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

Latent Dirichlet Allocation (LDA) is a widely used machine learning technique in topic modeling and data analysis. Training large LDA models on big datasets involves dynamic and irregular computation and is a major challenge for both algorithm optimization and system design. In this paper, we focus on the study of parallel efficiency and conduct a comprehensive analysis of state-of-the-art LDA training systems. They are MPI/C++ based implementations of LightLDA, F+NomadLDA, WarpLDA and a Hadoop/Java based implementation of HarpLDA+. We show that our proposed HarpLDA+ design achieves the best performance, owing this to synchronized communication with a timer control for load balance and scaling. The system optimization in HarpLDA+ effectively reduces the overhead of synchronization and communication, and therefore it outperforms other approaches.

1. INTRODUCTION

Latent Dirichlet Allocation (LDA) [2] is a standard topic modeling technique of data analysis. 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 optimization and implementation aspects impact the performance of LDA training systems. In this paper, we select four LDA trainers with different time complexities of sampling. Note that a LDA algorithm designed with lower time complexity does not necessarily scale well to high data volume and cluster size. The overarching question is how to deploy a sequential algorithm so that it can scale to large problems.

LDA is a computation that is irregular and the model size can be huge which changes over iterations during convergence. Meanwhile, parallel workers need to synchronize the model continually. To investigate the trade-offs between synchronized and asynchronous algorithms, we explore the system design space for LDA trainers. Instead of focusing only on accuracy and execution time, we optimize based on parallel efficiency and analyze global model partitioning, consistency and rate of convergence. We further propose a new mechanism based on a timer control to demonstrate the idea and implement it with Java HarpLDA+. 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, which is a less explored direction on this topic in the literature.
- Propose a new mechanism using Timer Control to reduce communication overhead and wait time in a synchronized system.
- Implement HarpLDA+ based on Hadoop and demonstrate excellent performance and scalability.
- Summarize our system design approach and applicability for other machine learning algorithms.

The outline of this paper is as follows: Section 2 introduces the background of the LDA algorithm and related work, while Section 3 analyzes the architecture and parallel efficiency of existing solutions. Section 4 describes our system design and implementation details of HarpLDA+ and Section 5 presents experimental results coupled with a performance analysis. Finally, Section 6 draws conclusions and discusses future work.

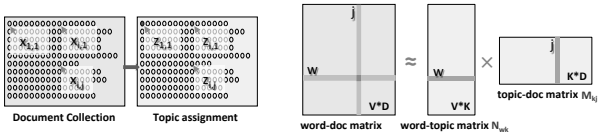


Figure 1: Latent Dirichlet Allocation

2. RELATED WORK

2.1 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

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/4.0/. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org.
Proceedings of the VLDB Endowment, Vol. 10, No. 10
Copyright 2017 VLDB Endowment 2150-8097/17/06.

models each document as a mixture of latent topics and each topic as a multinomial distribution over words. As seen in Fig. 1, LDA can be viewed as a sparse matrix decomposition technique on a normalized word-document matrix. But it is rooted in a probabilistic foundation and has quite different computation characteristics.

LDA follows a Bayesian approach to predict values with the full posterior distribution of the latent parameters. Many algorithms have been proposed to estimate the parameters for the LDA model. In the original paper, Blei used an EM-like algorithm to estimate the parameters directly by a lower bound decomposition. Later, Griffiths proposed CGS (Collapsed Gibbs Sampling) [5], a Markov chain Monte Carlo (MCMC) algorithm to solve this problem.

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. It is beneficial to finally be able to reduce the standard LDA model into a collapsed version of posterior distribution, in which only the topic assignment variables Z need to be visited.

In CGS, instead of keeping the dense matrices ϕ and θ, we only keep the sufficient statistics, i.e. the occurrence count matrices of word-topic (N_{wk}) and topic-doc (M_{kj}), which are sparse and contain integer values. Although CGS generally requires a large number of iterations to converge, it is memory efficient and therefore salable for large models. This is a major reason why the MCMC approach is widely adopted for large scale LDA training. In this paper, we focus on the LDA trainers using the CGS algorithm.

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. Hyperparameters α and β control the topic density in the final model as output. The model gradually converges during the process of iterative sampling.

2.2 Related Work on Parallel LDA-CGS

Typically, Gibbs sampling in LDA-CGS is stated as a sequential process. AD-LDA (Approximate Distributed LDA)[12] 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 [16], implemented the AD-LDA algorithm in both MPI and MapReduce, where the Allreduce operation is used for synchronization.

