Heterogeneity-Aware Distributed Machine Learning Training via Partial Reduce

Xupeng Miao, Xiaonan Nie, Yingxia Shao, Zhi Yang, Jiawei Jiang, Lingxiao Ma, Bin Cui
[1,2,4,6,7] Department of Computer Science & Key Lab of High Confidence Software Technologies (MOE), Peking University
[7]Institute of Computational Social Science, Peking University (Qingdao)
[3]School of Computer Science, BUPT, [5]ETH Zurich [1]Tencent Inc.
[1,2,4,6,7]{xupeng.miao, xiaonan.nie, yangzhi, xysmlx, bin.cui}@pku.edu.cn [3]shaoyx@bupt.edu.cn [5]jiawei.jiang@inf.ethz.ch

ABSTRACT

All-reduce is the key communication primitive used in distributed data-parallel training due to the high performance in the homogeneous environment. However, All-reduce is sensitive to stragglers and communication delays as deep learning has been increasingly deployed on the heterogeneous environment like cloud. In this paper, we propose and analyze a novel variant of all-reduce, called partial-reduce, which provides high heterogeneity tolerance and performance by decomposing the synchronous all-reduce primitive into parallel-asynchronous partial-reduce operations. We provide theoretical guarantees, proving that partial-reduce converges to a stationary point at the similar sub-linear rate as distributed SGD. To enforce the convergence of the partial-reduce primitive, we further propose a dynamic staleness-aware distributed averaging algorithm and implement a novel group generation mechanism to prevent possible update isolation in heterogeneous environments. We build a prototype system in the real production cluster and validate its performance under different workloads. The experiments show that it is 1.21×-2× faster than other state-of-the-art baselines.

CCS CONCEPTS

• Information systems → Data management systems; **• Computing methodologies → Machine learning**; **Distributed computing methodologies**.

KEYWORDS

Distributed machine learning, Heterogeneity, All-Reduce

ACM Reference Format:
Xupeng Miao, Xiaonan Nie, Yingxia Shao, Zhi Yang, Jiawei Jiang, Lingxiao Ma, Bin Cui. 2021. Heterogeneity-Aware Distributed Machine Learning Training via Partial Reduce. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21), June 18–27, 2021, Virtual Event, China.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3448016.3452773

1 INTRODUCTION

The success of modern machine learning (ML) lays the foundation of increasing amounts of data [11], emerging elaborate machine learning models [42], and the development of computing infrastructure [32]. With the increasing size of data, training the ML models on the massive data is becoming more time-consuming [12]. Distributed ML, which accelerates the training by introducing multiple computing nodes, gradually becomes an attractive and necessary solution in both academia and industry. There are many distributed ML systems intensively studied by database and system communities, such as Vertica-ML [14], SketchML [21], PS2 [44], PyTorch [33] and TensorFlow [3]. Most of the existing systems rely on a data-parallel computation schema and follow a distributed mini-batch SGD algorithm [24], where the model is trained over different parts of the dataset, and the model parameters are synchronized at the end of each iteration (i.e., update). The synchronization operation guarantees that the distributed training paradigm can achieve the same convergence property as the sequential training counterpart. The existing methods typically assume a homogeneous environment — there is no significant straggler in the computing nodes that might block the others. However, the homogeneous assumption is not always available in reality. Heterogeneous environments are ubiquitous in real-world applications:

Case 1: (Communication heterogeneity) Due to the massive amounts of data, many large companies (e.g., Google, Microsoft and Tencent) choose to store them on tens of geo-distributed data centers. The communication within a data center could be ten times faster than between data centers [18]. Even in the same data center, heterogeneity still exists because of different network interface cards, network PCIe switches, and hierarchical network topology.

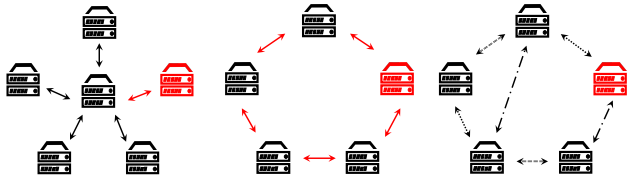
Case 2: (Hardware heterogeneity) Deep learning (DL) is compute-intensive and hence heavily relies on powerful but expensive accelerators (e.g., GPU). Since these accelerators are developing quickly, different generations of hardware could have quite distinct computation power. Collaborative training with heterogeneous clusters [4, 46] is much more cost-effective than replacing all outdated devices.

Case 3: (Resource sharing) A common practice in large companies nowadays is to utilize cluster schedulers, such as Kubernetes [7], to manage clusters of tens of thousands of CPU/GPU resources. The jobs could differ from the others in terms of memory usage, CPU/GPU core utilization, network bandwidth, and so on. To improve the cluster utilization, multiple containers could be placed on the same physical node or even the same physical GPU by virtualization. As a result, the containers of the same job could perform distinct hardware efficiency due to the resource sharing [41].

Several works have targeted this heterogeneous problem and proposed methods to solve it by introducing asynchrony during the synchronization so that the faster workers do not need to wait

SIGMOD '21, June 18–27, 2021, Virtual Event, China

ACM ISBN 978-1-4503-8343-1/21/06…$15.00
https://doi.org/10.1145/3448016.3452773



(a) Parameter server (b) All-Reduce (c) Partial-Reduce ($P = 2$)

Figure 1: Illustration of different architectures in heterogeneous environment. The red color represents for the effects from the straggler. The communication manners in partial reduce are adaptively changing in different iterations.

for the stragglers [8, 10, 17, 27]. These works are mostly built on top of a widely used distributed ML system architecture — parameter server (PS) [24]. PS provides a key-value based central storage of the global model parameters and flexible model accessing interfaces (i.e., `pull` and `push`). The architecture of PS matches the requirements of asynchronous communication, where each worker calculates its gradients and updates the shared model without waiting for the stragglers. However, most PS systems store models in main memory and manage them via CPUs. Therefore they cannot benefit from inter-GPU communication channels when DL models are evolving over GPUs. Instead of using PS, a range of existing deep learning frameworks (e.g., PyTorch) use All-Reduce [34] communication scheme and establish inter-GPU message passing to help GPU training. However, the heterogeneous problem is rarely studied in the All-Reduce system. This is not surprising because the existing All-Reduce primitives are fundamentally synchronous and render them fragile to stragglers, yielding the question we try to answer in this paper: *can we alleviate the efficiency degradation in the heterogeneous environment over the All-Reduce architecture while maintaining the best possible convergence performance?*

In this paper, we propose a novel primitive *partial reduce* (P-Reduce), where each worker only needs to synchronize among P random workers without waiting or blocking any other workers in the current iteration, as shown in Figure 1. As training proceeds, the local model from each worker gradually propagates to the others, and thus models at different workers converge collaboratively to the same optimal point. We theoretically prove that the convergence rate of partial reduce is $O(1/\sqrt{PK})$, where K is the number of iterations, similar to that of vanilla distributed SGD. We also notice the delayed model from the stragglers in the heterogeneous environment might be harmful to the statistical efficiency [43]. Consequently, we propose a dynamic partial reduce (i.e., weighted sum with dynamic weights) mechanism to to aggregate the models, instead of the constant partial reduce (i.e., directly model average). We implement a prototype and evaluate it on different workloads under both manually prepared heterogeneous environments and real-world productive clusters. The experimental results show our partial reduce can be at most 16.6× faster than the state-of-the-art baselines per iteration, and around 2× in total run time.

