Parallelization of Variational Inference for Bayesian Nonparametric Topic Models

Sonia Phene

A thesis submitted in partial fulfillment of the requirements for the degree of Bachelor of Science in Computer Science

at

Brown University

Supervisor: Professor Erik Sudderth
Reader: Professor Stefanie Tellex

April 2015

Acknowledgments: I would like to give a tremendous thank you to Michael Hughes and Erik Sudderth for their valuable insights, help, and advice.

Contents

1 Introduction **4**
1.1 Abstract . . . 4
1.2 Topic Modeling . . . 4
1.3 Variational Bayes . . . 6
1.4 bnpy framework . . . 7

2 Parallelization **8**
2.1 Motivation . . . 8
2.2 In Theory . . . 8
2.3 In Code . . . 9
2.3.1 Algorithms . . . 9

3 Results **11**
3.1 General Models . . . 11
3.2 Improvements . . . 11
3.2.1 Experiments on the E-step . . . 12
3.2.2 Improvement within the Pipeline . . . 13
3.3 Performance on datasets . . . 13
3.3.1 Toy Bars . . . 13
3.3.2 Wikipedia . . . 15
3.3.3 Science . . . 15
3.3.4 New York Times . . . 15

4 Conclusion **20**

Bibliography **21**

Chapter 1

Introduction

1.1 Abstract

We explore how to speed up topic modeling, a method that discovers hidden themes of documents, by parallelizing the expectation step (E-step) in variational inference. We conducted our experiments by contributing to the development of the open-source Bayesian nonparametric Python, bnpy, framework. Since the E-step is a significant portion of the variational inference algorithm and can be performed on each document independent of the others, we were able to achieve significant speed-ups by distributing this task to multiple workers. We evaluated our code on the Toy Bars, Wikipedia, Science, and New York Times datasets.

1.2 Topic Modeling

Topic modeling identifies the hidden topics or main ideas that are present in a set of documents. For example, the themes of scientific journal publications could be subjects such as neuroscience or chemistry and the idea of newspaper articles could include topics like politics or sports. Topic models work by analyzing the text in original documents, discovering the themes within them, and then categorizing documents into the themes without any firm labels (though some of them do incorporate information from metadata). Topic models discover the 'essence' of a document, much as a human reader would, but they do so at enormous scales.

As increasing amounts of information are posted online, there is a growing base of documents available for analysis. However, in order to make sense out of this data, we must be able to group and clearly understand what themes are present. Although many modern-day articles may be uploaded with tags that make it easier to find information, many historical records do not have such tags and even current documents are often uploaded without any summary information.

Due to the tremendous amount of data available, it would not be possible for humans to read everything and process each document. As a solution, statistics can be used to take in the words of a document and

draw insights over what theme is present in each document and how the documents relate to each other. These documents do not require any prior annotations [2].

The format of having some unknown topics and observed words lends itself to a hidden model structure where the documents contain observations of words, which come from some number of hidden topics. We model topics by assuming they are a distribution over a set of vocabulary (for example, the topic of computer science would have a high probability of words such as algorithm, runtime, and memory but low probability for words such as cat or water). If we knew the topics in advance, we could simply calculate probabilities for each of the documents given the words in them. Since we do not, we use topic modeling to reverse the generative process to determine the basis of topics. For Latent Dirichlet allocation, there is the assumption that all topics have the same vocabulary but different distributions over the words in it [2].

In considering the model let's look at the following variables:

- V is the vocabulary of words. Each word, x, is a V-dimensional vector where $x^v = 1$ represents the vth word.
- A document is described by $\mathbf{x}$=$(x_1, x_2, ...x_N)$
- M documents make up the corpus or collection of interest where D=($\mathbf{x_1}$,$\mathbf{x_2}$,...,$\mathbf{x_M}$)
- $\phi_{1:K}$ describes the topics where each ϕ_k describes the distribution over a vocabulary
- π_d is the distribution that describes the proportion of topics for the dth document and $\pi_{d,k}$ describes the topic proportion for topic k in document d.
- z_d is the topic assignments for the dth documents and is a vector of size N for the number of words in that document.

We assume a model where words are chosen in documents through a generative process. This process works under the assumption that every document was chosen to have a mix of topics from a global distribution of topics. Every word in document d was assigned a topic from the distribution π_d. The actual word is also influenced by ϕ_k the distribution over the topic assignment.

We can use the joint distribution to compute the posterior distribution, the probability of the hidden variables given the observations [3].

The posterior distribution, which is the distribution of interest to us is described by Eq. 1:
$p(\phi_{1:K}, \pi_{1:D}, z_{1:D}|x_{1:D})$=$\dfrac{p(\phi_{1:K}, \pi_{1:D}, z_{1:D}, x_{1:D})}{p(x_{1:D})}$

The joint distribution can be calculated by using Eq 2:
$p(\phi_{1:K}, \pi_{1:D}, z_{1:D}, x_{1:D})$=$\prod_{i=1}^{K} p(\phi_i) \prod_{d=1}^{D} p(\pi_d)(\prod_{n=1}^{N} p(z_{d,n}|\pi_d)p(x_{d,n}|\phi_{1:K}, z_{d,n}))$ This equation is derived from the Bayesian network as shown in figure 1.1. [5]

In calculating the posterior distribution, it is easy to compute the joint distribution. However, in order to calculate $p(x_{1:D})$, it would be necessary to sum the joint distribution over every possible variation of topic structure, which quickly becomes too large to solve. Instead, these methods can be solved through making approximations for the posterior, which we do by using variational inference described below.

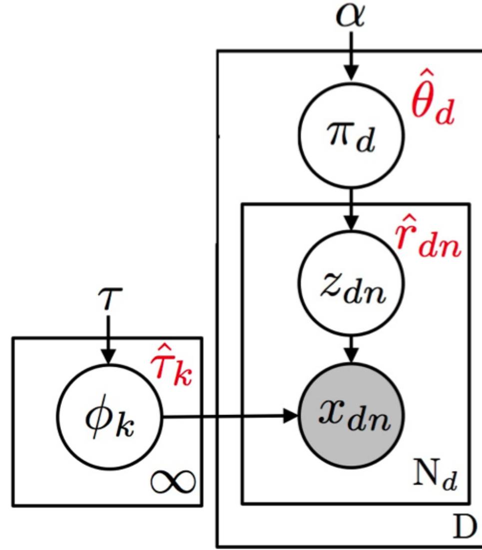


Figure 1.1: Bayesian network for topic modeling

1.3 Variational Bayes

Topic models attempt to find the posterior distribution through sampling or variational algorithms, the latter of which is the focus of the work in this thesis.

