Parallel LDA: Final Report

Angelica Feng, Judy Kong
zhixinf/junhank@andrew.cmu.edu
Project ID: 28
May 9th, 2018

Summary:

We introduced parallelism to the Latent Dirichlet Allocation topic model used in text classification. By distributing the corpus, running local sampling, and synchronizing updates with message passing, we achieved a maximum speedup of 9x compared to the sequential version. We demonstrated that our parallel LDA implementation had good performance both in terms of speedup and in terms of the objective function. We also implemented a few different variations of parallel LDA and compared their performances.

Background:

Describe the algorithm, application, or system you parallelized in computer science terms.

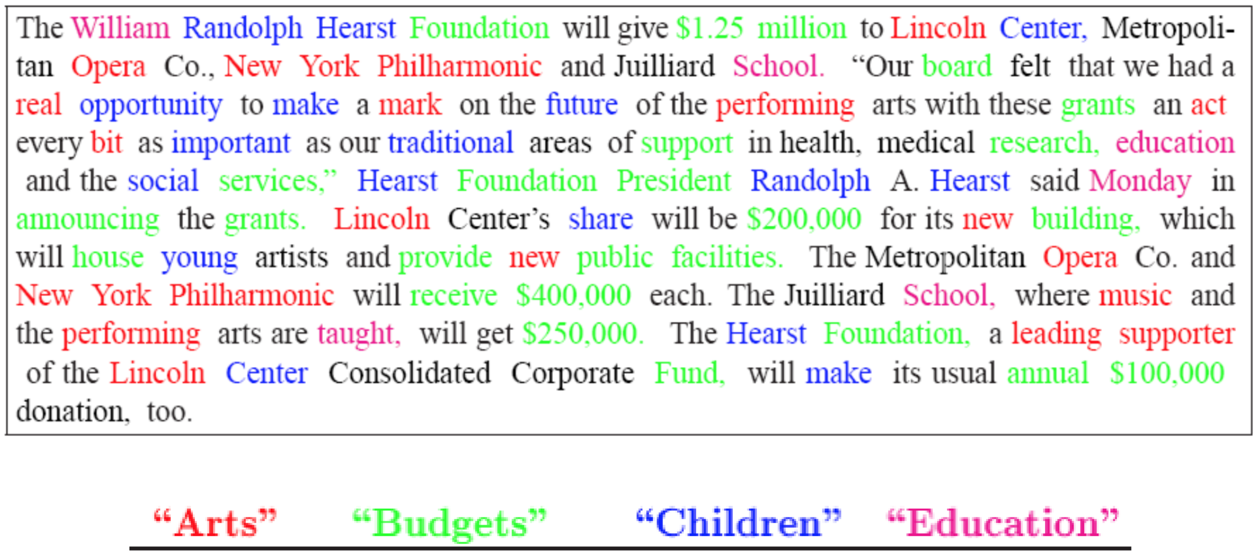
Text classification has always been an interesting topic of discussion in the field of machine learning. However, the massive data sets have always been a big challenge in text classification - each document might contain a big number of words already and there might be millions of documents in a big corpus. The training process usually requires a large number of iterations of parameter learning as well. The computation power needed is huge.

Latent Dirichlet Allocation is a widely used algorithm in text classification. The LDA topic model clusters word occurrences into latent classes (i.e. topics). LDA calculates the topics of each word based on different class distributions over each document, and Gibbs sampling has been widely used for parameter learning in LDA.

What are the algorithms inputs and outputs?

Latent Dirichlet Allocation takes in a set of D documents as its input, where each document contains a set of words (usually in the form of numerical labels of the words). In addition to the documents needed for training, LDA also needs the number of topics to classify the corpus into, the number of iterations to train for, and the values of alpha and beta parameter which are used for smoothing during training.

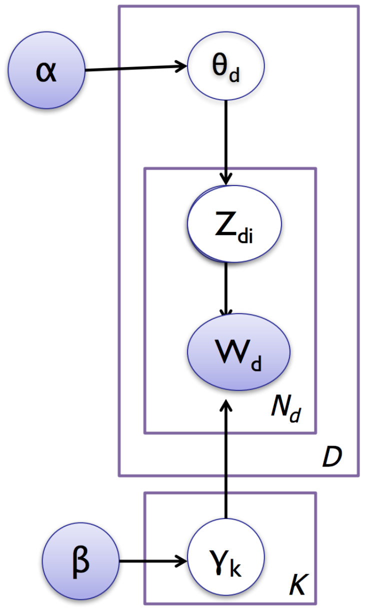
Because LDA is a method of unsupervised learning, it does not need the true labels of the input documents. While the total number of documents and unique words may be obtained through a traversal through the training documents, it is often useful and more efficient to have these 2 numbers ready as additional inputs to the LDA algorithm as well.



The William Randolph Hearst Foundation will give $1.25 million to Lincoln Center, Metropolitan Opera Co., New York Philharmonic and Juilliard School. "Our board felt that we had a real opportunity to make a mark on the future of the performing arts with these grants an act every bit as important as our traditional areas of support in health, medical research, education and the social services," Hearst Foundation President Randolph A. Hearst said Monday in announcing the grants. Lincoln Center's share will be $200,000 for its new building, which will house young artists and provide new public facilities. The Metropolitan Opera Co. and New York Philharmonic will receive $400,000 each. The Juilliard School, where music and the performing arts are taught, will get $250,000. The Hearst Foundation, a leading supporter of the Lincoln Center Consolidated Corporate Fund, will make its usual annual $100,000 donation, too.