Algorithm 1: LDA Collapsed Gibbs Sampling Algorithm

input : training data X, the number of topics K, hyperparamters α, β
output: topic assignment matrix Z_{ij}, topic-document matrix M_{kj}, word-topic matrix N_{wk}

```
Initialize M_kj, N_wk to zeros // Initialize phase
foreach document j ∈ [1, D] do
    foreach token position i in document j do
        z_{i,j} = k ~ Mult(1/K) // sample topics by multinomial distribution
        m_{k,j} += 1; n_{w,k} += 1 // token x_{i,j} is word w, update the model matrices
// Burn-in and Stationary phase
repeat
    foreach document j ∈ [1, D] do
        foreach token position i in document j do
            m_{k,j} -= 1; n_{w,k} -= 1 // decrease counts
            z_{i,j} = k' ~ p(z_{i,j} = k|rest) // sample a new topic by Equation (1)
            m_{k',j} += 1; n_{w,k'} += 1 // increase counts for the new topic
until convergence;
```

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 [15] extended AD-LDA to an asynchronous solution by a gossip protocol. [14][1] 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, [11] introduced Parameter Server as a general framework that scaled to thousands of servers. Another progression was presented by [7]. It proposed a "mixed" approach SSP (Stale Synchronous Parallel), which is a parameter server that can limit the maximum age of the staleness.

Other researchers have investigated synchronized algorithms. For instance, [17] 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, [9][8] 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, [22] HarpLDA adopted a synchronized design and proposed collective communication operator with a rotation pipeline which achieved better performance. Finally, [19] introduced F+Nomad-LDA based on idea of NOMAD[21], in which each variable (one column of the model matrix) 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. As shown in Algorithm 1, sampling a topic according to the distribution of equation (1) is the kernel com-

putation of CGS. 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 is $\mathcal{O}(K)$. In equation (1), the probability calculation $p(z_{ij} = k)$ mainly relies on the elements in the model matrices M_{kj} and N_{wk}. The sampling process should access one column in M_{kj} and one row in N_{wk} when calculating all K probabilities. Whether sampling in the order of document, i.e., visiting z_{ij} by row in Z, or sampling in the order of the word, i.e., by column, there is always one of the model matrices M or N which has to be randomly accessed.

Recently, many novel ideas have been proposed to optimize the sampling process. [18] SparseLDA decomposed equation (1) into three parts:

$$p\left(z_{ij} = k \mid rest\right) \propto \frac{N_{wk}^{\neg ij}(M_{kj}^{\neg ij} + \alpha) + \beta * M_{kj}^{\neg ij} + \alpha\beta}{\sum_w N_{wk}^{\neg ij} + V\beta} \quad (2)$$

The denominator remains constant when sampling on one word. Similarly, the third part of the numerator is also a constant; while the second part is non-zero only when M_{kj} is non-zero, and the first 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 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 [19] 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.

In fact, an algorithm called Alias Table allows us to draw subsequent samples from the same distribution in $\mathcal{O}(1)$ time. However, it cannot apply directly to the CGS sampling process since the distribution changes after each update. [10] solved this dilemma by using another sampling algorithm called Metropolis Hasting (MH) to draw each sample correctly from the stale alias table. Because the acceptance rate of MH is very high, and the $\mathcal{O}(K)$ cost of building the alias table can be amortized, it finally finds a $\mathcal{O}(1)$ algorithm for sampling. Alias-LDA uses it as part of equation (1) to achieve $\mathcal{O}(K_d)$ complexity. [20] 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. Also LightLDA is built on a SSP parameter server framework that implements a memory efficient system for very large models. [3] 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. The delay of updates enable reordering of the memory access, which changes from random access to a sequential scan. The intra-node performance of WarpLDA is impressive, but it relies on accessing the matrix Z by row and by column alternatively in one iteration, which leads to a huge data exchange in the distributed mode.

3. ARCHITECTURE AND PARALLEL EFFICIENCY

3.1 Parallel Efficiency

Parallelizing a sequential algorithm inevitably introduces overhead. AD-LDA [13] provides a basic analysis on the parallel efficiency of the proposed parallel CGS algorithm. With N training data points, K topics and P parallel workers, the time complexity for one epoch of naive CGS is $\mathcal{O}(NK)$. P parallel samplers can improve it to $\mathcal{O}(\frac{1}{P}NK + KW + C)$, in which C is the time involving the global sum of the count difference and KW is the time spent on the communication of the distributed model N_{wk}. The overhead comes from the *synchronization* operation.

Because of this overhead, adding more parallel workers does not necessarily lead to better performance. *Amdahls law* states that if a portion of a computation f can be parallelized with P workers, the other serial portion that cannot be improved, namely $1 - f$, will quickly dominate the performance. *Speedup* is defined as original sequential performance over parallel performance in parallel processing. The parallel processing speedup in time implied by Amdahls law is:

$$Speedup = \frac{T_{original}}{T_{enhanced}} = \frac{1}{1 - f + \frac{f}{P}} \quad (3)$$

Parallel efficiency is parallel speedup divided by the parallelism.

$$Parallel\ \ Efficiency = \frac{Speedup}{P} \quad (4)$$

The concept of parallel efficiency gives us an analysis tool to investigate the design of the parallel LDA trainers. *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. But recent architectures of manycore machines adopts NUMA (Non-Uniform Memory Access), which links several small nodes with a high-performance connection but still provides a shared memory programming model. Communication overhead across NUMA nodes cannot be overlooked, similar to the situation in distributed systems. The notion of overlapping communication with computation is a key design choice for high performance 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 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 workloads lead some workers waiting for the other busy workers in 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 [22][3] 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 same number of tokens with different distributions. In practice, variations

Table 1: Taxonomy of LDA Trainer Design

Trainer	Sampler	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

of node performance and stragglers are not uncommon even in homogeneous HPC clusters.