The former method, also known as a Monte Carlo technique, uses numerical samples to approximate the distribution. For the graphical model, the Markov Chain of hidden variables is run to collect samples from the distribution and then empirically estimate it. Gibbs sampling is used for computationally intensive, undirected graphs. This approach works by collecting enough data to make estimates of parameters that define the distribution [1].

Our work employs the use of variational Bayes, a method that estimates the posterior distribution by by optimizing the distribution for the hidden structure (from some set of distributions). It can also be used to determine the lower bound of the marginal likelihood of the observations given the model.

Since the denominator of Eq. 1 is intractable with large values, variational Bayes attempts to determine $q(z_{1:m}|v)$ to approximate $p(Z|x)$ where v is a set of parameters on the new distribution. q is chosen from a fixed family of distributions where it is as close to the real posterior as possible. To measure this difference indirectly, the Kullback-Leibler or KL divergence is used where

$KL(q||p)$=$E[\log \frac{q(Z)}{p(Z|x)}]$

=$E[\log q(Z)]$-$E[\log p(Z|x)]$

=-($E[\log p(Z,x)]$-$E[\log q(Z)]$)+$\log p(x)$

Using Jensen's inequality, we can find that the ELBO is $E[\log p(Z,x)]$-$E[\log q(Z)]$, so the KL divergence is just the negative ELBO and the sum of the log marginal probability of x. Since q does not depend on $p(x)$, we can simply choose to focus on the first two terms and maximize the ELBO.

In order to maximize the ELBO, coordinate ascent is used to iteratively optimize each variational distribution while holding the others constant. In the experiments in this thesis, we ensured the number of iterations were at a constant 100 iterations across trials by setting the convergence threshold too low to converge before that. This was done in the interest of having a fair comparison with equal numbers of iterations across trials.

Similar to Expectation-Maximization algorithm, there is an alternating structure of updating the model and beliefs and allowing them to feed back into one another.

1. Local step: update the beliefs of $z_{1:n}$ given the model parameters. That is, update assignments for words in each document given our belief about the topics and the proportion of each topic present in a document.

2. Summary step: Compress all of the knowledge from $z_{1:n}$ into N=[$count_1$...$count_K$], which computes the counts of the assignments across the documents. The summations across all the documents informs the topic distributions and mixture present.

3. Global step: Update thoughts about the global model parameters: τ, p, and w. Given the new assignments from the observations, update the model parameters.

1.4 bnpy framework

bnpy is a Python framework that supports Bayesian nonparametric machine learning. The goal of bnpy is to provide scalable variational learning algorithms. Bayesian nonparametric models consider all possible solutions for a problem, which implies an infinite parameter space. These models are solved by looking at finite subsets of data, which are determined from the observations. The main advantage here is that the model is allowed to adapt to the complexity of the data [7].

One major area of the bnpy framework is topic models, which is the focus of this thesis. For topic models in particular, Bayesian nonparametric models allow new topics to arise from input documents. However, it is important to bear in mind that the similarities in algorithms for mixture and topic models make it easy to apply the advances from this thesis to mixture models as well.

bnpy supports a variety of inputs such as the data-generated model for the likelihood including multinomial and Gaussian as well as a series of variational inference algorithms such as expectation maximization and memorized or stochastic variational Bayes. As a brief overview, stochastic online variational Bayes uses estimates of posterior distributions to apply complex Bayesian models at scale. However, it has the caveat of converging to a local rather than a global optima. Memoized online variational Bayes is a recently developed alternative. Memoization speeds up algorithms by caching complex function calls. It works on data in batches, which is useful for datasets that are too large to store in memory, such as the New York Times dataset. Memoized online variational Bayes stores sufficient statistics from the data sets, allowing it to scale to millions of examples [4].

In addition, bnpy also supports parameters on the data such as the number of documents examined, the words per document, and the number of topics considered. We used these parameters in order to run many of the experiments reported below. The speedups from parallelizing the framework were only apparent at larger amounts, which is fitting since the ultimate goal of parallelization is to improve performance on large datasets that previously would take days or weeks to run.

The code for the work done in this thesis can be found in the open source bnpy software framework at https://bitbucket.org/michaelchughes/bnpy-dev/wiki/

Chapter 2

Parallelization

2.1 Motivation

Due to the growing number of large datasets available, new methods must be developed in order to process all the information. Recent research has focused on ways to improve topic modeling. Zhai et. al applied map reduce to the latent Dirichlet allocation (LDA) in order to process massive numbers of documents [10]. Another method that uses Gibbs sampling to approximate LDA was developed by Newman et. al [6]. Furthermore, a few large-scale machine learning frameworks such as Mahout and GraphLab have tools to handle topic modeling.

A fundamental principle of the bnpy framework is that it can be used for large-scale machine learning algorithms, so it is important that it is capable of quickly processing an enormous corpus of data. We parallelized this algorithm so that bnpy can realistically handle more information.

2.2 In Theory

As described in section 1.3, variational inference relies on three main steps: a local step, summary step, and global step. In the local step, each document is independently viewed and the assignments of words to topics are made. In the summary step, these topics and words are summed up. Finally, the global beliefs are updated which can again feed back into the initial step. The local and summary steps can be executed in parallel since computing probabilities of each document only relies on the global parameters. While it is of course true that documents influence one another by feeding into global beliefs, computations of assignments can be done independently in the local step.

Furthermore, the computations in the local and summary steps take up the vast majority of the pipeline, up to 98% of the overall runtime. Parallelizing this part of the algorithm can lead to significant speed-ups.

In the parallelized variational Bayes framework that we have written, the local and summary steps are distributed to multiple workers. Each worker performs the local steps and then proceeds to compute the summary for the batch of documents it was working on. Then, there is a join where all of the sufficient

statistics are summed up. Once all of this information has been aggregated, the global step can be performed.

