High Performance LDA through Collective Model Data Communication Optimization

Bingjing Zhang[1], Bo Peng[1,2], and Judy Qiu[1]

[1] Indiana University, Bloomington, Indiana, USA
{zhangbj,pengb,xqiu}@indiana.edu
[2] Peking University, Beijing, China

Abstract
LDA is a widely used machine learning technique for big data analysis. The application includes an inference algorithm that iteratively updates on the model data until convergence. A major challenge is the scaling issue in parallelization owing to the fact that the size of model data is huge and parallel workers need to communicate with them continually. We identify three important features of the model data in parallel LDA computation: 1. The volume of model data required for local computation is high; 2. The time complexity of local computation is proportional to the size of required model data; 3. The size of model data size shrinks as it converges. By investigating collective and asynchronous communication methods of model data in different tools, we discovered that optimized collective communication can improve the model update speed, thus allowing the model to converge faster. The performance improvement derives not only from accelerated communication but also from the reduced iteration computation time as the model data size shrinks during the model convergence. To foster fast convergence, we design new collective communication abstractions, "lda-lgs" and lda-rtt", for model data communication, and implement Harp-LDA. We compare our new approach with Yahoo! LDA and Petuum LDA, two leading implementations favoring asynchronous communication methods in the field, on a 100-node, 4000-thread Intel Haswell cluster. The experiments show that "lda-lgs" can reach higher model likelihood with shorter or similar execution time compared with Yahoo! LDA, while "lda-rtt" can run up to 3.9 times faster compared with Petuum LDA when achieving similar model likelihood.

Keywords: Parallel Computing, LDA, Big Model Data, Communication Optimization

1 Introduction

Latent Dirichlet Allocation (LDA) [9] is an essential machine learning technique that has been widely used in areas such as text mining, advertising, recommender systems, network analysis, and genetics. Though extensive research on this topic exists, the data analysis community is still endeavoring to scale it to web-scale corpora to explore more subtle semantics with a big model which may contain billions of model parameters [12].

We identify that the size of model data required for the local computation is so large that sending such huge data to every worker results in communication bottlenecks. The computation also takes a long time for the large model data size. In addition, the model data size shrinks as the model converges. As a result, a faster model data communication method can speed up the model convergence, in which the model size shrinks and thereby the iteration execution time reduces.

By guaranteeing the algorithm correctness, various model data communication strategies are developed in parallel LDA. Existing solutions favor asynchronous communication methods, since it not only avoids global waiting, but also quickly makes the model update visible to other workers and thereby boosts the model convergence. We propose a more efficient approach in which the model data communication speed is improved upon with optimized collective communication methods. We abstract three new communication operations and implement them on top of Harp [22]. We utilize two Harp-LDA applications and compare them with Yahoo! LDA [8] and Petuum LDA [4], two well-known implementations in the domain. This is done on three datasets each with a total of 10 billion model parameters. The results on a 100-node, 4000-thread cluster show that collective communication optimizations can significantly reduce communication overhead and improve model convergence speed.

The following sections describe: LDA modeling and CGS algorithm (Section 2), big model data problem in parallel LDA (Section 3), the communication challenges of LDA model data (Section 4), Harp-LDA implementations (Section 5) performance results and comparisons (Section 6), and conclusions (Section 7).

2 Background

LDA modeling techniques can find the latent structures inside data which are abstracted as a collection of documents, each with a bag of words. It models each document as a mixture of latent topics, and each topic as a multinomial distribution over words. Then an inference algorithm works iteratively on the data until convergence and outputs the most likely topic assignments for the data. (see Fig. 1(a)). Similar to Singular Value Decomposition (SVD) (see Fig. 1(b)), LDA can be also viewed as a sparse matrix decomposition technique on the training data as a word-document matrix. But it roots on a probabilistic foundation and has totally different computation characteristics.

Among the inference algorithms for LDA, Collapsed Gibbs Sampling (CGS) [19] shows high scalability in parallelization [11, 18], especially compared with another commonly used algorithm, Collapsed Variational Bayes (CVB) [9, 3, 6]. CGS is a Markov chain Monte Carlo (MCMC) type algorithm. In the "initialize" phase, each training data point, or token, is assigned to a random topic denoted as z_{ij}. 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}:

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

Here superscript $\neg ij$ means that the corresponding token is excluded. V is 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}, are the model data. Hyper parameters α and β control the topic density in the final model output. The model data gradually converge in the process of iterative sampling. This is the phase where the "burn-in" occurs and finally reaches "stationary".

