Parallelism for LDA
Yang Ruan, Changsi An
(yangruan@indiana.edu, anch@indiana.edu)

1. Overview

As parallelism is very important for large scale of data, we want to use different technology to parallelize the latent dirichlet allocation algorithm. Latent dirichlet allocation (LDA) is a generative model that allows sets of observations to be explained by unobserved groups which explain why some parts of the data are similar.

There are two different implementation of this model, one is Expectation Maximum (EM method) and the other is Gibbs Sampling. Both of them use iterative algorithm to do the calculation, however, Gibbs Sampling is easier to implement in sequential code but harder to parallel while the EM method is easier to parallel as (Nallapati, Cohen et al.) has already use MapReduce technology to parallelize that. And we are going to parallel the Gibbs sampling implementation this time using iterative MapReduce and Message Passing Interface.

2. Algorithm

2.1 Guideline of Sequential Algorithm with LDA

LDA is a generative probabilistic model of a corpus. The basic idea is that documents are represented as random mixtures over latent topics, where each topic is characterized by a distribution over words(Blei, Ng et al. 2003). The LDA model is represented as a probalistic graphical model in Figure 1.

![Figure 1 graphic representation of LDA](image)

To clarify this model, assume we have the following convention: A corpus D : {d1, d2, .. dm}. Each corpus di : {w1, w2, ..., wn} is a sequence of words. In contrast to words, we have vocabulary V : {v1, v2, ..., vl} which is a non-duplicated projection of all words. Also we assume there exist K topics T = {t1, t2, ..., tk}.

The algorithm can be described in the following steps: first choose \( \theta \sim \text{Dirichlet}(\alpha) \). Then for each word \( w_i \) of document \( d_i \), sample a topic \( T_n \) which follows multinomial distribution with parameter \( \theta \), after that, sample a word \( w_i' \) according to \( p(w \mid T_n, \beta) \). After that, if \( w_i' \) is not \( w_i \), we will update \( \beta \) by augmenting the probability of word \( w_i' \) to topic \( T_n \) and diminishing the one of word \( w_i \).

Given the parameters \( \alpha \) and \( \beta \), the joint distribution of a topic mixture \( \theta \), a set of N topics \( z \), and a set of N words \( w \) is given by:

\[
p(\theta, z, w | \alpha, \beta) = p(\theta | \alpha) \prod_{n=1}^{N} p(z_n | \theta) p(w_n | z_n, \beta)
\]

(1)

Where \( p(z_n | \theta) \) is simply \( \theta_i \) is for the unique \( i \) such that \( z_n^i = 1 \).

We used gibbs sampling which is proposed by (Griffiths and Steyvers 2004) to implement our parallel LDA model. By considering the posterior distribution over the assignments of words to topics, \( P(z|w) \), we obtain estimates of \( \phi \) and \( \Theta \) by examining this posterior distribution. So we will assume \( \alpha \) and \( \beta \) both has a single value instead of being a vector. And the equation we used is shown as equation 2.

\[
p(z_{d,i} = k | z_{-(d,i)}, w_{d,i} = v, W_{-(d,i)}) \propto \frac{\binom{n_{d,k+\alpha}}{n_{d,k}+\alpha} \binom{n_w,k+\beta}{n_w,k+\beta}}{\sum_{d} \sum_{z} \sum_{w} \binom{n_{d,z,k+\alpha}}{n_{d,k}+\alpha} \binom{n_{w,z,k+\beta}}{n_{w,k}+\beta}}
\]

(2)
where the \( nd \) is a count that does for document \( d \) assigns topic \( k \), \( nw \) is a count that does for word \( v \) assigns to topic \( k \). So the first ratio expresses the probability of topic \( k \) under document \( d \) while the second ratio expresses the probability of word \( v \) under topic \( k \). We also write them into equation 3 and 4.

\[
\theta_{d,k} = \frac{(nd_{d,k} + \alpha)}{\sum_{d} ndsum_{d} + K\alpha}
\]

\[
\phi_{v,k} = \frac{(nw_{v,k} + \beta)}{\sum_{v} mw_{v,k} + V\beta}
\]

where the \( \theta \) is the probability of a topic given a document and \( \phi \) is the probability of a word given a topic.

