SDPipe: A Semi-Decentralized Framework for Heterogeneity-aware Pipeline-parallel Training

Xupeng Miao[*]
Carnegie Mellon University
xupeng@cmu.edu

Yining Shi[*]
Zhi Yang[*]
Peking University
shiyining@pku.edu.cn
yangzhi@pku.edu.cn

Bin Cui[*][†]
Peking University
bin.cui@pku.edu.cn

Zhihao Jia
Carnegie Mellon University
zhihao@cmu.edu

ABSTRACT

The increasing size of both deep learning models and training data necessitates the ability to scale out model training through pipeline-parallel training, which combines pipelined model parallelism and data parallelism. However, most of them assume an ideal homogeneous dedicated cluster. As for real cloud clusters, these approaches suffer from the intensive model synchronization overheads due to the dynamic environment heterogeneity. Such a huge challenge leaves the design in a dilemma: either the performance bottleneck of the central parameter server (PS) or severe performance degradation caused by stragglers for decentralized synchronization (like All-Reduce). This approach presents SDPipe, a new *semi-decentralized* framework to get the best of both worlds, achieving both high heterogeneity tolerance and convergence efficiency in pipeline-parallel training. To provide high performance, we decentralize the communication model synchronization, which accounts for the largest proportion of synchronization overhead. In contrast, we centralize the process of group scheduling, which is lightweight but needs a global view for better performance and convergence speed against heterogeneity. We show via a prototype implementation the significant advantage of SDPipe on performance and scalability, facing different environments.

PVLDB Reference Format:
Xupeng Miao, Yining Shi, Zhi Yang, Bin Cui, Zhihao Jia. SDPipe: A Semi-Decentralized Framework for Heterogeneity-aware Pipeline-parallel Training. PVLDB, 16(9): 2354 - 2363, 2023.
doi:10.14778/3598581.3598604

PVLDB Artifact Availability:
The source code of this research paper has been made publicly available at https://github.com/Hsword/VLDB2023_SDPipe/.

1 INTRODUCTION

Recently, Deep Neural Network (DNN) models gain a lot of attention due to its superior performance in various tasks like image recognition [55], natural language process [18] and graph mining [21, 59].

[*]School of CS & Key Lab of High Confidence Software Technologies (MOE), PKU
[†]Institute of Computational Social Science, PKU (Qingdao), China

Proceedings of the VLDB Endowment, Vol. 16, No. 9 ISSN 2150-8097.
doi:10.14778/3598581.3598604

With the increasing size of DNN models and the proliferation of training data, *data parallelism* [31, 33] has been widely used to accelerate the training, where the DNN model is replicated on multiple worker machines, with each worker processing a subset of the training data. However, if a DNN model cannot be fit into the memory of a single GPU, data parallelism cannot be used.

Pipeline-parallel training [26, 43, 68] has the potential to provide high training performance for large DNN models when data parallelism struggles. Pipeline-parallel is a combination of data parallelism and pipeline model parallelism, which partitions the layers of the model being trained into multiple *stages* – each stage contains a consecutive set of layers in the model. Each GPU executes both the forward and backward passes for the assigned stage and mini-batches can be processed in a pipelined manner across stages. It also uses data parallelism by assigning multiple workers to the same stage, processing different mini-batches in parallel. The workers of each stage maintain replicas of the corresponding model layers and periodically synchronize to guarantee convergence.

Recently, some large tech firms (e.g., Google [70], Microsoft, NVIDIA [30], and Tencent [47, 48]) have utilized pipeline-parallel to accelerate large foundation models [11] training on dedicated clusters equipped with hundreds of homogeneous AI chips and high-speed inter-connects. However, such homogeneous environments are fairly expensive and unrealistic for most researchers. In commodity clouds, heterogeneity is quite common. Besides deterministic heterogeneity (e.g., hardware devices [49] and network connections [25]), dynamic heterogeneity can introduce uncontrollable performance variations caused by dynamic virtual machine (VM) consolidation [64] or resource sharing and competition [42, 63, 69]. There is also a trend that users without strict latency requirements prefer cheap but unstable (or low-priority) cloud services (e.g., serverless computing [20, 28] and spot-instance [9, 58]) to save costs. Given such dynamic heterogeneous environments, the model synchronizations among data-parallel workers are sensitive to the slowest worker, resulting in low training efficiency. As illustrated in Figure 1, in pipeline-parallel training, the problem is more severe since the temporary stragglers will lead to idle waiting overheads for all the other stages and pipelines. Therefore, pipeline-parallel training cannot tolerate heterogeneity well.

In order to efficiently scale pipeline-parallel training, it's important to minimize the model synchronization overheads while preserving the convergence efficiency. The parallel synchronization schemes are crucial for communication-efficient training, which describes *when* and *how* the parameters of the different workers are synchronized. The default solution is to utilize existing

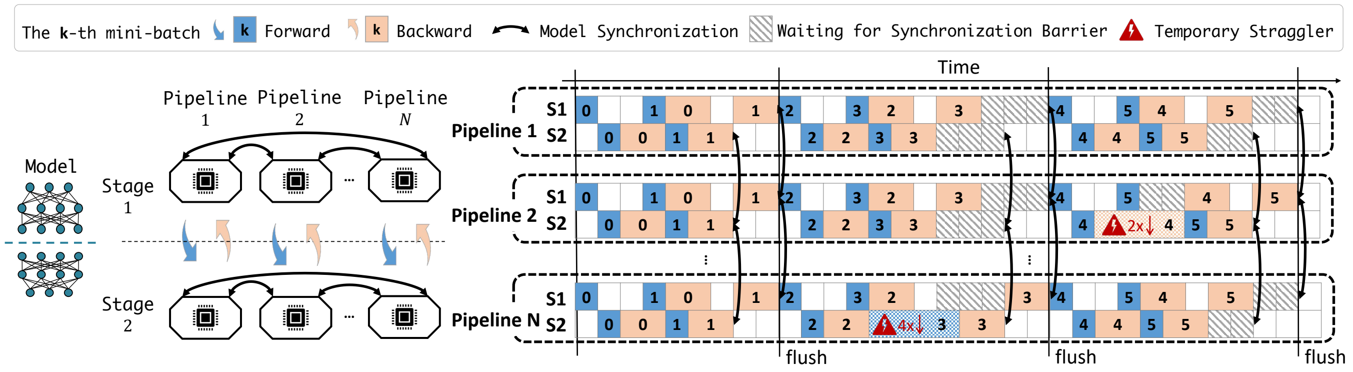


Figure 1: Pipeline-parallel training under a dynamic heterogeneous environment. The pipelines are synchronized periodically, while the flush points represent model synchronizations, e.g., flushed every two mini-batches of training. The worker of Stage 2 in Pipeline N is temporarily 4× slowing down during the forward computation of the 3-rd mini-batch. The worker of Stage 2 in Pipeline 2 is temporarily 2× slowing down during the backward computation of the 4-th mini-batch.

data parallelism synchronization schemes. At the system level, existing schemes can be classified into *centralized* or *decentralized* paradigms. The centralized scheme synchronizes the parameters among the workers across different pipelines via a parameter server (PS) [32]. Benefiting from the global view of the centralized PS, it provides a flexible way to tolerate heterogeneity, such as involving asynchronous training with bounded staleness [24], dropping the model updates from stragglers [14]. However, as all workers must communicate with the PS, it often leads to the central communication bottleneck [34], especially for large models. In the decentralized design, the communication is more evenly distributed by allowing point-to-point communication between workers. Unfortunately, the decentralized scheme relies on a fixed topology for communication without a global view, making it inflexible and vulnerable to system heterogeneity. For example, the typical distributed solution All-Reduce [51, 60, 62] requires global synchronization in every step, its performance is bounded by the slowest worker [37].