We summarize our main contributions as follows:

- We target the disadvantages of the All-Reduce training method in heterogeneous environments and propose a novel **partial reduce** algorithm for efficient distributed training.

Algorithm 1 Distributed mini-batch SGD Algorithm

Require: Initial model x_0, learning rate γ, batch size M, and total number of iterations K.

Worker $i = 1, \ldots, N$ in parallel:

1: **for** $k = 1, 2, \ldots, K$ **do**
2: Randomly sample a batch $\xi_k^i := (\xi_{k,1}^i, \xi_{k,2}^i, \ldots, \xi_{k,M}^i)$ from local data of the i-th worker.
3: Compute the stochastic gradient locally $g_k(\mathbf{x}_k; \xi_k^i) := 1/M \sum_{j=1}^{M} \nabla f(\mathbf{x}_k; \xi_{k,j}^i)$.
4: Average local gradients by $\Delta\mathbf{x}_k \leftarrow 1/N \sum_{j=1}^{N} g_j(\mathbf{x}_k; \xi_k^j)$.
5: Update the model $\mathbf{x}_{k+1} \leftarrow \mathbf{x}_k - \gamma\Delta\mathbf{x}_k$.

- We theoretically analyze the convergence property of partial reduce and obtain a convergence rate of $O(1/\sqrt{PK})$.
- We capture the staleness in the training process and propose *constant/dynamic partial reduce* mechanisms for better convergence performance.
- Experiments on the production heterogeneous cluster demonstrate the significant efficiency improvement of our method on different workloads.

2 PRELIMINARIES

2.1 Distributed SGD

Distributed machine learning relies on the distributed mini-batch SGD algorithm, as shown in Algorithm 1. We suppose that the input is the training dataset ξ and ξ_i represents the i-th data sample. The target is to find a model $\mathbf{x} \in \mathbb{R}^d$ (d is the total number of parameters in the model) that minimizes the empirical risk as follows:

$$\min_{\mathbf{x}} \Big[F(\mathbf{x}) := \frac{1}{|S|} \sum_{i} f(\mathbf{x}; \xi_i) \Big] \tag{1}$$

where $f(\cdot)$ is the loss function defined by the learning model. Mini-batch SGD samples a batch of data and compute the stochastic gradients to update the model parameters in each iteration. In the distributed setting, the total N workers compute stochastic gradients in parallel. The update rule is:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \gamma \Big[\frac{1}{N} \sum_{i=1}^{N} g_k(\mathbf{x}_k; \xi_k^i) \Big], \tag{2}$$

where γ is the learning rate, ξ_k^i are randomly sampled from the training set, and $g_k(\mathbf{x}_k; \xi_k^i)$ denotes the gradient from the i-th worker.

2.2 Communication Mechanism

PS based centralized training Distributed ML systems have been extensively studied in recent years to scale up ML for big data and large models. PS [24] is a trendy data parallelism architecture and distributes the calculation onto different machines, and various workers hold different training samples. The PS stores a global model, and each worker fetches the model from the PS, computes a mini-batch of stochastic gradients, and pushes them to the PS to update the global model under certain consistency protocol. Bulk synchronous parallel algorithms (BSP) [13, 15, 23] assume that all workers are fully synchronized, and the convergence rate is proved to be $O(1/\sqrt{NK})$, where N is the number of workers and K is

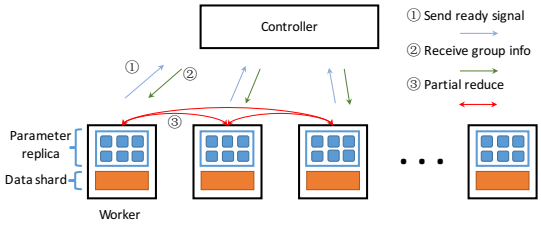


Figure 2: Overview of partial reduce execution.

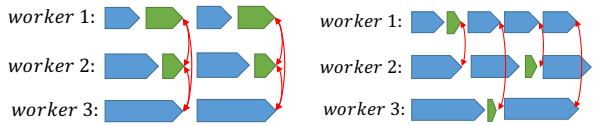
the number of iterations. In contrast, Asynchronous parallel algorithms (ASP) allow the workers to proceed without waiting for each other. It has been proved that ASP shares the same convergence rate with BSP when the staleness is upper bounded [27]. The existing PS-based approach mostly performs a centralized design, and PS may become a communication bottleneck and slow down the convergence.

Collective operation based decentralized training Another line of approach is decentralized training, which enables carefully scheduled the point-to-point communication between workers without suffering from the central communication bottleneck. All-Reduce is one of the most representative decentralized training methods and uses high-performance communication techniques for model synchronization, such as Ring-Reduce, assuming the workers are connected with a ring network. Recently, several distributed DL systems adopt All-Reduce, such as Horovod [35] and Parallax [22], show superior performance over PS under the same network bandwidth condition, especially for dense models. D-PSGD [28] is another decentralized training method, allows each worker synchronize models with its neighbor and has similar convergence property as All-Reduce. AD-PSGD [29, 31] further relaxes the consistency of model averaging for hardware efficiency.

2.3 Heterogeneous Training

Existing studies. In heterogeneous environments, both PS and collective operation methods rely on a fixed communication topology (i.e., between workers and PS or between workers themselves), which may be susceptible to heterogeneity. For PS, Stale Synchronous Parallel (SSP) [17] has been proposed that the fastest worker cannot exceed the slowest more than a predefined staleness. Heterogeneity aware PS [20] further involves a dynamic learning rate mechanism to handle the delayed gradients. Backup workers [3, 8] has been proposed to alleviate the straggler problem in PS by dropping the gradients from the slowest workers. For the collective settings, the communication pattern is more restrictive and suffers more from the stragglers. Eager-Reduce [25] with partial collective operations relaxes the global synchronization by permitting accumulated/empty gradients but suffering from the delayed gradients.

Heterogeneity modeling. As we introduced before, system heterogeneity may come from different aspects (e.g., communication, hardware, resource sharing), even the imbalanced workload partition can also lead to heterogeneity. A typical result of the heterogeneity is the different time costs on a single update among the workers. Our theoretical analysis assumes that the random distributions of the per-update time from these workers are independent. We consider both synthetic and real-world productive heterogeneous environments in our evaluations.



(a) All-Reduce (b) Partial-Reduce ($P = 2$)

Figure 3: Comparison of the idle time (in green).

3 PARTIAL REDUCE

In this section, we improve the the ML convergence performance in two aspects, including *hardware efficiency* and *statistical efficiency* [43]. We target the hardware efficiency degradation of All-Reduce on heterogeneous settings and propose the partial reduce algorithm. We also make a theoretical analysis of the convergence property of our method. Considering the delayed model parameters from the stragglers, we propose a dynamic partial reduce mechanism to achieve better statistical efficiency.