### 3. Parallel LDA

We used two popular distributed programming frameworks: an iterative MapReduce framework: Twister(J.Ekanayake, H.Li et al. 2010) and message passing interface written in java: MPJ Express(Baker, Carpenter et al.). The reason for choosing Twister is that Twister is an enhanced MapReduce runtime with an extended programming model that supports iterative MapReduce computations efficiently, and the LDA gibbs sampling is done in iterative pattern and it will be much faster to be parallelized under Twister which is specially designed for it than normal MapReduce framework like Hadoop. And MPI has always been a standard parallelism method for years. By using it we can have a control group where there will be a comparison between these two technologies.

#### 3.1 Parallel Gibbs Sampling

For parallel Gibbs Sampling, first assume we have \( M \) documents \( \{W_0, W_1, W_2, ..., W_M\} \), each document has a word collections \( \{w_0, w_1, w_2, ... w_n\} \) where \( n \) is the number of words in that document. And we divided this \( M \) documents onto \( P \) processors, so each processor will have \( M/P \) documents, and \( N/P \) words where \( N \) is the total number of words in all the documents. However, when each processor processes the documents by using Gibbs Sampling, they need \( nd, ndsum, nw, nwsum \) and \( z \) this 5 matrixes. We also have notations that \( V \) is the vocabulary of the total words in all documents, \( K \) is the total topic number. So \( nw[V][K] \) is the matrix contains counts word \( v (v = 1 ~ V) \) is assigned to topic \( k (k = 1 ~ K) \), \( nd[M/P][K] \) is the matrix contains counts document \( m (m = 1 ~ M/P) \) is assigned to topic \( k \), \( nwsum[K] \) is the matrix contains count of words assigned to topic \( k \), \( ndsum[M] \) contains the number of words in document \( m \), and \( z[M/P][D] \) is the matrix contains information of word in document \( m \) located in the place \( d (d = 1 ~ D) \) assigned topic where \( D \) is the number of words in document \( m \).

From the notations above, we can see that the \( nw \) and \( nwsum \) are two global matrix and \( nd \), \( ndsum \) and \( z \) are local matrix. So in each iteration, we need to merge \( nw \) and \( nwsum \) and keep \( nd \), \( ndsum \) and \( z \) updated in local until we compute the final result. And we also found out the \( nwsum \) can be computed directly from \( nw \), so the only matrix we need to pass from local to global is \( nw \).

#### 3.2 Twister-LDA

To use the iterative feature of Twister, the iterative part of the algorithm needs to be done within the program control of the iteration. Figure 2 shows the structure of the parallelism of the program: Twister-LDA.

![Figure 2 Structure of Twister-LDA](image)

There are two main parts of this program, the first part is the initialization, where each mapper reads the document and generates the data matrixes we need; the second part is Gibbs sampling which is an iterative part of the program.
In the first part, as shown in Figure 3, for each mapper, the documents has been split up into P chunks where P is the number of mappers, and each mapper reads the M/P document and generates the nd, ndsum and z locally and write them into a text file saved locally. Then each mapper will passed the word associate with the topic assigned to them passed to the reducer. The reducer will read the information, and then combine them into a global nw and generates a HashMap which contains all the vocabulary.

![Figure 3 Structure of Initialization for Gibbs Sampling](image)

In the second part, as shown in Figure 4, the combiner will broadcast the nw to each mapper, and at very first iteration, each mapper will read nd, ndsum and z into memory cache to do the gibbs sampling. After that, each mapper will just update the nd and z inside the memory cache without loading it again. This can save a lot of time compare to the current Hadoop framework. And each mapper will send out nw to the reducer and the reducer will merged the nw together using equation 5:

\[
\frac{nw_{\text{new}}}{P} = \frac{nw_{\text{original}}}{P} + \sum_{p=1}^{P} (nw_p -nw_{\text{original}})
\]

where we need to keep the \(nw_{\text{original}}\) on the reducer, and in the next iteration, the \(nw_{\text{new}}\) calculated this time will become the \(nw_{\text{original}}\).