"Arts" "Budgets" "Children" "Education"

After a certain number of training iterations, LDA will get an assignment of topic on each word in each document (the same word in different documents might be assigned with different topics), and based on that, output a table of most frequent words in each topic. For evaluation purposes, the objective function - log likelihood may also be computed and output either after each iteration or when the entire training process is over.



What are the key data structures?

The key data structures needed in LDA are the four parameters learned in the model - four different tables used to store different distributions and counts during Gibbs Sampling. Let D denote the number of documents, W denote the number of total number of unique words in all documents, and T denote the number of topics (classes) to train for. Then, the 4 tables used in LDA with Gibbs Sampling are:

- Document-Topic Table (docTopicTable) of size $D * T$
- Word-Topic Table (wordTopicTable) of size $W * T$
- Topic Table (topicTable) of size T
- Document-Word Topic Assignment Table (z) with the same dimensions as the input corpus (each word in each document gets assigned to a topic; thus the table has size W', where W' is the total length of all documents in the corpus).

All 4 tables are implemented using C integer arrays. The Topic Table is a straightforward 1-D array of length T. Both the document-topic table and word-topic table are implemented as a 1-D array but with 2-D indexing. Lastly, while the document-word topic assignment table (z table) could be implemented as an array of size $D * W$, lots of memory would be wasted in this implementation because most training documents will not contain all unique words that appear in all the documents, and words that do not appear in a document will never be used in the z Table. Moreover, D and W could be huge numbers, especially in large-scale text classification. Thus, we used a similar data structure as the representation of graphs in the GraphRats assignment, where our z table is a 1-D array of size W' and we have another array of size D that documents the starting index of every document in the z table array. This way we greatly reduce the amount of memory allocated for the z table.

What are the key operations on these data structures?

The key operation on the four tables described above is the Gibbs sampling process. In each iteration, we loop through every word in every documents in the training data. For each word, we obtain the current topic assigned to the word in the document by looking up the word in the z table. Then, we calculate a multi-nomial distribution of the posterior probabilities using the current probabilities, the smoothing parameters alpha and beta, and counts for this word, and then randomly sample a new posterior topic. At the end, all 4 tables decrease the counts related to the current word with its previously assigned topic, and then increase the counts for the word with its newly assigned topic. Thus, while the sampling process involves more complex operations such as multiplication and divisions, the key operation on the 4 tables are mostly additions and subtractions.

What is the part that computationally expensive and could benefit from parallelization?

Similar to most text classification algorithms (and machine learning algorithms in general), LDA has characteristics that give space for big potential improvements in performance with parallel algorithms - most corpus have very "sparse" dependencies, meaning that most parts of the corpus only depend on a small amount of data, for example, words inside a single document might depend on each other, but words in different documents might not have that tight correlation. In LDA, the relationships between words in different documents don't change much over the iterations of the Gibbs sampling process. This would allow us to distribute the corpus and run sampling on different processes with reasonable synchronization without influencing the classification result much.

Break down the workload. Where are the dependencies in the program? How much parallelism is there? Is it data-parallel? Where is the locality? Is it amenable to SIMD execution?

The bulk of the workload is in the generative Gibbs Sampling process for LDA. In fact, the Gibbs Sampling process for LDA is strictly sequential as the result of the the re-sampled topic for one word is factored into calculating the multi-nomial distribution of all the words sample afterwards.

However, as explained above, the major dependency of the algorithm is the distribution of topics over words inside each document. Due to the sparse dependencies of words across different documents, it is possible to sacrifice the relatively weak dependency for faster sampling, by allowing multiple processes to perform the sampling process independently on different parts of the training data for some time and then combining the results. Thus, is it naturally reasonable to split up the training documents into roughly equal chunks and have different processes train on different documents together in parallel. This is a form of message-passing model, as different processes are assigned different parts of the training data (in their private memory spaces), and need to communicate with each other regularly to obtain training results and update their own parameters.

Lastly, we attempt to utilize cache locality by iterating through all tables in a row by row order so that all memory accesses are close. We also assign documents to processes by block, so that different processes only need to access documents stored near each other in memory. However, the sizes of the tables prevent cache locality from being a huge impact on performance since many of the rows in the tables excess the cache sizes of the GHC machine (which we tested on). Moreover, while similar instructions are performed by each process for each word during the re-sampling process, the fact that the process depends heavily on the randomness of the sampling causes divergence in the work performed by each process. Thus, SIMD instructions are not ideal in this scenario. Thus, in the end it seems most reasonable to use the message-passing model for parallelizing LDA.

Approach:

Describe the technologies used. What language/APIs? What machines did you target?

We used the OpenMPI library's message passing interface to assign work to multiple processes and communicate between different processes. We tested on the GHC machines, which allowed a maximum of 16 hyper threaded parallel instances. We used the language C++ because C++ not only provides libraries for us that helps with reading inputs, string manipulations and sorting but is also compatible with C which we mostly used for our parallel implementations. Most importantly MPI supports both C and C++ and that both C and C++ are very efficient low-level programming languages.

