Asynchronous Optimization Methods for Efficient Training of Deep Neural Networks with Guarantees

Vyacheslav Kungurtsev,[1] **Malcolm Egan,** [2] **Bapi Chatterjee** [3] **Dan Alistarh** [3]

[1] Department of Computer Science, Faculty of Electrical Engineering Czech Technical University in Prague
[2] University of Lyon, INSA Lyon, INRIA
[3] IST Austria
vyacheslav.kungurtsev@fel.cvut.cz, malcom.egan@inria.fr, bapi.chatterjee@ist.ac.at, dan.alistarh@ist.ac.at

Abstract

Asynchronous distributed algorithms are a popular way to reduce synchronization costs in large-scale optimization, and in particular for neural network training. However, for nonsmooth and nonconvex objectives, few convergence guarantees exist beyond cases where closed-form proximal operator solutions are available. As training most popular deep neural networks corresponds to optimizing nonsmooth and nonconvex objectives, there is a pressing need for such convergence guarantees. In this paper, we analyze for the first time the convergence of stochastic asynchronous optimization for this general class of objectives. In particular, we focus on stochastic subgradient methods allowing for block variable partitioning, where the shared model is asynchronously updated by concurrent processes. To this end, we use a probabilistic model which captures key features of real asynchronous scheduling between concurrent processes. Under this model, we establish convergence with probability one to an invariant set for stochastic subgradient methods with momentum. From a practical perspective, one issue with the family of algorithms that we consider is that they are not efficiently supported by machine learning frameworks, which mostly focus on distributed data-parallel strategies. To address this, we propose a new implementation strategy for shared-memory based training of deep neural networks for a partitioned but shared model in single- and multi-GPU settings. Based on this implementation, we achieve on average about 1.2x speed-up in comparison to state-of-the-art training methods for popular image classification tasks, without compromising accuracy.

Introduction

Training deep neural networks (DNNs) is a difficult problem in several respects (Goodfellow et al. 2016). First, due to multiple layers of nonlinear activation functions, the resulting optimization problems are nonconvex. Second, ReLU activation functions and max-pooling in convolutional networks induce nonsmoothness, i.e., the objective is not differentiable everywhere. Finally, in applications it is often unreasonable to store entire data sets in memory in order to compute the objective or subgradients. As such, it is necessary to exploit stochastic methods.

Machine learning applications, including training deep neural networks, have also motivated optimization algorithms that use high performance parallel computing parallel. In this paper, we focus on the shared-memory paradigm, although our results can be efficiently extended to realistic distributed settings. Recent interest in this topic was sparked by (Recht et al. 2011), although precursors exist. Later work in (Lian et al. 2015) refined this analysis and generalised to nonconvex *smooth* problems, under restricted scheduling models. Subsequently, (Cannelli et al. 2019) introduced a more general and realistic probabilistic model of asynchronous computation on shared memory architectures.

Asynchronous proximal gradient methods have been studied in (Zhu, Niu, and Li 2018) for problems of the form $f(x) + g(x)$, where $f(x)$ is smooth and nonconvex, and $g(x)$ is nonsmooth but with an easily computable closed form proximal evaluation. This class of problems is only relevant for training DNNs wherein every activation function is smooth. However, current popular DNN models make extensive use of ReLU activation functions nor max-pooling, and thus this literature on convergence, speedup, etc. on asynchronous parallel stochastic (sub)gradient descent does not apply, when considered from the standpoint of mathematical rigor. This leaves a clear gap between practical implementation and convergence guarantees.

This gap is understandable, given that the general problem of nonsmooth and nonconvex stochastic optimisation is notoriously difficult (Bagirov, Karmitsa, and Mäkelä 2014). A standard framework to establish convergence of such algorithms in the centralized sequential setting is stochastic approximation, with early work in (Ermol'ev and Norkin 1998; Ruszczyński 1987) and comprehensive surveys in (Kushner and Yin 2003) and (Borkar 2009). In (Davis et al. 2018; Majewski, Miasojedow, and Moulines 2018), stochastic approximation for (sequential) nonsmooth and nonconvex problems has been recently developed, motivated by DNNs.

In this paper, we take a first step towards bridging the gap and establish the convergence of stochastic subgradient descent for nonsmooth and nonconvex problems in a realistic asynchronous computation framework. In particular, we show that generic asynchronous stochastic subgradient methods converge with probability one for a general class of nonsmooth and nonconvex objectives. Aside from the limited presentation in (Kushner and Yin 2003, Chapter 12), this

Algorithm	Val top-1 Accuracy	Time (Sec)	Val top-1 Accuracy	Time (Sec)
	CIFAR10		CIFAR100	
HW!	92.91±0.2	2149	68.95±0.1	2086
ASSM	92.97±0.2	2147	69.00±0.1	2088
PASSM	91.63±0.2	1192	68.32±0.2	1189
PASSM+	**92.92±0.2**	**1610**	**69.28±0.3**	**1617**
SGD	92.98	2512	68.9	2374
SGD (BS:1024)	92.23	1946	68.51	1962

Table 1: Resnet20 training for 300 epochs on a Nvidia GeForce RTX 2080 Ti, with a batch size of 128. For large-batch training, we follow (Goyal et al. 2017). Asynchronous training uses 4 concurrent processes. Standard hyperparameter values (He et al. 2016) were applied.

is the first result for this class of algorithms, combining the state of the art in stochastic approximation with that in asynchronous computation. In addition, inspired by the success of momentum methods (Zhang, Mitliagkas, and Ré 2017), we also establish for the first time convergence of stochastic subgradient descent with momentum in the context of asynchronous computation.