3.2 System Architecture

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 end of each epoch is an optimization method when the deviations are small.

The decision of whether to use stale values of N_{wk} and M_{kj} in the numerator of equation (1) leads to different solutions. It is also worth noting that performance is more sensitive on these two matrices than on the sum of topic count. Furthermore, the question of how the models are distributed, how they are updated and synchronized, is important when investigating solutions. The communication method to implement synchronization, either Asynchronous or Collective Communication, is also important as it creates further differences amongst the designs.

In Fig. 2, we summarize four parallel design patterns of current CGS trainers in this Model-Centric view:

- **Lock Pattern**. Using locks to maintain the consistency is a straightforward solution to parallelize an algorithm, but it is only feasible in a shared memory setting as the overhead caused by conflicts of locks may become very large.
- **AllReduce Pattern**. Working on a stale model and using collective communication to do synchronization on distributed model replicas is the first practical parallel solution. For instance, PLDA utilizes a MPI Allreduce operation to implement the system, which is easy to use, but the synchronization overhead for such operations is considerably big in case of large scales. Furthermore, an imbalanced workload introduces a large portion of wait time during this operation. On the other hand, this operation may have a large burden on memory if not optimized for it, especially if there is a need to fit all the model in one worker's main memory.
- **Asynchronous Pattern**. Asynchronous Pattern is a typical design to relax effectiveness for efficiency. It works on local replicas and synchronizes them through a group of parameter servers in a best effort rather than synchronizing with all other workers globally to keep models consistent. Hence, the algorithm works in an asynchronous fashion. Yahoo!LDA and LightLDA are in this category. Parameter servers remove wait time overhead (in the SSP version, wait time can exist), and will naturally use asynchronous communication to overlap the communication with computation in order to avoid the overhead of communication. Moreover, Asynchronous Pattern is a popular framework for large scale machine learning algorithms.
- **Rotation Pattern** Avoiding update conflicts is the central idea behind this category. When models are distributed into partitions, a scheduler is used to arrange exchanging the partitions among the workers while at the same time keeping model updates conflict free. The model partitions should 'rotate' among the workers, where the name Rotation Pattern is given. StradsLDA, F+NomadLDA and HarpLDA are in this category. An asynchronous communication mechanism is also preferred since it overlaps the communication time easily. The scheduler in F+Nomad-LDA works in a decentralized way that has no explicit global synchronization point, by which it wants to remove the wait time overhead. Strads-LDA is also in this category, but it has a global synchronization point and has performance issues on the wait time overhead. This approach can also be implemented by collective com-

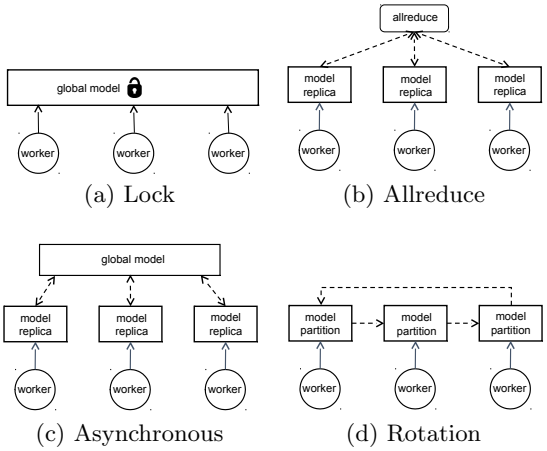


Figure 2: System Design Patterns

munication. HarpLDA adopts a collective communication programming model with a new operation called 'rotate'. Furthermore, pipelining is used to overlap communication time with computation, but the wait time overhead is still an issue. WarpLDA is special as it is able to avoid update conflicts with a full updates delay. By delaying all update operations at the end of an epoch, it decouples not only the read and write, but also the accessing order of the two matrices. Random memory access is reordered into a sequential scan on the two matrices alternatively. This can be viewed as a kind of rotation approach, and at the same time, it uses stale models on sampling calculations.

For a concrete trainer implementation, two levels of parallelism of intra-node multi-threading, and inter-node distributed design can adopt different kind of design patterns. See Table 1.

4. HARPLDA+: DESIGN AND IMPLEMENTATION

HarpLDA+ is an improved design of our previous work called HarpLDA [22]. It builds upon Harp[1], which is a Java collective communication library released as a plugin for Hadoop. Harp aims to merge HPC techniques into the Big Data software stack. In HarpLDA, we focus on communication optimization and propose new collective communication operators for LDA trainer design. In this paper, we focus on reducing the synchronization overhead.