Describe how you mapped the problem to your target parallel machine(s). Important: How do the data structures and operations you described map to machine concepts like cores and threads. (or warps, thread blocks, gangs, etc.)

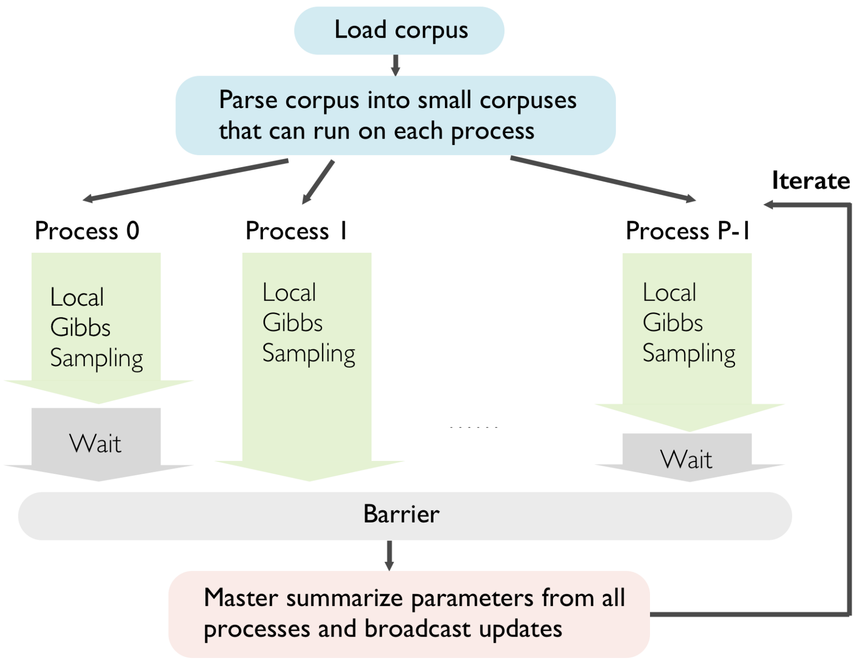
As mentioned in the background section, there are four tables that we need to learn as parameters of the model - the doc-topic table, the word-topic table, the topic table, and the z table (topic assignment table). Among the four tables, the doc-topic table and the z table could be parsed by document, while the word-topic table and the topic table need to summarize across documents.

Since we parallelized the algorithm based on documents using blocked assignment, each process only needs certain rows of the doc-topic table and the z table in order to perform local Gibbs sampling. Similar to the GraphRats assignment, in order for simpler indexing, each process stores the entire tables but only uses the rows assigned to it. Meanwhile, each process needs an updated (or "fairly" updated) copy of the entire word-topic table and topic table, and updates of these two tables need to be communicated across processes using message passing.

In order to gather updates from different processes, we introduce two new tables in our parallel implementation:

- Local Updates to Word-Topic Table (updateW), with the same dimensions as Word-Topic Table
- Local Updates to Topic Table (updateT), with the same dimensions as Topic Table

Thus, each process still goes through the same sampling process as the sequential version - iterate through each word in each document assigned to it, calculates the multi-nomial distribution of topics over the word, randomly sample a topic from the distribution, and updates its own part of the doc-topic table and the z table. Then each process stores the local updates to the word-topic table and the topic table in its local updateW and updateT. After a certain number of iterations, all processes send their local updates to the master, and the master accumulates the updates, adds them to the old word-topic table and topic table, and then sends the new tables back to each process.


Load corpus
Parse corpus into small corpuses that can run on each process
Iterate
Process 0
Process 1
Process P-1
Local Gibbs Sampling
Local Gibbs Sampling
Local Gibbs Sampling
Wait
......
Wait
Barrier
Master summarize parameters from all processes and broadcast updates

Did you change the original serial algorithm to enable better mapping to a parallel machine?

We did not make huge changes to the original serial algorithm when implementing the parallel LDA algorithm. However, because the generative process of Gibbs Sampling is strictly sequential, our parallel algorithm couldn't ensure the exact calculations of the original serial algorithm. As mentioned above, We split the training documents across all working processes, where each process independently performs the same sampling process as the serial algorithm on its own documents. Then, in order to make sure all the processes communicate learned information across each other, we added a step where all processes report their updated tables to the master process, who compiles all the updates and sends the updated tables to all other processes.

As explained above, only the word-topic table and topic table are updated by all processes and these updates need to be communicated to all other processes. To communicate theses updates, we modified the serial algorithm by using an updateW table and updateT table to store each process's update to the word-topic table and topic table. Thus, every time a process receives updated tables from the master process, we clear the update tables to all zeros. Then, instead of decrementing and incrementing the counts on the word-topic table and topic table, we record the changes to the updateW and updateT tables and only send the updateW and updateT tables to master, who can simply add up all updates from all processes to produce the overall updated word-topic and topic tables. The use of the updateW and updateT tables are the only major change to the original serial algorithm.

If your project involved many iterations of evaluation and optimization, please describe this process as well. What did you try that did not work? How did you arrive at your solution? The notes youve been writing throughout your project should be helpful here. Convince us you worked hard to arrive at a good solution.

