CS-449: Project Milestone 2

Vincent Coriou, Damian Dudzicz, Karthigan Sinnathamby

EPFL - Data Science MA2
Email: {firstname.lastname}@epfl.ch

***Abstract*—This report constitutes the second milestone of the CS-449 course project of the group number 1 at EPFL. In this report, we present and analyze a Scala implementation of the Hogwild! [1] solution for parallel SGD. We compare the results obtained with those yielded by the Python implementation of Bifano, Bachman and Alleman [2] with respect to accuracy and speed of convergence and those yielded by a system with an explicit locking system.**

I. Introduction

In the context of the CS-449 course at EPFL, we implement for the second milestone a multi-thread asynchronous stochastic gradient descent (SGD) method to classify the data gathered by Lewis et al. [3] on a single machine with parallel threads using unprotected shared memory as suggested by Hogwild! [1].

The main idea of the algorithm presented in the paper [1] is to ensure scalability by minimizing the bottleneck caused by the locking. It is achieved by running SGD in parallel without the need for locks on the update weights and gradients matrix. In [1], authors show that good numerical results and performances can be achieved if one removes the locks on the gradients and weights vector during the updating phase as the datapoints are sparse and thus are the gradients (for a SVM). Unlike the first milestone of the project, in this report, our main concern is to design an implementation very faithful to the original Hogwild! paper.

Hence the same setting as in [1] consisting of a local execution on a multi-core system is adopted in this report. Our implementation is written in Scala and tested in a local setting on a computer with a 4-core 2.9 GHz CPU and two threads per core.

We compare our implementation to the local version of the asynchronous Hogwild python implementation by Bifano, Bachman and Alleman [2] which uses GRPC communications. We also test our implementation using explicit locking. The aim of this report is to discuss and compare the various implementations to determine which one provide the best numerical results and fastest computation time.

II. Classification problem and parameters

A. DataSet Description

We remind the reader of the general specification of the dataset considered in the problem. The classification is performed on the *RCV1* data collection prepared by Lewis et al. in [3]. The dataset consists of over 800K manually categorized English news from *Reuters Ltd.* The data collection is divided in a train set of size 20K and test set of size 780K. Although, the maximal possible number of features for each entry of the data collection is over 47K, the actual number of non-null entries for a row in the matrix representation of the data is several orders smaller than the upper-bound i.e. around 100.

Due to the very sparse nature of the data collection it is represented in a specific format where each row can be of different length and holds all the indices where the entries are non-zero i.e. the row actually corresponds to the category with this given column index.

dataset entry index	non-null categories indices
0	[1,2,4]
1	[2]
2	[3,4]

Fig. 1. Simple example of the dataset representation

Hence, the reader can observe that the Hogwild! solution is particularly adapted to this data classification problem given the sparsity of the *RCV1* dataset.

B. Classification Specification

In a similar fashion and for continuity sake, we classify the entries of the dataset with respect to the "CCAT" category, i.e. if "CCAT" class is present in a given entry list of categories as in the previous milestone report.

The classification problem is addressed with the Hogwild! modified version of the SVM classifier using batch-based SGD with a different loss function to take into account the sparsity of the data.

1) Sparse Loss Function: As mentioned in the previous subsection, in order to take into account the sparse nature of the data, we adopt the same modified loss function (for a single point) for the SVM classifier as in [1],

$$\mathcal{L}_x(\mathbf{w}) = [1 - y(\mathbf{w}^\intercal \mathbf{x})]_+ + \lambda \cdot \sum_{u \in \mathbf{e}} w_u^2 \tag{1}$$

where $\mathbf{x} \in \mathbb{R}^{d+1}$ corresponds to the augmented vector of the datapoint with a prepended term which takes into account for the bias term present in the SVM expression.

$y \in \{-1, 1\}$ corresponds to the label of the vector i.e. 1 if the entry is present in the category "CCAT" and -1 otherwise.

$w \in \mathbb{R}^{d+1}$ is the weight vector considered for our classification.

$\mathbf{e}$ is the set of all non-zero coordinates of the datapoint $\mathbf{x}$. Eventually, $\lambda \in \mathbb{R}$ is the regularization term.

