Asynchronous SGD for DNN training on Shared-memory Parallel Architectures

1st Florent Lopez
Innovative Computing Laboratory
University of Tennessee
Knoxville, USA
flopez@icl.utk.edu

2nd Edmond Chow
College of Computing
Georgia Institute of Technology
Atlanta, USA
echow@cc.gatech.edu

3rd Stanimire Tomov
Innovative Computing Laboratory
University of Tennessee
Knoxville, USA
tomov@icl.utk.edu

4th Jack Dongarra
Innovative Computing Laboratory
University of Tennessee
Knoxville, USA
dongarra@icl.utk.edu

***Abstract*—We present a parallel asynchronous Stochastic Gradient Descent algorithm for shared memory architectures. Different from previous asynchronous algorithms, we consider the case where the gradient updates are not particularly sparse. In the context of the MagmaDNN framework, we compare the parallel efficiency of the asynchronous implementation with that of the traditional synchronous implementation. Tests are performed for training deep neural networks on multicore CPUs and GPU devices.**

***Index Terms*—Deep learning, Stochastic Gradient Descent, Asynchronous iterative methods, multicore CPU, GPU**

I. Introduction

The Stochastic Gradient Descent (SGD) algorithm is widely employed for training machine learning models such as Deep Neural Networks (DNNs). One of its most popular variants, namely the minibatch SGD, not only offers good convergence properties but is also easily parallelizable. However, parallel implementations of minibatch SGD can be inefficient and can have poor speedups due to the need for synchronization [1]. One solution to alleviate this problem is to perform operations asynchronously, in which case the improved efficiency of operations must be able to compensate for the degradation of the convergence due to the noise introduced by asynchrony. In shared-memory environments, the "Hogwild!" [2] asynchronous SGD algorithm has been shown to be efficient for training machine learning models when the gradient updates are sparse. In the context of distributed-memory systems, approaches using a parameter server, where asynchrony mitigate the communication costs, have been employed for training large scale DNNs [3], [4]. A study of the convergence behaviour for these asynchronous models is presented in [5].

In the context of the MagmaDNN framework [6], we implemented both synchronous and asynchronous parallel minibatch SGD. The asynchronous SGD algorithm presented in this paper is designed to be efficient on shared-memory architectures even when the gradient updates are not sparse. Using the OpenMP standard and the CUDA library, training can be performed on either multicore CPUs or NVIDIA GPUs. We test our algorithms on a Multi Layer Perceptron (MLP) and a Convolutional Neural Network (CNN) to perform classification tasks. We analyze the scalability and performance of these algorithms in terms of time to convergence, and show the benefits of the asynchronous algorithm over the synchronous algorithm. As mentioned, in this work, we focus on the case in which the SGD updates are not particularly sparse.

II. Model training with SGD

We seek to minimize the objective function $J(x, y, w) = \frac{1}{m}\sum_{i=1}^{m} L(f(x^{(i)}, w), y^{(i)})$ for the parameter w over a the training set composed of m samples denoted x and corresponding labels denoted y. the function f represents our model and L corresponds to the loss function.

The SGD algorithm is an iterative procedure where the parameter, denoted w_k at the k-th iteration, is updated along a descent direction approximated at a single random sample j such that $w_{k+1} \leftarrow w_k - \alpha g_k$ where the step size α is known as the *learning rate* and $g_k = \nabla_w J(x^{(j)}, y^{(j)}, w_k)$. However using only one sample generally leads to a poor estimate of the search direction and does not make an efficient use of the compute capabilities of modern machines. Alternatively, the *batch* SGD uses all available samples in the training set to calculate an averaged search direction from the gradients for each sample. This enables a good exploitation of the computational resources but can be unnecessarily expensive when there is (near) redundancy in the data, and often leads to poor generalizability of the trained models. Instead, the minibatch SGD uses a subset of the samples, resulting in a better trade-off between computational efficiency, convergence, and generalization. Moreover, we use momentum in the model updates which means that new iterates are computed using previous search directions such as: $w_{k+1} \leftarrow w_k - \alpha z_k$ where $z_k = \beta z_{k-1} + (1 - \beta) g_k$ and β is the momentum parameter.

III. Parallel SGD methods

There are several opportunities for parallelism that can be exploited in the minibatch SGD. This includes data parallelism where the batch of samples is split between several workers to compute the gradient in parallel, and, model parallelism where the model is distributed across the workers and processed in parallel. In this study we propose a strategy based on data parallelism and discuss the opportunities for model parallelism that appear in our approach.

It should be noted that in the context of DNN training, the choice of a stopping criterion is not trivial. A common approach consists in testing the loss of the model against a validation set and stop iterating whenever the loss is no longer decreasing. This however, requires dedicating computational resources during the training to evaluate the loss which impacts the performance of the training itself. In the context of our performance and scalability study, we prefer not to perform validation tests during the training and use a time limit instead. In this case, we compute the model accuracy at the end of the procedure to evaluate the convergence of our SGD variants.

A. Synchronous SGD

Algorithm 1 Parallel SGD

1: **while** $t < time\_budget$ **do**
2: $g \leftarrow 0$
3: **for** $i = 1, N$ **do in parallel**
4: $w^{local} \leftarrow w$
5: Randomly select a batch x (with labels y)
6: Compute $g^{local} := \nabla_w J(x, y, w^{local})$
7: $AtomicUpdate(g \leftarrow g + g^{local})$
8: **end for**
9: $w \leftarrow w - \frac{\alpha}{N}\ g$
10: **end while**

In Algorithm 1 we show the pseudocode for the synchronous parallel implementation of the SGD algorithm. At each iteration, the global gradient denoted g is computed in parallel by `N` workers before updating the global model parameter, denoted w, with a SGD step using a single worker. Each worker uses a copy of the model represented by w^{local} to compute the local gradients, g^{local}, that are reduced into the global one. This parallelization strategy can impact the convergence in several ways. If the batch size is fixed across the workers, increasing the number of workers is equivalent to increasing the batch size in the serial algorithm. In this case, the speed of convergence can be affected (it may go up or it may go down). Further, it is often observed that using large batch sizes generally leads to models that generalize poorly [7]. When increasing the number of workers, the time per iteration is expected to increase due to the atomic reduction of the local gradients. Alternatively, the batch size in each worker can be reduced and for example be divided by `N` in order to reduce the time for calculating the global gradient. However, considering that the batch sizes used in practice are relatively small, this strategy might not provide much benefit as it dramatically reduces the arithmetic intensity of operations.

Our parallel implementation relies on OpenMP tasks that are created at each iteration and synchronized before the model is updated by the main thread. In the CPU implementation, the gradient computations are directly executed on a CPU core whereas in the GPU implementation, each task is associated with a CUDA stream in which the CUDA kernels are launched and are executed independently from one another.

B. Asynchronous SGD

Algorithm 2 Parallel ASGD: master worker

1: $g \leftarrow 0$
2: **while** $t < time\_budget$ **do**
3: $w \leftarrow w - \frac{\alpha}{N-1}\ g$
4: **end while**

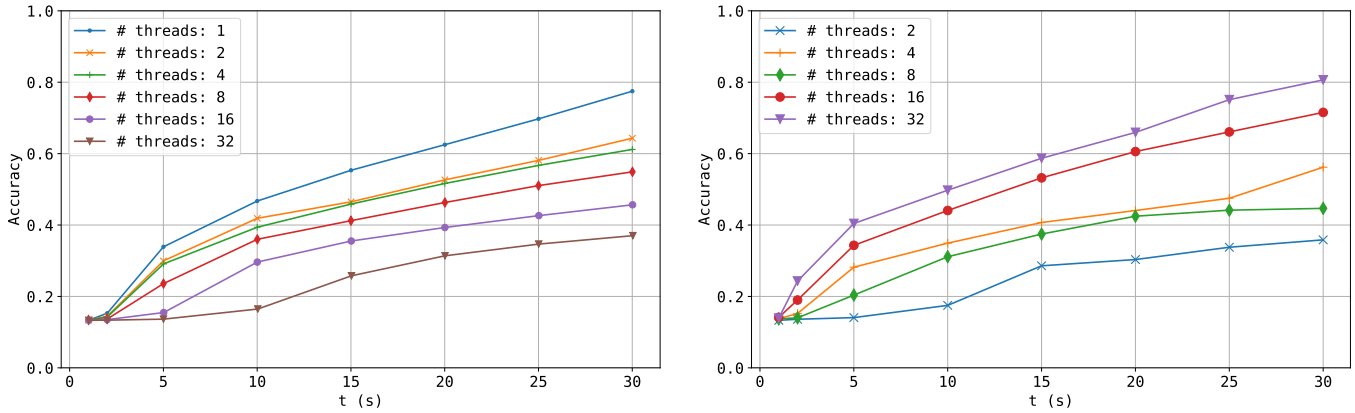
Algorithm 3 Parallel ASGD: slave workers

