LAGA: Lagged AllReduce with Gradient Accumulation for Minimal Idle Time

Ido Hakimi*, Rotem Zamir Aviv†, Kfir Y. Levy† and Assaf Schuster*
*Department of Computer Science, †Department of Electrical Engineering
Technion - Israel Institute of Technology, Haifa, Israel
Email: idohakimi@gmail.com

***Abstract*—Training neural networks on large distributed clusters has become a common practice due to the size and complexity of recent neural networks. These high-end clusters of advanced computational devices cooperate together to reduce the neural network training duration. In practice, training at linear scalability with respect to the number of devices is difficult, due to communication overheads. These communication overheads often cause long idle times for the computational devices. In this paper, we propose LAGA (Lagged AllReduce with Gradient Accumulation): a hybrid technique that combines the best of synchronous and asynchronous approaches, that scales linearly. LAGA reduces the device idle time by accumulating locally computed gradients and executing the communications in the background. We demonstrate the effectiveness of LAGA in both final accuracy and scalability on the ImageNet dataset, where LAGA achieves a speedup of up to 2.96x and 5.24x less idle time. Finally, we provide convergence guarantees for LAGA under the non-convex setting.**

***Index Terms*—optimization, non-convex, neural networks**

https://github.com/idoh/LAGA-Benchmark

1. Introduction

With the increase of available training data, the size and complexity of recent neural networks [58] [15] has grown tremendously. The training of these neural networks is time consuming and requires high-end clusters of advanced computational devices (workers). Thus, leveraging high-end computing environments (such as GPU clusters) to distribute the training process across multiple workers has become common practice in the training of neural networks. However, training a single model can still take up to several months on high-end clusters to complete [45]. Hence, reducing the training duration and making it more accessible is essential.

There are several techniques for optimizing neural networks, but the undoubtedly most popular is Stochastic Gradient Descent (SGD) or its variants [27] [31]. SGD is inherently sequential, which makes distributing the training process difficult [9]. Training on a large number of workers without making adjustments can have devastating effects on the training process and the final model quality [26] [21].

The common approach for distributing the computations on multiple workers is synchronous SGD (SSGD). In this approach, each worker computes its gradient on a different batch of examples. The gradients are then averaged across all workers and the parameters are updated with the average gradient. In the master-less setting, which is the popular approach today [51], the parameters are updated locally at every worker after each gradient synchronization. Before computing the next gradient, the synchronization operation has to be finished, during which the computational device is idle. SSGD suffers from high communication overheads [30], and without addressing this issue, training on more workers can overshadow its benefits. Thus, efficient communication is paramount for actual training speedups.

The AllReduce collective communication protocol is frequently used in distributed training [42] [29] for synchronizing gradients, thanks to its efficient NCCL [23] Ring-AllReduce implementation and almost constant communication complexity [56]. The communication complexity of Ring-AllReduce is theoretically independent of the number of workers [16] [5] and therefore it remains constant regardless of the number of workers. In practice, Ring-AllReduce is a key component for training neural networks on large clusters [6] [35].

Due to the massive size of recent neural networks, even an optimized AllReduce operation can still take a long time. Reducing the communication overhead reduces the device idle time and results in higher speedups. Reducing the synchronization frequency, which can be done both explicitly [55] [20] [12] [3] or implicitly by training on larger batch sizes [47] [24], reduces the communication overhead. [17] have empirically demonstrated that large-batch training can achieve fast convergence rate and high final accuracy when scaling the learning-rate linearly with respect to the batch size. To compensate for the large learning-rate, [17] introduced a learning-rate warm-up during the initial training epochs.

A recent systematical approach [29] [49] to hide the communication overhead, which we define as SSGD-OPT, suggested to overlap the computations of the backpropagation with the gradient synchronization. In SSGD-OPT, during the backpropagation the layers whose gradient has already been computed are synchronized in the background while the worker continues to compute the gradient of the rest of the layers. The gradient synchronization of the last layers is executed first since the backpropagation computes the gradient from the last to the first layer. This technique was shown to gain 15-25% speedup, depending on the neural architecture

and the cluster hardware topology, bandwidth, and latency. In cases where the backpropagation compute time is larger than the gradient synchronization time, SSGD-OPT can, in theory, completely hide all of the communication overhead. However, as detailed in Section 3, not all neural architectures are alike, so in some cases the gradient synchronization time is considerably larger than the backpropagation compute time.

Gradient accumulation has become standard feature in many deep learning training frameworks [42] [1] [14]. The basic idea of gradient accumulation is to accumulate gradients from multiple micro-batches and only then update the model parameters. This is particularly helpful in training very large neural networks [22], where workers can only fit one small micro-batch at a given time. From an optimization perspective, gradient accumulation is completely equivalent to training with a larger mini-batch size, since in both cases the gradient is averaged with respect to all computed examples. However, when combined with SSGD-OPT it is less efficient than one large batch, because the communications are overlapped only with the backpropagation of the last micro-batch. Our theoretical analysis shows that the communication savings of SSGD-OPT compared to SSGD are independent by the amount of gradient accumulations and therefore pairing SSGD-OPT with gradient accumulation is not efficient.

We propose LAGA, a novel and efficient algorithm that leverages gradient accumulation to further hide the communication overheads. LAGA relaxes the synchronization barrier by executing the gradient synchronization in the background, which introduces a *lag* into the optimization process. We further present a variation of LAGA that integrates Nesterov's Accelerated Gradient [34] for the lagged setting. LAGA achieves near linear scaling with a speedup of up to 2.96x and 5.24x less idle time compared to SSGD on the ImageNet dataset. We conduct a thorough evaluation on a variety of neural architectures and show that LAGA outperforms an optimized implementation of SSGD-OPT. We summarize the contributions of our paper as follows:

- We propose LAGA for efficient communications and demonstrate its speedup gains on a variety of neural network architectures.
- We conduct a theoretical communication overhead analysis which derives the optimal amount of gradient accumulations for LAGA to achieve linear scalability.
- We propose a variation for LAGA with Nesterov's Accelerated Gradient that achieves the same final accuracy as SSGD.
- We provide an open-source implementation of LAGA in both PyTorch [36] and Horovod [42] frameworks.
- We provide a convergence rate proof for LAGA.

2. Problem Setting

Generally, the optimization process of neural networks minimizes an objective function f given parameter vector $\mathbf{x}$. The value of $f(\mathbf{x})$ indicates how far from perfect the model is, given the parameter vector. At each iteration, a gradient is computed with respect to $\mathbf{x}$ and a training sample ξ. The gradient $\nabla F(\mathbf{x};\xi)$ is then used to update the model parameters vector, where $f(\mathbf{x}) \triangleq \mathbb{E}_{\xi\sim\mathcal{D}}[F(\mathbf{x};\xi)]$ is a smooth non-convex function. The SGD update step is defined as:

$$\mathbf{x}^{t+1} = \mathbf{x}^t - \eta\nabla F(\mathbf{x}^t;\xi^t) \ , \tag{1}$$

where η denotes the learning rate and $\mathbf{x}^t$ the model parameter vector at iteration t. SGD converges by taking iterative steps in the form of Equation (1) towards the minima.

Extending this notion to distributed neural network, consider the typical training setting where all N workers (computational devices) cooperate to minimize the given objective problem f:

$$\min_{\mathbf{x}\in\mathbb{R}^d} \quad f(\mathbf{x}) \triangleq \frac{1}{N}\sum_{i=1}^{N} f_i(\mathbf{x}) \ , \tag{2}$$

where $f_i(\mathbf{x}) \triangleq \mathbb{E}_{\xi_i\sim\mathcal{D}_i}[F_i(\mathbf{x};\xi_i)]$ is a smooth non-convex function and $\mathcal{D}_i$ can be possibly different for different i.