4.1 Programming Model Based on Collective Communication

Using collective communication within a scheduler 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 2)

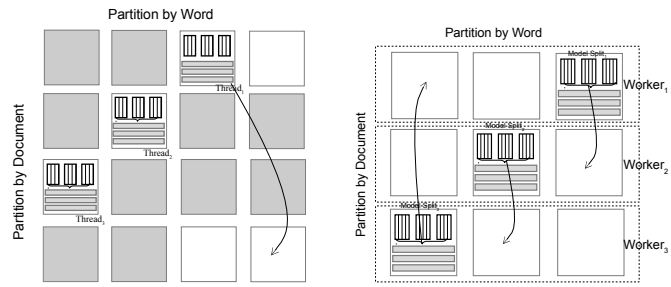
Algorithm 2: HarpLDA+ Parallel Pseudo Code

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

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

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. 2d). 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.

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



(a) dynamic scheduling (b) model rotation

Figure 3: Scheduling in Shared Memory and Distributed Systems

Algorithm 2 presents a general framework for scheduling, where multi-threading and distributed parallelism can adopt the same procedure. We can however improve it to reduce synchronization overhead, leveraging the computation characteristics in these two different environments.

4.2 Dynamic Scheduling in Shared Memory Systems

In a shared memory system with the assumption of uniform memory access, we can ignore the cost of data movement. Specifically, there is no communication cost in considering the system design, which makes scheduling a much easier task to accomplish.

Dynamic scheduling provides a low cost solution to remove synchronization overhead. 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 a general and effective solution, which also occurs in parallel matrix factorization design [4].

As in Fig. 3a, 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, two vectors

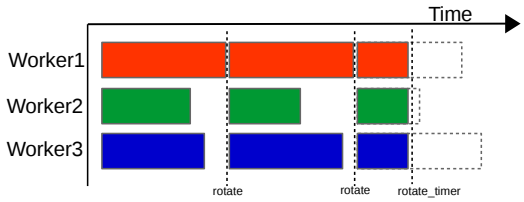
of L maintain the occupy status of each row and column, while another $L \times L$ matrix maintains a two level status: free, or finished (multiple levels can be extended to support repeat computing on each block and priority of each block). The scheduler can randomly select a free block by scanning 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.

In distributed systems, the cost of data movement cannot be omitted, thus the dynamic scheduling approach does not work anymore. As in Fig. 3b, 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 2 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.

4.3 Timer Control in Distributed Systems

Another overhead of a rotate call is the time to wait for all workers to finish their computation, which is the actual synchronization overhead. Due to load imbalance, the slowest worker will force all other workers to wait for it until it finishes its computation. Its hard to make the sampling work in a load balanced way for each parallel worker by data partitioning only, because even with the same size of data points, the samplers' execution times may vary. The wait time caused by workload imbalance cannot be reduced by pipelining either.

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, . . . 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 $1, 2, ..., d$. [6] 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 by using blocks that fit in cache. The dynamic scheduler picks a block uniformly from the free block list. While inside the block, we still keep the word order during sampling. Originally, we intended to investigate the performance of different sampling orders. But we found our block based approach provides a simple solution for load balancing.



(a) Timer Control

Figure 4: Timer to Control the Synchronization Point

The overhead comes from the wait time between the computation finishing and the start of synchronization, also known as the time point call rotate (see Fig. 4). If we adjust the synchronization point ahead of the finish point, we can remove this gap. Under the deterministic scan order, this kind of adjustment is a little bit difficult 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 in Fig. 4 demonstrates this mechanism.

Thus we propose a simple solution for LDA-CGS trainer, where 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 wait until finish all the blocks.

During the process of convergence, the model size shrinks and the computation time drops. In HarpLDA+, we design an auto-tuning mechanism to set the value of the timer for each iteration. First, the timer works best when the communication can be fully overlapped by computation, which means that the computation time or the number of the training data points being processed should have a lower bound L. Secondly, we need to make certain that all the workers stop at the same time before any of them finish. 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 cut in half.

4.4 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 stored in primitive arrays and

sorted from high to low to reduce the linear search time of sampling on a probability distribution. 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.

5. EXPERIMENTS

5.1 Setup of Experiments

Table 2: Datasets for LDA Training, where *DocLen* represents mean and std. dev. values of document length

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

Five datasets (see Table. 2) 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.

