristic rules to automatically determine the values of timer t_i based on the L, H settings.

- Rule 1: During the first iteration, we set the timer to a constant t_0, and obtain the processing ratio R_0 for each worker at the end of the iteration.
- Rule 2: When R_i is found to be smaller than L, adjust $t_{i+1} = t_i * 2$ in order to quickly catch up. (In the first iteration, repeat this step until R_{i+1} is in the range of L and H.)
- Rule 3: When R_i is found to be larger than H, t_{i+1} will be reduced in half.

D. Other Implementation Issues

For a high performance parallel LDA trainer, besides the key factor of the original sampling algorithm and the parallel system design, some implementation details may also be important.

HarpLDA+ is a Java application, where primitive data types are used in critical data structures. E.g., we found that using primitive arrays with array indexing for the model matrix is significantly faster than using a hashmap in HarpLDA+.

Furthermore, minor improvements for SparseLDA are very helpful. Topic counts are sorted periodically to reduce the linear search time of sampling. Caching is also used to avoid repeat calculations. When sampling multiple tokens with the same word and document, the topic probabilities calculated for the first token are reused for the tokens that ensue.

V. Experiments

A. Setup of Experiments

Dataset	$Docs$	$Vocabulary$	$Tokens$	$DocLen$
nytimes	299K	101K	99M	332/178
pubmed2m	2M	126K	149M	74/33
enwiki	3.7M	1M	1B	293/523
bigram	3.8M	20M	1.6B	434/767
clueweb30b	76M	1M	29B	392/532

Table II: Datasets for LDA Training, where $DocLen$ represents mean and std. dev. values of document length

Five datasets (see Table. II) are used in the experiments, which are open datasets that appear in related works frequently. The number of documents of a dataset varies from 300 Thousand to 76 Million, and the vocabulary of a dataset also varies from 100 Thousand to 20 Million. Finally, the total number of tokens ranges from 99 Million to 29 Billion. Thus, these datasets are diverse and representiative for our thorough experimentation and evaluation.

trainer	language	multithreading	communication
LightLDA	C++	Pthread	Zeromq+MPI
NomadLDA	C++	Intel TBB	MPI
WarpLDA	C++	OpenMP	MPI
HarpLDA+	Java	Java Thread	Harp Collective

Table III: Trainers for Experiments

We select four state-of-the-art CGS trainers for comparison in Table. III. They represent different system designs

