Lock-free Parallel Asynchronous SGD

Nelson Antunes, Manuel Cherep, Olivier Cloux, Cosmin-Ionuţ Rusu
Systems for Data Science (CS-449), EPFL, Switzerland

Abstract—**Stochastic Gradient Descent (SGD) is a widely used algorithm that achieves state-of-the-art performance on many machine learning tasks. Different algorithms have been proposed to parallelize SGD both synchronously and asynchronously. This work aims to implement a lock-free parallel asynchronous SGD on one machine using multiple processes. Moreover, we compare the results with a lock-safe approach and the original implementation of Hogwild-python [1]. As expected, we show that the absence of locks improves the performance without compromising the overall correctness of the result, by removing the locking overhead.**

I. Introduction

With the rise of Big Data, machine learning has come to the center of attention of many disciplines. However, if the amount and rate of collected data never ceases to grow, our machines reach a phase of "stalling". The everlasting exponential growing of computing power has dramatically slowed down, and all our attempts to overcome physical limitations have failed. To counter this problem, data science has turned towards systems composed of multiple machines.

Stochastic Gradient Descent (SGD) [2] is one of the most popular methods to optimize machine learning algorithms and has proved to be robust and simple simultaneously. The algorithm *Hogwild!*[3] proposes a way to split the computation asynchronously on multiple processes or machines while conserving correctness for sparse problems.

Our paper focuses on implementing SGD asynchronously on one machine using multiple processes. This ensures the scalability of the algorithm for enormous datasets while conserving correctness. Then, we focus on comparing our implementation with two asynchronous implementations [1] using a lock in two different ways, in order to find the best-quality solution.

The mathematical guarantees of Hogwild! are outside of the scope of this paper, but can be seen in detail in the original paper. We focus on solving a classification task over the Reuters (RCV1) dataset (See Section III-A), optimizing the Support Vector Machine (SVM) algorithm with SGD in across multiple workers locally.

II. Parallel SGD

A. Synchronous

One horizontally scalable implementation of parallel SGD can be achieved by having a master and multiple workers in a cluster of machines. The master splits the data equally among the workers which compute the gradient update rule and send their results to the master. The master then updates the state of the global weight vector, waiting for each of them to finish. After one such iteration, also called a *synchronous epoch*, the master repeats the process until convergence. This approach is limited performance-wise since in each epoch, the fastest workers will stay in the idle state waiting for the slowest worker to finish. Furthermore, the number of message passed in such a distributed approach scales linearly with the number of workers.

B. Asynchronous

In an asynchronous fashion, each node sends their gradients to all the other nodes, updating their view of the weight vector. The number of messages scales quadratically in such a design, but this will have much better performance in terms of running time given a reliable network. Moreover, assuming a sparse problem there is a guaranteed convergence even when locks are removed. The mathematical proof of this result is beyond the scope of this paper, but it can be seen in the Hogwild! paper [3].

Another asynchronous approach is a vertically scalable one, using processes on a single machine instead of multiple machines on a single cluster. In this case, message passing delays can be avoided by using shared memory, yielding a faster running time.

C. Architecture

Native Python does not support multithreading, therefore we decided to use a process-based approach where we spawn multiple processes that have access to a lock-free read and write memory space. We have also implemented a safe read and write mechanisms using locks for the sake of comparison.

Two type of processes are created in our architecture. Both of them have access to a read only memory where the train dataset is stored, and to a read-write memory for manipulating the weights.

1) Master process: The master process loads the train dataset into a read-only memory, creates and initializes the shared read-write memory for the weights and finally spawns multiple worker processes. It also creates a queue where the workers put the weight updates for the master to compute the validation loss. The master uses the validation loss to check the early stopping criterion. If convergence is achieved, the master terminates all the worker processes.

2) Worker process: A worker process is dedicated to compute the gradient and to update the weights. The process has access to the weights, the train set and a queue where it puts the weights resulting after a batch of iterations.

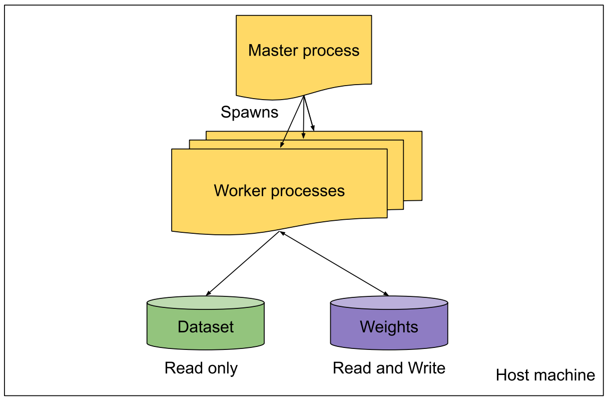

Fig. 1. Proposed architecture

III. Implementation of Parallel SGD

A. Reuters RCV1 Dataset

Reuters Corpus Volume I (RCV1) [4] contains over 800,000 manually categorized newswire stories in English made available by *Reuters, Ltd.* for research purposes. There are 103 topics and we focus on the topic "CCAT" (Corporate/Industrial) for the task of classification. A row can have up to 47,237 columns, nevertheless on average only 77 of them are non zero. Therefore, the problem is very sparse and ideal for using the approach in Hogwild!.

B. Support Vector Machine (SVM)

Support Vector Machine (SVM) [5] is the algorithm that we are optimizing using SGD. We use SVM to perform a supervised binary classification to detect whether a Reuters news article belongs to the topic category "CCAT" or not. The SVM tries to find the hyperplane that divides both classes (i.e. belong or not belong) with the maximum margin. The associated loss is:

$$\mathcal{L}(\boldsymbol{w}) = \frac{1}{n}[\sum_{i=1}^{n} max(0, 1 - y_i(\boldsymbol{w}\boldsymbol{x}_i - b))] + \lambda \left\|\boldsymbol{w}\right\|^2 \quad (1)$$

where $max(0, 1 - y_i(wx_i - b))$ is the *Hinge* loss function, y_i is the label of data point x_i, w is the vector of weights and b is the bias. The bias is removed from the equation and it is added as a column of ones to the data. We also introduce a regularization parameter λ which penalizes large weights thus avoiding overfitting.

1) Data Handling: Each worker process receives a reference to the shared memory where the data is stored (i.e train set and weights). The master process is responsible for loading that data into memory and passing the reference to the workers. The training set is read-only, while the weights can be read and written.

2) Validation Loss and Accuracy: The master is in charge of calculating the validation loss after each asynchronous epoch, where an epoch represents a batch of iterations. Calculating the validation loss is done in batches to avoid creating a bottleneck. Moreover, the master also calculates the train and test accuracies at the end of all asynchronous epochs.

3) Learning Rate: The update rule for the weights is

$$\boldsymbol{w} := \boldsymbol{w} - \eta \nabla \mathcal{L}(\boldsymbol{w}) \quad (2)$$

where $\boldsymbol{w}$ is the vector of weights, η is the learning rate and $\nabla \mathcal{L}(w)$ is the gradient of the loss function. Our learning rate is

$$\eta = \frac{0.03}{|workers|} \quad (3)$$

where $|workers|$ is the number of workers. We scale inversely with the number of workers to avoid divergence.

4) Stopping Criterion: We implement a stopping criterion known as Early Stopping [6]. This is a form of regularization that is used to avoid overfitting. In this case we keep a window

$$window = 100 * |workers| \quad (4)$$

where $|workers|$ is the number of workers. Thereby if the loss in the last iteration does not decrease with respect to the minimum loss in the window, we stop. Otherwise we run for a total of 1000 asynchronous epochs.

A different stopping criterion, which has proven to be more efficient, is explained in Section IV-B.

C. Process-based Implementation

Python uses a Global Interpreter Lock (GIL) - a mutex that allows only one thread to hold the control of the Python interpreter. This guarantees that only one thread can be in the state of execution at any point in time. While it does not impact single-threaded programs, it can be an important performance bottleneck in multi-threaded code.

Since we need to implement our algorithm in a parallel style, two solutions are available. On the one hand, we have other Python interpreters that do not implement the GIL, such as CPython, Jython, and PyPy. On the other hand, we have a more clean approach, using processes instead of threads. We have chosen to follow the latter approach using the *multiprocessing* package. This packages gives us an elegant way to create and wait for processes, and it provides simple ways to create shared memory spaces. *RawArray* provides a lock-free read-write shared memory space, while *Array* can be synchronized using a lock for reading and writing, giving us a simple way to switch the lock-free parallel version with the safe version.

All the worker processes have an object reference to both the dataset and the weights vector, as well as a reference to a queue where they have to put the weights after an asynchronous epoch. In addition, the master process also needs references to the worker processes so that it can terminate them as soon as the early stopping condition holds. The master uses the queue to get the weights from each worker and calculate the validation loss.

IV. Experiments

In order to analyze and compare our asynchronous implementations (i.e. with and without a lock) with the previous asynchronous implementation [1], we focus on three metrics:

- Running Time: The timing performance for the training, excluding reading data and calculating accuracies.
- Test Accuracy: The accuracy over the test dataset once the training has finished.
- Validation Loss Convergence: The convergence of the validation loss over time and its rate.

Our goal is to find the implementation that produces the best-quality result in the terms described above. In order to do so we run experiments exploring a different number of workers.

All these experiments were run under the same conditions in a personal computer with *24GB* of RAM and Processor i7-3820 at 3.8 GHz. Substantial memory is required to calculate the accuracy over the *14GB* of test set. Moreover, as seen in Section III-B we implement a comparable stopping criterion and the same learning rate. All the results shown below are the result of calculating the median over multiple runs to reduce the bias.

A. Number of Workers

Experiments were run for different number of workers in the interval $[1, 10] = \{x \in \mathbb{N} \mid 1 \leq x \leq 10\}$ in all three different implementations. Figure 2 shows that the execution time in both asynchronous with a lock implementations increases exponentially. This is due to the fact that locking is computationally expensive. The previous asynchronous implementation locks only for writing, while our implementation locks for both reading and writing. However, as we can see our implementation is as efficient as the previous one.

Contrarily, we can observe that our asynchronous with no lock implementation (i.e. Hogwild!) scales better with respect to the number of workers. The execution time remains nearly constant independently from the number of workers when there is no lock. The fact that it is not strictly constant is due to the fact that some communication is necessary to centralize the early stopping criterion (See Section IV-B).

Figure 3 shows the test accuracy for different number of workers and implementations. Both implementations with a lock achieve similar test accuracies. The main difference between those accuracies can be explained by the different stopping criteria. Stopping earlier, which is usually the case in our asynchronous with lock implementation, can yield a lower test accuracy. Contrarily, the accuracy for the asynchronous implementation with no lock increments until it becomes stable. Nonetheless, we cannot conclude that either implementation yields better results because they depend on multiple hyper-parameters that make it non deterministic (e.g. random validation or early stopping criteria).

Finally, we analyze the convergence of the validation loss. Figure 4 shows the validation loss over time in all three implementations for a different number of workers.

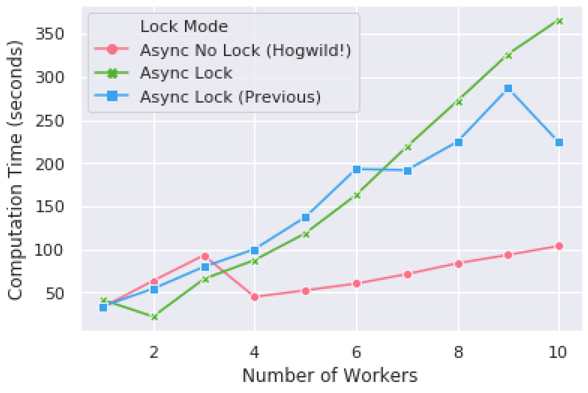

Fig. 2. Running time vs number of workers

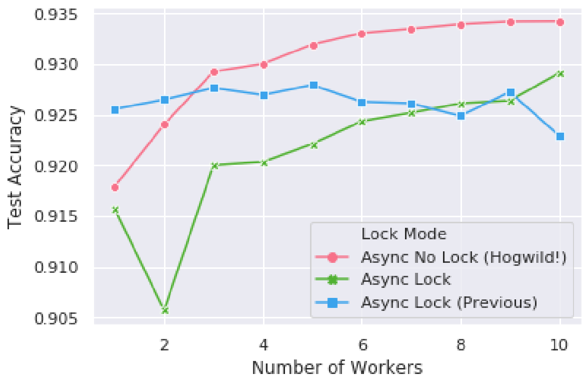

Fig. 3. Test accuracy vs number of workers

The implementation with no lock is clearly converging slower, and the validation loss fluctuates much more. This is expected since weights are sometimes overwritten. Contrarily, the implementations with a lock converge at a similar rate and their loss fluctuates less. However, the validation loss in our implementation fluctuates less which is probably due to the fact that we use a lock for both reading and writing. In all implementations, the convergence rate is higher as the number of workers increases.

B. Early Stopping

In Section III-B we described our stopping criterion. This stopping criterion aims to establish fair comparable experiments with respect to the previous implementation. However, the queue required to allow the master to calculate all the validation losses in the given order is an inefficient approach. It requires synchronization that is time consuming and as the number of workers grows the performance is deteriorated.

Alternatively, we implemented a more efficient stopping criterion where the master controls whether the weights have changed or not and calculates the validation loss. Since calculating the validation loss consumes more time than the SGD iterations, some weight changes are not observed by the master. Therefore, this approach yields an uniform sampling

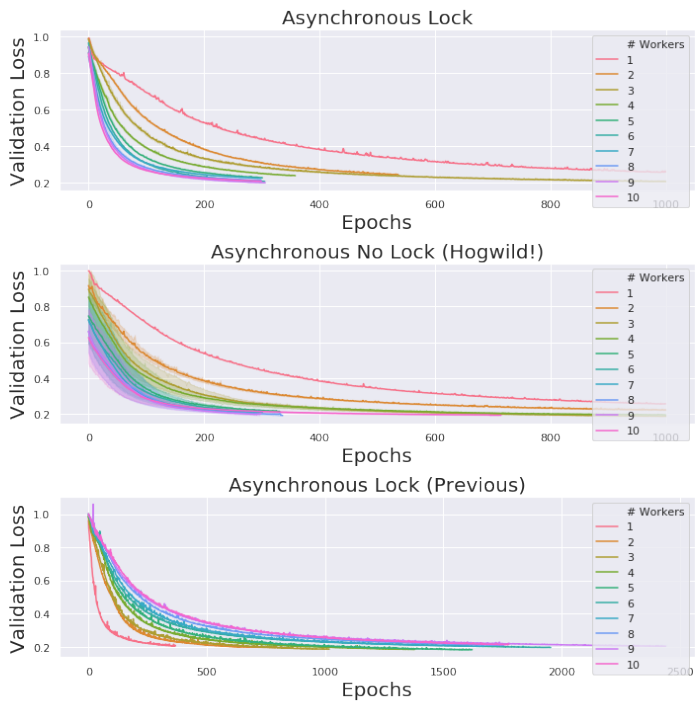

Fig. 4. Convergence of validation loss over time for different number of workers

of the weights changes because calculating the validation loss is constant in expectation. However, the uniform sampling is enough to observe the loss convergence and apply an stopping criterion. The stopping criterion needs a smaller window due to the uniform sampling.

Figure 5 shows running time compared to the number of workers, using the original stopping criterion versus sampling uniformly.

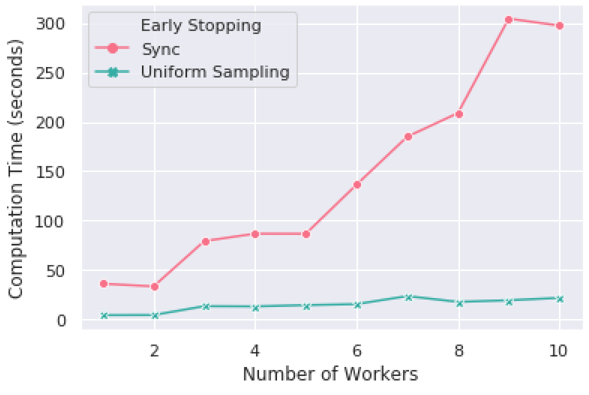

Fig. 5. Running time vs number of workers

As expected, eliminating the synchronization required by the queue and uniformly sampling instead, results in a faster execution that remains constant as the number of workers grows.

V. Discussion

The results shown in Section IV are the median of at least five different trials in order to find relevant experimental results. However, in order to draw significant statistical conclusions it would be necessary to run more experiments.

We have not been able to replicate the same results with the previous asynchronous implementation. They conclude that the asynchronous approach yields stable running time results, while we systematically see an increasing trend. This made difficult to actually find a similar stopping criteria that would be fair to make the comparisons.

We decided not to explore batch sizes in this case since it is irrelevant. The batches are only used to avoid calculating the validation loss for each iteration, which would create a bottleneck.

As seen in Section IV-B, sampling the weights uniformly after some iterations and calculating the validation loss is way more efficient that using a queue. However, we decided to present the slower approach to make fair comparisons with the previous Hogwild! implementation. We believe that our comparison shows only the difference between using a lock and no lock at all, without an extra variable (i.e. early stopping) that would condition all the results.

VI. Conclusion

The results shown in Section IV emphasize the importance of an efficient implementation of SGD. A faithful implementation of the original Hogwild! is faster than the asynchronous implementations using locks on the weights.

Our Hogwild! implementation of asynchronous SGD scales better than the compared asynchronous implementation, and it achieves a high accuracy systematically. This is likely due to a more efficient implementation without locking.

Lastly, all the asynchronous implementations compared are more efficient than a synchronous implementation time wise. However, it is important to remark that the asynchronous solution is suitable for sparse problems with no detriment for the test accuracy, and therefore it is not a silver bullet.

References

[1] M. A. Liamarcia Bifano, Roman Bachmann, “Distributed asynchronous sgd,” https://github.com/liabifano/hogwild-python, 2018.

[2] H. Robbins and S. Monro, “A stochastic approximation method,” *The annals of mathematical statistics*, pp. 400–407, 1951.

[3] F. R. Benjamin; Re Christopher; Wright Stephen J. Niu, “Hogwild!: A lock-free approach to parallelizing stochastic gradient descent. eprint arxiv:1106.5730,” 2011.

[4] D. D. Lewis, Y. Yang, T. G. Rose, and F. Li, “Rcv1: A new benchmark collection for text categorization research,” *Journal of machine learning research*, vol. 5, no. Apr, pp. 361–397, 2004.

[5] C. Cortes and V. Vapnik, “Support-vector networks,” *Machine learning*, vol. 20, no. 3, pp. 273–297, 1995.

[6] L. Prechelt, “Early stopping-but when?” in *Neural Networks: Tricks of the trade*. Springer, 1998, pp. 55–69.