![Figure 4 Structure of Gibbs Sampling using Twister](image)

When the main program meet the requirement to terminate the iterations, the mapper will send the nd, ndsum to the reducer and combine them into a large nd and ndsum to do the final calculation for \(\theta\) and \(\phi\) using equation 3 and 4.

3.3 MPJ-LDA

MPJ Express is a MPI extension of Java implementation. It is capable to run on distributed machines and multicore processors. The usage of it is also daemon control similar to Twister and with command line launcher.

Our implementation is also similar to Twister, each process has two roles: commander role and worker role. Each role is splitted by rank ID. To save space and speed up calculation, we also used partial data transportation as shown in Figure 5, that for each iteration, commander will extract a sub list from global list only the portion that will be used by the local worker.
Basically in MPJ-LDA, the initialization part is almost identical to Twister-LDA, shown in Figure 6:

The WP will send local word sequence and corresponding nw to C, compare to Twister that only sends the <word, topic> key-value pair to reducer. And Commander is also responsible to merge local result to global one.

In the second part, the MPJ-LDA does the gibbs sampling in a similar way to Twister-LDA, too. For each iteration, C send translated global nw and nwsum to each WP; On each WP, local nw will be updated by doing Gibbs sampling on each word of each document; Commander is responsible for receiving and merging local results to global one. And the structure is shown in Figure 7:
4. Performance and result analysis

4.1 Test environment

We use a data set which contains 1740 documents, 978381 words and the vocabulary size is 19889. The cluster we used is Polar Grid. Basically we have tested our AD-LDA algorithm on 1 ~ 8 processors on 1 ~ 8 nodes. For Twister, we have tested it on 1 node using 1 ~ 8 mappers and on 1 ~ 8 nodes but 1 mapper per node; For MPJ, we have tested it on 1 node using 1 ~ 8 processors.

4.2 Perplexity Result

At first, we want to test our result to see if it is accuracy or not. So we used the method to calculate perplexity which is introduced from (Newman, Asuncion et al. 2007) as in equation 6.

$$\text{Perplexity}(D) = \exp \left\{ - \frac{\sum_d \log \sum_k \theta_{d,k} \phi_{v,k}}{N} \right\}$$

where the D is the test documents and $\theta_{d,k} \phi_{v,k}$ can be found in the results and N is the total number of words in the test documents. And the results for our AD-LDA using 4 processor and 16 processors are shown in Figure 8 with comparison from sequential LDA:

![Figure 8 The x axis is the number of iterations and y axis is the perplexity](image-url)
Basically the perplexity is used to monitor the quality of the probabilistic model, and the lower the perplexity is, the better the model is. And from what we have observed from Figure 8, the ADLDA has the performance as good as the sequential LDA, so it demonstrate that our program runs correctly.

4.3 Speedup result

The performance of Twister-LDA and MPJ-LDA are similar as shown in Figure 9.

The performance of Twister-LDA and MPJ-LDA is not as good as expected, but we still can explain it. First of all, the computation versus communication: In each iteration, we need to broadcast nw in both implementations, where the size of the matrix is V * K; and inside each processor, the computation is O(N/P * K) = O(N*K/P) where the N*K won’t change, but the time that the computation needed is decreased as P increases. However, on the other hand, the communication size is V * K *P where the communication increases as P Increases. So the communication/computation = N/V * P^2. And in our case, the N is 978381, and V is 19889, so the communication overhead is much more than the computation time when P increases above 4.

5. Conclusion and future work

The Twister-LDA and MPJ-LDA are correctly implemented and similar to the method used by (Wang, Bai et al. 2009). The result is not as good as we expected when we start the project but it is still valuable. So our future work is to increase the size of our test documents and optimize our program to see if the speedup chart would be better.

References:


