Prague: High-Performance Heterogeneity-Aware Asynchronous Decentralized Training

Qinyi Luo*
University of Southern California
qinyiluo@usc.edu

Jiaao He*†
Tsinghua University
hja16@mails.tsinghua.edu.cn

Youwei Zhuo
University of Southern California
youweizh@usc.edu

Xuehai Qian
University of Southern California
xuehai.qian@usc.edu

Abstract

Distributed deep learning training usually adopts All-Reduce as the synchronization mechanism for data parallel algorithms due to its high performance in homogeneous environment. However, its performance is bounded by the slowest worker among all workers. For this reason, it is significantly slower in heterogeneous settings. AD-PSGD, a newly proposed synchronization method which provides numerically fast convergence and heterogeneity tolerance, suffers from deadlock issues and high synchronization overhead. Is it possible to get the best of both worlds — designing a distributed training method that has both high performance like All-Reduce in homogeneous environment and good heterogeneity tolerance like AD-PSGD?

In this paper, we propose *Prague*, a high-performance heterogeneity-aware asynchronous decentralized training approach. We achieve the above goal with intensive synchronization optimization by exploring the interplay between algorithm and system implementation, or statistical and hardware efficiency. To reduce synchronization cost, we propose a novel communication primitive, Partial All-Reduce, that enables fast synchronization among a group of workers. To reduce serialization cost, we propose static group scheduling in homogeneous environment and simple techniques, i.e., Group Buffer and Group Division, to largely eliminate conflicts with slightly reduced randomness. Our experiments show that in homogeneous environment, Prague is 1.2× faster than the state-of-the-art implementation of All-Reduce, 5.3× faster than Parameter Server and 3.7× faster than AD-PSGD. In a heterogeneous setting, Prague tolerates slowdowns well and achieves 4.4× speedup over All-Reduce.

CCS Concepts. • **Computer systems organization** → **Distributed architectures**; *Heterogeneous (hybrid) systems*; Special purpose systems; • **Software and its engineering** → Concurrency control.

Keywords. decentralized training; heterogeneity; machine learning; deep learning

ACM Reference Format:
Qinyi Luo, Jiaao He, Youwei Zhuo, and Xuehai Qian. 2020. Prague: High-Performance Heterogeneity-Aware Asynchronous Decentralized Training. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20), March 16–20, 2020, Lausanne, Switzerland*. ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3373376.3378499

1 Introduction

Deep learning is popular nowadays and has achieved phenomenal advancement in various fields including image recognition [51], speech processing [20], machine translation [14], gaming [46], health care [61] and so on. The key success of deep learning lies in the increasing size of training data as well as the increasing size of models that can achieve high accuracy. At the same time, it is difficult to train the large and complex models. It is common that training a model may take hours or even days [18]. Therefore, it is crucial to accelerate training in the distributed manner to better prompt wider applications of deep learning.

In distributed training, multiple workers running on a number of compute nodes cooperatively train a model with the help of communication between workers. The current widely used approach of distributed training is data parallelism [4], in which each worker keeps a replica of the whole model, processes training samples independently, and synchronizes the parameters every iteration. *Parameter Server (PS)* [33] was the first approach to support distributed training by introducing a central node which manages one or more shared versions of the parameters of the whole model.

*Both authors contributed equally to this research.
†Jiaao He did this work during his internship at USC.

ASPLOS '20, March 16–20, 2020, Lausanne, Switzerland

ACM ISBN 978-1-4503-7102-5/20/03…$15.00
https://doi.org/10.1145/3373376.3378499

More recently, *All-Reduce* [44], an alternative distributed solution utilizing the advanced Ring All-Reduce algorithm [16], was shown to provide superior performance than PS [27, 32, 50, 60]. To fundamentally improve the scalability, *decentralized training* [22, 23, 34–36, 38, 52, 53] recently received intensive research interests, after [35] showed for the first time theoretically that decentralized algorithms can outperform centralized ones. Unlike PS and All-Reduce, which use specific topology for communication, a decentralized training scheme can use an *arbitrary* connected communication graph to specify point-to-point communication between workers with doubly stochastic averaging.

The first key challenge of distributed training is the *intensive communication* among workers. During execution, gradients or parameter updates are transferred among workers in different nodes to achieve the eventually trained model. In PS, all workers need to communicate with the parameter servers — easily causing communication bottleneck even if the number of workers is relatively small. In All-Reduce, the communication is more evenly distributed among all workers, but since it logically implements all-to-all communication, the amount of parameters transferred is still large. More importantly, to hide communication latency, All-Reduce uses delicate pipelined operations among all workers. It makes this solution vulnerable to system heterogeneity — i.e., when the performance of different nodes/workers and/or the speeds of different communication links are different. Specifically, because All-Reduce requires global synchronization in every step, its performance is bounded by the slowest worker, thereby cannot tolerate heterogeneity well. We believe that *heterogeneity* is the second key challenge of distributed training.

To tolerate heterogeneity, both system and algorithm techniques have been proposed. At system level, backup worker [6] and bounded staleness [21] have been shown to be effective in mitigating the effects of random worker slowdown in both PS [2, 6, 21, 39, 45, 59] and decentralized training [38]. However, even with these two techniques, severe and continuous slowdown of some workers or communication links will eventually drag down other workers and the whole training. This motivates the more fundamental algorithm level solutions. In particular, AD-PSGD [36] probabilistically reduces the effects of heterogeneity with randomized communication. In an additional synchronization thread, each worker randomly selects one worker to perform *atomic* parameter averaging between the two and updates both versions. Atomicity requires that the synchronization is *serialized*: a worker needs to wait for the current synchronization to finish before starting another, no matter if it actively initiates a synchronization or is passively selected by another worker. While the slow workers inevitably have staler parameters and will drag down others' progress, this will only happen if they happen to be selected. Unfortunately, the implementation in [36] only supports a certain type of communication graphs

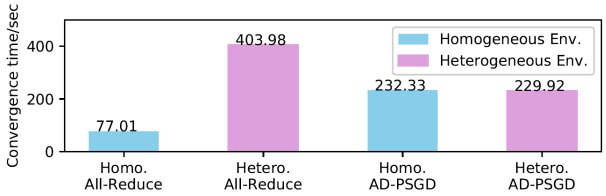


Figure 1. A Comparison Between All-Reduce [44] and AD-PSGD [36] in Homogeneous (Homo) Environment and Heterogeneous (Hetero) Environment.

and suffers from deadlock otherwise. More importantly, the parameter update protocol in AD-PSGD incurs significant synchronization overhead to guarantee atomicity.

Figure 1 shows the training performance[1] of VGG-16 model over CIFAR-10 dataset, of All-Reduce [44] and AD-PSGD on 4 GTX nodes running 16 GPUs as 16 workers in total in homogeneous and heterogeneous[2] execution environment. In Figure 1, we see AD-PSGD's excellent ability to tolerate heterogeneity — 1.75 times faster than All-Reduce. However, the figure also shows that All-Reduce is much faster (3.02×) than AD-PSGD in homogeneous environment. Thus, the **open question** is whether it is possible to achieve the performance that is *comparable to All-Reduce in a homogeneous environment while still maintaining superior ability to tolerate heterogeneity*?

In this paper, we propose *Prague*, a high-performance heterogeneity-aware asynchronous decentralized training approach. Compared to the state-of-the-art solutions, Prague gets the best of both worlds: it achieves better performance than All-Reduce even in homogeneous environment and significantly outperforms AD-PSGD in both homogeneous and heterogeneous environments. We achieve this almost ideal solution with intensive synchronization optimization by exploring the interplay between algorithm and system implementation, or statistical and hardware efficiency. To *reduce synchronization cost*, we propose a novel communication primitive, *Partial All-Reduce*, that enables fast synchronization among a group of workers. To *reduce synchronization cost*, we propose static group scheduling in homogeneous environment and simple but smart techniques, i.e., Group Buffer and Group Division, to largely eliminate conflicts with slightly reduced randomness.

We perform experiments on Maverick2 cluster of TACC Super Computer. We train a common model VGG-16 on CIFAR-10 dataset to look deeply into different algorithms. We also train several ResNets (ResNet-18, -50 and -200) on a large dataset, ImageNet, and the Transformer model on the News-Commentary dataset, to validate the optimizations. Our experiments show that in homogeneous environment, Prague is 1.2× faster than the state-of-the-art implementation of All-Reduce, 5.3× faster than Parameter Server

[1] Defined as the time of training to reach the same loss 0.32
[2] In the heterogeneous setting, one worker is randomly slowed down by 5 times.

and 3.7× faster than AD-PSGD. In a heterogeneous setting, Prague tolerates slowdowns well and achieves 4.4× speedup over All-Reduce.

2 Background and Motivation

2.1 Distributed Training

In distributed training, a single model is trained collaboratively by multiple workers, which run in distributed compute nodes. Training is most commonly accomplished with Stochastic Gradient Descent (SGD), which is an iterative algorithm that reaches the minimum of the loss function by continuously applying approximate gradients computed over randomly selected data samples. In each iteration, there are typically three steps: (1) randomly select samples from the data set; (2) compute gradients based on the selected data; and (3) apply gradients to the model parameters.

There are a number of schemes to achieve parallelism among multiple workers in distributed training: data parallelism [44, 50], model parallelism [9], hybrid parallelism [28, 48, 49, 57, 58], and pipeline parallelism [17]. Among them, data parallelism can be easily deployed without significant efficiency loss compared with other models. Thus, it is supported by many popular machine learning frameworks such as TensorFlow [1], MXNet [7] and PyTorch [40]. Recent papers [28, 48, 49, 57, 58] discussed the trade-offs between data parallelism and model parallelism and proposed the hybrid approach. In this paper, we focus on data parallelism, specifically, solving the open problem in asynchronous decentralized training.

