Scaling the Wild: Decentralizing Hogwild!-style Shared-memory SGD

Bapi Chatterjee
Department of Computer Science & Engineering
IIIT-Delhi, New Delhi, India
bapi@iiitd.ac.in

Vyacheslav Kungurtsev
Department of Computer Science
Czech Technical University in Prague
vyacheslav.kungurtsev@fel.cvut.cz

Dan Alistarh
Institute of Science and Technology
Austria
dan.alistarh@ist.ac.at

ABSTRACT

Powered by the simplicity of lock-free asynchrony, Hogwild! is a go-to approach to parallelize SGD over a shared-memory setting. Despite its popularity and concomitant extensions, such as PASSM+ wherein concurrent processes update a shared model with partitioned gradients, scaling it to decentralized workers has surprisingly been relatively unexplored. To our knowledge, there is no convergence theory of such methods, nor systematic numerical comparisons evaluating speed-up. In this paper, we propose an algorithm incorporating decentralized distributed memory computing architecture with each node running multiprocessing parallel shared-memory SGD itself. Our scheme is based on the following algorithmic tools and features: (a) asynchronous local gradient updates on the shared-memory of workers, (b) partial backpropagation, and (c) non-blocking in-place averaging of the local models. We prove that our method guarantees ergodic convergence rates for non-convex objectives. On the practical side, we show that the proposed method exhibits improved throughput and competitive accuracy for standard image classification benchmarks on the CIFAR-10, CIFAR-100, and Imagenet datasets. Our code is available at https://github.com/bapi/LPP-SGD.

1 Introduction

Stochastic gradient descent (SGD) is described as

\[ x_{k+1} = x_k - \alpha_k \nabla f(x_k), \]

where after \( k \) iterations, \( x_k \) is the estimate of the solution to the optimization problem: \( \min_{x \in \mathbb{R}^d} f(x) \), \( \alpha_k \) is step-size, and \( \nabla f(x_k) \) is a random variable such that \( \mathbb{E}[\nabla f(x_k)] = \nabla f(x_k) \). In machine learning applications, empirical loss \( f(x) : \mathbb{R}^d \rightarrow \mathbb{R} \) of a model \( x \in \mathbb{R}^d \) over a training set \( S \) is minimized using SGD. Commonly, we sample a (mini) batch \( B_k \subseteq S \) i.i.d. to compute \( \nabla f(x_k) = \nabla f(x_k, B_k) \).

As the size of training set \( S \) grows, adapting SGD to distributed – centralized shared-memory-based and decentralized message-passing-based – systems becomes imperative.

In a shared-memory setting, Hogwild! [Recht et al., 2011] is now a celebrated approach, wherein multiple concurrent processes (or threads) asynchronously compute \( \nabla f(x_k) \) to update a shared model \( x \in \mathbb{R}^d \) without any lock. Subsequently, Hogwild! saw developments such as Buckwild! [Sa et al., 2015], which uses lower precision arithmetic. Recently, Kungurtsev et al. [2021] presented PASSM+, which showed that on the shared-memory of a GPU, partial backpropagation with respect to (w.r.t.) blocks of a convolutional neural network (CNN) by concurrent processes can provably reduce the total training cost while offering a surprisingly competitive optimization result. However, none of these methods can scale beyond a single CPU or GPU.
Scaling the Wild: Decentralizing Hogwild!-style Shared-memory SGD

Table 1: ResNet-20/CIFAR-10 training for 300 epochs on 2 Nvidia GeForce RTX 2080 Ti GPUs. 

| Method   | U | B\textsubscript{loc} | Train Loss | Train Acc. | Test Loss | Test Acc. | Time (Sec) | Speed-up |
|----------|---|----------------------|------------|------------|----------|-----------|------------|----------|
| MB-SGD   | 1 | 128                  | 0.016      | 99.75      | 0.234    | 92.95     | 1754       | 1.0x     |
| MB-SGD   | 1 | 1024                 | 0.023      | 99.51      | 0.293    | 91.38     | 1201       | 1.46x    |
| PL-SGD   | 1 | 128                  | 0.018      | 99.69      | 0.245    | 92.98     | 1603       | 1.09x    |
| PL-SGD   | 1 | 1024                 | 0.154      | 94.69      | 0.381    | 87.81     | 1159       | 1.51x    |
| LAP-SGD  | 4 | 128                  | 0.012      | 99.83      | 0.270    | 92.92     | 1304       | 1.35x    |
| LAP-SGD  | 6 | 128                  | 0.023      | 99.51      | 0.266    | 92.91     | 1281       | 1.37x    |
| LAP-SGD  | 4 | 256                  | 0.010      | 99.90      | 0.280    | 92.93     | 1219       | 1.44x    |
| LPP-SGD  | 4 | 128                  | 0.019      | 99.62      | 0.270    | 92.98     | 1153       | 1.52x    |
| LPP-SGD  | 6 | 128                  | 0.021      | 99.58      | 0.262    | 92.84     | 1085       | 1.62x    |
| LPP-SGD  | 4 | 256                  | 0.019      | 99.65      | 0.267    | 92.95     | 1047       | 1.48x    |

In a decentralized setting, arguably, the most popular distributed SGD variant is \textit{minibatch data-parallel SGD} (MB-SGD) [Bottou, 2012, Dean et al., 2012], described as

$$x_{k+1} = x_k - \alpha_k \frac{1}{|\mathcal{Q}|} \sum_{q \in \mathcal{Q}} \nabla f(x_k, B_q^k).$$

Essentially, $\mathcal{S}$ is distributed over a set of workers/GPUs $\mathcal{Q}$, thereby $B_q^k \subseteq \mathcal{S}$ is sampled locally by a worker $q \in \mathcal{Q}$ at iteration $k$, and in the resulting local models. Clearly, L-SGD reduces the communication cost by reducing its frequency. Other techniques such as quantization [Alistarh et al., 2017] and sparsification [Stich et al., 2018], applied to message-passing distributed SGD, primarily address the high cost of communication. Recently, these methods also have been combined with L-SGD such as Basu et al. [2019], Koloskova et al. [2020]. Nonetheless, beyond using larger mini-batches, none of them can scale the training locally on GPUs though the contemporary GPUs efficiently host multiprocessing and thus HOGWILD! [MPS, 2020, Kungurtsev et al., 2021].

A major motivation for decentralizing HOGWILD!-style approaches in a multi-GPU setting is that models trained by large batch regimes generally have poor quality. It was investigated by Keskar et al. [2019], who described this behaviour in terms of sharpness of minima correlating the minibatch-size, which degrades generalization. Our experiments also highlight this fact, see Table 1. Leveraging multiple processes locally on each GPU can keep the batch-size small.

To our knowledge, only attempt to decentralize HOGWILD! was made by HOGWILD++ [Zhang et al., 2016a] – run HOGWILD! locally on each CPU of a non-uniform-memory-access (NUMA) machine and synchronize by averaging on a ring-graph topology. Taking advantage of cache-locality, HOGWILD++ exhibited improved throughput over HOGWILD! for training a number of convex models. However, HOGWILD++ does not have a convergence theory and no further development to this approach is known.

In this paper, we describe the \textit{locally-partitioned-asynchronous parallel SGD} (LPP-SGD), a new method that employs the following techniques:

1. The model on each worker is shared by multiple processes that perform lock-free partitioned gradient updates and overlaps the gradient computations – it significantly reduces the cost of synchronization.
2. Dedicated processes on each worker concurrently perform the model averaging, which is in-place, non-blocking, and thereby reduces the total gradient computation cost.
In the case of full (non-partitioned) gradient update by each process to the model throughout the training, we call it \textit{locally-asynchronous parallel SGD} (LAP-SGD). Essentially, LAP-SGD and LPP-SGD decentralize HOGWILD! and PASSM+, respectively.

A key contribution of the presented work is the convergence theory – under the standard assumptions, we prove that LAP-SGD and LPP-SGD ergodically converge with standard sublinear rate for non-convex objectives.

Experimental results in Table 1 give a glimpse of the efficacy of our algorithms. In the experiments, we have included \textit{post local SGD} (PL-SGD), which is a modified version of L-SGD by Lin et al. [2020]. PL-SGD synchronizes the distributed models more frequently during the initial phase of training to address the issue of poor generalization of L-SGD.

Our focus in this work is on showcasing the strength of combination of shared-memory and message-passing asynchrony for distributed SGD. Our method can be easily extended to incorporate techniques such as lower precision arithmetic, quantized and sparsified communication, etc.

## 2 System and Optimization Model

| Notation | Description |
|----------|-------------|
| $Q, Q$  | Set of workers (GPUs), Number of workers. |
| $x^q$   | Local shared model on worker $q \in Q$. |
| $a^q$   | A gradient updater process on worker $q \in Q$. |
| $U^q, U$ | Set of updater processes on $q$, Number thereof. |
| $a^q, P^q$ | Averaging process on worker $q$, $\{a^q \cup U^q\}$. |
| $x^q$   | $i^{th}$ block of the model (vector) $x^q$, where $i \in I$. |
| $\nabla f$ | Stochastic gradient of $f$ w.r.t. $i^{th}$ block. |
| $x^q[e]$ | $e^{th}$ index of the local model (vector) $x^q$. |
| $x^q[a, b]$ | Partition of $x^q$ from its $a^{th}$ to $b^{th}$ index. |
| $C^q$   | Shared iteration counter on worker $q \in Q$. |

Table 2: Frequently used notations.

We consider a $d$-dimensional differentiable empirical risk function $f : \mathbb{R}^d \to \mathbb{R}$ and a finite training dataset $S$ as described before. We aim to iteratively converge at a state $x^* \in \mathbb{R}^d$ such that $x^* = \arg \min_{x \in \mathbb{R}^d} f(x)$.

