JMLR: Workshop and Conference Proceedings 36:17–28, 2014 BIGMINE 2014

Collapsed Gibbs Sampling for Latent Dirichlet Allocation on Spark

Zhuolin Qiu QIUZHUOLIN@LIVE.COM
Bin Wu WUBIN@BUPT.EDU.CN
Bai Wang WANGBAI@BUPT.EDU.CN
Chuan Shi SHICHUAN@BUPT.EDU.CN
Le Yu YULEBUPT@GMAIL.COM
Beijing Key Lab of Intelligent Telecommunication Software and Multimedia
Beijing University of Posts and Telecommunications
Beijing 100876, China

Editors: Wei Fan, Albert Bifet, Qiang Yang and Philip Yu

Abstract

In this paper we implement a collapsed Gibbs sampling method for the widely used latent Dirichlet allocation (LDA) model on Spark. Spark is a fast in-memory cluster computing framework for large-scale data processing, which has been the talk of the Big Data town for a while. It is suitable for iterative and interactive algorithm. Our approach splits the dataset into $P * P$ partitions, shuffles and recombines these partitions into P sub-datasets using rules to avoid conflicts of sampling, where each of P sub-datasets only contains P partitions, and then parallel processes each sub-dataset one by one. Despite increasing the number of iterations, this method reduces data communication overhead, makes good use of Spark's efficient iterative execution and results in significant speedup on large-scale datasets in our experiments.

Keywords: LDA, Spark, collapsed Gibbs sampling

1. Introduction

The latent Dirichlet allocation (LDA) model is a general probabilistic framework that was first proposed by Blei et al. (2003) to discover topics in text documents. The main idea of the LDA model is based on the assumption that each document may be viewed as a mixture of various topics, where a topic is represented as a multinomial probability distribution over words. Learning the mixing coefficients for the various document-topic and topic-word distributions is a problem of Bayesian inference. Blei et al. (2003) developed a variational Bayesian algorithm to approximate the posterior distribution in the original paper; and an alternative inference method using collapsed Gibbs sampling to learn the model from data was subsequently proposed by Griffiths and Steyvers (2004). Both of the approaches have their advantages and disadvantages: the variational approach is arguably faster computationally but can possibly lead to inaccurate inferences and biased learning; though the collapsed Gibbs sampling approach is in principal more accurate, it has high computational complexity, which makes it inefficient on large datasets.

With the advent of the era of big data, the amount of data in our world has been exploding, and analyzing large datasets has becoming a key in many areas. LDA model, with the collapsed Gibbs sampling algorithm in common use, has now been broadly applied in machine learning and data mining, particularly in classification, recommendation and web search, in which both high accuracy and speed are required. Some improvements have been explored for adapting to big data based on variational Bayesian, such as Teh et al. (2006); Nallapati et al. (2007); Wolfe et al. (2008). The Collapsed Variational Bayes (CVB) algorithm has been implemented in Mahout and Wen et al. (2013) improved it by combining GPU and Hadoop. CVB converges faster than collapsed Gibbs sampling, but the latter attains a better solution in the end with enough samples. So there is a significant motivation to speedup collapsed Gibbs sampling for LDA model.

In this general context we introduce our parallel collapsed Gibbs sampling algorithm and demonstrate how to implement it on Spark, a new in-memory cluster computing framework proposed by Zaharia et al. (2010, 2012). The key idea of our algorithm is to reduce the time taken for communication and synchronization. Only a part of global parameters are parallel transferred in communication and no complex calculation is needed in synchronization. However, a disadvantage of our algorithm is that the number of iterations significantly increases. In order to overcome this problem, we adopt Spark, which is very suitable for iterative and interactive algorithm, to implement our method.

We first briefly review the related work on collapsed Gibbs sampling in Section 2; some prior knowledge, including the LDA model, collapsed Gibbs sampling for LDA and Spark, is reviewed in Section 3; in Section 4 we describe the details of our distributed approach on Spark via a simple example; in Section 5 we present perplexity and speedup results; finally, we discuss future research plans in Section 6.