1: $g^{local} \leftarrow 0$
2: **while** $t < time\_budget$ **do**
3: $w^{local} \leftarrow w$
4: $AtomicUpdate(g \leftarrow g - g^{local})$
5: Randomly select a batch x and labels y
6: Compute $g^{local} := \nabla_w J(x, y, w^{local})$
7: $AtomicUpdate(g \leftarrow g + g^{local})$
8: **end while**

Our asynchronous variant of the parallel SGD uses `N` workers, where one worker, defined as the master, performs the SGD iterations as shown in Algorithm 2, while the other `N-1` workers, defined as the slave workers, follow Algorithm 3 to compute the global gradient in parallel. As shown in Figure 3 slave workers compute their local gradients using randomly selected batches from the training set and subtract their previously computed gradient to the global one before updating it with the newly computed one. Note that we chose to perform the gradient update atomically as it led to better convergence in our experiments. Apart from this reduction, other operations such that the global gradient read in the master worker, and, the local model update from the global model in the slave workers are done non-atomically. In contrast to the Hogwild! algorithm, we perform the global model update in a single worker that is the master worker. This choice is motivated be the fact that the model updates can be dense and therefore concurrent updates could harm the convergence. It should also be noted that this algorithm differs from the parameter server approach presented in [4] and [3] because in our case, first, the master does not wait on new gradient updates to iterate and, second, the updates are not done atomically allowing for slave workers to read the model during the update.

IV. Experimental results

We now show experimental results for our implementation of the parallel SGD algorithm to perform classification using



(a) Synchronous SGD

(b) Asynchronous SGD

Fig. 1. Accuracy vs training times for the parallel synchronous (a) and synchronous (b) SGD algorithm for the MLP network training on the MNIST dataset. The training is performed on a 2×20 cores CPUs for a number of threads ranging between 2 and 32.

two different datasets. First we train a MLP model, with three fully connected layers, to classify images from the MNIST dataset [8] on a multicore CPU. Second, we train a CNN model, with one convolutional, one max pooling and 2 fully connected layers, to perform image classification on the CIFAR-10 dataset [9] on a GPU. In addition, the CNN model also includes a dropout layer which helps avoid overfitting the model.

We run our experiments on an NVIDIA DGX-1 system equipped with 2×20 cores Intel Xeon CPU E5-2698 v4 clocked at 2.2 GHz and 8×V100 NVIDIA GPU devices. In the following we set the batch size to 32, independently from the number of workers and we use a learning rate equal to 10^{-3}. The momentum parameter is set to 0.99. We run the training on these DNN models for time budgets varying between 1 and 30 seconds and measure the model accuracy afterwards.