We consider a set of workers $Q$ distributed over a complete graph structured network. We denote the number of workers as $Q = |Q|$ and assume that $Q \geq 2$. A worker $q \in Q$ maintains a local estimate $x^q$ of the model $x$. At initialization, $x^q = x^q \forall q, q' \in Q$. We form, possibly overlapping, blocks of $x^q$ as $x^q := \cup_{i \in I} x^q_i$, where $I$ is a finite index set.

A worker $q \in Q$ spawns a set of concurrent \textit{updater processes} $U^q = \{u^q\}$ and a designated \textit{averaging process} $a^q$. For simplicity, $|U^q| = |U^q| = U \forall q, q' \in Q$. On each $q \in Q$, $x^q$ allows shared read access to the set of concurrent processes $P^q = \{a^q \cup U^q\}$. We assume that each of $u^q \in U^q$ on each of $q \in Q$ have access to the entire training dataset $S$. Thereby, a $u^q \in U^q$ can compute a minibatch stochastic gradient with respect to a block of parameters by sampling a subset $B^q \subseteq S$ typically selected i.i.d. by a distribution $D$ as $\nabla_i f(x^q) := \nabla x^q_i f(x^q, B^q)$. Thus,

$$E_D[\nabla_i f(x)] = \nabla x^q_i f(x).$$

The processes $p^q \in P^q$ maintain a \textit{shared iteration counter} $C^q$, initialized to zero. The operation \texttt{read} and \texttt{inc} atomically increments $C^q$ and returns its value that was read before incrementing. The operation \texttt{read} provides an atomic read. $C^q$ essentially enables ordering the shared gradient updates, which determines the averaging rounds $j$. Finally, we have the following standard assumptions on $f$:

### Lower bound.

$$f(x) \geq f_m \forall x \in \mathbb{R}^d.$$  (4)

### L-Lipschitz Smoothness.

$$\|\nabla f(x) - \nabla f(y)\| \leq L\|x - y\| \forall x, y \in \mathbb{R}^d.$$  (5)

### Bounded second moment.

$$E_D[\|\nabla_i f(x)\|^2] \leq M^2 \forall x \in \mathbb{R}^d.$$  (6)
Bounded variance.
\[
\mathbb{E}_F (|| \hat{\nabla}_i f(x) - \nabla_i f(x)||^2) \leq \sigma^2 \forall x \in \mathbb{R}^d.
\] (7)
The bounded second moment assumption (6) is needed for HOGWILD!-style updates, see Nadiradze et al. [2021a].

3 Algorithm

Algorithm 1: Steps of LPP-SGD on a \( q \in Q \).

Given: model \( x \in \mathbb{R}^d \), training set \( S \), and sample processing budget \( T \) minibatches of size \( B_{loc} \).

Steps of an updater process \( u^q \):
1. Initialize \( s = 0 \); // Process-Local counter.
2. while \( s \leq T \) do
3. Read and increment the shared counter \( C^q \); \( s := \text{read}\&\text{inc}(C^q) \);
4. Lock-free collect a snapshot \( v^{q,s} \) of \( x^q \); \( v^{q,s}[e] := \text{read}(x^q[e]) \), \( \forall 1 \leq e \leq d \);
5. Select a block \( v^{q,s} \) following the Block Selection Rule and sample a minibatch \( B^q_s \subset S \);
6. Use \( B^q_s \) to compute a partial minibatch stochastic gradient w.r.t. the block \( v^{q,s} : \hat{\nabla}_i f(v^{q,s}) \);
7. Lock-free update the model block \( x^q_i : x^q_i[e] := -\alpha_s \hat{\nabla}_i f(v^{q,s})[e] \), \( \forall 1 \leq e \leq \text{length}(v^{q,s}) \);

Steps of the averaging process \( a^q \):
8. Initialize \( s_{\text{cur}} = s_{\text{pre}} = 0 \); // Iteration counters to track updates in the model based on the shared counter \( C^q \) and decide whether to synchronize.
9. Initialize \( j = 0 \); // Averaging round counter.
10. while \( s_{\text{cur}} \leq T \) do
11. Read the current value of the shared counter: \( s_{\text{cur}} := \text{read}(C^q) \);
12. Compute \( K_j \) corresponding to \( s_{\text{cur}} \) following the Synchronization Frequency Scheme;
13. if \( s_{\text{cur}} - s_{\text{pre}} \geq K_j \) then
14. Update counters: \( s_{\text{pre}} := s_{\text{cur}} ; j := j + 1 \);
15. Collect a lock-free snapshot \( v^{q,j} \) of \( x^q \); \( v^{q,j}[e] := \text{read}(x^q[e]) \), \( \forall 1 \leq e \leq d \);
16. Synchronize across the peers in \( Q \) to compute all-reduced-average \( v_j := \frac{1}{q} \sum_{q \in Q} v^{q,j} \);
17. Compute \( \Delta v^{q,j} = v_j - v^{q,j} \);
18. In-place lock-free update: \( x^q[e] := x^q[e] + \Delta v^{q,j} \), \( \forall 1 \leq e \leq d \);

We describe LPP-SGD in Algorithm 1, of which LAP-SGD is a special case. For simplicity, the steps taken by individual processes are given as sequential pseudocode, however, by implementation, each \( a^q \), \( u^q \in P^q \) work concurrently over the same address space associated with a worker \( q \in Q \), and all \( q \in Q \) run in parallel.

Block partitions. Considering a model implemented as an ordered list of tensors, a block is essentially a sub-list of successively ordered tensors. More formally, consider a local model estimate \( x^q \in \mathbb{R}^d \) and a set of positive integers \( \{b_0, b_1, \ldots, b_U\} \) s.t. \( b_0 = 0 \), \( b_U = d \) and \( \sum_{i=1}^U (b_i - b_{i-1}) = d \). We form the blocks \( \{x_i^q\}_{i \in I} \) ranked by the index set \( I = \{0, 1, \ldots, U\} \) s.t. \( x^q := \bigcup_{i \in I} x_i^q \) as the following:
\[
x_i^q := \begin{cases} x^q[1,d], & \text{if } i = 0, \\ x^q[b_{i-1} + 1, b_i], & \text{if } i > 0. \end{cases}
\] (8)

Essentially, block \( x_i^q \) represents the entire model vector \( x^q \) whereas other blocks form disjoint partitions of \( x^q \).

For a CNN model, \( x^q \) represents list of its layers (tensors). We partition it into blocks \( \{x_1^q, \ldots, x_U^q\} \) of lengths \( \{b_i - b_{i-1}\}_{i=1}^U \). The choice of the integers \( b_i \)s are such that the aggregated cost of concurrent partial backpropagation starting from the output layer to these blocks could be minimized [Kungurtsev et al., 2021].

Block Selection Rule. In LAP-SGD, each \( u^q \in U^q \) selects \( x_0^q \), i.e. full model, throughout the training. In LPP-SGD as well, during the initial phase of training, say, while \( s \leq T_{\text{init}} \) for \( s \) read at line 3 in Algorithm 1, each \( u^q \) selects the full model \( x_0^q \). After that, in alternate iterations, select the full model \( x_0^q \) and the block \( x_{\text{rank}(u^q)}^q \) : \( \text{rank}(u^q) \in \{1, \ldots, U\} \).

Discussion on the Rule. As described in Kungurtsev et al. [2021], we have total potential savings of \( F \frac{(c-1)}{2} \) floating point operations (flops) in \( c \) concurrent partial gradient computation steps by \( c \) processes, where \( F \) flops is the cost of
full gradient computation. Kungurtsev et al. [2021] also showed that partial gradient updates throughout the training causes poor optimization results. We observed that at line 3 Algorithm 1 by selecting $s \leq T/10$ roughly, it consistently resulted in the desired optimization quality. Thus, for about $9T/20$ iterations we perform partial gradient computation.

**Synchronization Frequency Scheme.** Our approach to fix the synchronization frequency $K_j$ is similar to that of post local SGD [Lin et al., 2020]: $K_j = 1$ until $T/2$ minibatches are processed, i.e. until $s_{cur}$ reads up to $T/2$ at line 10 in Algorithm 1, where $T$ is the total number of minibatches to be processed; thereafter we increase $K_j$ to $H > 1$. Typically, we choose $H = 8, 16, \text{etc.}$

### 4 Experiments

In this section, we present experimental evaluation of LAP-SGD and LPP-SGD against MB-SGD and PL-SGD schemes.

**Distributed training.** We use CNN models ResNet-20 [He et al., 2016], SqueezeNet [Iandola et al., 2017], and WiderResNet-16x8 [Zagoruyko and Komodakis, 2016] for the 10-100-class image classification tasks on datasets CIFAR-10/CIFAR-100 [Krizhevsky, 2009]. We also train ResNet-50 for a 1000-class classification problem on the IMAGENET [Russakovsky et al., 2015] dataset. We keep the sample processing budget identical across the methods i.e. the training procedure terminates on performing an equal number of epochs (full pass) on the training set. Our data sampling is i.i.d. – we partition 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.

**Platform specification.** We use the following specific settings: (a) S1: two Nvidia GeForce RTX 2080 Ti GPUs, (b) S2: four Nvidia GeForce RTX 2080 Ti GPUs, (c) S3: two S2 settings connected with a 100 GB/s infiniband link, and (d) S4: eight Nvidia V100 GPUs on a cluster. Other system-related details are given in the Appendix D.

**Implementation framework.** We used open-source Pytorch 1.5 Paszke et al. [2017] library. For cross-GPU/cross-machine communication we use NCCL [NCCL, 2020] primitives provided by Pytorch. MB-SGD is based on DistributedDataParallel Pytorch module. The PL-SGD implementation is derived from the author’s code LocalSGD [2020] and adapted to our setting. The autograd package of Pytorch allows specifying the leaf tensors of the computation graph of the CNN loss function w.r.t. which gradients are needed. We used this functionality in implementing partial gradient computation in LPP-SGD.