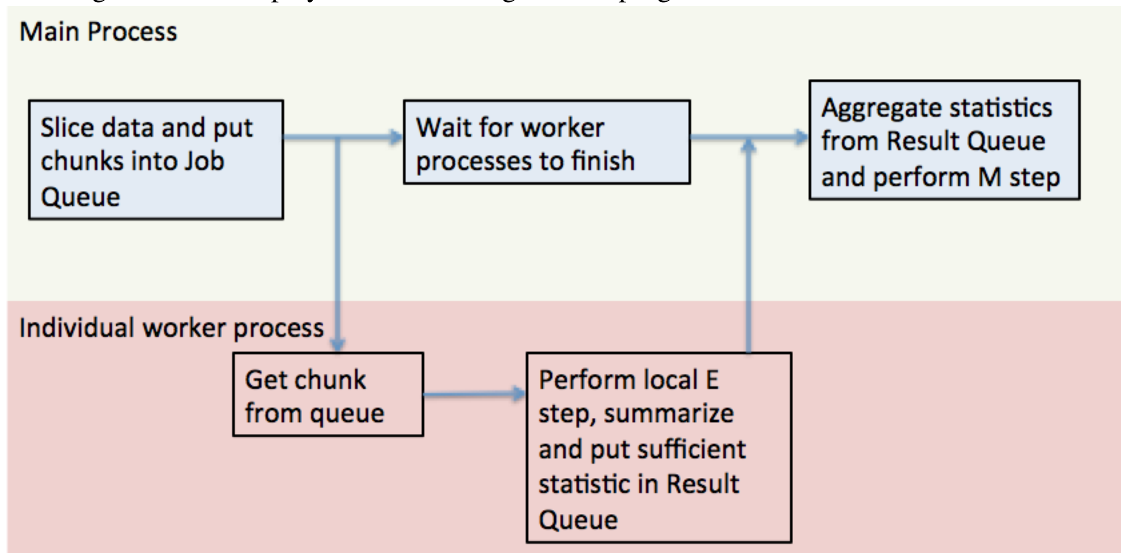
2.3 In Code

The bnpy code lends itself to being parallelized since there are a number of attributes such as the document topic count and prior belief about topics which are independent of the other documents. We capitalized on the fact that each row in these matrices can be written to in parallel without any lock since rows do not depend on nor write to one another. This means that each document could, in theory, be executed entirely in parallel. Practically, the logistical overhead of creating and destroying processes makes this slower than a single-threaded implementation. However, the benefit of executing code asynchronously exists when the appropriate number of processes can be determined.

In order to parallelize the Python code, we chose to move forward with the `multiprocessing` module. This framework is a standard library that is supported and maintained in Python directly and requires no additional packages or installation. Furthermore, it was important to get around the Global Interpreter Lock (GIL) in Python, which is a mutex that prevents multiple Python threads from executing at once because Python in general is not safe [9]. This boils down to meaning that even with a "multithreaded" program in Python, the GIL prevents threads from executing asynchronously.

In order to execute code in parallel, we instead created new processes, which are not as lightweight as threads and have more of a cost to create. However, once these processes are running. They operate faster than threads. In addition, the use of the `multiprocessing.Process` class allowed us to distribute calls truly asynchronously. The data of relevance was stored in shared memory to prevent duplicate copies from having to be created and passed around.

The diagram below displays the flow throughout the program.



2.3.1 Algorithms

The local, summary, and global steps are performed as described in section 1.3. In our framework, the local and summary steps are distributed out to multiple workers, the results are all then entirely joined, and finally the global update is performed.

Algorithm 1 describes what is happening in the master process. In our code, the Job Queue manages the parameters for all the tasks that must be completed. The Result Queue keeps track of all the summary statistics for the given batch of documents. After all the jobs are finished, it is the sum of the sufficient statistics that is used to compute the global parameters: τ, p, and w.

```
   input  : The model used in the learning algorithm
   output : The sufficient statistics over the entire corpus
 1 foreach slice in SliceGenerator do
 2 |   SliceArgs ← GetSlice;
 3 |   AddToJobQueue(SliceArgs);
 4 end
 5 // Pause until all jobs are completed
 6 JoinOnJobQueue // Combine results from all workers
 7 SuffStat ← GetFromResultQueue;
 8 while ResultQueue is not empty do
 9 |   SuffStat + =GetFromResultQueue;
10 end
11 return SuffStat
```

Algorithm 1: Distributing jobs to workers

Once given the parameters that describe which chunk of data to work on, each worker can perform the local and summary step. The observation and allocation models update the local parameters (in `CalcObsLocalParameters` and `CalcAllocLocalParameters`), which means that they compute the assignment of each word to the topic, $z_{1:n}$. The summary statistics computed below are the sum of the words for each topic for the given documents for each worker, compressing the knowledge from $z_{1:n}$ into N=[$count_1 ... count_K$] in the methods `CalcObsSummaryStats` and `CalcAllocSumaryStats`. Algorithm 2 describes what each of the processes does.

```
   input  : None, receives input arguments by iterating over JobQueue
   output : None, puts the sufficient statistics into the ResultQueue
 1 foreach slice from JobQueue do
 2 |   Start, Stop ← SliceArgs;
 3 |   DataSlice ← GetChunkOfData(Start,Stop);
 4 |   // Perform local step
 5 |   Likelihood ← CalcObsLocalParameters(DataSlice);
 6 |   Likelihood ← CalcAllocLocalParameters(Likelihood,DataSlice);
 7 |   // Perform summary step for this chunk
 8 |   SummaryStat ← CalcObsSummaryStats(Likelihood);
 9 |   SummaryStat ← CalcAllocSummaryStats(SummaryStat);
10 |   // Notify Queues
11 |   NotifyJobQueueTaskDone;
12 |   AddToResultQueue(SummaryStat);
13 end
```

Algorithm 2: The run function for each of the process workers

Chapter 3

Results

3.1 General Models

The goal of topic models is to find the hidden themes in a document. The bnpy pipeline saves vocabulary distributions over different topics. To make that concrete, let us consider topic distributions from the Science dataset (described in more detail below). Running variational Bayes on Science articles yields several topics. For example, here are the the highest probability words from three of them: Topic A has a distribution of .015 molecules, .011 dna, .009 years, .009 carbon...; Topic B has a distribution of .011 changes, .009 university, .007 science, .007 time, .007 region...; Topic C has a distribution of .015 cells, .012 work, .011 model, .008 disease, .007 immune. In general, the distributions provide insights over what topics exist in the document. Documents themselves can be assessed by looking at the topics that yield highest probability from the words present.