In this paper, we propose **SDPipe**, a new **semi-decentralized** scheme for **pipeline parallel** training which combines *centralized and decentralized* schemes to achieve the best of both worlds. Our key insight is to divide the functionality of a synchronization operation into two parts: one for generating a specific topology for communication, and the other for P2P communication between workers using the specific topology for communication. In SDPipe, we leverage the lightweight nature of the first step and employ a central scheduler to determine the size and membership of a worker group to perform model synchronization for a specific stage. This global view enables to always generate groups appropriately, which eliminates the penalty introduced by heterogeneity and contention. On the other hand, the heavy model synchronization is completely decentralized across worker groups, which distributes the communication load among nodes.

We further exploit the scheduling capabilities of SDPipe to enforce training performance and convergence. First, for group generation, we leverage the ready signals the scheduler collects from the workers to globally identify those who have finished the local model update iterations almost simultaneously. In this way, we could maximize the fast propagation of model parameter updates without the stragglers from a global view. Unlike PS and All-Reduce, which use specific topology for communication, our training scheme can appropriately adopt an arbitrary communication group to dynamic heterogeneity. Second, since we can globally track model update propagation progress, we provide a stronger convergence guarantee by controlling the speed of model update propagation. Specifically, we require that the update on any worker can be passed through group synchronization to all the workers within a specific iteration number, rather than eventually assumed by decentralized ones. We theoretically prove the convergence rate under such requirements. To achieve this requirement, the central scheduler converts the model propagation progress into a synchronization graph and explicitly enforces the group generation to maintain the connectivity of the synchronization graph. We also note that this enforcement might incur deadlock, and present a deadlock prevention method.

We evaluate the performance of SDPipe for both CNN and Transformer models using a large GPU cluster of up to 64 GPUs. Our experimental results demonstrate that the performance of SDPipe is faster than that of centralized pipeline-parallel via PS and decentralized pipeline-parallel via All-Reduce or AD-PSGD [35], respectively. In more heterogeneous settings or larger clusters, SDPipe achieves higher speedup than both centralized and decentralized baselines.

2 BACKGROUND & RELATED WORK

2.1 Data Parallelism

Existing data-parallelism distributed training approaches could be classified into the following two categories:

Centralized Training Frameworks. Centralized parallel SGD (C-PSGD) has been widely applied to many parameter server (PS) based DL frameworks (e.g., TensorFlow [7], MXNet [15]). The central server is responsible for handling model updates and connects to the other workers, which compute the stochastic gradients in parallel. Note that, the server does not have to be a physical dedicated machine. It is just a logical abstraction providing a global view of the model parameters/gradients, and it can be implemented in a distributed way and even co-located with workers (e.g., BytePS [29]). It applies a synchronization protocol to determine the synchronization patterns among the workers during the parallel optimization iterations. In bulk synchronous parallel (BSP) [19] protocol, each worker cannot step into the next iteration until all workers finish the current iteration and perform the global synchronization.

Decentralized Training Frameworks. Centralized frameworks utilize a central parameter server for model synchronization, while decentralized frameworks rely on the network topology and follow a fixed communication manner where the models are exchanged between neighbor workers. The decentralized parallel SGD (D-PSGD) was proposed by [45, 52, 53] and [34] first claimed that D-PSGD can achieve linear speedup with respect to the number of workers and obtains the similar convergence rate as C-PSGD. Recently, due to the development of high-speed inter-GPU connections (e.g., NVLink), many popular DL frameworks adapt to decentralized communication architecture, such as PyTorch [50] and Horovod [54].

2.2 Pipelined Model Parallelism

Modern DL models require a significant amount of memory to store model parameters and intermediate results during training. Model parallelism approaches [10, 40] have been widely studied to support large DL models that cannot fit into one GPU device. Pipeline parallelism [26] is one of the most popular categories of model parallelism techniques, which assigns subsets of successive DNN layers to different processors. Among them, PipeDream [43] and PipeDream-Flush [44] are the most popular pipeline training methods that use a 1F1B (one forward one backward) scheduling algorithm. Due to the complex dependencies across stages, efficient pipeline parallel training is more challenging than pure data parallelism, especially under dynamic heterogeneous clusters. Other model parallelism mechanisms (e.g., tensor model parallelism (TP) [30]) are dividing a single layer's computation into small pieces over different devices. These pieces are naturally synchronous and could be treated as a single stage's workload. Then our proposed SDPipe can easily handle the heterogeneity among stages from different pipelines. We also notice that the heterogeneity among devices within the same TP group is a completely different research problem beyond our research target in this approach.

2.3 Trend of Heterogeneous Training

The trend of distributed training over heterogeneous resources is not only *promising* but also *inevitable*. As the number of model parameters growing rapidly, large-scale DNN training requires more GPU resources as well as monetary costs. But for most researchers and organizations, maintaining a large homogeneous GPU cluster is economically infeasible. Recently, there are more practical examples of using heterogeneous resources to make large models more economically viable. For example, DT-FM [65] and Petails [12] propose to run large language models collaboratively by joining people's various geo-distributed GPUs spread over the world. The mixture of different accelerators and complex networking is a naturally dynamic heterogeneous environment.

The public GPU cloud is another cost-effective choice but heterogeneity also exists. GPU sharing is one of the main sources of dynamic heterogeneity in commodity clouds. With the rising of single GPU's performance, several public GPU cloud providers start to support virtual GPU containers, such as Vultr [6] and Alibaba [3]. They allow users to apply for containers equipped with only a partial GPU device (e.g., 1/2 or even 1/20 of a single GPU), sharing the same physical GPU with others. These virtualized GPUs could be a better choice compared with the exclusive on-demand GPUs due to lower price-FLOPS ratios. However, we observed that its performance (i.e., peak GPU FLOPS) might be changed due to unforeseen resource sharing in Figure 7a. Considering these, we believe that how to perform efficient distributed training on dynamic heterogeneous resources is an in-time and important research problem.

2.4 Heterogeneous Training Schemes

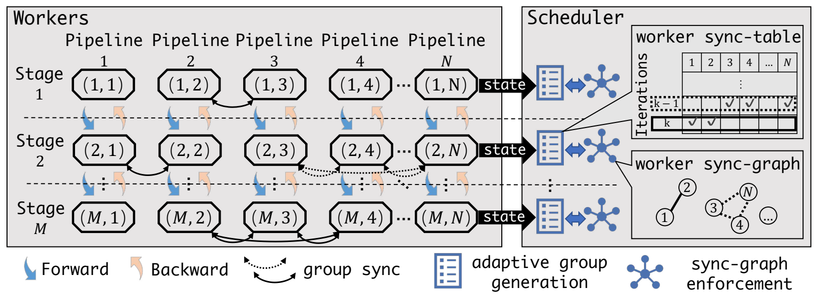
Heterogeneous training attracts lots of attention in recent years. Most of them belong to pure data parallelism approaches, taking a centralized approach with relaxed synchronization protocol [7, 14, 27], or a decentralized approach [35] relying on communication between neighboring workers. However, heterogeneity-aware pipeline-parallel training has been rarely studied. Figure 1 describes a running example of pipeline-parallel training where the model is partitioned into two stages and replicated by N pipelines among $2N$ workers. The workers for the same stage synchronize (i.e., flush) their model replicas among different pipelines every two mini-batches of training. Unlike stragglers in pure data parallelism (i.e., only blocks the synchronized workers), the stragglers in pipeline-parallel not only bring idle waiting overheads to the workers in the same stage, but also affect workers from the other stages due to the pipeline execution dependencies. For example, we suppose the worker of Stage 2 in Pipeline N is temporarily 4× slowing down during the forward computation of the 3-rd mini-batch. We found that the workers of Stage 1 are also blocked due to the dependency, i.e., the worker of Stage 1 in Pipeline N cannot start the backward computation of the 3-rd mini-batch. Considering a dynamic heterogeneous environment, pipeline-parallel training has shown very poor tolerance for sudden and unpredictable stragglers.