**Locally-asynchronous Implementation.** We use a process on each GPU, working as a parent, to initialize the CNN model and share it among the spawned child-processes. The child-processes work as updaters $u^q \in U^q$ on a GPU $q \in Q$ to compute the gradients and update the model. Concurrently, the parent-process, instead of remaining idle, as it happens commonly with such designs, acts as the averaging process $a^q$, thereby productively utilizing the entire address space occupied over the GPUs. The parent- and child-processes share the iteration and epoch counters.

**Hyperparameters (HPs).** Each of the methods use identical momentum and weight-decay for a given CNN/dataset case; we rely on their previously used values [Lin et al., 2020]. The LR schedule for MB-SGD and PL-SGD are identical to Lin et al. [2020]. For the LPP-SGD and LAP-SGD, we used cosine annealing schedule without any intermediate restart [Loshchilov and 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 $1.25 \times$ of the warmed-up LR of LAP-SGD for the given case, improves the results. Additional details are in Appendix.

**Experimental Results:**

Result tables use these abbreviations: 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. Irrespective of that, in each micro-benchmark we take the mean of 3 runs with different random seeds unless otherwise mentioned.

| $U$ | $B_{loc}$ | Tr.A. | Te.A. | $T$ | $K$ | P | Tr.A. | Te.A. | $T$ |
|-----|----------|-------|-------|-----|-----|---|-------|-------|-----|
| 4   | 128      | 99.62 | 92.81 | 1304| 1   | 150| 99.76 | 93.01 | 1266|
| 6   | 128      | 99.85 | 92.92 | 1266| 4   | 150| 99.85 | 92.78 | 1263|
| 8   | 128      | 99.62 | 92.64 | 1259| 8   | 150| 99.62 | 93.08 | 1262|
| 4   | 256      | 99.72 | 92.83 | 1219| 16  | 150| 99.78 | 92.83 | 1263|
| 6   | 256      | 99.94 | 92.24 | 1166|     |    |       |       |     |

Table 3: LAP-SGD.

Table 4: LAP-SGD.
Concurrence on GPUs. GPU on-device memory is a constraint. However, once the data-parallel resources get saturated, allocating more processes degrades the optimization quality vis-a-vis performance. For instance, as shown in Table 3—the average performance for different $U$ and $B_{loc}$ for training ResNet-20/Cifar-10 by LAP-SGD on setting $S1$—there is apparently no advantage beyond 6 and 8 concurrent processes with batch-sizes 128 and 256, respectively.

| Method   | $B_{loc}$ | $U$ | $Tr.L.$ | $Te.L.$ | $Tr.A.$ | $Te.A.$ | $T$  |
|----------|----------|-----|---------|---------|---------|---------|-----|
| LAP-SGD  | 64       | 4   | 0.007   | 1.341   | 99.98   | 75.97   | 1564 |
| LPP-SGD  | 64       | 3   | 0.008   | 1.554   | 99.97   | 75.42   | 1376 |
| MB-SGD   | 128      | -   | 0.007   | 1.348   | 99.97   | 73.29   | 2115 |
| MB-SGD   | 512      | -   | 0.034   | 1.918   | 99.64   | 63.74   | 2076 |
| PL-SGD   | 128      | -   | 0.020   | 1.354   | 99.81   | 72.38   | 2108 |
| PL-SGD   | 512      | -   | 0.393   | 1.758   | 88.91   | 60.47   | 1976 |

Table 5: WideResNet-16x8 on Cifar-100 over the setting $S3$.

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

Table 6: SqueezeNet on Cifar-10 over the setting $S1$.

| Method   | $B_{loc}$ | $U$ | $Q$ | $Tr.A.$ | $Te.A.$ | $T$  | $Q$ | $Tr.A.$ | $Te.A.$ | $T$  | $Q$ | $Tr.A.$ | $Te.A.$ | $T$  |
|----------|----------|-----|-----|---------|---------|-----|-----|---------|---------|-----|-----|---------|---------|-----|
| LAP-SGD  | 64       | 4   | (1) | 2       | 100.0   | 94.92 | 6368 | 1       | 100.0   | 94.91  | 3092 | 13   | 100.0   | 94.69   | 1562 |
| LPP-SGD  | 64       | 3   | (2) | 2       | 99.97   | 94.89 | 5551 | 2       | 99.96   | 94.87  | 2724 | 14   | 99.99   | 93.78   | 1377 |
| MB-SGD   | 128      | (3) | 2   | 100.0   | 93.98  | 8265 | (9)  | 4       | 100.0   | 94.22  | 4045 | 15   | 100.0   | 93.83   | 2116 |
| MB-SGD   | 512      | (4) | 2   | 95.68   | 88.42  | 7920 | (10) | 4       | 99.87   | 89.46  | 3913 | 16   | 87.54   | 73.97   | 2076 |
| PL-SGD   | 128      | (5) | 2   | 99.99   | 94.21  | 8293 | (11) | 4       | 99.98   | 94.02  | 4022 | 17   | 99.98   | 94.18   | 2105 |
| PL-SGD   | 512      | (6) | 2   | 95.34   | 88.33  | 7829 | (12) | 4       | 98.52   | 84.45  | 3784 | 18   | 89.19   | 81.80   | 1977 |

Table 7: WideResNet-16x8/Cifar-10 training in the settings $S1$: $Q = 2$, $S2$: $Q = 4$, and $S3$: $Q = 8$.

Synchronization Frequency. As mentioned, we follow the strategy introduced by PL-SGD of setting $K = 1$ for an initial set of iterations followed by increasing $K$. However, unlike PL-SGD, intuitively, with our non-blocking synchronization among GPUs, the increment events on the shared-counter $C^i$ are “grouped” on the real-time scale if the updater processes $v^i$ 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 4 lists the results of 5 random runs for 300 epochs over setting $S1$ of ResNet-20/Cifar-10 training with $B_{loc} = 128$ and $U = 6$ with different synchronization frequencies $K$. $P$ denotes the number of epochs up to which $K$ is necessarily 1. This micro-benchmark indicates that the overall latency and the final optimization result of our method may remain robust under small changes in $K$, which may not necessarily be the case with PL-SGD.

Scalability. Table 7 presents the results of WideResNet-16x8/Cifar-10 training in the settings $S1$, $S2$, and $S3$. We observe that in each setting the relative performance of different methods are approximately consistent. We observe a perfect scalability with the number of GPUs. Interestingly, increasing $B_{loc}$ from 128 to 512 does not improve the latency by more than ~4% of MB-SGD and PL-SGD—this non-linear scalability of data-parallelism was also observed by Lin et al. [2020]. However, there is clear degradation in optimization results by larger $B_{loc}$ for MB-SGD and PL-SGD. On the other hand, LAP-SGD and LPP-SGD not only speeds-up the training but also consistently improve on the baseline generalization accuracy.
Other CIFAR-10/CIFAR-100 Results. Performance of the methods on WIDERESNET-168/CIFAR-100 and SQUEEZENET/CIFAR-10 for 200 epochs are available in Tables 5 and 6, respectively. The relative latency of the methods are as seen in Table 7, whereas, in each case LPP-SGD recovers or improves the baseline results.

| Method     | $B_{loc}$ | $U$ | Tr.L. | Te.L. | Tr.A. | Te.A. | $T$  |
|------------|-----------|-----|-------|-------|-------|-------|------|
| LAP-SGD    | 32        | 4   | 0.823 | 77.97 | 1.215 | 76.58 | 54243|
| LPP-SGD    | 32        | 4   | 0.834 | 77.31 | 1.257 | 76.11 | 48066|
| MB-SGD     | 128       | -   | 0.809 | 78.63 | 1.220 | 76.07 | 52395|
| PL-SGD     | 128       | -   | 0.818 | 78.22 | 1.223 | 76.12 | 50402|

Table 8: RESNET-50/IMAGENET Training results.

Imagenet Training Results. We used the setting S4 for training RESNET-50 on 1000-classes Imagenet dataset. The computational resources of a GPU for RESNET-50/IMAGENET training are almost saturated, which can be seen in higher latency of LAP-SGD in Table 8. Yet, we observe that LPP-SGD exhibits improved performance with competitive generalization accuracy indicating the efficacy of partial back-propagation.

| Method     | $U$ | $B_{loc}$ | Tr.A. | Te.A. | $T$  |
|------------|-----|-----------|-------|-------|------|
| MB-SGD     | 1   | 1024      | 95.84 | 85.67 | 1324 |
| PL-SGD     | 1   | 1024      | 93.67 | 84.69 | 1304 |
| LAP-SGD    | 4   | 256       | 99.87 | 92.90 | 1232 |
| LPP-SGD    | 4   | 256       | 99.59 | 92.83 | 1071 |

Table 9: LARS performance.

LR Tuning strategy. It is pertinent to mention that the techniques, such as LARS [You et al., 2017b], which provide an adaptive LR strategy for large-batch regimes over many GPUs, with each worker locally processing a small batch, are insufficient in the case $B_{loc}$ is increased. For example, see Table 9, which lists the average performance of 5 runs on the setting S1 to train RESNET-20/CIFAR-10 using MB-SGD and PL-SGD combined with LARS with $\eta = 0.001$ [You et al., 2017b]. We proportionately scale the LR from $\alpha_0$ to $\alpha_0 \times \frac{B_{loc}}{B_{loc-base}}$, where $B_{loc-base} = 128$, $Q = 2$ and $\alpha_0 = 0.1$. Evidently, LARS did not help MB-SGD and PL-SGD in checking the poor generalization due to larger $B_{loc}$.