Another example can be seen from the five most probable words for four different topics from New York Times articles.

Topic 1	**Topic 2**	**Topic 3**	**Topic 4**
health	school	environmental	campaign
care	students	power	party
medical	schools	energy	republican
hospital	education	plant	election
patients	college	nuclear	democratic

3.2 Improvements

We were able to achieve significant speedups from parallelizing the E step in the algorithm. Overall, this is the portion of the code that tends to take the longest amount of time. On the Variational Bayes algorithm, this can take up to 98% of the overall runtime, which makes it an extremely valuable portion of the code for us to parallelize.

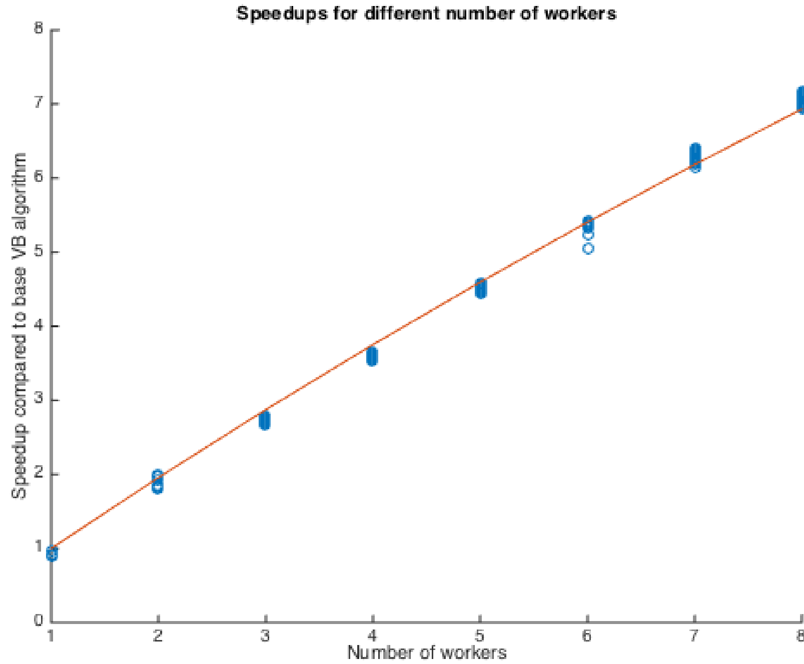


Figure 3.1: Speedup found for isolated tests of parallelized code

Amdahl's Law describes the maximum possible speed-up, S, for n processors and p portion of code that can be parallelized. $S(n) = \frac{1}{1-p+\frac{p}{n}}$ [8]. The greater p is, the higher the speed-up. Since it is difficult to know the true portion of the code that can be parallelized as it varies based on dataset size and number of topics, p can be approximated through running a regression on the runtimes we observed. In figures 3.1-3.10, the red line represent the predicted times or speedups as would be found using Amdahl's law. The actual data points are always represented as individual blue circles.

All of the experiments reported were run on a machine with an Opteron 6282 SE CPU and 64 GB of RAM on the Brown CS grid with 8 cores available.

3.2.1 Experiments on the E-step

First, we examined our results looking simply at the time it took to calculate the local parameters and summaries. That is, we created a new test class that simply ran this portion of the code and did not fully integrate with the overall pipeline.

This test class creates the job queues, randomly initializes a set of data, and then distributes it out to the workers. The overall time includes the task of running by each of the individual workers as well as setting up and tearing down the processes. For this reason, the code cannot be entirely parallelized and we can determine the overall speedup using Amdahl's law and considering the overhead involved. Running a regression on this data, shows that p=.978, which means that nearly all of it can be parallelized.

In figure 3.1, the speedup in 22 trials for 1-8 workers is shown and the absolute time is recorded in figure 3.2. As evidenced by the close clustering of the data, there are fairly consistent results in terms of the speedup provided with very little variation.

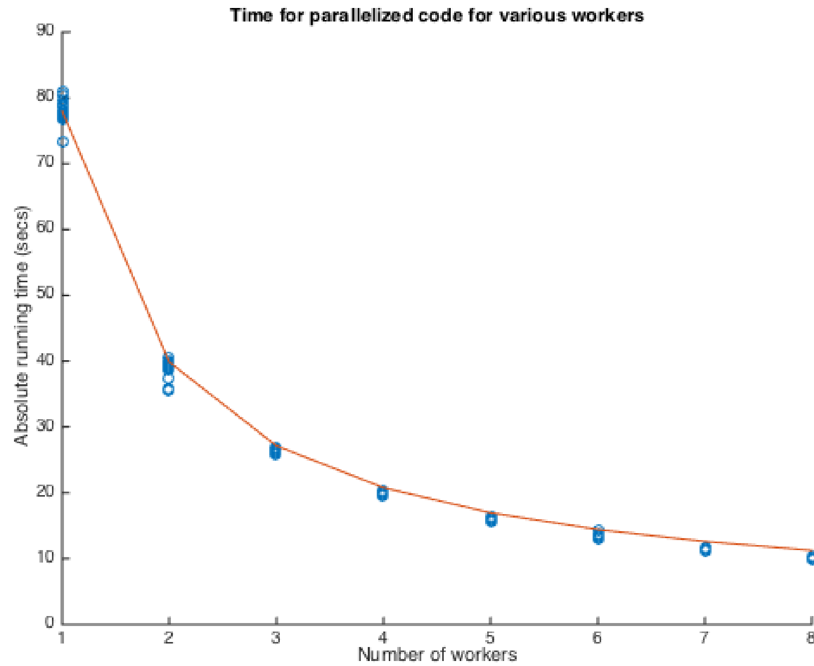


Figure 3.2: Absolute time for isolated test runs of parallelized code

3.2.2 Improvement within the Pipeline

For experiments within the pipeline, we first ran the algorithm on the Toy Bars dataset. The Toy Bars dataset assigns one topic to every word and is generated from the LDA topic model bag of words data.

Figures 3.3 and 3.4 show the speedup and absolute time, respectively, for the function call to `calcLocalParamsAndSummarize` took in the overall pipeline. This was found by placing calls to `time.time()` within the Python code before the function ran and after it returned. This is a very similar plot to the experiments on the E-step, except that it considers the code in the context of the overall pipeline.