2. Related Work

Various implementations and improvements have been explored for speeding up LDA model. Relevant collapsed Gibbs sampling methods are as follows:

- Newman et al. (2007, 2009) proposed two versions of LDA where the data and the parameters are distributed over distinct processors: Approximate Distributed LDA model (AD-LDA) and Hierarchical Distributed LDA model (HD-LDA). In AD-LDA, they simply implement LDA on each processor and update global parameters after a local Gibbs sampling iteration. HD-LDA can be viewed as a mixture model with P LDA, which optimizes the correct posterior quantity but is more complex to implement and slower to run.

- Porteous et al. (2008) presented a new sampling scheme, which produces exactly the same results as the standard sampling scheme but faster.

- An asynchronous distributed version of LDA (Async-LDA) was introduced by Asuncion et al. (2008). In Async-LDA, each processor performs a local collapsed Gibbs sampling step followed by a step of communicating with another random processor to gain the benefits of asynchronous computing. Async-LDA has been improved in GraphLab, a graph-based parallel framework for machine learning, which was proposed by Low et al. (2010).

- Yan et al. (2009) presented parallel algorithms of collapsed Gibbs sampling and collapsed variational Bayesian for LDA model on Graphics Processing Units (GPUs), which have massively built-in parallel processors with shared memory. And they proposed a novel data partitioning scheme to overcome the shared memory limitations.
- Wang et al. (2009) implemented collapsed Gibbs sampling on MPI and MapReduce called PLDA, then Liu et al. (2011) enhanced the implementation and called the new approach PLDA+.
- Xiao and Stibor (2010) proposed a novel dynamic sampling strategy to significantly improve the efficiency of collapsed Gibbs sampling and presented a straight-forward parallelization to further improve the efficiency.
- Ihler and Newman (2012) presented a modified parallel Gibbs sampler, which obtains the same speedups as AD-LDA, but provides an online measure of the approximation quality compared to a sequential sampler.

In these works, the parallel LDA was first proposed and respectively implemented on GraphLab, GPUs, MPI and MapReduce. All the above could run with high speed up ratio and parallel efficiency. Unlike their works, we improved the collapsed Gibbs sampling method and implemented it on Spark, which is a new model of cluster computing that aims to make data analytics fast.

3. Prior Knowledge

3.1. Latent Dirichlet Allocation

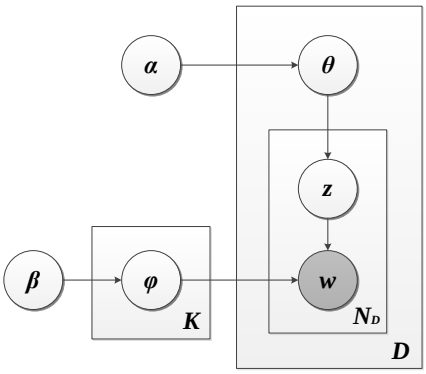

Figure 1: Graphical model for LDA.

Before delving into the details of our distributed algorithm for LDA on Spark, we first briefly review the standard LDA model. In LDA, each of D documents is modeled as a mixture over K latent topics, each being a multinomial distribution over W vocabulary words. In order to generate a new document j, we first draw a mixing proportion $\theta_{k|j}$ from a Dirichlet with parameter α. For the i^{th} word in the document, a topic assignment z_{ij} is

drawn with topic k chosen with probability $\theta_{k|j}$. Then word x_{ij} is drawn from the z_{ij}^{th} topic, with x_{ij} taking on value w with probability $\phi_{w|k}$, where $\phi_{w|k}$ is drawn from a Dirichlet prior with parameter β. Finally, the generative process is below:

$$\theta_{k|j} \sim Dir(\alpha) \quad \phi_{w|k} \sim Dir(\beta) \quad z_{ij} \sim \theta_{k|j} \quad x_{ij} \sim \phi_{w|z_{ij}} \tag{1}$$

where $Dir(\alpha)$ represents the Dirichlet distribution. Figure 1 shows the graphical model representation of the LDA model.