Algorithm 1 SSGD

1: Compute gradient $\mathbf{g}_i^t \leftarrow \nabla F_i\left(\bar{\mathbf{x}}^t;\xi_i^t\right)$
2: AllReduce $\bar{\mathbf{g}}^t \leftarrow \frac{1}{N}\sum_{i=1}^{N}\mathbf{g}_i^t$
3: Update local parameters $\bar{\mathbf{x}}^{t+1} \leftarrow \bar{\mathbf{x}}^t - \eta\bar{\mathbf{g}}^t$

The SSGD algorithm is described in Algorithm 1. Each worker i computes its gradient $\mathbf{g}_i^t \triangleq \nabla F_i\left(\bar{\mathbf{x}}^t;\xi_i^t\right)$ on a different sample from the training set ξ_i^t, where $\mathbb{E}_{\xi_i}[\mathbf{g}_i^t|\boldsymbol{\xi}^{[t-1]}] = \nabla f_i(\bar{\mathbf{x}}^t)$ and $\boldsymbol{\xi}^{[t-1]}$ denotes all the randomness up to iteration $t-1$. The gradients of the different workers are then averaged with an AllReduce operation $\bar{\mathbf{g}}^t \triangleq \frac{1}{N}\sum_{i=1}^{N}\mathbf{g}_i^t$. Finally, the parameters $\bar{\mathbf{x}}$ are updated with the average gradient $\bar{\mathbf{x}}^{t+1} = \bar{\mathbf{x}}^t - \eta\bar{\mathbf{g}}^t$. Although each worker holds a local copy of the model's parameters, these parameters are identical across workers since they are updated with the same average gradient $\bar{\mathbf{g}}^t$ and initialized with the same $\mathbf{x}^0$.

Notice that in Algorithm 1 the AllReduce operation in line 2 has to be finished before updating the parameters in line 3, meaning the worker has to wait during the AllReduce operation. This can cause long idle times in cases where the AllReduce operation is slow.

Algorithm 2 SSGD with Gradient Accumulation τ

1: Compute gradient $\mathbf{g}_i^t(\tau) \leftarrow \frac{1}{\tau}\sum_{j=1}^{\tau}\nabla F_i\left(\bar{\mathbf{x}}^t;\xi_{i,j}^t\right)$
2: AllReduce $\bar{\mathbf{g}}^t(\tau) \leftarrow \frac{1}{N}\sum_{i=1}^{N}\mathbf{g}_i^t(\tau)$
3: Update parameters $\bar{\mathbf{x}}^{t+1} \leftarrow \bar{\mathbf{x}}^t - \eta\bar{\mathbf{g}}^t(\tau)$

Algorithm 2 describes the SSGD algorithm with gradient accumulation, where τ denotes the amount of local gradient accumulations. Each worker accumulates τ local gradients $\mathbf{g}_i^t(\tau) \triangleq \frac{1}{\tau}\sum_{j=1}^{\tau}\nabla F_i\left(\bar{\mathbf{x}}^t;\xi_{i,j}^t\right)$, which are then used for the AllReduce operation. The gradients satisfy $\mathbb{E}[\mathbf{g}_i^t(\tau)|\boldsymbol{\xi}^{[t-1]}] =$

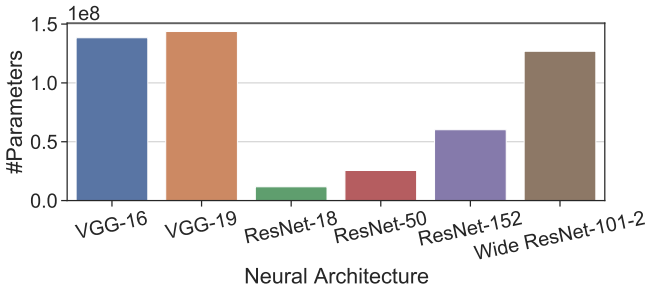
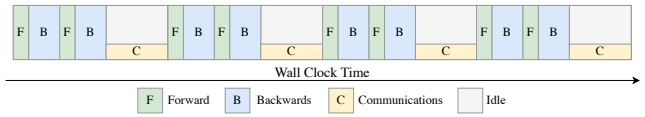
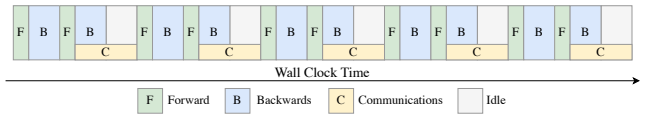


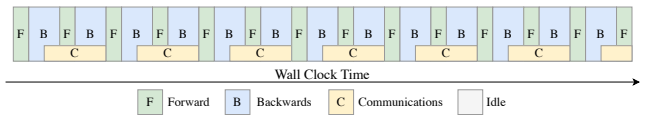
Figure 1: Neural architecture sizes (#Parameters).



(a) SSGD



(b) SSGD-OPT



(c) LAGA

Figure 2: Synchronization diagrams of the different algorithms where the gradient accumulation is two ($\tau = 2$). Notice, that SSGD has a large idle while SSGD-OPT reduces the idle time by overlapping the communications with the last back-propagation. LAGA eliminates all idle time by completely overlapping the communications with the forward and backward computations.

$\nabla f_i(\bar{\mathbf{x}}^t)$, where $\boldsymbol{\xi}^{[t-1]}$ denotes all the randomness up to iteration $t-1$, i.e. $\boldsymbol{\xi}^{[t-1]} \triangleq [\xi_{i,j}^k]_{i\in\{1,\cdots,N\},j\in\{1,\cdots,\tau\},k\in\{1,\cdots,t-1\}}$. Hence, the AllReduce operation $\bar{\mathbf{g}}^t(\tau) \triangleq \frac{1}{N}\sum_{i=1}^N \mathbf{g}_i^t(\tau)$ occurs only after τ gradient computations, which reduces the synchronization frequency.

3. Communication Overhead Analysis

In this section we analyse the communication overhead and the maximal achievable speedup when eliminating all communication overheads. We define T_{comp} and T_{comm} as the computation and communication time of a single iteration. The computation time T_{comp} equals the sum of the forward T_{fwd} and backpropagation T_{bwd} compute time, $T_{comp} = T_{fwd} + T_{bwd}$. In the SSGD algorithm, the sum of T_{comp} and T_{comm} determines the wall clock time of fully executing a single batch as shown in Figure 3. The ratio between T_{comp} and T_{comm} indicates the communication overhead which we define as $\psi = \frac{T_{comm}}{T_{comp}+T_{comm}}$, where $0 \leq \psi \leq 1$. A linear scalable algorithm would have an optimal communication overhead of $\psi = 0$, whereas an inefficient algorithm would have a high ψ. Deriving from ψ, the optimal achievable speedup from completely eliminating all the communication overhead of SSGD is $\frac{T_{comp}+T_{comm}}{T_{comm}}$. This is what we call linear scalability, and it is the maximal achievable speedup with respect to SSGD for a communication efficient algorithm. Thus, a high communication overhead indicates that there is a higher potential for speedup gains from reducing the communication time.

The communication time T_{comm} is linearly depended on the neural architecture size. That is, neural architectures with more parameters require synchronizing a larger gradient vector since the gradient is the same size as the number of parameters if no additional gradient compression methods are used [52]. The current trend in neural networks is to increase the architecture size [44] [39], which has delivered promising state-of-the-art results. In return, the T_{comm} also grows, which can increase the communication overhead. The current growth of these network architectures is much faster than the acceleration of the hardware NIC (network interface controller) and therefore even with the latest hardware, T_{comm} still grows every year.

Figure 1 shows the number of parameters on a variety of neural architectures. Neural architectures aren't alike; architectures with more parameters don't necessarily require more computation time. The neural architecture design attributes heavily affects its communication overhead and therefore its scalability complexity. Some of the most impactful design attributes include the hidden layer dimensions and convolution sizes. For example the VGG-16 [46] neural architecture is a short neural network that uses large convolutional layers which has a considerably smaller T_{comp} than Wide ResNet-101-2 [57] (Figure 3), although both neural architectures have roughly the same amount of parameters as shown in Figure 1. As a result, the communication overhead of VGG-16 is higher than that of Wide ResNet-101-2, which makes scaling VGG-16 more difficult.

