LOCAL SGD MEETS ASYNCHRONY

ABSTRACT

Distributed variants of stochastic gradient descent (SGD) are central to training deep neural networks on massive datasets. Several scalable versions of data-parallel SGD have been developed, leveraging asynchrony, communication-compression, and local gradient steps. Current research seeks a balance between distributed scalability–seeking to minimize the amount of synchronization needed–and generalization performance–seeking to achieve the same or better accuracy relative to the sequential baseline. However, a key issue in this regime is largely unaddressed: if "local" data-parallelism is aggressively applied to better utilize the computing resources available with workers, generalization performance of the trained model degrades.

In this paper, we present a method to improve the "local scalability" of decentralized SGD. In particular, we propose two key techniques: (a) shared-memory based asynchronous gradient updates at decentralized workers keeping the local minibatch size small, and (b) an asynchronous non-blocking in-place averaging overlapping the local updates, thus essentially utilizing all compute resources at all times without the need for large minibatches. Empirically, the additional noise introduced in the procedure proves to be a boon for better generalization. On the theoretical side, we show that this method guarantees ergodic convergence for non-convex objectives, and achieves the classic sublinear rate under standard assumptions. On the practical side, we show that it improves upon the performance of local SGD and related schemes, without compromising accuracy.

1 INTRODUCTION

In this paper, we consider the classic problem of minimizing an empirical risk, defined simply as

$$\min_{x \in \mathbb{R}^d} \sum_{i \in [I]} f_i(x),$$

where d is the dimension, $x \in \mathbb{R}^d$ denotes the set of model parameters, $[I]$ is the set of samples, and $f_i(x) : \mathbb{R}^d \to \mathbb{R}$ is the loss on the training sample $i \in [I]$. Stochastic gradient descent (SGD) (Robbins & Monro, 1951) is an extremely popular approach to solving this problem, described by the iteration $x_{k+1} = x_k - \alpha_k \nabla f_{B_k}(x_k)$, where $\nabla f_{B_k}(x_k) = \frac{1}{|B_k|} \sum_{i \in B_k} \nabla f_i(x_k)$ is the sum of gradients computed over samples, typically selected uniformly and randomly, as a minibatch $B_k \subseteq [I]$, and α_k is the learning rate at iteration k.

1.1 BACKGROUND ON DECENTRALIZED DATA-PARALLEL SGD

For better or worse, SGD and its variants currently represent the computational backbone for many large-scale optimization tasks, most notably the training of deep neural networks (DNNs). Arguably the most popular SGD variant is *minibatch SGD* (MB-SGD) (Bottou (2012)). In a distributed setting with decentralized workers $q \in [Q]$, it follows the iteration

$$x_{k+1} = x_k - \alpha_k \frac{1}{Q} \sum_{q=1}^{Q} \nabla f_{B_k^q},$$

where $B_k^q \subseteq [I]$ is a local minibatch selected by worker $q \in [Q]$ at iteration k. This strategy is straightforward to scale in a data-parallel way, as each worker can process a subset of the samples in parallel, and the model is then updated by the average of the workers' gradient computations. For convenience, we assume the same batch size per worker. This approach has achieved tremendous popularity recently, and there has been significant interest in running training with increasingly large batch sizes aggregated over a large number of GPUs, e.g. Goyal et al. (2017). An alternative approach is *parallel or local SGD* (L-SGD) (Zinkevich et al. (2010); Zhang et al. (2016c); Lin et al. (2020)):

$$x_{j,t+1}^q = x_{j,t}^q - \alpha_{j,t} \nabla f_{B_{j,t}^q},\ 0 \le t < K_j;\ x_{j+1,0}^q = \frac{1}{Q} \sum_q x_{j,K_j}^q,$$

where $x^q_{j,t}$ denotes the local model at worker $q \in [Q]$ after j synchronization rounds followed by t local gradient updates and $B^q_{j,t}$ is the local minibatch sampled at the same iteration. K_j denotes the number of local gradient update steps before the jth synchronization. Essentially, workers run SGD without any communication for several local steps, after which they globally *average* the resulting local models. This method is intuitively easy to scale, since it reduces the *frequency* of the communication. Recently, a variant called *post local SGD* (PL-SGD) (Lin et al. (2020)), was introduced to address the issue of loss in generalization performance of L-SGD, wherein the averaging frequency during the initial phase of training is high and is reduced later when optimization stabilizes.

Method	B_{loc}	Train Loss	Train Acc.	Test Loss	Test Acc	Time (Sec)	Quality/ Perf.
MB-SGD	128	0.016	99.75	0.234	92.95	1754	Baseline
MB-SGD	1024	0.023	99.51	0.293	91.38	1201	OK
PL-SGD	128	0.018	99.69	0.245	92.98	1603	Good
PL-SGD	1024	0.154	94.69	0.381	87.81	1159	Poor

Table 1: RESNET-20/CIFAR-10 training for 300 epochs on 2 GPUs. Throughout the training the local BS is kept constant across workers i.e. $|B^q_k| = B_{loc}, \forall q \in [Q], k \geq 0$ for MB-SGD and $|B^q_{j,t}| = B_{loc}, \forall q \in [Q], j, t \geq 0$ for PL-SGD. The LR is warmed up for the first 5 epochs to scale from $\underline{\alpha_0}$ to $\underline{\alpha_0} \times \frac{B_{loc} \times Q}{\underline{B_{loc}}}$, where $\underline{B_{loc}} = 128$, Q is the number of workers, 2 here, and $\underline{\alpha_0} = 0.1$. In PL-SGD, we average the model after each gradient update for first 150 epochs and thereafter averaging frequency K is set to 16 as in Lin et al. (2020); other HPs are identical to theirs. The listed results are average of 3 runs with different seeds.

