Lock-free Parallel SGD in Python

Timoté Vaucher, Patrik Wagner, Thevie Mortiniera

***Abstract*—This is the second milestone of a project based on a synchronous and asynchronous distributed version of Hogwild implemented in Spring 2018 during the course CS-449 "Systems for Data sciences" [1] and a Spark version implemented in the first milestone. This system exploits multi-core processing capabilities by running the algorithm in parallel over multiple processes, while using unprotected shared memory to concurrently modify a common representation of the weights vector as suggested in HOGWILD! [2]. We aim at showing the improvement of this new version compared to previous methods.**

I. Introduction

Stochastic Gradient Descent, referred to as SGD in this report, is one of the most popular training algorithm used in Machine Learning. However, its iterative nature makes it inherently serial. One way to overcome this obstacle is through parallelization over mini-batches. Because the size of the mini-batches is crucial for the performance accuracy and is also limited by computer memory, the need to find a fast and stable solution for parallelizing training over multiple independent nodes (computers) to achieve higher speedups has become more and more important. One such solution would shrink the development costs and would help resolve scalability issues. During the past years, multiple solutions have been proposed to tackle this challenge [3].

In this report, we focused on a python implementation of the original **HOGWILD!** paper [2], and proposed a faithful implementation in pure Python which exploits multi-core processing capabilities. We ran the algorithm in parallel over multiple processes due to the presence of the GIL in python. It uses unprotected shared memory to concurrently modify a common representation of the weights vector. We were asked to run different experiments and determine which of the three, among our multiprocessing implementation in pure Python, the "local" mode of Hogwild-Python from last year [1] and a slightly modified version that does use locks, produces the best-quality results in the shortest amount of time.

As in the previous implementations, we were given the Reuters RCV1 Dataset, which contains pre-classified English news. The training dataset was quite balanced since $9'653$ are labeled as "CCAT" and $11'154$ don't have a "CCAT" label, which gives us a $46/54\%$ ratio. It was composed of $23'149$ samples for training, which split between a training set and a validation set representing 10% of the total set, and $781'265$ samples for the test set. Only 77 columns out of the $47'237$ features are filled on average, turning the problem into a very sparse one [1].

II. Multiprocessing Design

A. Lock-free design

In this parallel implementation, we took advantage of recent computers having multiple cores that share memory, usually the RAM, as suggested in **HOGWILD!** [2]. We decided to implement our solution using the `multiprocessing` library of python due to the presence of the Global Interpreter Lock which doesn't allow multiple threads of the same process, i.e. running on the same interpreter, to execute instructions in parallel. In order to be able to access the same weight vector across all processes, we used a `multiprocessing.sharedctypes.Array` which creates an array allocated from shared memory. In our lock-free version, we passed `lock=False` when initializing the array.

We also decided to split equally the training data among the processes as well as the validation data to be able to compute the validation loss and therefore the convergence of the optimization step. We represented the data as dictionaries as implemented in **Hogwild-Python** [1] with 'features index – value' pairs as a basis to create it. For the interested reader, we also provide a more state-of-the-art solution using Scipy sparse matrices and taking advantages of their operations to get rid of slow python for-loops resulting in a speed up of a factor 2. The rest of this report uses the native python solution as a baseline as we were notified that our solutions had to be restricted to the standard library. We also kept the Loss function from our first milestone as shown below.

$$\mathcal{L}(w) = \sum_{i=1}^{n} \left(max(0, 1 - y_i w^T \hat{x_i}) + \lambda \sum_{u \in x_i} \frac{w_u^2}{d_i} \right)$$

Where u are the indices of non-zero features for a given sample x_i and d_i is the number of non-zero features of x_i.

B. Non-concurrent update

In the instructions, we were also asked to implement a slightly modified version which prevents potentially-conflicting updates to the weight vector. In order to achieve this goal, we wanted to acquire a lock before writing in the array. For this, we simply set `lock=True` to wrap the array with a synchronizer using `RLock` in order to get non-concurrent writes and make them process-safe. On the other hand, to take advantage of the work of concurrent processes, we did not lock the other read accesses as it empirically improved the convergence rate.