We complement the convergence analysis with a new efficient implementation strategy. Specifically, our main convergence result applies to an **A**synchronous **S**tochastic **S**ubgradient **M**ethod (**ASSM**), where each process updates all of the model parameters, and each partition is protected from concurrent updates by a lock. A variation is to assign *non-overlapping partitions* of the model to processes, which we call **P**artitioned ASSM (**PASSM**). This prevents overwrites, and allows us to update the model in a lock-free manner. In practice, ASSM updates the entire model, thus has an equivalent computation cost to the sequential minibatch stochastic gradient descent (**SGD**). By contrast, PASSM needs to compute block-partitioned stochastic subgradients. To implement this efficiently, we perform "restricted" backpropagation (see details in Section), which can provide savings proportional to the size of subgradients.

As shown in Table 1, PASSM is faster than both Large-Batch SGD and Hogwild! (HW!), but, in practice, it does not always recover validation (generalization) accuracy. To address this, we propose alternating sequences of PASSM and ASSM steps, denoted by **PASSM**+. From a technical perspective, the novelty of PASSM+ is to exploit concurrency to save on the cost of computation and synchronization without compromising convergence and generalization performance. A sample of the performance results for our Pytorch-based implementation is given in Table 1. PASSM+ matches the baseline in terms of generalization and yet provides on average $\sim$1.35x speed-up for identical sample processing. At the same time, it achieves $\sim$1.2x speed-up against a large-batch training method, with better generalization. The method is applicable to both shared-memory (where multiple processes can be spawned inside the same GPU up to its computational saturation) as well as in the standard multi-GPU settings.

Problem Formulation

Consider the minimization problem

$$\min_{x\in\mathbb{R}^n} f(x), \tag{1}$$

where $f:\mathbb{R}^n\to\mathbb{R}$ is locally Lipschitz continuous, but potentially nonconvex and nonsmooth. Further, we are in a setting where it is computationally infeasible to evaluate $f(x)$ or an element of the Clarke subdifferential $\partial f(x)$. In machine learning, $f(x)$ corresponds to a loss function evaluated on an n-dimensional model $x\in\mathbb{R}^n$, dependant on M input samples, each of dimension u, which can be represented as an input matrix $A\in\mathbb{R}^{M\times u}$, with target values $y\in\mathbb{R}^M$, one for each sample. That is, $f(x)=f(x;(A,y))$, where x is a parameter to optimize with respect to a loss function $\ell:\mathbb{R}\times\mathbb{R}\to\mathbb{R}$. Neural network training is then achieved by minimizing the empirical risk, where f admits the decomposition

$$f(x)=\frac{1}{M}\sum_{i=1}^{M}\ell(m(x;A_i);y_i)$$

where m generates predictions from model x and sample A_i.

We are concerned with algorithms solving (1) in a distributed fashion over shared-memory; i.e., using multiple concurrent processes. Typically, a process uses a CPU core for computation over the CPU itself using an accelerator such as a GPU. In the following, we will use the terms core and process interchangeably. We consider a standard asynchronous shared-memory setting (Recht et al. 2011), in which processes share a vector (tensor) representing the model x. In each iteration, each process reads the (possibly inconsistent) model, computes a (possibly partial) gradient with respect to it, and updates the model accordingly. This can lead to inconsistencies when a process reads the model, as other processes may be updating parts of it concurrently.

Specifically, we focus on the general *inconsistent read* scenario: before computation begins, each core $c\in\{1,\dots,\bar{c}\}$ is allocated a block of variables $I^c\subset\{1,2,\dots,n\}$, which it is responsible for updating. At each iteration, a core modifies a block of variables i^k, chosen randomly among I^c. Immediately after core c completes its iteration, it updates the model parameters over shared memory. The shared memory allows concurrent-read-concurrent-write access. Shared-memory systems offer atomic `read` and `fetch-and-add (faa)` primitives, using which processes can update model components individually.

A lock on updating the shared memory is only placed when a core writes to it, and hence the process of reading may result in computations based on model views that do not exist in memory. For instance: block 1 is read by core 1, then core 3 updates block 2 while core 1 reads block 2, and so core 1 computes an update with the values in blocks 1 and 2 that are inconsistent with the local view of core 3.

We index iterations based on when a core writes a new set of variable values into memory. Let $d_i^k=\{d_{i,1}^k,...,d_{i,n}^k\}$ be the vector of delays for each component of the variable used by core c associated with block i to evaluate a subgradient estimate, thus the j-th component of $\tilde{x}_i^k=(x_{i,1}^{k-d_{i,1}^k},\dots,x_{i,n}^{k-d_{i,n}^k})$ that is used in the computation of the

update at k may be associated with a different delay than the j'-th component.

In this paper, we study stochastic approximation methods, of which SGD is a special case. Since f in (1) is in general nonsmooth and nonconvex, we exploit generalized subgradient methods. Denote by ξ_i^k the mini-batch (i.e., a subset of the data $\{(A_i, y_i)\}_{i=1}^M$) used to compute an element of the subgradient $g_i(\tilde{x}_i^k; \xi_i^k)$. Consistent with standard empirical risk minimization methods (Bottou, Curtis, and Nocedal 2018), the mini-batch ξ_i^k is chosen uniformly at random from (A, y), independently at each iteration.

Consider the general stochastic subgradient algorithm under asynchronous updating and momentum in Algorithm 1, presented from the perspective of an individual block of variables i. Stochastic subgradient methods with momentum have been widely utilized due to their improved performance (Zhang, Mitliagkas, and Ré 2017). As such, during the k-th iteratation, updates incorporate subgradient estimates from previous iterations. The parameter $0 < m < 1$ is then the momentum constant, $\gamma^{k,i}$ the step-size and $g_i(\tilde{x}^{k,i}; \xi^{k,i})$ is an estimate of the Clarke subgradient at the point $x^{k,i}$. The variable u_i^k is the weighted sum of subgradient estimates needed to introduce momentum.