Although very popular in practice, these two approaches suffer from the same limitation: their generalization accuracy decreases for larger local batch sizes, as would be appropriate to fully utilize the computation power offered by the workers. We illustrate this in Table 1: examine the results of training RESNET-20 (He et al. (2016)) over CIFAR-10 (Krizhevsky (2009)) with MB-SGD and PL-SGD over a workstation packing two Nvidia GeForce RTX 2080 Ti GPUs (a current standard). We observe that as the local batch size B_{loc} grows, the throughput improves significantly, however, the optimization results degrade sharply, more glaringly with PL-SGD. Clearly, these methods can not tolerate a larger B_{loc}, though the GPUs can support them. This shortcoming of the existing methods in harnessing the growing data-parallelism is also identified via empirical studies (Golmant et al. (2018); Shallue et al. (2019)) existing in literature. To our knowledge no effective remedy (yet) exists to address this challenge.

1.2 LOCALLY-ASYNCHRONOUS PARALLEL SGD

Now, consider an implementation scheme as given in Algorithm 1. The model x^q on a GPU $q \in [Q]$ is shared by the processes $p \in P^q = \{\{a^q\} \cup U^q\}$ locally. The processes $p \in P^q$ also maintain a shared counter $\mathcal{S}^q$, initialized to 0. The operation `read-and-inc` implements an atomic (with lock) read and increment of $\mathcal{S}^q$, whereas, `read` provides an atomic read. $\mathcal{S}^q$ essentially enables ordering the shared gradient updates. In turn, this order streamlines the synchronization among workers, essentially in determining the averaging rounds j. The processes $u \in U^q$ asynchronously and lock-freely update x^q with gradients computed over a non-blocking, potentially inconsistent, snapshot $v^{a,q}$ of x^q, essentially going Hogwild! (Recht et al. (2011)), see Algorithm 1a.

Initialize $s = 0$;
while $s \leq T$ **do**
  $v^{u,q}[i] := x^q[i], \forall\, 1 \leq i \leq d$;
  $s :=$ `read-and-inc`$(\mathcal{S})$;
  Compute $\nabla f_{B^q_s}(v^{u,q})$;
  $x^q[i] \;-=\; \alpha_s \nabla f_{B^q_s}(v^{u,q})[i], \;\forall\, 1 \leq i \leq d$;

(a) Local asynchronous gradient update by process $u \in U^q$.

Initialize $s_{cur} = s_{pre} = |U^q|, j = 0$;
while $s_{cur} \leq T$ **do**
  $s_{cur} :=$ `read`$(\mathcal{S})$; **if** $s_{cur} - s_{pre} \geq K_j$ **then**
    $v^{a,q}_j[i] := x^q[i], \forall\, 1 \leq i \leq d$; $s^q_j := s_{cur} - |U^q|$;
  if $v^{a,q}_j$ *is ready for all* q **then**
    Communicate across $a^r, r \in [Q] \setminus \{q\}$ to compute $\overline{v_j} := \frac{1}{Q} \sum_{q \in [Q]} v^{a,q}_j$;
    Compute $\Delta v^q_j = \overline{v_j} - v^{a,q}_j$; $s_{pre} := s_{cur}$;
    $x^q[i] \;+=\; \Delta v^q_j[i], \forall\, 1 \leq i \leq d$; $j = j + 1$

(b) Asynchronous non-blocking in-place averaging.

Algorithm 1: Locally-asynchronous Parallel SGD (LAP-SGD)

The process a^q concurrently keeps on atomically reading $\mathcal{S}^q$, see Algorithm 1b. As soon as it notices an increment K_j in $\mathcal{S}^q$, i.e. x^q got concurrently updated with K_j number of gradients , it takes a non-blocking snapshot $v^{a,q}_j$ of x^q and synchronizes with a^r of peers $r \in [Q]/q$ to compute the average $\overline{v_j}$ of the snapshots. Thereafter, a^q adds the difference of the average with the snapshot $v^{a,q}_j$ to the model x^q without blocking the concurrent asynchronous local gradient updates. We call this method *locally-asynchronous parallel SGD* (LAP-SGD). This method closely resembles Hogwild++ (Zhang et al. (2016a)), which targets the heterogeneous NUMA based multi-core machines, though

there are key differences which we describe in Section 4. Results of the same training task as before by LAP-SGD is available in Table 2. The distinction of this implementation is that it harnesses the compute power of the GPUs not by increasing the size of B_{loc} but by concurrently computing many minibatch gradients. Evidently, LAP-SGD provides speed-up without losing the quality of optimization in comparison to the baseline methods.

Recently, Kungurtsev et al. (2019) presented a shared-memory based method wherein they showed that partitioned gradient updates for some iterations during the course of training over a shared model can reasonably save on total computation cost by means of restricted backpropagation without necessarily losing on optimization quality. Their method is limited to a centralized shared-memory setting. Moreover, aiming to establish convergence under non-smoothness assumption, they ensure write consistency under a model-wide lock. Having designed our asynchronous parallel SGD, it inspires us to adapt the partitioned gradient update strategy to our lock-free decentralized setting.

$\|U^q\|$	B_{loc}	Train Loss	Train Acc.	Test Loss	Test Acc	Time (Sec)	Quality/ Perf.
4	128	0.012	99.83	0.270	92.92	1304	Excellent
6	128	0.023	99.51	0.266	92.91	1281	Excellent
4	256	0.010	99.90	0.280	92.93	1219	Excellent

Table 2: RESNET-20/CIFAR-10 training for 300 epochs on 2 GPUs by LAP-SGD. The LR is warmed up similar to PL-SGD.We follow the same averaging schedule and other HPs as those of PL-SGD. The listed results are average of 3 runs.

More specifically, building on LAP-SGD, we consider locally partitioned gradient computation along with asynchronous lock-free updates. Essentially, we partition the model x^q to $\{x^q_{i(u)}\}$ for $u \in U^q$, $i(u) \cap i(w) = \emptyset, \forall u, w \in U^q$ (i.e., non-overlapping block components of the vector x). With that, a partitioned gradient computation will amount to computing $\nabla_{i(u)} f_{B^q_s}(v^{q,u})$, the minibatch gradient with respect to the partition $x^q_{i(u)}$ at line 5 in Figure 1a. Accordingly, the update step at line 6 in Figure 1a transforms to $x^q[i] \ -= \ \alpha_s \nabla f_{B^q_s}(v^{q,u})[i], \ \forall \ i \in i(u)$. It is to be noted that we do not use write lock for iterations at any stage. Having devised a partitioned update scheme, we propose *locally-partitioned-asynchronous parallel SGD* (LPP-SGD) as described below.