Table 3: Trainers for LDA Training

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

We select four state-of-the-art CGS trainers for comparison in Table. 3. They represent different system designs of Section 3.2. StradsLDA and HarpLDA are not selected because they have a similar distributed design and inferior performance. Through well-designed control experiments, we were able to investigate the effectiveness of various system designs and draw solid conclusions. To evaluate the performance of LDA implementations, we use the following metrics. Firstly, we choose *Model log likelihood* of the word-topic model to represent the status of convergence and the quality of a model, in which a higher likelihood value indicates a higher convergence level and a better model quality. Secondly, we select three main evaluation metrics as follows: 1) *Convergence speed* is the standard metric to evaluate a trainer's performance, which depicts the relation between convergence level and training time. 2) *Convergence rate* evaluates the effectiveness of the algorithm by depicting the relationship between convergence level and model update count. 3) *Throughput* evaluates the efficiency in a system perspective view by measuring the model update counts per second. 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 NomadLDA and mpich2 3.0.4 for LightLDA. We set the hyper-parameters $\alpha = 50/K$ and $\beta = 0.01$ in all the experiments.

5.2 Experimental Results

5.2.1 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. 5, each row contains the results of a specific dataset and topic number K, and column 1 to column 3 are performance results of single a thread on the three datasets. Nytimes and pubmed2m are relatively small while enwiki is of medium size. The following conclusions can be drawn from the experiments:

- **Convergence Rate** represents the effectiveness of model updates. Standard SparseLDA sampler, NomadLDA, is always the fastest. HarpLDA+ is a bit slower due to the caching of the model for identical words. In the experimental comparisons without timer control, the timer itself does not make an observable difference in convergence rate. 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 than the other trainers. The order of convergence rate is constantly stable in parallel versions of these trainers. Also, due to the space limitation, this metric is not listed in future experimental results.

- **Throughput** represents the efficiency of model updates. 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. Also these figures show the trend of throughput for CGS training, which can be slow in the beginning when the models are randomly initialized to be large, and gets faster during the process of convergence when models shrink. Finally, the throughput rate reaches stability when the algorithm convergences. For systems with the $\mathcal{O}(1)$ MH sampling algorithm, the throughput is less sensitive to these changes.

- **Convergence Speed** represents the overall performance resulting from the combination of efficiency and effectiveness of model updates. 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 as is specially designed for very large datasets and limited memory settings. The algorithm splits the training data and model into slices and swaps them between memory and disk by I/O pipelines. The parameter

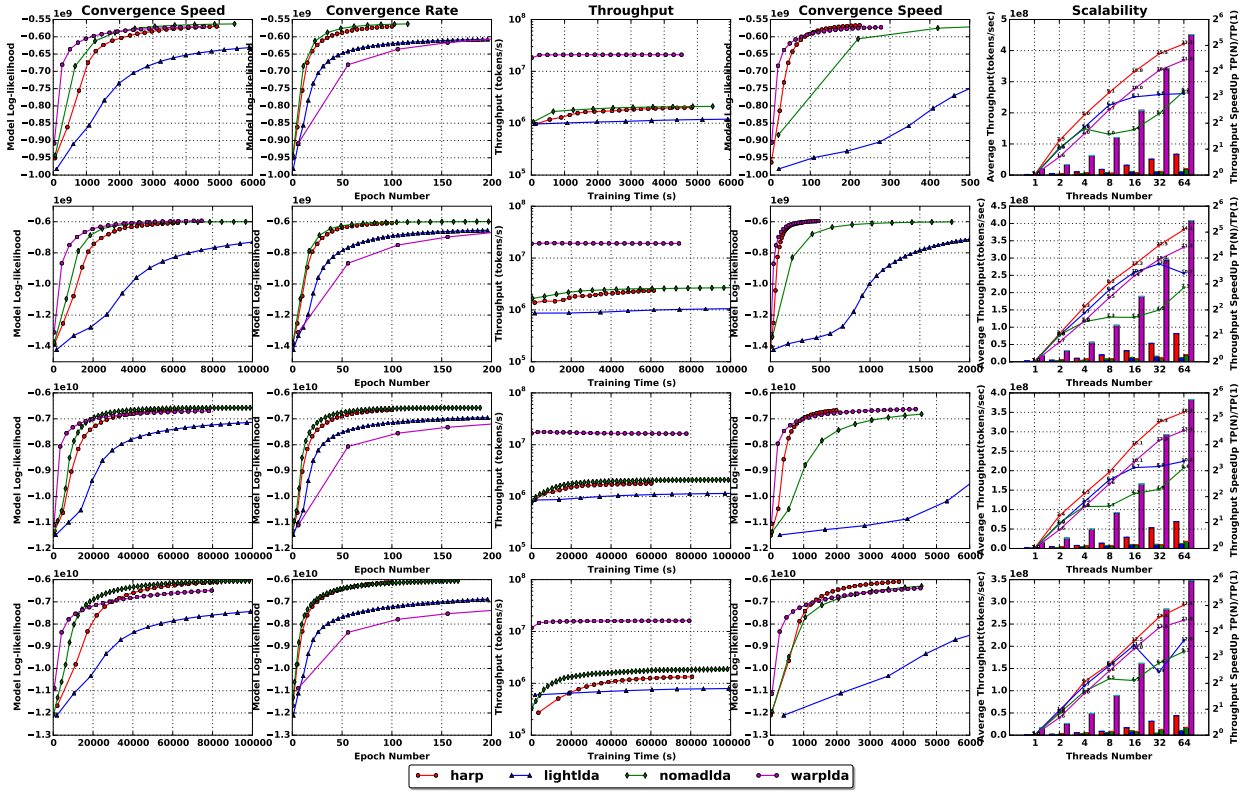


Figure 5: Single Node Performance on nytimes (K=1K, 1st row), pubmed2m (K=1K, 2nd row), enwiki (K=1K, 3rd row) and enwiki (K=10K, 4th row). Column 1 to 3 are results of single thread, column 4 is result of 32 threads.