In *data parallelism*, each worker consumes training data independently and computes gradients based on its own sampled data. The gradients obtained by distributed workers are then gathered and applied to model parameters during *synchronization*, then the updated model is subsequently used in the next iteration. Synchronization is both an essential part of parallelizing SGD and a critical factor in determining the training performance.

2.2 Existing Synchronization Approaches

There are three main categories of approaches to performing synchronization in data parallelism: Parameter Servers (PS), All-Reduce, and decentralized approaches.

Training with PS involves using one or more central nodes called *Parameter Servers* that gather gradients from all workers and also send back the updated model to the workers. This straightforward approach enables relatively easy management of the training process. However, PS has limited scalability due to the communication hotspots at Parameter Servers. Parameter Hub [37] provides a new approach to removing the bottleneck of communication by introducing a new network device to work as Parameter Server. While promising, it requires special hardware supports that do not exist in common distributed environment (e.g., Amazon AWS).

In contrast to PS, All-Reduce replaces the use of central nodes with carefully scheduled global communication to achieve better parallelism. The state-of-the-art solutions [32, 44, 50] leverage Ring All-Reduce [40], an advanced all-reduce algorithm that effectively utilizes the bandwidth between computation devices. Specifically, workers are organized as a ring, and gradients are divided into chunks and passed over the ring in a parallel manner. Different chunks of gradients are first accumulated to different workers, which are then broadcast to all workers in a parallel manner. This algorithm achieves ideal parallelism within the theoretical upper bound. Another algorithm, Hierarchical All-Reduce [8, 32], has been successfully scaled up to 4560 nodes with 27360 GPUs. Utilizing All-Reduce algorithms based on MPIs [10, 12, 15] and NCCL [7], Horovod [44] enables high-performance data parallelism and is proved to be effective and efficient — based on All-Reduce algorithms and high performance implementations, researchers were able to use the fastest supercomputer , Summit [11], to train a deep learning model in exascale [32].

Recently, decentralized approaches that allow point-to-point communication between workers by specifying a communication graph received intensive research interests. Both PS and All-Reduce can be considered to be using a specific communication graph. Therefore, they can be viewed as special cases of a generalized decentralized training scheme where workers of possibly different functionalities are connected by a communication graph and cooperate to train a model. In this sense, generalized decentralized training allows more flexibility and thus provides more opportunities for optimization. Two main algorithms proposed so far are Decentralized Parallel SGD (D-PSGD) [35] and Asynchronous D-PSGD (AD-PSGD) [36]. In D-PSGD, every worker has its own version of parameters, and only synchronizes with its neighbors according to the graph. As training proceeds, local information at a worker propagates along edges of the communication graph and gradually reaches every other worker. Thus, models at different workers will collaboratively converge to the same optimal point. The convergence rate has been proved to be similar to that of PS and All-Reduce [35]. Like All-Reduce, D-PSGD does not suffer from communication bottleneck. However, it relies on a fixed communication topology, which may be suspectible to heterogeneity (more discussion in Section 2.3).

To tolerate heterogeneity, AD-PSGD [36] introduces a random communication mechanism on top of D-PSGD. Instead of synchronizing with all the neighbors specified by the communication graph, a worker randomly selects a single neighbor, and performs an *atomic model averaging* with the neighbor, regardless of whether they are in the same iteration or not. While the slow workers inevitably have staler parameters and will affect the training of the global

model, such effects only happen when it is selected, which is probabilistic.

2.3 Challenges and Problems

Communication With the continuously increasing compute capability (e.g., GPUs), communication has become more important and the focus of recent optimizations [3, 25, 35, 41, 56]. Although the communication bottleneck in PS has been eliminated by approaches based on Ring All-Reduce, its strongly synchronized communication pattern has lower heterogeneity tolerance. The generalized decentralized training captures both PS and All-Reduce and enables more optimization opportunities.

Heterogeneity It refers to the varying performance of different nodes (workers) and speed of different communication links. In distributed environments, it is commonly known as the straggler problem. Heterogeneity can be deterministic or dynamic. *Deterministic* heterogeneity is due to different compute capabilities of the hardware (e.g., older CPU/GPU/TPU mixed with newer versions) and network bandwidth discrepancy. As the computing resources evolve over time, the future platforms may contain a mix of computing devices of multiple generations and combine dense homogeneous accelerators using more bursty cluster networking. *Dynamic* heterogeneity [29] can occur due to resource sharing, background OS activities, garbage collection, caching, paging, hardware faults, power limits, etc. Especially, to amortize cost, cloud vendors may employ consolidation/virtualization to share the underlying resources among multiple DNN service requests. The trend of heterogeneity and the "long tail effects" have been discussed and confirmed in other recent works [6, 13, 25, 29, 36]. Due to heterogeneity, slow workers/links may drag down the whole training process.

A number of countermeasures for different synchronization schemes have been proposed, such as asynchronous execution [42], bounded staleness [21], backup workers [6], adjusting the learning rate of stale gradients [29], sending accumulated gradients over bandwidth-scarce links when they reach a significance threshold [25], etc. Unfortunately, these techniques are mostly applicable to only PS and decentralized training.

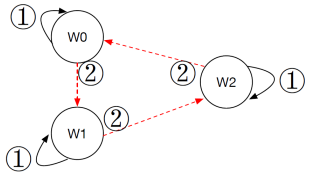
For All-Reduce, with the delicate communication schedule, it is difficult to apply these ideas — making it inherently vulnerable to heterogeneity. From the computation aspect, a global barrier is introduced by the All-Reduce operation, so the throughput of computation is determined by the slowest worker. From the communication aspect, although Ring All-Reduce algorithm is ideal in theory, the speed of sending chunks along the ring is bounded by the edge with the slowest connection.

Considering the delicacy of All-Reduce, and due to the well-known limits of PS, tolerating heterogeneity in decentralized training is particularly important. Recent work Hop [38] proposed the first detailed distributed protocol to support decentralized training [35] with backup worker and bounded staleness to tolerate random slowdown. Although the results are promising, the proposed techniques are essentially *system* techniques to mitigate the effects of heterogeneity. Due to the bounded iteration gap, the severe and continuous slowdown of some workers or communication links will eventually drag down other workers and the entire training. The alternative way is *algorithmic* technique, with AD-PSGD [36] as an excellent example. While AD-PSGD is both communication-efficient and tolerates heterogeneity well, the atomic model averaging step poses a key challenge for synchronization.

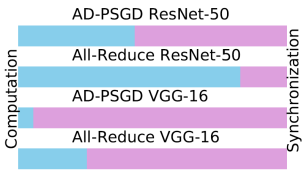
Synchronization Conflict The atomic model averaging requires that two model averaging operations are *serialized* if they involve the same worker. This requirement is to ensure fast convergence, and more relaxed semantic will increase the mutual influence of model updates from different workers — making the global trained model more vulnerable to "staler" updates. Note that the problem is different from the synchronization relaxation in HOGWILD! [42], where conflict happens when two workers try to update the same shared parameter and conflict is expected to be rare, since HOGWILD! requires the cost function to be "sparse" and separable. In the algorithm, workers only update a small fraction of the parameters in each iteration, and the sparsity ensures that updates from different workers rarely involve the same parameter. Therefore, the algorithm can still converge even without any locks. However, in AD-PSGD, the conflict is of a different nature and is expected to be frequent, because every worker can initiate model averaging and it is likely that two workers end up choosing the same worker.

To ensure atomic model averaging and avoid deadlock as exemplified in Figure 2(a), AD-PSGD divides the workers into two sets — active set and passive set, and requires that edges in the communication graph only exist between the two sets, i.e., neighbors of active workers can only be passive workers, and vice versa. This division is only possible when the communication graph is bipartite. In the implementation, only active workers are allowed to initiate model averaging, while passive workers can only respond. This is slightly different from the original algorithm, in which every worker can initiate averaging. When an active worker needs to synchronize, it sends its model to the selected neighbor and blocks until it gets a response. Possible violation of atomicity only happens when two active workers select the same passive worker, and it can be avoided by letting the passive worker deal with the requests one by one. Note that this scheme will incur deadlock if all workers are allowed to initiate model averaging or if the graph is not bipartite.

Besides the restriction of the communication graph between workers, the synchronization overhead is a more crucial problem in a distributed environment. When training



(a) An example deadlock happens when all workers first lock themselves (①), and then try to lock their neighbors in a cycle (②), which blocks forever.



(b) Computation and synchronization ratio of different algorithms on different tasks

Figure 2. Synchronization Issues of AD-PSGD

Require: A set of workers represented as nodes V in a graph and the connection among them are represented by an adjacency matrix W

1: **for** worker $i \in V$ **do**
2: Initialize model weights x_i
3: **while** not reached convergence **do**
4: Step 1. Read the local model x_i from memory
5: Step 2. Compute gradients over randomly selected samples ξ_i, and update weights: $x_i \leftarrow x_i' - \eta_k \cdot \nabla F(x_i; \xi_i)$
6: Step 3. Randomly select a neighbor j
7: Step 4. **Atomically** average weights with the selected neighbor and update the local model as well as the selected neighbor's model: $x_i, x_j \leftarrow \frac{1}{2}(x_i + x_j)$
8: **end while**
9: **end for**

Notes: x_i' may be different from x_i since it may have been modified by other workers in their averaging step (i.e., step 4). To ensure the correctness of execution, it is crucial to implement the averaging step atomically with certain locking mechanisms.

Figure 3. AD-PSGD Algorithm

VGG-16 model over CIFAR-10, and ResNet-50 model over ImageNet using AD-PSGD on 16 GPUs, Figure 2(b) shows that more than 90% of the time can be spent on synchronization in AD-PSGD. This is measured by comparing per iteration time of workers without synchronization (i.e., skip the synchronization operation to see the actual time of computation) and workers with the synchronization enabled.