3.3 Performance on datasets

This section describes the performance on the Toy Bars, Wikipedia, Science, and New York Times datasets. For all of these sets, bnpy was run with the following parameters: three laps of iterations (meaning it performs the local, summary, and global updates three times), a converge threshold of .000001 and 100 iterations in the coordinate ascent step. For all of these sets except for New York Times, the experiments were performed with K=50 topics.

3.3.1 Toy Bars

We measured the overall runtime that it took on a Toy Bars dataset which was generated from 10 true topics and a vocabulary of 900 words. The experiment was run with 15,000 documents and 200 words per document. The Toy Bars dataset is a "toy" set of words and documents that we create, which allows us to specify the number of documents as well as words per document. Figures 3.5 and 3.6 show the speedup

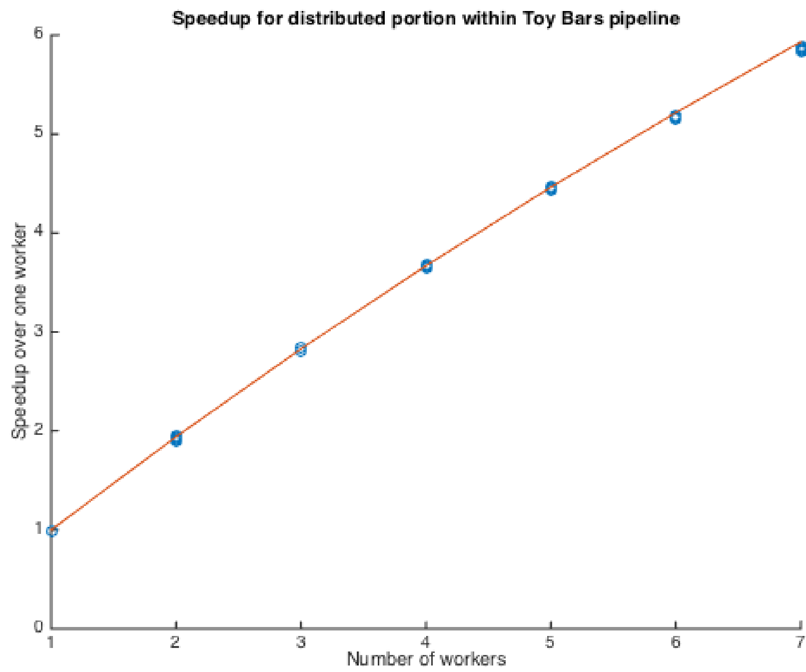


Figure 3.3: Speedup for measuring the time it took the parallelized code within the pipeline

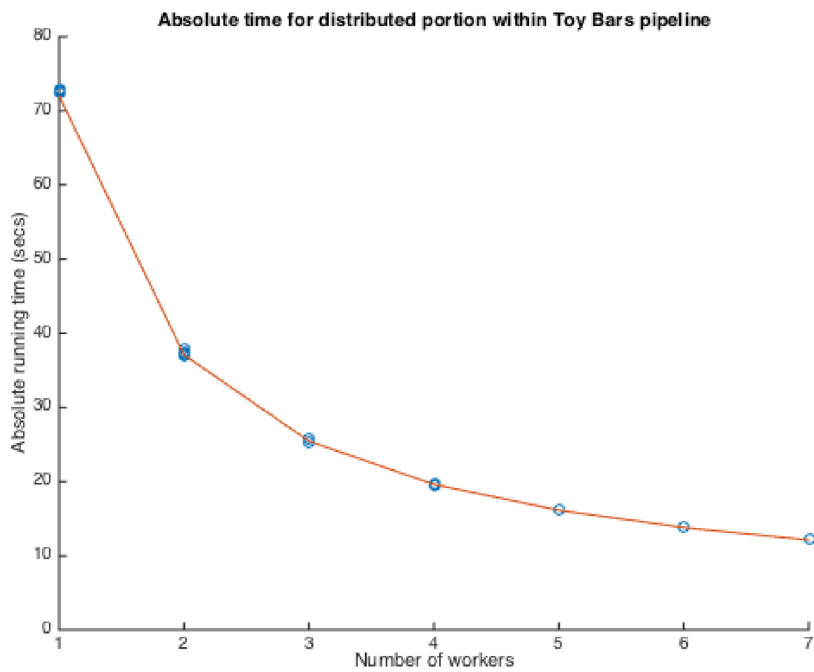


Figure 3.4: Absolute time for the parallelized code within the pipeline

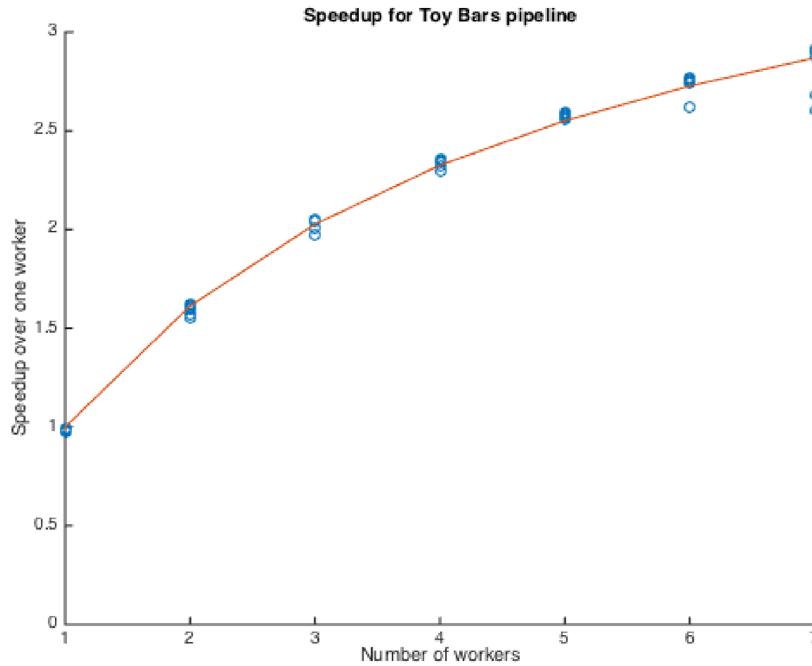


Figure 3.5: Speedup for trials with the Toy Bars dataset

and absolute time that was achieved by running with multiple workers. Running a regression and using Amdahl's Law, we find that p=.76. This experiment was run for 12 trials for 1-7 workers.