Given the training data with N words $x = x_{ij}$, it is possible to infer the posterior distribution of latent variables. An efficient procedure is to use collapsed Gibbs sampling, which samples the latent variables $z = z_{ij}$ by integrating out $\theta_{k|j}$ and $\phi_{w|k}$. The conditional probability of z_{ij} is computed as follows:

$$p(z_{ij} = k|z^{\neg ij}, x, \alpha, \beta) \propto (\alpha + n_{k|j}^{\neg ij})(\beta + n_{x_{ij|k}}^{\neg ij})(W\beta + n_k^{\neg ij})^{-1} \tag{2}$$

where the superscript $\neg ij$ means the corresponding data-item is excluded in the count values, $n_{k|j}$ denotes the number of tokens in document j assigned to topic k, $n_{x_{ij|k}}$ denotes the number of tokens with word w assigned to topic k, and $n_k = \sum_w n_{w|k}$.

3.2. Spark

Spark is a fast and general engine for large-scale data processing that can run programs up to 100x faster than Hadoop MapReduce in memory, or 10x faster on disk. The project started as a research project at the UC Berkeley AMPLab in 2009, and was open sourced in early 2010. After being released, Spark grew a developer community on GitHub and entered Apache in 2013 as its permanent home. A wide range of contributors now develop the project (over 120 developers from 25 companies) spa.

Relative to Hadoop, Spark was designed to run more complex, multi-pass algorithms, such as the iterative algorithms that are common in machine learning and graph processing; or execute more interactive ad hoc queries to explore the data. The core problem of these applications is that both multi-pass and interactive applications need to share data across multiple MapReduce steps. Unfortunately, the only way to share data between parallel operations in MapReduce is to write to a distributed file system, which adds substantial overhead due to data replication and disk I/O. Indeed, it had been found that this overhead could take up more than 90% of the running time of common machine learning algorithms implemented on Hadoop. Spark overcomes this problem by providing a new storage primitive called resilient distributed dataset (RDD), which represents a read-only collection of objects partitioned across a set of machines and provides fault tolerance without requiring replication, by tracking how to compute lost data from previous RDDs. RDD is the key abstraction in Spark. Users can explicitly cache an RDD in memory or disk across machines and reuse it in multiple parallel operations.

Spark includes MLlib, which is a library of machine learning algorithms for large data, including classification, regression, clustering, collaborative filtering, dimensionality reduction, as well as underlying optimization primitives. However, no topic model algorithm has been added so far.

4. Distributed Learning for LDA on Spark

We now introduce our distributed version of LDA where the data and the parameters are distributed over distinct processors on Spark (which we will call Spark-LDA). Table 1 shows the symbols associated with Spark-LDA which are used in this paper. We first simply distribute N_d and N_w over P processors, with $N_d^i = N_d/P$ and $N_w^j = N_w/P$ on each processor, then show how to split, shuffle and recombine the datasets X to P sub-datasets X_p with P partitions.

Table 1: Symbols associated with Spark-LDA

Symbol	Description
X	Datasets
X_p	The p^{th} sub-datasets
K	Number of topics
D	Number of documents
W	Number of words in the vocabulary
N_d	Document-topic count matrix
N_w	Word-topic count matrix
N_d^i	The i^{th} part of N_d
N_w^j	The j^{th} part of N_w
N_k	Word count assigned with topic k
N_k^p	Local copy of N_k on the p^{th} processor
$\tilde{N}_k^p$	Difference between N_k and N_k^p after sampling on a processor

In order to avoid potential read/write conflicts on N_d and N_w, it must be confirmed that each partition in a sub-dataset cannot contain the same documents and words. Ihler and Newman (2012) proposed a data partition scheme to meet the requirement. However, this scheme requires an efficient shared memory, because each processor needs to access all partitions. Obviously it is not suitable for cluster computing framework, so we made some modifications to apply it on Spark. We first put the datasets X into $P * P$ partitions, through dividing the rows and columns into P parts, and put N_d and N_w into N_d^i and N_w^j corresponding X; then select P partitions from X, each of them containing different documents and words, to recombine to one sub-datasets. A simple example of the above process is shown in Figure 2, and we will describe the details in the following.