3 Partial All-Reduce

Based on the results in Section 2.3, we focus on the synchronization challenge for decentralized training. This section first presents a deep analysis of AD-PSGD which motivates our key contribution of Partial All-Reduce primitive.

3.1 AD-PSGD Insights

AD-PSGD algorithm is shown in Figure 3. Similar to traditional training such as PS and All-Reduce, in one iteration, it computes gradients first, and then performs synchronization; the difference is that it only synchronizes with a randomly selected neighbor, instead of all other workers. Therefore, the global barrier is removed, enabling higher training throughput and better heterogeneity tolerance.

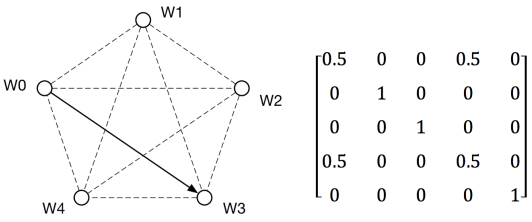


Figure 4. Synchronization in AD-PSGD

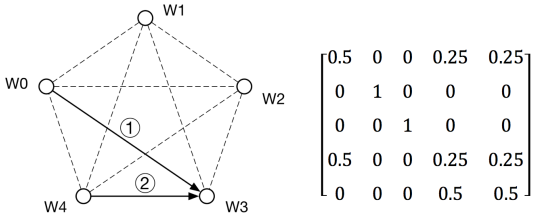


Figure 5. Conflict Between Two Pairs of Workers

In AD-PSGD, each worker i has a local version of parameters, which can be seen as a single concatenated vector x_i, as the shapes do not matter in synchronization. All the weight vectors, after being concatenated together, can be represented as a matrix $X = [x_1 x_2 \dots x_n] \in \mathbb{R}^{N \times n}$ where N is the total size of weights in the model, and n is the number of workers.

In this formalization, one iteration at a worker in AD-PSGD algorithm can be seen as one update to X. Formally, it can be represented as: $X_{k+1} = X_k W_k - \gamma \partial_g(\hat{X_k}; \xi_k^i, i)$. Here, $\partial_g(\hat{X_k}; \xi_k^i, i)$ is the update to x_i according to gradient computation at worker i based on the previous version of the local model $\hat{x_i}$ and a random subset of the training samples ξ_k^i. W_k is a *synchronization matrix* that represents the process of model averaging: $x_i, x_j \leftarrow \frac{1}{2}(x_i + x_j)$, where j is the randomly selected neighbor of i.

Figure 4 shows an example of W_k, in which worker 0 performs a synchronization with worker 3. More generally, for an update between worker i and worker j, the non-zero entries of matrix W_k are: $W_{i,i}^k = W_{i,j}^k = W_{j,i}^k = W_{j,j}^k = 0.5$, $W_{u,u}^k = 1, \forall u \neq i, j$.

In AD-PSGD, a *conflict* happens when two workers i, j both select another worker u for synchronization. In order to keep the atomic property of weight updating, the two operations need to be serialized. In matrix formalization, assume that W_k represents the synchronization between i and u, W_{k+1} represents the synchronization between j and u. Ignoring the gradient entry in the update, the updated weight X_{k+2} can be represented as: $X_{k+2} = X_{k+1} W_{k+1} = (X_k W_k) W_{k+1} = X_k (W_k W_{k+1})$.

Figure 5 shows an example of two workers w_0 and w_4 requiring synchronization with the same worker w_3 ($i = 0, j = 4, u = 3$). The matrix on the right shows the production of W_k and W_{k+1} as a *fused synchronization matrix* $W_{fused} = W_k W_{k+1}$, which shows the final update over all the weights.

We can observe that the production is *commutative* in AD-PSGD — W_k and W_{k+1} can be exchanged (not mathematically but logically). It is because the order of synchronization is determined by the order of acquiring a lock, which is completely random. Based on the atomicity requirement, the key insight is that in AD-PSGD, although the two synchronizations can be mathematically fused, they have to be executed *sequentially*.

3.2 Fast Synchronization with Partial All-Reduce

The straightforward implementation of atomic model averaging with distributed coordination incurs high overhead as shown in Figure 2. Our goal is to propose a communication primitive that can both realize the semantics of the algorithm and enable efficient implementation. We propose *Partial All-Reduce or P-Reduce* primitive that updates a group of workers with the averaged parameters among them. The semantics of P-Reduce can be expressed with the notion of synchronization matrix. Given a group of workers $G = \{w_1, w_2, \ldots, w_k\}$ $W_{P-Reduce}$ involves modifying the weights of all the workers in G. The entries of synchronization matrix $W_{P-Reduce}$ are denoted as F^G, which contains the following non-zero entries: $F^G_{i,j} = \frac{1}{|G|}, \forall i, j \in G$, $F^G_{u,u} = 1, \forall u \notin G$. The implementation of P-Reduce is efficient since it can leverage Ring All-Reduce, the high-performance algorithm that can compute the mean of several copies of weights in $O(N)$ time. Essentially, performing a P-Reduce for a group G is a generalization of the conventional All-Reduce in deep learning training which performs All-Reduce among *all* workers. Next, we explain how can we leverage P-Reduce to improve the efficiency of synchronization in AD-PSGD.

From Figure 3, we can see that Step 3 and 4 can be directly implemented by applying P-Reduce to a random group of size 2. We will need to ensure the atomicity — when the two groups overlap, the corresponding P-Reduce operations need to be serialized. In this case, P-Reduce accelerates the model averaging between two workers with a single collected operation, instead of performing individual read and write operations among workers.

The more interesting case is that, with the configurable size of P-Reduce groups, a single P-Reduce can approximate a sequence of serialized synchronizations (P-Reduce among two workers) in AD-PSGD. Figure 6 shows an example of $W_{P-Reduce}$ when a P-Reduce is performed among worker group $\{0, 3, 4\}$. Comparing it with the matrix in Figure 5, we see that the actual non-zero values are different but *the position of the non-zeros are the same*. Based on this observation, we consider P-Reduce among the three workers $\{0, 3, 4\}$ an *approximation* of two serialized synchronizations: P-Reduce between $\{0, 3\}$ and $\{3, 4\}$ performed in any order. We call this idea *approximate group fusion*, which fuses multiple synchronizations approximately into one with reduced synchronization cost. In the precise group fusion as shown in

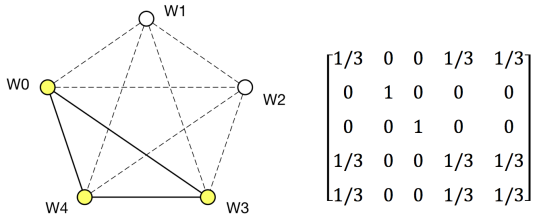


Figure 6. Synchronization with Partial All-Reduce

Require: A set of worker represented as nodes V in a graph and their connection represented by a weighted adjacency matrix W

1: **for** worker $i \in V$ **do**
2: Initialize model parameters x_i
3: **while** not reached convergence **do**
4: Step 1. Read the local model x_i from memory
5: Step 2. Compute gradients over randomly selected samples ξ_i, and update parameters: $x_i \leftarrow x_i - \eta_k \cdot \nabla F(x_i; \xi_i)$
6: Step 3. Randomly generate a group G *including* i.
7: Step 4. Atomically average parameters in group G using P-Reduce:
8: $\bar{x}_G = \frac{1}{|G|} \sum_{\forall g \in G} x_g$
9: $x_g \leftarrow \bar{x}_G, \forall g \in G$
10: **end while**
11: **end for**

Figure 7. Proposed Algorithm Using P-Reduce

Figure 5, the workers update their weights to a certain linear combination of the weights of each worker in the group. We will explain in Section 3.3 that the approximation with P-Reduce satisfies all algorithm requirements in AD-PSGD.

We present a new asynchronous decentralized training algorithm in Figure 7 using P-Reduce as the efficient synchronization primitive. Compared to the original AD-PSGD algorithm, there are two key differences. First, in Step 3, each worker can randomly generate a group that may be larger than 2, as long as it contains itself, w_i. The group in AD-PSGD of size 2 (one worker randomly selects a neighbor) becomes a special case. It essentially enlarges the *unit of synchronization* to groups of any size. Larger groups have two implications: (1) potentially enable fast propagation of model parameter updates among workers, speeding up convergence; and (2) increase the chance of conflicts. Thus the new algorithm allows the system to explore such a trade-off. The second difference from AD-PSGD is that the synchronization operation is performed by the new primitive P-Reduce involving the workers in the group, instead of using individual messages among workers. This directly reduces the cost of synchronization.

Although P-Reduce of more than two workers can approximate the effects of serialized synchronizations (group fusion), our algorithm in Figure 7 does *not* fuse groups during execution. Instead, the effects of fusing two groups of size 2 in AD-PSGD is reflected as generating group of arbitrary

size in Step 3 of Figure 7. The algorithm still requires atomicity among P-Reduce operations with overlapped groups. If two G's do not share common workers, the two P-Reduce can execute concurrently. In an ideal situation, P-Reduce groups can be determined and scheduled in a manner to avoid any conflict. It is the motivation for static scheduling in Section 4.2. Compared to All-Reduce, P-Reduce retains the efficient implementation while avoiding the global barrier.

3.3 Convergence Property Analysis

To guarantee that models at different workers converge to the same point, three requirements for W_k are proposed in AD-PSGD [36]. In the following, we show that although F^G is not exactly the same as the result of multiplying a sequence of synchronization matrices in a certain order, our definition of F^G in P-Reduce satisfies all three convergence properties as AD-PSGD does.

Doubly stochastic averaging W_k is doubly stochastic for all k. The sum of each row and each column equals to 1 in both W_k and F_k^G.