III. Experiments

The chosen metrics to compare the different implementations were the validation loss and the accuracy. In order to find the conditions under which the best performances were achieved, we first needed to determine the effect of the mini-batch size of SGD on the running time, the regularizer and the learning rate. Those hyper-parameters were found by performing a grid-search which yielded following results : $\epsilon = 0.035$, $\lambda = 0.0001$ and batch size = 100. In addition, we ran all the experiments on an *Intel® Core™ i7-8550U* CPU with 16Gb of RAM.

A. Effect of the number of processes on the convergence time of the loss function

As we used multiprocessing, we wanted to see how the number of processes affected the validation loss convergence time.

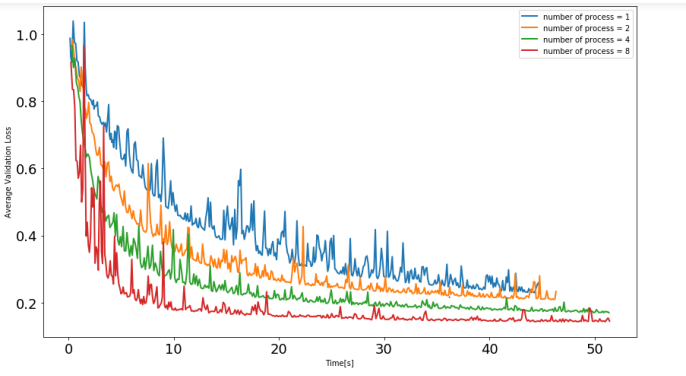

Figure 1. Number of processes versus mean time per 400 iterations per Process

As expected, our implementation took advantage of the different cores present in the test machine as the execution time was similar. We noted that the algorithm converged faster as the number of process increases. This is due to the fact that the number of elements seen per iteration is proportional to the number of processes, as each sees a 100 items per epoch.

B. Absence of locks

We wanted to compare our new lock-free architecture with the modified version that does use locks at the update step, to verify that the absence of locks really helps running time performance.

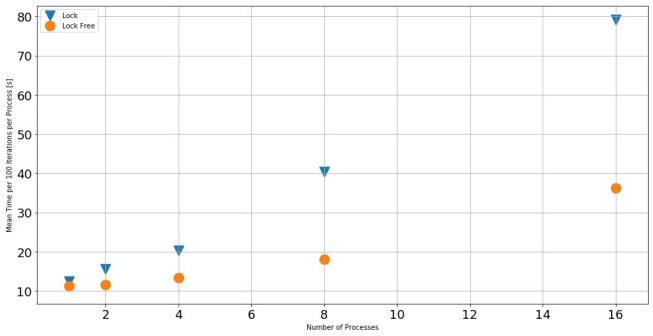

Figure 2. Number of processes versus mean time per 100 iterations per Process

When looking at the plot above, we noticed that the higher the number of processes involved, the higher the gap of performance between the lock-free and the one with locks. This follows from the fact that each process has to compete to get the lock before the update step. This empirically verified that the absence of locks actually improved the running time performance. We also tried with 16 processes, twice the number of available cores, and noted an overall loss of performance. It may be mainly due to the fact that 2 processes were competing for the resources of a single core.

C. Models comparisons

We wanted to compare the performance of our multi-core version against their synchronous and asynchronous versions and the lock version. As stated in the previous milestone, we assumed, as a rule of thumb, that their presented model was the best one. Every test was done on 4 processes, respectively 4 workers.

1) Time versus validation Loss: Although the three implementations outperformed the synchronous one, our lock-free implementation yielded better results than the asynchronous one. Finally, as expected, the implementation using locks took more time to converge to a similar average validation loss.

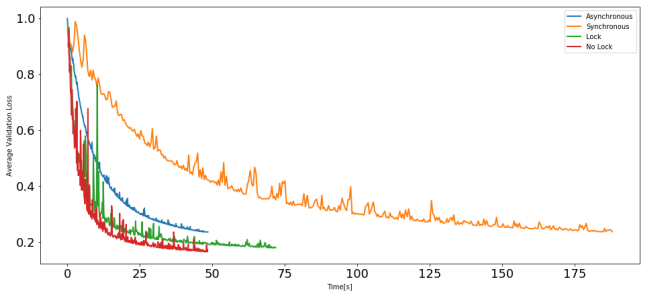

Figure 3. Average loss of the four implementations on the validation set over 400 iterations

2) Accuracy Performance: When observing figure 4, we noticed that the four implementations roughly yielded similar results with a slightly better performance for our model.