server architecture also introduces inter-process communication. In single thread experiments, we tune the slice number to be at least two in order to make the I/O pipeline effective. This method works well but the overall performance of LightLDA is still not comparative on this setting.

5.2.2 *Intra-node Parallel Efficiency*

We test on increasing number of threads and its impact on performance as seen in Fig. 5. 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 which case the effectiveness of WarpLDA gets worse. Fig. 5 column 5 *Scalability* represents the trend of efficiency under a multithreading environment. The bar chart shows the average throughput increasing along with the parallelism, and the SpeedUp of throughput shows the parallel efficiency of these increases. HarpLDA+ demonstrates the best scalability which explains the boost of its performance from a single thread to a large number of threads.

In this subsection, we investigate the details of multi-threading parallelism in an experiment setting: Dataset = enwiki, $K = 1K$, Node Number = 1, and Thread Number = 32, which is 4 less than the physical core number to guarantee a setting of the largest number of threads with no computation resources contention. For the experiments, an advanced profiling tool is utilized to exhibit the time breakdown. Through Concurrency Analysis by VTune Amplifier[2], five categories of time spent in the application can be given. *Elapsed time* is the total running time, *CPU Time* is time during which the CPU is actively executing the application, *Wait Time* occurs when software threads are waiting due to APIs that block or cause synchronization. CPU Time includes three parts: *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, such as system synchronization APIs, Intel TBB, and OpenMP.

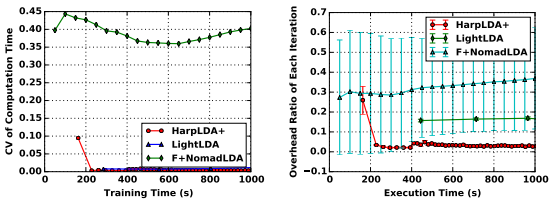
Table 4: Time Breakdown by VTune Concurrency Analysis

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

After normalization on the Elapsed Time, we get the experiment results in Table. 4. 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 mainly comes from the default static scheduler in OpenMP. NomadLDA has zero wait time, but this

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

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[3]. 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. We also find that it is critical for parallel efficiency to be careful on the serial code introduced by optimizations. E.g., sorting the model is an optimization to make sampling on large K faster and implemented in the main thread. When the sort is occurring, the wait time ratio increases to 20% in this experiment. Only when deploying the trainer on a large number of nodes, can the overhead can be amortized in order to make this optimization really useful.



(a) Load Balance (b) Overhead Ratio

Figure 6: Load Balance and Overhead Ratio

Within another experiment, we poll these trainers with thread level logs to record the actual sampling time in each iteration. VTune profiling is favorable, but it still has some limitations. For instance, it can give details of spin time and wait time for known libraries, but is limited for unknown ones, including Java. And even in the effective cases, further breakdown to get the actual working ratio for specific applications is beyond its capability. With the real working time of each thread, we can check the status of load balance among threads and also the overhead of parallelism including the non-parallel part of the code and synchronization overhead.

Thread level logs are not included in WarpLDA as it is implemented with multithreading by OpenMP. Large task granularity enables HarpLDA and LightLDA to add these logs easily, as synchronization occurs after the sampling on a block or a slice of data. F+NomadLDA has much smaller task granularity where each thread assigns the updated model to another thread after sampling on one word. In this case, using the high performance clock_gettime function is adopted to record the time, which makes the overhead tolerable.

CV (coefficient of variation) is a metric to evaluate load balance among the threads. On the vector of the sampling time of threads, it calculates the ratio of the standard deviation to the mean. See Fig. 6a, F+NomadLDA has a very large CV level depicting serious problems with load imbalance. Fig. 6b is an error bar chart elucidating the overhead time ratio for each iteration. Overhead time for each thread is the iteration time excluding the actual sampling time spent. 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.

[3]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.

5.2.3 Distributed Parallel Efficiency

In this section, we run LDA trainers in distributed mode to investigate their capability of scaling-out. 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 dataset in each iteration among all the workers. The data volume is at least 10 times larger when compared with the other systems. 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 so that each node trains on its own local model replica. When it scales to more nodes, some low frequency words may not appear in all the partitions of the training set, i.e., the vocabulary size of a local model replica will become smaller than that of the global model. According to the equation to calculate model likelihood, the value contributed by the 'missing' low frequency words is approximately constant. As LightLDA reports model likelihood on behalf of each local model, the actual likelihood of the global model can only be estimated by a re-calibration of its reported value with the constant gap. Two versions of HarpLDA+, harp-timer and harp-notimer, are included based on whether to use timer control or not.

We test different LDA trainers as seen in Fig. 7. The first experiment runs on enwiki with K=[1K,10K] and clueweb30b with K=5K (nomadlda fails due to out-of-memory problems in 10K experiments). Because LightLDA and NomadLDA do not have a good parallel efficiency under large thread number, we set it to 16 instead of the physical core number.

As for the chart, column one represents convergence speed, where harp-timer 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. Note that harp results on enwiki will be updated later. *Load Balance of Computation* and *Overhead* in distributed mode are similar to those in the multithreading mode, here, the vectors of the average computation time and overhead time of all the threads on each node are used. harp-timer demonstrates significant differ-

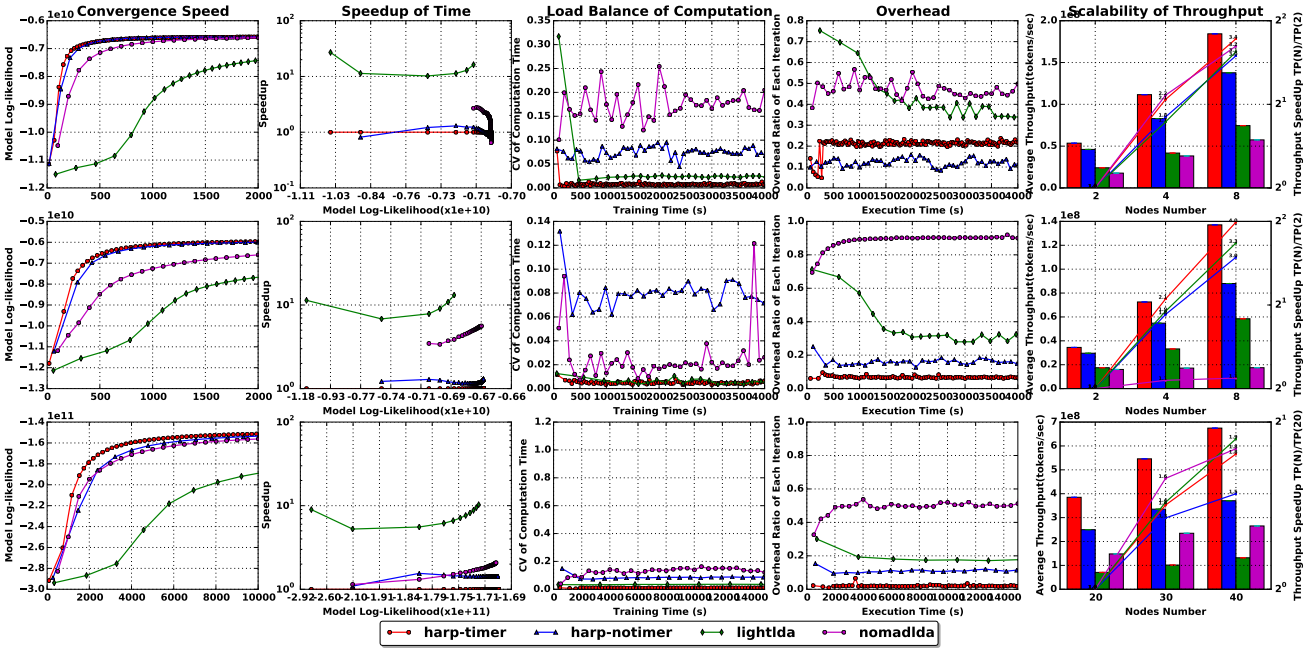


Figure 7: Distribute Performance. enwiki with [2,4,8]x16 (K=1K, 1st row, K=10K, 2nd row), clueweb30b with [20,30,40]x16 (K=5K, 3rd row). The first four columns are the results of 8x16 and 40x16 respectively.

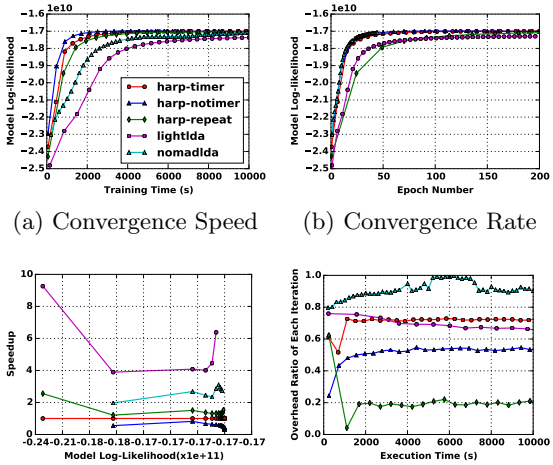
ences from harp-notimer under these two metrics, which is the factor behind the boost in performance.