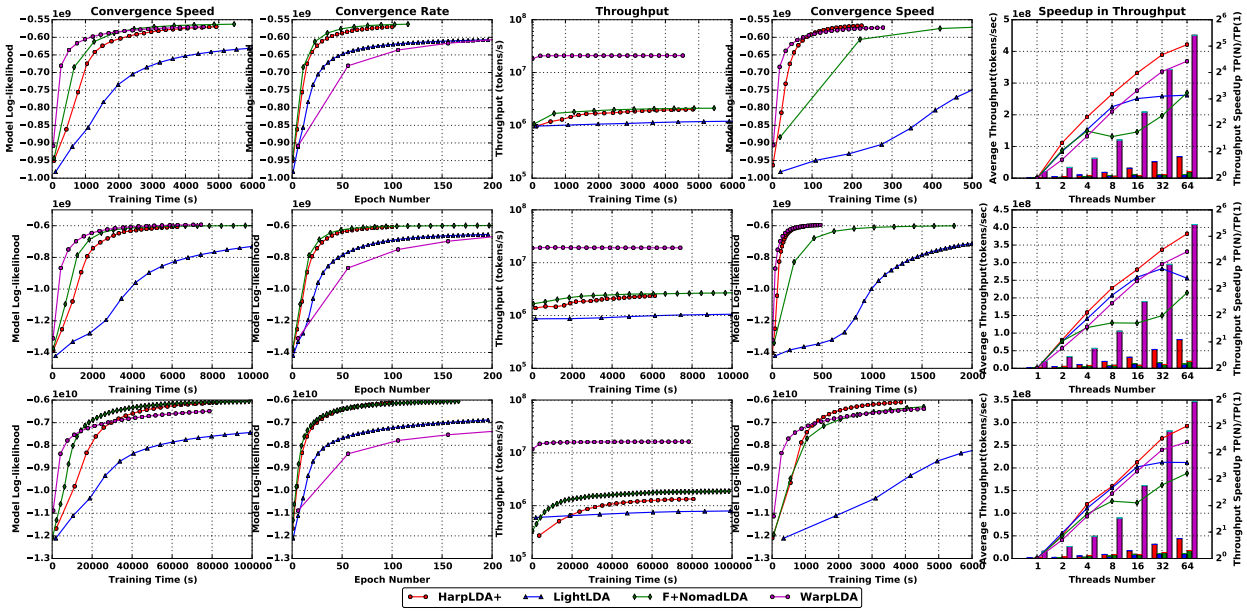


Figure 3: Single Node Performance on nytimes (K=1K, 1st row), pubmed2m (K=1K, 2nd row), enwiki (K=10K, 3rd row). Column 1 to 3 are results of single thread, column 4 is result of 32 threads.

of Section III-B. To evaluate the performance, we use the following metrics. Firstly, we choose *Model log likelihood* of the word-topic model to represent the status of convergence. Secondly, we select three main evaluation metrics as follows: 1) *Convergence rate* evaluates the effectiveness of the algorithm by depicting the relationship between convergence level and model update count. 2) *Throughput* evaluates the efficiency in a system perspective view by measuring the model update counts per second. 3) *Convergence speed* is the metric to evaluate a trainer's overall performance, which depicts the relation between convergence level and training time. It represents the overall performance resulting from the combination of efficiency and effectiveness of model updates.

In regards to hardware configuration, all 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. As for the software configuration, all C++ trainers are compiled with gcc 4.9.2 and -O3 compilation optimization. HarpLDA+ compiles with Java 1.8.0 64 bit Server VM and runs in Hadoop 2.6.0. The MPI runtime is mvapich2 2.3a for F+NomadLDA and mpich2 3.0.4 for LightLDA. For MH trainers, we set the MH step parameter to 1 for WarpLDA and select best one for LightLDA, 16 for clueweb and 4 the other datasets. We set the hyper-parameters $\alpha = 50/K$ and $\beta = 0.01$ in all the experiments.

B. Experimental Results

1) Performance of Sequential Algorithm: We first analyze the performance of the sampling algorithm by evaluating the trainers in a single thread setup.

As shown in Fig. 3 column 2, *Convergence Rate*. Standard SparseLDA sampler, NomadLDA, is always the fastest. HarpLDA+ is a bit slower due to the caching of the model for identical words. Both MH samplers, LightLDA and WarpLDA are significantly slower because they are an approximation for the original CGS, while WarpLDA is the slowest because of its update delay strategy making each update much less effective. The order of convergence rate is constantly stable in parallel versions of these trainers.

In column 3, *Throughput*. WarpLDA has much better throughput than the others due to its memory optimization by removing the random matrix access. Among the others, NomadLDA performs a little better.

In column 1, *Convergence Speed*. WarpLDA is the fastest trainer due to its successful trade-off between the efficiency and effectiveness of updates. Furthermore, F+Nomad-LDA is faster than HarpLDA+ and demonstrates even better performance than WarpLDA in the case of large K. LightLDA is constantly the slowest.

2) Intra-node Parallel Efficiency: We test on increasing number of threads and its impact on performance as seen in Fig. 3. In column 4, the convergence speed at 32 threads shows that the rank of NomadLDA drops, and HarpLDA+ takes its place while running as well as WarpLDA and even exceeds at large K. In Fig. 3 column 5 *Speedup in Throughput*, the bar chart shows the average throughput

increasing along with the parallelism, and the SpeedUp shows the parallel efficiency of these increases. HarpLDA+ demonstrates the best parallel efficiency which explains the boost of its performance from a single thread to a large number of threads.