3.3.2 Wikipedia

We also ran experiments on a Wikipedia set, which consisted of 7,961 documents and a vocabulary of 6,131 words. Figures 3.7 and 3.8 show the speedup and absolute time for 12 trials of 1-8 workers. A regression shows that p=.87.

3.3.3 Science

The Science dataset had 13,077 documents and a vocabulary size of 4,403. Figures 3.9 and 3.10 show the speedup and absolute time for 12 trials of 1-8 workers. A regression shows that p=.91.

3.3.4 New York Times

The New York Times dataset was the largest set we ran our experiments on with two million documents and a vocabulary size of 8,000, which is so large that it needed to be run with the memoized version of the algorithm. There were 200 batches of documents, each of size 9084. Since the set is too large to store in memory, it must be loaded in batches from disk.

This yielded significant speed-ups where Figures 3.11 and 3.12 show the speedup and absolute time for 4 trials of 1, 4, and 8 workers. This was run for K=10 topics. Running a regression on the data, we find that p=.97, the largest of any set. As expected (and desired), the most significant improvements are seen with the larger datasets, which is also evidenced by the increasing p, the portion of code that can be parallelized.

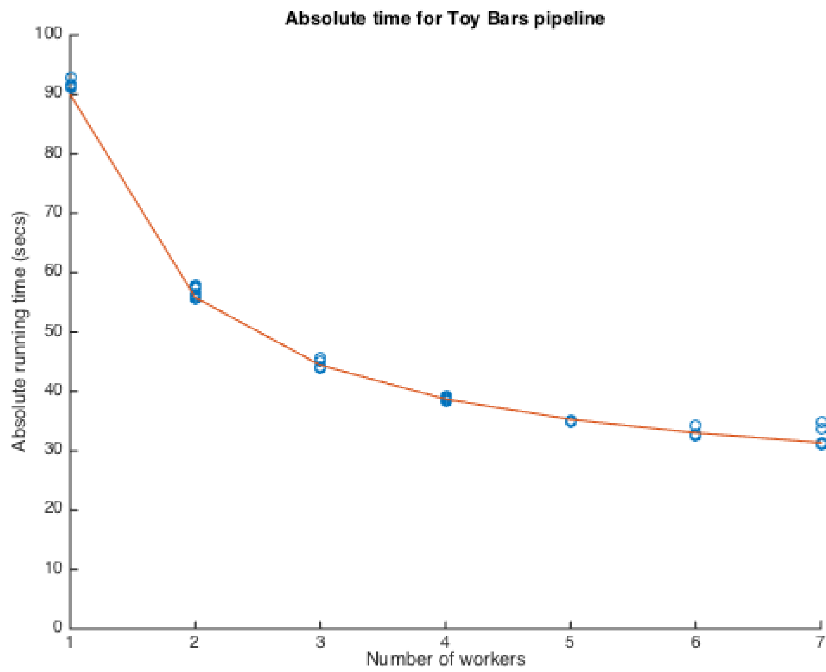


Figure 3.6: The absolute running time for the Toy Bars dataset

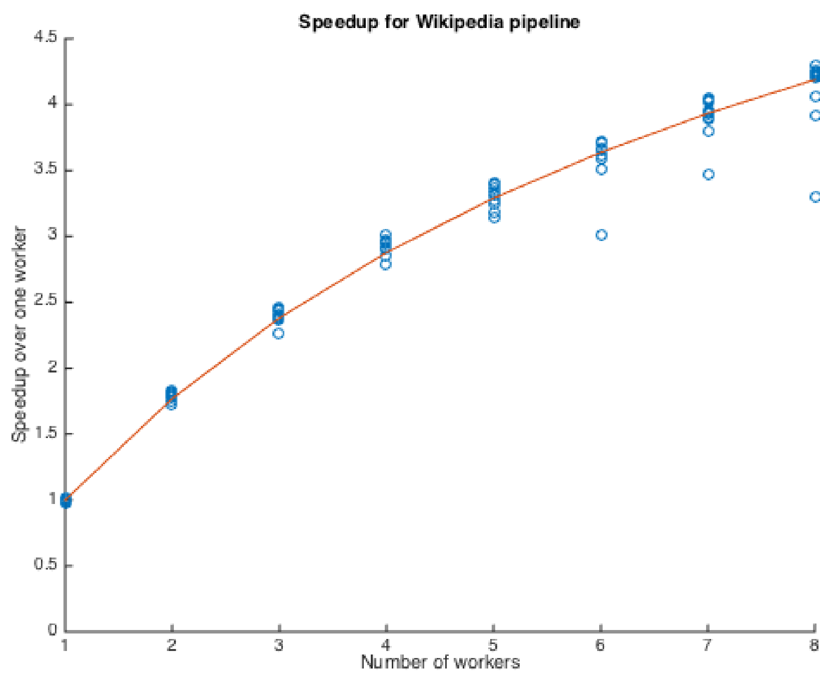


Figure 3.7: Speedup for trials with the Wikipedia dataset

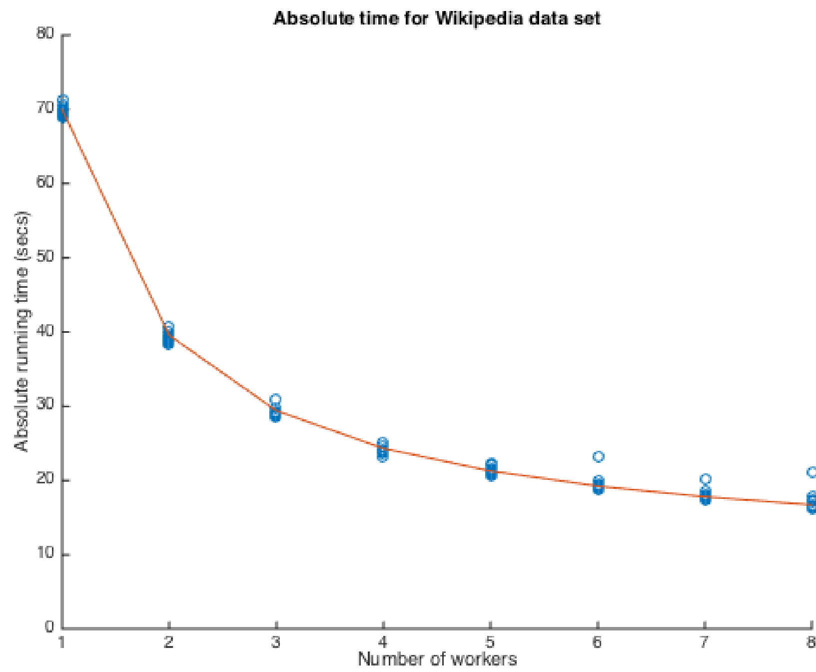


Figure 3.8: The absolute running time for the Wikipedia dataset

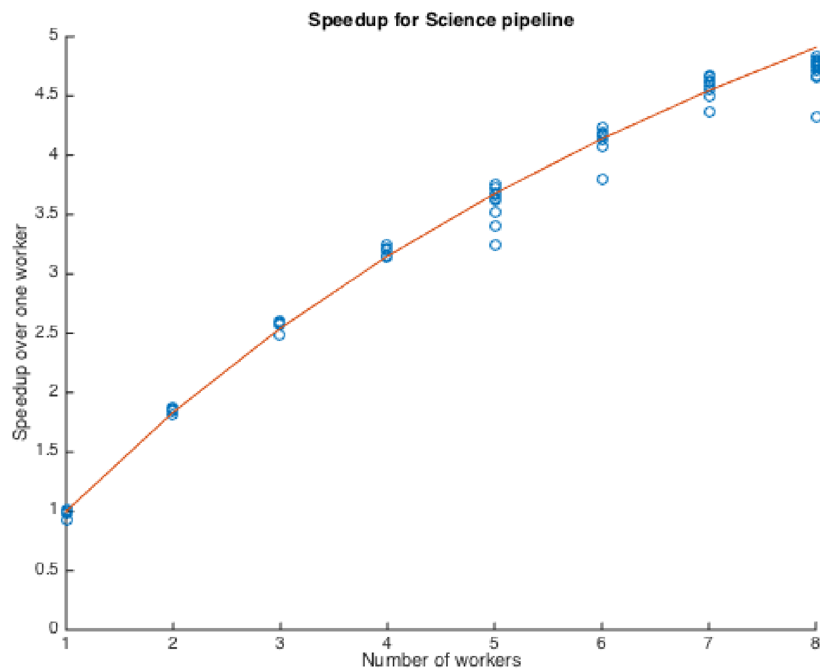


Figure 3.9: Speedup for trials with the Science dataset

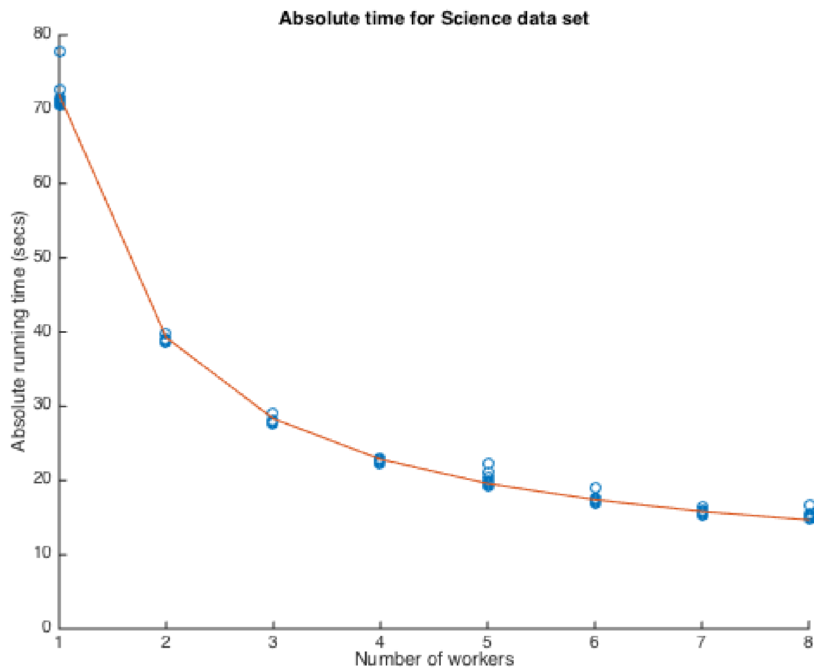


Figure 3.10: The absolute running time for the Science dataset

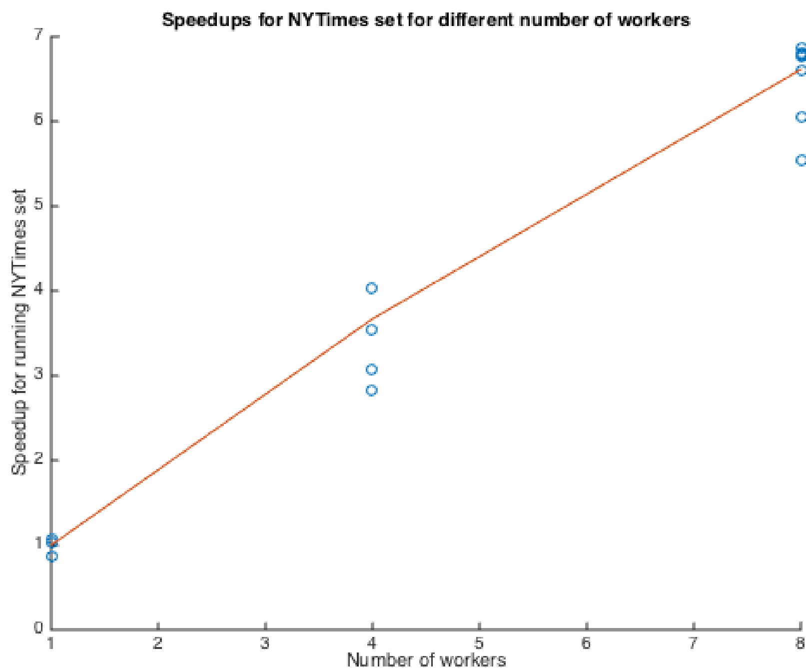


Figure 3.11: Speedup for trials with the New York Times dataset

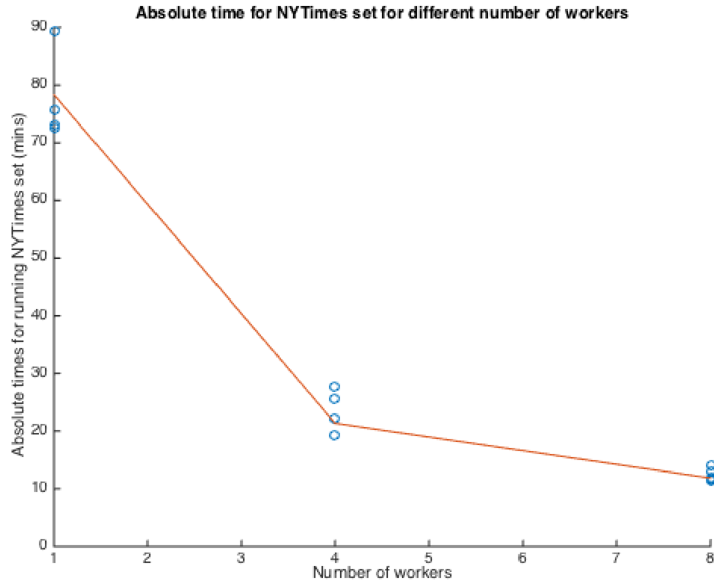