A. MLP training on Multicore CPU

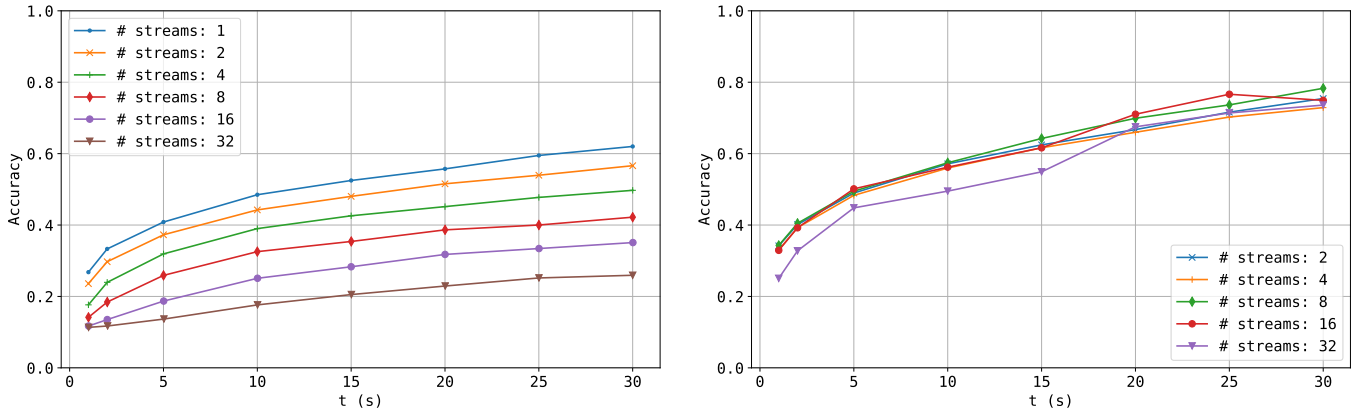
In Figure 1(a) we show the accuracy of the MLP model on the classification task for the synchronous algorithm when using a number of threads ranging between 2 and 32. We see that when training our MLP model with the synchronous algorithm, the speed of convergence is degraded when the number of threads is increased. This can be explained by the following factors: Although the overall batch size used to compute the global gradient increases with the number of threads, the iteration count does not vary much as a result of using larger batch sizes. For example, when using 2 threads, it takes 1071 iterations to reach 40% accuracy against 990 iterations when using 16 threads. However, because the scalability of the parallel gradient computation is quite inefficient, partly due to the atomic reduction, increasing the number of threads actually slows down speed of convergence.

We can see in Figure 1(b) that the speed of convergence for the asynchronous algorithm is relatively poor when using few threads and can dramatically improved by increasing the number of threads. As an example, although the algorithm only reaches slightly less than 40% accuracy after 30 seconds of training when using 2 threads, it is able to reach 80% accuracy in the same amount of time when using 32 threads. In this experiment, we observed that the number of SGD steps performed by the main thread is relatively independent from the number of workers. On another hand, the total number of global gradient updates tend to increase with the number of workers although as in the synchronous case, the gradients updates are limited by the atomic reduction. This results in an improvement in the convergence of the algorithm. We found that the maximum total number of gradient updates is obtained when using 8 threads, for which the performance is less than when using 4, 16 or 32 threads. This suggests that computing more gradients updates per SGD step does not necessarily improve the convergence. We believe that this can be caused by the increased number of conflicts while accessing the global variable concurrently.

B. CNN training on GPU

The accuracy achieved when training the CNN model on the GPU with the synchronous SGD is shown in Figure 2(a). Similar to the observation made with the MLP training, the speed of convergence of the synchronous SGD algorithm decreases when increasing the number of CUDA streams. As in the previous experiment we observed that, increasing the number of workers, which is equivalent to increasing the batch size, does not reduce the number of iteration significantly whereas the cost per iteration for computing the global gradient increases. As a result, the synchronous SGD is slowed down by increasing the number of workers. Note that it would certainly be more suitable for this synchronous algorithm to use batched kernels [10] for computing the gradients in parallel on the GPU, especially for such small granularity of operations.

As we see in Figure 2(b), the asynchronous algorithm reaches good speed of convergence when using 2 streams and outperforms the best synchronous training times obtained



(a) Synchronous SGD

(b) Asynchronous SGD

Fig. 2. Accuracy vs training times for the parallel synchronous (a) and synchronous (b) SGD algorithm for the CNN network training on the CIFAR-10 dataset. The training is performed on a NVIDIA V100 GPU for a number of CUDA streams ranging between 2 and 32.

in serial mode. However, increasing the number of workers beyond that gives no benefits as opposed to the multicore case. This can be explained by the fact that, unlike the multicore case, the number of iterations is reduced when we increase the number of workers. This is the result of the distribution of the GPU resources between the streams which is done dynamically and without control from the user.

