Parallelizing the training of sequential and linear models

COMP 5704 Wei Li

Abstract

Linear models cover a large set of applications and are widely used but are inherently hard to be trained in parallel due to their inherent serial nature. To train linear models in a multicore system, one can either use thread locks to control memory access or do parallel updates without synchronization. Hogwild proved that SGD training can be done in parallel without synchronization and the speedup depends on the problem's sparsity. Recent work like SAUS extends the speedup and scalability based on the asynchronous in Hogwild. In this project, we implemented Hogwild and SAUS and empirically experiment with the speedup and sparsity, the result shows that asynchronous methods are better than locking schemes.

1. Introduction

Linear models are widely used and are still popular for sparse and high dimensional problems. Given the ubiquity of large-scale data problems and the availability of low-commodity clusters and multicore systems, paralleling the training of linear machine learning models to speed them up is an obvious choice. However, optimization algorithms like SGD's scalability are limited by their inherently sequential nature (Recht et al., 2011). Machine learning tasks are typically scheduled as a series of training steps, where a processor takes the current state, calculates the update and applies the update to the state. Each step can only be done after the last step has been finished. Otherwise, gradients will be computed on stale versions of the solution, and the latest update may overwrite others' work.

Asynchronous methods provide an idea to properly parallelize and provide a good speedup to the training of linear models. Asynchronous algorithms roots in the concept of parallelism and concurrency. Parallelism means splitting a task into smaller subtasks so that different parts of a task can be processed in parallel, which makes the whole process faster. Concurrency is about managing the shared resources to do multiple tasks at once. In the case of training linear models, a natural way to parallel training is to split the whole training process by training steps. However, if workloads are split as a number of steps, there are the concurrency problems that memory can be simultaneously accessed by multiple cores, and writes from different cores may overwrite each other. To run a training successfully, we need precise control of memory reads and writes.

Traditionally the concurrency problem is solved by threads synchronization through locking. However, the overhead induced by acquiring locks becomes far more expensive comparing to the actual computation of updates. As a result, the ideal way to parallel the training of linear models is asynchronous training without using locks, and maintain the models' convergence. The possibility of asynchronous linear model training has been analyzed and many methods and good practices have been proposed. It is observed that in conditions like sparse machine learning, the process can be parallelized without any synchronization, and the model can still converge as well as the serial counterpart.

In the project, an earlier asynchronous method Hogwild and a recent method Stochastic Atomic Update Scheme (SAUS) were implemented in C and pthreads library. We empirically evaluated the effect of sparsity to Hogwild on its speedup and convergence. We also compared how Asynchronous methods like Hogwild is superior to the locking method. The SAUS (Raff et al., 2018) is another similar asynchronous method, it was only implemented for the Stochastic Dual Coordinate Ascent (SDCA) method in Java in the paper. We implemented the method in C and also made an attempt to apply the method to Stochastic gradient descent (SGD). However, there are the parameter tuning problems of SGD that failed to produce converging models.

The rest of the report is organized as follows. In section 2, we review related works in parallel linear model training and. In section 3, we formalize and limit the problem to be discussed in the paper. In section 4, we introduce the two algorithms that is implemented in the project. Then, the experiment details and experimental results are presented in section 5. Finally, the project report is concluded in section.

2. Literature Review

There are two types of parallel linear models training researches. One direction of researching focuses on paralleling the training of large data sets that are distributed in a cluster of machines and are pre-processed in a MapReduce-

like framework(Dean & Ghemawat, 2004). In such a scenario, only appropriate MapReduce-preprocessed data can be small enough to be trained parallelly in a number of machines. However, this framework's performance is often impaired by the low data reading rate due to communication and the frequent checkpointing for fault tolerance(Recht et al., 2011).

(Zinkevich et al., 2010) proposed running many instances of stochastic gradient descent on different machines and averaging their output. However, (Recht et al., 2011) shows in their experiment that this method does not outperform a serial scheme. (Dekel et al., 2012) and (Agarwal et al., 2010) also propose schemes based on the idea of averaging of gradients via a distributed protocol and achieved linear speedup.

Another stream of studies looks at paralleling linear models training in a single multicore systems. Unlike cluster servers, local workstations have shared memory with low communication latency and high bandwidth. These advantages move the parallel computation bottleneck from slow communication to frequent synchronization (or locking) – as the workers work on the same memories, they must be more careful about concurrency.

A parallel linear learning method should have two primary goals: (1) Model Quality, that the algorithm should be able to converge to the same result as its serial counterpart and perform equally well; (2) Training Efficiency, that as the number of processors increases, the speed of training should increase at least at the same scale. Both the cluster training and single-machine training need to be optimized for the two goals. In terms of Training Efficiency, cluster training aims at better communication between different hosts and single-machine training aims at avoiding synchronization and locking among workers.