Spectral gap It requires the existence of $\rho \in [0, 1)$, such that: $max\{|\lambda_2(\mathbb{E}[W_k^T W_k])|, |\lambda_n(\mathbb{E}[W_k^T W_k])|\} \leq \rho, \forall k$. Basically, $(F^G)^T F^G = F^G$. Here, $\mathbb{E}[F^G]$ can be regarded as a Markov Transition Matrix. According to the Expander Graph Theory [24], the spectral gap condition is fulfilled if the corresponding graph of random walk is connected. That means the update on any worker can be passed through several groups to the whole graph. When creating the group generation methods in the following section, this property is ensured by random group generation or static pre-determined group scheduling.

Dependence of random variables W_k is a random variable dependent on i_k[3], but independent on ξ_k and k. Up to now, the only requirement on the generated group G_k is that it should contain the initial worker i_k. Theoretically, it is generated randomly without any connection to k or ξ_k. Therefore, this condition is also satisfied.

4 Group Generation and Conflict Detection

With P-Reduce, a group of workers becomes the basic unit of the synchronization procedure. As a type of collective operation, all workers in the group need to call P-Reduce function. It means that all group members should have the same group information to initiate the P-Reduce. It is non-trivial to obtain the consistent group among all workers inside the group. This section discusses how to generate the groups and serialize conflicting groups.

[3] i_k is the worker initiating the synchronization.

4.1 Group Generator

In Figure 7, each worker needs to randomly generate a group. This can be performed by each worker based on the communication graph with randomly selected neighbors. The workers in each group will collectively perform P-Reduce. The system needs to ensure atomicity — P-Reduces of groups with overlapping workers selected must be serialized. This can be implemented in either a centralized or distributed manner. In general, a distributed protocol involves multiple rounds of communication and coordination between workers. For simplicity, Prague implements a centralized component and offloads the group generation functionality from the workers to a dedicated component *Group Generator (GG)*. When a worker needs to perform a synchronization, it just needs to contact GG without any group information, and then GG can select the group on behalf of the worker and maintain the atomicity. In the following, we explain the protocol using an example. We will find that the communications between workers and GG are only small messages, and do not introduce communication or scalability bottleneck.

In Figure 8, we consider four workers W_0,W_4,W_5,W_7 among a total number of 9 workers. In the beginning, W_0 and W_7 finish an iteration and need to perform a synchronization. Instead of generating groups locally, they both send a synchronization request to GG, indicated in ① and ②. GG maintains the atomicity with a local lock vector — a bit vector indicating whether each worker is currently performing a P-Reduce. This vector is initialized as all 0s. Assume that there is no other synchronization being performed in the system, and GG receives the request from W_0 first. After that, GG randomly generates a group $[0, 4, 5]$ on behalf of W_0 (③) and sets the corresponding bits in the lock vector (④). Then, GG notifies the workers W_0, W_4, and W_5 (⑤) in the group so that they can collectively perform the P-Reduce. Later, GG receives the synchronization request from W_7 and randomly generates a group $[4, 5, 7]$. Unfortunately, it is conflicting with the first group due to the two overlapped workers W_4 and W_5, and needs to be serialized. We can achieve this by simply blocking the group $[4, 5, 7]$ and storing it in a pending group queue (⑥). In the meantime, W_0,W_4 and W_5 receive the notifications from GG and perform P-Reduce (⑦). They also need to acknowledge GG to release the locks (⑧). After the locks for group $[0, 4, 5]$ are released in GG, the group $[4, 5, 7]$ can be performed after setting the corresponding bits in the lock vector.

Note that in the actual implementation of GG, each worker can request a group at the *beginning* of an iteration, so that the small amount of communication with GG is overlapped with the gradient computation, hiding the overhead of group generation. Moreover, since the performance and reliability of GG are crucial, we recommend selecting a dedicated stable device for GG to minimize resource sharing, e.g., a stable node on the Cloud or the login node of a large cluster. We

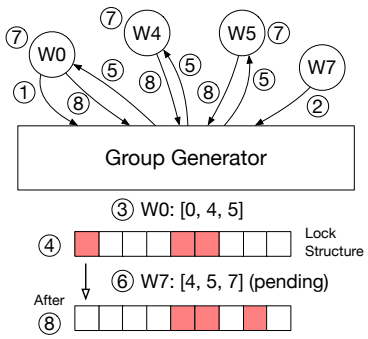


Figure 8. GG Generates Groups on Behalf of Workers

Iteration	W0	W1	W2	W3	W4	W5	W6	W7	W8	W9	W10	W11	W12	W13	W14	W15
4k	G5	-	G1		G5	-	G2		G5	-	G3		G5	-	G4	
4k+1	G1				G2				G3				G4			
4k+2	G1	G5	-	G1	G2	G6	-	G2	G3	G5	-	G3	G4	G6	-	G4
4k+3	G1				G2				G3				G4			

Figure 9. A Conflict-Free Static Scheduling Strategy

Phase	L.W. 0	L.W. 1	L.W. 2	L.W. 3
0	Sync with L.W. 0s *on ALL nodes*	No sync	Sync with L.W. 3	Sync with L.W. 2
1	Sync L.W. 0-3			
2	Sync with L.W. 3	Sync with L.W. 1 *on the opposite node on the ring*	No sync	Sync with L.W. 0
3	Sync L.W. 0-3			

Notes: This table shows the rules that generate the schedule for 4 workers running on *one node*. The rules are *the same for all 4 nodes*. L.W. k stands for Local Worker k, the k-th worker on this node. The schedule has 4 phases, each corresponds to one training step. It repeats itself after every 4 steps.

Figure 10. An Example of the Static Scheduling Algorithm

believe that this effort is worthwhile and fairly lightweight. For algorithms like All-Reduce that do not provide support against stragglers, we would have needed *every* worker device to be reliable and dedicated in order to ensure good performance.

4.2 Decentralized Static Scheduler

As we have seen in the example in Figure 8, two overlapping groups need to be serialized to ensure atomicity, causing delay in the execution. We can eliminate the conflict by statically scheduling the groups in a conflict-free manner, completely eliminating serialization overhead.

We design a conflict-free schedule as shown in Figure 9. There are 16 workers in total, and the schedule is periodic with a cycle length of 4. Every row corresponds to an iteration, and colored blocks with group indices indicate the grouping of workers. For example, in the first row, W_0, W_4, W_8 and W_{12} are all colored yellow with an index "G1", which means that these 4 workers are in the same group in the $(4k)$-th iteration, for any $k \in \mathbb{N}$. Group indices do not indicate the sequence of execution; in fact, groups in the same row are expected to execute concurrently. In addition, some workers do not participate in synchronization in certain iterations, and this is shown by gray blocks marked with a hyphen "-". For instance, W_2, W_6, W_{10} and W_{14} do not participate in any group in the $(4k + 2)$-th iteration, for any $k \in \mathbb{N}$. Skipping synchronization can decrease the frequency of communication and thus shorten the training time. It is a technique that has been proved helpful in [30, 56].

To implement static scheduling, a naive way is to store the schedule table in the GG, and workers can access it by contacting the GG. Alternatively, we can store the table inside each worker, saving a round trip of communication between the worker and the GG. Since every worker has the same schedule table stored locally, a consistent view of the groups is naturally ensured.

In fact, storing a table is unnecessary, since the schedule is generated in a rule-based manner. For example, our previously proposed schedule is based on a worker's rank in its node. In an example where 4 workers are on a node, the rule of scheduling is shown in Figure 10. In this way, a worker can simply call a local function S to obtain its group in an iteration. The logic of S guarantees that the schedule is consistent among all the workers, and a conflict-free static schedule is therefore achieved.

4.3 Discussion: Random vs. Static

Although static scheduling can ideally eliminate conflict and speed up execution, randomized group generation is more suitable for heterogeneous environment. We compare the different characteristics of the two approaches below.

Random GG is centralized, but it is different from Parameter Servers in that it does not involve massive weight transfer and costs minor CPU and network resources compared with gradient accumulation or weight synchronization. In our experiment, we find that GG can be placed on a node together with workers without incurring any performance loss. In random GG, contacting the GG induces communication overhead, and conflicting groups need to be serialized, resulting in additional wait time.

On the contrary, GG implemented as a static scheduler has no communication latency. With a proper design of S, it not only fully parallelizes synchronization, but also utilizes the architecture of the worker devices to accelerate every single P-Reduce operation. For example, it can schedule more intra-node synchronizations, and reduce the number of large-scale inter-node synchronizations. However, the S function is pseudo random, which breaks the strict convergence condition of AD-PSGD, although the resulting algorithm still converges well in our experiments.

When a certain worker is slower than others, the original AD-PSGD algorithm is able to tolerate the slowdown. However, the static scheduler does not have such ability, as the schedule is in fact fixed. Synchronizations with the

slow worker will slow down the whole training. As for random GG, the stragglers' effect can be largely ameliorated. Well-designed group generation strategy can ensure that at any time, most workers will be able to proceed without depending on the few slow workers, thus tolerating slowdown. Also, slowdown detection and conflict avoidance mechanisms, which will be discussed in the following section, can be easily integrated into random GG, making it more advantageous in a heterogeneous environment.

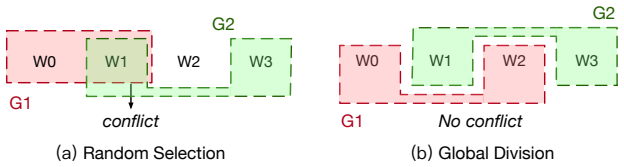
5 Smart Randomized Group Generation

The basic implementation of the scheduler in GG is to always randomly generate a group as specified in Step 3 of Figure 7. With the centralized GG, our objective is to leverage the global runtime information to generate groups in a more intelligent manner to: (1) avoid conflicts; and (2) embrace heterogeneity. For example, a worker may have already been assigned to several groups and thus have several pending P-Reduces to perform. If the worker is still selected to be included in a new group, then other workers will have to wait for all the prior scheduled P-Reduces to finish. Similarly, when a slow worker is in a group, the whole group may be blocked by this worker. Moreover, performing P-Reduce in different groups cost different time due to architecture factors. The group selection can even introduce architectural contentions on communication links. Based on the above insights, we propose intelligent scheduling mechanisms for GG to further improve performance.