We first implemented and optimized the sequential benchmark of the LDA algorithm (mostly from scratch). Our first sequential implementation used vectors for 1-D tables and vectors of vectors for 2-D tables. We attempted to use vectors for its useful functionalities such as being able to get the sizes. However, it turns out that vectors are very expensive to operate on. Also, vectors have bad locality because it is not guaranteed that elements of neighboring indices in a vector are physically stored in adjacent memory. In fact, after we switch to arrays from vector, we obtain a 1.5x to 2x speedup on the sequential implementation.

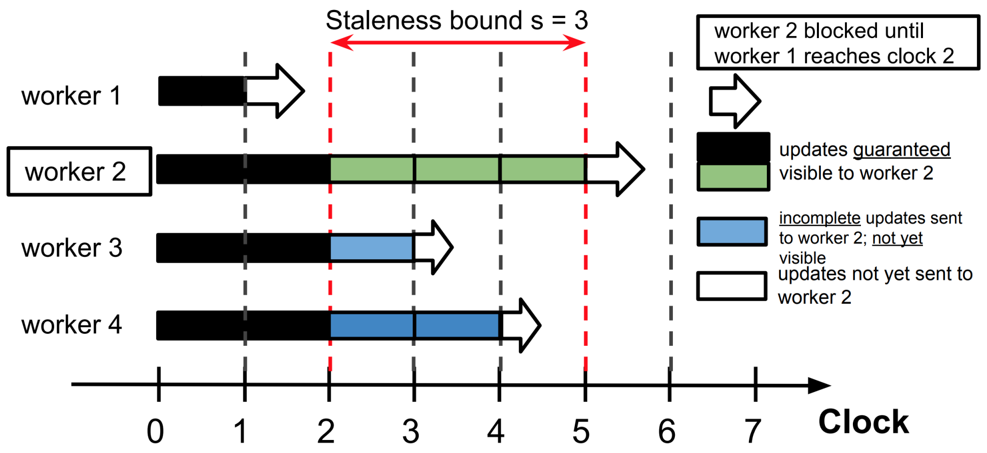
Also, when sampling a new topic for some word w in document d, the sequential algorithm stores an array of probabilities for each topic (as the posterior multi-nomial distribution) and then uses a random number to determine which of the topics becomes the newly assigned topic for w. However, a linear search was initially used in our basic algorithm to search through the probabilities array for the new topic. We implemented binary search instead of linear search, hoping to speed up the sampling process. However, while binary search is asymptotically faster, it is only efficient on large arrays. In our algorithm we only search for at most the number of topics we train for, which is often not a huge number. Thus, there was no speedup by using a binary search instead of a linear search.

After we finished optimizing our sequential benchmark we started working on our first parallel implementation, with the simple synchronization rule of synchronizing updates to the word-topic table and the topic table across all processes at the end of every single iteration. We tried two ways to implement it:

- **IMPLEMENTATION A** According to our "send update" - "receive new table" mechanism, we used MPI_Isend and MPI_Irecv for the non-master branches at the end of each iteration when all the local sampling is over. We used an MPI_Waitall to make sure each process receives a new copy of the tables before they proceed to the next iteration. The master uses MPI_Irecv to gather updates from all processes with an MPI_Waitall (so that the order it receives updates from processes doesn't matter), and adds up all received updates to its own copy of the tables. Then the master uses an MPI_Isend to send the updates back to each process, and keeps on running its own local Gibbs sampling.
- **IMPLEMENTATION B** Since the update gathering process is summing up the same entries to the same table across different processes, we changed the synchronization process at the end of each iteration from explicit MPI_Isend and MPI_Irecv in each process into MPI_Reduce using addition and MPI_Broadcast, so that MPI_Reduce gathers all updates to master and master adds up the accumulated updates to its copy of the tables, and then broadcast the tables back to each process.

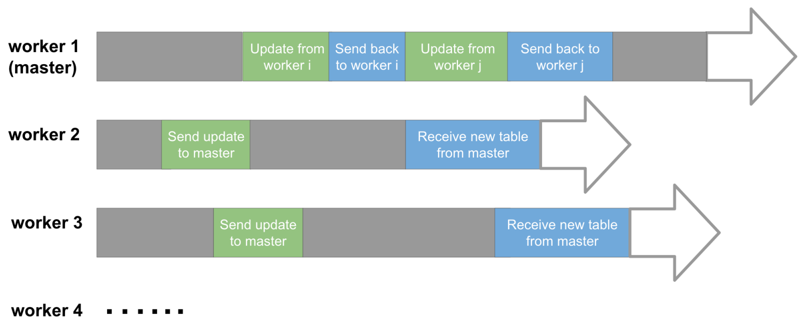
We thought implementation A would be faster since implementation B introduces an implicit barrier across all processes. However, it turned out that implementation B is faster - since in implementation A, all processes still need to wait for master to finish accumulating all the updates, it's not much faster than adding a barrier to all processes. Also, implementation A performs all the additions in master sequentially, which would be less efficient than MPI_Reduce. Moreover, since implementation A needs two large tables to store updates from all processes, the memory consumption is extremely huge, while implementation B only needs two tables with the same size as the update tables to store the accumulated updates, which makes implementation B more economy in terms of memory usage. Thus, we decided to use implementation B.

Then we looked up more possible optimization of parallel LDA, and found that lots of papers mentioned the concept of "staleness" - in order to achieve more speedup, we allow the usage of reasonably "stale" data in some of the tables, so that each process can run local Gibbs sampling for more iterations and less synchronization is needed.


Staleness bound s = 3
worker 2 blocked until worker 1 reaches clock 2
worker 1
worker 2
worker 3
worker 4
updates guaranteed visible to worker 2
incomplete updates sent to worker 2; not yet visible
updates not yet sent to worker 2
0
1
2
3
4
5
6
7
Clock

In order to add the staleness feature, we added a counter in our synchronous implementation of parallel LDA (implementation B) and determines whether or not we need synchronization at each iteration. If not, we just keep on running local Gibbs sampling.

With the concept of staleness in mind, we thought that while synchronized updates work well for a small staleness value, a large staleness value could result in possibly unbalanced workload. For example, some process x may take 0.001 secs faster to compute 1 iteration than process y so a staleness of 2 would only leave process x waiting for 0.002 secs to synchronize, while a staleness of 1000 would leave process x waiting for 1 total second for process y during synchronization. Hoping to optimize this problem, we attempted to implement asynchronous updates on multiple processes. In contrast to our previous synchronous version, during the synchronization phase after every s iterations (where s is the staleness), the asynchronous version does not require each process to obtain a copy of the new table after they send the updates to master, but instead directly starts with the next round of sampling, and updates the tables upon receiving the new tables (which could be after they start with the next s iterations of sampling).



We also tried two ways to do this:

- **IMPLEMENTATION C** We attempted to use non-blocking message passing in this implementation. For each non-master process, for every s iterations that it finishes (s = staleness), it uses MPI_Isend to send its own updates to the master process and then uses MPI_Irecv with a MPI_Waitall to receive updated tables form the master process before it proceeds. For the master process, for every s iterations it finishes, it first updates the tables with its own updates, and then uses a while loop to probe for incoming messages. Within the while loop, the is a for loop constantly probing from process 1 to process p, in that order. MPI_Iprobe is used for non-blocking probing and whenever a message is found, the master process will use MPI_Recv to receive that message, update the tables and then send the updated tables back to the process that sent the message. The master will continue this process until it has counted to have received 2 messages updates from each worker process.
- **IMPLEMENTATION D** We then realized that while work processes benefit the most from the asynchronous implementation above, the master process must wait to process all updates before it can proceed. Thus, it seems useless to use a while loop that keeps looping and using non-blocking probe since the master must wait until it finds a message from any of the worker processes. Thus, in this implementation, instead of a while loop we uses a for loop for $2 * (p - 1)$ iterations to make sure all processes have been communicated with (p = number of processes). And in each iteration, we use MPI_Probe to look for messages from any worker process, and then retrieve the tag and source from the status of the probing to determine which process the message came from and which of the 2 tables it is sending. Then master updates its tables just like before and sends out the updated tables.

We initially wrote implementation C as it was simpler in terms of identifying the source and type of update message - when a message is probed, the source and type of message are definite. However, this even decreases the speedup compared to the synchronous version. Thus we wrote implementation D, and it turned out that implementation D was faster because it removes the costly while loop and the constant probing - whenever master sees an incoming message, no matter where it's from and which table the update is relevant to, it receives the message and processes the update. Since the computations after a message is received are the same for the two implementations, implementation D is overall faster with a faster probing method. Thus, we chose implementation D as our asynchronous version of parallel LDA implementation.

If you started with an existing piece of code, please mention it (and where it came from) here.

The LDA algorithm was briefly discussed in another class we've taken. We referred to lecture materials and pseudocode from relevant papers when writing our own implementation of the LDA algorithm (both the sequential and the parallel versions) from scratch but did not use code from any other sources.

Results:

Overall, we consider ourselves to be fairly successful at achieving our goals. We were able to see significant speedup with the use of multiple processors. We will discuss the specifics of how we evaluated the performance of our parallel algorithms below.

If your project was optimizing an algorithm, please define how you measured performance. Is it wall-clock time? Speedup? An application specific rate? (e.g., moves per second, images/sec)

The goal of our project is to optimize a parallel version of the LDA algorithm. Therefore, we measured the wall-clock time of our optimized sequential LDA algorithm and used this time as a basis for calculating speedup for our parallel algorithms. For our parallel LDA algorithms, we measured the wall-clock time of running the entire program, excluding time for calculating log-likelihood for evaluation purposes. We used the wall-clock time to calculate the speedup compared to the sequential implementation, in order to determine the overall performance of our parallel optimization.

Please also describe your experimental setup. What were the size of the inputs? How were benchmark data generated?

We experimented with the 20 News data set, which is composed of 18774 documents (pieces of news) that belong to 20 different topics. This document set had a total of 60057 unique words and the total length of all documents is 1414350. For our experimental purpose, we used 0.1 for both alpha and beta during training.

As mentioned above, we used wall-clock time of running the sequential program to generate the benchmark to evaluate the performance of our parallel algorithms. We tested the performance with different values of the three parameters below:

- Number of iterations: the total number of iterations through the entire document set to train for.

- Staleness: the staleness parameter used in parallel LDA that determine how often synchronization is performed between the master and worker processes.
- Number of processes: The number of processes used in training, ranging from 1 (sequential) to 16.