5 Convergence Theory

Note that, unlike LAP-SGD or LPP-SGD, the previous works on Local SGD analyses, e.g. Zhou and Cong [2017], Stich [2018], Haddadpour et al. [2019], etc. did not have to consider either lock-free multiple local gradient updates, or in-place overlapping model updates by averaging. Clearly, significant challenges arise in our case because (a) the snapshots used for gradient computations are inconsistent due to the lock-free concurrent gradient updates, and (b) they also could potentially have been read from the local shared-memory before the last averaging across workers could take place – a novel challenge for any analysis.

Nevertheless, we firstly need to fix a sequence of model estimates for convergence. To this end, we define an abstraction comprising two auxiliary variables as described below. For simplicity we use a constant LR $\alpha$ in this discussion.

Minor iteration view: In the local lock-free setting of each worker $q \in Q$, we follow the elastic consistency model of Nadiradze et al. [2021a] for shared-memory asynchronous SGD. Now, because we are working at the level of blocks, we define the operation $-|_i$ to denote subtraction from the $t^{th}$ block. For example, if $a = (1 \ 2 \ 3 \ 4)^T$ and $b = (-10 \ -20)^T$, we have that $a - |b_{[2,3]}b = (1 \ 12 \ 23 \ 4)^T$. With that, considering worker $q$, the $t + 1^{st}$ minor iteration view after the $j^{th}$ averaging round, assuming that the $s^{th}$ ($s$ is a read value of the shared counter $C^q$) snapshot $v^{q,s}$ was used for partial stochastic gradient computation is defined by the following iteration:

$$w^{q}_{j,t+1} := w^{q}_{j,t} - |_t \alpha \tilde{V}_t(f(v^{q,s})).$$

(9)

Major iteration view: Let $K^q_j$ be the final minor iteration count on worker $q$ before the $(j+1)$st synchronization round, we define the $j^{th}$ major iteration view as

$$\bar{x}_j := \frac{1}{Q} \sum_{q=1}^{Q} w^{q}_{j,K^q_j}.$$  

(10)
This $\bar{x}_j$ is the averaged iterate that we take as the reference sequence in our convergence theory. At initialization, $\bar{x}_0 = v^{q,0}$ $\forall q \in Q$. The update rule of $\bar{x}_j$ can be written as

$$\bar{x}_{j+1} = \frac{1}{Q} \sum_{q=1}^{Q} \left[ \bar{x}_j - \frac{1}{t} \sum_{\ell=0}^{K_q} \nabla f(v^{q,\ell}) \right].$$ (11)

To streamline these views, we use the following notions:

1. The **Snapshot Order** of a minor iteration view is the value of $C^q$ before the snapshot collection.
2. The **Update Order** of a minor iteration view is defined as the order of the atomic read steps of the model before starting its update with a local gradient by a process.
3. The **Iteration Order** of the $j^{th}$ minor iteration after the $j^{th}$ major iteration is the tuple $(j, t)$.

More detail on these orders are provided in Appendix A. It is pertinent to mention here that the selection of a block $i \in I$ depends on the worker $q$ and the iteration order $(j, t)$. Thus, precisely expressing, $i \equiv i(q, j, t)$. Table 10 lists out some extra notations that we will be frequently using in this discussion in addition to those listed out in Table 2.

| Notation | Description |
|----------|-------------|
| $w_{j,t}^q$, $w_{q,s}$ | The minor iteration view with iteration order $(j, t)$, with update order $u$, with snapshot order $s$. |
| $v_{j,t}^q$, $v_{j,t}^q[e]$ | Snapshot vector to compute a partial stochastic gradient for the minor iteration $(j, t)$, its $e^{th}$ index. |
| $\nabla f(v_{j,t}^q)$ | Stochastic gradient of $f(v_{j,t}^q)$ w.r.t. its $i(q, j, t)^{th}$ block. |

Table 10: Additional frequently used notations.

With these views, in order to obtain a converging sequence of averaged iterates, we need to place a reasonable bound on the inconsistencies due to gradient computations for minor iterations using the snapshots that may have components read before the latest major iteration. To this end, we describe a stochastic delay model as described below.

Because the read and updates at the level of indexes of the model vector are atomic, we have the following:

$$v_{j,t}^q[e] \in \bigcup_{k \in \{0, 1, \ldots, u\}} \{w_k^q[e]\},$$ (12)

where $e \in i(q, j, t)$ and $i(q, j, t) \subseteq \{1, 2, \ldots, d\}$ and $u$ is the update order of minor iteration $(j, t)$. Essentially, relation (12) captures the fact that a component of an inconsistent snapshot would belong to a historical minor iteration view. Notice that, as the lock-free read and updates of the shared model work at the level of indexes, we have an advantage of dropping the dependence of the expressions over the blocks $i \in I$ in the convergence analysis. We can now define the event describing relation (12) as the following.

**Delay Event and Probability.** Define $\Gamma_{j,t}^{q,k,e}$ as the event that $v_{j,t}^q[e] = w_k^q[e]$, and $p_{j,t}^{q,k,e}$ as its probability, for some $j \geq 0$, $0 \leq t \leq K_q^j$, and $0 \leq k \leq u$, where $u$ is the update order of minor iteration view $(j, t)$. We call $\Gamma_{j,t}^{q,k,e}$ a delay event. Notice that, defining this event captures the possibility that $v_{j,t}^q$ could have been collected at a minor iteration before the synchronization round $j$.

**Assumptions Governing the Asynchronous Computation.** In the non-convex setting, in general, we can simply hope that the objective function decrease achieved by local asynchronous SGD update with a gradient computed after an averaging step outweighs the potential worst-case increase that might take place due to an update by a gradient computed before the averaging step. In order to derive an expected decrease in the objective, we need to bound the probability of an increase, which means bounding the probability that a component of the snapshot was collected at a minor iteration older than the previous averaging.

The significant quantity that separates these expected favorable and unfavorable iterations is,

$$\Pr \left[ \bigcap_{e \leq k} \bigcup_{j \geq k} \Gamma_{j,t}^{q,k,e} \right],$$

where $u$ is the update order of minor iteration $(j, t)$ and $k$ is the update order of the minor iteration just after the $j^{th}$ synchronization round. This quantity is the probability that every component of the inconsistent snapshots were due to the minor iterations after $a^q$ performed averaging.

**Assumption 5.1.** For all $(j, t, q)$ it holds that the quantity

$$\bar{p}^{(j, t, q)} := \Pr \left[ \bigcap_{e \leq k} \bigcup_{j \geq k} \Gamma_{j,t}^{q,k,e} \right]$$

is finite.
satisfies that there exists \( p_0 > 0 \) such that \( P^{(j,t,q)} > p_0 \) for all \((j, t, q)\).

Assumption 5.1 implies that an inconsistent snapshot read at line 4 to compute a partial stochastic gradient does not consist of components updated by minor iterations too differing in their update orders.

**Elastic Consistency.** For lock-free local updates, we can make a statement to bound the error in the sense of Elastic Consistency of Nadiradze et al. [2021a]. In particular, if \( s \) is the snapshot order of minor iteration \((j, t)\), it holds that,

\[
\mathbb{E} \left[ \left\| u_{j,t}^g - v_{j,t}^g \right\| \mid \cap_{l \geq k} \cup_{j \geq k} \bar{q}_{j,t} \right] \leq \alpha^2 B^2.
\]

In inequation (13) the event \( \cap_{l < k} \cup_{j < k} \bar{q}_{j,t} \) used for conditioning is precisely the cases where the snapshots for computing partial stochastic gradients were taken after the last averaging round, i.e., if every component satisfies this condition then the elastic consistency bound holds.

Note that under this condition, the maximum delay in terms of number of local asynchronous SGD updates is \( \tau_{\text{max}} = \bar{K} := \max_{j,t} K^q_j \). Thus, following Appendix B.6 of Nadiradze et al. [2021a], the following Lemma defines the Elastic consistency constant:

**Lemma 5.1.** \( B = \sqrt{d}KM \), where \( d \) is the dimension of the model, \( \bar{K} \) is the maximum minor iteration count defined above, and \( M \) is as defined for the second moment bound in Equation 6.

We now present the main theorem proposing the ergodic convergence for a general non-convex objective:

**Theorem 5.1.** Under the Assumptions 3, 4, 5, 6, 7, defining \( f_m, L, M, \sigma \) and the stochastic delay model described by Assumption 5.1, it holds that,

\[
\frac{1}{Q} \sum_{j=1}^{J} \sum_{q=1}^{Q} \sum_{t=0}^{K_j^q} \left\{ [\alpha C_1 - \alpha^2 C_2] \mathbb{E} \left[ \| \nabla_i f(\bar{x}) \| \right] \right] + [\alpha C_3 - \alpha^2 C_4] \mathbb{E} \left[ \| \nabla_i f(\bar{x}) \| \right] \leq f(\bar{x}_0) - f_m
\]

where \( C_1, C_2, C_3, \) and \( C_4 \) depend on \( L, B, M, \sigma \) and probabilistic quantities defining the asynchronous computation. Thus for any set of such constants, if sufficiently small \( \alpha = \Theta(1/\sqrt{J}) \) is chosen then Algorithm 1 ergodically converges with the standard \( O(1/\sqrt{J}) \) rate for nonconvex objectives.

The core idea to prove Theorem 5.1 is arguing that the worst-case increase in the objective due to “bad gradient updates”, which are computed over the snapshots collected before the last averaging, are bounded. With such a bound we then derive the convergence of the “good updates” along the lines of standard local SGD analysis such as Zhou and Cong [2017]. We defer the detailed proof to Appendix A.

**Distributed Speedup and Local scalability:**

**Distributed speedup** with the number of workers \( Q \) can be ascertained from the dependence on \( Q \) in Theorem 5.1. In implementation, the proposed methods exhibit an almost linear data-parallel speedup w.r.t. \( Q \), as shown in Table 7, where the effect of synchronization and communication costs is limited.