5.1 From Random Group to Random Division

An intuitive way of reducing conflict is to have a *Group Buffer (GB)* for each worker, which includes the ordered list of groups that include the corresponding worker. When a group is formed, the group information is inserted in the GB of all workers involved. The consensus group order can be easily ensured among all GBs since the GG, as a centralized structure, generates groups sequentially. Based on GB, when GG receives a synchronization request from a worker, it can first look up the worker's GB. If it is empty, a new group is generated for the worker; otherwise, the first existing group in the worker's GB will serve as the selected group.

The main insight is that P-Reduce is a collective operation. If W_i initiates a synchronization with W_j, i.e., W_i and W_j are in the same group, P-Reduce of this group is only performed when W_j also requests its synchronization. Therefore, the simple mechanism can avoid generating a new group for W_j when it is already scheduled and ready to execute a P-Reduce. However, with random group generation, nothing would prevent the selection of W_j into a different group *not* initiated by W_i. In this case, the overlapping groups and the corresponding P-Reduce operations have conflict and will be serialized.



Notes: In random selection shown in (a), after G_1 is generated by a request from W_0 and W_1 gets its group, no information is left to avoid the conflict that another request from W_3 may also generate a group including W_1. In GD shown in (b), two groups are both generated upon the first request. Therefore, the second request directly gets a conflict-free group from the buffer.

Figure 11. An Example of Global Division

To further reduce conflicts, we propose an operation called *Global Division (GD)* that *divides all current workers with empty GBs into several non-conflicting groups*. It is inspired by the static scheduling. A GD is called whenever a worker needs to generate a group and its GB is empty. A simple example is shown in Figure 11. In total we have 4 workers and initially all GBs are empty. On the left, random selection shows a possible scenario without GD optimization. The groups are randomly generated, so if G_1 initiated by W_0 includes W_0 and W_1, another group G_2 initiated by W_3 can still include W_1 as the overlapped worker, thus causing a conflict. On the right, with GD, when W_0 requests a group, the GG will not only generate one for it, i.e., $[W_0, W_2]$, but also randomly generate groups for other workers, i.e., only $[W_1, W_3]$ in this example as there are only 4 workers. In this way, when later W_3 requests a group, GG will directly provide the non-conflicting $[W_1, W_3]$ generated before.

It is worth emphasizing two conditions. First, a GD only generates groups for the current "idle" workers (including the caller worker) that are not assigned to any group. Thus, when a worker requests a group, it is possible to generate groups in the above manner for just a *subset* of workers. Second, a GD is only called when the initiator's GB is empty, otherwise the first group in the initiator's GB will be returned.

Indeed, the proposed schemes to avoid conflict make the group generation not fully random. However, we argue that the effects are not critical. For the first optimization based on GB, we only reuse the existing group involving the worker who is requesting synchronization. This group is still generated in a fully random manner (without using GD). For GD, essentially we generate a *random group partition* among all idle workers together, which is triggered by the first worker in the set who initiates a synchronization. So the difference is between randomly generating each group and generating a random partition. We acknowledge that they are not the same but believe that our method does not significantly reducing the randomness. We leave the theoretical analysis as the future work. However, based on the results shown in our evaluation, the ideas work very well in practice.

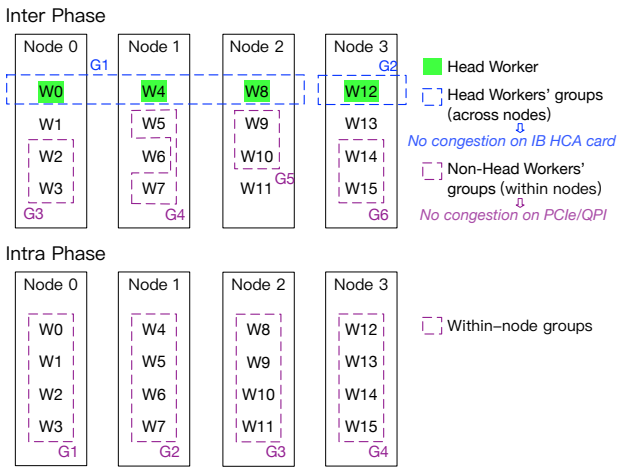


Figure 12. An Example of Inter-Intra Synchronization

5.2 Architecture-Aware Scheduling

If the groups are randomly divided, multiple groups may all need to use the network bandwidth at the same time, causing congestion, which is not optimal in the perspective of architecture. In fact, All-Reduce is fast because it has a balanced utilization of different connections between different devices, such as Infiniband HCA cards, QPI paths[4], and PCIe slots. To better utilize the bandwidth of different connections, we propose a new communication pattern called *Inter-Intra Synchronization* that can be naturally incorporated with GD. Here, a node, commonly running 4 or 8 workers, are considered a unit. The scheme has an *Inter* and an *Intra* phase.

***Inter* phase** One worker on each node is selected as *Head Worker* of the node. All the *Head Workers* are randomly divided into several groups to synchronize in an inter-node manner. At the same time, the workers that are not *Head Worker* are randomly assigned to groups with only local workers in the same node. In this way, only the Head Worker can generate inter-node communication while the others only incur local communication, which can be carefully arranged to avoid congestion on PCIe switches or QPI.

***Intra* phase** Workers within a node synchronize with *all other local workers* collectively. In another word, it involves a P-Reduce among all the workers in the same node, without any inter-node communication. Following the *Inter* phase, the updates from workers on other nodes can be quickly propagated among local workers in this phase.

The two phases can be realized easily with GD operations. Specifically, two groups are inserted into the GB of each worker. Each group is generated by a GD, one is mainly among Head Workers in different nodes (the *Inter* phase), the other is purely among local workers in the same node (the *Intra* phase). An example can be seen in Figure 12.

[4]The Intel QuickPath Interconnect between CPU sockets within one node

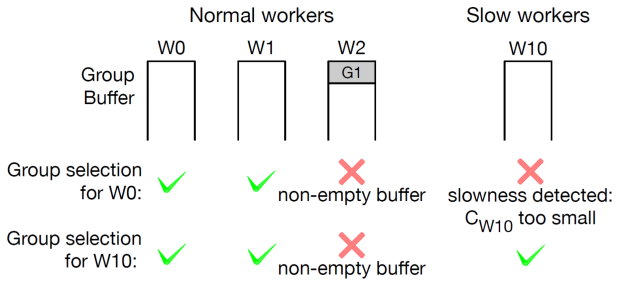


Figure 13. Tolerating Slow Workers

It is worth noting that the proposed *Inter-Intra Synchronization* is not the same as hierarchical All-Reduce [8], which is mathematically equivalent to All-Reduce among all workers with acceleration brought by the hierarchical architecture. After an All-Reduce, all workers end up with the same weight. Differently, *Inter-Intra* synchronization strategy spreads multiple partial updates through P-Reduce in an architecture-aware and controlled manner. Thus, workers end up with *different* weights after the synchronization.

5.3 Tolerating Slowdown

The mechanisms proposed so far are mainly effective in homogeneous execution environment but do not help with slowdown situations. Slow workers involved in groups can block the current and other groups as mentioned earlier.

We propose a simple solution by keeping track of execution information in GG. Specifically, an additional counter for each worker is placed in GG, which records how many times the worker requires a group. When a worker is significantly slower than other workers, the value of its counter should be also much smaller than the average. As a *GD* starts when a worker with an empty GB requests a group, an additional rule is added to filter the workers who can get a group in the division: the worker's counter, c_w, should not be significantly smaller than the initiator's counter, c_i, i.e., $c_i - c_w < C_{thres}$, where C_{thres} is a constant that can be adjusted.

This filter works as follows. When a fast worker initiates a GD, only fast workers are assigned to groups, avoiding the problem of being blocked by slow workers. For example, in Figure 13, when W_0 initiates a GD, W_{10} is detected as a slow worker and thus excluded by the filter rule, because C_{W10} is too small. (W_2 is also excluded because its group buffer is not empty.) Then, when a slow worker initiates a division, some faster workers may be involved to synchronize with it. But the selected workers must have empty buffers as defined in the GD operation. In Figure 13, when the slow worker W_{10} wants to initiate a GD, only W_2 is excluded because of its non-empty group buffer. In this way, neither the fast workers nor the slow worker needs to wait for a long time for synchronization. By the filter rule, the effect of slow workers is minimized.

6 Implementation

We implement the proposed algorithms and protocols using TensorFlow and its extensions. Specifically, Prague is implemented as customized operators of TensorFlow.

6.1 Partial All-Reduce

Partial All-Reduce is implemented as a GPU TensorFlow Operator. It takes the variables and the group as input tensor, and outputs a new tensor representing the result of synchronization. NCCL [26] is used to execute All-Reduce, and MPI is used to help create NCCL communicator. We use a simple but effective strategy to concatenate all weights into one tensor. Specifically, all weights are flattened and concatenated into one tensor for faster P-Reduce, and are separated and reshaped after the P-Reduce operation.

In NCCL, the upper bound of existing communicators is 64. But it is inefficient to destroy all the communicators after use. To save the time of creating communicators, a distributed cache for communicators is used, which provides consistent presence of communicators. It does not remove cached items, but simply stops caching when its size exceeds a threshold.