We make the following standard assumption:

Assumption 0.1. *The stochastic subgradient estimates $g(x, \xi)$ satisfy*

(i) $\mathbb{E}_\xi [g(x; \xi)] \in \partial f(x) + \beta(x)$;

(ii) $\mathbb{E}_\xi \left[dist(g(x; \xi), \partial f(x))^2\right] \leq \sigma^2$;

The term $\beta(x)$ in (i) of Assumption 0.1 is required to allow the possibility that a different element of the subgradient at x is estimated upon repeated visits to x. Given that i) the set of DNN empirical loss functions is continuously differentiable on a set of dense measure, and ii) since updates are noisy, the probability that the exact same x will be used for evaluation is zero, in practice $\beta(x)$ will always be zero. However, we make this mathematically rigorous in the convergence theory in (Kungurtsev et al. 2019).

Algorithm 1 Asynchronous Stochastic Subgradient Method

1: **Input:** x_0, block i
2: **while** Not converged **do**
3: Sample ξ_i^k from the data (A, y).
4: Read $x_i^k = \tilde{x}_i^k = (x_{i,1}^{k-d_{i,1}^k}, \ldots, x_{i,n}^{k-d_{i,n}^k})$ from the shared memory.
5: Compute the subgradient estimate $g_i(\tilde{x}_i^k; \xi_i^k)$.
6: Update the momentum vector $u_i^{k+1} \leftarrow m \cdot u_i^k + g_i(\tilde{x}_i^k; \xi_i^k)$ in and write to the local cache.
7: Write, with a lock, to the shared memory $x_i^{k+1} \leftarrow x_i^k - (1-m)\gamma_i^k u_i^{k+1}$.
8: $k \leftarrow k + 1$.
9: **end while**

Continuous-Time Reformulation

We can re-write Algorithm 1 as:

$$x_i^{k+1} = x_i^k + (1-m)\gamma_i^k \sum_{j=1}^{k} m^{k-j} Y_i^j, \tag{2}$$

where Y_i^j is based on an estimate of the partial subgradient with respect to block variables indexed by i at local iteration j. Each term Y_i^k corresponds to $g(\tilde{x}_i^k; \xi_i^k)$, denoted as

$$Y_i^k = g_i((x_{i,1}^{k-d_{i,1}^k}, \ldots, x_{i,n}^{k-d_{i,n}^k})) + \beta_i^k + \delta M_i^k,$$

where $g_i(x)$ denotes block i of a selection of an element of the subgradient of $f(x)$, β_i^k is defined in Assumption 0.1, and δM_i^k is a martingale difference sequence.

In order to translate the discrete-time updates into real-time updates, we now require an interpolation. This is a standard approach (Kushner and Yin 2003), which provides a means of establishing that the sequence of iterates converges to the flow of a differential inclusion with an equilibrium point at a stationary solution. To this end, define $\delta\tau_i^k$ to be the real elapsed time between iterations k and $k+1$ for block i and $T_i^k = \sum_{j=0}^{k-1} \delta\tau_i^j$. As such, the real time delay corresponding to $d_{i,j}^k$ is $\sum_{l=k-d_{i,j}^k}^{k-1} \delta\tau_i^l$.

The convergence of the process $\{x^k\}$ is a property of its tail behavior. Thus for $\sigma \geq 0$, let $p_i(\sigma) = \min\{j : T_i^j \geq \sigma\}$; i.e., the first iteration at or after σ. The inter-update times for block i starting at the the first update at or after σ is $\delta\tau_i^{k,\sigma}$ and the corresponding step sizes are then denoted by $\gamma_i^{k,\sigma}$.

The required notion of continuous time is then defined by

$$\tau_i^{k,\sigma} = \sum_{j=0}^{k-1} \gamma_i^{j,\sigma} \delta\tau_i^{j,\sigma}. \tag{3}$$

The shifted interpolation of $\{x_i^k\}$ is then given by

$$\hat{x}_i^\sigma(t) = x_i^{k,\sigma},\ t \in [\tau_i^{k,\sigma}, \tau_i^{k+1,\sigma}),\ \text{where} \tag{4}$$

$$x_i^{k+1,\sigma} = x_i^{k,\sigma} + (1-m)\gamma_i^{k,\sigma} \sum_{j=1}^{k+p_i(\sigma)} m^{k-j+p_i(\sigma)} Y_i^j \tag{5}$$

with $x_i^{0,\sigma} = x_i^{p_i(\sigma)}$.

We now detail the assumptions on the real elapsed times, communication delays, and subgradient estimator bias. These ensure that the real-time delays do not grow without bound, either on average, or with substantial probability. A basic for which these assumptions hold is the standard assumption that the delays are bounded, i.e., that there exists a δ such that $d_{i,j}^k \leq \delta$ for all i, j and k (and so each $d_j^{k,i} \in \mathcal{D} \triangleq \{0, ..., \delta\}^n$).

Assumption 0.2. $\{\delta\tau_i^{k,\sigma}; k, i\}$ *is uniformly integrable.*

Assumption 0.3. *There exists a function $u_{k+1,i}^\sigma$, uniformly integrable random variables $\{\Delta_{k+1,i}^{\sigma,+}\}$ and a random sequence $\{\psi_{k+1,i}^\sigma\}$ such that*

$$\mathbb{E}^{+}_{k,i}[\delta\tau_i^{k+1,\sigma}] = u^{\sigma}_{k+1,i}(\hat{x}_i^{\sigma}(\tau^{\sigma}_{k+1,i} - \Delta^{\sigma,+}_{k+1,i}), \psi^{\sigma}_{k+1,i})$$

and there is a $\bar{u}$ such that for any compact set A,

$$\lim_{m,k,\sigma} \frac{1}{m} \sum_{j=k}^{k+m-1} \mathbb{E}_{k,i}[u^{\sigma}_{j,i}(x, \psi^{\sigma}_{k+1,i}) - \bar{u}_i(x)] I_{\{\psi^{\sigma}_{k+1,i} \in A\}} = 0$$

The expectation $\mathbb{E}^{+}_{k,i}[\cdot]$ is defined with respect to the σ-algebra $\mathcal{F}^{+}_{k,i}$, while $\mathbb{E}_{k,i}$ is defined with respect to $\mathcal{F}_{k,i}$, which are detailed in the supplement. The random variables $\Delta^{\sigma,+}_{k+1,i}$ correspond (appropriately scaled) to the delays $d^k_{i,j}$.