V. Discussion

Our experiments suggest that the convergence of the asynchronous algorithm depends on the balance between the number of model updates (or SGD iterations) and the number of gradient updates. If the time for computing gradients greatly surpass the cost for performing a SGD iteration, the convergence would certainly be slowed down. This would happen, for example, when training deeper and more complex models. Additionally, exploiting model parallelism, such as updating each of the DNN layers in parallel, could also deteriorate the convergence by introducing imbalance between the number of SGD steps and gradient udpates. Finally, it has been suggested [11] that asynchrony can have a stabilization effect similar to using momentum, and can improve the convergence of SGD. This could partially explain the good convergence behavior obtained with our asynchronous algorithm.

One limitation of our approach lies in the memory footprint required for the training since we need multiple copies of the model. This can be particularly constraining for training deeper models on very large datasets, and especially on systems with limited memory like GPUs. One solution consists of distributing the dataset and possibly the model across several CPU or GPU nodes. In this case potential convergence issues can arise because internode communication, being much more costly than on a shared memory system, could drastically limit the updates to the global gradient.

VI. Conclusions and Future Work

In this work we have presented an asynchronous SGD algorithm for training DNNs which not only offers good convergence but is able to outperform the synchronous variant on multicore CPUs and GPUs. In the future we plan to implement a distributed-memory version of our algorithms.

References

[1] Y. Bengio, R. Ducharme, P. Vincent, and C. Janvin, "A neural probabilistic language model," *J. Mach. Learn. Res.*, vol. 3, pp. 1137–1155, Mar. 2003.

[2] F. Niu, B. Recht, C. Re, and S. J. Wright, "Hogwild! a lock-free approach to parallelizing stochastic gradient descent," in *Proceedings of the 24th International Conference on Neural Information Processing Systems*, ser. NIPS'11. Red Hook, NY, USA: Curran Associates Inc., 2011, p. 693–701.

[3] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, M. aurelio Ranzato, A. Senior, P. Tucker, K. Yang, Q. V. Le, and A. Y. Ng, "Large scale distributed deep networks," in *Advances in Neural Information Processing Systems 25*. Curran Associates, Inc., 2012, pp. 1223–1231.

[4] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su, "Scaling distributed machine learning with the parameter server," in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'14. USA: USENIX Association, 2014, p. 583–598.

[5] X. Lian, Y. Huang, Y. Li, and J. Liu, "Asynchronous parallel stochastic gradient for nonconvex optimization," in *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2*, ser. NIPS'15. Cambridge, MA, USA: MIT Press, 2015, p. 2737–2745.

[6] D. Nichols, N.-S. Tomov, F. Betancourt, S. Tomov, K. Wong, and J. Dongarra, "Magmadnn: Towards high-performance data analytics and machine learning for data-driven scientific computing," in *High Performance Computing*. Cham: Springer International Publishing, 2019, pp. 490–503.

[7] N. S. Keskar, D. Mudigere, J. Nocedal, M. Smelyanskiy, and P. T. P. Tang, "On large-batch training for deep learning: Generalization gap and sharp minima," *CoRR*, vol. abs/1609.04836, 2016.

[8] Y. LeCun, C. Cortes, and C. J. Burges, "The MNIST dataset of handwritten digits," http://yann.lecun.com/exdb/mnist/, 1998.

[9] A. Krizhevsky, "Learning multiple layers of features from tiny images," Tech. Rep., 2009.

[10] J. Dongarra, I. Duff, M. Gates, A. Haidar, S. Hammarling, N. J. Higham, J. Hogg, P. V. Lara, P. Luszczek, M. Zounon, S. D. Relton, S. Tomov, T. Costa, and S. Knepper, "Batched blas (basic linear algebra subprograms) 2018 specification," 2018-07 2018.

[11] I. Mitliagkas, C. Zhang, S. Hadjis, and C. Ré, "Asynchrony begets momentum, with an application to deep learning," in *54th Annual Allerton Conference on Communication, Control, and Computing, Allerton 2016, Monticello, IL, USA, September 27-30, 2016*, 2016, pp. 997–1004.