2) Gradient Computation and Weights Vector Update: We compute the gradient for each datapoint $x \in B$ and merge the obtained vector by taking the average of the sum of the values for each coordinate u of all the vectors (if an index is missing for a gradient it is implicitly set to 0). Thus the computation can be expressed in the following manner. Let us consider the gradient of a single datapoint $x \in B$ from the batch with respect to a non-zero coordinate u:

$$\partial_u \mathcal{L}_x(\mathbf{w}) = \begin{cases} 0 & \text{if } x_u = 0 \\ 2\lambda w_u & \text{if } 1 - y(\mathbf{w}^\intercal \mathbf{x}) < 0 \\ -yx_u + 2\lambda w_u, & \text{otherwise} \end{cases} \quad (2)$$

Then the gradient of the loss for a coordinate u and this batch B can be computed as follows:

$$\partial_u \mathcal{L}(\mathbf{w}) = \frac{1}{d_{b_u}} \sum_{x \in B} \partial_u \mathcal{L}_x(\mathbf{w}) \quad (3)$$

where d_{b_u} corresponds to the number of points in the considered batch B which have a non-zero coordinate u.

Finally, the update of the weights vector $\mathbf{w}$ is performed in a classical fashion

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t+1)} - \gamma \cdot \partial \mathcal{L}(\mathbf{w}) \quad (4)$$

where γ corresponds to the chosen learning rate.

III. Hogwild! Implementation

A. Final Asynchronous Implementation

The implementations is written in Scala. It relies on a thread pool of W threads where each corresponds to a worker computing gradients on some points of the train set. The train set is split among all the workers after being shuffled, that is each worker has access to a set of size $\frac{|\text{train dataset}|}{|W|}$.

Each thread acquires a copy of the weights vector and takes a batch of size B from the set it is assigned to. Then it computes a gradient to update the weights vector using this copy of the weights vector. Each worker update the weights vector using unprotected shared memory. At the next iteration, each worker will iterate over the next B elements from its respective set and so on until, it has seen all the set it was assigned to at which point it restarts with the B first elements.

Adopting this solution guarantees to go over the full dataset and not to omit some points as it could be the case with random sampling.

Each thread runs independently of the other threads and do not wait until the completion of the computation of all the other workers to pursue. The threads gets information from the others work through the weight vector that is shared among all of them as they acquire a private copy at the beginning of their respective SGD. At the end of an iteration, each worker acquires a more recent copy of the weights vector (after it has been updated by the worker) and compute the validation loss to be logged.

An early stopping is implemented in order to stop the iterations as soon as the loss has reasonably converged.

B. Previous Implementations Attempts

We first considered using a shared `TrieMap` from the concurrent Scala library as a parallel data structure to merge the gradients updates of all the workers before updating the weights on an additional worker $W + 1$ which does not compute the gradients. The data structure is presented as concurrent and allows atomic update of its entries. It has not be considered in our final implementation as it didn't allow a good scaling with the increase of the number of workers, in particular due to this worker $W + 1$. In addition, the previous implementation was significantly slower to our current version.

C. Implementation with Lock

To see whether it is the fact that we do not lock the weights that speeds things up, we consider adding a locking system to our implementation. This is done with the `scala.concurrent.Lock` object.

We consider two locking systems consisting of the minimal locking setting and the most restrictive one.

The first one (*Lock Weight*) locks the weight vector and the second one (*Lock Gradient*) locks all the gradient computation process done by a worker. The former is a minimal locking (we cannot lock less) while the latter is a maximal locking (we cannot lock more).

In other words, with *Lock Weight*, each worker request a lock when he copies the weight vector (that is twice by mini-batch iteration - one to start the gradient computations and another one before computing the loss for the current mini-batch) and when he wants to update it. With *Lock Gradient*, $W - 1$ workers have to wait for the worker currently working to release the lock.

IV. Experiments

A. Preliminary remarks

In this project, we are interested in comparing the performances of three different implementations: our implementation of Hogwild! with and without locking and also the local version of the asynchronous mode of Hogwild-python from [2]. We desire to see which one is the fastest to produce a final result and the one that yields the best numerical values.