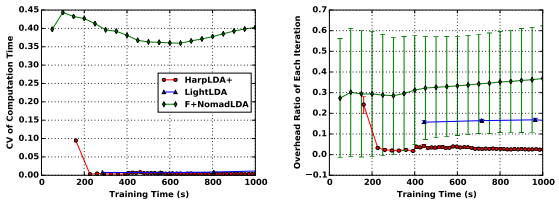
Trainer	CPU Time			Wait
	Effective	Spin	Overhead	Time
WarpLDA	0.91	0	0	0.09
NomadLDA	0.75	0.24	0	0
LightLDA	0.25	0	0	0.74
HarpLDA+	0.98	0	0	0.02

Table IV: Time Breakdown by VTune Concurrency Analysis. *Effective Time* is CPU time spent in the user code, *Spin time* is wait time during which the CPU is busy, and *Overhead time* is CPU time spent on the overhead of known synchronization and threading libraries, *Wait Time* occurs when software threads are waiting due to APIs that block or cause synchronization. Run with parameters: Dataset = enwiki, $K = 1K$, 1 Node and 32 Threads.

Concurrency Analysis by VTune Amplifier[2] is utilized to exhibit the time breakdown with normalized results in Table. IV. WarpLDA demonstrates excellent efficiency, as it not only decouples the memory access to the two model matrices but also removes the model update conflicts, in which all threads are running in a pleasingly parallel fashion programmed in OpenMP. In this implementation, load imbalance is observed to contribute to the 9% wait time. This may come from the default static scheduler in OpenMP. NomadLDA has zero wait time, but this does not necessarily signal efficiency. All threads keep trying to pop a model column from the concurrent queue to run sampling, and yield when the pop call fails. A large number of yield calls are observed to give 24% on spin time. Load imbalance is the main reason behind the inefficiency as well. F+NomadLDA supports different kinds of schedulers, but in our test, the default Shift version and the Load Balance version do not show much differences. LightLDA shows a very high Wait Time ratio. After analysis of the hot-spots, a problem is found in the thread safe queue code. At the end of each iteration, all sampling threads push the updated model (delta actually) to a shared queue which will later be pushed to the parameter server by aggregator threads. High contention for this object causes reduced parallel efficiency. HarpLDA+ wins in this test to perform the best with only 2% wait time. When comparing with other trainers, the overhead in our Java dynamic scheduler is much less than what we expected.

As VTune profiling has limitations to further breakdown the actual working time inside the effective time for specific applications, we add thread level logs to record the actual sampling time in each iteration. See Fig. 4a, F+NomadLDA has a very large CV level depicting serious problems with

[2]https://software.intel.com/en-us/intel-vtune-amplifier-xe



(a) Load Balance (b) Overhead Ratio

Figure 4: Load Balance and Overhead Ratio. CV (coefficient of variation) is the ratio of the standard deviation to the mean of the sampling time. Overhead time for each thread is the iteration time excluding the actual sampling time spent.

load imbalance. Fig. 4b, F+NomadLDA again shows a high overhead ratio and variance. LightLDA is better but still larger than 10%. In contrast, HarpLDA+ presents a relatively large overhead ratio in the first iteration because a fixed timer of 1 second is set in the beginning, while constant overheads of hundreds of milliseconds make the ratio value appear high. As seen in the charts, HarpLDA+ demonstrates the best load balance and a small overhead. WarpLDA is excluded in this experiment beacause it is implemented with OpenMP and thread log can not be added.

3) Distributed Parallel Efficiency: In this section, we test the LDA trainers in distributed mode. WarpLDA is not included because the official source code release does not support distributed mode. Moreover, we expect that the distributed design presented in its paper might not scale well because of the need to exchange the whole training data-set in each iteration among all the workers. F+NomadLDA runs on an InfiniBand network directly supported by mvapich2, but lightlda runs on IPoIB (TCP/IP protocol on InfiniBand network) supported by mpich2, and as a Java application, HarpLDA+ runs on IPoIB too. This means F+NomadLDA can potentially utilize a bandwidth which is at least two times larger in these experiments and be easier to scale. LightLDA adopts a parameter server approach and reports model likelihood on behalf of each local model, the actual likelihood of the global model is estimated by a re-calibration of its reported value with a constant gap. A version of HarpLDA+ without timer control is included, named Harp-notimer.