Formally, now consider, for instance Yu et al. [2019]; in their presentation of interpreting linear speedup from the convergence theory, the learning rate \( \alpha \) is declared to be proportional to \( \sqrt{Q} \). In effect, since, in practice, the gradient updates are averaged across workers, the effective \( \alpha \) is always divided by \( Q \). So then we can consider, in the context of Theorem 5.1, scaling \( \alpha \) by \( Q \) to attempt obtaining linear speedup effectively. However, this change would add an additional factor of \( Q \) to the \( \alpha^2 \) terms in (14), which will negatively balance the effect away from linear speedup, the extent of which will depend on the particular constants.

Additional results indicating linear speedup for local SGD appear in Haddadpour et al. [2019]. However, they use an additional assumption of a \( PL \) inequality, which can relate the gradient norm to the objective value even for nonconvex functions. While their technical analysis is impressive and interesting, we restrict our theoretical results to considering the most general case.

**Local Scalability**, i.e., speedup with the number of local updaters processes \( U \), of asynchronous shared memory SGD can, in theory, achieve linear speedup while \( U \) is bounded by \( O[D] \), where \( D \) is the total number of minibatch updates [Lian et al., 2015], if the delay probabilities are not affected by the number of processes. However, in practice, scheduling an additional process on a GPU has substantial overhead, which does influence the delay probabilities, thus limiting the speedup. For LPP-SGD the more partitions the blocks are separated across, the smaller the size of gradient
update, which decays the theoretical speedup. However, in practice, the reduced gradient computation cost of partial backpropagation handsomely improves the balance towards faster performance.

6 Related Work

In the shared-memory setting, HOGWILD! Recht et al. [2011] is now a celebrated approach to scale SGD. However, it remains applicable to a centralized setting of a single worker, thus, is not known to have been practically utilized for fast large scale CNN 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.

As mentioned HOGWILD!++ [Zhang et al., 2016b], has a decentralized design targeted to NUMA machines, thus, it resembles LAP-SGD. However, there are significant differences, which we discuss in Appendix C. In the local setting of a GPU, the partial backpropagation of LPP-SGD derives from PASSM+ [Kungurtsev et al., 2021], however, unlike them our gradient updates are lock-free. Nonetheless, the similarities end there only – LPP-SGD is a decentralized method unlike PASSM+.

Periodic model averaging after several sequential local updates is now a popular approach for distributed SGD: Woodworth et al. [2020a], Stich [2019], Basu et al. [2019], Nadiradze et al. [2021b]. This method has been analyzed with various assumptions on the implementation setting and the problem: Khaled et al. [2020], Woodworth et al. [2020b], Koloskova et al. [2020]. Yet, unlike us, none of these methods utilize shared-memory asynchrony or partial backpropagation. 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.

Chen et al. [2016] report that asynchrony causes performance degradation based on limited empirical observations on specific hardware. Yet, others [Regatti et al., 2019] showed that asynchrony favors generalization on a distributed parameter server. Our work empirically, as well as with analytical insights, establishes that asynchronous noise – from concurrent updates and from distributed averaging – is a boon for the generalization performance.

7 Generalization Discussion

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 and 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 minibatch gradients as statistical estimates of the gradients at model estimates $x$ and $x'$ are going to be more similar when $x$ and $x'$ are closer together,
3. Locally asynchronous gradients computed via snapshots taken before a previous all-to-all averaging step, and thus far from the current state of the model in memory, present additional noise, though they provide additional challenge for the convergence theory (see Section 5).

Thus, effectively, the presence of these “highly asynchronous” stochastic gradients, while being potentially challenging from the convergence perspective, effectively brings the analogy of greater injected noise for local SGD 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.
8 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”, we introduced a fresh approach in the direction to address the challenging trade-off of optimization quality vs. scalability.

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 LPP-SGD becomes readily useful.

We observed that for RESNET-50/IMAGENET training the performance gain of LPP-SGD over PL-SGD is roughly 5%. However, as other benchmarks showed, predictively, this will reasonably change on higher end GPUs, such as NVIDIA A100 with 40 GB memory and 624 teraflops peak performance. As an outlook, we considerably did not include the distributed scalability techniques such as quantized [Alistarh et al., 2017] or sparsified- [Seide et al., 2014, Alistarh et al., 2018] communication, or for that matter, decentralized scalability methods such as SGP [Assran et al., 2019]. These approaches can always supplement a locally scalable scheme, which is the focus of this work. Our future goal includes combining LPP-SGD and these techniques.

In the experiments, we observed that the system-generated noise in some cases effectively improved the generalization accuracy. 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 utilization was less than 100%, try out LPP-SGD instead of arduously, and at times unsuccessfully, tuning the hyperparameters in order to harness the data-parallelism.

References
Takuya Akiba, S. Suzuki, and K. Fukuda. Extremely large minibatch sgd: Training resnet-50 on imagenet in 15 minutes. ArXiv, abs/1711.04325, 2017.
Dan Alistarh, Demjan Grubic, Jerry Li, Ryota Tomioka, and M. Vojnovic. Qsgd: Communication-efficient sgd via gradient quantization and encoding. In NIPS, 2017.
Dan Alistarh, Torsten Hoefler, Mikael Johansson, Nikola Konstantinov, Sarit Khirirat, and Cédric Renggli. The convergence of sparsified gradient methods. In Advances in Neural Information Processing Systems, pages 5977–5987, 2018.
Mahmoud Assran, Nicolas Loizou, Nicolas Ballas, and Michael G. Rabbat. Stochastic gradient push for distributed deep learning. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, ICML, 2019.
Debraj Basu, Deepesh Data, Can Karakus, and Suhas N. Diggavi. Qsparse-local-sgd: Distributed SGD with quantization, sparsification and local computations. In NeurIPS 2019, pages 14668–14679, 2019.
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, pages 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.
Jianmin Chen, Xinghao Pan, Rajat Monga, Samy Bengio, and Rafal Jozefowicz. Revisiting distributed synchronous sgd. arXiv preprint arXiv:1604.00981, 2016.
Xiaowu Dai and Yuhua Zhu. Towards theoretical understanding of large batch training in stochastic gradient descent. arXiv preprint arXiv:1812.00542, 2018.
Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, Marc’Aurelio Ranzato, Andrew W. Senior, Paul A. Tucker, Ke Yang, and Andrew Y. Ng. Large scale distributed deep networks. In 26th Annual Conference on Neural Information Processing Systems 2012, pages 1232–1240, 2012.
Noah Golmant, Nikita Vemuri, Zhexue 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.
Farzin Haddadpour, Mohammad Mahdi Kamani, Mehrdad Mahdavi, and Viveck R. Cadambe. Local SGD with periodic averaging: Tighter analysis and adaptive synchronization. In NeurIPS, pages 11080–11092, 2019.

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), pages 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.

Ahmed Khaled, Konstantin Mishchenko, and Peter Richtárik. Tighter theory for local SGD on identical and heterogeneous data. In AISTATS, volume 108 of Proceedings of Machine Learning Research, pages 4519–4529. PMLR, 2020.

Anastasia Koloskova, Nicolas Loizou, Sadra Boreiri, Martin Jaggi, and Sebastian U. Stich. A unified theory of decentralized SGD with changing topology and local updates. In ICML, volume 119 of Proceedings of Machine Learning Research, pages 5381–5393. PMLR, 2020.

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

V. Kungurtsev, Malcolm Egan, Bapi Chatterjee, and Dan Alistarh. Asynchronous stochastic subgradient methods for general nonsmooth nonconvex optimization. AAAI, 2021.

Xiangru Lian, Yijun Huang, Yuncheng Li, and Ji Liu. Asynchronous parallel stochastic gradient for nonconvex optimization. In Neural Information Processing Systems 2015, pages 2737–2745, 2015.

Tao Lin, S. Stich, and M. Jaggi. Don’t use large mini-batches, use local sgd. ICLR, 2020.

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

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

Hiroaki Mikami, Hisahiro Suganuma, Pongsakorn U.-Chupala, Yoshiki Tanaka, and Y. Kageyama. Imagenet/resnet-50 training in 224 seconds. ArXiv, abs/1811.05233, 2018.

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

Giorchi Nadiradze, Ilia Markov, Bapi Chatterjee, and Dan Alistarh. Elastic consistency: A general consistency model for distributed stochastic gradient descent. In AAAI, 2021a.

Giorchi Nadiradze, Amirmojtaba Sabour, Peter Davies, Ilia Markov, Shigang Li, and Dan Alistarh. Decentralized {sgd} with asynchronous, local and quantized updates, 2021b.

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

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, pages 693–701, 2011.

Jayanth Regatti, Gaurav Tendolkar, Yi Zhou, Abhishek Gupta, and Yingbin Liang. Distributed sgd generalizes well under asynchrony. 2019 57th Annual Allerton Conference on Communication, Control, and Computing (Allerton), pages 863–870, 2019.

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.

Frank Seide, Hao Fu, Jasha Droppo, Gang Li, and Dong Yu. 1-bit stochastic gradient descent and its application to data-parallel distributed training of speech dnns. In INTERSPEECH, pages 1058–1062, 2014.

Samuel I. 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.
Scaling the Wild: Decentralizing Hogwild!-style Shared-memory SGD

Sebastian U. Stich. Local SGD converges fast and communicates little. In 7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019. OpenReview.net, 2019.

Sebastian U Stich, Jean-Baptiste Cordonnier, and Martin Jaggi. Sparsified sgd with memory. Advances in Neural Information Processing Systems, 31, 2018.

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), pages 8871–8875, 2020.

Blake E. Woodworth, Kumar Kshitij Patel, and Nati Srebro. Minibatch vs local SGD for heterogeneous distributed learning. In NeurIPS 2020, 2020a.