$\|U^q\|$	B_{loc}	Train Loss	Train Acc.	Test Loss	Test Acc	Time (Sec)	Quality/ Perf.
4	128	0.019	99.62	0.270	92.98	1153	Excellent
6	128	0.021	99.58	0.262	92.84	1085	Excellent
4	256	0.019	99.65	0.267	92.95	1047	Excellent

Table 3: LPP-SGD performance. Other details are identical to those in Table 2.

1. Processes $u \in U^q$ maintain a process-local variable `last_iter` which can take two values `PARTITIONED` and `FULL`. Each $u \in U^q$ initializes `last_iter` as `FULL`.
2. **While** $s \leq T_{st}$, each process $u \in U^q$ performs LAP-SGD updates as in lines 3 to 6 of Figure 1a.
3. **If** $T_{st} < s \leq T$, each process $u \in U^q$ performs
   (a) a partitioned gradient computation and update: $x^{q,u}[i] \ -= \ \alpha_s \nabla f_{B^q_s}(v^{u,q})[i], \ \forall i \in i(u)$ **if** `last_iter` = `FULL`, and sets `last_iter` = `PARTITIONED`.
   (b) an LAP-SGD update **if** `last_iter` = `PARTITIONED`, and sets `last_iter` = `FULL`.

Essentially, after some initial *stabilizing* epochs each process $u \in U^q$ alternates between a full and a partitioned lock-free asynchronous gradient updates to the model x^q. Our experiments showed that $T_{st} = \frac{T}{10}$ was almost always sufficient to obtain a competitive optimization result. The results of a sample implementation of LPP-SGD are available in Table 3. It is clear that LPP-SGD handsomely speeds up the computation and provides equally competitive optimization results.

2 CONVERGENCE THEORY

1 Initialize $\bar{x}_0 = x^q_0$ for all q;
2 **for** $j = 1, ..., J$ **do**
3 **for** *all* q **do**
4 Set $x^{q,j}_0 = \bar{x}_j$;
5 **for** $t = 1, ..., K_j$ **do**
6 Let $x^{q,j}_t = x^{q,j}_{t-1} - \alpha_{j,t,q} \tilde{\nabla}_{(i(j,t,q)} f(v^{q,j}_t)$;
7 Let $\bar{x}_{j+1} = \frac{1}{Q} \sum_{q=1}^{Q} x^{q,j}_{K_j}$;

Algorithm 2: Iterations of the view $\bar{x}_j$.

At a naive first glance, studying the convergence properties of *locally asynchronous SGD* would be an incremental to existing analyses for local SGD, e.g. Stich (2018); Zhou & Cong (2017), in particular the local stochastic gradient evaluations at delayed lock-free Hogwild!-like parameter vectors. However, there is one significant difference that presents a theoretical challenge: sometimes the vectors used for gradient computation or components thereof, have been read from the local shared memory before the last averaging across GPUs had taken place. Especially in the non-convex case, *a priori* it is impossible to place a reasonable bound on the accuracy of these gradient evaluations relative to what they "should be" in order to achieve descent.

In order to present a convergence rate result, we need to define an anchor point on which we can consider convergence to some approximate stationary point in expectation. This is not trivial as both local iterations and averaging is performed asynchronously across different GPUs at distinct moments in time, with the time at which each iteration occurs potentially varying with system-dependent conditions, while the averaged iterates are what is important to consider for convergence.

We seek to define the *major iteration* $\bar{x}_j$ as consistent with the analysis presented for the convergence of local SGD in the nonconvex setting in Zhou & Cong (2017). In this case, with asynchrony, $\bar{x}_j$ is a theoretical construct, i.e., it may not exist at any particular GPU's memory at any moment in time. It is defined as $x^q_{s^q_j} + \Delta v_j$ where $x^q_{s^q_j}$ is the state of the model for GPU q when $v^{a,q}_j$ was saved and made available for averaging for "major iteration" j. Thus although de facto averaging could have taken place after a set of local updates in time, these local updates correspond to updates after iteration j conceptually. This makes $\bar{x}_j$ properly correspond to the equivalent iterate in Zhou & Cong (2017).

With that, we consider a sequence of *major iteration views* $\{\bar{x}_j\}$ with associated inter-averaging local iteration quantities K_j and *local views* $\{v^{q,j}_t\}$ at which an unbiased estimate of the (possibly partitioned) gradient is computed, with $0 \leq t < K_j$ as well as the local model in memory $\{x^{q,j}_t\}$. The partition, which could be the entire vector in general, is denoted by $i(j,t,q)$. As each GPU has its own corresponding annealed stepsize, we denote it in general as $\alpha_{j,t,q}$ as well.

We state the formal mathematical algorithm as Algorithm 2. Note that this is the same procedure as the practical "what the processor actually does" Algorithm 1, however, with the redefined terms in order to obtain a precise mathematical sequence well-defined for analysis.

We make the following standard assumptions on unbiased bounded variance for the stochastic gradient, a bounded second moment, gradient Lipschitz continuity, and lower bonudedness of f.

Assumption 2.1. *1. It holds that $\tilde{\nabla}_i f(v^{q,j}_t)$ satisfies, independent of i,*

$$\mathbb{E}\left[\tilde{\nabla}_i f(v^{q,j}_t)\right] = \nabla_i f(v^{q,j}_t);\ \mathbb{E}\left[\left\|\tilde{\nabla}_i f(v^{q,j}_t) - \nabla_i f(v^{q,j}_t)\right\|^2\right] \leq \sigma^2;\ \mathbb{E}\left[\left\|\tilde{\nabla}_i f(v^{q,j}_t)\right\|^2\right] \leq G.$$

2. f is Lipschitz continuously differentiable with constant L and is bounded below by f_m.