In the multicore scenario, a pivotal and most commonly used method is the Hogwild (Recht et al., 2011). Prior to the work, most locking scheme and ideas for parallelizing stochastic gradient descent is presented in (Bertsekas & Tsitsiklis, 1989). They describe, for example, using stale gradient updates computed across many processors in a master-worker setting, they proved convergence of those approaches, but did not mention the speed of the convergence. Another locking scheme is round-robin (Zinkevich et al., 2009) where the processors are ordered and each one updates the decision variable in sequence. When gradient computation is costly, the time needed by locking is dwarfed, so the method achieved linear speedup. However, the method does not scale well when gradients can be easily computed.

(Recht et al., 2011) proved and tested that in the case when the data access is sparse enough, the possibility of memory overwrites are rare, and even they occur, they introduce barely any error into the computation. They present the Hogwild method which requires no locking and achieved linear speedup. They also outperform the Round Robin method's speed by an order of magnitude in applications where gradient can be effectively computed. The asynchronous strategy has become the dominant method to parallel linear models in the multicore systems. After Hogwild, many works adapted the method or use the asynchronous strategy in their methods. In these adaptations a common observation is that models often diverge after many cores are added, or when the feature set is dense.(Hsieh et al., 2015)(Leblond et al., 2017)(Tran et al., 2015)(Zhao & Li, 2016). Since Wild does not scale well in multiple-CPUs system(multi-sockets) (Zhang et al., 2016) proposed Hogwild++, which uses multiple local weight vectors to avoid performance regressions of SGD on multiple CPUs systems. However, this approach introduces a new hyper-parameter to be tuned which consumes CPU time depending on the hardware and data.

Some of the studies also look into another type of gradient method, namely the Stochastic Dual Coordinate Ascent (SDCA), which covers a wide set of possible linear models. Coordinate descent is a classical optimization technique (Bertsekas, 1995). SDCA tries to solve the dual problem of risk minimization and shows faster speed than the primal solver such as SGD. (Shalev-Shwartz & Zhang, 2013a) and (Chang et al., 2008) shows that solving the dual problem with SDCA is faster on large-scale data sets. Another advantage of SDCA is that it is not sensitive to the learn rate, which make it easier to be tuned. SDCA is successfully implemented in a linear model package for large data sets called LIBLINEAR (Fan et al., 2008).

After (Tran et al., 2015) implemented a parallel version of SDCA based on the asynchronous idea from Hogwild. (Hsieh et al., 2015) developed the PASSCoDe framework for parallel implementations of the SDCA algorithm and compared Hogwild with Atomic and lock-based methods. Their experiment shows that the Hogwild algorithm provides the best overall efficiency. They also found that the atomic approach's performance varies significantly depending on the sparsity of the data set. Since Hogwild shows difficulty of converging in multiple CPUs system, (Zhang & Hsieh, 2016) developed PASSCoDe-fix to fix this problem. In the approach, although updates are performed wildly, there are convergence checks at each iteration, which result in a semi-asynchronous solution. Although the adaption is successful, the method is only specific to the SDCA, and its complex implementation slows down the computation. Following the work, (Raff et al., 2018) presented a stochastic update policy called Stochastic Atomic Update Scheme (SAUS) which introduces no new tunable hyperparameters,

and is not algorithm-specific. The method shows superior speedup on systems with more than 8 cores and scales up nicely up to 80 cores.

3. Problem Statement

Methods that are listed in section 2 covers a wide range of linear models and optimization methods. Since this is an empirical study, we limit the discussion to the training of linear regression and logistic regression models. In this section, we define the linear regression problem and logistic regression problem. We also discuss SGD and SDCA, the two optimization methods that are used in the implementation. Also, the experiment data generation will be discussed.

3.1. Linear regression model

A linear regression model f takes $x_i \in \mathbb{R}^n$ as input and output a real number y_i via the linear transformation

$$y_i = f(x) = w * x_i$$

where $w \in \mathbb{R}^n$ is the vector of weights, also the vector to be optimized. We will use squared error as our loss function, which for a single data point is written as

$$l(x_i, y_i) = (f(x_i) - y_i)^2 = (w * x_i - y_i)^2$$

The goal of linear regression is to minimize the loss function to 0. In other word, we solve the following optimization problem:

$$\hat{w} = \arg\min_w \|y - Xw\|^2$$

X is a n by m matrix, m is the number of data instance in X.

In the experiment, instead of using linear problem benchmarks data sets, we generate the problem on the fly. We set the size of data set m, the number of features n. The 'true' weight of each feature is sampled stochastically from a standard Gaussian distribution. There is also a sparsity parameter $p \in (0, 1]$ which controls what the percent of features in a data instance is a non-zero value. The whole sparse data set is also denoted as X, The regression value y is calculated as

$$y = w * X + \delta$$

where δ is a noise term sampled from standard Gaussian distribution.

3.2. Logistic regression model

The model of logistic regression is essentially the same as linear regression, but a sigmoid function is applied to the product of the linear transformation $w * x_i$, which maps the output to the range of $[0, 1]$.

When generating the data, we do the same thing as linear regression, but after calculate the regression value y from

$$y = w * X + \delta$$

instead of using y directly as the label, we set y as its sign.

$$y \leftarrow sign(y)$$

3.3. Stochastic gradient descent

Most of the parametric linear models can be solved efficiently with Stochastic gradient descent. SGD is good for its strong theoretical guarantees, robustness against noise and fast learning. In the linear regression case, stochastic gradient descent can be easily implemented by three steps: sampling data instances, calculate the gradient of the loss function with respect to the weight and apply the gradient to the weight by a multiplying the learning rate. The whole training process using SGD can be described as:

- sampling label y and data x from X
- calculate the gradient by:

$$\mathbb{E}\left[G_w(x, y)\right] = \nabla_w \ell(x, y)$$

$$G_w(x, y) := -2(w \cdot x - y)x$$

- apply the gradient update to w

$$w_{t+1} = w_t - \lambda G_w(x, y)$$

  where λ is the learning rate.

However, one downside of SGD also has to do with the hyperparameter learning rate λ. The algorithm is very sensitive to the choice of learning rate. A bad learning rate makes the model diverge. Even when a good learning rate is chosen, setting a constant learning rate often makes the model converge to an undesirable location. When training linear model in parallel and asynchronously, where the problem diverge a lot, the problem of hyperparameter tuning becomes more salient.

3.4. The problem

In the project, we try to parallelize the training process of linear regression and logistic regression using asynchronous methods. We implement Hogwild and Stochastic Atomic Update Scheme (SAUS) in C. The aim of the implementation is to empirically test asynchronous methods' effect and speedup on parallel linear model training. In the experiment, we check the convergence of the models and the speedup of the training process and compare the result of Hogwild and locking scheme.

4. Proposed Solution

In this section, the two methods that are implemented in the project will be discussed. Firstly, the Hogwild lays a theoretical ground to the asynchronous training and leaded to empirical studies that confirmed the conclusion. Then, the recent method Stochastic Atomic Update Scheme (SAUS) presented a more effective, and also not very complex asynchronous parallel algorithm on the idea of asynchronous training.

4.1. Hogwild!

Hogwild! (Recht et al., 2011) proposes a simple strategy to eliminating the overhead associated with locking: run the optimization algorithm in parallel without locks. During the training, each thread simply executes gradient update steps asynchronously. The possibility of collision and overwrite is also allowed. the lock-free scheme appears doomed to fail as processors could overwrite each other's progress. However, Hogwild! theoretically proved that when the data access is sparse, individual SGD steps only modify a small part of the decision variable, thus memory overwrites are rare and they introduce barely any error into the computation when they do occur.

When executing Hogwild, we assume a shared memory model with p processors. The decision variable x is accessible to all processors. It is also assumed that the componentwise addition operation is atomic, that for a variable x_v and a scalar a, the operation $x_v \leftarrow x_v + a$ can be performed atomically without a separate locking structure. On the contrary, the operation of updating many components at once, like vector addition, requires a locking structure.

Algorithm 1 HOGWILD! update for individual processors

loop
  Sample e uniformly at random from E
  Read current state x_e and evaluate $G_e(x_e)$
  for $v \in e$ **do** $x_v \leftarrow x_v - \gamma G_{ev}(x_e)$
end loop

In Hogwild SGD update, each thread executes the procedure in Algorithm 1. A number of samples are drawn uniformly from the data set, then the thread access the global state vector x_e and evaluates its gradient. After obtaining the gradient, the thread updates all the locally calculated gradient to the global state vector. Note that although for a single variable, update operation – the swapping operation – is atomic, since updating the gradient involving updating many variables in a vector, collision is excepted to happen during the execution of the process.

The segmented Hogwild is a slightly different implementation to Hogwild. The methods segments the target weight vector into partitions and threads keep a local copy of the global weight vector. Each thread only updates its assigned partition in the local vector. To communicate with the global weight, each thread periodically updates its assigned partition to the global weight vector and copy other partitions from the global weight vector to the local weight vector.

4.2. SAUS

Like Hogwild, Stochastic Atomic Update Scheme (SAUS) aims at training linear models asynchronously without locking structures. However, besides the lock-free update, SAUS introduced local weight to each one of the processors. To alleviate the communication overhead problem, in each step, the processor shares information between local and global weight vector stochastically.

There are two stages in the algorithm, one is the UPDATE stage, and the other is the ACCUMULATE stage. The UPDATE and ACCUMULATE steps are defined in Algorithm 2.

Algorithm 2 Stochastic Atomic Update Scheme

procedure UPDATE(Thread context p, weight vector $\boldsymbol{w}$, update $\boldsymbol{z}$, Probability multiplier ϱ)
  rand $\sim \mathcal{U}(0,1)$
  $\boldsymbol{w}^p \leftarrow$ weight vector local to thread p.
  for each non-zero value $z_i \in \boldsymbol{z}$ **do**
    $w_i^p \leftarrow w_i^p + z_i$
    if rand $\leq \varrho/|P|$ **then**
      $w_i \leftarrow w_i + w_i^p$ ▷ atomic update
      $w_i^p \leftarrow 0$
procedure ACCUMULATE(All thread contexts P)
  for $p \in [0, |P|-1]$ **do** ▷ in parallel
    for $i \in [{}^{pD}/_{|P|}, {}^{(p+1)D}/_{|P|}]$ **do**
      $w_i \leftarrow w_i + w_i^p$ ▷ non-atomic, but safe

Assume we have P total threads performing the work, and each thread is indexed by p. Initially, each thread $p \in P$ has its corresponding local weight vector w^p stored in thread context. The vector w^p is initialized as all 0. This local vector will not be seen by other threads, so updates to this vector do not need any lock and are thread-safe. In each one of the training steps, processors execute the subprocess UPDATE in parallel.

In the UPDATE stage, Like Hogwild, each processor p sample data instances and calculate the weight update, which is stored in a cache z. Each non-zero value $z_i \in z$ will be updated to the local vector w_i^p. Then, for the very value w_i^p (indexed by i) stored in the local vector, we will give each thread a $1/P$ chance to do an atomic update that transfers the information from w_i^p to the global vector w_i. This approach allows each thread to maintain local information, which will be periodically shared with the global vector.

When a thread wins the Bernoulli trial, the update will be done using the accumulated values in w^p but only for the non-zero updates that have just been updated locally. This means the stochastic update will not fully transfer all information, instead, it transfers information from frequently used features that are more likely to be updated.

Also, in the scheme, each processor updates the global weight vector w stochastically. Obviously the scheme reduces the number of communication and the possibility of collision.

The second stage is ACCUMULATE. ACCUMULATE is executed at the end of each epoch. It runs through all local vectors w^p and accumulates them into the shared weight vector w. It serves as a barrier to ensure the global vector is completely up-to-date at a regular frequency. Although updating all local vectors w^p to w is not atomic, the update can be implemented in parallel without the use of a lock, while still thread-safe. It is done by temporarily allowing each thread to access all local vectors $w^1 ... w^p$, and each thread reading a disjoint range of values to update them into the shared w.

5. Experimental Evaluation

In the experiment, we run experiments on locking scheme and Hogwild to train linear regression and logistic regression models in parallel. We compare the two methods in their convergence and speedup. We also experiment on the how problem sparsity affects the speedup of Hogwild. The experiment was run on two different machines, one has up to 8 cores and the other has up to 20 cores. More detail related to the experiment is given in section 5.1 - 5.4, and the results are given and discussed in section 5.4.

5.1. C implementation

The experiments were coded in C and Python. All the functionality that involves computation and training, like gradient calculation, SGD, and asynchronous update methods, were implemented in C and pthreads library to get the most of the control over parallel training. On the other hand, Python was used to generated training data sets and to manage the experiments and results.

5.2. Hardware

We conducted the same experiments on 2 different machines. One is my local machine with the following configuration: an Intel Core i7 9700 CPU(with 8 cores without hyperthreading) with 16GB of RAM. The experiment is done with a Windows Subsystem for Linux (WSL) installed with Ubuntu 18.04. Another machine is a virtual instance in Carleton Openstack with the following configuration: 20 virtual CPUs and 16GB RAM, the kernel is Ubuntu 18.04, and hyperthreading is also disabled.

5.3. Experiment

Three methods were implemented to parallelize linear regression and logistic regression training, they are 1) a naive-locking scheme, 2) Hogwild 3) SAUS. However, since SAUS is expected to be applied to step-size free optimization algorithm like SDCA, we encountered difficulty in tuning SGD with SAUS (the model diverge easily). Consequently, we only report the experiment result for the other two methods.

The naive-locking scheme is a synchronous method to parallel linear model training. Each thread acquires a lock to the global weight vector before their update. The method serves to compare and show the effect of asynchronous update methods. The other one, Hogwild, is described in section 4.1.

We generate the linear regression and logistic regression problem as described in section 3. All the problem is of 1000 features and 10^5 different data instances. We fix all training to 10^6 steps for consistent comparison. After the training terminates, each data instance is expected to be seen 10 times.

To save the time of hyperparameter tuning, also to be fair, we fix the learning rate to all training to 0.00001. Obviously, the setting is not optimal and some models may not be trained as well or even unable to converge. As the point of the project is exploring the speedup of the asynchronous methods, although some models does not converge to 0, we can still see the speedup by fixing the number of steps.

5.4. Experimental results

We compare different methods in terms of their model quality (the convergence), the speedup and the effect of problem sparsity. We trained the combination of Hogwild&linear regression/ Naive scheme&linear regression/ Hogwild&logistic regression/ Naive scheme&logistic regression on different data sparsity and number of cores. The sparsity options are chosen from $[0.5\%, 1\%, 10\%, 20\%, 50\%, 100\%]$. And the number of cores is from 1 to 8 in the PC and from 1 to 20 in the virtual machine.

5.4.1. THE CONVERGENCE

linear regression: The result of logistic regression training in the 8 core machine is given in figure 1. Firstly, using Hogwild or the locking scheme can train the linear model. When the data density is greater than 10%, the model can converge to the optimal. In the sparser case, the models can still be optimized. Also, an obvious observation is that when using Hogwild, adding more core always produces faster training

than 1 thread execution. On the other hand, using the naive locking method, adding more cores is always slower than 1 thread execution. The result confirms the statement that acquiring locks for update steps actually take more time than the update calculation itself. Another observation is that the denser the data set is, the longer it takes to train the given steps, and the more complete the training is (its convergence). When the data is very sparse, the training ends up barely begin after 10^6 steps. The result also shows that when the data is dense, the effect of using more cores is not obvious.

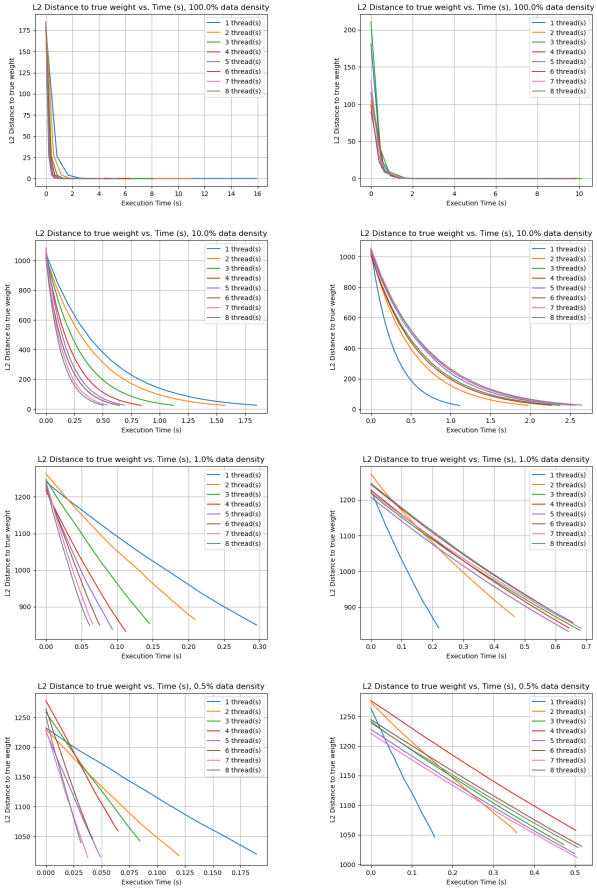

Figure 1. Execution time(s) vs. L2 distance to the true value, training linear regression with Hogwild (left column) and the Naive method (right column) in different data density, training was conducted in 8 cores machine

logistic regression: The result of logistic regression training in the 8 core machine is given in figure 2. The learning rate did not give fast training comparing to linear regression as the model was still being optimized by the end of the given steps. Nevertheless, the models still show signs that they are being optimized. The convergence behavior of logistic regression is similar to linear regression. We also have the observation that the locking scheme runs slower when more cores are added.

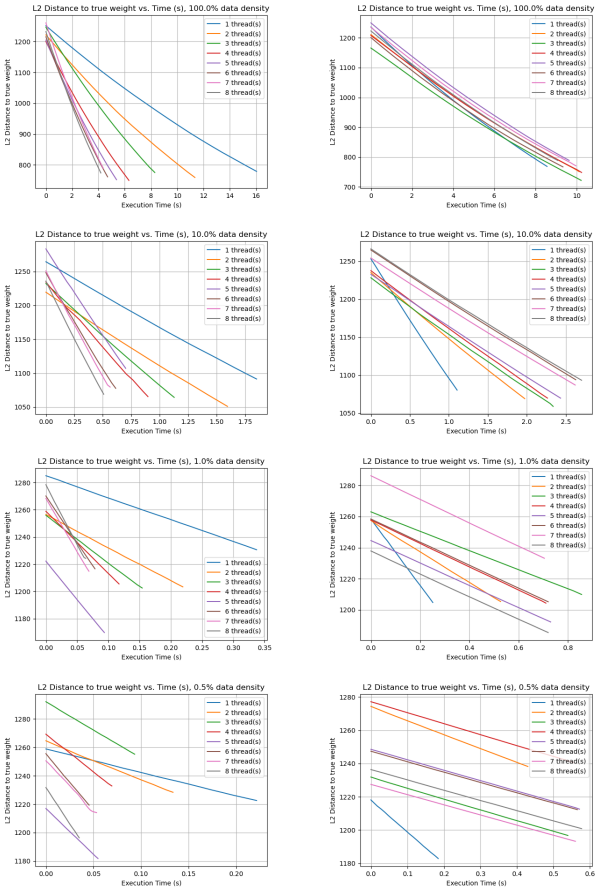

Figure 2. Execution time(s) vs. L2 distance to the true value, training logistic regression with Hogwild (left column) and the Naive method (right column) in different data density, training was conducted in 8 cores machine

The result of the convergence in the 20 cores machine is included in the Appendix A.

5.4.2. THE SPEEDUP

We plot the number of cores against execution time for various sparsities and the speedup curve for various sparsities. All Figures are linear regression problem with 10^5 data points and 1000 features. Figure 3 is number of cores against execution time in 8-cores machine. Figure 4 is number of cores against execution time in 20-cores machine. Figure 5 is speedup in 8-cores machine. Figure 6 is speedup in 20-cores machine.

Comparing Figure 3 and Figure 5, it can be seen that there are some differences in the result of the same experiment in two different systems. The curve of the 20-cores machine is zigzag comparing to the smooth change in the 8-cores

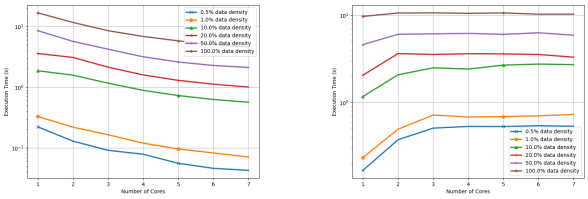

Figure 3. Execution time(s) vs. num of cores, training linear regression with Hogwild (left column) and the Naive method (right column) in different data density, training was conducted in 8 cores machine

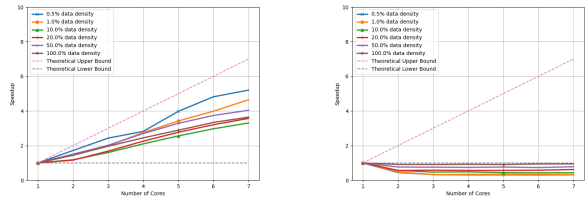

Figure 4. Speedup vs. num of cores, training linear regression with Hogwild (left column) and the Naive method (right column) in different data density, training was conducted in 8 cores machine

machine. It is also observed in Figure 5 that in Hogwild case, some experiments use more time after more cores are added, which does not appear in the 8-cores machine. Other than the observation above, the two figures share the same trend, 1) the sparser the data is, the faster the execution (but does not seems like linear relation). 2) adding more core speed up the execution in Hogwild case, and slow down the execution in the Naive locking scheme.

Comparing the Hogwild speedup in 8-cores machine and 20-cores machine, it can be seen that the 8-cores machine produced much clearer relation of data density and the speedup, whereas in the 20-cores machine everything tangled up. In the 8-cores result, the less dense data generally produce better speedup with Hogwild. The observation does not always true, as the speedup of 10% and 20% is worse than that of 100% and 50%. The speedup of the 20-cores machine is also not as good as the 8-cores machine. Other factors related to the hardware and system should be taken into account.

Finally, although we were not able to use SAUS to train SGD linear models due to the hyperparameter tuning problem, we observed that SAUS algorithm cause significantly fewer atomic collisions during the training. The result is intuitive since SAUS use a random update with a probability smaller than 1. For example, for a linear regression problem of 1000 features, 10^5 data points and 10% density, when SAUS and Hogwild are given 10^6 steps of training, their number of atomic collisions is shown in Figure 7.

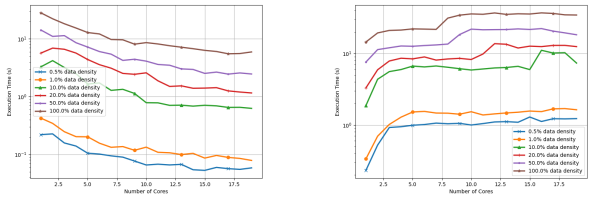

Figure 5. Execution time(s) vs. num of cores, training linear regression with Hogwild (left column) and the Naive method (right column) in different data density, training was conducted in 20 cores machine

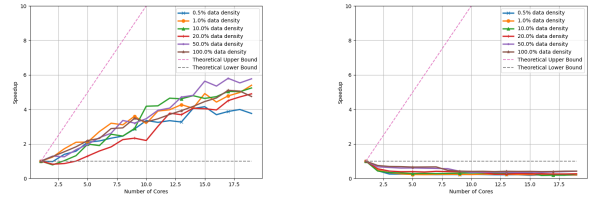

Figure 6. Speedup vs. num of cores, training linear regression with Hogwild (left column) and the Naive method (right column) in different data density, training was conducted in 20 cores machine