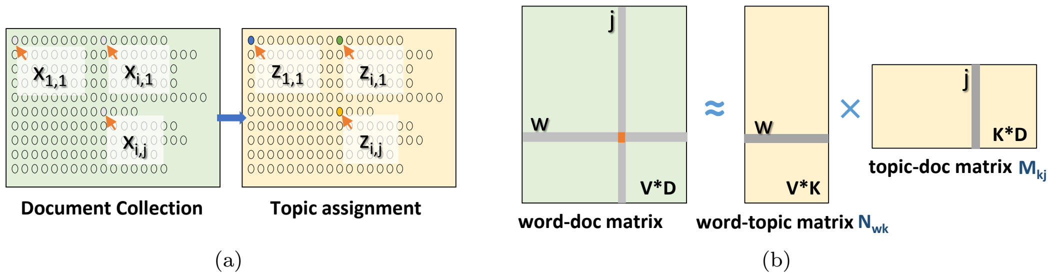


Figure 1: (a) Topics Discovery (b) View of Matrix Decomposition

The sampling performance is more memory bounded than computation bounded, since the computation itself is simple and mainly relies on accessing two large sparse model matrices in memory. In Algorithm. 1, sampling occurs by the column order on the word-document matrix, called “sample by document”. Although M_{kj} can be cached when sampling all the tokens in a document j, the memory access to N_{wk} is random since tokens are from different words. Symmetrically, sampling can run by “sample by word”. In both cases, the computation complexity is highly related to the model data size. SparseLDA [21] is an optimized CGS sampling implementation mostly used in the state-of-the-art LDA trainers. In Line 10 of Algorithm. 1, the conditional probability is usually computed for each k with total K times, but SparseLDA decreases the complexity to the number of non-zero items in N_{wk} and M_{kj}, which can be much smaller than K on average.

Algorithm 1: LDA Collapsed Gibbs Sampling Algorithm

```
input  : training data $X$, topic count $K$, hyperparamters $\alpha, \beta$
output: topic assignment matrix $Z_{ij}$, topic-document matrix $M_{kj}$, word-topic matrix $N_{wk}$
1  Initialize $M_{kj}, N_{wk}$ to zeros // Initialize phase
2  foreach document $j \in [1, D]$ do
3      foreach token position $i$ in document $j$ do
4          $z_{i,j} = k \sim Mult(\frac{1}{K})$ // sample topics by multinomial distribution
5          $m_{k,j} \mathrel{+}= 1; n_{w,k} \mathrel{+}= 1$ // token $x_{i,j}$ is word $w$, update the model matrices
   // Burn-in and Stationary phase
6  repeat
7      foreach document $j \in [1, D]$ do
8          foreach token position $i$ in document $j$ do
9              $m_{k,j} \mathrel{-}= 1; n_{w,k} \mathrel{-}= 1$ // decrease counts
10             $z_{i,j} = k' \sim p(z_{i,j} = k|rest)$ // sample a new topic by Eq.(1)
11             $m_{k',j} \mathrel{+}= 1; n_{w,k'} \mathrel{+}= 1$ // increase counts for the new topic
12 until convergence;
```

3 Big Model Data Problem in Parallel LDA

Sampling on Z_{ij} in CGS is a strict sequential procedure, although it can be parallelized without much loss in accuracy [11]. Parallel LDA can be performed in a distributed environment or a shared-memory environment. Because of the huge volume of the training documents, we focused on the distributed environment which is formed by a number of compute nodes deployed with a single worker process apience. Here every worker takes a partition of the training document set and performs the sampling procedure with multiple threads. The workers either communicate through point-to-point communication or collective communication.