For ensuring network load balance during communication, the number of columns of each partition, which are associated with N_w^j, must be equal with each other. In addition, we must ensure that the number of words in each partition is close to each other, so as to balance the load of each processor during the sampling process for each sub-dataset. In fact, we only need to ensure that the value is minimal between the maximum and minimum. We take the following steps to meet the above requirements:

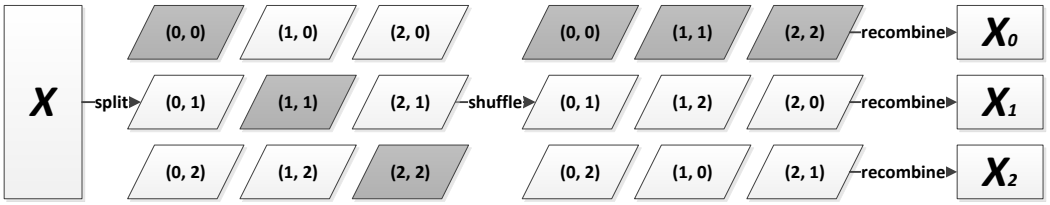

Figure 2: (x, y) represent partitions. In this case, X is split into 9 partitions, which are shuffled and recombined to X_0, X_1 and X_2.

- We first put the datasets X into P partitions, through dividing the columns into equal P parts; then swap the columns to make the number of words in each partition closer. After that, we divide the columns of N_w corresponding to X.
- Then we divide the rows into P parts, now the dataset X has been split into $P * P$ partitions; and make the number of words in each partition, which belongs to the same sub-dataset after recombination, closer. Though the number of rows is not necessarily equal, it is still very difficult to find an efficient way to solve this problem. Here, we use a simple randomized method to swap the rows and calculate the difference between the maximum and minimum number of words in each partition which belongs to the same sub-dataset. The smaller difference means better partitions. We can run this method for several times and select the best one by calling an RDDs cache method to tell Spark to try and keep the dataset in memory. It is fast, since the dataset is in memory and can be calculated parallel. In our experiment, it works well.
- Finally, we select P non-conflict partitions and recombine them into a sub-dataset. Repeat the process and we will get P sub-datasets at last.

Consider the sampling process for a sub-dataset. Each sub-dataset contains P non-conflict partitions, which can be distributed to P processors; recall that N_d^i and N_w^j have been distributed, too. So we can shuffle N_d^i and N_w^j corresponding to the partitions, and let them, which contain the same documents and words, on one processor. Then standard collapsed Gibbs sampling algorithm will be executed on each processor, N_d^i and N_w^j will be updated at the same time. After completing the sampling of all partitions in a sub-dataset, we calculate N_k by $\tilde{N}_k^p$, which tracked changes in N_k^p, and N_k is then broadcasted to all processors and stored as N_k^p. The above process via a simple example is given in Figure 3. Thus we have completed the sampling process for a sub-dataset, and then we need to repeat this process for remaining sub-datasets. After sampling all the sub-datasets, we have completed an iteration of the global.

To implement the algorithm 1 on Spark, there are two issues that must be resolved. One is that it is asked to operate three RDDs, X_p, N_d and N_w, during the sampling process.