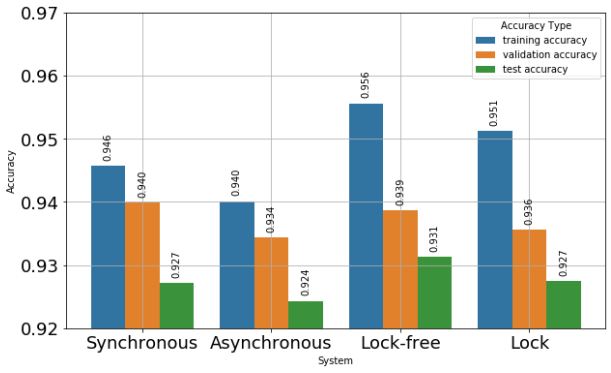

Figure 4. Accuracy performance of the four implementations on the train, validation and test set

IV. Discussion

A. Usage of memory

The first point that we want to discuss is the usage of memory by our implementation compared to Hogwild-python. The authors of **Hogwild-Python**[1] said that in order to read the test set, they needed up to $20Gb$ of RAM on the Kubernetes coordinator. In order to run their code locally, we condensed some list comprehensions in order for the test set to fit into the $16Gb$ memory of the test machine. In our implementation, we further improved the preprocessing of the test set to reduce the memory footprint of the entire program to about $5Gb$.

B. Change in the gradient of the regularizer

The lower validation loss and overall increase in accuracy achieved by our solution comes from an updated interpretation of regularization part of the Loss function

$$reg(w)_{x_i} = \lambda \sum_{u \in x_i} \frac{w_u^2}{d_i}$$

In [1], they simply computed the gradient as

$$\nabla reg(w)_{x_i} = 2\lambda \cdot \sum_{u \in x_i} \frac{w_u}{d_i}$$

And subsequently subtracted that value to every component of the gradient for x_i. Yet, we think it is more accurate to think of the above loss in its vectorial form

$$reg(w)_{x_i} = \lambda \cdot \frac{\|w_u\|^2}{d_i}$$

which yields the following gradient

$$\nabla reg(w)_{x_i} = 2\lambda \cdot \frac{w_u}{d_i}$$

To us, it makes more sense to have a regularizer that acts like the L_2*-regularizer*, that is component-wise. This is also empirically confirmed by a lower loss and an increased accuracy.

C. Comparison between asynchronous and lock-free

We finally note that the asynchronous and the lock-free model have similar running time as seen in figure 3. This make sense as both versions are running in an asynchronous manner on a single machine, that is, each process doesn't wait for the other ones and the update of the weight is non-blocking. Furthermore, the computing of the gradient and overall structure is very similar, so we can expect both code to run at a similar speed.

V. Conclusion

Throughout our project, we ran multiple experiments to assess the performance of the models using validation loss and accuracy at train and validation time as metrics. Our lock-free multi-core implementation in native Python yielded better results in a similar time-fashion as the asynchronous Hogwild-Python implementation.

We managed to exploit multi-core processing capabilities verifying experimentally that, the higher was the number of processes the faster was the validation loss convergence time. Furthermore, we empirically verified that the absence of locks really helps performance without compromising the overall correctness of the algorithm as our lock-free implementation converged in less time than the one with locks while reaching comparable results.

Finally, as already stated previously, using the full potential of external libraries such as *numpy* and *scipy* yield even better results than native python. In a future work and provided with more time and computing power at our disposal, it would have been interesting to investigate more recent algorithms like **Downpour SGD** [4], developed by researchers at Google as part of their DistBelief framework whose variants are still used in today's distributed TensorFlow framework.

REFERENCES

[1] L. Bifano, R. Bachmann, and M. Allemann. Distributed asynchronous sgd. [Online]. Available: https://github.com/liabifano/hogwild-python

[2] B. Recht, C. Ré, S. J. Wright, and F. Niu, "Hogwild!: A lock-free approach to parallelizing stochastic gradient descent." *eprint arXiv:1106.5730, 2011.*, 2011.

[3] B. Holländer. Accelerating deep learning using distributed sgd - an overview. [Online]. Available: http://tiny.cc/distribSGD

[4] G. C. Jeffrey Dean. Large scale distributed deep networks. [Online]. Available: https://www.cs.toronto.edu/~ranzato/publications/DistBeliefNIPS2012_withAppendix.pdf