LDA model data contains four parts: I. Z_{ij} - topic assignments on tokens, II. N_{wk} - token counts of words on topics (word-topic matrix), III. M_{kj} - token counts of documents on topics (document-topic matrix), and IV. $\sum_w N_{wk}$ - token counts of topics. Here Z_{ij} is always stored along with the training tokens. For the other three, because the training tokens are partitioned by document, M_{kj} is stored locally, while N_{wk} and $\sum_w N_{wk}$ are shared. For the shared model data, a parallel LDA implementation may use the "latest model" or a "stale model" in the sampling procedure. "Latest model" means the current model used in computation is up-to-date and not modified simultaneously by other workers, while "stale model" means the model values are old. The consistency of the model data is important to the convergence speed.

Now we calculate the size of N_{wk}, a huge but sparse $V * K$ matrix. Note that the word distribution in the training data generally follows the power-law. If we sort the words based on their frequencies from high to low, for a word with rank i, its frequency is $freq(i) = C * i^{-\lambda}$. Then for the size of training tokens W, we have

$$W = \sum_{i=1}^{V}(freq(i))) = \sum_{i=1}^{V}(C * i^{-\lambda}) \approx C * (ln(V) + \gamma + \frac{1}{2V}) \tag{2}$$

Here λ is a constant near 1, constant $C = freq(1)$. To simplify the analysis, we consider $\lambda = 1$. Then W is the partial sum of harmonic series which have logarithmic growth, where γ is the Euler-Mascheroni constant ≈ 0.57721. The actual model size depends on the non-zero cell count of the matrix (denoted as S). In the "initialize" phase of CGS, with random topic assignment, a word i gets $max(K, freq(i))$ non-zero cells. If $freq(J) = K$, then $J = C/K$, and we get:

$$S_{init} = \sum_{i=1}^{J} K + \sum_{i=J+1}^{V} freq(i) = W - \sum_{i=1}^{J} freq(i) + \sum_{i=1}^{J} K = C * (lnV + lnK - lnC + 1) \tag{3}$$

The actual initial model size S_{init} is logarithmic to matrix size $V * K$, so $S << V * K$. However this does not mean S_{init} is small. Since C can be very large, even $C * ln(V * K)$ can result in a large number. But in the progress of iterations, the model data size shrinks as the model converges. When a stationary state is reached, the average number of topics per word drops to a certain small constant ratio of K, which is determined by the concentration parameters α, β and the nature of the training data itself.

The local vocabulary size V' on each worker determines the model data volume required for computation. When documents are randomly partitioned to N processes, every word with a frequency larger than N gets a high probability to occur on all the processes. If $freq(L) = N$ at rank L, we get: $L = \frac{W}{(lnV+\gamma)*N}$. For a large training dataset, the ratio between L and V is often very high. This means the local computation requires most of the model data. Fig. 2 shows the difficulty of controlling the local vocabulary size in random document partitioning. When 10 times more partitions are introduced, there is only a sub-linear decrease of the vocabulary size per partition. Taking "clueweb" and "enwiki" datasets as examples (the contents of these datasets are discussed in Section 6), in "clueweb" each partition gets 92.5% of V when the training documents are randomly split into 128 partitions. "enwiki" is around 12 times smaller than "clueweb". It gets 90% of V with 8 partitions, keeping a similar ratio. In summary, though the local model data size reduces as the number of compute nodes grows, the percentage is quite high in many situations.