We must also define some assumptions on the probabilistic model governing the asynchronous computation. As these are fairly technical we defer them to the appendix.

Theorem 2.1. *Given assumption 2.1, it holds that,*

$$\begin{aligned}\tfrac{1}{Q}\textstyle\sum_{j=1}^{J}\sum_{q=1}^{Q}\sum_{t=0}^{K_j-1}[\alpha_{j,t,q}C_1 - \alpha^2_{j,t,q}C_2]\mathbb{E}\left\|\nabla_{i(j,t,q)} f(v^{q,j}_t)\right\|^2 \\ +\tfrac{1}{Q}\textstyle\sum_{j=1}^{J}\sum_{q=1}^{Q}\sum_{t=0}^{K_j-1}[\alpha_{j,t,q}C_3 - \alpha^2_{j,t,q}C_4]\mathbb{E}\left\|\nabla_{i(j,t,q)} f(\bar{x}_j)\right\|^2 \\ \leq f(\bar{x}_0) - f_m\end{aligned}$$

where C_1, C_2, C_3, and C_4 depend on L, B and probabilistic quantities defining the asynchronous computation (see Appendix. Thus there exists a set of such constants such that if $\alpha_{j,t,q} = \Theta\left(\frac{1}{\sqrt{J}}\right)$ then Algorithm 2 ergodically converges with the standard $O(1/\sqrt{J})$ rate for nonconvex objectives.

Proof Summary: The proof follows the structure of the ergodic convergence proof of K-step local SGD given in Zhou & Cong (2017), wherein at each round of averaging there are QK_j total updates to the model vector associated with the Q GPUs and K_j minor iterations.

Insofar as these updates are close (stochastically as an unbiased estimate, and based on the local models not having changed too much) to the globally synchronized model vector at the last averaging step, there is an expected amount of descent achieved due to the sum of these steps. This is balanced with the amount of possible error in this estimate based on how far the model vector had moved.

In cases wherein $v^{q,j}_{t,i} = x^{q,j}_{s,i}$ for $s < 0$ (i.e., the stochastic gradients are taken, due to local asynchrony, at model vectors with components which existed in memory before the last averaging step), we simply bound the worst-case increase in the objective.

To balance these two cases, the analysis takes an approach, inspired partially by the analysis given in Cartis & Scheinberg (2018) of separating these as "good" and "bad" iterates, with "good" iterates corresponding to views read after the last model was stored for averaging, with some associated guaranteed descent in expectation, and "bad" iterates those read beforehand.

By considering the stochastic process governing the amount of asynchrony as being governed by probabilistic laws, we can characterize the probability of a "good" and "bad" iterate and ultimately

seek to balance the total expected descent from one, and worst possible ascent in the other, as a function of these probabilities.

3 IMPLEMENTATION DETAILS AND NUMERICAL RESULTS

Decentralized training. We evaluate the proposed methods LAP-SGD and LPP-SGD comparing them against existing MB-SGD and PL-SGD schemes, using CNN models RESNET-20 (He et al. (2016)), SQUEEZENET (Iandola et al. (2017)), and WIDERESNET-16x8 (Zagoruyko & Komodakis (2016)) for the 10-/100-class image classification tasks on datasets CIFAR-10/CIFAR-100 (Krizhevsky (2009)). We also train RESNET-18 for a 1000-class classification problem on IMAGENET (Russakovsky et al. (2015)) dataset. We keep the sample processing budget identical across the methods. We use the typical approach of partitioning the sample indices among the workers that can access the entire training set; the partition indices are reshuffled every epoch following a seeded random permutation based on epoch-order. To this effect we use a shared-counter among concurrent process $u \in U^q$ in asynchronous methods. Thus, our data sampling is I.I.D.

Platform specification. Our experiments are based on a set of Nvidia GeForce RTX 2080 Ti GPUs (Nvidia (2020)) (referred to as GPUs henceforth) with 11 GB on-device memory. We use the following specific settings: (a) **S1:** a workstation with two GPUs and an Intel(R) Xeon(R) E5-1650 v4 CPU running @ 3.60 GHz with 12 logical cores, (b) **S2:** a workstation with four GPUs and two Intel(R) Xeon(R) E5-2640 v4 CPUs running @ 2.40 GHz totaling 40 logical cores, and (c) **S3:** two **S2** workstations connected with a 100 GB/s infiniband link. The key enabler of our implementation methodology are multiple independent client connection between a CPU and a GPU. Starting from early 2018 with release of Volta architecture, Nvidia's technology Multi-process Service (MPS) efficiently support this. For more technical specifications please refer to their doc-pages MPS (2020).

Implementation framework. We used open-source Pytorch 1.5 (Paszke et al. (2017)) library for our implementations. For cross-GPU/cross-machine communication we use NCCL (NCCL (2020)) primitives provided by Pytorch. MB-SGD is based on `DistributedDataParallel` Pytorch module. PL-SGD implementation is derived from author's code (LocalSGD (2020)) and adapted to our setting. Having generated the computation graph of the loss function of a CNN, the `autograd` package of Pytorch allows a user to specify the leaf tensors with respect to which gradients are needed. We used this functionality in implementing partitioned gradient computation in LPP-SGD.

Locally-asynchronous Implementation. One key requirement of the proposed methods is to support a non-blocking synchronization among GPUs. This is both a challenge and an opportunity. To specify, we use a process on each GPU, working as a parent, to initialize the CNN model and share it among spawned child-processes. The child-processes work as $u \in U^q$, $\forall q \in [Q]$ to compute the gradients and update the model. Concurrently, the parent-process, instead of remaining idle as it happens commonly with such concurrency models, acts as the averaging process $a^q \in P^q$, $\forall q \in [Q]$, thereby productively utilizing the entire address space occupied over the GPUs. The parent- and child-processes share the iteration and epoch counters. Notice that, here we are using the process-level concurrency which is out of the purview of the thread-level global interpreter lock (GIL) (Python (2020)) of python multi-threading framework.