Assumption 0.4.

$$\lim_{m,k,\sigma} \frac{1}{m} \sum_{j=k}^{k+m-1} \mathbb{E}_{k,i}[\beta^{\sigma}_{j,i}] = 0 \textit{ in mean.} \tag{6}$$

Assumption 0.5. *The sequence of iterates $\{x_i^k\}$ is tight.*

Under the other assumptions in this section, recall that Assumption 0.4 trivially holds if the estimate of the subgradient is unbiased. Assumption 0.5 is necessary since we have considered an unconstrained problem, and can be guaranteed by imposing assumptions A6.1.1' and A6.7.1 in Kushner and Yin (2003), or boundedness of the iterates.

Main Convergence Result

We now present our main convergence result. The proof is available in Section 3 of (Kungurtsev et al. 2019).

Theorem 0.1. *Suppose Assumptions 0.1, 0.2, 0.3, 0.4 and the step size conditions detailed in the supplement hold.*

Then, the following system of differential inclusions,

$$\begin{aligned} \tau_i(t) &= \int_0^t \bar{u}_i(\hat{x}(\tau_i(s)))\mathrm{d}s, \\ \dot{x}_i(t) &\in \partial_i f(\hat{x}(\tau_i(t))), \end{aligned} \tag{7}$$

$$\dot{\hat{x}}_i(t)\bar{u}_i(\hat{x}) \in \partial_i f(\hat{x}(t))$$

holds for any $\overline{u}$ satisfying 0.3. On large intervals $[0, T]$, $\hat{x}^{\sigma}(\cdot)$ spends nearly all of its time, with the fraction going to one as $T \to \infty$ and $\sigma \to \infty$, in a small neighborhood of a bounded invariant set of

$$\dot{\hat{x}}_i(t) \in \partial_i f(x(t)). \tag{8}$$

Theorem 0.1 provides conditions under which the time-shifted interpolated sequence of iterates converges weakly to an invariant set of a differential inclusion. This result can be strengthened to convergence with probability one to a block-wise stationary point via modification of the methods in (Dupuis and Kushner 1989). Details of the proof are available in Section 3.3 of (Kungurtsev et al. 2019) along with a discussion of properties of the limit point in Section 3.4.

Discussion. First, bounded invariant sets of the differential inclusion correspond to points x such that $0 \in \partial_i f(x)$. Thus, since $\hat{x}^{\sigma}$ is the interpolation of the set of iterates, the Theorem states that the iterates x_i^k end up converging to these types of stationary points with probability one.

Now consider three variations of stochastic subgradient methods discussed in the introduction: 1) standard sequential SGD with momentum, 2) PASSM, where each core updates only a block subset i of x and is lock-free, and 3) ASSM, which defines the standard parallel asynchronous implementation in which every core updates the entire vector x, taking a lock to ensure consistency.

SGD asymptotically converges to stationary points for nonconvex nonsmooth objectives, corresponds to the case of no delays, and $i = [n]$; thus, this Theorem matches the state of the art (although it also extends the theory for the sequential centralized case to the algorithm with momentum).

By simply taking the block i to be the entire vector, ASSM can be taken to a special case of PASSM, and thus Theorem 0.1 proves asymptotic convergence for both schemes. The Theorem points to their comparative advantages and disadvantages: 1) in order for indeed Theorem 0.1 to apply to ASSM, write locks are necessary, thus limiting the potential time-to-epoch speedup, however, 2) whereas if i is the entire vector the limit point of ASSM is a stationary point, i.e., a point wherein zero is in the Clarke subdifferential of the limit, in the case of PASSM, the limit point is only coordinate-wise stationary, zero is only in the i component subdifferential of f. In the smooth case, these are identical, however, this is not necessarily the case in the nonsmooth case. Thus PASSM, relative to ASSM can exhibit better speedup allowing better use of the hardware, however, may converge to a weaker notion of stationarity and thus in practice a higher value of the objective.

Implementation and Numerical Results

We describe the implementation and experimental results of the three specialized variants of the shared-memory-based distributed Algorithm 1, called ASSM, PASSM, and PASSM+. We also compare against the classic fully-asynchronous HogWild! (HW!) (Recht et al. 2011) algorithm. This algorithm can be seen as a variant of ASSM where the entire parameter vector is updated, but *without read locks*. As a result, each component of the read parameter vector could have a different delay due to potential read-write overlaps, and potentially inducing additional error.

Block Partitioned Subgradients. In backpropagation-based CNN training, the weight and bias vectors of the layers constitute the leaves of the computational graph, generated during the forward pass. During the backward pass, we can compute subgradients of the loss with respect to blocks (groups) of these parameters independently. However, computing the subgradient with respect to the farthest layer from the output (the input layer) incurs the cost of a full backpropagation. Thus, it is important to carefully partition the layers to concurrent processes.

In particular, consider a shared-memory system with c concurrent processes and a model with w parameters whose backpropagation cost is of F FLOPs. In this setting, we distribute roughly w/c parameters as in a partition to each

of the processes. Traversing from the partition that includes the input layer to the one that includes the output layer, the computational savings by the processes would be $0, \frac{F}{c}, \frac{2F}{c}, \ldots, \frac{(c-1)F}{c}$ FLOPs per backpropagation. Summing it up, we have total potential savings of $F\frac{(c-1)}{2}$ flops in c concurrent backpropagation steps by c processes. Thus, by partitioning among, say, 4 concurrent processes, on average PASSM saves roughly $\frac{3F}{8}$ FLOPs per backpropagation.