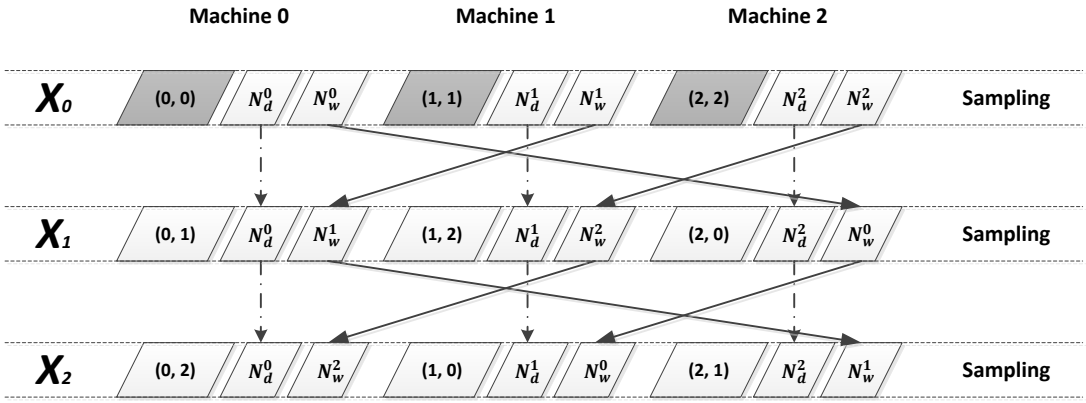

Figure 3: A simple example of the sampling process. Dotted line represents that the transmission on the same machine and the solid line indicates the transfer between different machines.

Algorithm 1 Spark-LDA

for $i < iteration$ **do**
  /* Sample one sub-dataset */
  for $j < number of sub\text{-}datasets$ **do**
    for each processor p in parallel **do**
      /* Sampling */
      Initialize $\tilde{N}_k^p = \{0\}$ and $N_k^p = N_k$
      Sample $z \in X_p$ with N_d^i, N_w^j, N_k^p and update $N_d^i, N_w^j, N_k^p, \tilde{N}_k^p$
    end for
    /* Communication and synchronization */
    Shuffle N_d^i, N_w^j and update $N_k = N_k + \sum_p \tilde{N}_k^p$
  end for
end for

However, Spark is not good at operating more than two RDDs by default. We take the following steps to solve the problem. First, we combine X_p, N_d and N_w into one RDD and start sampling; after sampling, we then split the RDD into three RDDs.

The other problem is the way to partition RDD. Spark computes the hash code as the partition id by default, and it provides another way to partition RDD by determining the ranges of RDD. Both of the two methods cannot meet our requirements, so we implement a new partition method, which uses our specified id as partition id. After finishing that, we implement a location function to confirm the partitions of X and N_d are placed as our algorithm's request.

Obviously, this parallel approach significantly increases the number of iterations; however, all operations are performed in memory and Spark has a highly efficient compression to reduce the size of data transfer in network, so the speed is very fast.

5. Experiments

We used three text datasets retrieved from the UCI Machine Learning Repository for evaluation (http://archive.ics.uci.edu/ml/datasets/Bag+of+Words): KOS blog entries, NIPS full papers and NY Times news articles. These three datasets span a wide range of collection size, content and average document length. The KOS dataset is the smallest one, NYT dataset is relatively large, while NIPS dataset is moderately sized. For each text collection, after tokenization and removal of stopwords, the vocabulary of unique words was truncated by only keeping words that occurred more than ten times. The characteristics of these three datasets are shown in Table 2.

Table 2: Size parameters for the three datasets used in experiments

Dataset	KOS	NIPS	NYT
Total number of documents, D	3,430	1,500	300,000
Size of vocabulary, W	6,909	12,419	102,660
Total number of words, N	467,714	1,932,365	99,542,125

We measured performance in two ways. First, we compared LDA (standard collapsed Gibbs sampling on a single processor) and our distributed algorithm, Spark-LDA, using two small datasets, KOS and NIPS, by computing the perplexity on the test set. Secondly, we measured the speedup of Spark-LDA using all these datasets. All experiments were carried out on the computers, each one equipped with an Intel Xeon 2.00GHz CPU and 8GBytes memory. LDA was executed on one computer while Spark-LDA was on 25 computers including 1 master and 24 workers.

In our perplexity experiments, we split each dataset into a training set and a test set by assigning about 10% of the words in each document to the test set. And for every document, the words in test set were never seen in training set. Then we learned our models on the training set, and measured the performance on the test set using perplexity which was computed as $Perplexity(x^{test}) = \exp\left(-\frac{1}{N^{test}} \log p(x^{test})\right)$. For each of our experiments, we performed $S = 10$ different Gibbs runs, with different random initializations and each ran lasting 1000 iterations, and we obtained a sample at the end of each of those runs. Then perplexities were calculated using:

$$\log p(x^{test}) = \sum_{j,w} \log \frac{1}{S} \sum_{s} \sum_{k} \theta^s_{k|j} \phi^s_{w|k}$$
$$\theta^s_{k|j} = \frac{\alpha + n^s_{k|j}}{K\alpha + n^s_j} \quad \phi^s_{w|k} = \frac{\beta + n^s_{w|k}}{W\beta + n^s_k} \tag{3}$$

where $\alpha = 50/K$ and $\beta = 0.1$.