3.1 Constant Partial Reduce

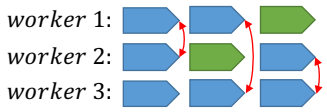
3.1.1 P-reduce. To reduce the idle time of each worker, our intuition is to relax the fully synchronous requirement during the All-Reduce training iterations. As shown in Fig. 3, we propose a new primitive *Partial Reduce (P-Reduce)* to synchronize among P workers, rather than waiting for all the other workers so that the training process will continue even if part of the workers slow down. We present our method in Fig. 2 and Alg. 2. In P-Reduce, each worker randomly samples a batch (lines 1-2) and execute computation on an initial model replica (lines 3-4). When the gradient updates are finished, the worker sends a ready signal to the controller (line 5). The controller has a producer-consumer queue to collect these signals. Once enough (i.e., P) workers are ready, the controller pops P signals from the queue and informs them to execute a partial reduce in this temporary worker group. In *constant partial reduce*, we perform a simple model average among P workers so that the model aggregation weight is set to be a constant $1/P$ (lines 6-7).

Compared to the original All-Reduce, P-Reduce preserves the communication bandwidth utilization while avoids the global barrier. Unlike the global model average per iteration in All-Reduce, P-Reduce narrows the size of the synchronization group and breaks it into a sequence of partial synchronization steps. Note that multiple worker groups could execute partial reduce in parallel if they are permitted by the controller. Each worker can step into the next iteration without waiting for the other workers to finish updating. The computation and communication can be done in parallel, which means that the computation/communication time of a worker group might hide that of another group. Due to the random arrivals of the ready signals on the controller, each worker has the chance to perform model average with the others as the iteration goes on.

3.1.2 Global view formalization. By counting each partial reduce as one iteration, the update rule of each iteration can be viewed as:

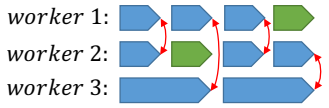
$$\mathbf{X_{k+1}} = (\mathbf{X_k} - \eta \mathbf{G}_k)\mathbf{W}_k, \tag{3}$$

where η is the learning rate, matrices $\mathbf{X_k}$ and $\mathbf{G_k}$ contain the local model vector $\mathbf{x}_k^i$ and gradient vector $\mathbf{g}(\mathbf{x}_k^i)$ of each worker i at the kth iteration, and $\mathbf{W}_k$ is the synchronization matrix used for model averaging (via P-reduce operation).



$$\mathbb{E}(\mathbf{W}_k) = \frac{1}{3}\underbrace{\begin{bmatrix}1/2 & 1/2 & 0\\ 1/2 & 1/2 & 0\\ 0 & 0 & 1\end{bmatrix}}_{\text{update}(\mathbf{x}^1,\mathbf{x}^2)} + \frac{1}{3}\underbrace{\begin{bmatrix}1/2 & 0 & 1/2\\ 0 & 1 & 0\\ 1/2 & 0 & 1/2\end{bmatrix}}_{\text{update}(\mathbf{x}^1,\mathbf{x}^3)} + \frac{1}{3}\underbrace{\begin{bmatrix}1 & 0 & 0\\ 0 & 1/2 & 1/2\\ 0 & 1/2 & 1/2\end{bmatrix}}_{\text{update}(\mathbf{x}^2,\mathbf{x}^3)} = \begin{bmatrix}2/3 & 1/6 & 1/6\\ 1/6 & 2/3 & 1/6\\ 1/6 & 1/6 & 2/3\end{bmatrix} \quad \rho = 0.5$$

(a) Homogeneous environment



$$\mathbb{E}(\mathbf{W}_k) = \frac{1}{4}\underbrace{\begin{bmatrix}1/2 & 1/2 & 0\\ 1/2 & 1/2 & 0\\ 0 & 0 & 1\end{bmatrix}}_{\text{update}(\mathbf{x}^1,\mathbf{x}^2)} + \frac{1}{4}\underbrace{\begin{bmatrix}1/2 & 0 & 1/2\\ 0 & 1 & 0\\ 1/2 & 0 & 1/2\end{bmatrix}}_{\text{update}(\mathbf{x}^1,\mathbf{x}^3)} + \frac{1}{4}\underbrace{\begin{bmatrix}1/2 & 1/2 & 0\\ 1/2 & 1/2 & 0\\ 0 & 0 & 1\end{bmatrix}}_{\text{update}(\mathbf{x}^1,\mathbf{x}^2)} + \frac{1}{4}\underbrace{\begin{bmatrix}1 & 0 & 0\\ 0 & 1/2 & 1/2\\ 0 & 1/2 & 1/2\end{bmatrix}}_{\text{update}(\mathbf{x}^2,\mathbf{x}^3)} = \begin{bmatrix}5/8 & 1/4 & 1/8\\ 1/4 & 5/8 & 1/8\\ 1/8 & 1/8 & 3/4\end{bmatrix} \quad \rho = 0.625$$

(b) Heterogeneous environment

Figure 4: Illustration of ρ under different environments. We suppose $N = 3$, $P = 2$, the blue arrow blocks represent an iteration, the green arrow blocks represent the idle time and the red double arrows represent the partial reduce.

According to the Algorithm 2 (line 6-7 of the worker component), the kth partial reduce operation only involves a worker group $\mathcal{S}_k$ containing P workers selected by the controller, and the corresponding synchronization matrix $\mathbf{W}_k$ can be defined as:

$$\mathbf{W}_k(ij) = \begin{cases} 1/P, & \text{if workers } i, j \in \mathcal{S}_k \\ 1, & \text{if worker } i \notin \mathcal{S}_k \text{ and } i = j, \\ 0, & \text{otherwise} \end{cases} \tag{4}$$

It worth to note that in the worker component our algorithm, the local update (line 4) can be exchanged logically with the communication to the controller (line 5-6) due to their asynchronous nature. Equivalently, the gradient matrix $\mathbf{G}_k$ can be defined as follows:

$$\mathbf{G}_k(i) = \begin{cases} \mathbf{g}(\mathbf{x}_k^i; \xi_k^i), & \text{if worker } i \in \mathcal{S}_k \\ \mathbf{0}, & \text{otherwise} \end{cases} \tag{5}$$

According to Algorithm 2 (line 4 of the controller component), $\mathcal{S}_k$ (and thus $\mathbf{W}_k$) is independent on of the data samples ξ_k at the kth iteration, and only depends on the arrival of ready signals from workers to request the kth group. Moreover, the speed of workers to generate ready signals could vary significantly in cloud environment due to resource sharing and network latency, leading to high dynamics and randomness of forming groups at different iterations. This makes $\mathbf{W}_k$ *largely* uncorrelated with k.

3.2 Theoretical Analysis

3.2.1 Assumptions. We now provide theoretical analysis for the update rule given in E.q. (3) with variable synchronization matrix $\mathbf{W}_k$ defined in E.q. (4). We make the following commonly used assumptions [6, 29, 30, 39]:

Assumption 1.

(1) **Lipschitzian gradient:** $\|\nabla F(\mathbf{x}) - \nabla F(\mathbf{y})\| \leqslant L\|\mathbf{x} - \mathbf{y}\|$
(2) **Unbiased estimation:** $\mathbb{E}_{\xi|\mathbf{x}}[g(\mathbf{x})] = \nabla F(\mathbf{x})$
(3) **Bounded variance:** $\mathbb{E}_{\xi|\mathbf{x}}[g(\mathbf{x}) - \nabla F(\mathbf{x})] \leq \sigma^2$

Note that the unbiased estimation is satisfied through the distributed file system (e.g., HDFS) or storing a portion of dataset and shuffling the local data among the workers periodically [9, 47].