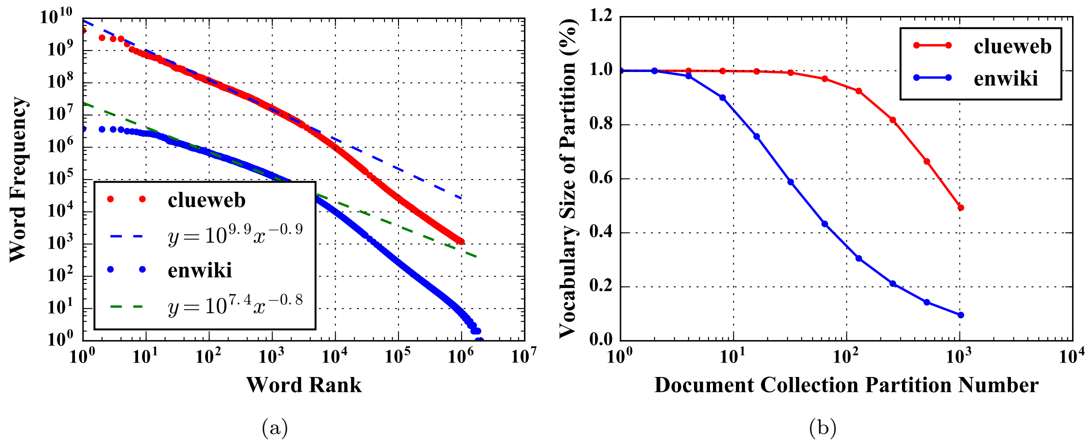


Figure 2: (a) Zipf's Law of the word frequencies (b) Vocabulary Size vs. Document Partitioning

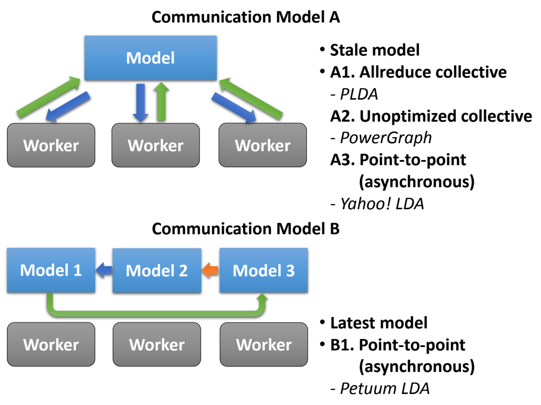
4 Communication Challenges of LDA Model Data

The analysis in the previous sections shows three properties of the big LDA model data: 1. The initial model size is huge but reduces as the model data converges; 2. The model data required in local computation is a high percentage of all the model data; 3. The local computation time is related to the local model size. All these indicate model data communication optimization is necessary because it can accelerate the process of model update and results in a huge benefit to computation and communication in later iterations. Of the various communication methods used in the state-of-the-art implementations, we summarize them all into two communication models (see Fig. 3(a)).

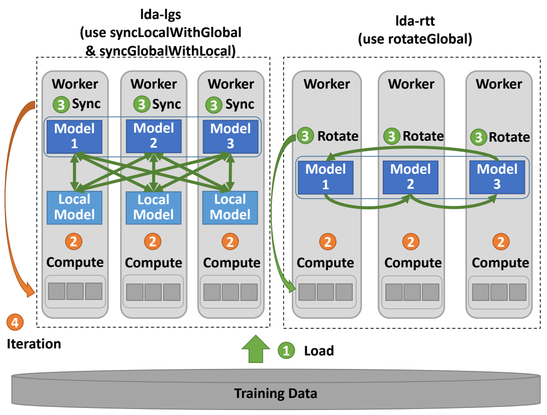
In Communication Model A, the computation occurs on stale model data. Before performing the sampling procedure, workers fetch the related model data to local. After the computation, they send updates back to the model data. There are many models in this category. In A1, without holding a shared model data, it synchronizes local models through "allreduce" operation. PLDA [17] follows this model. "allreduce" is routing optimized, but it communicate model data without consideration of the model data requirement in the local computation, causing high memory usage and high communication load. In A2, model data are just fetched and returned directly in a collective way. PowerGraph follows this model [13, 5]. Though it sends less data compared with A1, the performance is low for lack of routing optimization. A more popular model is A3, which uses asynchronous point-to-point communication. Yahoo! LDA [20, 10] and Parameter Server [14] follow this model. Here each worker independently fetches and updates the related model data without waiting for other workers. This model can ease the communication overhead. However, its model update rate is not guaranteed. The number of updates on the model data per iteration varies so that a word's model data may be updated by either model changes from all the training tokens, a part of them, or even no change. A solution to this problem is to combine A3 and A2. This is implemented in Petuum (version 0.93) LDA [15].

In Communication Model B, each worker first takes a partition of the model data after which the model is "shifted" between workers. When all the partitions are accessed by all the workers, an iteration is completed. There is only one model B1 which uses asynchronous point-to-point communication. Petuum(version 1.1) LDA [16, 4] follows this model.

A better solution for Communication Model A can be a conjunction of A1 and A2 with new collective communication abstractions. There are three advantages to such a strategy.