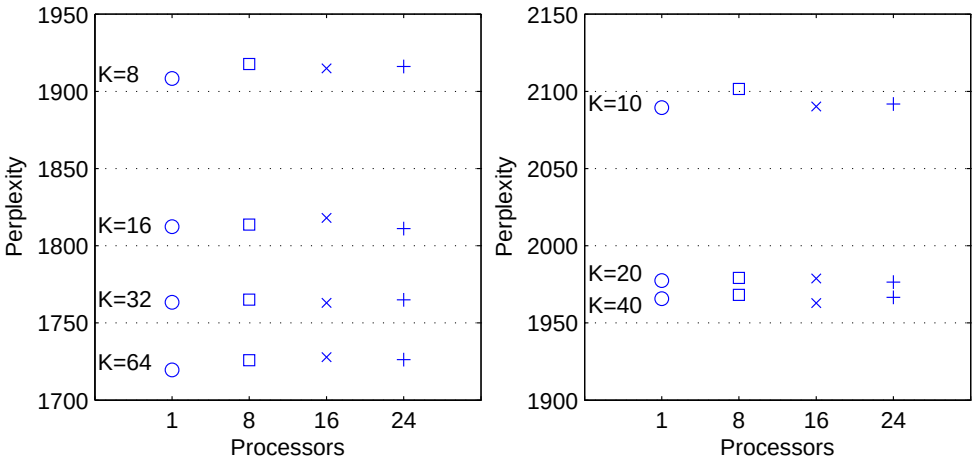

Figure 4: Test set perplexity versus number of processors P for KOS (left) and NIPS (right).

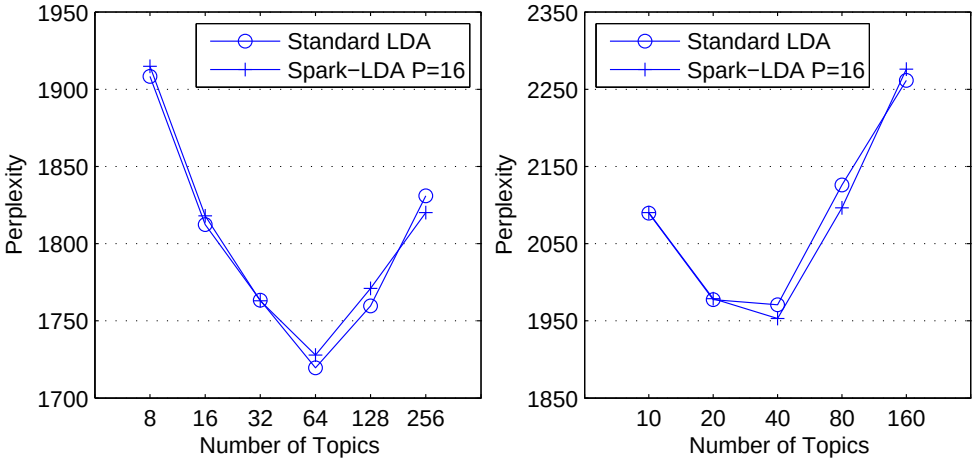

Figure 5: Test set perplexity versus number of topics K for KOS (left) and NIPS (right).

Figure 4 shows the test set perplexity for KOS and NIPS. We used standard LDA to compute perplexity for $P = 1$ and Spark-LDA for $P = 8, 16, 24$. The figure clearly shows that the perplexity results were of no significant difference between LDA and Spark-LDA for varying numbers of topics. It suggests that our algorithm converged to models having the same predictive power as standard LDA. We also used Spark-LDA to compute perplexities for KOS with $K = 128, 256$, NIPS with $K = 80, 160$, and the results presented the same conclusion. Because the perplexities increased with the number of topics increasing in the current parameters, which made the figure confusing, we only showed parts of these results in Figure 5. Limited by the hardware conditions, we did not perform experiments on more than 24 processors.