Assumption 2.

(1) **Stochastic averaging:** $\mathbf{W}_k$ is doubly stochastic for all k, i.e., $\mathbf{W}_k = \mathbf{W}_k^\top$, $\mathbf{W}_k \mathbf{1}_N = \mathbf{1}_N$.

Algorithm 2 Partial Reduce Algorithm

Require: Local models $\{\mathbf{x}_0^i\}_{i=1}^N$ with the same initialization, learning rate γ, batch size M, and total number of iterations K.

Worker $i = 1, \ldots, N$ in parallel:
1: **for** $k = 1, 2, \ldots, K$ **do**
2: Randomly sample a batch $\xi_k^i := (\xi_{k,1}^i, \xi_{k,2}^i, \ldots, \xi_{k,M}^i)$ from local data of the i-th worker.
3: Compute the stochastic gradient locally $g_k(\mathbf{x}_k^i; \xi_k^i) := 1/M \sum_{j=1}^M \nabla f(\mathbf{x}_k^i; \xi_{k,j}^i)$.
4: Update the local model $\mathbf{x}^i \leftarrow \mathbf{x}_k^i - \gamma g_k(\mathbf{x}_k^i; \xi_k^i)$.
5: Send ready signal i to the controller.
6: Receive a partial reduce group $\mathcal{S} := [i_1, i_2, \ldots, i_P]$.
7: Aggregate local models by $\mathbf{x}_{k+1}^i \leftarrow 1/P \sum_{j=1}^P \mathbf{x}^{i_j}$.
8: Output the average of the models on all workers for inference.

Controller:
1: $Q = Queue()$
2: **while** True **do**
3: **if** $Q.size() \geq P$ **then**
4: $\mathcal{S} \leftarrow$ pop P elements $[i_1, i_2, \ldots, i_P]$ from Q.
5: Send the partial reduce group to the workers in G.
6: Receive a ready signal i.
7: $Q.push(i)$.

(2) **Dependence of random variables:** $\mathbf{W}_k$ is a random variable independent on ξ_k and k.
(3) **Spectral gap:** There exists a $\rho \in [0, 1)$ such that

$$\max\{|\lambda_2(\mathbb{E}[\mathbf{W}_k])|, |\lambda_N(\mathbb{E}[\mathbf{W}_k])|\} \leq \rho, \forall k, \tag{6}$$

where λ_i is the i-th largest eigenvalue and $\mathbb{E}(\mathbf{W}_k) = \frac{1}{K}\sum_{k=1}^K \mathbf{W}_k$. For example, Figure 4(a) is a homogeneous environment where all workers have the same speed. There are three possible communication groups: $(\mathbf{x}^1, \mathbf{x}^2)$, $(\mathbf{x}^2, \mathbf{x}^3)$, and $(\mathbf{x}^1, \mathbf{x}^3)$ with equal probability (i.e., 1/3). Therefore, we derive the corresponding $\mathbb{E}(\mathbf{W}_k)$ and $\rho = 0.5$.

3.2.2 P-Reduce Convergence Property. Suppose $\mathbf{u}_k = \frac{1}{N}\sum_{i=1}^N \mathbf{x}_k^i$ is the average of N local models. We propose an upper bound of the convergence rate as follows.

Theorem 1 (**Convergence of Partial Reduce**). *We assume the bound of gradient variance σ^2 is in inverse proportion to the*

mini-batch size M. We define $\bar{\rho} = \frac{\rho}{1-\rho} + \frac{2\sqrt{\rho}}{(1-\sqrt{\rho})^2}$. *For partial reduce with P, under Assumptions 1–2, if the learning rate satisfies*

$$\eta L + \frac{2N^3\eta^2\bar{\rho}}{P^2} \leq 1, \tag{7}$$

where $\eta = \frac{P}{N}\gamma$, *and all local models are initialized at a same point* $\mathbf{u}_1$, *then the average-squared gradient norm after K iterations is bounded as follows*

$$\mathbb{E}\Big[\frac{1}{K}\sum_{k=1}^{K}\|\nabla F(\mathbf{u}_k)\|^2\Big] \leq \underbrace{\frac{2[F(\mathbf{u}_1)-F_{inf}]}{\eta K} + \frac{\eta L\sigma^2}{P}}_{\text{SGD error}} + \underbrace{\frac{2\eta^2L^2\sigma^2N^3\bar{\rho}}{P^2}}_{\text{network error}} \tag{8}$$

The detailed proof of Theorem 1 is provided in [2]. We note the update rule given by equation (3) can be connected with the cooperative SGD framework [39] by treating the local models of workers out of a partial reduce group $\mathcal{S}_k$ as the auxiliary variables in each iteration k. We prove our theorem by extending the cooperative SGD framework to enable variable model-average matrix $\mathbf{W}_k$. The theorem shows that the convergence bound can be decomposed into two parts: 1) The first two items come from the distributed SGD training, which is similar to that of the vanilla SGD [6]. When we increase P, the upper bound could become more substantial and suggest us to give more iterations for convergence. 2) The last item is the network error, resulted from the model difference among the workers during training. Based on these, if the learning rate is $\gamma = N/(L\sqrt{PK})$ and K is sufficiently large, we further derive that the convergence rate becomes $O(1/\sqrt{PK})$, which is similar to that of BSP. A smaller ρ means faster update spreading in the network, leading to better convergence.

Figure 4 shows how the heterogeneous environment affects the spectral gap ρ. As we can see, if $N = 3$ and $P = 2$ in partial reduce, we have $\rho = 0.5$ in the homogeneous setting (Figure 4(a)). We further suppose one worker of them are two times slower than the others, resulting in a larger $\rho = 0.625$ (Figure 4(b)). We conclude that the more heterogeneous the environment is, the smaller spectral gap $1-\rho$ and the larger network error bound we have. Specially, when the environment is homogeneous and the model updated are merged via AllReduce (i.e., $P = N$), all elements in $\mathbb{E}[\mathbf{W_k}]$ are $1/P$ and $\rho = 0$. In this case, the network error decreases to zero.

3.3 Dynamic Partial Reduce

In constant partial reduce, we use a default model aggregation weight $1/P$. In the following, we discuss its limitations and propose a novel mechanism — *dynamic partial reduce* to aggregate the models.

3.3.1 Limitations of constant partial reduce. We first describe an abstract example to illustrate the staleness during our partial reduce. Suppose different workers start from the same model initialization, the slow workers might meet other fast workers after different iterations. As shown in Fig. 5, we assume $P = 2$ and worker j is three times slower than worker i. When worker j finish the first iteration, worker i is already very close to the optimal point. However, $\mathbf{x}_i$ has to be averaged with the delayed model $\mathbf{x}_j$ in constant partial reduce, which leads to a model degradation on worker i. Therefore, it is still worthy to study the model aggregate rule to prevent the impacts from the staleness.

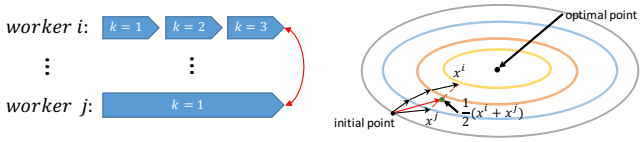


Figure 5: Limitations of constant partial reduce.