As in Fig. 5, column one represents convergence speed, where HarpLDA+ has the best overall performance. In column two, called speedup of time, harp-timer is used as the base to calculate its speedup over other trainers, which is the ratio of the training time to reach the same convergence level. HarpLDA+ is more than 6x faster than LightLDA, 2x faster than NomadLDA and about 50% slower when timer control is not used. *Load Balance of Computation* and *Overhead* in distributed mode are similar to those in the multi-threading mode, here, the vectors of the average computation

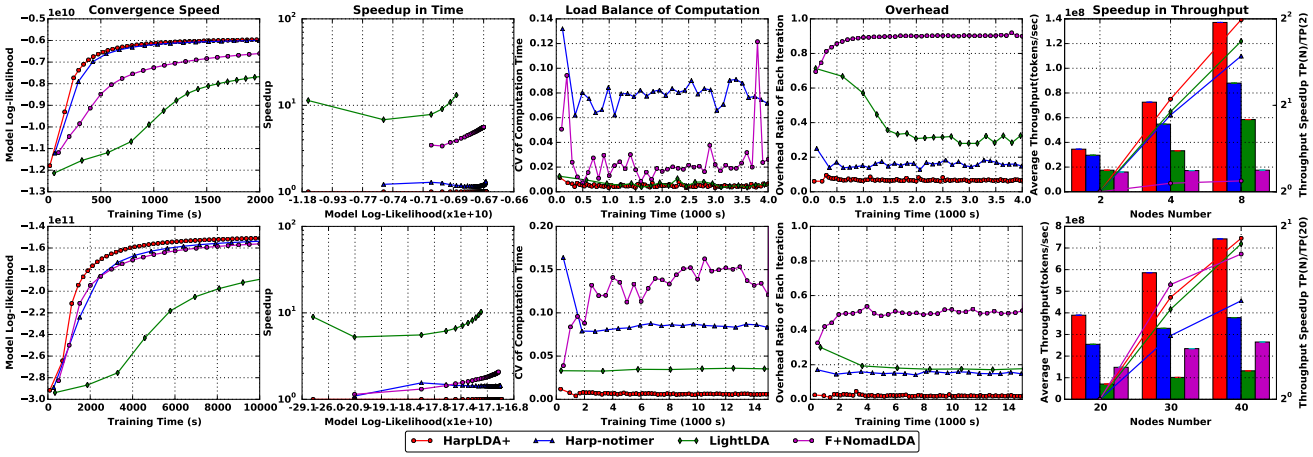


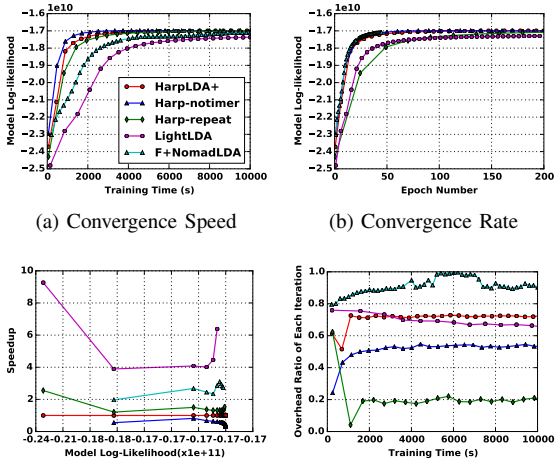
Figure 5: Distributed Performance. enwiki with [2,4,8]x16 (K=10K, 1st row), clueweb30b with [20,30,40]x16 (K=5K, 2nd row)(nomadlda fails due to out-of-memory problems in 10K experiments). The first four columns are the results of 8x16 and 40x16 respectively.

time and overhead time of all the threads on each node are used. harp-timer demonstrates significant differences from harp-notimer under these two metrics, which is the factor behind the boost in performance.