We adopt as a metric of performances the value of the loss computed for a validation set. It is important to note, that the implementation realized for Hogwild!-Python has a different loss function definition and a different gradient corresponding to it. This might result in some uncertainties in the comparison of results of [2] and those of our asynchronous solution. Eventually, we also compute the accuracy of each model over a test set.

Furthermore, the notion of mini-batch is not present in the original Hogwild! [1]. Indeed, the paper put forward the speed-up achieved by not using lock on the weights and formulate the problem as a pure online SGD that is each worker does an update with a single datapoint and not a mini-batch. To speed things up, Hogwild! Python and our implementation are implemented using batches even though the core operation remains an online SGD. That is an update is done after seeing

all the datapoints of a mini-batch but the gradient are computed point by point and not directly on the mini-batch.

Moreover, tuning the batch size has proven not to be necessary in the Milestone 1 due to this very last fact.

B. Protocol

In order to compare the performances, we run the trainings of each implementation with *a number of workers going from 1 to 10* and a fixed batch size $B = 256$. Note that in order to have a more robust metric on the performances of the various systems and settings it would require to run several times the same experiment in order to reduce the variance, we however have decided not to pursue in this direction as the variations were not significant and it also induced important computations on our machine which computational power is not only dedicated to this project.

In addition, the experiments are run with the following additional parameters *We choose an early-stopping patience of 25, a regularization term* λ *of* 10^{-5} *and a learning rate* γ *of 0.06.*

We record the validation loss as a function of time and compute the test accuracy at the end of the training.

V. Results

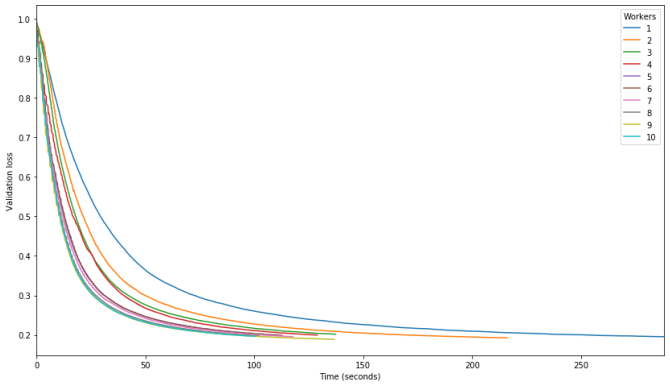
In the following subsection we present and comment in depth the results obtained while running the experiments for a varying number of workers.

A. Results Comparison

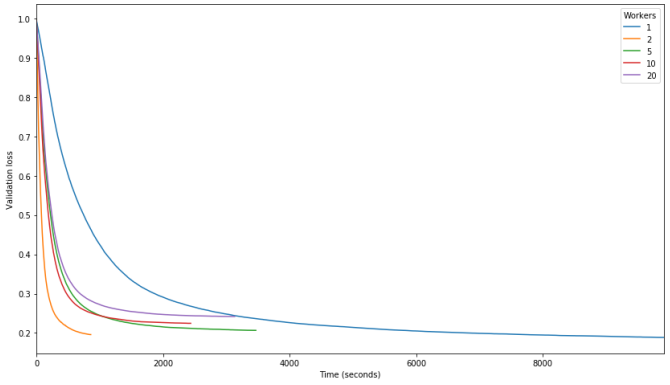
1) ***Validation loss as a function of time****:* The first remark concerning (Fig. 3) to be made regards the speed-up achieved by our implementation without lock compared to Hogwild! Python. This figure depicts the time needed to achieve 1.01 of the lowest loss (to counter long patience of the early-stopping) versus the number of workers. Our implementation takes less time to train considering a number of worker more than 2.

Another remark (Figure 2) is that with the increase of the number of workers, the execution speed of our implementation also increases. This is indeed the expected behaviour from an asynchronous parallel system. As such, we have a significant speed-up (quasi linear) with the number of workers. However, after a number of workers greater than 4, we cannot observe this linear speed-up but we may assume to some extent that this tendency continues. As our implementation is run on a local computer with 2 threads per core for a total of 4 cores, we cannot genuinely expect that all the threads are solely dedicated to our trainings (OS and environment requirements to be taken into account). As such, from our observations, we cannot conclude that this property holds for an arbitrary large number of workers, nonetheless we may consider it plausible given the convergence rate on the plots of Figure 2.