Combining SSGD with gradient accumulation reduces the communication time T_{comm} by a factor of τ (the amount of gradient accumulations), since the synchronization frequency is reduced to every τ iterations as shown in Figure 2. We define the communication overhead $\psi_{SSGD}(\tau)$ of SSGD with respect to τ as:

$$\psi_{SSGD}(\tau) \triangleq \frac{T_{comm}}{\tau \cdot T_{comp} + T_{comm}} \ , \tag{3}$$

where $\psi_{SSGD}(\tau)$ is inversely depended on τ. Notice that $\psi_{SSGD}(\tau)$ shrinks as τ grows, however, only for large values of τ does SSGD achieve near linear scalability. Since increasing τ also increases the SGD effective batch, we would like to not increase it indefinitely, as after a certain point increasing τ would hurt the final accuracy [43]. Therefore, additional techniques are required to further hide the communications.

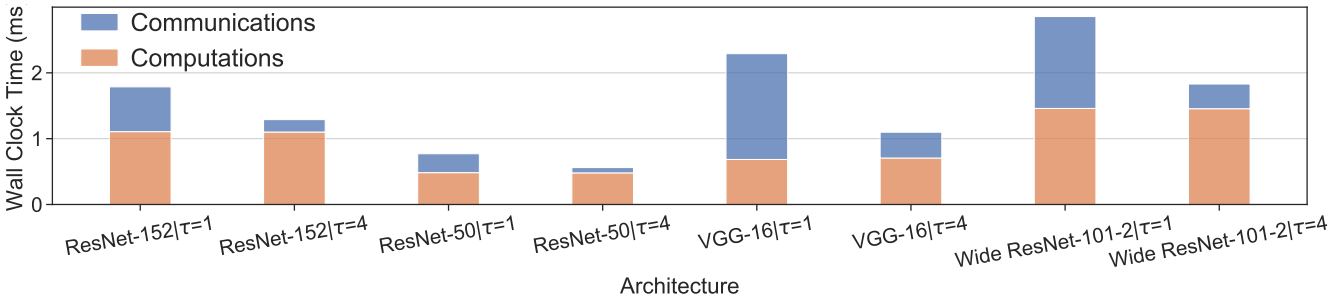


Figure 3: The computation and communications wall clock time of SSGD. For each of these neural architectures we ran 5000 iterations on eight workers with a micro-batch size of 32. We see that ResNet-50 [19] has a much lower ψ than VGG-16 [46] and therefore has less potential for speedup gains from reducing the communication overhead. Next we look at the effect of τ which lowers the wall clock time. Notice that when we change τ the T_{comp} maintains the same and only T_{comm} shrinks down. We see that even when $\tau = 4$ the T_{comm} is still noticeable, for example the VGG-16 architecture with $\tau = 4$ has roughly the same ψ as ResNet-50 with $\tau = 1$.

Next we analyse the communication overhead of SSGD-OPT $\psi_{OPT}(\tau)$ which overlaps the communication with the backpropagation compute time T_{bwd} of the last micro-batch.

$$\psi_{OPT}(\tau) = \begin{cases} 0 & T_{comm} \leq T_{bwd} \\ \frac{T_{comm} - T_{bwd}}{\tau \cdot T_{comp} + T_{comm}} & \text{Otherwise.} \end{cases} \quad (4)$$

Based on the communication overhead of SSGD-OPT and SSGD, Equation (4) and Equation (3) respectively, we analyse the theoretical communication savings of SSGD-OPT compared to SSGD when the communication time is high.

$$\psi_{SSGD}(\tau) - \psi_{OPT}(\tau) \leq \frac{T_{bwd}}{\tau \cdot T_{comp} + T_{comm}} \quad (5)$$

Equation (5) shows that the communication savings of SSGD-OPT compared to SSGD diminish as τ grows. So as τ grows the backpropagation overlapping of SSGD-OPT will be less important. We note that in scenarios where $T_{comm} > T_{bwd}$ the communication overhead of SSGD-OPT will never be completely eliminated even for large values of τ. Only in cases where $T_{comm} \leq T_{bwd}$ will SSGD-OPT achieve true linear scalability, which does not hold in practice for many neural architectures as shown in Figure 3. Furthermore, the term $T_{comm} \leq T_{bwd}$ is independent of τ and therefore increasing τ does not aid it.

4. LAGA

Algorithm 3 LAGA-SGD

1: Compute gradient $\mathbf{g}_i^t(\tau) \leftarrow \frac{1}{\tau}\sum_{j=1}^{\tau} \nabla F_i\left(\bar{\mathbf{x}}^t; \xi_{i,j}^t\right)$
2: Update parameters $\bar{\mathbf{x}}^{t+1} \leftarrow \bar{\mathbf{x}}^t - \eta \bar{\mathbf{g}}^{t-1}(\tau)$
3: Non-Blocking AllReduce $\bar{\mathbf{g}}^t(\tau) \leftarrow \frac{1}{N}\sum_{i=1}^{N} \mathbf{g}_i^t(\tau)$

In this section we describe our method LAGA, which leverages gradient accumulation to further hide the communication overheads. LAGA introduces a *lag* into the AllReduce operation by overlapping the next micro-batches computations with the previous AllReduce operation. Figure 2c illustrates the inner mechanics of LAGA, where the communications (AllReduce operations) are executed in the background while the worker continues to compute the next micro-batches.

Algorithm 3 describes the LAGA-SGD variation in more details. Unlike SSGD (Algorithm 1), where the parameters $\bar{\mathbf{x}}^t$ are updated by $\bar{\mathbf{g}}^t$, in LAGA-SGD these parameters are updated with the previous average gradient $\bar{\mathbf{g}}^{t-1}(\tau)$. Notice that the computations of $\mathbf{g}_i^t(\tau)$ are independent of the communication synchronization of $\bar{\mathbf{g}}^t(\tau) = \frac{1}{N}\sum_{i=1}^{N} \mathbf{g}_i^t(\tau)$ and therefore the computations and communications are executed in parallel.

4.1. The Importance of Gradient Accumulation

The communication overhead ψ of LAGA is different than that of SSGD and SSGD-OPT since the gradient synchronization is executed in the background of the computations of multiple micro-batches when $\tau > 1$. Figure 2c illustrates this important attribute of LAGA, where the idle time is completely eliminated and therefore the computations are executed without any interruptions.

$$\psi_{LAGA}(\tau) = \begin{cases} 0 & T_{comm} \leq \tau \cdot T_{comp} \\ \frac{T_{comm} - \tau \cdot T_{comp}}{\tau \cdot T_{comp} + T_{comm}} & \text{Otherwise.} \end{cases} \quad (6)$$

Equation (6) shows that the communication overhead of LAGA $\psi_{LAGA}(\tau)$ can be reduced by increasing τ. Given a large enough τ that satisfies the inequality $T_{comm} \leq \tau \cdot T_{comp}$, the communication overhead of LAGA are completely eliminated and equal to zero. Therefore, LAGA can hide all communication overheads and achieve linear scalability given a respective τ that holds $\tau \geq \frac{T_{comm}}{T_{comp}}$. We define τ_{LAGA} as the optimal (minimal) number of accumulation that LAGA requires to achieve linear scalability, $\tau_{LAGA} = \lceil \frac{T_{comm}}{T_{comp}} \rceil$. Increasing the amount of gradient accumulations beyond

τ_{LAGA} would not gain any speedups as LAGA has already reached linear scalability.

Next we analyse the theoretical communication savings of LAGA compared to SSGD, where we compare $\psi_{LAGA}(\tau)$ and $\psi_{SSGD}(\tau)$ respectively when the communication time T_{comm} is high.

$$\psi_{SSGD}(\tau) - \psi_{LAGA}(\tau) \leq \frac{\tau \cdot T_{comp}}{\tau \cdot T_{comp} + T_{comm}} \quad (7)$$

Equation (7) shows that the communication savings of LAGA increase as τ grows, unlike the communication savings of SSGD-OPT which diminish. This essential property is what helps LAGA achieve linear scalability, even in extreme cases where $T_{comm} \gg T_{comp}$.

4.2. Theoretical Convergence Proof of LAGA

In this section, we prove the convergence rate of LAGA-SGD and find that it achieves the same convergence rate as SSGD [13].

Assumption 1.