At a high level, our subgradient partitioning may seem similar to model partitioning. This is not the case: model partitioning relies on pipeline parallelism (Harlap et al. 2018), whereas we perform forward/backward passes independently and concurrently.

The Hybrid PASSM+ Method. The main advantage of PASSM is lower compute cost, as seen in Table 1. Yet, the analysis itself suggests that the quality of the stationary point to which it converges is lower relative to the other algorithms, which is evident from the same experimental results.

To mitigate this shortcoming, our idea is to spend some of the FLOPs we saved by partitioned subgradient computation on *periodic full-gradient updates*. That is, during the course of training by PASSM, perform some ASSM epochs to update the model with full subgradient computation. Essentially, this method comprises iterations by both PASSM and ASSM. We call the resulting hybrid method *PASSM+*.

At the technical level, we implemented this via a *non-blocking switching controller* (see (Kungurtsev et al. 2019)) which achieves efficient *barrier-free alternation* between concurrent ASSM and PASSM iterations. We now describe the process of hyperparameter tuning for this method.

Hyperparameter (HP) Tuning. While ASSM and HW! can use the baseline HP values, in PASSM+, along with the usual HPs such as momentum, weight-decay and learning rate (LR), we also need to tune the number of ASSM epochs, i.e. frequency of the alternation, sufficient to match the baseline accuracy. While we keep the momentum and weight decay identical to the baseline method, we tune the LR and the *frequency* and *length* of ASSM iterations in PASSM+.

One natural observation is that performing *warm-up*, that is, full subgradient updates in the initial phase of training, helps final accuracy. Further exploration showed that, along with warm-up, having periodic ASSM iterations spread across roughly 50% of all iterations is sufficient to match the baseline accuracy *without running for additional time*.

For learning rate (LR) tuning, we adopt a variant of the multi-step scheduler, which dampens the LR by a constant factor γ when a fixed portion of the sample-processing budget is consumed. For PASSM+, we explored both multi-step and cosine LR schedulers, along with a grid search (Pontes et al. 2016) to tune the LR while switching between ASSM and PASSM. The schemes we implement is the following:

1. Firstly, with a cosine scheduler, we perform roughly 10% ASSM iterations initially and during this phase warm up (Goyal et al. 2017) the LR starting from the baseline-LR up to $1.25\times$ (determined by a grid search) of the same. For the remaining iterations, along with continuously annealing the LR towards 0, we alternate between ASSM and PASSM epochs (a full pass on the training samples).
2. Alternating between ASSM and PASSM iterations engages the switching controller, which incurs additional cost. Observing that, with a multi-step scheduler, we perform roughly 25% ASSM iterations initially. After that, we switch to PASSM iterations. Further down the course of optimization, we schedule roughly 10% ASSM iterations around the LR dampening phases. Additionally, we marginally dampen the LR by a factor of $(1 - 1/c)$, where $c > 1$ is the number of concurrent processes, when switching from ASSM to PASSM. With a multi-step scheduler, continuous alternation between ASSM and PASSM iterations consistently lost accuracy.

For the basic PASSM partitioned method, no HP tuning worked to recover baseline accuracy within an unchanged sample processing budget; therefore, for PASSM we use the baseline hyperparameters.

Experimental Setup. Our implementations of image classification tasks are based on Pytorch 1.5[1] and Python multi-processing. Subgradient computation of a CNN via backpropagation is provided by the autograd module of Pytorch. Having generated a computation graph during the forward pass, we can specify the leaf tensors along which we need to generate subgradients in a call to `torch.autograd.grad()`. We use this functionality in order to implement "restricted" backpropagation in PASSM.

We implemented the methods on two settings. First, a machine – referred as **S1** in further discussion – packing a single Nvidia GeForce RTX 2080 Ti GPU and an Intel(R) Xeon(R) CPU E5-1650 v4 running @ 3.60 GHz with 6 physical cores amounting to 12 logical cores with hyper-threading. However, for bigger networks and/or datasets, the on-device memory and compute resources of a single GPU become insufficient. For those cases we used a multi-GPU machine – referred as **S2** further – with four Nvidia GeForce RTX 2080 Ti GPUs and two Intel(R) Xeon(R) CPU E5-2640 v4 @ 2.40GHz CPUs, totaling 40 logical CPU threads. The asynchronous processes use individual GPUs to compute the model and subgradients and one of the GPUs is used for shared update and synchronization. It is important to mention that CPU threads play significant role in loading data and pre-processing the images before the forward pass. Both the machines have 256 GB RAM.

NVIDIA GeForce RTX 2080 Ti GPUs allow concurrent launch of CUDA kernels by multiple processes using NVIDIA's multi-process service (MPS)[2]. Specifically, MPS allocates the SMs to concurrent CPU-GPU connections, based on their availability. We implement non-blocking data transfer between GPUs via CPU main-memory, allocating independent CPU threads for this task.

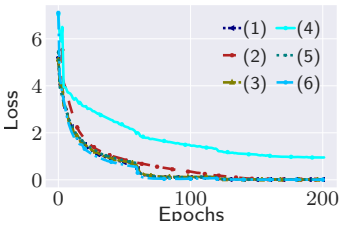
Experimental Observations and Discussion. We now present and discuss the results for well-known CNN architectures for image classification in Figures 1, 2, and 3. Each of the experimental runs is seeded, however, there is always a system-dependent randomization in asynchronous updates due to multiprocessing. Therefore, we take the average of three runs. We discuss experimental observations below.