Figure 3.12: The absolute running time for the New York Times dataset

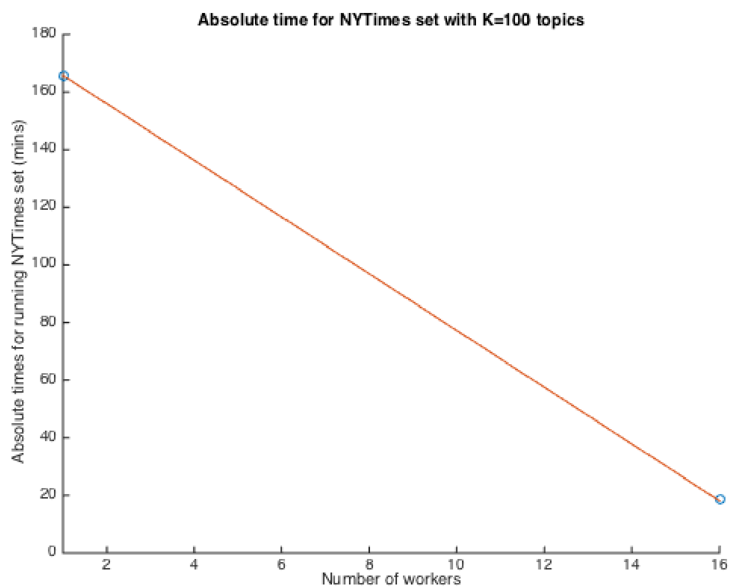


Figure 3.13: The absolute running time for K=100 topics on the New York Times dataset

In addition, we ran the experiment on the New York Times dataset with K=100 topics and used 16 cores in order to handle the magnitude of calculations. Using our above estimate of p=.97, we would expect to see a speed-up of 9.15 from Amdahl's Law. In our data, we observed a speedup of 9.02, which is very close to the prediction. The discrepancy can be explained by the fact that Amdahl's Law displays an ideal speedup and does not fully consider that additional processors can add more contention. Figure 3.13 shows the run time for two trials each of one and 16 workers.

Chapter 4

Conclusion

Overall we saw significant speedups with our code. The parallelization of the local step enabled us to run topic models much faster with the same level of accuracy and overall performance. The greatest improvement was seen on the largest datasets, which is where it was most important and desired. The local step grows in terms of $O(D * N_d * K)$, so as we increase the number of documents in the corpus or the number of words in a document, the local step becomes a more dominant portion of the pipeline. Increasing the fraction of the pipeline that can be parallelized leads to a greater speedup, which enables work to be done in a reasonable time frame on bigger datasets.

Given the growth of enormous corpora of information such as online encyclopedias (Wikipedia, being one) and uploaded historical records, parallelization means data can be processed more quickly and summarized. With the use of large cloud engines and servers, our code can process this information and provide insights that would have taken orders of magnitude more time. These results can also be applied to mixture models, which have a similar structure of alternating between local and global steps.