LightLDA, as an asynchronous approach, has less problems of load imbalance than the synchronized approaches. A general random partition works well in normal cases. The default staleness is set to one which can tolerate any performance undulations only if the lag of the local model replica is less than two iterations. This mechanism is effective in order to provide stability and good scalability for different cluster configurations. This is showcased in the scalability column. On the other hand, LightLDA has a lower convergence rate root from its asynchronous design and is less efficient during the multi-threading parallel implementation.

NomadLDA is observed to have the most load imbalance problems and also upholds a very high overhead ratio. In the ewniki 10K experiment, the overhead even reaches 90%, i.e., most of the workers are waiting for data. When using a scheduler approach and running directly on the InifiBand network for our experiments, this result is not expected. One possible reason is the task granularity. It takes very small granularity to schedule on each column of the word-topic model, which seems to have a large overhead, especially when K increases to a large number.

4) Communication Intensive Case Study: The following experiment runs on the bigram data-set, which has a $20M$ vocabulary size that is used to test the special communication intensive case in the distributed mode. We set the parameter of the bound in HarpLDA+ to [150%, 350%] to overlap the communication time in this special case, i.e., the dynamic scheduler keeps assigning those sampled blocks to free threads until the timer timeout. This trainer is named Harp-repeat.



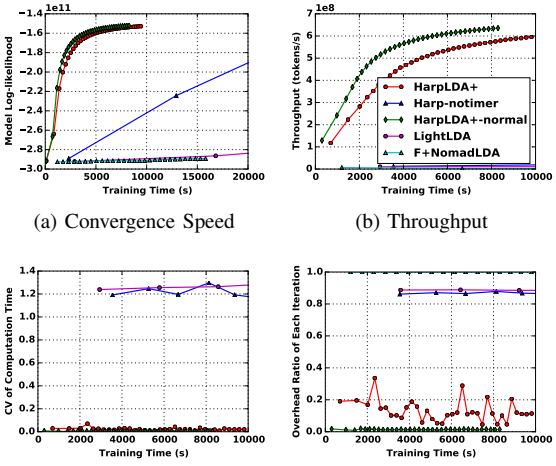
(a) Convergence Speed (b) Convergence Rate

(c) Speedup in Time (d) Overhead

Figure 6: Distributed Performance on bigram with 10x8 K=500

As in Fig. 6d, when the communication time dominates in the training process, all the trainers have a large overhead ratio. In LightLDA, as SSP forces the workers to keep the staleness of the local model within a range, the overhead of the wait time problem comes back. Harp-repeat significantly decreases the overhead and increases the throughput, but at the same time, the effectiveness drops, as in 6b, to be worse than LightLDA. Hence, harp-repeat does not gain in the final overall performance. In contrast, harp-notimer retains the best overall performance, thus, further optimizations should focus on how to decrease the size of the model that needs to be exchanged.

5) Straggler Case Study: The notion of a straggler is a normal situation in cloud computations, where some nodes are significantly slower than others in a job for many different reasons. In our experiment HPC cluster, we also encounter the stragglers more often than expected. In Fig.



(a) Convergence Speed (b) Throughput

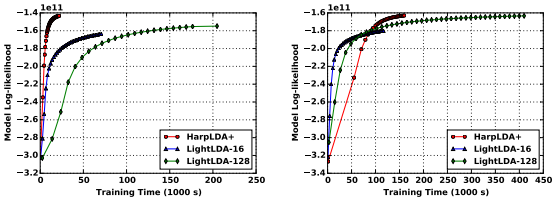
(c) LoadBalance (d) Overhead

Figure 7: Straggler Test on clueweb30b with 40x16 K=5K

7. All the trainers except harp-timer are severely affected by the emergence of a straggler. When, the CV value increases up to around 1.0, the overhead ratio increases more than 80%, throughput drops sharply, and as a result the overall performance drops sharply. For instance, the task does not even converge in 60,000(s) time where a normal run needs about 10,000(s). LightLDA benefits by its SSP design to represent a stable and scaleable trainer in a cluster with minor variances. However, it cannot endure in the case of large variances such as the straggler, in this case it stalls. In contrast, NomadLDA has a load balance scheduler which is designed to deal with these kinds of situations. When some nodes are detected to be slow and the number of tasks in its task queue is too large, the scheduler will decrease the probability of the new model to be sent to it. The design should work better than what this experiment shows. harp-timer shows a robust performance in the case of the straggler. The speedup on the convergence speed of a normal run is about 1.25, which means it lost about 25% performance when a straggler was encountered.