Hyperparameters (HPs). Each of the methods use identical momentum (Sutskever et al. (2013)) and weight-decay (Krogh & Hertz (1991)) for a given CNN/dataset case; we rely on their previously used values (Lin et al. (2020)). The learning rate (LR) schedule for MB-SGD and PL-SGD are identical to Lin et al. (2020). For the proposed methods we used cosine annealing schedule without any intermediate restart (Loshchilov & Hutter (2017)). Following the well-accepted practice, we *warm up* the LR for the first 5 epochs starting from the baseline value used over a single worker training. In some cases, a grid search (Pontes et al. (2016)) suggested that for LPP-SGD warming up the LR up to 1.25× of the *warmed-up* LR of LAP-SGD for the given case, improves the results.

In the following discussion we use these abbreviated notations: $\mathbf{U} : |U^q|$, $\mathbf{B} : B_{loc}$, Tr.L.: Training Loss, Tr.A.: Training Accuracy, Te.L.: Test Loss, Te.A.: Test Accuracy, and T: Time in Seconds. The asynchronous methods have inherent randomization due to process scheduling by the operation system. Therefore, each micro-benchmark presents the mean of 3 runs unless otherwise mentioned.

$\mathbf{U}$	$\mathbf{B}$	Tr.A.	Te.A.	T
4	128	99.62	92.81	1304
6	128	99.85	92.92	1266
8	128	99.62	92.64	1259
4	256	99.72	92.83	1219
6	256	99.90	92.24	1166

Table 4: LAP-SGD performance.

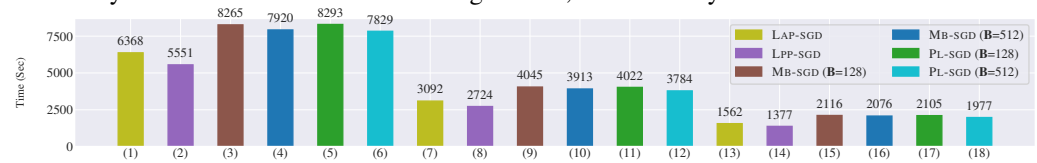
Concurrency on GPUs. We allocate the processes on a GPU up to the availability of the on-device memory. However, once the data-parallel compute resources get

saturated, allocating more processes degrades the performance. For example, see Table 4 which lists the average performance of 5 runs for different combinations of $\mathbf{U}$ and $\mathbf{B}$ for training ResNet-20/Cifar-10 by Lap-SGD on the setting **S1**.

Asynchronous Averaging Frequency. Following PL-SGD, as a general rule, for the first half of the training, call it up until $\mathbf{P}$ epochs, we set the averaging frequency $\mathbf{K} = 1$. However, notice that, unlike PL-SGD, setting a $\mathbf{K} < Q$ in Lap-SGD and Lpp-SGD may not necessarily increase the averaging rounds j in aggregation. Intuitively, in a locally-asynchronous setting, along with the non-blocking (barrier-free) synchronization among GPUs, the increment events on the shared-counter $\mathcal{S}$ would be "grouped" on the real-time scale if the processes $u \in U^q$ do not face delays in scheduling, which we control by allocating an optimal number of processes to maximally utilize the compute resources. For instance, Table 5 lists the results of 5 random runs of ResNet-20/Cifar-10 training with $\mathbf{B} = 128$ and $\mathbf{U} = 6$ with different combinations of $\mathbf{K}$ and $\mathbf{P}$ over **S1**. This micro-benchmark indicates that the overall latency and the final optimization result of our method may remain robust under small changes in $\mathbf{K}$, which it may not be the case with PL-SGD.

K	P	Tr.A.	Te.A.	T
1	150	99.76	93.01	1266
4	150	99.85	92.78	1263
8	150	99.62	93.08	1262
8	0	99.78	92.83	1263
16	150	99.90	93.16	1261

Table 5: Lap-SGD performance.



Method		Q	Tr.A.	Te.A.	T		Q	Tr.A.	Te.A.	T		Q	Tr.A.	Te.A.	T
Lap-SGD	(1)	2	100.0	94.92	6368	(7)	4	100.0	94.91	3092	(13)	8	100.0	94.69	1562
Lpp-SGD	(2)	2	99.97	94.89	5551	(8)	4	99.96	94.87	2724	(14)	8	99.99	93.78	1377
Mb-SGD-128	(3)	2	100.0	93.98	8265	(9)	4	100.0	94.22	4045	(15)	8	100.0	93.83	2116
Mb-SGD-512	(4)	2	95.68	88.42	7920	(10)	4	99.87	89.46	3913	(16)	8	87.54	73.97	2076
Pl-SGD-128	(5)	2	99.99	94.21	8293	(11)	4	99.98	94.02	4022	(17)	8	99.98	94.18	2105
Pl-SGD-512	(6)	2	95.34	88.33	7829	(12)	4	98.52	84.45	3784	(18)	8	89.19	81.80	1977

Table 6: WideResNet-16x8/Cifar-10 training in the settings **S1**: $\mathbf{Q} = 2$, **S2**: $\mathbf{Q} = 4$, and **S3**: $\mathbf{Q} = 8$. Mb-SGD-128 indicates Mb-SGD method with $B_{loc} = 128$ and similarly for others. For Lap-SGD, we spawned 4 concurrent processes on each GPU: $\mathbf{U} = 4$, whereas for Lpp-SGD $\mathbf{U} = 3$ concurrent processes were spawned.