3.3.2 Staleness-aware heuristic algorithm. Our intuition is to penalize the stale model parameters during the model aggregation step — the more substantial the staleness, the smaller weights. We are inspired by the exponential moving average (EMA), which has been successfully applied in deep learning model training to increase the generalization performance and model robustness [36, 38, 40, 45]. This technique basically adds up the current model and historical models during training, and the weights of past models are exponential decayed. The model parameters with EMA after k iterations is $\bar{\mathbf{x}}_k = \alpha\bar{\mathbf{x}}_{k-1} + (1-\alpha)\mathbf{x}_k$, where $\alpha \in [0, 1)$ and $\bar{\mathbf{x}}_0 = 0$. It can be implemented highly efficiently since it needs no iterative optimization nor extra memory. The result can be easily rewritten as $\bar{\mathbf{x}}_k = (1-\alpha)\sum_{i=1}^{k}\alpha^{k-i}\mathbf{x}_i$. In practice, we have

$$\bar{\mathbf{x}}_k = \sum_{i=1}^{k}\beta_i\mathbf{x}_i, \quad \beta_i = \frac{(1-\alpha)\alpha^{k-i}}{1-\alpha^k}, \tag{9}$$

where β_i are the normalized weights and the denominator $1-\alpha^k$ is the bias correction.

3.3.3 Algorithm implementation. In dynamic partial reduce, each worker could send its current iteration number to the controller while sending the ready signal. The controller collects P iteration numbers ($\{k_1, k_2, ..., k_P\}$) in descending order from the workers in a group. We define the relative iteration number $\hat{k}_i = \max_{1\leq j\leq P} k_j - k_i + 1$ and the range of $\hat{k}_i$ is $[1, \hat{k}_{\max}]$. To distribute an exponential decay β_i for each worker, we replace k in Eq. (9) with $\hat{k}_i$. There is a strong possibility that not every relative iteration number in $[1, \hat{k}_{\max}]$ could be sent from the P workers. The application of the exponential moving average (EMA) approach with Eq. (9) requires storing and managing historical versions of models (i.e., $\{\mathbf{x}_i\}_{i=1}^{k}$) for consecutive iterations. To avoid such costs incurred by EMA, we consider a conservative approximation of using the initial model x_1 to approximate all the intermediate model versions that have not been stored for the EMA-based aggregation. Other approximation strategies are also possible, such as approximate intermediate model to the version of the closest iteration number. If some workers have the same relative iteration number, their weights are divided equally. In this way, all $\hat{k}$ in $[1, \hat{k}_{\max}]$ has specific model parameters so that we can apply Eq. (9) successfully. Note that after the partial reduce, the workers in the group should update their current iteration k with $\max_{1\leq j\leq P} k_j$, because all of their models are the latest.

4 IMPLEMENTATION

In this section, we describe the implementation details of our prototype system. Generally, we follow the traditional data-parallel distributed training, and each worker handles a subset of the whole training data by data sharding approach. For the model parameters, we deploy a model replication on each worker with the same initialization. Each worker could be a physical computation node or a virtual instance, determined by specific hardware environments.

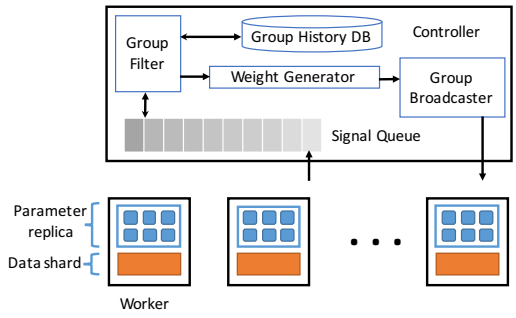


Figure 6: Illustration of the controller architecture.

Components. As shown in Fig. 6, we break down the key components in our prototype — controller, consisting of a signal queue, the group filter, the group history database, the weight generator, and the group broadcaster. The workers send their ready signals to the controller when they are ready for partial reduce. The *signal queue* collects these signals and sends them to the *group filter* in the FIFO order. The group filter fetches P signals each time and verifies if these workers could compose an appropriate group based on the records in the *group history database*. The weight generator obtains the group and generates the model aggregation weights based on the specific partial reduce algorithm. Finally, the group broadcaster send back these information to these workers in the group.

Group frozen avoidance. Our proposed method relies on the sequential partial reduce operations among random worker groups to spread the latest model parameters to all the workers. In most cases, the group filter could directly adopt the groups without any side effects. However, in rare cases, the worker group could freeze the range of the partial reduce inside the current group and prevent the communication from outside workers. Suppose there are four workers and P = 2, worker 1 always averages only with worker 2, and worker 3 always averages only with worker 4, workers could still reach consensus under the given assumptions. Apparently, such isolated groups lead to two independent training processes and each uses half of the cluster and wastes resources.

To tackle this problem, we construct a sync-graph by connecting workers in the same group for recent T P-reduce groups and use the group filter to check whether the sync-graph is connected. If the group frozen (i.e., disconnected sync-graph) is detected, the group filter will interact with the signal queue and add few edges (i.e., forming groups) between connected components to make the sync-graph connected. Notice that each P-reduce operation adds $P-1$ links over the sync-graph of N nodes, we require $T \geq \lceil \frac{N-1}{P-1} \rceil$ which is the minimum number to make the graph connected under random group formation. If the graph is not connected, we could identify the isolation.

Prototype. Our prototype is built on top of PyTorch, and we utilize the partial reduce primitive with `torch.distributed` communication package, including `send`, `receive`, `broadcast`, `all_reduce`. We use Gloo as the backend to support both CPU and GPU clusters (other backends, e.g., NCCL are also possible). Besides, we also implement a message queue with TCP/IP protocols for the communication between the controller and the workers. Note that, unlike the parameter server suffering from the central node bottleneck, the controller doesn't store any model parameters or gradients. Each message from the workers is only a few bytes so that it will not involve any communication overheads. The involved communication worlds are constructed before the training process starts and reused during training to avoid the construction overheads.

Our partial reduce operations for each parameter matrix are after all gradients have been computed for a batch in our current implementation. Some optimized distributed library, such as `DistributedDataParallel` (DDP) [26], could provide overlapping between communication and the backward computation. Unfortunately, it requires a fixed communication world (e.g., `init_process_group` in PyTorch) during the training process, limiting its application for the partial reduce with dynamic worker groups. We leave it as the future work and expect relative benefits of partial reduce still holds in the setting with overlapping computation.

5 EXPERIMENTS

5.1 Experimental Setup

In this section, we compare our prototype system with the following baselines, including All-Reduce, Eager-Reduce [25], AD-PSGD [29], PS BSP, PS ASP, PS HETE [20] and PS BK [3, 8].

Datasets and models. We choose three image classification datasets in our experiments, including CIFAR10 (10 classes), CIFAR100 (100 classes) and the largest benchmark dataset ImageNet (1000 classes). We evaluate our system and baselines on three different kinds of CNNs: ResNet [16], VGG [37] and DenseNet [19].

Experimental setting. We implement all of these models in PyTorch 1.5.0 and select SGD optimizer with the learning rate of 0.1, the batch size of 256, the momentum of 0.9 and the weight decay of 1e-4. We evaluate them on a GPU cluster, and each node is equipped with 376 GB RAM, 24 cores, 8 Nvidia Tesla V100 32 GB cards, and 10 GB Ethernet. The testing accuracy thresholds of convergence are set to be 90% for the CIFAR10 dataset and 70% for the CIFAR100 dataset, as reported in [19]. For the ImageNet dataset, to obtain the standard terminated test accuracy as reported in [16], we adopted similar learning rate tuning scheme as proposed in [1], i.e., start from 0.1 and decay by 10 every 20 epochs. All experiments are executed five times, and the averaged results are reported.