(a)



(b)

Figure 3: (a) Communication Models (b) Harp-LDA Implementations

First, considering the model data requirement in local computation, it reduces the data communicated. And second, it optimizes routing through searching "one-to-all" communication patterns. Finally it maintains the model update rate compared with asynchronous methods. For Communication Model B, using collective communication is also helpful because it maximizes bandwidth usage between compute modes and avoids flooding the network with small messages, which is what B1 does.

5 Harp-LDA Implementations

Based on the analysis above, we parallelize LDA with optimized collective communication abstractions. We use "table" abstraction defined in Harp [22] to organize the shared model data. Each table may contain one or more model data partitions, and the tables defined on different processes are associated in order to manage a distributed model dataset. We partition the model data based on the range of word frequencies in the training dataset. The lower the frequency of the word, the higher the partition ID given. Then we map partition IDs to process IDs based on the modulo operation. In this way, each process contains model data partitions with words whose frequencies rank from high to low.

We add three collective communication operations. The first two operations, "syncGlobalWithLocal" and "syncLocalWithGlobal", are paired. Here another type of table is defined to describe the local model data. Each partition in these tables is considered a local version of a global partition according to the corresponding ID. "syncGlobalWithLocal" merges partitions from different local model data tables to one in the global tables while "syncLocalWithGlobal" redistributes the partitions in the global model data tables to local tables. Lastly, "rotateGlobal" considers processes in a ring topology and shifts the partitions in the model data table from one process to the next neighbor.

We present two parallel LDA implementations. One is "lgs", which uses "syncGlobalWithLocal" paired with "syncLocalWithGlobal". Another is "rtt", which uses "rotateGlobal" (see Fig. 3(b)). In both implementations, the computation and the communication are pipelined, i.e., the model data is sliced in two and communicated in turns. Model Data Part IV is synchronized through A1 at the end of every iteration. SparseLDA algorithm is used for the sampling procedure. "lgs" samples by document while "rtt" samples by word. This is done to keep the consistency between implementations for unbiased communication performance comparisons in future experiments.

6 Experiments

Experiments are done on a cluster [2] with Intel Haswell architecture. This cluster contains 32 nodes each with two 18-core 36-thread Xeon E5-2699 processors and 96 nodes each with two 12-core 24-thread Xeon E5-2670 processors. All the nodes have 128GB memory and are connected with 1Gbps Ethernet (eth) and Infiniband (ib). For testing, 31 nodes with Xeon E5-2699 and 69 nodes with Xeon E5-2670 are used to form a cluster of 100 nodes each with 40 threads. All the tests are done with Infiniband through IPoIB support.

Several datasets are used (see Table 1). The model data settings are comparable to other research work [12], each with a total of 10 billion parameters. α and β are both fixed to a common used value 0.01 to exclude dynamic tuning. We test several implementations: "lgs", "lgs-4s" ("lgs" with 4 rounds of model synchronization per iteration, each round with 1/4 of the training tokens) and "rtt". To evaluate the quality of the model results, we use the model data likelihood on the training dataset to monitor the model convergence. We compare our implementations with two leading implementations, Yahoo! LDA and Petuum LDA, and thereby learn how the communication methods affect LDA performance by studying the model convergence speed.

Dataset	Num. of Docs	Num. of Tokens	Vocabulary	Doc Len. AVG/STD	Num. of Topics	Init. Model Size
clueweb	50.5M	12.4B	1M	224/352	10K	14.7GB
enwiki	3.8M	1.1B	1M	293/523	10K	2.0GB
bi-gram	3.9M	1.7B	20M	434/776	500	5.9GB

Both "enwiki" and "bi-gram" are English articles from Wikipedia [7].
"clueweb" is 10% of ClueWeb09 which is a collection of English web pages [1].