The rate of convergence is shown in Figure 6. As the number of processors increased, the rate of convergence slowed down in our experiments, since the information for sampling in

each processor was not accurate enough. However, all of these results eventually converged to the same range after the burn-in period of 1000 iterations.

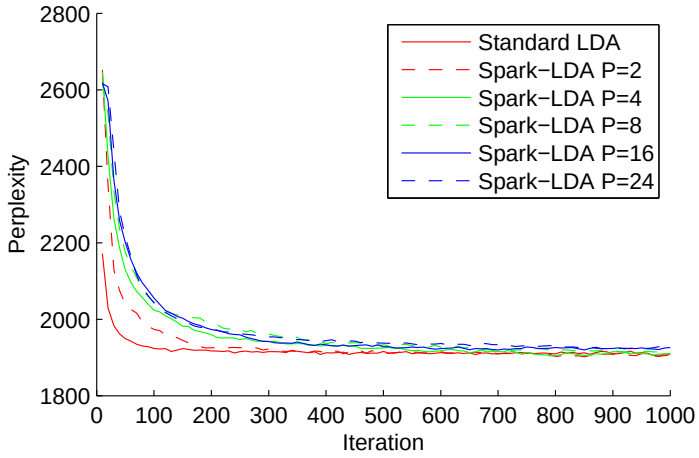

Figure 6: Test set perplexity versus iteration on KOS, $K = 8$.

We respectively performed speedup experiments on two small datasets (KOS, NIPS) and one large dataset (NYT). The speedup was computed as $Speedup = \frac{1}{(1-S)+S/P}$, where S was the percent of sampling time and P was the number of processes. The results are shown in Figure 7 (left). The figure clearly shows that speedup increased as the dataset size. The phenomenon reflected that our algorithm performed well on large dataset, but the effect was not significant on small datasets.

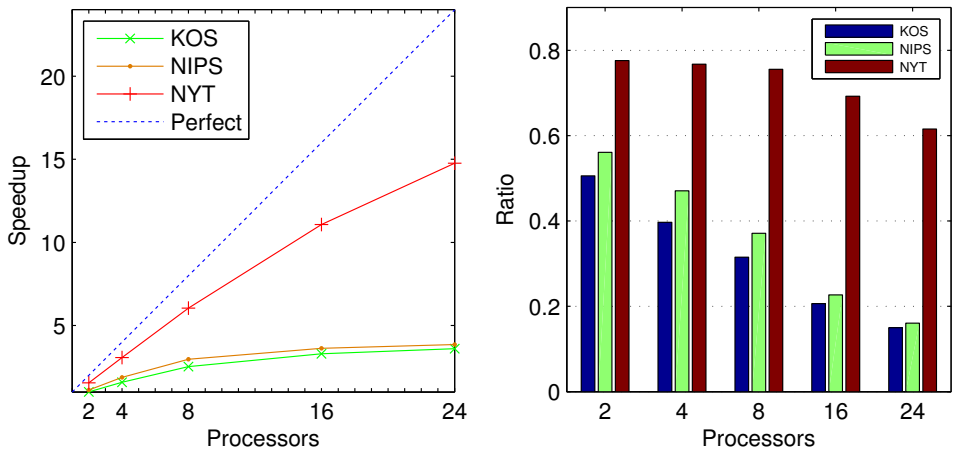

Figure 7: Speedup of Spark-LDA (left) and the proportion of the sampling time per iteration (right) on KOS, NIPS and NYT

In order to analyze the causes of this phenomenon, we then measured the proportion of the sampling time per iteration and showed the results in Figure 7 (right). With the

number of processors increasing, the number of words in each partition was fewer, and the execution time of sampling was less, too. In our experiments, only about 40ms was taken for sampling per iteration on KOS with $K = 8, P = 24$. However, starting jobs consumed constant time, so the proportion of the sampling time got smaller. This process explained why the performance of Spark-LDA was not good on small datasets. For large dataset, this phenomenon also occurred but still within the acceptable range in our experiments.