5.2 End-to-End Comparison

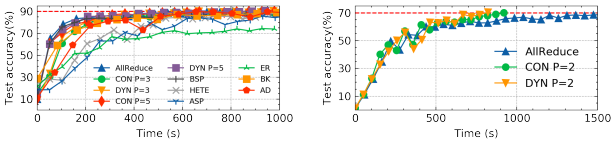
We first conduct an end-to-end comparison with all baselines on three models on CIFAR10. To simulate different heterogeneity conditions, we create the synthetic heterogeneous environment by selecting HL (out of N) workers to share a single physical GPU, and deploying other individual workers on $N - HL$ independent GPUs. Due to resource (e.g., GPU cores, PCIe bandwidth) sharing, the workers on the same GPU might be slower than the other workers. Specially, if the heterogeneity level (HL) is 1, it becomes homogeneous because each GPU is monopolized by a worker.

To measure the end-to-end performance, we use the total run time (in seconds) for the same convergence threshold. We further decouple the end-to-end performance into statistical efficiency and hardware efficiency and analyze them with different metrics. For the statistical efficiency, we use the number of updates (i.e., iterations) until the convergence. For the hardware efficiency, we use the average time per update takes.

5.2.1 Comparison with PS-based approaches. As shown in Table 1, PS BSP performs slightly slower than partial reduce in terms of total

Table 1: End-to-End comparison on CIFAR10. CON and DYN are constant and dynamic partial reduce in abbreviation. N/A represents that they cannot converge to the threshold. We use the bold font to mark the metrics for methods with the optimal run time. AR: All-Reduce, ER: Eager-Reduce, AD: AD-PSGD, BK: synchronous SGD with 3 backup workers

Model	Metrics	HL	Collective Operation			Parameter Server				Partial Reduce (P=3)		Partial Reduce (P=5)	
			AR	ER	AD	BSP	ASP	HETE	BK	CON	DYN	CON	DYN
ResNet-34	run time (s)	1	530	N/A	482	605	768	756	588	423	**393**	448	456
		3	1150		731	1204	996	861	734	630	**603**	639	658
	#update	1	1226		6342	1275	10213	9935	1482	3030	2706	1500	1484
		3	1425		7029	1438	10335	8789	1556	3209	3017	1607	1621
	per-update time (s)	1	0.432	0.217	0.076	0.475	0.075	0.076	0.397	0.140	0.145	0.299	0.307
		3	0.807	0.349	0.104	0.837	0.096	0.098	0.472	0.196	0.200	0.398	0.406
VGG-19	run time (s)	1	580	N/A	582	734	1099	757	732	493	**398**	499	588
		3	897		1146	1019	1391	909	867	626	**608**	731	687
	#update	1	2025		11640	1985	16713	11620	2532	5301	4204	2528	2890
		3	2025		11344	2005	19625	12893	2852	5451	5147	2942	2761
	per-update time (s)	1	0.286	0.141	0.050	0.370	0.066	0.065	0.289	0.093	0.095	0.197	0.203
		3	0.443	0.199	0.062	0.508	0.071	0.071	0.304	0.115	0.118	0.248	0.249
DenseNet-121	run time (s)	1	964	N/A	870	977	952	913	923	**660**	694	947	964
		2	1432		1224	1532	1299	1243	1129	781	**705**	1269	1009
	#update	1	1176		6850	1180	12009	11556	1487	2798	2746	1779	1947
		2	1125		7948	1192	13273	12551	1643	2886	2487	2202	1666
	per-update time (s)	1	0.820	0.397	0.127	0.828	0.079	0.079	0.621	0.236	0.253	0.532	0.495
		2	1.273	0.470	0.154	1.286	0.098	0.099	0.687	0.271	0.283	0.576	0.606



(a) VGG-19 on CIFAR10 (b) ResNet-34 on CIFAR100

Figure 7: Convergence performance on CIFAR.

run time. PS ASP alleviates the synchronization cost but involves more updates, resulting in slower convergence speed. PS HETE outperforms ASP due to the staleness-aware learning schema, but it is still slower than partial reduce. For PS BK, we use 5 out of total 8 workers for each synchronization (i.e., 3 backup workers). However, our partial-reduce (P=5) still significantly outperforms BK up to 1.47×, given the same number of nodes required in each synchronization. The key advantage of our approach over BK is that the stragglers are not contributing in BK whereas all workers could contribute in P-reduce with parallel-asynchronous partial-reduce operations. This allows our approach to achieve high resource (e.g., GPU) utilization in training clusters, thus improving the distributed training performance. Furthermore, as shown in Table 1, compared to BK, partial reduce (P=5) achieves lower per-update time due to high resource utilization.

With the new flexible parallel-asynchronous primitive, P-reduce solves the dilemma between high sensitivity to heterogeneity and low resource utilization faced by BK method. For example, Table 1 shows that although training more iterations due to the asynchronous nature, P-reduce (P=3) could achieve even higher runtime speedup (up to 1.84×) than BK by further relaxing the workers required in each synchronization. By contrast, if BK would make such relaxation, the majority of workers cannot be used. In summary, our approach offers the following advantages with respect to the BK [8]: (1) more flexibility in heterogeneity tolerance, (2) high resource utilization, thus leading to high scalability as verified in the following Sec. 5.3.

5.2.2 Comparison with Collective-based approaches. As shown in Table 1, partial reduce always outperforms these baselines in total run time. Especially for the heterogeneous setting (HL>1), DYN (P=3) can be at most 1.48 – 2.01× faster than All-Reduce. Compared to the constant partial reduce, the dynamic partial reduce can be aware of the staleness and reduce the number of needed updates until convergence. Compared to All-Reduce, partial reduce based methods require more updates before convergence, but the per-update time is 4× faster than All-Reduce.

For AD-PSGD, each worker computes gradients first, and performs an atomic model averaging with a randomly selected neighbor (regardless of its status). The model used for averaging can be inconsistent with the gradient update. As shown in Table 1, partial reduce (P=3) could achieve up to 1.74× speedup in a heterogeneous setting (DenseNet-121 under HL=2). This is because of two fundamental differences: (1) we allow synchronizing with a group of workers (more than two workers), enabling fast propagation of model updates among workers. (2) AD-PSGD hurts the model quality due to the inconsistent model update and results in a loose upper bound of the convergence rate [29]. In P-Reduce, thanks to the controller, we guarantee the consistency between model average and gradient computation without extra synchronization overheads.

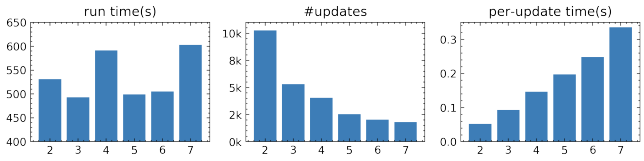


Figure 8: Partial reduce results over different P.