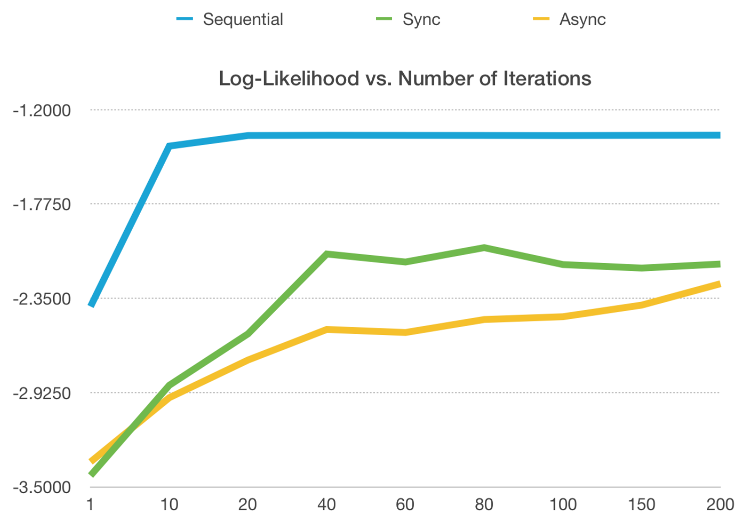
When evaluating the performance of our parallel program, we first experimented with the effect of the number of processes on the speedup of the parallel programs. To do this, we used 10000 iteration and a staleness of 100, and trained (on the documents set described above) using 2 processes all the way through 16 processes. We performed this with both the synchronized approach and the asynchronous approach.

Next, as explained above, we thought that staleness may also affect the speedup of our parallel algorithms, especially for the asynchronous algorithm. Therefore, using 16 processors training for 10000 iterations, we trained using staleness at 20, 50, 100, 200, 500, and 1000. We also ran these experiments on both the synchronous and asynchronous algorithms.

For both of these experiments, we used the speedup as the main measurement of how optimized our parallel implementations are over the serial algorithm. We will be presenting graphs and discuss the speedups we obtained in the next sections. We also recorded the log-likelihood together with the run-time for every experiment mentioned above, since as mentioned before, the parallel LDA algorithm does not strictly follow the sequential order of the serial algorithm and therefore we must measure the accuracy of the parallel LDA algorithms while aiming to achieve a high speedup. This will also be further discussed in the next section.

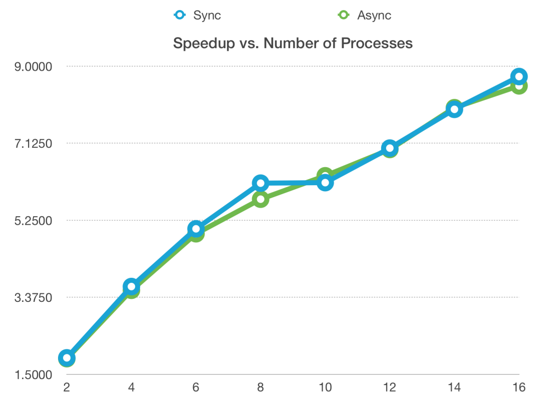
Provide graphs of speedup or execution time. Please precisely define the configurations being compared. Is your baseline single-threaded CPU code? It is an optimized parallel implementation for a single CPU?

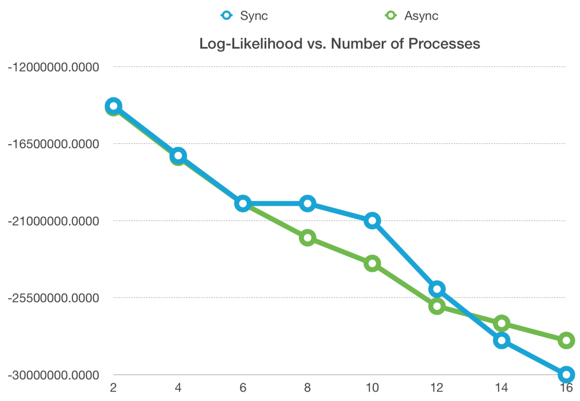
Before we analyze the speedup, in order to show that our parallel implementation still achieves a good result in terms of the objective function and does not sacrifice log-likelihood much, we plotted the convergence of the sequential program and the two implementations of our parallel algorithm as below.



We can see that both of the parallel programs converge slightly slower than the sequential version, but still achieve reasonable log-likelihood after a number of iterations - the sequential version eventually gets to a log-likelihood of around -1.3e7, while both of the parallel programs eventually arrive at log-likelihood of around -2.1e7 to -2.2e7. When we printed out the most frequent words in each category at the end of the programs, the result we got from both parallel programs still made sense. These results show that parallelism indeed sacrifices a bit of the objective function, but does not harm the training result much.

As explained in the section above, our baseline is our sequential implementation of LDA algorithm run on the CPU, while our parallel algorithms runs on the CPU as well but using multiple processes from multiple cores. Below are the graphs and results we found from the experiments described in the section above.