Scalability. Table 6 presents the results of WideResNet-16x8/Cifar-10 training in the three system settings that we consider. We observe that in each setting the relative speed-up of different methods are approximately consistent. In particular, we note the following: (a) reduced communication cost helps PL-SGD marginally outperform Mb-SGD, (b) interestingly, increasing B_{loc} from 128 to 512 does not improve the latency by more than ~4%; this non-linear scalability of data-parallelism was also observed by Lin et al. (2020) , (c) each of the methods scale by more than 2x as the implementation is moved to **S2**, which has 40 CPU cores, from **S1** which has 12 CPU cores, furthermore, this scalability is approximately 2x with respect to performance on **S3** in comparison to **S2**, this shows that for each of the methods we utilize available compute resources maximally, (d) in each setting Lap-SGD achieves ~30% better throughput compared to Mb-SGD standard batch, (e) in each setting Lpp-SGD outperforms Lap-SGD by ~12% making it the fastest method, (f) the training and test accuracy of local large minibatch is poor, and (g) the methods Lap-SGD and Lpp-SGD consistently improve on the baseline generalization accuracy.

Method	B	U	Tr.L.	Te.L.	Tr.A.	Te.A.	T	Method	B	U	Tr.L.	Te.L.	Tr.A.	Te.A.	T
Lap-SGD	128	6	0.295	1.191	**92.23**	**69.43**	**1285**	Mb-SGD	1024	-	0.464	1.212	86.46	67.04	1244
Lap-SGD	128	4	0.299	1.198	**92.15**	**69.78**	**1325**	Mb-SGD	128	-	0.360	1.111	89.93	69.73	1754
Lpp-SGD	128	6	0.404	1.173	88.69	**69.47**	**1085**	Pl-SGD	1024	-	0.373	1.152	89.48	67.97	1198
Lpp-SGD	128	4	0.378	1.161	89.50	**69.75**	**1154**	Pl-SGD	128	-	0.379	1.099	89.29	69.67	1613

Table 7: Performance of ResNet-20 on Cifar-100 over the setting **S1**.

Method	B	U	Tr.L.	Te.L.	Tr.A.	Te.A.	T	Method	B	U	Tr.L.	Te.L.	Tr.A.	Te.A.	T
Lap-SGD	64	4	0.007	1.341	**99.98**	**75.97**	**1564**	Mb-SGD	512	-	0.034	1.918	99.64	63.74	2076
Lpp-SGD	64	3	0.008	1.554	**99.97**	**75.42**	**1376**	Pl-SGD	128	-	0.020	1.354	99.81	72.38	2108
Mb-SGD	128	-	0.007	1.348	99.97	73.29	2115	Pl-SGD	512	-	0.393	1.758	88.91	60.47	1976

Table 8: Performance of WideResNet-16x8 on Cifar-100 over the setting **S3**.

Other Cifar-10/Cifar-100 Results. Performance of the proposed methods in comparison to the baselines for other training tasks on Cifar-10/Cifar-100 datasets are available in Tables 7,

Method	B	U	Tr.L.	Te.L.	Tr.A.	Te.A.	T	Method	B	U	Tr.L.	Te.L.	Tr.A.	Te.A.	T
LAP-SGD	128	4	0.004	0.385	**99.95**	**92.48**	**1251**	MB-SGD	128	-	0.005	0.294	99.93	92.41	2243
LPP-SGD	256	3	0.007	0.355	99.89	**92.55**	**1027**	PL-SGD	1024	-	0.005	0.329	99.93	91.99	1202
MB-SGD	1024	-	0.003	0.325	99.97	91.50	1580	PL-SGD	128	-	0.006	0.283	99.91	92.51	1843

Table 9: Performance of SQUEEZENET on CIFAR-10 over the setting **S1**.

8, and 9. In each case we use $\mathbf{K} = 16$ in LAP-SGD, LPP-SGD, and PL-SGD after 50% of total sample processing budget. As an overview, the relative latency of the methods are as seen in Table 6, whereas in each case LAP-SGD and LPP-SGD recovers or improves the baseline training results.

Imagenet Training Results. Having comprehensively evaluated the proposed methods on CIFAR-10/CIFAR-100, here we present their performance on 1000-classes Imagenet dataset.

	Method	B	U	Tr.L.	Tr.A.	Te.L.	Te.A.	T
(1)	LAP-SGD	32	4	1.319	68.97	1.215	**70.58**	**26237**
(2)	LPP-SGD	32	4	1.397	68.31	1.257	69.91	24066
(3)	MB-SGD	128	-	1.237	70.63	1.220	70.07	26772
(4)	MB-SGD	32	-	1.339	68.53	1.231	70.13	30706
(5)	PL-SGD	128	-	1.256	70.22	1.223	70.02	25850
(6)	PL-SGD	32	-	1.350	68.32	1.236	70.02	31468

Table 10: RESNET-18/IMAGENET Training results.

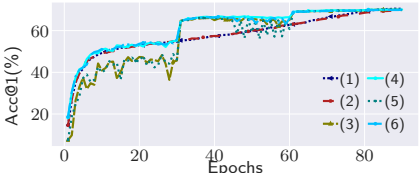


Figure 2: Top-1 Test Accurcy.

Notice that for this training task, with 8 commodity GPUs at our disposal we are very much in the small minibatch setting. Plethora of existing work in the literature efficiently train a RESNET on IMAGENET with BS up to multiple thousands. Other system-dependent constraint that our considered setting faces is that there is hardly any leftover compute resources for us to exploit in the local setting of a worker. Yet, we see for RESNET-18, see Table 10, that LAP-SGD improves on generalization accuracy of the baseline with speed-up.

3.1 ON THE GENERALIZATION OF THE PROPOSED METHODS

Let us comment on a theoretical basis of the remarkable performance in terms of generalization error of our methods. In Lin et al. (2020), the model of SGD algorithms as a form of a Euler-Maruyama discretization of a Stochastic Differential Equation (SDE) presents the perspective that the batch-size can correspond to the inverse of the injected noise. Whereas distributed SGD combined stochastic gradients as such effectively results in an SGD step with a larger batch-size, local SGD, by averaging models rather than gradients, maintains the noise associated with the local small-batch gradients. Given the well-established benefits of greater noise in improving generalization accuracy (e.g. Smith & Le (2018) and others), this presents a heuristic argument as to why local SGD tends to generalize better than distributed SGD. In the Appendix we present an additional argument for why local SGD generalizes well.