A recent approach HetPipe [49] moves a step from PipeDream-Flush by involving the traditional SSP [27] protocol with a centralized PS to handle the static heterogeneity from different versions of GPUs. But it only treats the entire pipeline as a virtual worker in pure data parallelism, suffering from the central PS bottleneck. What's more, it neglects the fine-grained per-stage synchronization nature in pipeline-parallel and still struggles with the dynamic heterogeneous environments. To the best of our knowledge, no previous approaches can solve the model synchronization problem in dynamic heterogeneity environments under the combination of both pipeline parallelism and data parallelism.

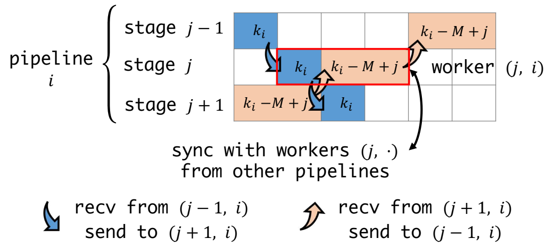
3 SDPIPE DESIGN

In this section, we describe SDPipe, a semi-decentralized training system specialized for pipeline-parallel training of a large DNN model in a heterogeneous GPU cluster. SDPipe performs pipeline parallel training by partitioning the layers of the model being trained into multiple stages – each stage contains a consecutive set of DNN layers. Meanwhile, SDPipe performs data parallelism for individual stages—multiple workers can be assigned to a given stage, processing different mini-batches in parallel. Figure 2a illustrates the SDPipe training framework, the workers are organized as a $M \times N$ mesh structure to perform N-way data parallelism and M-way pipeline parallelism training. Each worker is denoted as (j, i), who is responsible for the j-th stage of the i-th pipeline.

The separation of the central scheduling and decentralized synchronization makes our semi-decentralized training framework (Section 3.1) fundamentally different from existing centralized and



(a) Illustration of the SDPipe training framework.


pipeline i
stage j − 1
stage j
stage j + 1
k_i
$k_i - M + j$
worker (j, i)
sync with workers (j, ·) from other pipelines
recv from (j − 1, i) send to (j + 1, i)
recv from (j + 1, i) send to (j − 1, i)

(b) Illustration of the proposed algorithm

Figure 2: Illustration of the SDPipe training framework and the algorithm

decentralized systems. Specifically, the traditional centralized approach [49] treats the workers within the same pipeline as a virtual worker and relies on a central PS to aggregate model updates from all virtual workers and then broadcast back. The decentralized architecture fully utilizes inter-GPU connection to synchronize model parameters among the workers within the same pipeline stage. Our SDPipe involves a central scheduler to determine the centralized synchronization scheduling during the training process. But the scheduler never stores the model parameters or directly engages in the model synchronization like PS. SDPipe provides a dynamic fine-grained worker group scheduling for each stage and the workers within the same temporary group synchronize parameters in a decentralized manner. Unlike prior work, the global view of the central scheduler provides the following key *scheduling* capabilities:

Adaptive Group Generation. The key idea of fine-grained group scheduling is to allow partial fast workers within the same stage to synchronize without waiting the others. To achieve this goal, the scheduler monitors the current training state (such as compute, ready, or sync) of each worker at each iteration k. By exploiting the arrivals of ready signals at the scheduler, we do not need to explicitly restrict a fixed communication topology required by previous decentralized work, thus providing high scheduling flexibility and adaptivity to heterogeneous settings (Section 3.2).

Convergence Enforcement. The flexible best-effort group scheduling are beneficial to the hardware utilization but wild worker groups might hurt training semantics. To address this issue, the scheduler maintains this history of temporary worker groups, which is organized in a collection of rows in SyncTable. Each row records the group membership at each iteration. By converting the recent synchronization history into a synchronization graph (Section 3.3), we can explicitly enforce the convergence of asynchronous decentralized group synchronizations by enforcing the connectivity of the synchronization group at the central scheduler (Section 3.4).

3.1 Semi-decentralized Framework

We then introduce the detailed workflow of SDPipe as follows. Each worker (j, i) maintains a replica of model partition $\mathbf{x}^{j,i}$ for stage j, and performs the following two steps within each training iteration, as illustrated in Figure 2b and Algorithm 1.

Step A: Local Computation. We suppose k_i is the mini-batch index of input data feeds into pipeline $(\cdot, i)$. The local computation step follows the 1F1B schedule like PipeDream, consisting of both forward (step A1) and backward (step A2) computation. At the

Algorithm 1 Semi-Decentralized Parallel with 1F1B

```
Ensure: Worker $(j, i)$, stage $j (\leq M)$ of pipeline $i (\leq N)$:
1:  for $k_i = 1, 2, \ldots, K$ do
2:      Step A1: Forward computation
3:      Receive $\mathbf{o}^F_{j-1,i}$ from worker $(j-1, i)$.
4:      Forward compute $\mathbf{o}^F_{j,i} = f(\mathbf{x}^{j,i}; \mathbf{o}^F_{j-1,i})$.
5:      Send $\mathbf{o}^F_{j,i}$ to worker $(j+1, i)$.
6:      Step A2: Backward computation
7:      Receive $\mathbf{o}^B_{j+1,i}$ from worker $(j+1, i)$.
8:      Backward compute $\mathbf{o}^B_{i,j}, \mathbf{g}^{j,i}_{k_i} = f_b(\mathbf{x}^{j,i}; \mathbf{o}^B_{j+1,i})$.
9:      Send $\mathbf{o}^B_{j,i}$ to worker $(j-1, i)$.
10:     Update the local model $\mathbf{x}^{j,i} \leftarrow \mathbf{x}^{j,i} - \gamma \mathbf{g}^{j,i}_{k_i}$.
11:     Step B: Model synchronization
12:     Send ready signal $(j, i)$ to the scheduler.
13:     Receive a group $\mathcal{S}^j$ from the scheduler.
14:     Aggregate local models by $\mathbf{x}^{j,i} \leftarrow 1/|\mathcal{S}^j| \sum_{(j,u)\in\mathcal{S}^j} \mathbf{x}^{j,u}$.
15: end for
```

beginning of each iteration, each worker (j, i) first receives the forward output $\mathbf{o}^F_{j-1,i}$ from worker $(j-1, i)$ (line 3) and performs the forward computation with its local model partition $\mathbf{x}^{j,i}$ (line 4). Specially, the first stage's workers skip the receiving step and takes a mini-batch of input data as $\mathbf{o}^F_{0,i}$. Then it sends the forward output to the worker of the next stage (line 5) unless it is in the last stage. During the forward step of worker (j, i), as shown in Figure 2b, worker $(j+1, i)$ is sending $\mathbf{o}^B_{j+1,i}$ (the backward output of the next stage corresponding to the $k_i - M + j$-th mini-batch of data) concurrently to worker (j, i) (line 7). Especially, the workers in the last stage skip such receiving process naturally. Then worker (j, i) could perform the backward computation and send the output to the worker of the previous stage (lines 8-9). In the meanwhile, it updates the local model cell using the generated gradients (line 10).