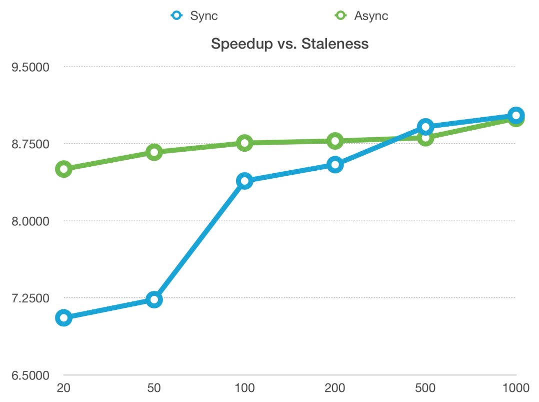


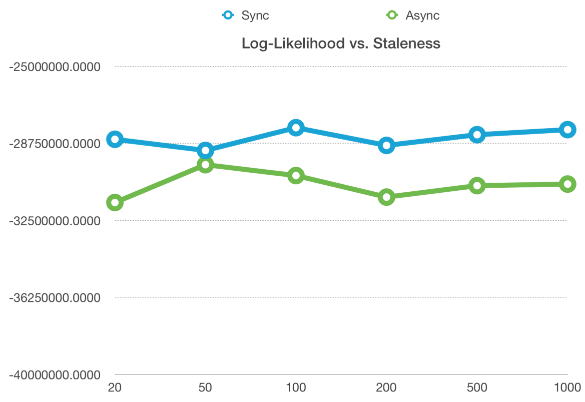


The graph on the left above is the result of the first experiment, where we explored the scalability of our parallel algorithms by investigating how the speedup of the algorithms scale with the number of processes used.

As shown, for both the synchronous version and asynchronous version we were able to see a closet o-linear speedup up until 6-8 processes. And even though the speedup is no longer linear afterwards, we still see a 9x speedup using 16 processes. This pattern is expected because as the number of processes increases, communication becomes more prominent and takes a larger portion of the run time. Moreover, it could also be possible that when running with more than 8 processes, the performance of hyper-threads instead of physically separate cores are much lower, lowering the overall speedup we are seeing. Also, while we expected the asynchronous version to be slightly faster than the synchronous version, due to compensation for work imbalance, it turned out that both implementation yielded very similar speedups. This is because the workload could have been mostly balance for both versions, and that even though the asynchronous implementation allows workers processors to only wait for no process other than the master, the master process is loaded with extra work for communication between processes as master needs to separately receive and send 2 messages to every worker process during a synchronization update. This causes the asynchronous implementation to really have no significant improvement over the synchronized implementation, in which we used MPI_Reduce and MPI_Bcast which are very efficient built-in communication tools.

The graph on the right above shows the converged log-likelihood values vs. the number of processes used during training, for 10000 iterations. It can be seen that the log-likelihood for fewer processes are lower than the log-likelihood for training that used more processes. This is also expected due to the nature of the parallelized algorithm. As explained many time above, our parallel algorithms was not able to fully follow the computation for the serial algorithm because the serial algorithm is strictly sequential. Therefore, having each process train different document separately decreases the training accuracy, which is why we see a lower log-likelihood value after the same number of iterations of training. However, this does not impact the correctness of the parallel algorithms.





The two graphs above here are results of our second experiment, where we experimented on the relationship between speedup and staleness. As mentioned, staleness controls how often synchronization happens for our parallel algorithms. As show in the graph on the left, both the synchronous and asynchronous implementations show an obvious increase in speedup due to increased staleness. This shows larger staleness does decrease the overall communication time, leading to better performance.

We also recorded the log-likelihood values vs. staleness. This is because we expected larger staleness to cause a lower log-likelihood value due to the less communication of information between different processes. Staleness **does** have an impact when it comes to extreme conditions when we set the staleness to be equal to the number of iterations (using staleness of 10000 for 10000 iterations), i.e. we only synchronize at the end of the entire training process - the log-likelihood decreases to around -3.4e7 for the synchronous version and decreases to around -3.7e7 for the asynchronous version. However, as plotted above, it seems that with relatively small staleness, for both synchronous and asynchronous implementations, the log-likelihood values are rather consistent. These results indicate that while staleness may increase the performance (speedup) of the parallel algorithms, it does not have a huge impact on the accuracy of the training algorithm, which is great.

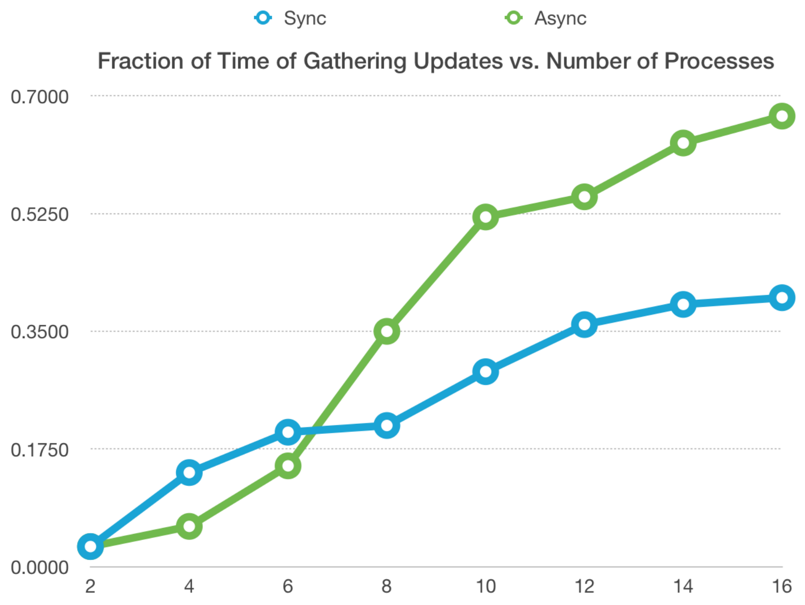
Recall the importance of problem size. Is it important to report results for different problem sizes for your project? Do different workloads exhibit different execution behavior?