However, we see that our particular variation with asynchronous local updates and asynchronous averaging seems to provide additional generalization accuracy above and beyond local SGD. We provide the following explanation as to why this could be the case, again from the perspective of noise as it would appear in a SDE. Let us recall three facts,

1. The noise appearing as a discretization of the Brownian motion term in the diffusion SDE, and, correspondingly the injected noise studied as a driver for increased generalization in previous works on neural network training is i.i.d.,
2. Clearly, the covariances of a mini-batch gradients as statistical estimates of the gradient at x and x' are going to be more similar when x and x' are closer together,
3. (see Section 2) A challenging property from the perspective of convergence analysis with locally asynchronous updates is gradients taken at models taken at snapshots before a previous all-to-all averaging step, and thus far from the current model in memory.

Thus, effectively, the presence of these "highly asynchronous" stochastic gradients, while being potentially problematic from the convergence perspective, effectively brings the analogy of greater injected noise for local SGD over distributed data-parallel closer to accuracy by inducing greater probabilistic independence, i.e., the injected noise, for these updates, is far close to the i.i.d. noise that appears in a discretized SDE.

4 RELATED WORK

In the previous sections we cited the existing literature wherever applicable, in this section we present a brief overview of closely related works and highlight our novelty. In the shared-memory

setting, HOGWILD! (Recht et al. (2011)) is now the classic approach to implement SGD. However, it remains applicable to a centralized setting of a single worker and therefore is not known to have been practically utilized for large scale DNN training. Its success led to designs of variants which targeted specific system aspects of a multi-core machine. For example, Buckwild! (Sa et al. (2015)) proposed using restricted precision training on a CPU. Another variant, called HOGWILD!++ (Zhang et al. (2016b)), harnesses the non-uniform-memory-access (NUMA) architecture based multi-core computers. In this method, threads *pinned* to individual CPUs on a multi-socket mainboard with access to a common main memory, form clusters.

In principle, the proposed LAP-SGD can be seen as deriving from HOGWILD!++. However, there are important differences: (a) at the structural level, the averaging in HOGWILD!++ is directed on a ring graph of thread-clusters, further, it is a token based procedure where in each round only two neighbours synchronize, whereas in LAP-SGD it is all-to-all, (b) in HOGWILD!++ each cluster maintains two copies of the model: a locally updating copy and a buffer copy to store the last synchronized view of the model, whereby each cluster essentially passes the "update" in the local model since the last synchronization to its neighbour, however, this approach has a drawback as identified by the authors: the update that is passed on a ring of workers eventually "comes back" to itself thereby leading to divergence, to overcome this problem they decay the sent out update; as against this, LAP-SGD uses no buffer and does not track updates as such, averaging the model with each peer, similar to L-SGD, helps each of the peers to adjust their optimization dynamics, (c) it is not known if the token based model averaging of HOGWILD!++ is sufficient for training DNNs where generalization is the core point of concern, in place of that we observed that our asynchronous averaging provides an effective protocol of synchronization and often results in improving the generalization, (d) comparing the HOGWILD!++ thread-clusters to concurrent processes on GPUs in LAP-SGD, the latter uses a dedicated process that performs averaging without disturbing the local gradient updates thereby maximally reducing the communication overhead, (e) finally, the convergence theory of LAP-SGD guarantees its efficacy for DNN training, which we demonstrated experimentally, by contrast, HOGWILD!++ does not have any convergence guarantee.

Recently, Wang et al. (2020) proposed Overlap-local-SGD, wherein they suggested to keep a model copy at each worker, very similar to HOGWILD!++, which is simultaneously averaged when sequential computation for multiple iterations happen locally. They showed by limited experiments that it reduced the communication overhead in a non-iid training case based on CIFAR-10, however, not much is known about its performance in general cases. The asynchronous partitioned gradient update of LPP-SGD derives from Kungurtsev et al. (2019), however, unlike them we do not use locks and our implementation setting is decentralized, thus scalable.

5 CONCLUSION

Picking from where Golmant et al. (2018) concluded referring to their findings: *"These results suggest that we should not assume that increasing the batch size for larger datasets will keep training times manageable for all problems. Even though it is a natural form of data parallelism for large-scale optimization, alternative forms of parallelism should be explored to utilize all of our data more efficiently"*, our work introduces a fresh approach in this direction to addressing the challenge.

In our experiments, we observed that the natural system-generated noise in some cases effectively improved the generalization accuracy, which we could not obtain using the existing methods irrespective of any choice of seed for random sampling. The empirical findings suggest that the proposed variant of distributed SGD has a perfectly appropriate place to fit in the horizon of efficient optimization methods for training deep neural networks. As a general guideline for the applicability of our approach, we would suggest the following: monitor the resource consumption of a GPU that trains a CNN, if there is any sign that the consumption was less than 100%, try out LAP-SGD and LPP-SGD instead of arduously, and at times unsuccessfully, tuning the hyperparameters in order to harness the data-parallelism. The asynchronous averaging protocol makes LAP-SGD and LPP-SGD specially attractive to settings with large number of workers.

There are a plethora of small scale model and dataset combinations, where the *critical batch size*– after which the returns in terms of convergence per wall-clock time diminish–is small relative to existing system capabilities (Golmant et al. (2018)). To such cases LAP-SGD and LPP-SGD become readily useful. Yet, exploring the efficiency of LAP-SGD and LPP-SGD to train at massive scales, where hundreds of GPUs enable training IMAGENET in minutes (Ying et al. (2018)), is an ideal future goal. We also plan to extend the proposed methods to combine with communication optimization approaches such as QSGD (Alistarh et al. (2017)).

REFERENCES

Dan Alistarh, Demjan Grubic, Jerry Li, Ryota Tomioka, and M. Vojnovic. Qsgd: Communication-efficient sgd via gradient quantization and encoding. In *NIPS*, 2017.

Nils Berglund. Kramers' law: Validity, derivations and generalisations. *arXiv preprint arXiv:1106.5799*, 2011.

Léon Bottou. Stochastic gradient descent tricks. In *Neural networks: Tricks of the trade*, pp. 421–436. Springer, 2012.