Step B: Model Synchronization. After finishing the above local computation steps, workers within the same pipeline stage have to perform a model synchronization before moving to the next iteration. SDPipe adopts a fine-grained model synchronization within the temporary worker group generated by the central scheduler. There might be multiple concurrent groups each time and their synchronizations can be performed in an asynchronous-parallel manner. In this way, SDPipe decentralizes the workload of heavy

Algorithm 2 Group Generation Algorithm

```
Ensure: SDPipe scheduler for stage $j$:
 1: $\mathcal{S}^j = \emptyset$
 2: while True do
 3:     while within a time period $T$ do
 4:         Receive a ready signal from worker $(j, i)$.
 5:         $\mathcal{S}^j = \mathcal{S}^j \cup \{(j, i)\}$.
 6:     end while
 7:     if $\mathcal{S}^j$ is valid (see Sec. 3.4) then
 8:         Send the group information to the workers in $\mathcal{S}^j$.
 9:         $\mathcal{S}^j \leftarrow \emptyset$.
10:     end if
11: end while
```

model synchronization across workers while providing high heterogeneity tolerance by eliminating the impact of stragglers on the other worker groups. To achieve this, the worker sends a ready-to-sync signal to the scheduler (line 12) once gradient updates are completed. When the worker receives a group $\mathcal{S}^j$ from the central scheduler, they exchange their models within the given group and update by model averaging (lines 13-14).

3.2 Adaptive Group Generation

The central scheduler provides a global view to schedule the workers from different pipelines at each stage. It maintains a global clock k which represents the number of group-sync operations in the past. From the central scheduler's view, each group-sync operation for a specific stage j can be formulated as:

$$\mathbf{X}_{k+1}^j = (\mathbf{X}_k^j - \gamma \mathbf{G}_k^j)\mathbf{W}_k^j, \tag{1}$$

where γ is the learning rate, matrices $\mathbf{X}_k^j$ and $\mathbf{G}_k^j$ contain the local model vector $\mathbf{x}_k^{j,i}$ and gradient vector $\mathbf{g}(\mathbf{x}_k^{j,i})$ of each worker (j, i) at the kth iteration. $\mathbf{W}_k^j$ is the synchronization matrix representing multiple groups' model averaging, which can be defined as:

$$\mathbf{W}_k^j(u, v) = \begin{cases} \frac{1}{|\mathcal{S}_k^j|}, & \text{if workers } u, v \in \mathcal{S}_k^j \\ 1, & \text{if worker } u \notin \mathcal{S}_k^j \text{ and } u = v, \\ 0, & \text{otherwise,} \end{cases} \tag{2}$$

where $\mathcal{S}_k^j$ represents the corresponding worker group at the k-th iteration in the stage j.

Previous decentralized training algorithms fall in the paradigm of fixed group size for all iterations. For example, All-Reduce requires all N workers to participate in the model synchronization. Local-SGD [56] and AD-PSGD [35] imposes group size of 1 and 2, respectively. P-reduce [38] and Prague [37] allow a larger but fixed group size. Since group size greatly influences both the heterogeneity tolerance and the propagation of model parameter updates among workers, the lack of flexibility and adaptivity in the fixed group size paradigm can leave the synchronization design in a dilemma: large group size is sensitive to the straggler, thereby cannot tolerate heterogeneity well. However, a small group size leads to slow update propagation and thus the convergence speed.

With the central scheduler, as shown in Algorithm 2, we could leverage the ready signals from the workers to identify those who have finished the local model update iterations almost simultaneously (within a time period of T). By exploiting the arrivals of ready signals at the scheduler, we do not need to explicitly restrict a fixed size or pattern of group formation required by previous work, thus providing high flexibility and adaptivity to dynamic heterogeneous settings. In particular, if the heterogeneity is more apparent in the current iteration, our method tends to reduce group size to avoid straggler slowdown (adapt to local-SGD in an extreme heterogeneous case). Otherwise, our method tends to increase group size to enlarge the unit of synchronization (adapt to All-Reduce in the homogeneous case). Using a too large budget (i.e., $T = \infty$) would degrade it to All-Reduce, while disabling the waiting time (i.e., $T = 0$) would lead to many single-worker groups and might slowdown the model propagation efficiency. Finally, the group validation is used to enforce convergence, which will be described in Section 3.4.

3.3 Convergence Guarantee

Given asynchronous group synchronization and flexible group generation, a key challenge for the central scheduler is to prompt model replicas at different workers to converge to the same point.

Sync-graph Principle. In our semi-decentralized framework, the central scheduler provides a way to globally control the model update propagation across the workers. Based on the SyncTable, we construct the synchronization graph to represent the historical groups. There exists N nodes in total to denote the workers. We can transform each group $\mathcal{S}_k^j$ into an edge set $E^j(k) = \{(u, v) : u, v \in \mathcal{S}_k^j\}$ (i.e., $\mathbf{W}_k^j(u, v) > 0$) representing the nodes within the group are fully connected. By convention, we take each node to be an in-neighbor of itself (each node in the graph has a self-loop). In SDPipe, based on the sync-graph, we provide a stronger guarantee on the speed of update propagation with a constant threshold P.

Assumption 1. ***Sync-graph connectivity:*** *for a given $P \geq 1$, the graph with edge set $\cup_{k=l}^{l+P-1} E^j(k)$ is strongly connected for any $l \geq 1$ and any stage j.*

We suppose two nodes from the same stage are connected only if they have been in the same worker group in a short past duration (i.e., $P - 1$ iterations). If two nodes from the same stage have not been "connected" for a long time, it means that their recent model updates are not shared, leading to model divergence. Given the above assumption, we can derive the following theorem to guarantee the convergence of SDPipe. We suppose $1 \leq n_1 \leq \mathcal{S}_k^j \leq n_2 \leq N$, where n_1 and n_2 are the maximum and the minimum worker group sizes during the entire training process, respectively. Note that, they are only used to help analysis and do not need to be tuned as hyper-parameters. Then we have (detailed proofs are in [5]):

Theorem 1 (Convergence of SDPipe). *We assume the bound of gradient variance σ^2 is in inverse proportion to the mini-batch size. For SDPipe, under Assumption 1 and some commonly used assumptions [5], if the learning rate satisfies*

$$\eta L + \frac{2MN^3\eta^2L^2t_1}{n_2^2}\left(\frac{1}{1-t_2} + \frac{2}{(1-\sqrt{t_2})^2}\right) \leq 1, \tag{3}$$

where $\eta = \frac{n_2\gamma}{N}$, $t_1 = 4N(\frac{1+n_1^{NP}}{1-n_1^{-NP}})^2$ and $t_2 = (1-n_1^{-NP})^{2/(NP)}$, and all local models are initialized at $\mathbf{u}_1$, given $\mathbf{u}_k = \sum_{i=1}^N \mathbf{x}_k^i/N$, then the

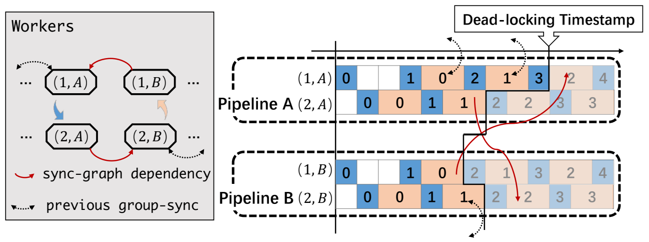


Figure 3: Illustration of dead-lock

average-squared gradient norm after K iterations is bounded as

$$\mathbb{E}\Big[\frac{1}{K}\sum_{k=1}^{K}\|\nabla F(\mathbf{u}_k)\|^2\Big] \leq \underbrace{\frac{2[F(\mathbf{u}_1)-F_{inf}]}{\eta K}}_{SGD\ error} + \tag{4}$$

$$\underbrace{\frac{\eta L\sigma^2}{n_2} + \frac{2M\eta^2L^2\sigma^2N^3n_1t_1}{n_2^3}\Big(\frac{1}{1-t_2}+\frac{2}{(1-\sqrt{t_2})^2}\Big)}_{network\ error}. \tag{5}$$

Theorem 1 shows that the convergence bound is mainly determined by the last two network error items. They describe the model difference among the workers during pipeline training, and the latter is positive correlated the sync-graph connectivity budget P.

3.4 Sync-graph Enforcement

Enabled by the global view, SDPipe explicitly enforces connectivity through the proposed sync-graph principle by the central scheduler.