Blake E. Woodworth, Kumar Kshitij Patel, Sebastian U. Stich, Zhen Dai, Brian Bullins, H. Brendan McMahan, Ohad Shamir, and Nathan Srebro. Is local SGD better than minibatch sgd? In ICML, volume 119 of Proceedings of Machine Learning Research, pages 10334–10343. PMLR, 2020b.

Y. You, Z. Zhang, J. Demmel, K. Keutzer, and Cho-Jui Hsieh. Imagenet training in 24 minutes. 2017a.

Yang You, Igor Gitman, and B. Ginsburg. Large batch training of convolutional networks. arXiv: Computer Vision and Pattern Recognition, 2017b.

Hao Yu, Rong Jin, and Sen Yang. On the linear speedup analysis of communication efficient momentum sgd for distributed non-convex optimization. In International Conference on Machine Learning, pages 7184–7193. PMLR, 2019.

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), pages 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), pages 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.
A Convergence Theory

The convergence analysis of our algorithm requires precise modelling and ordering of concurrent processes in time – the gradient updates by processes \( u^q \) at line 7 and the vector difference updates by process \( a^q \) at line 17 for lock-free averaging – in Algorithm 1 on the model \( x^q \) in the shared-memory setting of a worker \( q \in Q \). To this end, we define the following orders.

**Snapshot Order.** The snapshot collections over the shared memory of a worker \( q \in Q \) are ordered by the shared-counter \( C^q \) as read at line 4 in Algorithm 1. Correspondingly, the partial gradients, as computed using these snapshots, get ordered too. We define the snapshot order of a gradient update as the order by value of \( C^q \) of the snapshot collections.

**Update Order.** The updates at each of the components of the model vector \( x^q \) are atomic by means of fetch&add. Furthermore, each process \( u^q \) updates the components of \( x^q \) in the same order (from left to right). This would enable us to order the partial gradient updates by when they write over a common model component. However, given that there are independent lock-free updates over the blocks of the model, we cannot use any particular component, for example, the first component of \( x^q \), to order the gradient updates. To overcome this issue, we assume that each time before updating the model at line 7 in Algorithm 1, an updater process \( u^q \) atomically accesses the reference (pointer) to \( x^q \) over the shared memory. Thus, we order the updates by their atomic access to this reference. We call this order the update order.

**Iteration Order.** Because a single designated process \( a^q \) on each worker \( q \in Q \) performs averaging, the synchronization rounds (corresponding the vector difference updates at line 17 in Algorithm 1) have a natural order, which is also the order of the major iteration views as defined in Section 4. Now, similar to the update order as defined above, we assume that each time before \( a^q \) starts writing over the model \( x^q \), it atomically accesses the memory-reference to it. Thereby, we have a well-defined arrangement of the gradient updates before or after a particular averaging update over the shared model. Thus we define the iteration order of a gradient update as the dictionary order \((j, t)\) to denote that it is the \( t \)th gradient update after the \( j \)th averaging round.

With the above description, each of the minor iteration views as defined in Section 5 has well-defined snapshot, update, and iteration orders. With that, we define the following delay model.

A.1 The Proof Overview

The proof follows the structure of the ergodic convergence proof of K-step local SGD given in Zhou and Cong [2017], wherein in each round of averaging there are \( \sum K_j^q \) total updates to the model vector associated with the \( q \in Q \) workers.

Insofar as the local asynchronous SGD updates are computed over the snapshots close to the globally synchronized model vector at the last averaging step, there is an expected amount of descent, relative to the value at the last averaging step, achieved from these steps. The task of the convergence theory is to quantify how much descent can be guaranteed, and then this is to be balanced with the amount of possible ascent that could occur from computing SGD updates at snapshots collected far from the one at the last averaging step.

In particular, we need to bound the worst-case increase in the objective due to SGD updates computed over the snapshots collected before the updates by the last averaging and scale it based on the probability that such an old snapshot is used for the computation.

**Good and Bad Iterates.** To balance these two cases, the analysis takes an approach, inspired partially by the analysis given in Cartis and Scheinberg [2018] of separating these as “good” and “bad” iterates: the “good” iterates correspond to snapshots (all components thereof) for SGD updates at line 4 read after the last snapshot (all components thereof) for averaging at line 14 was taken in Algorithm 1, with some associated guaranteed descent in expectation, whereas the “bad” iterates are those read before that. Thus we split the expected descent as expected descent with a “good” iterate multiplied by its probability, added with the worst case ascent of a “bad” iterate multiplied by its probability.

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.
A.2 Proof of Main Convergence Theorem

Recall the major iteration views \( \bar{x}_j \) are defined as the average of the last minor iteration views. Restating its update rule here for reference in this section,

\[
\bar{x}_{j+1} = \frac{1}{Q} \sum_{q=1}^{Q} \left[ \bar{x}_j - \frac{\alpha}{\sum_{t=0}^{K_j}} \nabla_i f(v_{j,t}^q) \right],
\]

where \(|_i|\) is the block-restricted subtraction as we defined in Section 5.

We are now ready to prove the convergence theorem. The structure of the proof will follow Zhou and Cong [2017], who derives the standard sublinear convergence rate for local SGD in a synchronous environment for nonconvex objectives.

**Theorem A.1 (Restatement of Theorem (5.1)).** Under Assumptions 3, 4, 5, 6, and 7 together with Assumptions 5.1 on delay probabilities, it holds that,

\[
\frac{1}{Q} \sum_{q=1}^{Q} \sum_{t=0}^{K_j} \left\{ [\alpha C_1 - \alpha^2 C_2] \mathbb{E} \left[ \left\| \nabla_i f(v_{j,t}^q) \right\|^2 + [\alpha C_3 - \alpha^2 C_4] \mathbb{E} \left[ \left\| \nabla_i f(\bar{x}_j) \right\|^2 \right] \right] \right\} \leq f(\bar{x}_0) - f_m
\]

where \( C_1, C_2, C_3, \) and \( C_4 \) depend on \( L, B, M, \sigma \) and probabilistic quantities defining the asynchronous computation. Thus there exists a set of such constants \( s \) t. \( \alpha = \Theta \left( \frac{1}{\sqrt{T}} \right) \) then Algorithm 1 ergodically converges with the standard \( O(1/\sqrt{T}) \) rate for nonconvex objectives.

**Proof.** We begin with the standard application of the Descent Lemma,

\[
f(\bar{x}_{j+1}) - f(\bar{x}_j) \leq -\langle \nabla f(\bar{x}_j), \frac{1}{Q} \sum_{q=1}^{Q} \sum_{t=0}^{K_j} \nabla_i f(v_{j,t}^q) \rangle + \frac{1}{Q} \sum_{q=1}^{Q} \sum_{t=0}^{K_j} \left\| \nabla_i f(v_{j,t}^q) \right\|^2
\]

Now, since \( \mathbb{E} \left[ \nabla_i f(v_{j,t}^q) \right] = \nabla_i f(v_{j,t}^q) \) using the unbiasedness Assumption 3,

\[
-\mathbb{E} \langle \nabla f(\bar{x}_j), \nabla_i f(v_{j,t}^q) \rangle = -\frac{1}{2} \left( \left\| \nabla_i f(\bar{x}_j) \right\|^2 + \mathbb{E} \left[ \left\| \nabla_i f(v_{j,t}^q) \right\|^2 \right] - \mathbb{E} \left[ \left\| \nabla_i f(v_{j,t}^q) \right\|^2 \right] \right)
\]

We now split the last term by the two cases and use Equation 13,

\[
\mathbb{E} \left[ \left\| \nabla_i f(v_{j,t}^q) \right\|^2 \right] - \mathbb{E} \left[ \left\| \nabla_i f(\bar{x}_j) \right\|^2 \right] \\
= \mathbb{E} \left[ \left\| \nabla_i f(v_{j,t}^q) \right\|^2 \right] \cap \mathbb{E} \left[ \left\| \nabla_i f(\bar{x}_j) \right\|^2 \right] + \mathbb{E} \left[ \left\| \nabla_i f(v_{j,t}^q) \right\|^2 \right] \cap \mathbb{E} \left[ \left\| \nabla_i f(\bar{x}_j) \right\|^2 \right] \\
\leq \alpha^2 B^2 \mathbb{P} \left[ \mathbb{E} \left[ \left\| \nabla_i f(\bar{x}_j) \right\|^2 \right] + 2 \left( \mathbb{E} \left[ \left\| \nabla_i f(\bar{x}_j) \right\|^2 \right] \right) \mathbb{P} \left[ \mathbb{E} \left[ \left\| \nabla_i f(\bar{x}_j) \right\|^2 \right] \right] \right]
\]

and thus combining with equation 18 we get the overall bound,

\[
-\mathbb{E} \langle \nabla f(\bar{x}_j), \nabla_i f(v_{j,t}^q) \rangle \\
\leq -\frac{1}{2} \left( \left\| \nabla_i f(\bar{x}_j) \right\|^2 + \mathbb{E} \left[ \left\| \nabla_i f(v_{j,t}^q) \right\|^2 \right] - \alpha^2 B^2 \right) \mathbb{P} \left[ \mathbb{E} \left[ \left\| \nabla_i f(\bar{x}_j) \right\|^2 \right] \right] \\
+ \frac{1}{2} \left( \left\| \nabla_i f(\bar{x}_j) \right\|^2 + \mathbb{E} \left[ \left\| \nabla_i f(v_{j,t}^q) \right\|^2 \right] \right) \mathbb{P} \left[ \mathbb{E} \left[ \left\| \nabla_i f(v_{j,t}^q) \right\|^2 \right] \right] \\
\leq -\frac{1}{2} \left( \left\| \nabla_i f(\bar{x}_j) \right\|^2 + \mathbb{E} \left[ \left\| \nabla_i f(v_{j,t}^q) \right\|^2 \right] \right) \mathbb{P} \left[ \mathbb{E} \left[ \left\| \nabla_i f(\bar{x}_j) \right\|^2 \right] \right] \\
+ \frac{1}{2} \left( \left\| \nabla_i f(\bar{x}_j) \right\|^2 + \mathbb{E} \left[ \left\| \nabla_i f(v_{j,t}^q) \right\|^2 \right] \right) + \frac{\alpha^2 B^2}{2}
\]