1) **Smoothness:** *Each function $f_i(\mathbf{x})$ is L-smooth.*
2) **Bounded variance and second moment:** *There exits constants $\sigma > 0$ and $G > 0$ such that*

$$\mathbb{E}_{\zeta_i \sim \mathcal{D}_i} \|\nabla F_i(\mathbf{x}; \zeta_i) - \nabla f_i(\mathbf{x})\|^2 \leq \sigma^2, \forall \mathbf{x}, \forall i$$
$$\mathbb{E}_{\zeta_i \sim \mathcal{D}_i} \|\nabla F_i(\mathbf{x}; \zeta_i)\|^2 \leq G^2, \forall \mathbf{x}, \forall i$$

Theorem 1. *Consider Algorithm 3 with the above assumptions. Then, for $0 < \eta \leq \frac{1}{L}$ Algorithm 3 achieves*

$$\frac{1}{T}\sum_{t=1}^{T} \mathbb{E}\left[\|\nabla f(\bar{\mathbf{x}}^{t-1})\|^2\right] \leq \frac{2\left(f(\bar{\mathbf{x}}^0) - f^*\right)}{\eta T} + \frac{\eta^2 L^2 G^2}{4} + \frac{L\eta\sigma^2}{N}$$

Corollary 1. *Using $\eta = \frac{\sqrt{N}}{L\sqrt{T}}$ yields,*

$$\frac{1}{T}\sum_{t=1}^{T} \mathbb{E}\left[\|\nabla f(\bar{\mathbf{x}}^{t-1})\|^2\right] \leq \frac{2L}{\sqrt{NT}}\left(f(\bar{\mathbf{x}}^0) - f^*\right) + \frac{NG^2}{4T} + \frac{\sigma^2}{\sqrt{NT}} . \quad (8)$$

Theorem 1 and Corollary 1 imply that when T is large enough, i.e., $T > N^3$, Algorithm 3 has a convergence rate of $O\left(\frac{1}{\sqrt{NT}}\right)$. Namely, it achieves a linear speed-up with respect to the number of workers. In practice, T is usually much larger than N^3.

The proof of Theorem 1 is presented below.

Proof. The proof follows similar steps as the proof of Theorem 1 in [55]. From the smoothness of f,

$$\mathbb{E}[f(\bar{\mathbf{x}}^t)] \leq \mathbb{E}[f(\bar{\mathbf{x}}^{t-1})] + \underbrace{\mathbb{E}\left[\langle \nabla f(\bar{\mathbf{x}}^{t-1}), \bar{\mathbf{x}}^t - \bar{\mathbf{x}}^{t-1}\rangle\right]}_{\text{(B)}} + \frac{L}{2}\underbrace{\mathbb{E}[\|\bar{\mathbf{x}}^t - \bar{\mathbf{x}}^{t-1}\|^2]}_{\text{(A)}} .$$

Bounding (A). Using the relations described in Algorithm 3 we have,

$$\begin{aligned}
\text{(A)} &= \eta^2 \mathbb{E}\left[\left\|\frac{1}{N}\sum_{i=1}^{N} \mathbf{g}_i^{t-2}(\tau)\right\|^2\right] \\
&= \eta^2 \mathbb{E}\left[\left\|\frac{1}{N}\sum_{i=1}^{N} \mathbf{g}_i^{t-2}(\tau) - \nabla f_i\left(\bar{\mathbf{x}}^{t-2}\right)\right\|^2\right] \\
&+ \eta^2 \mathbb{E}\left[\left\|\frac{1}{N}\sum_{i=1}^{N} \nabla f_i\left(\bar{\mathbf{x}}^{t-2}\right)\right\|^2\right] \\
&= \frac{\eta^2}{N^2}\sum_{i=1}^{N} \mathbb{E}\left[\left\|\mathbf{g}_i^{t-2}(\tau) - \nabla f_i\left(\bar{\mathbf{x}}^{t-2}\right)\right\|^2\right] \\
&+ \eta^2 \mathbb{E}\left[\left\|\frac{1}{N}\sum_{i=1}^{N} \nabla f_i\left(\bar{\mathbf{x}}^{t-2}\right)\right\|^2\right] \\
&\leq \frac{\eta^2\sigma^2}{N} + \eta^2 \mathbb{E}\left[\left\|\frac{1}{N}\sum_{i=1}^{N} \nabla f_i\left(\bar{\mathbf{x}}^{t-2}\right)\right\|^2\right] ,
\end{aligned}$$

where in the second equality we used the known inequality $\mathbb{E}[\|\mathbf{z}\|^2] = \mathbb{E}[\|\mathbf{z} - \mathbb{E}[\mathbf{z}]\|^2] + \|\mathbb{E}[\mathbf{z}]\|^2$ and the third follows from $\mathbf{g}_i^{t-2}(\tau) - \nabla f_i\left(\bar{\mathbf{x}}^{t-2}\right)$ having zero mean and is independent across workers.

Bounding (B). Using the relations described in Algorithm 3 we have,

$$\begin{aligned}
\text{(B)} &= -\eta \mathbb{E}\left[\langle \nabla f(\bar{\mathbf{x}}^{t-1}), \bar{\mathbf{g}}^{t-2}(\tau)\rangle\right] \\
&= -\eta \mathbb{E}\left[\langle \nabla f(\bar{\mathbf{x}}^{t-1}), \frac{1}{N}\sum_{i=1}^{N} \nabla f_i\left(\bar{\mathbf{x}}^{t-2}\right)\rangle\right] \\
&= -\frac{\eta}{2}\mathbb{E}\left[\|\nabla f(\bar{\mathbf{x}}^{t-1})\|^2 + \left\|\frac{1}{N}\sum_{i=1}^{N} \nabla f_i\left(\bar{\mathbf{x}}^{t-2}\right)\right\|^2\right] \\
&+ \underbrace{\frac{\eta}{2}\mathbb{E}\left[\left\|\nabla f(\bar{\mathbf{x}}^{t-1}) - \frac{1}{N}\sum_{i=1}^{N} \nabla f_i\left(\bar{\mathbf{x}}^{t-2}\right)\right\|\right]}_{\text{(C)}} ,
\end{aligned}$$

where in the third equality we used the identity $\langle \mathbf{z}_1, \mathbf{z}_2\rangle = \frac{1}{2}(\|\mathbf{z}_1\|^2 + \|\mathbf{z}_2\|^2 - \|\mathbf{z}_1 - \mathbf{z}_2\|^2)$, and the second equality follows from,

$$\begin{aligned}
&\mathbb{E}[\langle \nabla f(\bar{\mathbf{x}}^{t-1}), \bar{\mathbf{g}}^{t-2}(\tau)\rangle] = \\
&= \mathbb{E}\left[\mathbb{E}\left[\langle \nabla f(\bar{\mathbf{x}}^{t-1}), \frac{1}{N}\sum_{i=1}^{N} \mathbf{g}_i^{t-2}(\tau)\rangle | \boldsymbol{\xi}^{[t-3]}\right]\right] \\
&= \mathbb{E}\left[\langle \nabla f(\bar{\mathbf{x}}^{t-1}), \frac{1}{N}\sum_{i=1}^{N} \mathbb{E}\left[\mathbf{g}_i^{t-2}(\tau) | \boldsymbol{\xi}^{[t-3]}\right]\rangle\right] \\
&= \mathbb{E}\left[\langle \nabla f(\bar{\mathbf{x}}^{t-1}), \frac{1}{N}\sum_{i=1}^{N} \nabla f_i\left(\bar{\mathbf{x}}^{t-2}\right)\rangle\right] ,
\end{aligned}$$

where the second line follows the law of total expectation and the third follows from $\bar{\mathbf{x}}^{t-1}$ being determined by $\boldsymbol{\xi}^{[t-3]}$ and linearity of expectation.