Group Validation. Given the sync-graph connectivity requirement for each pipeline stage, the generated worker group would be verified by our framework before informing the workers. Specifically, in each iteration k, the scheduler converts the sync-graph $G(V, E_{P-1})$ based on the synchronization history of the latest $P-1$ iterations. The scheduler check whether the graph with the edge set $E_{P-1} \cup E(k)$ is strongly connected. If the validation fails, the scheduler should *wait* for more ready workers and re-generate the group to pass the validation. It guarantees that at any time, the last P worker groups always make the workers form a connected graph. This constraint prevents the model degradation and provides a significant convergence guarantee. Once the candidate group passes the validation, these corresponding workers formalize a temporary group and the scheduler informs them to execute a group-sync operation to exchange their models by group synchronization.

Deadlock Prevention. Note that the scheduler should *wait* for more ready workers if the validation fails. It also means that the worker could be blocked until it passes the validation together with the incoming ready workers. This enforcement scheme might incur deadlock if waiting occurs across the multiple stages in pipelines. Figure 3 describes an running example of deadlock across two pipelines A, B containing two stages. Suppose that worker $(1, B)$, worker $(2, A)$ just finished the backward computation and required to wait and synchronize with worker $(1, A)$ and worker $(2, B)$, respectively, due to the sync-graph enforcement. However, worker $(2, B)$ would not be able to produce synchronization opportunity because of the pipeline dependency from blocked worker $(1, B)$. Similarly, worker $(2, A)$ and $(1, A)$ are also blocked. Therefore, these four workers forms a waiting graph which leads to a deadlock. In general, such deadlock can include a various number of workers and form frequently under the dynamic heterogeneous environments.

In SDPipe, we design a simple but effective deadlock prevention mechanism. Specifically, the scheduler ensures that no sync-graph enforcement operation on one stage is allowed to be performed until the previous sync-graph enforcement operation has been completed on other stages. If the system has not finished one enforcement operation, the scheduler would *temporarily* ignore later group validation operation in the current iteration, which prevents mutual waiting across stages (like using an exclusive synchronization variable for enforcement). It may occasionally violate the sync-graph connectivity requirement for a few iterations, but would not hurt the final model convergence guarantee as most are performed without conflict. The deadlock problem further suggests introducing a global view of all workers, rather than making fully independent group synchronizations for different stages.

4 IMPLEMENTATION

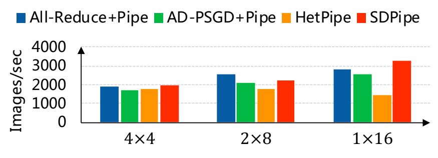
SDPipe's implementation is open sourced [4], built on top of Hetu [39, 41], a distributed deep learning runtime (implemented with more than 30k LOC in Python/C++/CUDA)[1]. Note that it is also possible to apply our methods to any DL framework which supports pipeline and data parallel training (e.g. PyTorch).

Scheduler. We implemented the scheduler in C++ with ZeroMQ. During training, a scheduler server is launched to serve the synchronization requests sent from each worker. For each pipeline stage, the scheduler server maintains a disjoint set to track whether the sync-graph connectivity condition is met. In our implementation, we allow the timeout duration used in the scheduler to be adaptively changed accordingly to the workers' pattern, which means that we will slightly enlarge the duration when there are too few ready workers within the maximum timeout duration T.

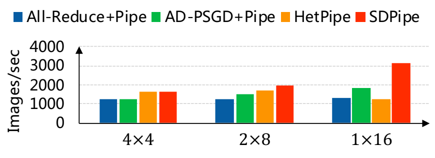
Worker Communication. Our pipeline workers utilize NCCL [2] for both model averaging communication and peer-to-peer pipeline operations. Upon each backward stage, the pipeline worker would query the sync-group from the scheduler server. The query is performed in the host asynchronously and would not block CUDA kernel execution. After the worker received the sync-group returned by the scheduler, a new NCCL communicator object is created for the sync-group. The communicator will be freed after the model averaging is completed. We persist a few communicators to handle some frequently appearing sync-groups.

Optimizer Variance Reduction. The group-sync operation averages the model weights (i.e., parameters) among the workers. But for some Adam-based optimizers with states (e.g., the first and second moments of the gradients), model averaging may enlarge the variance of the second moments estimations. Specifically, Adam updates exponential moving averages of the squared gradients locally. Based on the common assumptions, the variance of local gradients from a single worker is $N\times$ larger than those from all N workers. Ignoring such differences may result in poor convergence properties. In our approach, we maintain an additional version of the first moments and compute the variance by using the expectation of square (i.e., second moments) to minus square of expectation (i.e., square of first moments). Then we scale the variance with $1/N$ and rebuild the second moments with lower variance.

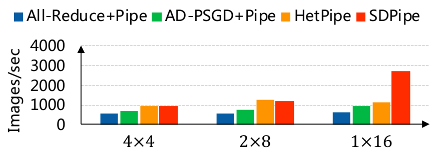
[1]https://github.com/PKU-DAIR/Hetu/



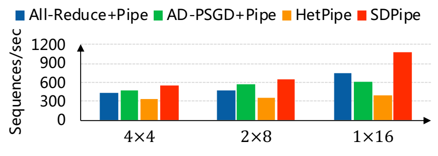
(a) ResNet-50 Homo



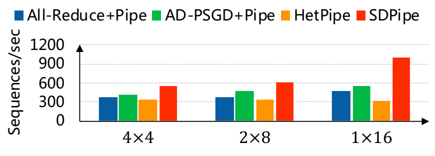
(b) ResNet-50 Hete-A



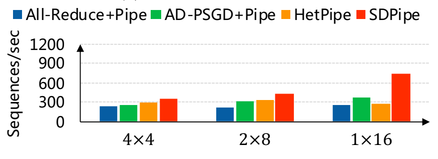
(c) ResNet-50 Hete-B



(d) BERT-base Homo



(e) BERT-base Hete-A



(f) BERT-base Hete-B

Figure 4: Throughput with different pipeline configurations ($M \times N$) under homogeneous and heterogeneous settings.

Fault Tolerance. To extend our prototype system to environments with dynamic availability (e.g., spot instance), we need automatic recovery mechanisms [8, 57] to store the training status when node preemption is going to happen and recover it back. Then our proposed SDPipe could be applied by treating unavailable nodes as heterogeneous stragglers with zero training throughput. Currently, our implementation does not include these fault tolerance functionalities and we leave them as our future work.

5 EXPERIMENTS

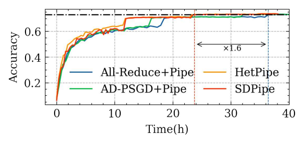
5.1 Experimental Setup

Baselines. We select three related baselines for comparisons: *All-Reduce+Pipe* and *AD-PSGD+Pipe* [35] perform the decentralized schemes for each stage among different pipelines. Unlike All-Reduce+Pipe, AD-PSGD+Pipe synchronizes after each mini-batch but with only a neighboring worker in the same stage. *HetPipe* [49] is one of most closely related works. It integrates PipeDream-Flush of a virtual worker and performs the traditional SSP protocol with the centralized PS. It has a staleness threshold s and we use the same number of stages as suggested.

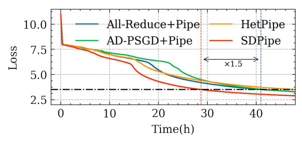
To make a fair comparison, we implement all these three baselines by using the most popular PipeDream-Flush 1F1B scheduling for better hardware utilization and efficiency. We assume that we do not have the dynamic heterogeneity pattern in advance so we adopt a default homogeneous model partitioning as suggested by [26].