Coralia Cartis and Katya Scheinberg. Global convergence rate analysis of unconstrained optimization methods based on probabilistic models. *Mathematical Programming*, 169(2):337–375, 2018.

Xiaowu Dai and Yuhua Zhu. Towards theoretical understanding of large batch training in stochastic gradient descent. *arXiv preprint arXiv:1812.00542*, 2018.

Noah Golmant, Nikita Vemuri, Zhewei Yao, Vladimir Feinberg, A. Gholami, Kai Rothauge, M. Mahoney, and Joseph E. Gonzalez. On the computational inefficiency of large batch sizes for stochastic gradient descent. *ArXiv*, abs/1811.12941, 2018.

Priya Goyal, P. Dollár, Ross B. Girshick, P. Noordhuis, L. Wesolowski, Aapo Kyrola, Andrew Tulloch, Y. Jia, and Kaiming He. Accurate, large minibatch sgd: Training imagenet in 1 hour. *ArXiv*, abs/1706.02677, 2017.

Kaiming He, X. Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770–778, 2016.

Forrest N. Iandola, Matthew W. Moskewicz, K. Ashraf, Song Han, W. Dally, and K. Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and ¡1mb model size. *ArXiv*, abs/1602.07360, 2017.

Nitish Shirish Keskar, Jorge Nocedal, Ping Tak Peter Tang, Dheevatsa Mudigere, and Mikhail Smelyanskiy. On large-batch training for deep learning: Generalization gap and sharp minima. In *5th International Conference on Learning Representations, ICLR 2017*, 2019.

A. Krizhevsky. Learning multiple layers of features from tiny images. 2009.

A. Krogh and J. Hertz. A simple weight decay can improve generalization. In *NIPS*, 1991.

V. Kungurtsev, Malcolm Egan, Bapi Chatterjee, and Dan Alistarh. Asynchronous stochastic subgradient methods for general nonsmooth nonconvex optimization. *arXiv: Optimization and Control*, 2019.

Tao Lin, S. Stich, and M. Jaggi. Don't use large mini-batches, use local sgd. *ArXiv*, abs/1808.07217, 2020.

LocalSGD. `https://github.com/epfml/LocalSGD-Code`. 2020.

I. Loshchilov and F. Hutter. Sgdr: Stochastic gradient descent with warm restarts. In *ICLR*, 2017.

Nvidia MPS. https://docs.nvidia.com/deploy/mps/index.html. 2020.

Giorgi Nadiradze, Ilia Markov, Bapi Chatterjee, Vyacheslav Kungurtsev, and Dan Alistarh. Elastic consistency: A general consistency model for distributed stochastic gradient descent. *arXiv preprint arXiv:2001.05918*, 2020.

NCCL. `https://docs.nvidia.com/deeplearning/nccl/index.html`. 2020.

Nvidia. `https://www.nvidia.com/de-at/geforce/graphics-cards/rtx-2080-ti/`. 2020.

Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

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

Python. https://docs.python.org/3/glossary.html#term-global-interpreter-lock. 2020.

Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in neural information processing systems*, pp. 693–701, 2011.

Herbert Robbins and Sutton Monro. A stochastic approximation method. *The annals of mathematical statistics*, pp. 400–407, 1951.

Olga Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Zhiheng Huang, A. Karpathy, A. Khosla, M. Bernstein, A. Berg, and Li Fei-Fei. Imagenet large scale visual recognition challenge. *International Journal of Computer Vision*, 115:211–252, 2015.

C. D. Sa, Ce Zhang, K. Olukotun, and C. Ré. Taming the wild: A unified analysis of hogwild-style algorithms. *Advances in neural information processing systems*, 28:2656–2664, 2015.

Christopher J. Shallue, Jaehoon Lee, J. Antognini, Jascha Sohl-Dickstein, Roy Frostig, and G. Dahl. Measuring the effects of data parallelism on neural network training. *J. Mach. Learn. Res.*, 20: 112:1–112:49, 2019.

Samuel L Smith and Quoc V Le. A bayesian perspective on generalization and stochastic gradient descent. In *International Conference on Learning Representations*, 2018.

Sebastian U Stich. Local sgd converges fast and communicates little. *arXiv preprint arXiv:1805.09767*, 2018.

Ilya Sutskever, J. Martens, G. Dahl, and Geoffrey E. Hinton. On the importance of initialization and momentum in deep learning. In *ICML*, 2013.

Jianyu Wang, Hao Liang, and G. Joshi. Overlap local-sgd: An algorithmic approach to hide communication delays in distributed sgd. *ICASSP 2020 - 2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 8871–8875, 2020.

Chris Ying, S. Kumar, Dehao Chen, T. Wang, and Youlong Cheng. Image classification at supercomputer scale. *ArXiv*, abs/1811.06992, 2018.

Sergey Zagoruyko and Nikos Komodakis. Wide residual networks. *ArXiv*, abs/1605.07146, 2016.

Huan Zhang, Cho-Jui Hsieh, and V. Akella. Hogwild++: A new mechanism for decentralized asynchronous stochastic gradient descent. *2016 IEEE 16th International Conference on Data Mining (ICDM)*, pp. 629–638, 2016a.

Huan Zhang, Cho-Jui Hsieh, and Venkatesh Akella. Hogwild++: A new mechanism for decentralized asynchronous stochastic gradient descent. In *2016 IEEE 16th International Conference on Data Mining (ICDM)*, pp. 629–638. IEEE, 2016b.

Jian Zhang, C. D. Sa, Ioannis Mitliagkas, and C. Ré. Parallel sgd: When does averaging help? *ArXiv*, abs/1606.07365, 2016c.

Fan Zhou and Guojing Cong. On the convergence properties of a k-step averaging stochastic gradient descent algorithm for nonconvex optimization. *arXiv preprint arXiv:1708.01012*, 2017.

Martin Zinkevich, M. Weimer, Alex Smola, and L. Li. Parallelized stochastic gradient descent. In *NIPS*, 2010.