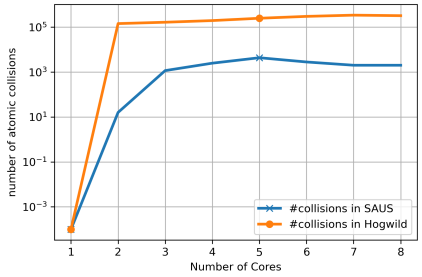

Figure 7. Number of collisions in SAUS and Hogwild

6. Conclusions

In this paper, we successfully experiment with the speedup and convergence of parallel linear model training with Hogwild, and compare the result with a naive implementation of the locking scheme. The result shows that Hogwild is able to parallel linear regression and logistic regression training when the data is either sparse or dense. Also, the result confirms that using asynchronous update schemes is superior to the locking scheme, as using locks will induce worse performance when more cores are added. The experiment also shows a vague connection between sparsity and the performance of Hogwild, that sparser data set may boost the performance of Hogwild in the 8-cores machine. However, the result is not conclusive, as the 20-cores machine experiment shows some contradictory results.

This study can be extended to SAUS and Stochastic Dual Coordinate Ascent (SDCA) given more time to tune the hyperparameters. Also, more studies and experiments are needed to confirm the condition in which Hogwild can scale up to more cores, and what the upper bound is.

References

Agarwal, A., Wainwright, M. J., and Duchi, J. C. Distributed dual averaging in networks. In *Advances in Neural Information Processing Systems*, pp. 550–558, 2010.

Asanovic, K., Bodik, R., Catanzaro, B., Gebis, J., Husbands, P., Keutzer, K., Patterson, D., Plishker, W., Shalf, J., Williams, S., and Yelick, K. The landscape of parallel computing research: A view from berkeley. 2006.

Bertsekas, D. and Tsitsiklis, J. Parallel and distributed computation: Numerical methods. 1989.

Bertsekas, D. P. Nonlinear programming. *Journal of the Operational Research Society*, 48:334, 1995.

Chang, K.-W., Hsieh, C.-J., and Lin, C. Coordinate descent method for large-scale l2-loss linear support vector machines. *J. Mach. Learn. Res.*, 9:1369–1398, 2008.

De, S. and Goldstein, T. Efficient distributed sgd with variance reduction. *2016 IEEE 16th International Conference on Data Mining (ICDM)*, pp. 111–120, 2016.

Dean, J. and Ghemawat, S. Mapreduce: Simplified data processing on large clusters. 2004.

Dekel, O., Gilad-Bachrach, R., Shamir, O., and Xiao, L. Optimal distributed online prediction using mini-batches. *The Journal of Machine Learning Research*, 13:165–202, 2012.

Fan, R.-E., Chang, K.-W., Hsieh, C.-J., Wang, X.-R., and Lin, C. Liblinear: A library for large linear classification. *J. Mach. Learn. Res.*, 9:1871–1874, 2008.

Fuller, S. and Millett, L. I. The future of computing performance: Game over or next level? 2014.

Gan, W., Lin, C.-W., Fournier-Viger, P., Chao, H.-C., and Yu, P. S. A survey of parallel sequential pattern mining. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 13:1 – 34, 2019.

Hsieh, C.-J., Yu, H.-F., and Dhillon, I. Passcode: Parallel asynchronous stochastic dual co-ordinate descent. *ArXiv*, abs/1504.01365, 2015.

Leblond, R., Pedregosa, F., and Lacoste-Julien, S. Asaga: asynchronous parallel saga. In *Artificial Intelligence and Statistics*, pp. 46–54. PMLR, 2017.

Nesterov, Y. Efficiency of coordinate descent methods on huge-scale optimization problems. *SIAM J. Optim.*, 22: 341–362, 2012.

Raff, E. Jsat: Java statistical analysis tool, a library for machine learning. *J. Mach. Learn. Res.*, 18:23:1–23:5, 2017.

Raff, E., Hamilton, B. A., and Sylvester, J. Linear models with many cores and cpus: A stochastic atomic update scheme. *2018 IEEE International Conference on Big Data (Big Data)*, pp. 65–73, 2018.

Recht, B., Ré, C., Wright, S. J., and Niu, F. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *NIPS*, 2011.

Ruder, S. An overview of gradient descent optimization algorithms. *ArXiv*, abs/1609.04747, 2016.

Saha, A. and Tewari, A. On the nonasymptotic convergence of cyclic coordinate descent methods. *SIAM J. Optim.*, 23:576–601, 2013.

Shalev-Shwartz, S. and Zhang, T. Stochastic dual coordinate ascent methods for regularized loss. *J. Mach. Learn. Res.*, 14:567–599, 2013a.

Shalev-Shwartz, S. and Zhang, T. Stochastic dual coordinate ascent methods for regularized loss minimization, 2013b.

Tran, K., Hosseini, S., Xiao, L., Finley, T., and Bilenko, M. Scaling up stochastic dual coordinate ascent. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 1185–1194, 2015.