6.2 Group Generator

Group Generator is a centralized controller among all workers. It requires low latency remote function call. RPC is used in this scenario. The server is a light-weight Python program implemented by gRPC Python package. C++ is used in the core of the algorithms. It can be started and killed easily.

The client is wrapped up as another TensorFlow Python Operator. One function, the static scheduler, is implemented according to the scheduling rules. Another function, the dynamic group generator using the centralized GG, also uses gRPC. We can easily switch between the methods of group generation using executing flags.

7 Evaluation

7.1 Evaluation Setup

7.1.1 Hardware Environment. We conduct our experiment on *Maverick2* cluster of *TACC* Super Computer. Maverick2 is a cluster managed by SLURM. In the GTX partition, a node is configured as shown in the table in Figure 14.

Model	Super Micro X10DRG-Q Motherboard
Processor	2 x Intel(R) Xeon(R) CPU E5-2620 v4
GPUs	4 x NVidia 1080-TI GPUs
Network	Mellanox FDR Infiniband MT27500 Family ConnectX-3 Adapter

Figure 14. Configuration of a Node in GTX Partition, Maverick2 Cluster, TACC Super Computer [5]

7.1.2 Dataset and Model. We evaluate Prague on two machine learning tasks: image classification and machine translation. For image classification, we train models on both medium and large data sets: (1) VGG-16 model [47] on CIFAR-10 [31] image classification dataset; and (2) ResNet-18, ResNet-50 and ResNet-200 [19] on the ImageNet [43] dataset. For machine translation, we train the Transformer model [55] on the News-Commentary [54] dataset. The training models are implemented using TensorFlow [1].

Model	VGG-16	ResNet-18	ResNet-50	ResNet-200	Transformer
Per-GPU batch size	128	256*	96*	32*	2048*
Initial learning rate	0.1	0.128	0.128 PR: 0.02	0.128 PR: 0.008	2.0

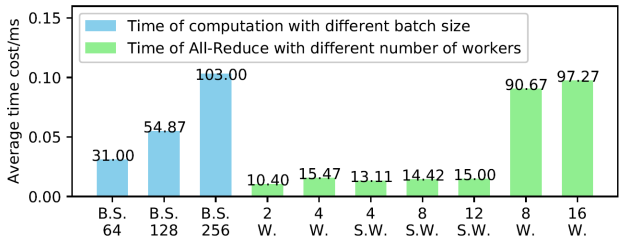
Notes: When the initial learning rate is different for Prague than for the other algorithms, the learning rate for Prague is shown under "PR:". Batch size with an asterisk (*) is the maximum possible to fit into the memory.

Figure 15. Hyper-Parameters Used in the Evaluation

7.1.3 Baseline Setup. Parameter Server is already integrated in TensorFlow. We implement AD-PSGD using remote variable access supported by the TensorFlow distributed module. Horovod [44] is adopted to set up a high-performance state-of-the-art baseline, which significantly outperforms many other implementations of All-Reduce. It is configured with NCCL2 [26] in order to achieve the best All-Reduce speed. We also tune the size of the fusion buffer — which is used for Tensor Fusion [44] in Horovod — for better utilization of the Inifiniband network. Tensor Fusion aims at reducing the overhead introduced by performing All-Reduce operations on small-sized gradients. In all test runs, each worker occupies a whole GPU. For better affinity, we bind the process of each worker to the CPU socket it is directly attached to. In random GG, the group size is 3. For both VGG-16 and the ResNets, momentum optimizer is used with $momentum = 0.9$ and $weight\_decay = 10^{-4}$.

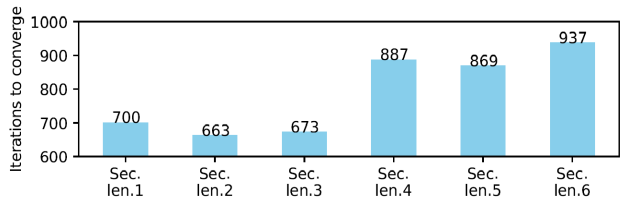
To determine the batch size and the initial learning rate, we performed a grid search over possible combinations and ran each setting for 30 mins to select the optimal one based on the resulting train accuracy. Across all the models, we considered in total 8 batch sizes (32, 64, 96, 128, 256, 512, 1024, 2048) and more than 20 learning rates. During training, learning rate was divided by 10 whenever the validation accuracy (which was run after each epoch) stopped increasing compared to the previous epoch. The resulting parameters are shown in Figure 15.

7.1.4 Methodology. We use the time it takes for the model (randomly initialized using a fixed random seed across different experiments) to achieve $loss = 0.32$ as the metric of performance on VGG-16. As for the experiments on ImageNet, since the time taken to run the programs till convergence is long, we divide the experiments into 2 types: (1) fixed-time experiments to show that Prague can reduce the loss value faster; and (2) full-length experiments to show that Prague



Notes: B.S. means the batch size is 64, 128, 256. W. means running 2, 4, 8, 16 workers densely placed on 1, 1, 2, 4 nodes. S.W. means running 4, 8, 12 workers, one on a node, using 4, 8, 12 nodes.

Figure 16. A Micro-Benchmark Showing the Cost of Different Operations in Computation and Synchronization.



Notes: The frequency of communication is controlled by a hyper-parameter *Section Length*, – # of iterations between two synchronizations.

Figure 17. Effects of Reducing Synchronization

can achieve competitive accuracy compared to All-Reduce. We also conduct fixed-time experiments for evaluation using Transformer. In addition, we inspect the loss w.r.t. iteration curve and the average duration of an iteration to analyze the effect of our optimizations.

7.2 Interactions between Computation, Communication and Convergence

In order to better understand how much time communication takes in deep learning training compared to computation time, we first measured the time of computation with different batch sizes and time of communication with different settings[5]. Figure 16 shows the time comparisons. Because of better utilization of SIMD devices, the computation is slightly more efficient when the batch size is larger. Interestingly, All-Reduce among workers within a single node or workers separately placed across different nodes are significantly faster than having multiple nodes with each running multiple workers.

Although reducing communication by lowering synchronization frequency can increase the throughput of training, it effects the convergence speed. Figure 17 presents a simple experiment to show that the number of iterations needed to converge generally increases as communication frequency becomes lower. To achieve the best performance of convergence time, setting a proper level of synchronization intensity is necessary. This result shows that we cannot simply improve AD-PSGD by enlarging the amount of computation between synchronizations.

[5]Size of weight to be synchronized is independent of batch size

7.3 Speedup in Homogeneous Environment

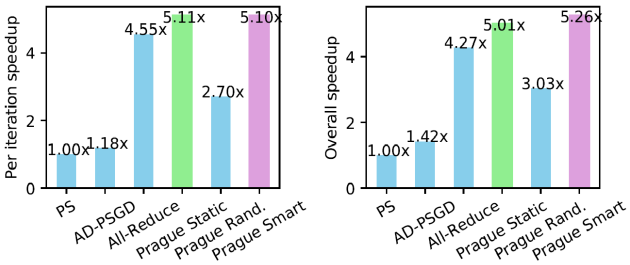
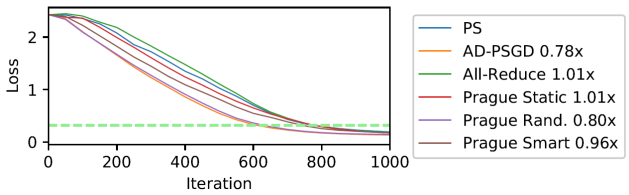


Figure 18. Per-Iteration Speedup and Overall Speedup

In a homogeneous environment with 16 workers on 4 nodes, firstly, VGG-16 trained over CIFAR-10 is used to compare Prague with *different ways of group generation* against Parameter Server, All-Reduce and AD-PSGD. The per-iteration speedup and convergence time speedup is shown in Figure 18. Prague is much faster than Parameter Server and the original AD-PSGD. All-Reduce is also much faster than these two baselines, due to the high throughput provided by Horovod. However, Prague with both static scheduler and smart GG even outperforms All-Reduce thanks to its smaller synchronization groups and architecture-aware scheduling.



Notes: The speedup in the figure means the number of iterations to converge compared to Parameter Server.

Figure 19. Convergence Curve in Terms of Number of Iterations for Corresponding Algorithms in Figure 18

Shown in Figure 19, AD-PSGD has better convergence speed in terms of number of iterations. All-Reduce is mathematically equivalent to Parameter Server. They are slightly different due to random sampling and competition in synchronization. Prague with static scheduler has similar convergence speed as Parameter Server, but it gains speedup due to its higher throughput. We see that the number of iterations in random GG is less than smart GG, which is smaller than static scheduling. This is due to the decreasing amount of randomness from random GG to smart GG and to static scheduling.

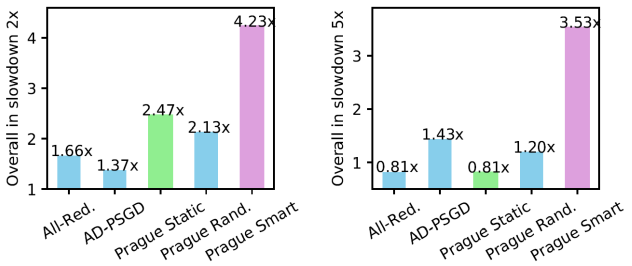
These results further demonstrate the trade-offs between execution efficiency and statistical efficiency [62]. Although AD-PSGD needs fewer iterations to converge to the same error, the execution time of each iteration is seriously affected by the synchronization overhead, shown in Figure 2 (b). Prague successfully explores this trade-off by slightly

sacrificing statistical efficiency, i.e., running more iterations (0.96x vs. 0.78x), — mainly caused by the reduced randomness, to gain significant speedup in per iteration execution time (5.10x vs. 1.18x) and eventually lead to overall execution time speedup (5.26x vs. 1.42x).