Bounding (C). By the definition of $f(\mathbf{x}) = \sum_{i=1}^{N} f_i(\mathbf{x})$ we have,

$$
\begin{aligned}
\text{(C)} &= \frac{1}{N^2}\mathbb{E}\left[\left\|\sum_{i=1}^{N}\nabla f_i(\bar{\mathbf{x}}^{t-1}) - \nabla f_i\left(\bar{\mathbf{x}}^{t-2}\right)\right\|^2\right] \\
&\leq \frac{1}{N}\mathbb{E}\left[\sum_{i=1}^{N}\left\|\nabla f_i(\bar{\mathbf{x}}^{t-1}) - \nabla f_i\left(\bar{\mathbf{x}}^{t-2}\right)\right\|^2\right] \\
&\leq \frac{L^2}{N}\sum_{i=1}^{N}\mathbb{E}\left[\left\|\bar{\mathbf{x}}^{t-1} - \bar{\mathbf{x}}^{t-2}\right\|^2\right] \\
&= L^2\mathbb{E}\left[\left\|\eta\bar{\mathbf{g}}^{t-3}(\tau)\right\|^2\right] = L^2\eta^2\mathbb{E}\left[\left\|\frac{1}{N}\sum_{i=1}^{N}\mathbf{g}_i^{t-3}(\tau)\right\|^2\right] \\
&\leq L^2\eta^2\mathbb{E}\left[\frac{1}{N}\sum_{i=1}^{N}\left\|\mathbf{g}_i^{t-3}(\tau)\right\|^2\right] \leq L^2\eta^2G^2 \ ,
\end{aligned}
$$

where the second and sixth lines are by using $\left\|\sum_{i=1}^{N}\mathbf{z}_i\right\|^2 \leq N\sum_{i=1}^{N}\|\mathbf{z}_i\|^2$. The third line is due to the smoothness of each f_i, and the forth and fifth are by the definition of $\bar{\mathbf{x}}^t$ and $\bar{\mathbf{g}}$.

Combining the above bounds implies,

$$
\begin{aligned}
\mathbb{E}[f(\bar{\mathbf{x}}^t)] &\leq \mathbb{E}[f(\bar{\mathbf{x}}^{t-1})] - \frac{\eta}{2}\mathbb{E}\left[\|\nabla f(\bar{\mathbf{x}}^{t-1})\|^2\right] + \frac{\eta^3L^2G^2}{2} \\
&+ \frac{L\eta^2\sigma^2}{2N} - \frac{\eta-\eta^2L}{2}\mathbb{E}\left[\left\|\frac{1}{N}\sum_{i=1}^{N}\nabla f_i\left(\bar{\mathbf{x}}^{t-2}\right)\right\|^2\right] \\
&\leq \mathbb{E}[f(\bar{\mathbf{x}}^{t-1})] - \frac{\eta}{2}\mathbb{E}\left[\|\nabla f(\bar{\mathbf{x}}^{t-1})\|^2\right] + \frac{\eta^3L^2G^2}{2} + \frac{L\eta^2\sigma^2}{2N} \ ,
\end{aligned}
$$

where the second inequality follows from $0 < \eta \leq \frac{1}{L}$.

Rearranging and dividing by $\frac{\eta}{2}$ yields,

$$
\begin{aligned}
\mathbb{E}\left[\|\nabla f(\bar{\mathbf{x}}^{t-1})\|^2\right] \leq& \frac{2}{\eta}\left(\mathbb{E}[f(\bar{\mathbf{x}}^{t-1})] - \mathbb{E}[f(\bar{\mathbf{x}}^t)]\right) \\
&+ \frac{\eta^2L^2G^2}{4} + \frac{L\eta\sigma^2}{N} \ .
\end{aligned}
$$

Summing over T and dividing by T gives

$$
\begin{aligned}
\frac{1}{T}\sum_{t=1}^{T}\mathbb{E}\left[\|\nabla f(\bar{\mathbf{x}}^{t-1})\|^2\right] &\leq \frac{2}{\eta T}\left(\mathbb{E}[f(\bar{\mathbf{x}}^0)] - \mathbb{E}[f(\bar{\mathbf{x}}^T)]\right) \\
&+ \frac{\eta^2L^2G^2}{4} + \frac{L\eta\sigma^2}{4N} \\
&\leq \frac{2}{\eta T}\left(f(\bar{\mathbf{x}}^0) - f^*\right) + \frac{\eta^2L^2G^2}{4} + \frac{L\eta\sigma^2}{N} \ .
\end{aligned}
$$

where the last inequality uses f^* being the minimum of the minimization problem. □

4.3. LAGA-SGD with Momentum

Momentum [38] is a widely adopted SGD variation that has been demonstrated to accelerate SGD convergence and reduce oscillation [48]. Previous empirical works [43] [10] suggest that momentum is crucial for achieving high final test accuracy when training on large mini-batch sizes. Mathematically, momentum is similar to an exponentially-weighted moving average of past gradients. We denote the momentum vector as $\bar{\mathbf{m}}^t = \gamma\bar{\mathbf{m}}^{t-1} + \bar{\mathbf{g}}^{t-1}(\tau)$, where γ is the momentum coefficient.

Algorithm 4 LAGA-SGDM

1: Compute gradient $\mathbf{g}_i^t(\tau) \leftarrow \frac{1}{\tau}\sum_{j=1}^{\tau}\nabla F_i\left(\bar{\mathbf{x}}^t;\xi_{i,j}^t\right)$
2: Update momentum $\bar{\mathbf{m}}^t \leftarrow \gamma\bar{\mathbf{m}}^{t-1} + \bar{\mathbf{g}}^{t-1}(\tau)$
3: Update parameters $\bar{\mathbf{x}}^{t+1} \leftarrow \bar{\mathbf{x}}^t - \eta\bar{\mathbf{m}}^t$
4: Non-Blocking AllReduce $\bar{\mathbf{g}}^t(\tau) \leftarrow \frac{1}{N}\sum_{i=1}^{N}\mathbf{g}_i^t(\tau)$

Algorithm 4 describes the LAGA-SGDM (LAGA with SGD-Momentum) algorithm in details. Notice that $\bar{\mathbf{m}}$ is equal across all workers since it is updated with the same average gradient $\bar{\mathbf{g}}$ and initialized to zero.

Recent works [33] [18] show that *lag* and momentum do not work well together. In asynchronous settings, where the *lag* is more significant, training with momentum reduces the convergences speed and might cause complete divergence [59] [7].

To remedy, we start by observing that the *lag* causes LAGA to compute the gradient on $\bar{\mathbf{x}}^t$ but rather apply it on $\bar{\mathbf{x}}^{t+1}$. [34] proposed Nesterov Accelerated Gradient (NAG), which computes the gradient after taking into account the momentum vector trajectory. In NAG, the gradient is computed on $\mathbf{y} = \mathbf{x} - \eta\gamma\bar{\mathbf{m}}^{t-1}$ but applied on $\mathbf{x}$, where the former already includes the momentum vector impact on the next update step. [8] reparameterized NAG so the gradient $\nabla F(\mathbf{y};\xi)$ is both computed and applied on the same set of parameters $\mathbf{y}$, but still maintain the same benefits of NAG. We apply the same methodology of [8] to LAGA-SGDM and define $\bar{\mathbf{y}}^t \triangleq \bar{\mathbf{x}}^t - \eta\gamma\bar{\mathbf{m}}^{t-1}$.

$$
\begin{aligned}
\bar{\mathbf{y}}^{t+1} &= \bar{\mathbf{y}}^t + \eta\gamma\bar{\mathbf{m}}^{t-1} - \eta\gamma\bar{\mathbf{m}}^t - \eta\bar{\mathbf{m}}^t \\
&= \bar{\mathbf{y}}^t + \eta\gamma\bar{\mathbf{m}}^{t-1} - \eta\gamma\bar{\mathbf{m}}^t - \eta\gamma\bar{\mathbf{m}}^{t-1} - \eta\bar{\mathbf{g}}^{t-1}(\tau) \\
&= \bar{\mathbf{y}}^t - \eta\left(\gamma\bar{\mathbf{m}}^t + \bar{\mathbf{g}}^{t-1}(\tau)\right)
\end{aligned} \tag{9}
$$

Equation (9) shows the reparameterized update step of LAGA-SGDM. We name this variation as LAGA-SGDN (LAGA with SGD-NAG).