Since there are a lot of different parameters involved and to be experimented with in our parallel LDA algorithm (for example, the log-likelihood, the staleness, the number of iterations, etc.), our main focus of experiment was the influence of these parameters on the speedup and log-likelihood of the training process. Thus, we chose to stick with the single data set of 20News and did not test much with different problem sizes.

Also, in text classification, it's generally hard to have convincing classification results with data sets that are too small, so the training data sets are typically very large (in our case, with 18774 documents and a total of 1414350 words in the corpus). Thus, we can claim that our algorithm achieves a good speedup for a large data set.

Important: What limited your speedup? Is it a lack of parallelism? (dependencies) Communication or synchronization overhead? Data transfer (memory-bound or bus transfer bound). Poor SIMD utilization due to divergence? As you try and answer these questions, we strongly prefer that you provide data and measurements to support your conclusions. If you are merely speculating, please state this explicitly. Performing a solid analysis of your implementation is a good way to pick up credit even if your optimization efforts did not yield the performance you were hoping for.

Since we used message passing among different processes and the messages to be passed are relatively long, our main bottleneck was the communication time (including the computations involved in accumulating updates in the MPI_Reduce). With larger number of processes, the number of messages to be passed increases and the process of accumulating updates from multiple processes also gets more computation intensive.



As shown in the figure above, we timed the local Gibbs sampling part of each program as well as the synchronization part. We found that, as the number of processes increases, the proportion of time spent on gathering updates get substantially larger, from 3% with 2 processes to around 60% with 16 processes. With the use of MPI_Reduce, it's relatively hard to separate the time spent on message passing itself and the time spent on accumulating the numbers, but our assumption is that as the number of processes go up, accumulating updates from different processes also gets more computation intensive, thus leading to longer synchronization time.

Also, as we discussed before, the sequential nature of Gibbs sampling itself prevents more parallelism to be involved - at each iteration of Gibbs sampling, we still need to sequentially go through the sampling process of each learned parameter.

Deeper analysis: Can you break execution time of your algorithm into a number of distinct components. What percentage of time is spent in each region? Where is there room to improve?

As discussed in the section above, we timed the part our program spent on local Gibbs sampling and on gathering updates to the parameters from different processes. According to our reported breakdown of computation and communication time above, as the number of processes go up, the proportion of time spent on communicating between processes and gathering updates gets much larger, from 3% with 2 processes to around 60% with 16 processes.

The work imbalance would also be a bottleneck when master takes on more computations in gathering the results. As we researched different possible designs of parallelism for LDA, we've also seen algorithms in which each process sends the updates to another random process (instead of the master) - due to time limitations, we weren't able to implement this algorithm, which might further improve speedup but would also converge much slower and sacrifice the log-likelihood even more.

Was your choice of machine target sound? (If you chose a GPU, would a CPU have been a better choice? Or vice versa.)

We chose to test our parallel implementation and experiments on the GHC machine clusters. This choice is very reasonable and convenient because our implementation requires machines that have MPI installed and supports a good number of parallel processes. The GHC machines have MPI and each machine has 8 cores where each core has 2 hyper-threads. This allows us to test anywhere up to 16 parallel processes for our algorithm.

However, a downside of testing on the GHC machines is that run time on the GHC machines could vary greatly depending on how much of the machine's CPU is in use by other users. We've noticed some inconsistencies when timing our experiments. However, we were able to overcome this by running our experiments multiple times and only taking the most common results that we get. However, this issue could have been avoided if we tested on the Latedays cluster using the full power of one of the worker nodes at the cost of lower speedup due to the limited computation power of Latedays clusters.

We chose to run this on a CPU because we chose to implement the parallel algorithm using the message-passing model, which involves multiple processes and different computations on different processes. Our worker processors must communicate information to each other throughout the algorithm and we need to be able to control how and what is communicated, so SIMD is not a good option for our algorithm. Thus, we do not think implementing this using a GPU would be better.

References:

Blei, David M., Andrew Y. Ng, and Michael I. Jordan. ”Latent dirichlet allocation.” Journal of machine Learning research 3.Jan (2003): 993-1022.

Newman, David, et al. ”Distributed algorithms for topic models.” Journal of Machine Learning Research 10.Aug (2009): 1801-1828.

Smyth, Padhraic, Max Welling, and Arthur U. Asuncion. ”Asynchronous distributed learning of topic models.” Advances in Neural Information Processing Systems. 2009.

A. Ihler and D. Newman, ”Understanding Errors in Approximate Distributed Latent Dirichlet Allocation,” in IEEE Transactions on Knowledge and Data Engineering, vol. 24, no. 5, pp. 952-960, May 2012.

N. Besimi, B. io and A. Besimi, ”Overview of data mining classification techniques: Traditional vs. parallel/distributed programming models,” 2017 6th Mediterranean Conference on Embedded Computing (MECO), Bar, 2017, pp. 1-4.

Division of Work:

Equal work was performed by both project members.