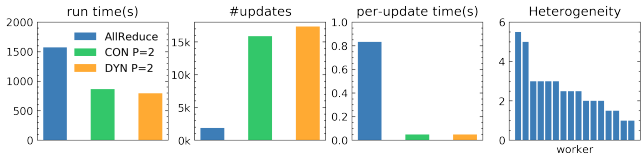


Figure 9: Comparison on ResNet-34 on CIFAR100.

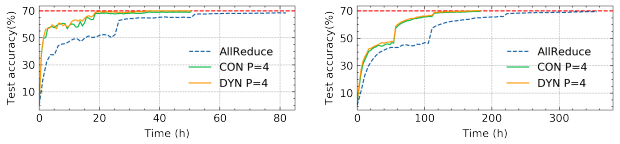
For ER, the per update time in Table 1 is smaller than AR due to the partial synchronization, but it cannot achieve the convergence thresholds as shown in Figure 7(a). There are two key differences between ER and our approach: (1) ER performs gradient aggregation, suffers from stale gradient problem, and significantly affects convergence quality. (2) ER requires the majority to participate for a synchronization, limiting the degree of heterogeneity tolerance. By contrast, P-reduce adopts model averaging for better convergence and flexible group size P for high heterogeneity tolerance.

5.2.3 Impact of group size. We further analyze the hyperparameter P in our partial reduce. Here we explore the impact of P by varying it on the VGG-19 for constant partial reduce under HL=1 setting. As shown in Fig. 8, for partial reduce, the per-update time is positively correlated with P, and the number of updates performs conversely, which verifies the convergence property we proved in Theorem 1. The total run time is the product of them and thus might have two possible minimum values. In our settings, partial reduce achieves the optimal total run time with $P = 3$ and 5. Note that, the best P varies across different workloads and need to be tuned in practice.

5.3 Evaluation on Production Environment

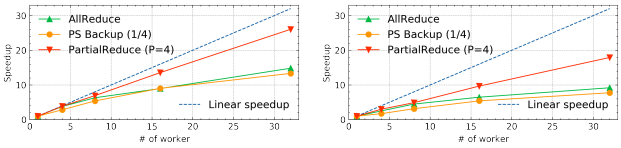
To verify the effectiveness of partial reduce in a real productive cluster, we conduct experiments on a heterogeneous production cluster from our industrial partner Tencent Inc. Each worker is a container (i.e., instance) requested from the cluster scheduler and equipped with a virtual V100 GPU. Note that, the production environment could be highly heterogeneous (as shown in Figure 9) due to resource sharing.

5.3.1 Convergence performance. We first apply 16 workers and compare partial reduce with AR on ResNet-34 on CIFAR100. Figure 7(b) shows the superior convergence performance of partial reduce. We also investigate the detail statistics in Figure 9 and find that our partial reduce methods are around 16.6× faster than AR in per-update time and achieve 2× speedup in total run time. Furthermore, we evaluate our approach on larger workloads, such as ResNet-18 and VGG-16 on ImageNet with 32 workers. Figure 10(a) and 10(b) demonstrate that partial reduce has competitive convergence guarantees as AR (e.g., the standard terminated test accuracy as reported in [16]) on both workloads, but with high convergence speed in terms of training time.



(a) ResNet-18 on ImageNet (b) VGG-16 on ImageNet

Figure 10: Convergence performance on ImageNet.



(a) ResNet-18 speedup (b) VGG-16 speedup

Figure 11: Scalability study on ImageNet

5.3.2 Scalability study. We also conduct a scalability study in terms of run time speedup on ImageNet with 1, 4, 8, 16, 32 workers respectively. As shown in Fig. 11(a) and Fig. 11(b), both AR and PS BK (using a quarter of workers for backup) have limited scalability. They suffer from more heterogeneity as involves more workers in a shared environment. By contrast, our partial reduce (P=4) achieves high heterogeneity tolerance and improved sociability with flexible parallel-asynchronous p-reduce operations. We also note that all methods show better scalability on ResNet-18 than VGG-16. Because ResNet-18 is computation-intensive while VGG-16 is communication-intensive and more difficult to scale up, which is consistent with prior work [5].

6 CONCLUSIONS

We analyzed the existing distributed ML communication strategies under a heterogeneous environment. All-Reduce is a high-performance communication schema but suffering from the stragglers due to heterogeneity. To tackle this problem, we proposed a novel partial reduce primitive, allowing partial synchronization among a worker group without waiting for stale workers. We theoretically proved the convergence property of our method to be $O(\sqrt{PK})$ with sufficiently large iterations K. To further improve the convergence performance, we proposed a dynamic partial reduce schema by considering the stragglers' delayed model parameters. We implemented a prototype system and evaluated it in both the experimental environment and the production environment. We found that our prototype could be 1.21×-2× faster than existing state-of-the-art baselines. Moreover, our heterogeneity-aware algorithms were able to achieve better hardware efficiency while preserving rapid convergence performance.

7 ACKNOWLEDGEMENT

This work is supported by the National Key Research and Development Program of China (No. 2018YFB1004403), the National Natural Science Foundation of China (No. 61832001, U1936104, 61702015), PKU-Tencent joint research Lab, Beijing Academy of Artificial Intelligence (BAAI), CAAI Huawei MindSpore Open Fund, and The Fundamental Research Funds for the Central Universities 2020RC25. Zhi Yang is the corresponding author.