Algorithm 5 LAGA-SGDN

1: Compute gradient $\mathbf{g}_i^t(\tau) \leftarrow \frac{1}{\tau}\sum_{j=1}^{\tau}\nabla F_i\left(\bar{\mathbf{x}}^t;\xi_{i,j}^t\right)$
2: Update momentum $\bar{\mathbf{m}}^t \leftarrow \gamma\bar{\mathbf{m}}^{t-1} + \bar{\mathbf{g}}^{t-1}(\tau)$
3: Update parameters $\bar{\mathbf{y}}^{t+1} \leftarrow \bar{\mathbf{y}}^t - \eta\left(\gamma\bar{\mathbf{m}}^t + \bar{\mathbf{g}}^{t-1}(\tau)\right)$
4: Non-Blocking AllReduce $\bar{\mathbf{g}}^t(\tau) \leftarrow \frac{1}{N}\sum_{i=1}^{N}\mathbf{g}_i^t(\tau)$

Algorithm 5 describes in details the LAGA-SGDN algorithm. The model parameter vector $\bar{\mathbf{y}}$ is used for both

computing the gradient and applying it, but it maintains the same essential properties to that of NAG. In the next section we present empirical results which show the benefits of LAGA-SGDN in final accuracy and convergence speed.

5. Experiments

In this section we conduct an empirical evaluation of LAGA, where we analyse the training speed of LAGA as well as its convergence rate and final accuracy. Our evaluation consists of the following algorithms:

- SSGD: The synchronization is executed right after the computations of the backpropagation.
- SSGD-OPT: The synchronization is overlapped with the computations of the backpropagation [29].
- LAGA: The synchronization is executed with a non-blocking call in the background.

We conduct our experiments on a system with eight NVIDIA® GeForce® RTX 2080 Ti GPUs [2] that each have 11GB of internal memory. We run our code on the PyTorch [36] framework, which is a highly popular and open-sourced deep learning framework for training neural networks. All algorithms utilize the efficient NCCL [23] for fast and optimized AllReduce operations. For consistency and reproducibility, we run all of our experiments on the exact same publicly available Docker [32] image (provided by Horovod [42]). This simplifies the reproducibility of our software setup.

We provide an open-source implementation of LAGA written in both PyTorch and Horovod [42] frameworks. Horovod is a highly popular open-sourced distributed neural network framework which is heavily used by the machine learning industry. Providing an implementation for both frameworks expands the compatibility of LAGA for future research directions and adaptations. We note that although LAGA is implemented with an AllReduce operation, it is also fully compatible in the parameter-server setting which has recently received several efficient implementations [60] [11] [25] [50] [37] [28].

5.1. Communication Efficiency

In this section we focus on the training speed of the different algorithms. We evaluate on a wide range of neural architectures which provide interesting insights about their properties. For accurate and comparable measurements, each experiment in this section was computed 5000 times, which we present with both the mean and the 95% confidence interval (marked as the black lines on-top of the bar plots).

We first analyse the idle time. Figure 4 shows the idle time of the different algorithm on various neural architecture and gradient accumulation settings. On Wide ResNet-101-2 [57] with $\tau = 1$, LAGA has 3.89x less idle time than SSGD and 2.67x less than SSGD-OPT. Increasing τ on communication intensive neural architecture, such as VGG-16, results in even higher gains for LAGA, where in the VGG-16 neural architecture with $\tau = 4$, LAGA has 5.24x and 4.54x fewer idle time compared to SSGD and SSGD-OPT respectively. Finally, on the highly popular and communication friendly ResNet-50 neural architecture with $\tau = 1$ (as demonstrated in Figure 3), LAGA reduces the idle time by 2.3x and 1.45x compared to SSGD and SSGD-OPT respectively. We note that on newer and faster computational devices (with the same communication bandwidth) the idle time reduction would be even higher.

Next, we analyse the *scalability* (also known as *scaling efficiency*), which we define as the wall-clock-time ratio to that of a linear scalable algorithm. Intuitively, a communication efficient algorithm would have a scalability of close to one. Figure 5 shows the scalability of the algorithms on a wide variety of different neural architectures. Figure 5b shows the scalability when training with $\tau = 2$, where we notice that LAGA achieves near linear scalability on certain neural architectures and outperforms both SSGD and SSGD-OPT. Neural architectures with a high T_{comm}, such as VGG-19 [46] (Figure 3), are difficult to scale; however, LAGA manages to achieve near linear scalability whereas SSGD and SSGD-OPT achieve a significantly lower scalability. Notice that the scalability of all algorithms improves as we increase τ, which empirically justifies our theoretical analyse in Section 3. These results are in correspondence to Figure 4, lowering the idle time results in higher scalability.

Figure 6 shows the speedup gains of LAGA on different τ compared to SSGD without accumulations. As τ increases the speedup gains of LAGA grow as well, reaching a speedup of up to 2.96x. Notice that when $\tau = 4$ the speedup gains of LAGA start to saturates since LAGA is near linear scalability and the amount of gradient accumulations is already close to τ_{LAGA}. We note that neural architectures with a higher communication overhead have a higher τ_{LAGA} and therefore enjoy speedup gains on even larger amount of gradient accumulations.

5.2. Final Accuracy and Convergence Rate

In the previous section we saw that LAGA trains faster than SSGD alternatives and scales better when joint with gradient accumulation. In this section, we focus on the convergence rate and final accuracy with respect to epochs rather than time, where synchronous algorithms usually dominate thanks to their fast convergence speed.

Our accuracy evaluations focus on the ResNet-50 [19] neural architecture on the ImageNet [40] dataset, which is a highly popular benchmark in image classification [4] [54].

The training follows the same schedule and hyperparameters as [17], which we detail below. Each worker computes the gradient on a batch size of 32 and the batches are shuffled after each epoch. The initial learning-rate is 0.1 (when $\tau = 1$) and is scaled linearly with respect to τ. Furthermore, the learning-rate is gradually warmed-up during the early iterations (first 5 epochs) and is decayed by 0.1 at epochs 30, 60, and 80. The training is spanned across 90 epochs in total, with a momentum coefficient of 0.9 and a

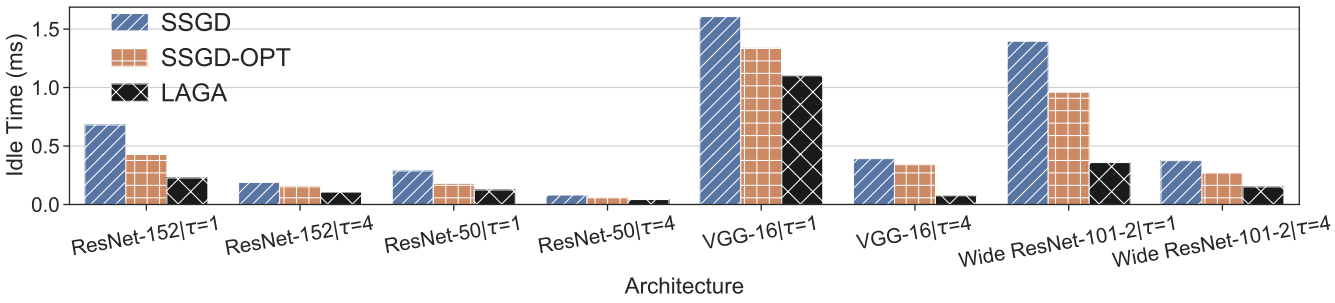
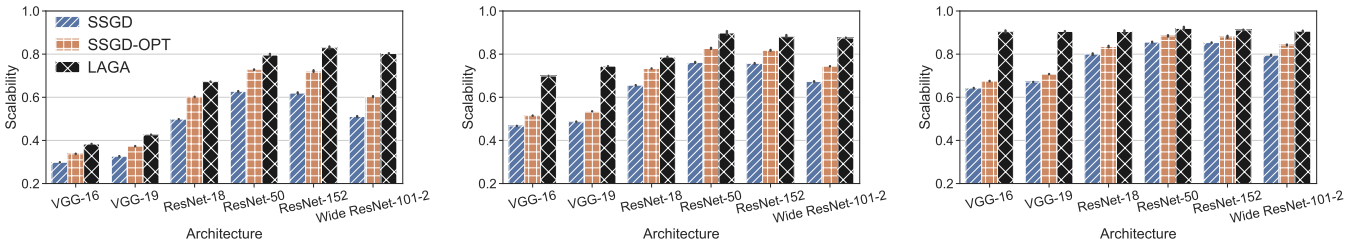


Figure 4: The idle time (ms) due to communication overhead. Increasing τ reduces the idle time for all algorithms and neural architectures, where LAGA is the least idle in all the settings.