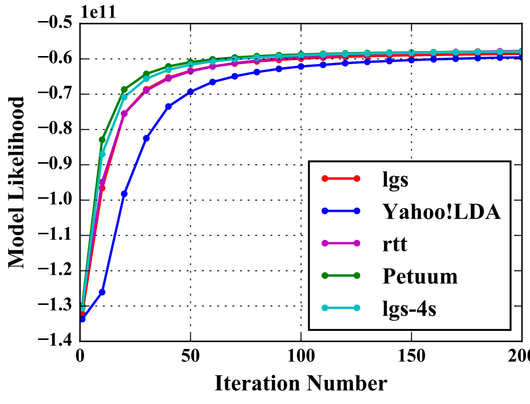
Table 1: Training Data Settings in the Experiments

6.1 Model Convergence Speed on Iteration

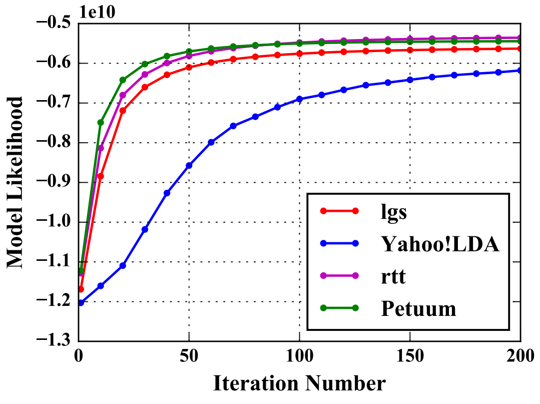
We compare the model convergence speed by analyzing model outputs on Iteration 1, 10, 20... 200. In one iteration, every training token is sampled once. Thus the number of model updates in each iteration is equal. Then we see how the model converges with the same amount of model updates.

On "clueweb" (see Fig. 4(a)), Petuum has the highest model likelihood on all the iterations. Due to "rtt"'s preference of using stale thread-local data in multi-thread sampling, the convergence speed is slower. The lines of "rtt" and "lgs" are overlapped for their similar convergence speeds. In contrast to "lgs", the convergence speed of "lgs-4s" is as high as Petuum. This shows that increasing the model update rate improves the convergence speed. Yahoo! LDA has the slowest convergence speed because asynchronous communication did not guarantee all the model updates were seen in each iteration. On "enwiki" (see Fig. 4(b)), as before, Petuum achieves the highest accuracy out of all iterations. "rtt" converges to the same model likelihood level as Petuum at iteration 200. "lgs" demonstrates slower convergence speed but still achieves high model likelihood, while Yahoo! LDA has both the slowest convergence speed and the lowest model likelihood at iteration 200.

These results match with our previous analysis. Though the number of model updates is the same, an implementation using stale model data converges slower than one using the latest model. For those using stale model data, "lgs-4s" is faster than "lgs" while "lgs" is faster than Yahoo! LDA. This means by increasing the model update rate, the model data used in computation is newer, and the convergence speed is improved.



(a)



(b)

Figure 4: (a) Model Convergence Speed on "clueweb" (b) Model Convergence Speed on "enwiki"

6.2 Model Convergence Speed on Elapsed Time

We first compare the execution speed between "lgs" and Yahoo! LDA. On "clueweb", we show the convergence speed based on the elapsed execution time (see Fig. 5(a)). Yahoo! LDA takes more time to finish Iteration 1 due to its slow model initialization, which demonstrates it has a sizable overhead on the communication end. In later iterations, while "lgs" converges faster, Yahoo! LDA catches up after 30 iterations. This observation could be explained by our slower computation speed. To counteract the computation overhead, we increase the number of model synchronization per iteration to four. Thus the computation overhead is reduced by using a newer and smaller model. Although the execution time for "lgs-4s" is still slightly longer than Yahoo! LDA, it obtains higher model likelihood and maintains faster convergence speed in the whole execution.

Similar results are shown on "enwiki", but this time "lgs" not only achieves higher model likelihood but also had faster model convergence speed throughout the whole execution (see Fig. 5(c)). From both experiments, we learn that though the computation was slow in "lgs", with collective communication optimization, the model size quickly shrinks so that its computation time is reduced significantly. At the same, although Yahoo! LDA does not have any extra overhead other than computation in each iteration, its iteration execution time reduces slowly because it keeps computing with an older model (see Fig. 5(b)(d)).