Zhang, H. and Hsieh, C.-J. Fixing the convergence problems in parallel asynchronous dual coordinate descent. *2016 IEEE 16th International Conference on Data Mining (ICDM)*, pp. 619–628, 2016.

Zhang, H., Hsieh, C.-J., and Akella, V. Hogwild++: A new mechanism for decentralized asynchronous stochastic gradient descent. *2016 IEEE 16th International Conference on Data Mining (ICDM)*, pp. 629–638, 2016.

Zhao, S.-Y. and Li, W.-J. Fast asynchronous parallel stochastic gradient descent: A lock-free approach with convergence guarantee. In *AAAI*, pp. 2379–2385, 2016.

Zinkevich, M., Smola, A., and Langford, J. Slow learners are fast. In *NIPS*, 2009.

Zinkevich, M., Weimer, M., Smola, A., and Li, L. Parallelized stochastic gradient descent. In *NIPS*, 2010.

APPENDIX A.

Result of the convergence in the 20 cores machine are given in Figure 8 and Figure 9.

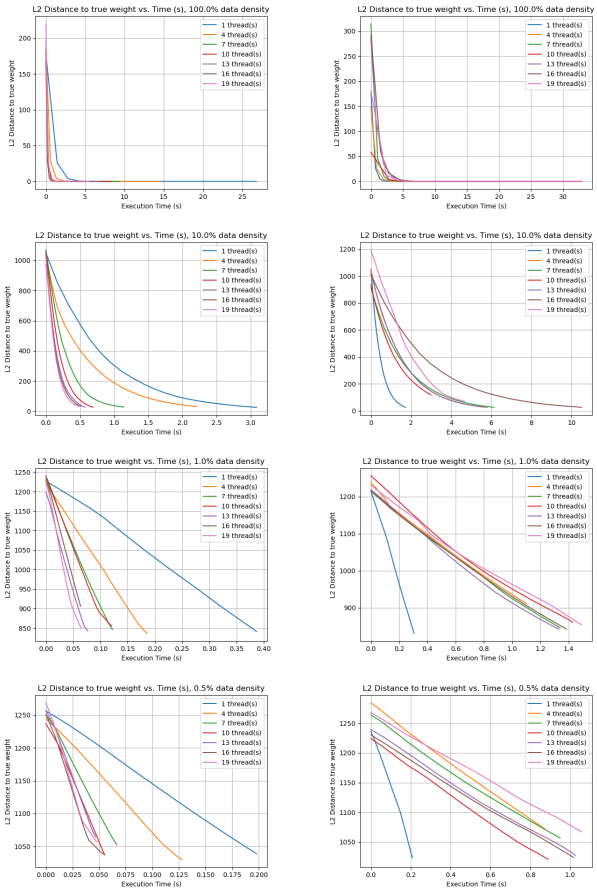

Figure 8. Execution time(s) vs. L2 distance to the true value, training linear regression with Hogwild (left column) and the Naive method (right column) in different data density, training was conducted in 20 cores machine

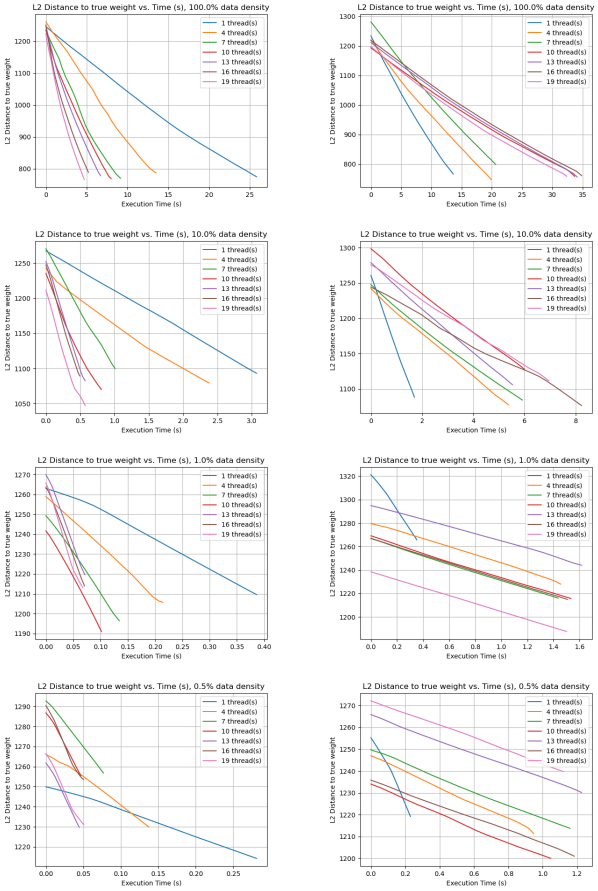

Figure 9. Execution time(s) vs. L2 distance to the true value, training logistic regression with Hogwild (left column) and the Naive method (right column) in different data density, training was conducted in 20 cores machine