Experimental Setting. We evaluate SDPipe on two kinds of representative workloads, including both ResNet-50 [23] on ImageNet-1K [17] and BERT-base [18] on English Wikipedia datasets. We evaluate them on a GPU cluster, and each node is equipped with 8 Nvidia Tesla A100 cards and 10 Gb Ethernet. For the ImageNet dataset, to obtain the standard terminated test accuracy of 74% as reported in [23]. For the BERT-base, we only perform the pre-training process and determine to terminate when the training loss is around 3. The training hyper-parameters are listed in the artifacts [5]. We set P to be 10 and T to be $1/N$ of the iteration time. All experiments are executed five times, and the average results are reported.

Heterogeneity Simulation. Motivated by existing heterogeneous training approaches [22, 37, 38], we follow them to simulate the real straggler patterns by injecting sleep commands into workers. We simulate the dynamic heterogeneous settings in our experiments by adding certain costs to each worker independently with a certain possibility (i.e., 10%) at every iteration. Specifically, we provide two



(a) ResNet-50 Convergence



(b) BERT-base Convergence

Figure 5: Convergence performance comparison with 4×4 workers under heterogeneous-B setting.

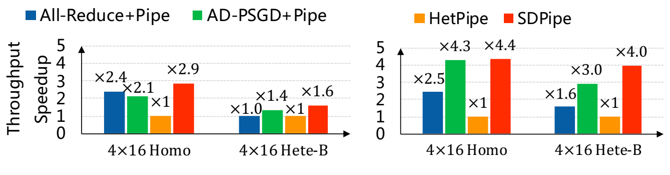
simulated heterogeneous conditions: A) 0.1s on ResNet and 0.2s on BERT; B) 0.3s on ResNet and 0.6s on BERT. These latency data are collected from a real cloud provided by our industrial partner.

5.2 Throughput

We first investigate the training throughput (i.e., the number of images/sequences processed per second for ResNet-50/BERT-based respectively) with various pipeline-data parallelism configurations $M \times N$ under different heterogeneous settings. In a homogeneous environment, Figure 4a and 4d illustrate both SDPipe and decentralized schemes provide superior performance than HetPipe, as the communication is more evenly distributed than centralized PS. We also see that SDPipe tends to outperform decentralized ones as the data parallelism increases. This is because the higher data-parallelism degree implies the more arrival time variation, which could benefit from our adaptive group scheduling. While HetPipe tends to perform worse since the increasing workers bring central communication overheads. As increasing the heterogeneity, All-Reduce+Pipe is heavily affected and gradually outperformed by the others. All-Reduce+Pipe and AD-PSGD+Pipe could even become slower than HetPipe, which is also suffering from the heterogeneity. Here AD-PSGD fails to beat All-Reduce in some homo-settings because of the random communication topology and group scheduling overheads. By contrast, SDPipe significantly outperforms all these baselines. It only incurs a 1.4× slowdown in hete-B setting, still considerably better than pipeline-parallel training via All-Reduce or AD-PSGD with up to 4.3× and 2.8× speedup, respectively.

5.3 Convergence Speed

We next examine the convergence performance for all baselines with 4×4 workers (i.e., 4 pipelines and each consists of 4 stages). Figure 5 illustrate the convergence curves under the heterogeneous-B setting. With the high throughput and the convergence enforcement, SDPipe almost always achieves the best performance compared to the other baselines under different settings. For ResNet-50,



(a) ResNet-50 **(b) BERT-base**

Figure 6: The throughput speedups compared with HetPipe when extending to 64 workers (16 pipelines).

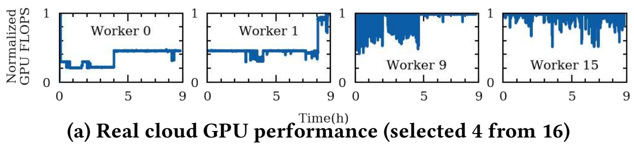
SDPipe is close to HetPipe as the model synchronization overhead has not become a bottleneck for 4×4 worker configuration. However, for the larger BERT-base model, we see that SDPipe outperforms HetPipe by around 1.6× due to central PS's communication.

The evaluation also verifies that those two decentralized baselines are vulnerable to system heterogeneity [67]. Since All-Reduce +Pipe is bounded by the slowest worker while AD-PSGD+Pipe is limited by small group size (e.g., 2), we adapt a synchronization unit that allows for groups of any size while excluding dynamic stragglers. As a result, we have achieved fast propagation of model parameter updates among workers. Although AD-PSGD+Pipe achieves higher throughput than All-Reduce+Pipe on heterogeneous settings, it still cannot bring significant overall speedups due to its poor convergence efficiency. Overall, SDPipe achieves about 1.5× end-to-end convergence speedup compared to decentralized ones.

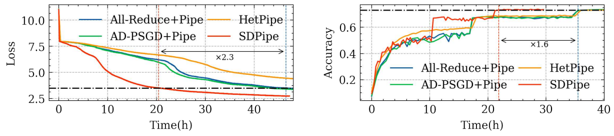
We also have another interesting finding from the evaluation results in Figure 5b. HetPipe and AD-PSGD+Pipe take more time than All-Reduce+Pipe to the target loss, even though they have higher throughput as shown in Figure 4f. In other words, these two heterogeneity-aware baselines can not accelerate the training process for BERT-base model under a severe dynamic heterogeneous environment. This is because the asynchrony involved by AD-PSGD+Pipe and HetPipe significantly affects the convergence quality and requires more training iterations. Compared to these methods, SDPipe utilizes the semi-decentralized design, showing advantages in both hardware and statistical efficiency [67] and resulting in a more significant overall speedup. Note that, given long enough time budgets, all approaches would reach similar final convergent results. To make a meaningful comparison, we eliminate the extreme convergence results after the target accuracy/loss.

5.4 Speedup on Large Cluster Scale

The aforementioned experiments were all conducted over 16 workers. Next, we investigate how our throughput speedup over other baselines is improved using a large-scale cluster including 64 workers. Given more workers, the communication bottleneck of the parameter server becomes more apparent for HetPipe. Figure 6 shows the speedup of SDPipe and decentralized baselines over HetPipe with 64 (16×4) workers under both homogeneous and heterogeneous-B settings. Compared to those in 16 (4×4) workers, we see that the speedups of both SDPipe and decentralized baselines are higher than HetPipe due to its central bottleneck facing more workers. However, we see that other decentralized baselines suffer from significant performance degradation in heterogeneous settings. Compared with those decentralized baselines, SDPipe provides a stable high speedup through adaptive scheduling.



(a) Real cloud GPU performance (selected 4 from 16)



(b) BERT-base **(c) ResNet-50**

Figure 7: Convergence performance comparison with 4 × 4 workers under real cloud heterogeneous setting.

5.5 Performance on Real Cloud GPUs

To evaluated our approach on a real heterogeneous environment, we provide a further comparison over commodity clouds facing dynamic GPU sharing. As the rising of single GPU's performance, several public GPU cloud providers start to support virtual GPU containers, such as Vultr [6] and Alibaba Cloud [3]. They allow users to apply for a container with only a partial sharing GPU device (e.g., 1/2 or even 1/20 of a single GPU). Such a fine-grained GPU service not only helps to improve the resource utilization but also saves users' costs. Specifically, we launch 16 GPU instances on a popular GPU cloud provider [6]. Each instance is a virtualized GPU container that contains a 1/20 NVIDIA A100 GPU. Since the performance of sharing GPUs could be affected by the cluster scheduling behaviors, we observe significant dynamic heterogeneity among these instances as shown in Figure 7a (full 16 workers' results are in [5]). To make a comparable evaluation, we record the observed real GPU performance changes (i.e., running FLOPS normalized by the peak value) and replay them on dedicated GPUs to run our experiments (i.e., adding extra overheads to each GPU worker during training to reproduce its performance slowdown). Figure 7b and Figure 7c show that our proposed SDPipe still outperforms these baselines. Compared with our simulated experiments (i.e., heterogeneous-B setting), the overall speedups are more remarkable (×2.3 and ×1.6 on BERT-base and ResNet-50 respectively) because the real heterogeneous conditions are much severer.