6) Large Model Case Study: Finally, we test the trainers on very large models, with K set to 100K and 1M respectively. NomadLDA fails in such settings with out-of-memory errors.

In Fig. 8b, when K increases to $1M$, LightLDA runs much faster than HarpLDA+ due to time complexity of $\mathcal{O}(1)$ in the MH sampling algorithm. However this only happens at the beginning of the training phase, and then it slows down and is surpassed by HarpLDA+ because



(a) K=100K (b) K=1M

Figure 8: Big Model Convergence Speed clueweb30b with 30x30

of its ineffectiveness of computation, despite using a very large MH step parameter such as 128 in Fig. 8b. In this big model experiment, HarpLDA+ demonstrates impressive performance, given that its time complexity is $\mathcal{O}(K_d + K_w)$ and LightLDA is $\mathcal{O}(1)$.

VI. Conclusions and Future Work

In this paper, we investigate the system design of large scale LDA trainers with a focus on parallel efficiency. We have identified four general design patterns from investigating the state-of-the-art systems. They are Locking, Allreduce, Asynchronous, and Rotation. Based on these, we introduce HarpLDA+, selecting the Rotation pattern and proposing a new synchronized LDA training system with timer control. This entails a two level parallelism design, in which a dynamic scheduler is used for multi-threading, while model rotation with timer control is used for distributed parallelism. Through extensive experiments, we demonstrate that the HarpLDA+ outperforms the other state-of-the-art LDA trainers surveyed in this paper. The timer control minimizes synchronization and communication overhead in HarpLDA+ and improves the performance by 50%.

From HarpLDA+, we've gained useful insights in designing a large scale Machine Learning system.

- Optimization of a sequential algorithm is critical but does not necessarily lead to high performance parallel systems. The trade-offs between effectiveness and efficiency are the key factors in optimizing a distributed Machine Learning system.
- The choices of data structures and parallel system design are critical for good performance in our Java HarpLDA+. Implementation details including programming languages and high performance off-the-shelf communication libraries do not always guarantee good performance as shown in Table III, Figures 3 and 5.
- Asynchronous parallel designs are favorable for scalability and robustness. However, with increasing parallelism and computation capacity provided by manycore and GPU servers, synchronized parallel designs can achieve better performance on a moderate sized cluster for big data problems in Table II.

Incorporating more parallelism, such as vectorization, into LDA trainers can be further explored in future work. Also, the similar characteristics of two large families of Machine Learning algorithms CGS (MCMC algorithm) and SGD (Numerical Optimization algorithm) are interesting topics for building a learning system.

VII. Acknowledgments

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

References

[1] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent dirichlet allocation," *J. Mach. Learn. Res.*, vol. 3, pp. 993–1022, 2003.

[2] T. L. Griffiths and M. Steyvers, "Finding scientific topics," *Proceedings of the National academy of Sciences of the United States of America*, vol. 101, no. Suppl 1, pp. 5228–5235, 2004.

[3] D. Newman, A. Asuncion, P. Smyth, and M. Welling, "Distributed Algorithms for Topic Models," *J. Mach. Learn. Res.*, vol. 10, pp. 1801–1828, Dec. 2009.

[4] Y. Wang, H. Bai, M. Stanton, W.-Y. Chen, and E. Y. Chang, "Plda: Parallel latent dirichlet allocation for large-scale applications," in *Algorithmic Aspects in Information and Management*. Springer, 2009, pp. 301–314.