[1] https://pytorch.org/

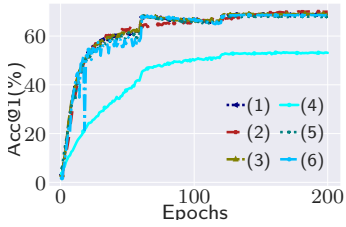
[2] https://docs.nvidia.com/deploy/mps/index.html

Algo	Train Loss	Test Loss	Train Acc.	Test Acc.	Time (S)
(1) ASSM	0.010	1.337	99.92	69.20	3733
(2) PASSM+	0.009	1.401	99.96	**70.13**	**2998**
(3) HW!	0.012	1.281	99.90	69.43	3709
(4) PASSM	0.944	1.928	72.21	53.39	2223
(5) SGD	0.012	1.288	99.92	68.48	3868
(6) SGD (BS=512)	0.006	1.393	99.99	68.76	3240

(a) Performance Summary.



(b) Train Loss.



(c) Top1 Val Accuracy.

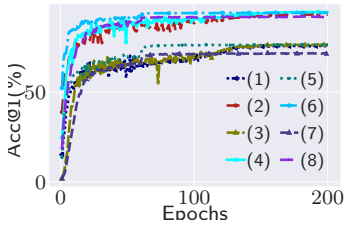
Figure 1: We train ShuffleNet (Zhang et al. 2018) with 925618 parameters arranged in 198 trainable tensors, over CIFAR100 on the single-GPU **S1**. Asynchronous methods use BS=128 and train with 4 processes. The large-batch training gets LR warm-up. PASSM+ follows the Cosine annealing LR scheme. The initial LR, weight-decay, momentum are identical across the methods. PASSM+ provides 1.1x speed-up compared to the large-batch method and 1.3x compared to the baseline, with a superior validation accuracy, by a non-trivial margin.

model	Data	Train Loss	Test Loss	Test Acc.	Time (S)
(1) DN	C100	0.002	1.063	77.02	8520
(2) DN	C10	0.000	0.325	94.15	8578
(3) RN	C100	0.002	1.101	78.69	8053
(4) RN	C10	0.000	0.288	94.97	7915

(a) Performance Summary of PASSM+.

model	Data	Train Loss	Test Loss	Test Acc.	Time (S)
(5) DN	C100	0.002	1.178	76.42	12166
(6) DN	C10	0.000	0.274	94.12	12163
(7) RN	C100	0.002	1.518	76.22	11873
(8) RN	C10	0.001	0.337	94.12	11833

(b) Performance Summary of SGD.



(c) Top1 Val Accuracy.

Figure 2: In the multi-GPU setting S2, we train two larger architectures (a) DN: DenseNet121 (Huang et al. 2017) with 6956298 parameters in 362 trainable tensors, and (b) RN: ResNext50 (Xie et al. 2016) with 14788772 parameters in 161 trainable tensors, over datasets CIFAR-10/CIFAR-100 (C10/C100). The initial LR, weight-decay, momentum are identical across methods. Here SGD is distributed over 4 GPUs via the state-of-the-art `DistributedDataParallel` framework in Pytorch. PASSM+ spawns 4 concurrent processes running over individual GPUs. Here, SGD is a large-batch implementation, which computes subgradients at BS=128 and updates the model at BS=512 on aggregation. Compared to the SOTA implementation, PASSM+ provides on average speed-up of 1.4x with improving the validation accuracy. We explain the speed-up in terms of (a) reduced flops during backpropagation, (b) reduced communication cost for partitioned subgradients across GPUs, and (c) reduced synchronization cost.

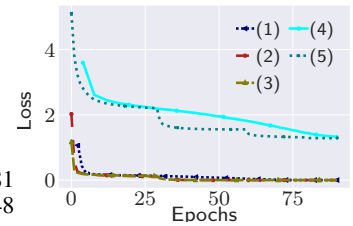
1. **Parallel Speedup.** In our knowledge, this is the first work which comprehensively reports the performance of shared-memory based training for CNNs. Please note that each of the algorithm variants, including SGD, as implemented on a GPU is inherently parallel by way of SIMD execution at the level of vector and tensor arithmetic. In this setting, increasing the minibatch size (BS) definitely helps SGD by way of reducing the traffic between CPU and GPU and better parallelization of bigger tensors. At the same time, this induces poorer generalization behavior (Goyal et al. 2017).
   If we examine the performance of HW! and ASSM, as we increase the number of processes, there is no speedup, though generalization does not change much because the BS is unchanged. This suggests that for asynchronous methods with full subgradient update there is no advantage by increasing the concurrency once the saturation of resources achieved. On the contrary, it may prove detrimental as there is a cost of synchronization involved.
2. **Advantage PASSM+.** In this context, PASSM shows

Algo	#P	BS	Sch.	Train Loss	Train Acc.	Test Loss	Test Acc.	Time (Sec)
SGD	1	128	MS	0.016	99.72	0.239	92.98	2512
SGD	1	1024	MS	0.010	99.89	0.272	92.23	1946
HW!	4	128	Cos	0.011	99.90	0.276	92.91	2145
HW!	8	128	Cos	0.011	99.85	0.266	92.93	2153
ASSM	4	128	Cos	0.010	99.87	0.255	92.97	2147
ASSM	8	128	Cos	0.011	99.86	0.271	92.85	2158
PASSM	4	128	Cos	0.065	98.07	0.288	91.63	1192
PASSM	8	128	Cos	0.183	94.68	0.314	89.93	985
PASSM+	4	128	Cos	0.017	99.70	0.260	**92.92**	**1610**
PASSM+	6	128	Cos	0.020	99.60	0.264	**92.78**	**1569**
PASSM+	8	128	Cos	0.020	99.70	0.267	**92.78**	**1549**
PASSM+	4	128	MS	0.024	99.53	0.256	**92.52**	**1598**

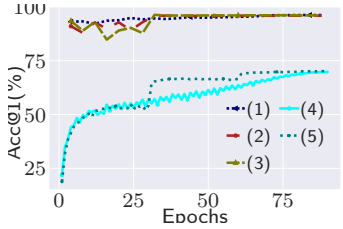
Table 2: Resnet20 with 272474 parameters in 65 trainable tensors training over CIFAR-10 for 300 epochs on the setting S1. Momentum and weight-decay are identical across the methods. Schedulers – MS: Multi-step, Cos: Cosine.