6 CONCLUSIONS

This paper presented the design and implementation of SDPipe, an efficient pipeline-parallel framework for training large DNN models in heterogeneous settings. At its core, SDPipe proposed the semi-decentralized framework, to provide high performance and heterogeneity tolerance. SDPipe used adaptive group scheduling and sync-graph connectivity enforcement to achieve the full performance potential of such framework. Comprehensive evaluation demonstrated SDPipe's high throughput and fast convergence compared to both existing centralized and decentralized schemes, especially in more heterogeneous and larger-scale GPU clusters.

ACKNOWLEDGMENTS

This work is supported by National Key R&D Program of China (2022ZD0116315), National Natural Science Foundation of China (61832001,U22B2037), and PKU-Tencent joint research Lab. Bin Cui is the corresponding author.

REFERENCES

[1] 2017. PyTorch. https://github.com/pytorch/examples/tree/master/imagenet.

[2] 2021. NCCL. https://developer.nvidia.com/nccl.

[3] 2023. Alibaba Cloud Virtual GPU Instance. https://www.alibabacloud.com/help/en/elastic-gpu-service/latest/vgpu-accelerated-instance-families.

[4] 2023. SDPipe. https://github.com/Hsword/VLDB2023_SDPipe.

[5] 2023. SDPipe Artifacts and Proofs. https://github.com/Hsword/VLDB2023_SDPipe/blob/main/VLDB2023_SDPipe_Artifacts_and_Proofs.pdf.

[6] 2023. Vultr. https://www.vultr.com.

[7] 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.

[8] Sanjith Athlur, Nitika Saran, Muthian Sivathanu, Ramachandran Ramjee, and Nipun Kwatra. 2021. Varuna: scalable, low-cost training of massive deep learning models. *Proceedings of the Seventeenth European Conference on Computer Systems* (2021).

[9] Sanjith Athlur, Nitika Saran, Muthian Sivathanu, Ramachandran Ramjee, and Nipun Kwatra. 2022. Varuna: scalable, low-cost training of massive deep learning models. In *EuroSys*. ACM, 472–487.

[10] Tal Ben-Nun and Torsten Hoefler. 2019. Demystifying Parallel and Distributed Deep Learning: An In-depth Concurrency Analysis. *ACM Comput. Surv.* 52, 4 (2019), 65:1–65:43.

[11] Rishi Bommasani, Drew A. Hudson, Ehsan Adeli, Russ Altman, Simran Arora, Sydney von Arx, Michael S. Bernstein, Jeannette Bohg, Antoine Bosselut, Emma Brunskill, Erik Brynjolfsson, Shyamal Buch, Dallas Card, Rodrigo Castellon, Niladri S. Chatterji, Annie S. Chen, Kathleen Creel, Jared Quincy Davis, Dorottya Demszky, Chris Donahue, Moussa Doumbouya, Esin Durmus, Stefano Ermon, John Etchemendy, Kawin Ethayarajh, Li Fei-Fei, Chelsea Finn, Trevor Gale, Lauren Gillespie, Karan Goel, Noah D. Goodman, Shelby Grossman, Neel Guha, Tatsunori Hashimoto, Peter Henderson, John Hewitt, Daniel E. Ho, Jenny Hong, Kyle Hsu, Jing Huang, Thomas Icard, Saahil Jain, Dan Jurafsky, Pratyusha Kalluri, Siddharth Karamcheti, Geoff Keeling, Fereshte Khani, Omar Khattab, Pang Wei Koh, Mark S. Krass, Ranjay Krishna, Rohith Kuditipudi, and et al. 2021. On the Opportunities and Risks of Foundation Models. (2021). arXiv:2108.07258

[12] Alexander Borzunov, Dmitry Baranchuk, Tim Dettmers, Max Ryabinin, Younes Belkada, Artem Chumachenko, Pavel K. Samygin, and Colin Raffel. 2022. Petals: Collaborative Inference and Fine-tuning of Large Models. *ArXiv* abs/2209.01188 (2022).

[13] Léon Bottou, Frank E. Curtis, and Jorge Nocedal. 2018. Optimization Methods for Large-Scale Machine Learning. *SIAM Rev.* 60, 2 (2018), 223–311.

[14] J. Chen, Rajat Monga, S. Bengio, and R. Józefowicz. 2016. Revisiting Distributed Synchronous SGD. *ArXiv* abs/1702.05800 (2016).

[15] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. *CoRR* abs/1512.01274 (2015). arXiv:1512.01274

[16] Jichan Chung, Kangwook Lee, Ramtin Pedarsani, Dimitris Papailiopoulos, and Kannan Ramchandran. 2017. Ubershuffle: Communication-efficient data shuffling for sgd via coding theory. *NeurIPS*.

[17] 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.

[18] 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.

[19] 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.

[20] Runsheng Guo, Victor Guo, Antonio Kim, Josh Hildred, and Khuzaima Daudjee. 2022. Hydrozoa: Dynamic Hybrid-Parallel DNN Training on Serverless Containers. In *MLSys*.

[21] William L. Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive Representation Learning on Large Graphs. In *NeurIPS*. 1024–1034.

[22] Aaron Harlap, Henggang Cui, Wei Dai, Jinliang Wei, Gregory R Ganger, Phillip B Gibbons, Garth A Gibson, and Eric P Xing. 2016. Addressing the straggler problem for iterative convergent parallel ML. In *SoCC*. 98–111.

[23] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *CVPR*. 770–778.

[24] 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.

[25] 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.

[26] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Xu Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. 2019. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. In *NeurIPS*. 103–112.

[27] Jiawei Jiang, Bin Cui, Ce Zhang, and Lele Yu. 2017. Heterogeneity-aware Distributed Parameter Servers. In *SIGMOD*. 463–478.

[28] Jiawei Jiang, Shaoduo Gan, Yue Liu, Fanlin Wang, Gustavo Alonso, Ana Klimovic, Ankit Singla, Wentao Wu, and Ce Zhang. 2021. Towards Demystifying Serverless Machine Learning Training. In *SIGMOD*. ACM, 857–871.

[29] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. 2020. A Unified Architecture for Accelerating Distributed DNN Training in Heterogeneous GPU/CPU Clusters. In *USENIX Symposium on Operating Systems Design and Implementation.*

[30] Paresh Kharya and Ali Alvi. 2021. Using DeepSpeed and Megatron to Train Megatron-Turing NLG 530B, the World's Largest and Most Powerful Generative Language Model. *NVIDIA Developer Blog* (2021).

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] Xiangru Lian, Wei Zhang, Ce Zhang, and Ji Liu. 2018. Asynchronous Decentralized Parallel Stochastic Gradient Descent. In *ICML*, Vol. 80. 3049–3058.

[36] 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

[37] Qinyi Luo, Jiaao He, Youwei Zhuo, and Xuehai Qian. 2020. Prague: High-Performance Heterogeneity-Aware Asynchronous Decentralized Training. In *ASPLOS*. 401–416.

[38] Xupeng Miao, Xiaonan Nie, Yingxia Shao, Zhi Yang, Jiawei Jiang, Lingxiao Ma, and Bin Cui. 2021. Heterogeneity-Aware Distributed Machine Learning Training via Partial Reduce. In *SIGMOD*. ACM, 2262–2270.