Whereas in the case of Hogwild!-Python, this trend follows an opposite tendency. With the increase of the number of workers, the computation time required for the trainings also increases. According to the Figure 3, this tendency almost follows a positive linear relation. The most probable reason



(a) Our implementation without lock



(b) Hogwild-Python! asynchronous local mode

Fig. 2. Performance Comparison

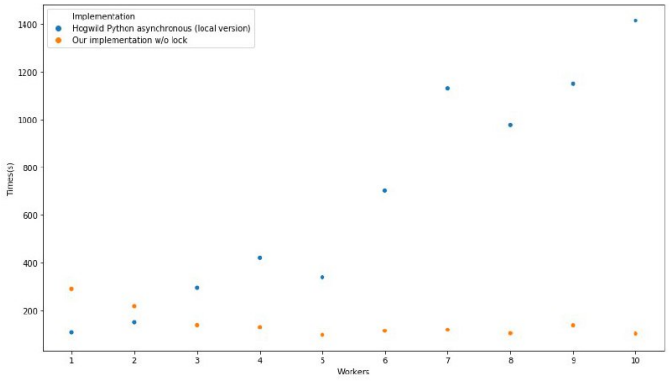


Fig. 3. Total Runtime - time needed to achieve 1.01 of the minimal loss - versus worker)

for this consists in the fact that Hogwild!-Python uses communication protocol between the workers and this constitutes the bottleneck of the system. Due to the architecture proposed by Bifano et al. in [2] the number of transmitted messages increases with the number of additional workers in order to maintain communication between the working nodes. Thus the messages passed induce a worsening of the performances.

2) ***Test Accuracy****:* The reader can observe that the loss values achieved by our implementation (Fig. 2) are comparable to those of the Hogwild!-Python implementation. Those are

generally around 0.22.

This similarity can also be observed in the test accuracy. Figure 4 displays the accuracy computed on the test set for the two systems as a function of the number of workers. The values are computed at the end of the end of the training.

We observe that both systems achieve high test accuracy (around 0.932 for Hogwild!-Python and around 0.925 for our implementation). Even though our implementation has a slightly lower accuracy, the speed-up presented and discussed in the previous section makes our implementation more interesting and efficient. Moreover, one can notice that the test accuracy remains in general constant regardless of the number of workers for both systems, thus displaying a relative stability.

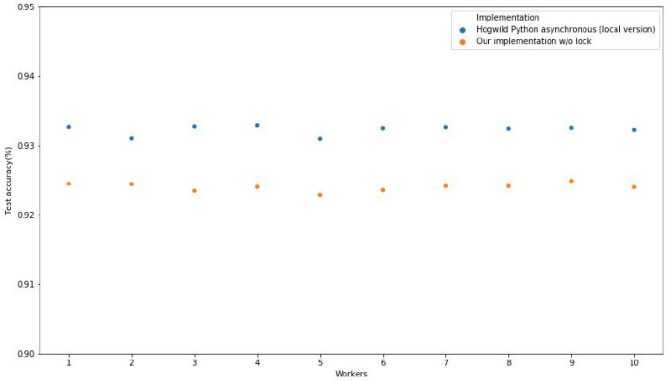


Fig. 4. Test Accuracy versus the number of workers

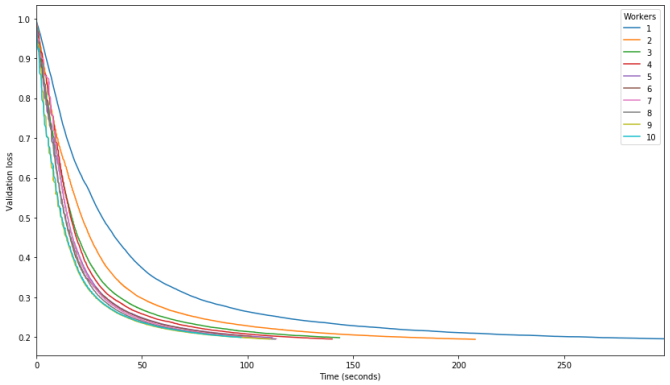
3) ***Locking****:* In this subsection, we discuss the results with regards to the locking mechanisms we have described in Section III-B.