REFERENCES

[1] 2017. PyTorch. https://github.com/pytorch/examples/tree/master/imagenet.
[2] 2021. Theorem Proofs. https://github.com/DMALab/Partial-Reduce.
[3] Martín 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 Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *OSDI*. 265–283.
[4] Md. Maksudul Alam, Kalyan S. Perumalla, and Peter Sanders. 2019. Novel Parallel Algorithms for Fast Multi-GPU-Based Generation of Massive Scale-Free Networks. *Data Sci. Eng.* 4, 1 (2019), 61–75.
[5] Dan Alistarh, Demjan Grubic, Jerry Li, Ryota Tomioka, and M. Vojnovic. 2017. QSGD: Communication-Efficient SGD via Gradient Quantization and Encoding. In *NeurIPS*.
[6] Léon Bottou, Frank E. Curtis, and Jorge Nocedal. 2018. Optimization Methods for Large-Scale Machine Learning. *SIAM Rev.* 60, 2 (2018), 223–311.
[7] Brendan Burns, Brian Grant, David Oppenheimer, Eric A. Brewer, and John Wilkes. 2016. Borg, Omega, and Kubernetes. *ACM Queue* 14, 1 (2016), 10.
[8] J. Chen, Rajat Monga, S. Bengio, and R. Józefowicz. 2016. Revisiting Distributed Synchronous SGD. *ArXiv* abs/1702.05800 (2016).
[9] Jichan Chung, Kangwook Lee, Ramtin Pedarsani, Dimitris Papailiopoulos, and Kannan Ramchandran. 2017. Ubershuffle: Communication-efficient data shuffling for sgd via coding theory. *NeurIPS*.
[10] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, Marc'Aurelio Ranzato, Andrew W. Senior, Paul A. Tucker, Ke Yang, and Andrew Y. Ng. 2012. Large Scale Distributed Deep Networks. In *NeurIPS*. 1232–1240.
[11] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Fei-Fei Li. 2009. ImageNet: A large-scale hierarchical image database. In *CVPR*. 248–255.
[12] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *NAACL-HLT*. 4171–4186.
[13] Wenfei Fan, Kun He, Qian Li, and Yue Wang. 2020. Graph algorithms: parallelization and scalability. *Sci. China Inf. Sci.* 63, 10 (2020), 1–21.
[14] Arash Fard, Anh Le, George Larionov, Waqas Dhillon, and Chuck Bear. 2020. Vertica-ML: Distributed Machine Learning in Vertica Database. In *SIGMOD*. 755–768.
[15] Saeed Ghadimi, Guanghui Lan, and Hongchao Zhang. 2016. Mini-batch stochastic approximation methods for nonconvex stochastic composite optimization. *Math. Program.* 155, 1-2 (2016), 267–305.
[16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *CVPR*. 770–778.
[17] Qirong Ho, James Cipar, Henggang Cui, Seunghak Lee, Jin Kyu Kim, Phillip B. Gibbons, Garth A. Gibson, Gregory R. Ganger, and Eric P. Xing. 2013. More Effective Distributed ML via a Stale Synchronous Parallel Parameter Server. In *NeurIPS*. 1223–1231.
[18] Kevin Hsieh, Aaron Harlap, Nandita Vijaykumar, Dimitris Konomis, Gregory R. Ganger, Phillip B. Gibbons, and Onur Mutlu. 2017. Gaia: Geo-Distributed Machine Learning Approaching LAN Speeds. In *NSDI*. 629–647.
[19] Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q. Weinberger. 2017. Densely Connected Convolutional Networks. In *CVPR*. 2261–2269.
[20] Jiawei Jiang, Bin Cui, Ce Zhang, and Lele Yu. 2017. Heterogeneity-aware Distributed Parameter Servers. In *SIGMOD*. 463–478.
[21] Jiawei Jiang, Fangcheng Fu, Tong Yang, and Bin Cui. 2018. SketchML: Accelerating Distributed Machine Learning with Data Sketches. In *SIGMOD*. 1269–1284.
[22] Soojeong Kim, Gyeong-In Yu, Hojin Park, Sungwoo Cho, Eunji Jeong, Hyeonmin Ha, Sanha Lee, Joo Seong Jeong, and Byung-Gon Chun. 2019. Parallax: Sparsity-aware Data Parallel Training of Deep Neural Networks. In *EuroSys*. 43:1–43:15.
[23] Bofang Li, Aleksandr Drozd, Yuhe Guo, Tao Liu, Satoshi Matsuoka, and Xiaoyong Du. 2019. Scaling Word2Vec on Big Corpus. *Data Sci. Eng.* 4, 2 (2019), 157–175.
[24] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. 2014. Scaling Distributed Machine Learning with the Parameter Server. In *OSDI*. 583–598.
[25] Shigang Li, Tal Ben-Nun, Salvatore Di Girolamo, Dan Alistarh, and Torsten Hoefler. 2020. Taming unbalanced training workloads in deep learning with partial collective operations. In *PPoPP*. 45–61.
[26] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. 2020. PyTorch Distributed: Experiences on Accelerating Data Parallel Training. *PVLDB* 13, 12 (2020), 3005–3018.
[27] Xiangru Lian, Yijun Huang, Yuncheng Li, and Ji Liu. 2015. Asynchronous Parallel Stochastic Gradient for Nonconvex Optimization. In *NeurIPS*. 2737–2745.
[28] Xiangru Lian, Ce Zhang, Huan Zhang, Cho-Jui Hsieh, Wei Zhang, and Ji Liu. 2017. Can Decentralized Algorithms Outperform Centralized Algorithms? A Case Study for Decentralized Parallel Stochastic Gradient Descent. In *NeruIPS*. 5330–5340.
[29] Xiangru Lian, Wei Zhang, Ce Zhang, and Ji Liu. 2018. Asynchronous Decentralized Parallel Stochastic Gradient Descent. In *ICML*, Vol. 80. 3049–3058.
[30] Yucheng Lu, Jack Nash, and Christopher De Sa. 2020. MixML: A Unified Analysis of Weakly Consistent Parallel Learning. *CoRR* abs/2005.06706 (2020). arXiv:2005.06706
[31] Qinyi Luo, Jiaao He, Youwei Zhuo, and Xuehai Qian. 2020. Prague: High-Performance Heterogeneity-Aware Asynchronous Decentralized Training. In *ASPLOS*. 401–416.
[32] X. Miao, L. Ma, Z. Yang, Y. Shao, B. Cui, L. Yu, and J. Jiang. 2020. CuWide: Towards Efficient Flow-based Training for Sparse Wide Models on GPUs. *TKDE* (2020), 1–1. https://doi.org/10.1109/TKDE.2020.3038109
[33] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. abs/1912.01703 (2019).
[34] Pitch Patarasuk and Xin Yuan. 2009. Bandwidth optimal all-reduce algorithms for clusters of workstations. *J. Parallel Distributed Comput.* 69, 2 (2009), 117–124.
[35] Alexander Sergeev and Mike Del Balso. 2018. Horovod: fast and easy distributed deep learning in TensorFlow. *CoRR* abs/1802.05799 (2018).
[36] Haidong Shao, Hongkai Jiang, Haizhou Zhang, Wenjing Duan, Tianchen Liang, and Shuaipeng Wu. 2018. Rolling bearing fault feature learning using improved convolutional deep belief network with compressed sensing. *Mechanical Systems and Signal Processing* 100 (2018), 743–765.
[37] Karen Simonyan and Andrew Zisserman. 2015. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *ICLR*.
[38] Antti Tarvainen and Harri Valpola. 2017. Mean teachers are better role models: Weight-averaged consistency targets improve semi-supervised deep learning results. In *NeurIPS*. 1195–1204.
[39] Jianyu Wang and Gauri Joshi. 2019. Cooperative SGD: A Unified Framework for the Design and Analysis of Communication-Efficient SGD Algorithms. In *ICML Workshop*.
[40] Xintong Wang and Yunfei Feng. 2018. An Ensemble Learning Algorithm for Indoor Localization. In *ICCC*. 774–778.
[41] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, Fan Yang, and Lidong Zhou. 2018. Gandiva: Introspective Cluster Scheduling for Deep Learning. In *OSDI*. 595–610.
[42] Xu Xie, Fei Sun, Xiaoyong Yang, Zhao Yang, Jinyang Gao, Wenwu Ou, and Bin Cui. 2021. Explore User Neighborhood for Real-time E-commerce Recommendation. arXiv:cs.IR/2103.00442
[43] Ce Zhang and Christopher Ré. 2014. DimmWitted: A Study of Main-Memory Statistical Analytics. *PVLDB* 7, 12 (2014), 1283–1294.
[44] Zhipeng Zhang, Bin Cui, Yingxia Shao, Lele Yu, Jiawei Jiang, and Xupeng Miao. 2019. PS2: Parameter Server on Spark. In *SIGMOD*. 376–388.
[45] Shuai Zheng and James T Kwok. 2017. Follow the moving leader in deep learning. In *ICML*. 4110–4119.
[46] Weimin Zheng. 2020. Research trend of large-scale supercomputers and applications from the TOP500 and Gordon Bell Prize. *Sci. China Inf. Sci.* 63, 7 (2020).
[47] Martin Zinkevich, M. Weimer, Alex Smola, and L. Li. 2010. Parallelized Stochastic Gradient Descent. In *NeurIPS*.