Combining these, we get,

\[
\frac{1}{Q} \sum_{q=1}^{Q} \sum_{t=0}^{K_j} \alpha \mathbb{P} \left[ \mathbb{E} \left[ \left\| \nabla_i f(v_{j,t}^q) \right\|^2 \right] \right] - \alpha \left( Q K_j + \frac{B}{2} \right) \mathbb{E} \left[ \left\| \nabla_i f(v_{j,t}^q) \right\|^2 \right] \\
+ \frac{1}{Q} \sum_{q=1}^{Q} \sum_{t=0}^{K_j} \alpha \mathbb{P} \left[ \mathbb{E} \left[ \left\| \nabla_i f(v_{j,t}^q) \right\|^2 \right] \right] - \alpha \left( Q K_j + \frac{B}{2} \right) \mathbb{E} \left[ \left\| \nabla_i f(\bar{x}_j) \right\|^2 \right] \\
\leq f(\bar{x}_j) - f(\bar{x}_{j+1})
\]
Summing over $j$ obtains the expression in Theorem A.1 with $C_1 = \mathbb{P}\left[\bigcap_c \left(\cup_{l \geq k}^{\mu} \mathbb{P}^{q,l,e}_j\right)\right]$, $C_2 = K_j^q + \frac{B}{2}$, $C_3 = \mathbb{P}\left[\bigcap_c \left(\cup_{l \geq k}^{\mu} \mathbb{P}^{q,l,e}_j\right)\right]$, and $C_4 = \frac{B^2}{2}$, whereof, Assumption 5.1 ensures the existence of $C_1$ and $C_3$.

**Remark** In order to achieve asymptotic convergence, Assumption 5.1 can be relaxed further, in particular, it just cannot be summable, i.e., the condition,

$$\sum_{j=1}^{\infty} \sum_{q=1}^{Q} \sum_{t=0}^{K_j^q} \mathbb{P}\left[\bigcap_c \left(\cup_{l \geq k}^{\mu} \mathbb{P}^{q,l,e}_j\right)\right] = \infty$$

is sufficient for guaranteeing asymptotic convergence.

**Discussion.** Now, let us consider two extreme cases: if there is always only one local stochastic gradient update for all workers for all major iterations $j$, i.e., $K_j^q \equiv 1$, any delay means reading a vector in memory before the last major iteration, and thus the probability of delay greater than zero must be very small in order to offset the worst possible ascent.

On the other hand, if in general $K_j^q \gg \tau$, then while the first $\tau$ minor iteration views could be as problematic at a level depending on the probability of the small delay times, for $t > \tau$ clearly the vector $v_j^{q,t}$ satisfies Equation 13.

Thus we can sum up our conclusions in the following statements:

1. Overall, considering distribution for the delays, the higher the mean, variance, and thickness of the tails of this distribution, the more problematic convergence would be,

2. The larger the quantity of local updates each worker performs in between averaging, the more likely a favorable convergence would occur.

The first is of course standard and obvious. The second presents the interesting finding that if you are running asynchronous stochastic gradient updates on local shared memory, performing local SGD with a larger gap in time between averaging results in more robust performance.

This suggests a certain fundamental harmony between asynchronous concurrency and local SGD, more “aggressive” locality, in the sense of doing more local updates between averaging, coincides with expected performance gains and robustness of more “aggressive” asynchrony and concurrency, in the sense of delays in the computations associated with local processes.

In addition, to contrast the convergence in regards to the block size, which in turn depends on the number of updater processes, clearly the larger the block, equivalently, smaller the number of updater processes the faster the overall convergence, since the norms of the gradient vectors appear. An interesting possibility to consider is if a process can roughly estimate or predict when averaging could be triggered, robustness could be gained by attempting to do block updates right after an expected averaging step, and full parameter vector updates later on in the major iteration.

**B An Argument for Increased Generalization Accuracy**

**B.1 Wide and Narrow Wells**

In general it has been observed that whether a local minimizer is wide or narrow, or how “flat” it is, seems to affect its generalization properties [Keskar et al., 2019]. Motivated by investigating the impact of batch size on generalization, Dai and Zhu [2018] analyzed the generalization properties of SGD by considering the escape time from a “well”, i.e., a local minimizer in the objective landscape, for a constant stepsize variant of SGD by modeling it as an overdamped Langevin-type diffusion process,

$$dX_t = -\nabla f(X_t)dt + \sqrt{2\epsilon}dW_t$$

In general “flatter” minima have longer escape times than narrow ones, where the escape time is the expectation in the number of iterations (defined as a continuous parameter in this sense) until the iterates leave the well to explore the rest of the objective landscape. Any procedure that increases the escape time for flatter minima as compared to narrower ones should, in theory, result in better generalization properties, as it is more likely then that the procedure will return an iterate that is in a narrow minimizer upon termination.

Denote with indexes $w$ for a “wide” valley local minimizer and $n$ for a “narrow” value, which also corresponds to smaller and larger minimal Hessian eigenvalues, respectively.
We now contrast two procedures and compare the difference in their escape times for narrow and wider local minimizers. We also have from the same reference, and now with averaging, the escape time from a narrow valley satisfies

\[ E[\tau_n] = H e^{C/\epsilon} \]

and letting the constant \( H \) depend on the type of minimizer, it holds that that \( H_w > H_n \), i.e., this time is longer for a wider valley.

We also have from the same reference,

\[ P[\tau_e > s E[\tau_e]] = e^{-s} \]

### B.2 Averaging

We now contrast two procedures and compare the difference in their escape times for narrow and wider local minimizers. One is the standard SGD, and in one we perform averaging every \( \tau_a \) time. In each case there are \( Q \) workers, in the first case running independent instances of SGD, and in the other averaging their iterates. We average the models probabilistically as it results in a randomized initialization within the well, and thus the escape time is a sequence of independent trials of length \( \tau_a \) with an initial point in the well, i.e., escaping at time \( \tau_e \) means that there are \( Q \lfloor \frac{\tau}{\tau_a} \rfloor \) trials wherein none of the \( Q \) trajectories, while periodically averaged, stay within the local minimizer value.

For ease of calculation, let us assume that \( \tau_a = \frac{1}{2} \mathbb{E}[\tau^w] = 2 \mathbb{E}[\tau^w] \), where \( \tau_e^w \) and \( \tau_n^w \) are the calculated single process escape time from a wide and narrow well, respectively.

If any one of the local runs escapes, then there is nothing that can be said about the averaged point, so a lack of escape is indicated by the case for which all trajectories, while periodically averaged, stay within the local minimizer value.

Now consider if no averaging takes place, we sum up the probabilities for the wide valley that they all escape after time \((i-1)\tau\) time and, given that they do so, not all of them escape after \(i\tau_a\).

\[
E[\tau^w_n] \leq \sum_{i=1}^{\infty} P(\tau^w_e > (i-1)\tau_a)^Q \left(1 - P(\tau^w_e > \tau_a | \tau^w_e > (i-1)\tau_a)^Q\right)i\tau_a \\
\leq \sum_{i=1}^{\infty} \left(e^{-\frac{Q(i-1)}{2}} \left(1 - e^{-\frac{Q}{2}}\right)\right)i\tau_a
\]

For the narrow well this is,

\[
E[\tau^w_n] \geq \sum_{i=1}^{\infty} P(\tau^w_e > (i-1)\tau_a)^Q \left(1 - P(\tau^w_e > \tau_a | \tau^w_e > (i-1)\tau_a)^Q\right)(i-1)\tau_a \\
\geq \sum_{i=1}^{\infty} \left(e^{-2Q(i-1)} \left(1 - e^{-2Q}\right)\right)(i-1)\tau_a
\]

The difference in the expected escape times satisfies,

\[
E[\tau^w_e - \tau^w_n] \leq \sum_{i=1}^{\infty} \left[\left(e^{-\frac{Q(i-1)}{2}} \left(1 - e^{-\frac{Q}{2}}\right)\right) - \left(e^{-2Q(i-1)} \left(1 - e^{-2Q}\right)\right)\right](i-1)\tau_a
\]

Recall that in the case of averaging, if escaping takes place between \((i-1)\tau_a\) and \(i\tau_a\) there were no escapes with less than \(\tau_a\) for \(M\) processors multiplied by \((i-1)\) times trials, and at least one escape between \((i-1)\tau_a\) and \(i\tau_a\), i.e., not all did not escape between these two times.