	Model/Data	Algo	BS	Train Loss	Test Loss	Test Acc.	Time (S)
(1)	res32/SVHN	PASSM+	256	0.001	0.177	96.21	1048
(2)	res32/SVHN	SGD	256	0.001	0.156	95.91	1260
(3)	res32/SVHN	SGD	1024	0.001	0.148	96.31	1388
(4)	res18/IN	PASSM+	64	1.329	1.236	69.82	146381
(5)	res18/IN	SGD	256	1.282	1.218	69.85	159048

(a) Performance Summary.



(b) Train Loss.



(c) Top1 Val Accuracy.

Figure 3: This set of results describes two contrasting cases (a) in the setting S1, ResNet32 with 466906 parameters in 101 tensors is trained over SVHN (Netzer et al. 2011) images of small cropped digits with 73257 training samples and 26032 test samples, and (b) in a machine with 4 Nvidia GeForce GTX 1080 Ti GPUs and other system specifications as those of the setting S2, ResNet18 with 11181642 parameters in 62 tensors is trained over the Imagenet (IN) dataset (Russakovsky et al. 2015) containing training set of 1.3 million images of 1000 classes and 50000 test samples. Both tasks are trained for 90 epochs. SGD follows a multi-step LR scheme and with initial LR warm-up for 5 epochs, whereas PASSM+ follows the Cosine LR rule. We observe that in case (a) PASSM+ provides up to 1.32x speed-up compared to the baseline. However, for the imagenet training task, the speed-up is around 1.08x. The reduced speed-up in case (b) can be explained in terms of high resource requirement resulting in increased contention and thereby a bottleneck over the shared GPU.

good speedup as we increase the number of processes because of the reduced FLOPs, at the cost of worse accuracy. Observing the performance of PASSM+, as concurrency grows, it shows scalability, and at the same time its optimization results remain equally competitive. It establishes that the approach of interleaving the full and partitioned SGD iterations is effective. Finally, with multi-step scheduling and reduced use of changing states of locking and lock-free updates, there is some speedup, but occasionally the generalization drops.

Related Work

HogWild! (HW!) (Recht et al. 2011) has become the classic reference for shared-memory based asynchronous SGD. However, despite significant interest in asynchronous methods, HW! or similar methods does *not* have theoretical guarantees in the general stochastic nonsmooth nonconvex setting considered here. The convergence of HW! under assumptions of Lipschitz smoothness and nonconvexity was derived in e.g. (Nadiradze et al. 2020). We note that HW! is well-known to scale on convex optimization tasks, e.g. (Recht et al. 2011), however, its performance for large scale CNN training is not very thoroughly studied.

The basic structure of HW! was extended by HogWild++ (Zhang, Hsieh, and Akella 2016), which focused on multi-CPU (multi-socket) machines with non-uniform memory access (NUMA). HogWild++ showed limited throughput advantage over HW! for convex regression problems, but do not provide any convergence analysis. Similarly, Buckwild! (Sa et al. 2015) proposed speeding up HW! by using restricted bit-precision update of the model, and has an adapted convergence analysis for convex Lipschitz smooth models, or restricted nonconvex smooth objectives.

(Sun, Hannah, and Yin 2017) analyzed partitioned asynchronous SGD for both convex and nonconvex smooth models, but did not report any implementation results. With regards to the convergence theory for nonsmooth problems in the synchronous setting, recently (Davis et al. 2018) presented asymptotic convergence with probability one, and (Zhang et al. 2020) proposed a non-asymptotic complexity analysis for computing stationary points of nonsmooth nonconvex Hadamard semi-differentiable functions, which encompass training CNNs and ReLU layers. (Shamir 2020) explored the hardness of finding near-approximate-stationary points of the same class of functions.

Conclusion

We proved the first asymptotic convergence results (with probability 1) for asynchronous parallel stochastic subgradient descent methods with momentum, for general nonconvex nonsmooth objectives. This is the first such characterization covering DNNs, finally matching theory to common practice, closing an important gap in the literature. We furthermore presented an in-depth analysis of the performance of such schemes in a shared-memory setting, coupled with an efficient implementation strategy for training deep CNNs. We showed that a variant of our method can be efficiently implemented on GPUs, demonstrating speed-up versus state-of-the-art methods, without losing generalization accuracy. The experimental results provided a thorough exploration of both the potential and limitations of speedup for a comprehensive set of variants of shared memory asynchronous multi-processing.

Future work could include a look at iteration complexity from the standpoint of (Zhang et al. 2020). Numerically, the results indicate that whereas some speedup is achievable by a careful procedure, significant gains for training DNNs would require an entirely novel approach.

Acknowledgments

Vyacheslav Kungurtsev was supported by the OP VVV project CZ.02.1.01/0.0/0.0/16_019/0000765 "Research Center for Informatics." Bapi Chatterjee was supported by the European Union's Horizon 2020 research and innovation

programme under the Marie Sklodowska-Curie grant agreement No. 754411 (ISTPlus). Dan Alistarh has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 805223 ScaleML).

References

Bagirov, A.; Karmitsa, N.; and Mäkelä, M. M. 2014. *Introduction to Nonsmooth Optimization: theory, practice and software*. Springer.