We also conducted a 5-hour fixed-length experiment with the Transformer model trained over the News-Commentary dataset. We observe that Prague achieved 4× speedup on the execution time of each step compared to ring All-Reduce. To reach the same loss of 2.0, Prague achieves 3.9× speedup over All-Reduce on total execution time. The improvement here is more significant than that in the CNN task — this is due to the nature of the model: the major calculation in Transformer is dgemm, compared to CNNs, it has more parameters but simpler computation, so Prague achieves more significant advantage in throughput. On BLEU score (an accuracy metric in NLP.), Prague reached 25.5, while All-Reduce obtained 21, and the reference BLEU score is 27. This shows that Prague can actually achieves *higher* accuracy for certain type of models. Additional results on fixed-time experiments with ResNet-20 and ResNet-200 can be found in Figure 21 and 22, respectively.

In terms of the final accuracy, the full-length experiment on ResNet-50 reached a final accuracy of 74.16% for smart GG and 74.05% for All-Reduce. The results show that the relaxation in Prague does not prevent training from reaching high accuracy compared to the state-of-the-art. Both algorithms reached their respective final accuracy at similar times — since the validation accuracy was only run periodically, it is hard to compute the exact speedup.

7.4 Heterogeneity Tolerance



Notes: The baseline is still Parameter Server without slowdown in Figure 18 for convenience of comparison.

Figure 20. Overall Speedup of All-Reduce, Prague with Static Scheduler and Prague with Random and Smart GG in Heterogeneous Environment (2x or 5x Slowdown on One Worker).

One of the key advantages of Prague is better tolerance of heterogeneity. We conducted experiments on VGG-16, ResNet-20 and ResNet-200 to demonstrate this advantage. Here we discuss the results on VGG-16 in detail. Additional results on fixed-time experiments with ResNet-20 and ResNet-200 can be found in Figure 21 and 22, respectively.

Algorithm	Homogeneous		5x slowdown	
	Final loss	Images/s	Final loss	Images/s
All-Reduce	2.06	376.47	2.99	89.8
AD-PSGD	2.48	234.9	2.88	104.1
Prague Static	**2.05**	**415.8**	-	-
Prague Smart	2.07	412.3	**2.82**	**169.5**

Notes: We did not evaluate Prague Static with 5x slowdown because the static algorithm was not meant for heterogeneous environment.

Figure 21. Fixed-Time (2h) Experiments on ResNet-18

Algorithm	Homogeneous		5x slowdown	
	Final loss	Images/s	Final loss	Images/s
All-Reduce	4.12	**54.78**	31.98	13.01
AD-PSGD	11.61	28.57	25.79	12.45
Prague Static	3.52	52.34	-	-
Prague Smart	**3.45**	53.43	**5.10**	**24.62**

Notes: We did not evaluate Prague Static with 5x slowdown because the static algorithm was not meant for heterogeneous environment.

Figure 22. Fixed-Time (2.5h) Experiments on ResNet-200

Based on the same setup for VGG-16 in section 7.3, heterogeneity is simulated by adding 2 or 5 times the normal iteration time of sleep every iteration on one randomly selected slow worker. The result is shown in Figure 20. In terms of the capability to tolerate slowdown, experiment results of 2× slowdown show that: (1) random GG (3.03x vs. 2.13x) is slightly worse than AD-PSGD (1.42x vs. 1.37x), but it is much faster due to more efficient P-Reduce as the synchronization primitive; (2) smart GG (5.26x vs. 4.23x) is better than random GG (3.03x vs. 2.13x); and (3) while both suffer from more slowdown, Prague static (5.01x vs. 2.47x) is still considerably better than All-Reduce (4.27x vs. 1.66x). We also see that with 2× slowdown, All-Reduce is still faster than AD-PSGD although much slower than itself in homogeneous setting. With 5× slowdown, All-Reduce can only achieve a little more than half of the performance in AD-PSGD. We see that random GG is slightly slower than AD-PSGD, this is because the larger group size (3) in Prague can increase the chance of conflicts. Nevertheless, smart GG outperforms AD-PSGD with a large margin.

7.5 Validating Results on a Larger Scale

The experiments discussed before were all conducted using 16 workers evenly distributed over 4 nodes, with 1 GPU per worker. In this section, we show the training performance of ResNet-50 on ImageNet using 8 nodes with a total of 32 workers. Each experiment was run for a fixed length of 10 hours. We conduct experiment in this manner to avoid affecting other experiments on the cluster, as TACC Super Computer is shared by thousands of researchers.

Algorithm	Total iterations	Top 1 Accuracy	Top 5 Accuracy
All-Reduce	55800	66.83%	84.81%
AD-PSGD	32100	58.28%	78.00%
Prague Static	58200	63.79%	82.38%
Prague Smart	56800	64.21%	82.78%

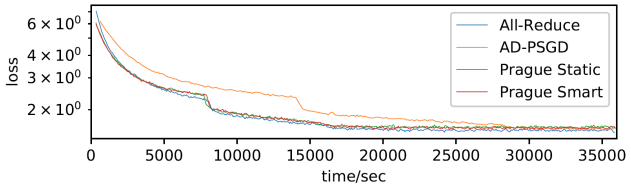


Figure 23. Iterations Trained, Final Training Accuracy of Different Algorithms After Training for 10 Hours, and Loss Curve During the 10 Hours.

The training accuracy and the loss curves for the 10-hour executions are shown in Figure 23. The execution environment is homogeneous without slower workers. We see that All-Reduce performs the best in this case, followed by Prague with smart GG. AD-PSGD suffers from throughput limitation. In ResNet-50 over ImageNet, the upper bound of effective batch size is very large. Therefore, although we make our best effort to enlarge the batch size, All-Reduce obtains much bigger convergence advantage numerically, while Prague can train more iterations using the same time. The smart GG performs better than static scheduler because it has more randomness in synchronization. Judging from the loss curve, Prague has competitive convergence speed compared with the state-of-the-art approach, All-Reduce.

8 Conclusion

This paper propose *Prague*, a high performance heterogeneity-aware asynchronous decentralized training approach. To reduce synchronization cost, we propose a novel communication primitive, Partial All-Reduce, that allows a large group of workers to synchronize quickly. To reduce synchronization cost, we propose static group scheduling in homogeneous environment and simple techniques (Group Buffer and Group Division) to avoid conflicts with slightly reduced randomness. Our experiments show that in homogeneous environment, Prague is 1.2× faster than the state-of-the-art implementation of All-Reduce, and is 5.3× faster than Parameter Server and 3.7× faster than AD-PSGD. In a heterogeneous setting, Prague shows 4.4× speedup over All-Reduce.

Acknowledgments

We thank our shepherd Prof. Michael Carbin and the anonymous reviewers for their insightful comments and suggestions. This work is supported by National Science Foundation (Grant No. CCF-1657333, CCF-1717754, CNS-1717984, CCF-1750656, CCF-1919289).

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, 2016.

[3] Dan Alistarh, Demjan Grubic, Jerry Li, Ryota Tomioka, and Milan Vojnovic. Qsgd: Communication-efficient sgd via gradient quantization and encoding. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 1709–1720. Curran Associates, Inc., 2017.

[4] Tal Ben-Nun and Torsten Hoefler. Demystifying parallel and distributed deep learning: An in-depth concurrency analysis, 2018. cite arxiv:1802.09941.

[5] Texas Advanced Computing Center. *Maverick2 User Guide - TACC User Portal*. https://portal.tacc.utexas.edu/user-guides/maverick2.

[6] Jianmin Chen, Rajat Monga, Samy Bengio, and Rafal Jozefowicz. Revisiting distributed synchronous sgd. In *International Conference on Learning Representations Workshop Track*, 2016.

[7] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *CoRR*, abs/1512.01274, 2015.

[8] Minsik Cho, Ulrich Finkler, and David Kung. Blueconnect: Novel hierarchical all-reduce on multi-tired network for deep learning, 2018.

[9] Adam Coates, Brody Huval, Tao Wang, David Wu, Bryan Catanzaro, and Ng Andrew. Deep learning with cots hpc systems. In Sanjoy Dasgupta and David McAllester, editors, *Proceedings of the 30th International Conference on Machine Learning*, volume 28.3 of *Proceedings of Machine Learning Research*, pages 1337–1345, Atlanta, Georgia, USA, 17–19 Jun 2013. PMLR.

[10] MPI contributors. *MPI: A Message-Passing Interface Standard*, 2015. https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf.

[11] IBM Corporation and Oak Ridge National Laboratory. *Summit - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband | TOP500 Supercomputer Sites*. https://www.top500.org/system/179397.

[12] Intel Corporation. *IntelÂő MPI Library | IntelÂő Software*. https://software.intel.com/en-us/mpi-library.

[13] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Commun. ACM*, 56(2):74–80, February 2013.

[14] Stephen Doherty. The impact of translation technologies on the process and product of translation. *International Journal of Communication*, 10:969, 02 2016.

[15] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings,*

11th European PVM/MPI Users' Group Meeting, pages 97–104, Budapest, Hungary, September 2004.

[16] Priya Goyal, Piotr Dollár, Ross B. Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch SGD: training imagenet in 1 hour. *CoRR*, abs/1706.02677, 2017.

[17] Aaron Harlap, Deepak Narayanan, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, and Phillip B. Gibbons. Pipedream: Fast and efficient pipeline parallel DNN training. *CoRR*, abs/1806.03377, 2018.

[18] K. Hazelwood, S. Bird, D. Brooks, S. Chintala, U. Diril, D. Dzhulgakov, M. Fawzy, B. Jia, Y. Jia, A. Kalro, J. Law, K. Lee, J. Lu, P. Noordhuis, M. Smelyanskiy, L. Xiong, and X. Wang. Applied machine learning at facebook: A datacenter infrastructure perspective. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 620–629, Feb 2018.