We can see that in Fig. 5a, that *Lock Weight* requires a similar training time (or greater) than our implementation with no lock. *Lock Weight* only locks the weights vector, while the gradient computation over the mini-batch for each worker constitutes the bottleneck of our implementation. Hence, due to the sparsity of the considered problem, locking of the weights does not have a meaningful impact on our implementation.

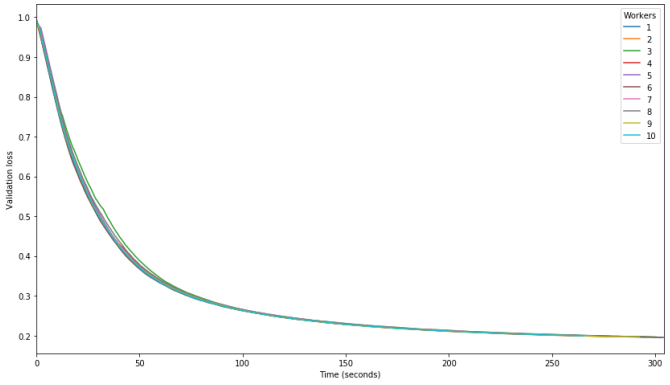
Considering *Lock Gradient* performance in Fig. 5b, we observe that regardless of the number of workers, all computation times equal the time required by our lock-free solution with only one worker. Note that this time constitutes the worst performance observed for our lock-free setting. This constitutes of an expected behaviour as all the workers are required to wait for the worker currently working to release the lock acquired on the gradient.

Both of the considered lock implementations do not compromise the correctness of the algorithm and the test accuracy of the problem as it is expected from a classical thread-safe execution of SGD.

Note that in the original Hogwild! paper [1], as previously mentioned, there is no notion of mini-batch but only of an online SGD. Indeed, Niu and al. considers the setting were a subset of non-null coordinate are considered in opposition to a sample of datapoints. The former corresponds to the setting



(a) Our implementation with Lock Weight



(b) Our implementation with Lock Gradient

Fig. 5. Performance of our implementation with lock

where each worker computes the gradient for one datapoint at a time and does the update after this single computation. This implies that each worker needs to request for a lock on the weight vector for each datapoint gradient. Thus, in such a system, with a ”real” lock, one could expect to observe a clear difference for the cases with and without lock.

In our implementation, we cannot properly put forward this phenomenon in our experiments. Indeed, due to the concept of mini-batch and the Scala *map*, it is not possible to consider such a locking system in our implementation.

Nevertheless, the presented plots show indirectly the phenomenon. *Lock Weight* and *Lock Gradient* lower and upper bound respectively the training time of the ”real” locking system. Indeed, as said in III-B, these locking settings correspond to the the limit cases of the possible locking systems for our problem. Hence, we demonstrate indirectly with our results that without locking, the system, indeed, works faster for the same numerical performances (test loss value and accuracy).

VI. Conclusion

Overall, in this report we present an implementation that is faithful to the original Hogwild! [1] design. In addition, our solution is faster and scales better than Hogwild!-Python with the number of workers. Finally, we demonstrate in an indirect manner that our system is indeed faster without the use on any locking system as suggested by [1].

REFERENCES

[1] F. Niu, B. Recht, C. Re, and S. J. Wright, "Hogwild!: A lock-free approach to parallelizing stochastic gradient descent," 2011.

[2] R. L. Bifano and M. Allemann. Distributed asynchronous sgd. [Online]. Available: https://github.com/liabifano/hogwild-python/blob/master/report.pdf

[3] D. D. Lewis, Y. Yang, T. G. Rose, and F. Li, "Rcv1: A new benchmark collection for text categorization research," *J. Mach. Learn. Res.*, vol. 5, pp. 361–397, Dec. 2004.