Borkar, V. S. 2009. *Stochastic approximation: a dynamical systems viewpoint*. Springer.

Bottou, L.; Curtis, F.; and Nocedal, J. 2018. Optimization methods for large-scale machine learning. *SIAM Review* 60(2): 223–311.

Cannelli, L.; Facchinei, F.; Kungurtsev, V.; and Scutari, G. 2019. Asynchronous parallel algorithms for nonconvex optimization. *Mathematical Programming* 1–34.

Davis, D.; Drusvyatskiy, D.; Kakade, S.; and Lee, J. D. 2018. Stochastic subgradient method converges on tame functions. *arXiv preprint arXiv:1804.07795* .

Dupuis, P.; and Kushner, H. J. 1989. Stochastic approximation and large deviations: Upper bounds and wp 1 convergence. *SIAM Journal on Control and Optimization* 27(5): 1108–1135.

Ermol'ev, Y. M.; and Norkin, V. 1998. Stochastic generalized gradient method for nonconvex nonsmooth stochastic optimization. *Cybernetics and Systems Analysis* 34(2): 196–215.

Goodfellow, I.; Bengio, Y.; Courville, A.; and Bengio, Y. 2016. *Deep learning*, volume 1. MIT press Cambridge.

Goyal, P.; Dollár, P.; Girshick, R.; Noordhuis, P.; Wesolowski, L.; Kyrola, A.; Tulloch, A.; Jia, Y.; and He, K. 2017. Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677* .

Harlap, A.; Narayanan, D.; Phanishayee, A.; Seshadri, V.; Devanur, N. R.; Ganger, G. R.; and Gibbons, P. B. 2018. PipeDream: Fast and Efficient Pipeline Parallel DNN Training. *CoRR* abs/1806.03377.

He, K.; Zhang, X.; Ren, S.; and Sun, J. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 770–778.

Huang, G.; Liu, Z.; van der Maaten, L.; and Weinberger, K. Q. 2017. Densely Connected Convolutional Networks. In *CVPR*, 2261–2269.

Kungurtsev, V.; Egan, M.; Chatterjee, B.; and Alistarh, D. 2019. Asynchronous Optimization Methods for Efficient Training of Deep Neural Networks with Guarantees. *arXiv preprint arXiv:1905.11845* .

Kushner, H.; and Yin, G. G. 2003. *Stochastic approximation and recursive algorithms and applications*, volume 35. Springer Science & Business Media.

Lian, X.; Huang, Y.; Li, Y.; and Liu, J. 2015. Asynchronous parallel stochastic gradient for nonconvex optimization. In *NIPS*, 2737–2745.

Majewski, S.; Miasojedow, B.; and Moulines, E. 2018. Analysis of nonsmooth stochastic approximation: the differential inclusion approach. *arXiv preprint arXiv:1805.01916* .

Nadiradze, G.; Markov, I.; Chatterjee, B.; Kungurtsev, V.; and Alistarh, D. 2020. Elastic Consistency: A General Consistency Model for Distributed Stochastic Gradient Descent. *ArXiv* abs/2001.05918.

Netzer, Y.; Wang, T.; Coates, A.; Bissacco, A.; Wu, B.; and Ng, A. 2011. Reading Digits in Natural Images with Unsupervised Feature Learning.

Pontes, F. J.; da F. de Amorim, G.; Balestrassi, P.; Paiva, A. P.; and Ferreira, J. R. 2016. Design of experiments and focused grid search for neural network parameter optimization. *Neurocomputing* 186: 22–34.

Recht, B.; Re, C.; Wright, S.; and Niu, F. 2011. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *NIPS*, 693–701.

Russakovsky, O.; Deng, J.; Su, H.; Krause, J.; Satheesh, S.; Ma, S.; Huang, Z.; Karpathy, A.; Khosla, A.; Bernstein, M.; Berg, A.; and Fei-Fei, L. 2015. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision* 115: 211–252.

Ruszczyński, A. 1987. A linearization method for nonsmooth stochastic programming problems. *Mathematics of Operations Research* 12(1): 32–49.

Sa, C. D.; Zhang, C.; Olukotun, K.; and Ré, C. 2015. Taming the Wild: A Unified Analysis of Hogwild-Style Algorithms. *NIPS* 28: 2656–2664.

Shamir, O. 2020. Can We Find Near-Approximately-Stationary Points of Nonsmooth Nonconvex Functions? *ArXiv* abs/2002.11962.

Sun, T.; Hannah, R.; and Yin, W. 2017. Asynchronous Coordinate Descent under More Realistic Assumptions. In *NIPS*.

Xie, S.; Girshick, R.; Dollár, P.; Tu, Z.; and He, K. 2016. Aggregated Residual Transformations for Deep Neural Networks. *arXiv preprint arXiv:1611.05431* .

Zhang, H.; Hsieh, C.-J.; and Akella, V. 2016. HogWild++: A New Mechanism for Decentralized Asynchronous Stochastic Gradient Descent. *ICDM* 629–638.

Zhang, J.; Lin, H.; Sra, S.; and Jadbabaie, A. 2020. On Complexity of Finding Stationary Points of Nonsmooth Nonconvex Functions. *ArXiv* abs/2002.04130.

Zhang, J.; Mitliagkas, I.; and Ré, C. 2017. YellowFin and the Art of Momentum Tuning. *CoRR* abs/1706.03471.

Zhang, X.; Zhou, X.; Lin, M.; and Sun, J. 2018. ShuffleNet: An Extremely Efficient Convolutional Neural Network for Mobile Devices. In *CVPR*, 6848–6856.

Zhu, R.; Niu, D.; and Li, Z. 2018. Asynchronous Stochastic Proximal Methods for Nonconvex Nonsmooth Optimization. *arXiv preprint arXiv:1802.08880* .