In summary, what used to take weeks or months can now be done in a matter of hours or days. Our code is among the first of its kind to use parallelized variational inference to implement topic models. These improvements allow for faster processing and utilizes the multiple cores on machines more efficiently to deliver timely results.

Bibliography

[1] Bishop, Christopher M., Pattern Recognition and Machine Learning, New York: Springer, 2006.

[2] Blei, David M., "Probabilistic Topic Models.", Communications of the ACM, 2012, 77., http://www.cs.princeton.edu/ blei/papers/Blei2012.pdf.

[3] Blei, David M., "Variational Inference." Lecture from Princeton, https://www.cs.princeton.edu/courses/archive/fall11/cos597C/lectures/variational-inference-i.pdf.

[4] Hughes, Michael C., and Erik B. Sudderth., "Memoized Online Variational Inference for Dirichlet Process Mixture Models.", NIPS, 2013., http://cs.brown.edu/ sudderth/papers/nips13dpMemoized.pdf.

[5] Hughes, Michael C., Dae Il Kim and Erik B. Sudderth., "Reliable and Scalable Variational Inference for the Hierarchical Dirichlet Process" AISTATS, 2015., http://jmlr.org/proceedings/papers/v38/hughes15.pdf

[6] Newman, David, Arthur Asuncion, Padhraic Smyth, and Max Welling., "Distributed Algorithms for Topic Models.", The Journal of Machine Learning Research, 10 (2009): 1801-828.

[7] Orbanz, Peter, and Yee Whye Teh., "Bayesian Nonparametric Models.", Encyclopedia of Machine Learnin, 2010, 81-89. http://stat.columbia.edu/ porbanz/papers/porbanz_OrbanzTeh_2010_1.pdf.

[8] Padua, David A., Encyclopedia of Parallel Computing., New York: Springer, 2011.

[9] Python Wiki, https://wiki.python.org/moin/GlobalInterpreterLock.

[10] Zhai, Ke, Jordan Boyd-Graber, Nima Asadi, and Mohamad Alkhouja., "Mr. LDA: A Flexible Large Scale Topic Modeling Package Using Variational Inference in MapReduce.", WWW ’12, 2012, 879-88.