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.

5.2.4 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. When the communication time dominates in the training process, all kinds of system designs have a large overhead time ratio. In scheduler approach systems, most of the time is spent on exchanging the model partitions. Moreover, in standard parameter server approach systems, the overhead can be small as the system continuously samples on the stale local model replicas. However, 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. The HarpLDA+ timer control with the default parameter of low bound 40%, and high bound 80% will also fail in this test because even the computation time subsuming 100% of the training points cannot overlap the large communication time. We extend the bound 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

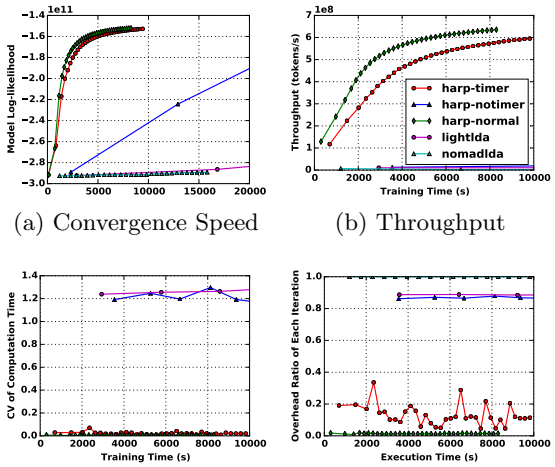
(c) Speedup in Time (d) Overhead

Figure 8: Distributed Performance on bigram with 10x8 K=500

As in Fig. 8d, all the trainers have a large overhead ratio. Harp-repeat significantly decreases the overhead and increases the throughput, but at the same time, the effectiveness drops, as in 8b, 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.

5.2.5 *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. Furthermore, by unknown reasons, the performance of some nodes deteriorate while the job is running. Their computation capacity sometimes drops to 10% of the normal capacity. We now show test results when stragglers are encountered. In Fig. 9. 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.



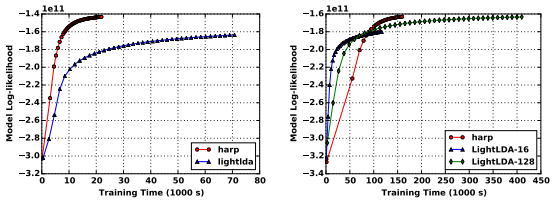
(a) Convergence Speed (b) Throughput

(c) LoadBalance (d) Overhead

Figure 9: Straggler Test on clueweb30b with 40x16 K=5K

5.2.6 *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.



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

Figure 10: Big Model Convergence Speed clueweb30b with 30x30

In Fig. 10b, 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 of its ineffectiveness of communication, despite using a very large MH step parameter such as 128 in Fig. 10b. In this big model experiment, HarpLDA+ demonstrates impressive performance, given that its time complexity is $\mathcal{O}(K_d + K_w)$ and LightLDA is $\mathcal{O}(1)$.

6. 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 model 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 out-of-the-shelf libraries do not always guarantee good performance as shown in Table 3, Figures 5 and 7.

- Asynchronous parallel designs are favorable for scalability and robustness. However, with increasing parallelism and computation capacity provided by many-core and GPU servers, synchronized parallel designs can achieve better performance on a moderate sized cluster for big data problems in Table 2.

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.

7. 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.

8. REFERENCES

[1] 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*, pages 123–132. ACM, 2012.

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

[3] J. Chen, K. Li, J. Zhu, and W. Chen. WarpLDA: A Cache Efficient O(1) Algorithm for Latent Dirichlet Allocation. *Proc. VLDB Endow.*, 9(10):744–755, June 2016.

[4] 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)*, 6(1):2, 2015.

[5] T. L. Griffiths and M. Steyvers. Finding scientific topics. *Proceedings of the National academy of Sciences of the United States of America*, 101(Suppl 1):5228–5235, 2004.

[6] 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*, pages 1–9, 2016.

[7] 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*, pages 1223–1231.

[8] 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*, page 5. ACM, 2016.

[9] 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*, pages 2834–2842, 2014.

[10] 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*, pages 891–900. ACM, 2014.

[11] 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.

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

[13] D. Newman, P. Smyth, M. Welling, and A. U. Asuncion. Distributed inference for latent dirichlet allocation. In *Advances in neural information processing systems*, pages 1081–1088, 2007.

[14] A. Smola and S. Narayanamurthy. An architecture for parallel topic models. *Proc. VLDB Endow.*, 3(1-2):703–710, Sept. 2010.

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

[16] 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*, pages 301–314. Springer, 2009.

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

[18] 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*, KDD '09, pages 937–946. ACM.

[19] 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*, pages 1340–1350. ACM, 2015.

[20] 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.

[21] 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*, 7(11):975–986, 2014.

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