The expected first escape time for any trajectory among \(Q\) from a wide valley, thus, is,

\[
E[\tau^w_e] \leq \sum_{i=1}^{\infty} \left(1 - P[\tau^w_e > \tau_a]\right)^{\lfloor i-1/Q \rfloor} \left(1 - P[\tau^w_e > \tau_a]^{\lfloor i/Q \rfloor}\right)i\tau_a \\
\leq \sum_{i=1}^{\infty} e^{-\frac{Q(i-1)}{2}} \left(1 - e^{-\frac{Q}{2}}\right)\tau_a i
\]

And now with averaging, the escape time from a narrow valley satisfies

\[
E[\tau^w_n] \geq \sum_{i=1}^{\infty} \left(1 - P[\tau^w_n > \tau_a]\right)^{\lfloor i-1/Q \rfloor} \left(1 - P[\tau^w_n > \tau_a]^{\lfloor i/Q \rfloor}\right)\tau_a (i-1) \\
\geq \sum_{i=1}^{\infty} e^{-2\left(i-1\right)Q} \left(1 - e^{-2iQ}\right)\tau_a (i-1).
\]

Thus the difference in the expected escape times satisfies

\[
E[\tau^w_e - \tau^w_n] \leq \sum_{i=1}^{\infty} \left[\left(e^{-\frac{Q(i-1)}{2}} \left(1 - e^{-\frac{Q}{2}}\right)\right) - \left(e^{-2\left(i-1\right)Q} \left(1 - e^{-2iQ}\right)\right)\right](i-1) + e^{-\frac{Q(i-1)}{2}} (1 - e^{-\frac{Q}{2}})\tau_a.
\]
It is clear from the above expressions that the upper bound for the difference is larger in the case of averaging. This implies that averaging results in a greater difference between the escape times between wider and narrow local minimizers, suggesting that, on average if one were to stop a process of training and use the resulting iterate as the estimate for the parameters, this iterate would more likely come from a flatter local minimizer if it was generated with a periodic averaging procedure, relative to standard SGD. Thus it should be expected, at least by this argument that better generalization is more likely with periodic averaging.

Note that since they are both upper bounds, this isn’t a formal proof that in all cases the escape times are more favorable for generalization in the case of averaging, but a guide as to the mathematical intuition as to how this could be the case.

C More on Related Work

The main contribution of the present work is the algorithm LPP-SGD, where the shared models on each worker receive partial asynchronous gradient updates locally in addition to being averaged across the workers in a non-blocking fashion. On the other hand, in LAP-SGD each local gradient update is on the full model vector. In the existing literature, a variant of celebrated HOGWILD! [Recht et al., 2011], called HOGWILD!++ [Zhang et al., 2016b] was proposed, which 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 may resemble HOGWILD!++. However, there are important differences:

1. At the structural level, the averaging in HOGWILD!++ is binary on a ring graph of thread-clusters. Furthermore, it is a token based procedure where in each round only two neighbours synchronize, whereas in LAP-SGD it is all-to-all.
2. 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. By contrast, 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.
3. It is not known if the token based model averaging of HOGWILD!++ is sufficient for training deep networks where generalization is the core point of concern. As against that, we observed that our asynchronous averaging provides an effective protocol of synchronization and often results in improving the generalization.
4. Comparing the thread-clusters of HOGWILD!++ 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.
5. Finally, we prove convergence of LAP-SGD for non-convex problems, by contrast, HOGWILD!++ does not have any convergence guarantee.

Thus, in its own merit, LAP-SGD is a novel distributed SGD method. However, to our knowledge, this is the first work to suggest averaging the models on distributed workers, where locally they are asynchronously updated with partial stochastic gradients, which is the proposed method LPP-SGD.

D Experiments and Hyperparameters

System Details. More detail on the specific settings: (a) S1: two Nvidia GeForce RTX 2080 Ti GPUs and an Intel(R) Xeon(R) E5-1650 v4 CPU running @ 3.60 GHz with 12 logical cores, (b) S2: four Nvidia GeForce RTX 2080 Ti GPUs and two Intel(R) Xeon(R) E5-2640 v4 CPUs running @ 2.40 GHz totaling 40 logical cores, (c) S3: two S2 settings connected with a 100 GB/s infiniband link, and (d) S4: eight Nvidia V100 GPUs, 2 each on 4 nodes in a supercomputing cluster; each node contains two Intel(R) Xeon(R) CPUs running @ 2.70 GHz totaling 72 logical cores; the nodes on the cluster are managed by slurm and connected with 100 GB/s infiniband links.

Nvidia Multi-process Service. It is pertinent to mention that the key enabler of the presented fully non-blocking asynchronous implementation methodology is the support for 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]. Now, we present extra experimental results.
Hyperparameters’ Details. The hyperparameters of the experiments are provided in Table 13. In each of the experiments, we have a constant momentum of 0.9. In each of them, as mentioned before, we use standard multi-step diminishing learning rate scheduler with dampening factor $\gamma$ at designated epochs for the methods MB-SGD and PL-SGD, whereas, we use cosine annealing for the methods LAP-SGD and LPP-SGD without restart. Cosine annealing starts immediately after warm-up, which in each case happens for the first 5 epochs. For LPP-SGD, we fix initially $E_{st}$ number of epochs when we perform full asynchronous SGD updates i.e. LAP-SGD.

More CIFAR-10/CIFAR-100 Results. Performance of the methods for RESNET-20/CIFAR-100 for 300 epochs and SQUEEZENET/CIFAR-100 for 200 epochs are available in Tables 11 and 12, respectively. In LAP-SGD, LPP-SGD, and PL-SGD we use $K = 1$ up to 150 and 100 epochs in the two respective experiments and after that $K = 16$. In summary, as seen before, the relative latency of the methods are as seen in Table 7, whereas in each case LAP-SGD and LPP-SGD recovers or improves the baseline training results.

| Method   | $B_{loc}$ | $U$ | Tr.L.  | Te.L.  | Tr.A.  | Te.A.  | $T$ |
|----------|-----------|-----|--------|--------|--------|--------|-----|
| LAP-SGD  | 128       | 6   | 0.295  | 1.191  | 92.23  | 69.43  | 1285|
| LAP-SGD  | 128       | 4   | 0.299  | 1.198  | 92.15  | 69.78  | 1325|
| LPP-SGD  | 128       | 6   | 0.404  | 1.173  | 88.69  | 69.47  | 1085|
| LPP-SGD  | 128       | 4   | 0.378  | 1.161  | 89.50  | 69.75  | 1154|
| MB-SGD   | 1024      | -   | 0.464  | 1.212  | 86.46  | 67.04  | 1244|
| MB-SGD   | 128       | -   | 0.360  | 1.111  | 89.93  | 69.73  | 1754|
| PL-SGD   | 1024      | -   | 0.373  | 1.152  | 89.48  | 67.97  | 1198|
| PL-SGD   | 128       | -   | 0.379  | 1.099  | 89.29  | 69.67  | 1613|

Table 11: RESNET-20 on CIFAR-100 over the setting S1.

| Method   | $B_{loc}$ | $U$ | Tr.L.  | Te.L.  | Tr.A.  | Te.A.  | $T$ |
|----------|-----------|-----|--------|--------|--------|--------|-----|
| LAP-SGD  | 128       | 4   | 0.037  | 1.549  | 99.56  | 70.17  | 1263|
| LPP-SGD  | 128       | 4   | 0.070  | 1.504  | 98.79  | 69.97  | 1071|
| MB-SGD   | 1024      | -   | 0.043  | 1.294  | 99.60  | 68.30  | 1684|
| MB-SGD   | 128       | -   | 0.049  | 1.178  | 99.29  | 70.06  | 2134|
| PL-SGD   | 1024      | -   | 0.061  | 1.259  | 99.25  | 68.46  | 1209|
| PL-SGD   | 128       | -   | 0.062  | 1.207  | 99.00  | 69.38  | 1857|

Table 12: SQUEEZENET on CIFAR-100 over the setting S1.

| Exp. | Model      | Dataset   | Epochs | $B_{loc}$ | $\alpha$ | $\alpha_{DP}^U$ | $\alpha_{DP}^L_{AP}$ | $\alpha_{DP}^L_{PP}$ | $\gamma$ | Dampen-ing at | Wt. decay | $T_{st}$ |
|------|------------|-----------|--------|-----------|----------|----------------|---------------------|---------------------|---------|-------------|----------|---------|
| Table 1 | RESNET-20   | CIFAR-10  | 300    | 128       | 0.1      | 0.2            | 0.2                 | 0.25                | 0.1     | 150, 225    | 0.0008   | 20      |
| Table 7 | WIDERESNET-16x8 | CIFAR-10  | 200    | 128       | 0.1      | 0.2            | 0.2                 | 0.2                 | 0.1     | 60, 120, 160 | 0.0001   | 20      |
| Table 5 | WIDERESNET-16x8 | CIFAR-100 | 200    | 128       | 0.1      | 0.8            | 0.8                 | 1.0                 | 0.2     | 60, 120, 160 | 0.0001   | 20      |
| Table 6 | SQUEEZENET  | CIFAR-10  | 200    | 128       | 0.1      | 0.2            | 0.2                 | 0.2                 | 0.2     | 60, 120, 160 | 0.0001   | 20      |
| Table 8 | RESNET-50   | IMAGENET  | 90     | 32        | 0.0125   | 0.8            | 0.1                 | 0.125               | 0.1     | 60, 120, 160 | 0.00001  | 20      |
| Table 11 | RESNET-20   | CIFAR-100 | 300    | 128       | 0.1      | 0.2            | 0.2                 | 0.2                 | 0.1     | 150, 225    | 0.0008   | 20      |
| Table 12 | SQUEEZENET  | CIFAR-100 | 200    | 128       | 0.1      | 0.2            | 0.2                 | 0.2                 | 0.2     | 60, 120, 160 | 0.0001   | 20      |

Table 13: Hyperparameters used in the experiments. $\alpha$: Initial Learning Rate (LR), $\alpha_{DP}^U$: Warmed-up LR for Data-parallel methods MB-SGD and PL-SGD with standard batch size $B_{loc}$ on each GPU (for large local batch sizes we scale the LR in the ratio of used batch size and $B_{loc}$), $\alpha_{DP}^L_{AP}$: Warmed-up LR for LAP-SGD, $\alpha_{DP}^L_{PP}$: Warmed-up LR for LPP-SGD. In general, we follow the standard LR warm-up strategy as discussed before: warm up for the first 5 epochs to scale from $\alpha$ to $\alpha \times \frac{B_{loc} \times Q}{B_{loc}}$, where $Q$ is the number of workers. “Dampening at” are the epoch counts where we dampen $\alpha_{DP}^U$ by the multiple $\gamma$ following the standard multi-step diminishing LR tuning scheme.