6. Discussion and Conclusions

In this paper, we described the details of our distributed approach. The results of our experiments show that our algorithm ensured accuracy and achieved impressive speedup. However, there are still two issues to be solved. One is low speedup caused by too few words in each partition; the other is how to make the speedup close to the perfect when the dataset is big enough. Both of them need to operate multiple RDDs at the same time. Limited to the status of Spark, it is difficult to solve these two problems. Spark, after all, is still in a period of rapid development, and it will be maturing with the contribution of many developers. We will continue to focus on Spark and optimize our algorithm in the future.

Acknowledgments

This work is supported by the National Key Basic Research and Department(973) Program of China (No.2013CB329603), the National Science Foundation of China (Nos.61375058, and 71231002), the Ministry of Education of China and China Mobile Research Fund (MCM20130351) and the Special Co-construction Project of Beijing Municipal Commission of Education.

References

Apache spark - lightning-fast cluster computing. URL https://spark.apache.org.

Arthur U Asuncion, Padhraic Smyth, and Max Welling. Asynchronous distributed learning of topic models. In *NIPS*, volume 2, pages 2–3, 2008.

David M Blei, Andrew Y Ng, and Michael I Jordan. Latent dirichlet allocation. *the Journal of machine Learning research*, 3:993–1022, 2003.

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

Alexander Ihler and David Newman. Understanding errors in approximate distributed latent dirichlet allocation. *Knowledge and Data Engineering, IEEE Transactions on*, 24 (5):952–960, 2012.

Zhiyuan Liu, Yuzhou Zhang, Edward Y Chang, and Maosong Sun. Plda+: Parallel latent dirichlet allocation with data placement and pipeline processing. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 2(3):26, 2011.

Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M Hellerstein. Graphlab: A new framework for parallel machine learning. *arXiv preprint arXiv:1006.4990*, 2010.

Ramesh Nallapati, William Cohen, and John Lafferty. Parallelized variational em for latent dirichlet allocation: An experimental evaluation of speed and scalability. In *Data Mining Workshops, 2007. ICDM Workshops 2007. Seventh IEEE International Conference on*, pages 349–354. IEEE, 2007.

David Newman, Arthur U Asuncion, Padhraic Smyth, and Max Welling. Distributed inference for latent dirichlet allocation. In *NIPS*, volume 20, pages 1081–1088, 2007.

David Newman, Arthur Asuncion, Padhraic Smyth, and Max Welling. Distributed algorithms for topic models. *The Journal of Machine Learning Research*, 10:1801–1828, 2009.

Ian Porteous, David Newman, Alexander Ihler, Arthur Asuncion, Padhraic Smyth, and Max Welling. Fast collapsed gibbs sampling for latent dirichlet allocation. In *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 569–577. ACM, 2008.

Yee Whye Teh, David Newman, and Max Welling. A collapsed variational bayesian inference algorithm for latent dirichlet allocation. In *NIPS*, volume 6, pages 1378–1385, 2006.

Yi Wang, Hongjie Bai, Matt Stanton, Wen-Yen Chen, and Edward Y Chang. Plda: Parallel latent dirichlet allocation for large-scale applications. *Algorithmic Aspects in Information and Management*, pages 301–314, 2009.

La Wen, Jianwu Rui, Tingting He, and Liang Guo. Accelerating hierarchical distributed latent dirichlet allocation algorithm by parallel gpu. *Journal of Computer Applications*, 33(12):3313–3316, 2013.

Jason Wolfe, Aria Haghighi, and Dan Klein. Fully distributed em for very large datasets. In *Proceedings of the 25th international conference on Machine learning*, pages 1184–1191. ACM, 2008.

Han Xiao and Thomas Stibor. Efficient collapsed gibbs sampling for latent dirichlet allocation. *Journal of Machine Learning Research-Proceedings Track*, 13:63–78, 2010.

Feng Yan, Ningyi Xu, and Yuan Qi. Parallel inference for latent dirichlet allocation on graphics processing units. In *NIPS*, volume 9, pages 2134–2142, 2009.

Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, pages 10–10, 2010.

Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.