[19] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Identity mappings in deep residual networks. In *European conference on computer vision*, pages 630–645. Springer, 2016.

[20] Geoffrey Hinton, Li Deng, Dong Yu, George Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Brian Kingsbury, and Tara Sainath. Deep neural networks for acoustic modeling in speech recognition. *IEEE Signal Processing Magazine*, 29:82–97, November 2012.

[21] Qirong Ho, James Cipar, Henggang Cui, Jin Kyu Kim, Seunghak Lee, Phillip B. Gibbons, Garth A. Gibson, Gregory R. Ganger, and Eric P. Xing. More effective distributed ml via a stale synchronous parallel parameter server. In *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'13, pages 1223–1231, USA, 2013. Curran Associates Inc.

[22] Rankyung Hong and Abhishek Chandra. Decentralized distributed deep learning in heterogeneous wan environments. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, pages 505–505, New York, NY, USA, 2018. ACM.

[23] Rankyung Hong and Abhishek Chandra. Dlion: Decentralized distributed deep learning in micro-clouds. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, Renton, WA, July 2019. USENIX Association.

[24] Shlomo Hoory, Nathan Linial, and Avi Wigderson. Expander graphs and their applications. *Bull. Amer. Math. Soc. 43 (2006), 439-561*, 2006.

[25] Kevin Hsieh, Aaron Harlap, Nandita Vijaykumar, Dimitris Konomis, Gregory R. Ganger, Phillip B. Gibbons, and Onur Mutlu. Gaia: Geo-distributed machine learning approaching LAN speeds. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 629–647, Boston, MA, 2017. USENIX Association.

[26] Sylvain Jeaugey. Nccl 2.0. *GTC*, 2017.

[27] Xianyan Jia, Shutao Song, Wei He, Yangzihao Wang, Haidong Rong, Feihu Zhou, Liqiang Xie, Zhenyu Guo, Yuanzhou Yang, Liwei Yu, et al. Highly scalable deep learning training system with mixed-precision: Training imagenet in four minutes. *arXiv preprint arXiv:1807.11205*, 2018.

[28] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond data and model parallelism for deep neural networks. *CoRR*, abs/1807.05358, 2018.

[29] Jiawei Jiang, Bin Cui, Ce Zhang, and Lele Yu. Heterogeneity-aware distributed parameter servers. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 463–478, New York, NY, USA, 2017. ACM.

[30] Peng Jiang and Gagan Agrawal. Accelerating distributed stochastic gradient descent with adaptive periodic parameter averaging: Poster. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, PPoPP '19, pages 403–404, New York, NY, USA, 2019. ACM.

[31] A. Krizhevsky and G. Hinton. Learning multiple layers of features from tiny images. *Master's thesis, Department of Computer Science, University of Toronto*, 2009.

[32] Thorsten Kurth, Sean Treichler, Joshua Romero, Mayur Mudigonda, Nathan Luehr, Everett Phillips, Ankur Mahesh, Michael Matheson, Jack Deslippe, Massimiliano Fatica, et al. Exascale deep learning for climate analytics. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, page 51. IEEE Press, 2018.

[33] Mu Li. Scaling distributed machine learning with the parameter server. In *International Conference on Big Data Science and Computing*, page 3, 2014.

[34] Youjie Li, Mingchao Yu, Songze Li, Salman Avestimehr, Nam Sung Kim, and Alexander Schwing. Pipe-sgd: A decentralized pipelined sgd framework for distributed deep net training. In *Proceedings of the 32Nd International Conference on Neural Information Processing Systems*, NIPS'18, pages 8056–8067, USA, 2018. Curran Associates Inc.

[35] Xiangru Lian, Ce Zhang, Huan Zhang, Cho-Jui Hsieh, Wei Zhang, and Ji Liu. Can decentralized algorithms outperform centralized algorithms? a case study for decentralized parallel stochastic gradient descent. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 5330–5340. Curran Associates, Inc., 2017.

[36] Xiangru Lian, Wei Zhang, Ce Zhang, and Ji Liu. Asynchronous decentralized parallel stochastic gradient descent. In *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, pages 3049–3058, 2018.

[37] Liang Luo, Jacob Nelson, Luis Ceze, Amar Phanishayee, and Arvind Krishnamurthy. Parameter hub: a rack-scale parameter server for distributed deep neural network training. *CoRR*, abs/1805.07891, 2018.

[38] Qinyi Luo, Jinkun Lin, Youwei Zhuo, and Xuehai Qian. Hop: Heterogeneity-aware decentralized training. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, pages 893–907, New York, NY, USA, 2019. ACM.

[39] Krishna Giri Narra, Zhifeng Lin, Mehrdad Kiamari, Salman Avestimehr, and Murali Annavaram. Slack squeeze coded computing for adaptive straggler mitigation. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC âĂŹ19, New York, NY, USA, 2019. Association for Computing Machinery.

[40] Pitch Patarasuk and Xin Yuan. Bandwidth optimal all-reduce algorithms for clusters of workstations. *J. Parallel Distrib. Comput.*, 69(2):117–124, February 2009.

[41] Yanghua Peng, Yibo Zhu, Yangrui Chen, Yixin Bao, Bairen Yi, Chang Lan, Chuan Wu, and Chuanxiong Guo. A generic communication scheduler for distributed dnn training acceleration. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP âĂŹ19, page 16âĂŞ29, New York, NY, USA, 2019. Association for Computing Machinery.

[42] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In J. Shawe-Taylor, R. S. Zemel, P. L. Bartlett, F. Pereira, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 24*, pages 693–701. Curran Associates, Inc., 2011.

[43] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.

[44] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in TensorFlow. *arXiv preprint arXiv:1802.05799*, 2018.

[45] Xiaogang Shi, Bin Cui, Yingxia Shao, and Yunhai Tong. Tornado: A system for real-time iterative analysis over evolving data. In *Proceedings*

of the 2016 International Conference on Management of Data, SIGMOD '16, pages 417–430, New York, NY, USA, 2016. ACM.

[46] David Silver, Aja Huang, Christopher J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–503, 2016.

[47] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014.

[48] Linghao Song, Fan Chen, Youwei Zhuo, Xuehai Qian, Hai Li, and Yiran Chen. Accpar: Tensor partitioning for heterogeneous deep learning accelerator arrays. In *26th IEEE International Symposium on High Performance Computer Architecture, HPCA 2020, San Diego, CA, USA, February 22-26, 2020*, page to appear, 2020. to appear.

[49] Linghao Song, Jiachen Mao, Youwei Zhuo, Xuehai Qian, Hai Li, and Yiran Chen. Hypar: Towards hybrid parallelism for deep learning accelerator array. In *25th IEEE International Symposium on High Performance Computer Architecture, HPCA 2019, Washington, DC, USA, February 16-20, 2019*, pages 56–68, 2019.

[50] Peng Sun, Wansen Feng, Ruobing Han, Shengen Yan, and Yonggang Wen. Optimizing network performance for distributed dnn training on gpu clusters: Imagenet/alexnet training in 1.5 minutes. *arXiv preprint arXiv:1902.06855*, 2019.

[51] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Computer Vision and Pattern Recognition (CVPR)*, 2015.

[52] Hanlin Tang, Shaoduo Gan, Ce Zhang, Tong Zhang, and Ji Liu. Communication compression for decentralized training. In *NeurIPS*, 2018.

[53] Hanlin Tang, Xiangru Lian, Ming Yan, Ce Zhang, and Ji Liu. d^2: Decentralized training over decentralized data. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 4848–4856, StockholmsmÃďssan, Stockholm Sweden, 10–15 Jul 2018. PMLR.

[54] Jörg Tiedemann. Parallel data, tools and interfaces in opus. In Nicoletta Calzolari (Conference Chair), Khalid Choukri, Thierry Declerck, Mehmet Ugur Dogan, Bente Maegaard, Joseph Mariani, Jan Odijk, and Stelios Piperidis, editors, *Proceedings of the Eight International Conference on Language Resources and Evaluation (LREC'12)*, Istanbul, Turkey, may 2012. European Language Resources Association (ELRA).

[55] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 5998–6008. Curran Associates, Inc., 2017.

[56] Jianyu Wang and Gauri Joshi. Adaptive communication strategies to achieve the best error-runtime trade-off in local-update sgd. *ArXiv*, abs/1810.08313, 2018.

[57] Minjie Wang, Chien chin Huang, and Jinyang Li. Supporting very large models using automatic dataflow graph partitioning. 2018.

[58] Minjie Wang, Chien-chin Huang, and Jinyang Li. Supporting very large models using automatic dataflow graph partitioning. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys âĂŹ19, New York, NY, USA, 2019. Association for Computing Machinery.

[59] Eric P. Xing, Qirong Ho, Wei Dai, Jin-Kyu Kim, Jinliang Wei, Seunghak Lee, Xun Zheng, Pengtao Xie, Abhimanu Kumar, and Yaoliang Yu. Petuum: A new platform for distributed machine learning on big data. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '15, pages 1335–1344, New York, NY, USA, 2015. ACM.

[60] Masafumi Yamazaki, Akihiko Kasagi, Akihiro Tabuchi, Takumi Honda, Masahiro Miwa, Naoto Fukumoto, Tsuguchika Tabaru, Atsushi Ike, and Kohta Nakashima. Yet another accelerated sgd: Resnet-50 training on imagenet in 74.7 seconds. *arXiv preprint arXiv:1903.12650*, 2019.

[61] Kun-Hsing Yu, Andrew Beam, and Isaac Kohane. Artificial intelligence in healthcare. *Nature Biomedical Engineering*, 2, 10 2018.

[62] Ce Zhang and Christopher Ré. Dimmwitted: A study of main-memory statistical analytics. *Proceedings of the VLDB Endowment*, 7(12):1283–1294, 2014.