Next we compare "rtt" and Petuum LDA on "clueweb"and "bi-gram". On "clueweb", both sides' execution times and the model likelihood achieved are similar (see Fig. 5(c)). Both are around 2.7 times faster than the results in "lgs" and Yahoo! LDA. This is because they use the latest model data for sampling, and using "sample by word" leads to better performance. Though "rtt" has higher computation time compared with Petuum LDA, the communication overhead per iteration is lower. When the execution arrives at the final few iterations, while computation time per iteration in "rtt" is still higher, the whole execution time per iteration becomes lower ((see Fig. 5(f)(g)(h))). This is because Petuum communicates each word's model data in small messages and generates high overhead. On "bi-gram", the results show that Petuum does not perform well when the number of words in the model data increases. The high overhead in communication causes the convergence speed to be very slow, and Petuum cannot even continue executing after 60 iterations due to a memory outage (see Fig. 5(d)). Fig. 5(j)(k)(l) shows this performance difference on communication.

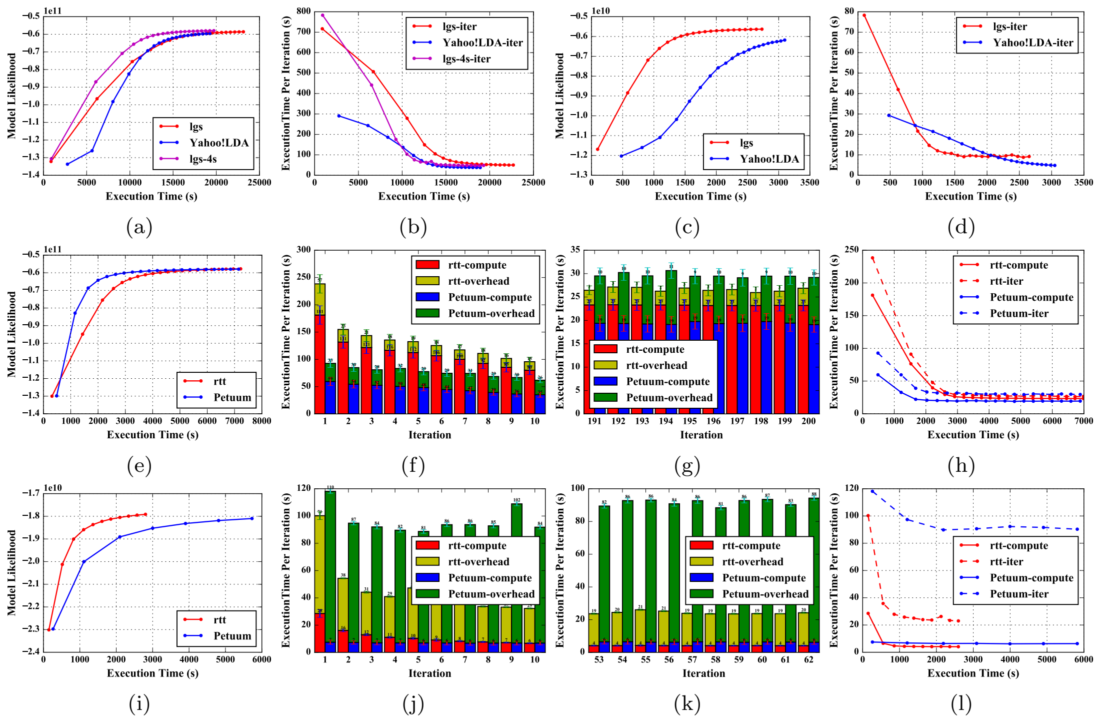


Figure 5: (a) Model Likelihood vs. Elapsed Time on “clueweb” (b) Iteration Time vs. Elapsed Time on “clueweb” (c) Model Likelihood vs. Elapsed Time on “enwiki” (d) Iteration Time vs. Elapsed Time on “enwiki” (e) Model Likelihood vs. Elapsed Time on “clueweb” (f) First 10 Iteration Times on “clueweb” (g) Final 10 Iteration Times on “clueweb” (h) Iteration Time vs. Elapsed Time on “clueweb” (i) Model Likelihood vs. Elapsed Time on “bi-gram” (j) First 10 Iteration Times on “bi-gram” (k) Final 10 Iteration Times on “bi-gram” (l) Iteration Time vs. Elapsed Time on “bi-gram”

7 Conclusion

Through the analysis on the LDA model data, we identify three model data properties in parallel LDA computation: 1. The model data requirement in local computation is high; 2. The time complexity of local sampling is related to the required model data size; 3. The model data size shrinks as it converges. These properties suggest that using collective communication optimizations can improve the model update speed, which allows the model to converge faster. When the model converges quickly, the model data shrinks greatly and the iteration execution time also reduces. We show that optimized collective communication methods perform better than asynchronous methods in parallel LDA. “lgs” results in faster model convergence and higher model likelihood at iteration 200 compared to Yahoo! LDA. On “bi-gram”, “rtt” shows significantly lower communication overhead than Petuum LDA, and the total execution time of “rtt” is 3.9 times faster. On “clueweb”, although the computation speed of the first iteration is 2- to 3-fold slower, the total execution time remains similar.

Despite the implementation differences between “rtt”, “lgs”, Yahoo! LDA, and Petuum LDA, the advantages of collective communication methods are evident. Compared with asynchronous communication methods, collective communication methods can optimize routing between parallel workers and maximize bandwidth utilization. Though a collective communication will result in global waiting, the resulting overhead is not as high as speculated. The chain reaction set off by improving the LDA model update speed amplifies the benefits of using collective communication methods.

In future work, we will focus on improving intra-node LDA performance in many-core systems, and apply our model data communication strategies to other machine learning algorithms facing big model data problems.

Acknowledgments

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

References

[1] Clueweb. `http://lemurproject.org/clueweb09.php/`.

[2] FutureSystems. `https://portal.futuresystems.org`.

[3] Mahout LDA. `https://mahout.apache.org/users/clustering/latent-dirichlet-allocation.html`.

[4] Petuum LDA. `https://github.com/petuum/bosen/wiki/Latent-Dirichlet-Allocation`.

[5] PowerGraph LDA. `https://github.com/dato-code/PowerGraph/blob/master/toolkits/topic_modeling/topic_modeling.dox`.

[6] Spark LDA. `http://spark.apache.org/docs/latest/mllib-clustering.html`.

[7] Wikipedia. `https://www.wikipedia.org`.

[8] Yahoo! LDA. `https://github.com/sudar/Yahoo_LDA`.

[9] D. Blei, A. Ng, and M. Jordan. Latent Dirichlet Allocation. *The Journal of Machine Learning Research*, 3:993–1022, 2003.

[10] A. Ahmed et al. Scalable Inference in Latent Variable Models. In *WSDM*, 2012.

[11] D. Newman et al. Distributed Algorithms for Topic Models. *The Journal of Machine Learning Research*, 10:1801–1828, 2009.

[12] E. Xing et al. Petuum: A New Platform for Distributed Machine Learning on Big Data. In *KDD*, 2015.

[13] J. Gonzalez et al. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *OSDI*, 2012.

[14] M. Li et al. Scaling Distributed Machine Learning with the Parameter Server. In *OSDI*, 2014.

[15] Q. Ho et al. More Effective Distributed ML via a Stale Synchronous Parallel Parameter Server. In *NIPS*, 2013.

[16] S. Lee et al. On Model Parallelization and Scheduling Strategies for Distributed Machine Learning. In *NIPS*, 2014.

[17] Y. Wang et al. PLDA: Parallel Latent Dirichlet Allocation for Large-scale Applications. In *Algorithmic Aspects in Information and Management*, pages 301–314. Springer, 2009.

[18] Y. Wang et al. Peacock: Learning Long-Tail Topic Features for Industrial Applications. *ACM Transactions on Intelligent Systems and Technology*, 6(4), 2015.

[19] P. Resnik and E. Hardist. Gibbs Sampling for the Uninitiated. Technical report, University of Maryland, 2010.

[20] A. Smola and S. Narayanamurthy. An Architecture for Parallel Topic Models. *Proceedings of the VLDB Endowment*, 3(1-2):703–710, 2010.

[21] L. Yao, D. Mimno, and A. McCallum. Efficient Methods for Topic Model Inference on Streaming Document Collections. In *KDD*, 2009.

[22] B. Zhang, Y. Ruan, and J. Qiu. Harp: Collective Communication on Hadoop. In *IC2E*, 2014.