[5] P. Smyth, M. Welling, and A. U. Asuncion, "Asynchronous distributed learning of topic models," in *Advances in Neural Information Processing Systems*, 2009, pp. 81–88.

[6] A. Smola and S. Narayanamurthy, "An architecture for parallel topic models," *Proc. VLDB Endow.*, vol. 3, no. 1-2, pp. 703–710, Sep. 2010.

[7] A. Ahmed, M. Aly, J. Gonzalez, S. Narayanamurthy, and A. J. Smola, "Scalable inference in latent variable models," in *Proceedings of the fifth ACM international conference on Web search and data mining*. ACM, 2012, pp. 123–132.

[8] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su, "Scaling distributed machine learning with the parameter server," in *Operating Systems Design and Implementation (OSDI)*, 2014.

[9] Q. Ho, J. Cipar, H. Cui, S. Lee, J. K. Kim, P. B. Gibbons, G. A. Gibson, G. Ganger, and E. P. Xing, "More effective distributed ml via a stale synchronous parallel parameter server," in *Advances in neural information processing systems*, pp. 1223–1231.

[10] F. Yan, N. Xu, and Y. Qi, "Parallel inference for latent dirichlet allocation on graphics processing units," in *Advances in Neural Information Processing Systems*, 2009, pp. 2134–2142.

[11] S. Lee, J. K. Kim, X. Zheng, Q. Ho, G. A. Gibson, and E. P. Xing, "On model parallelization and scheduling strategies for distributed machine learning," in *Advances in Neural Information Processing Systems*, 2014, pp. 2834–2842.

[12] J. K. Kim, Q. Ho, S. Lee, X. Zheng, W. Dai, G. A. Gibson, and E. P. Xing, "STRADS: a distributed framework for scheduled model parallel machine learning," in *Proceedings of the Eleventh European Conference on Computer Systems*. ACM, 2016, p. 5.

[13] B. Zhang, B. Peng, and J. Qiu, "High performance lda through collective model communication optimization," *Procedia Computer Science*, vol. 80, pp. 86–97, 2016.

[14] H.-F. Yu, C.-J. Hsieh, H. Yun, S. V. N. Vishwanathan, and I. S. Dhillon, "A scalable asynchronous distributed algorithm for topic modeling," in *Proceedings of the 24th International Conference on World Wide Web*. ACM, 2015, pp. 1340–1350.

[15] H. Yun, H.-F. Yu, C.-J. Hsieh, S. V. N. Vishwanathan, and I. Dhillon, "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.

[16] L. Yao, D. Mimno, and A. McCallum, "Efficient methods for topic model inference on streaming document collections," in *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, ser. KDD '09. ACM, pp. 937–946.

[17] A. Q. Li, A. Ahmed, S. Ravi, and A. J. Smola, "Reducing the sampling complexity of topic models," in *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2014, pp. 891–900.

[18] J. Yuan, F. Gao, Q. Ho, W. Dai, J. Wei, X. Zheng, E. P. Xing, T.-Y. Liu, and W.-Y. Ma, "LightLDA: Big Topic Models on Modest Compute Clusters," *arXiv preprint arXiv:1412.1576*, 2014.

[19] J. Chen, K. Li, J. Zhu, and W. Chen, "WarpLDA: A Cache Efficient O(1) Algorithm for Latent Dirichlet Allocation," *Proc. VLDB Endow.*, vol. 9, no. 10, pp. 744–755, Jun. 2016.

[20] W.-S. Chin, Y. Zhuang, Y.-C. Juan, and C.-J. Lin, "A Fast Parallel Stochastic Gradient Method for Matrix Factorization in Shared Memory Systems," *ACM Transactions on Intelligent Systems and Technology (TIST)*, vol. 6, no. 1, p. 2, 2015.

[21] B. D. He, C. M. De Sa, I. Mitliagkas, and C. R, "Scan Order in Gibbs Sampling: Models in Which it Matters and Bounds on How Much," in *Advances In Neural Information Processing Systems*, 2016, pp. 1–9.