(a) $\tau = 1$ (b) $\tau = 2$ (c) $\tau = 4$

Figure 5: A scalability comparison of the different algorithms.

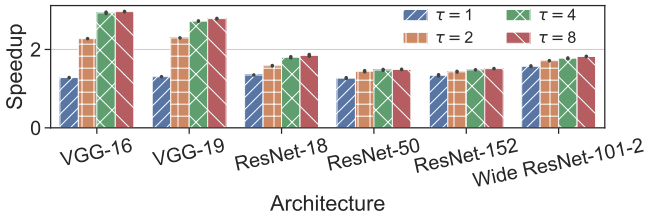


Figure 6: Speedup of LAGA.

TABLE 1: ResNet-50 ImageNet Final Test Accuracy

#Accumulations	SSGD	LAGA-SGDM	LAGA-SGDN
1	76.4%	76.44%	76.3%
2	76.24%	75.95%	76.28%
4	76.23%	75.63%	76.28%

weight-decay coefficient of $1e-4$. The training code will be publicly available in our published code repository.

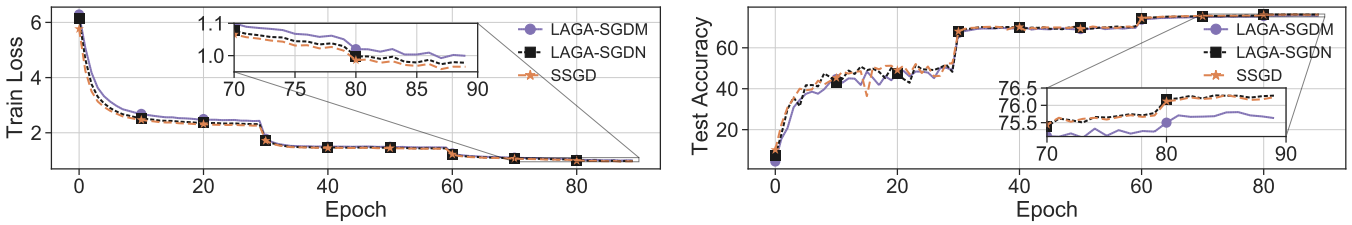
Figure 7a compares the training convergence. LAGA-SGDM converges slightly slower than SSGD due to the *lag* which hurts the accuracy of the gradient. LAGA-SGDN remedies this issue and therefore converges at almost the same rate as SSGD. Correspondingly, Figure 7b compares the test accuracy convergence, where once again, LAGA-SGDM falls short of SSGD and LAGA-SGDN. This shows the importance of NAG when training with LAGA. We note that additional techniques for mitigating the *lag*, such as [61], are compatible with LAGA.

Table 1 details the final test accuracy with different gradient accumulations. When $\tau = 1$ LAGA-SGDM is on par with SSGD and LAGA-SGDN, and even slightly outperforms them. However, when trained with more gradient accumulations, the final accuracy of LAGA-SGDM starts to deteriorate. Conversely, SSGD and LAGA-SGDN maintain high final accuracy even when τ grows. Maintaining high accuracy with large values of τ is important since increasing τ improves the scalability (Figure 5).

6. Conclusions

In this paper we presented LAGA, a novel and efficient algorithm that leverages gradient accumulation to further hide the communication overheads. We conducted a thorough study on a variety of neural architectures and showed that LAGA achieves near linear scaling with a speedup of up to 2.96x and 5.24x less idle time on the ImageNet dataset. We provide an open-source implementation of LAGA in both PyTorch and Horovod frameworks for a wide-range compatibility. We showed that LAGA can achieve the same final accuracy and convergence speed as SSGD. Finally, we provided a theoretical convergence proof for LAGA, which achieves the same theoretical convergence rate to that SSGD.

The scaling of large neural networks, such as Transformer based architectures [15], is an ongoing challenge [25] due to high communication to computation ratio (large ψ). In such cases, LAGA would benefit high speedup gains. Future works that combine LAGA with communication reduction algorithms [52] [41] [53] can further reduce the communication overhead.



(a) Train Loss Convergence

(b) Test Accuracy Convergence

Figure 7: Results of ResNet-50 on ImageNet where $\tau = 4$.

Acknowledgements

The work on this paper was supported in part by the Israeli Ministry of Science, Technology, and Space and by The Hasso Plattner Institute at the Technion. K.Y. Levy acknowledges support from the Israel Science Foundation (grant No. 447/20).

References

[1] Nvidia apex. *GitHub. Note: https://github.com/NVIDIA/apex*.

[2] Nvidia® geforce® rtx 2080 ti. *https://www.nvidia.com/en-eu/geforce/graphics-cards/rtx-2080-ti/*.

[3] A. F. Aji and K. Heafield. Making asynchronous stochastic gradient descent work for transformers. In *Proceedings of the 3rd Workshop on Neural Generation and Translation*, 2019.

[4] T. Akiba, S. Suzuki, and K. Fukuda. Extremely large minibatch sgd: Training resnet-50 on imagenet in 15 minutes. *arXiv preprint arXiv:1711.04325*, 2017.

[5] S. Alqahtani and M. Demirbas. Performance analysis and comparison of distributed machine learning systems. *arXiv preprint arXiv:1909.02061*, 2019.

[6] A. A. Awan, C.-H. Chu, H. Subramoni, and D. K. Panda. Optimized broadcast for deep learning workloads on dense-gpu infiniband clusters: Mpi or nccl? In *Proceedings of the 25th European MPI Users' Group Meeting*, pages 1–9, 2018.

[7] S. Barkai, I. Hakimi, and A. Schuster. Gap-aware mitigation of gradient staleness. In *International Conference on Learning Representations*, 2020.

[8] Y. Bengio, N. Boulanger-Lewandowski, and R. Pascanu. Advances in optimizing recurrent networks. In *IEEE International Conference on Acoustics, Speech and Signal Processing*, 2013.

[9] K. S. Chahal, M. S. Grover, K. Dey, and R. R. Shah. A hitchhiker's guide on distributed training of deep neural networks. *Journal of Parallel and Distributed Computing*, 2020.

[10] K. Chen and Q. Huo. Scalable training of deep learning machines by incremental block training with intra-block parallel optimization and blockwise model-update filtering. In *2016 ieee international conference on acoustics, speech and signal processing (icassp)*, pages 5880–5884. IEEE, 2016.

[11] Y. Chen, Y. Peng, Y. Bao, C. Wu, Y. Zhu, and C. Guo. Elastic parameter server load distribution in deep learning clusters. SoCC '20, New York, NY, USA, 2020. Association for Computing Machinery.

[12] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, M. a. Ranzato, A. Senior, P. Tucker, K. Yang, Q. Le, and A. Ng. Large scale distributed deep networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012.

[13] O. Dekel, R. Gilad-Bachrach, O. Shamir, and L. Xiao. Optimal distributed online prediction using mini-batches. *Journal of Machine Learning Research*, 2012.

[14] W. Falcon. Pytorch lightning. *GitHub. Note: https://github.com/PyTorchLightning/pytorch-lightning*, 2019.

[15] W. Fedus, B. Zoph, and N. Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *arXiv preprint arXiv:2101.03961*, 2021.

[16] E. Gebremeskel. Analysis and comparison of distributed training techniques for deep neural networks in a dynamic environment, 2018.

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

[18] I. Hakimi, S. Barkai, M. Gabel, and A. Schuster. Taming momentum in a distributed asynchronous environment. *arXiv preprint arXiv:1907.11612*, 2019.

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

[20] J. R. Hermans, G. Spanakis, and R. Möckel. Accumulated gradient normalization. In *Asian Conference on Machine Learning*, pages 439–454. PMLR, 2017.