[39] Xupeng Miao, Xiaonan Nie, Hailin Zhang, Tong Zhao, and Bin Cui. 2022. Hetu: A highly efficient automatic parallel distributed deep learning system. *Sci. China Inf. Sci.* (2022). https://doi.org/10.1007/s11432-022-3581-9

[40] Xupeng Miao, Yujie Wang, Youhe Jiang, Chunan Shi, Xiaonan Nie, Hailin Zhang, and Bin Cui. 2023. Galvatron: Efficient Transformer Training over Multiple GPUs Using Automatic Parallelism. *Proc. VLDB Endow.* 16, 3 (2023), 470–479. https://doi.org/10.14778/3570690.3570697

[41] Xupeng Miao, Hailin Zhang, Yining Shi, Xiaonan Nie, Zhi Yang, Yangyu Tao, and Bin Cui. 2022. HET: Scaling out Huge Embedding Model Training via Cache-enabled Distributed Framework. *Proc. VLDB Endow.* 15, 2 (2022), 312–320.

[42] Jayashree Mohan, Amar Phanishayee, Janardhan Kulkarni, and Vijay Chidambaram. 2022. Synergy: Resource Sensitive DNN Scheduling in Multi-Tenant Clusters. *OSDI* (2022).

[43] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. 2019. PipeDream: generalized pipeline parallelism for DNN training. In *SOSP*. 1–15.

[44] Deepak Narayanan, Amar Phanishayee, Kaiyu Shi, Xie Chen, and Matei Zaharia. 2021. Memory-Efficient Pipeline-Parallel DNN Training. In *ICML*. 7937–7947.

[45] Angelia Nedic, Alexander Olshevsky, Asuman E. Ozdaglar, and John N. Tsitsiklis. 2009. On Distributed Averaging Algorithms and Quantization Effects. *IEEE Trans. Autom. Control.* 54, 11 (2009), 2506–2517.

[46] Angelia Nedic and Asuman E. Ozdaglar. 2009. Distributed Subgradient Methods for Multi-Agent Optimization. *IEEE Trans. Autom. Control.* 54, 1 (2009), 48–61.

[47] Xiaonan Nie, Yi Liu, Fangcheng Fu, Jinbao Xue, Dian Jiao, Xupeng Miao, Yangyu Tao, and Bin Cui. 2023. Angel-PTM: A Scalable and Economical Large-scale Pre-training System in Tencent. *Proceedings of the VLDB Endowment* (2023).

[48] Xiaonan Nie, Xupeng Miao, Zilong Wang, Zichao Yang, Jilong Xue, Lingxiao Ma, Gang Cao, and Bin Cui. 2023. FlexMoE: Scaling Large-scale Sparse Pre-trained Model Training via Dynamic Device Placement. In *SIGMOD*. ACM. https://doi.org/10.1145/3588964

[49] Jay H Park, Gyeongchan Yun, M Yi Chang, Nguyen T Nguyen, Seungmin Lee, Jaesik Choi, Sam H Noh, and Young-ri Choi. 2020. {HetPipe}: Enabling Large {DNN} Training on (Whimpy) Heterogeneous {GPU} Clusters through Integration of Pipelined Model Parallelism and Data Parallelism. In *ATC*. 307–321.

[50] Adam Paszke and Sam Gross. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *NeurIPS*. 8024–8035.

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

[52] Sundhar Srinivasan Ram, Angelia Nedic, and Venugopal V. Veeravalli. 2009. Asynchronous gossip algorithms for stochastic optimization. In *CDC*. IEEE, 3581–3586.

[53] S. Sundhar Ram, Angelia Nedic, and Venugopal V. Veeravalli. 2010. Distributed Stochastic Subgradient Projection Algorithms for Convex Optimization. *J. Optim. Theory Appl.* 147, 3 (2010), 516–545.

[54] Alexander Sergeev and Mike Del Balso. 2018. Horovod: fast and easy distributed deep learning in TensorFlow. *CoRR* abs/1802.05799 (2018).

[55] Karen Simonyan and Andrew Zisserman. 2015. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *ICLR*.

[56] Sebastian U. Stich. 2019. Local SGD Converges Fast and Communicates Little. In *ICLR*. OpenReview.net.

[57] John Thorpe, Pengzhan Zhao, Jon Eyolfson, Yifan Qiao, Zhihao Jia, Minjia Zhang, Ravi Netravali, and Guoqing Harry Xu. 2022. Bamboo: Making Preemptible Instances Resilient for Affordable Training of Large DNNs. *ArXiv* abs/2204.12013 (2022).

[58] John Thorpe, Pengzhan Zhao, Jonathan Eyolfson, Yifan Qiao, Zhihao Jia, Minjia Zhang, Ravi Netravali, and Guoqing Harry Xu. 2023. Bamboo: Making Preemptible Instances Resilient for Affordable Training of Large DNNs. *NSDI* (2023).

[59] Sonal Tuteja and Rajeev Kumar. 2022. A Unification of Heterogeneous Data Sources into a Graph Model in E-commerce. *Data Sci. Eng.* 7, 1 (2022), 57–70. https://doi.org/10.1007/s41019-021-00174-0

[60] Guozheng Wang, Yongmei Lei, Zeyu Zhang, and Cunlu Peng. 2023. A Communication Efficient ADMM-based Distributed Algorithm Using Two-Dimensional Torus Grouping AllReduce. *Data Sci. Eng.* 8, 1 (2023), 61–72. https://doi.org/10.1007/s41019-022-00202-7

[61] Jianyu Wang and Gauri Joshi. 2019. Cooperative SGD: A Unified Framework for the Design and Analysis of Communication-Efficient SGD Algorithms. In *ICML Workshop*.

[62] Adam Weingram, Yuke Li, Hao Qi, Darren Ng, Liuyao Dai, and Xiaoyi Lu. 2023. xCCL: A Survey of Industry-Led Collective Communication Libraries for Deep Learning. *J. Comput. Sci. Technol.* 38, 1 (2023), 166–195. https://doi.org/10.1007/s11390-023-2894-6

[63] Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang, Cheng Wang, Jian He, Yong Li, Liping Zhang, Wei Lin, and Yu Ding. 2022. MLaaS in the Wild: Workload Analysis and Scheduling in Large-Scale Heterogeneous GPU Clusters. In *NSDI*. USENIX Association, 945–960.

[64] 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.

[65] Binhang Yuan, Yongjun He, Jared Quincy Davis, Tianyi Zhang, Tri Dao, Beidi Chen, Percy Liang, Christopher Re, and Ce Zhang. 2022. Decentralized Training of Foundation Models in Heterogeneous Environments. In *Advances in Neural Information Processing Systems*, Alice H. Oh, Alekh Agarwal, Danielle Belgrave, and Kyunghyun Cho (Eds.). https://openreview.net/forum?id=UHoGOaGjEq

[66] Kun Yuan, Qing Ling, and Wotao Yin. 2016. On the Convergence of Decentralized Gradient Descent. *SIAM J. Optim.* 26, 3 (2016), 1835–1854.

[67] Ce Zhang and Christopher Ré. 2014. DimmWitted: A Study of Main-Memory Statistical Analytics. *PVLDB* 7, 12 (2014), 1283–1294.

[68] Shixiong Zhao, Fanxin Li, Xusheng Chen, Tianxiang Shen, Li Chen, Sen Wang, Nicholas Zhang, Cheng Li, and Heming Cui. 2022. NASPipe: high performance and reproducible pipeline parallel supernet training via causal synchronous parallelism. In *ASPLOS*. ACM, 374–387. https://doi.org/10.1145/3503222.3507735

[69] Yihao Zhao, Yuanqiang Liu, Yanghua Peng, Yibo Zhu, Xuanzhe Liu, and Xin Jin. 2022. Multi-resource interleaving for deep learning training. In *SIGCOMM*. ACM, 428–440.

[70] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Joseph E Gonzalez, et al. 2022. Alpa: Automating Inter-and Intra-Operator Parallelism for Distributed Deep Learning. *OSDI* (2022).

[71] Martin Zinkevich, M. Weimer, Alex Smola, and L. Li. 2010. Parallelized Stochastic Gradient Descent. In *NeurIPS*.