[21] E. Hoffer, I. Hubara, and D. Soudry. Train longer, generalize better: closing the generalization gap in large batch training of neural networks. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[22] Y. Huang, Y. Cheng, A. Bapna, O. Firat, D. Chen, M. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu, et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. In *Advances in Neural Information Processing Systems*, 2019.

[23] S. Jeaugey. Optimized inter-gpu collective operations with nccl 2, 2017.

[24] X. Jia, S. Song, W. He, Y. Wang, H. Rong, F. Zhou, L. Xie, Z. Guo, Y. Yang, L. Yu, et al. Highly scalable deep learning training system with mixed-precision: Training imagenet in four minutes. *arXiv preprint arXiv:1807.11205*, 2018.

[25] Y. Jiang, Y. Zhu, C. Lan, B. Yi, Y. Cui, and C. Guo. A unified architecture for accelerating distributed DNN training in heterogeneous gpu/cpu clusters. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 2020.

[26] N. S. Keskar, D. Mudigere, J. Nocedal, M. Smelyanskiy, and P. T. P. Tang. On large-batch training for deep learning: Generalization gap and sharp minima. *arXiv preprint arXiv:1609.04836*, 2016.

[27] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

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

[29] S. Li, Y. Zhao, R. Varma, O. Salpekar, P. Noordhuis, T. Li, A. Paszke, J. Smith, B. Vaughan, P. Damania, and S. Chintala. Pytorch distributed: Experiences on accelerating data parallel training. *VLDB Endow.*, 2020.

[30] Y. Lin, S. Han, H. Mao, Y. Wang, and B. Dally. Deep gradient compression: Reducing the communication bandwidth for distributed training. In *International Conference on Learning Representations*, 2018.

[31] I. Loshchilov and F. Hutter. Decoupled weight decay regularization. In *International Conference on Learning Representations*, 2019.

[32] D. Merkel. Docker: Lightweight linux containers for consistent development and deployment. *Linux J.*, 2014(239), Mar. 2014.

[33] I. Mitliagkas, C. Zhang, S. Hadjis, and C. Ré. Asynchrony begets momentum, with an application to deep learning. In *2016 54th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, pages 997–1004. IEEE, 2016.

[34] Y. Nesterov. A method for unconstrained convex minimization problem with the rate of convergence o (1/kˆ 2). In *Doklady AN USSR*, volume 269, pages 543–547, 1983.

[35] D. K. Panda, A. A. Awan, and H. Subramoni. High performance distributed deep learning: a beginner's guide. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, pages 452–454, 2019.

[36] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. In *Advances in neural information processing systems*, 2019.

[37] Y. Peng, Y. Zhu, Y. Chen, Y. Bao, B. Yi, C. Lan, C. Wu, and C. Guo. A generic communication scheduler for distributed dnn training acceleration. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 16–29, New York, NY, USA, 2019. Association for Computing Machinery.

[38] B. Polyak. Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics and Mathematical Physics*, 1964.

[39] S. Rajbhandari, J. Rasley, O. Ruwase, and Y. He. Zero: Memory optimizations toward training trillion parameter models. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '20. IEEE Press, 2020.

[40] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. S. Bernstein, A. C. Berg, and F. Li. ImageNet large scale visual recognition challenge. *International Journal of Computer Vision*, 2015.

[41] A. Sapio, M. Canini, C.-Y. Ho, J. Nelson, P. Kalnis, C. Kim, A. Krishnamurthy, M. Moshref, D. Ports, and P. Richtarik. Scaling distributed machine learning with in-network aggregation. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 785–808. USENIX Association, Apr. 2021.

[42] A. Sergeev and M. Del Balso. Horovod: fast and easy distributed deep learning in tensorflow. *arXiv preprint arXiv:1802.05799*, 2018.

[43] C. J. Shallue, J. Lee, J. Antognini, J. Sohl-Dickstein, R. Frostig, and G. E. Dahl. Measuring the effects of data parallelism on neural network training. *Journal of Machine Learning Research*, 20(112):1–49, 2019.

[44] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro. Megatron-lm: Training multi-billion parameter language models using gpu model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

[45] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, et al. Mastering the game of go without human knowledge. *nature*, 550(7676):354–359, 2017.

[46] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. In Y. Bengio and Y. LeCun, editors, *3rd International Conference on Learning Representations*, 2015.

[47] S. L. Smith, P.-J. Kindermans, and Q. V. Le. Don't decay the learning rate, increase the batch size. In *International Conference on Learning Representations*, 2018.

[48] I. Sutskever, J. Martens, G. E. Dahl, and G. E. Hinton. On the importance of initialization and momentum in deep learning. In *Proceedings of the 30th International Conference on Machine Learning*, 2013.

[49] K. Tanaka, Y. Arikawa, T. Ito, K. Morita, N. Nemoto, F. Miura, K. Terada, J. Teramoto, and T. Sakamoto. Communication-efficient distributed deep learning with gpu-fpga heterogeneous computing. In *2020 IEEE Symposium on High-Performance Interconnects (HOTI)*, pages 43–46. IEEE, 2020.

[50] I. Thangakrishnan, D. Cavdar, C. Karakus, P. Ghai, Y. Selivonchyk, and C. Pruce. Herring: Rethinking the parameter server at scale for the cloud. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '20. IEEE Press, 2020.

[51] J. Verbraeken, M. Wolting, J. Katzy, J. Kloppenburg, T. Verbelen, and J. S. Rellermeyer. A survey on distributed machine learning. *ACM Computing Surveys (CSUR)*, 53(2):1–33, 2020.

[52] T. Vogels, S. P. Karimireddy, and M. Jaggi. Powersgd: Practical low-rank gradient compression for distributed optimization. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

[53] H. Xu, C.-Y. Ho, A. M. Abdelmoniem, A. Dutta, E. Bergou, K. Karatsenidis, M. Canini, and P. Kalnis. Grace: A compressed communication framework for distributed machine learning. In *Proc. of 41st IEEE Int. Conf. Distributed Computing Systems (ICDCS)*, 2021.

[54] Y. You, Z. Zhang, J. Demmel, K. Keutzer, and C.-J. Hsieh. Imagenet training in 24 minutes. *arXiv preprint arXiv:1709.05011*, 2017.

[55] H. Yu, S. Yang, and S. Zhu. Parallel restarted sgd with faster convergence and less communication: Demystifying why model averaging works for deep learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 2019.

[56] K. Yu, T. Flynn, S. Yoo, and N. D'Imperio. Layered sgd: A decentralized and synchronous sgd algorithm for scalable deep neural network training. *arXiv preprint arXiv:1906.05936*, 2019.

[57] S. Zagoruyko and N. Komodakis. Wide residual networks. In E. R. H. Richard C. Wilson and W. A. P. Smith, editors, *Proceedings of the British Machine Vision Conference (BMVC)*, pages 87.1–87.12. BMVA Press, September 2016.

[58] W. Zeng, X. Ren, T. Su, H. Wang, Y. Liao, Z. Wang, X. Jiang, Z. Yang, K. Wang, X. Zhang, et al. Pangu-α: Large-scale autoregressive pretrained chinese language models with auto-parallel computation. *arXiv preprint arXiv:2104.12369*, 2021.

[59] W. Zhang, S. Gupta, X. Lian, and J. Liu. Staleness-aware async-sgd for distributed deep learning. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*, IJCAI'16, page 2350–2356. AAAI Press, 2016.

[60] W. Zhao, D. Xie, R. Jia, Y. Qian, R. Ding, M. Sun, and P. Li. Distributed hierarchical gpu parameter server for massive scale deep learning ads systems. In I. Dhillon, D. Papailiopoulos, and V. Sze, editors, *Proceedings of Machine Learning and Systems*, volume 2, pages 412–428, 2020.

[61] S. Zheng, Q. Meng, T. Wang, W. Chen, N. Yu, Z.-M. Ma, and T.-Y. Liu. Asynchronous stochastic gradient descent with delay compensation. In D. Precup and Y. W. Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 4120–4129. PMLR, 06–11 Aug 2017.