Mitigating Stragglers in the Decentralized Training on Heterogeneous Clusters

Donglin Yang
University of North Carolina at Charlotte
Charlotte, NC, USA
dyang33@uncc.edu

Wei Rang
University of North Carolina at Charlotte
Charlotte, NC, USA
wrang@uncc.edu

Dazhao Cheng
University of North Carolina at Charlotte
Charlotte, NC, USA
Dazhao.Cheng@uncc.edu

ABSTRACT

Decentralized algorithms, e.g., AllReduce, have been widely applied as the synchronization strategy for data-parallel distributed deep learning due to its superior performance over centralized ones. The synchronous Stochastic Gradient Descent (SGD) approach guarantees accuracy for various deep learning models, but its performance suffers from stragglers, i.e., "long-tail effects." The straggler can be caused by the inherent load imbalance from workloads or system heterogeneity. Despite existing optimizations to support centralized algorithms against stragglers, little effort has been explored in decentralized training algorithms.

This paper proposes a Randomized Non-blocking AllReduce (RNA) protocol to mitigate the straggler problem. To avoid "long-tail effects" brought by the strict barrier in the AllReduce, we propose a decentralized, relaxed, and randomized sampling approach to implement partial AllReduce operation. To handle heterogeneity at a large scale, we combine the traditional Parameter Servers (PS) with AllReduce to implement a hierarchical synchronization mechanism. We theoretically demonstrate the convergence analysis and detail the system implementation. The experiment results on representative deep learning models show nearly 1.8× speedup over the state-of-the-art Horovod and 1.3× speedup over AD-PSGD on a heterogeneous cluster.

CCS CONCEPTS

• **Computer systems organization → Distributed architectures**; • **Software and its engineering**; • **Computing methodologies** → *Parallel algorithms*;

KEYWORDS

Distributed deep learning, Decentralized training, Heterogeneity

ACM Reference Format:
Donglin Yang, Wei Rang, and Dazhao Cheng. 2020. Mitigating Stragglers in the Decentralized Training on Heterogeneous Clusters. In *21st International Middleware Conference (Middleware '20), December 7–11, 2020, Delft, Netherlands.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3423211.3425693

1 INTRODUCTION

Deep learning (DL) [32] has achieved great success in various domains such as image classification [54], natural language processing [10], object detection [55], speech recognition [24], etc. Obtaining accurate deep learning models is a computation-intensive process, which requires large amounts of data and substantial computing capacity. There is a trend to distribute the training process across clusters to accelerate the training process. Distributed training is an iterative process, which adopts the most popular algorithm called mini-batch Stochastic Gradient Descent (SGD) [14] to compute gradients and update models until convergence. Data parallelism [34][13][12] is one of the most popular approaches to distributing the training process. In this strategy, different machines in a distributed environment have a complete copy of the model. The procedure of distributed training is bottlenecked by the parameters communication. In a cluster environment, two typical approaches are widely used to synchronize model parameters at the end of each iteration: centralized [34] and decentralized algorithms [36]. For the centralized approach, nodes are divided into two categories: Parameter Servers (PS) and workers. PS stores the model parameters while workers execute the training process in each iteration. In the decentralized approach, every worker performs the computation and maintains a copy of the parameters. Recently it has been theoretically proven that decentralized approaches can outperform centralized ones [36].

In a distributed fashion, the main performance bottleneck comes from the communication hotspot, which is caused by the frequent access to global models. At the end of each iteration, gradients or parameters are frequently transferred among workers until the model parameters are fully updated. In PS, all nodes have to communicate with central servers, leading to a communication bottleneck. It is reported that more than 90% of iteration time is required for communication for a wide and deep neural network model, e.g., VGG16 [5]. This problem has been alleviated with decentralized algorithms, which implement all-to-all communication logically. This bandwidth-optimal communication protocol has been shown to outperform centralized approaches, especially for neural networks with large models. Moreover, decentralized algorithms can achieve better scalability, which is independent of the number of workers. In this paper, we focus on one of the most popular implementations of decentralized algorithms, Ring AllReduce [49]. The execution of AllReduce follows the Bulk Synchronous Parallel (BSP) model. In it, parallel processes execute the same task at the same iteration, and the generated updates must be synchronized on parameters when all tasks are finished. The strict global barrier at each iteration ensures the model accuracy but makes it vulnerable to "long-tail effects." All processes have to wait for each other to complete the propagation before the AllReduce operation is triggered. The slowest worker bounds the performance. Recently, though many studies have been proposed to overcome straggles in the centralized fashion [5][6][6][61], it falls short in supporting decentralized approaches.

In this paper, we investigate the causes of straggler in terms of computation. Firstly, the straggler can be caused by the inherent load imbalance. A dynamic neural network such as LSTM [18] and RNN [51] model sequences of data (e.g., video and sentences). The computation graph topology depends on input values, whose data samples could have variable shapes. For example, the recurrent structure of the network leads to that the training overhead is proportional to the length of the input video [65]. Also, the sentences in the training dataset for a language model [57] have various input lengths, resulting in an unbalanced workload across different batches. We observe that varying input lengths result in computation load imbalance, leading to the inefficiency during the synchronization phase. Secondly, the heterogeneity from the system itself can also cause "long-tail effects." Shared clusters and clouds often exhibit significant hardware and performance heterogeneity due to multi-tenant interference and continuous machine maintenance [40][6].

Several pieces of research have explored the robustness of deep learning processes [37][39]. In particular, AD-PSGD [37] is the first that proposes to use randomized communication to reduce the effects of stragglers probabilistically. In contrast to waiting for all processes, each worker randomly selects one worker to average parameters between the two. However, this design incurs significant synchronization overhead to ensure atomicity. It also requires manual efforts to avoid scheduling conflicts[39]. Inspired by the fact that deep learning training process is robust to bounded errors, we propose to relax the global barrier without changing the communication graph to mitigate the impact from "long-tail effects." We offer a new solution, called Randomized Non-blocking AllReduce, that allows the AllReduce operations to proceed the synchronization without waiting for the completion of all processes' computation.

The challenge of partial AllReduce lies in when and how to terminate input data processing. For instance, each process is unaware of the progress of others in the existing computing platforms. The implementation of Horovod requires when all of the processes inform gradients' readiness, and then the synchronization is executed. In a global view, the critical challenge to achieve a partial AllReduce is deciding the time to trigger the sync, i.e., determining the number of processes contributing to their updates. It is a trade-off between system efficiency and algorithm efficiency. To tackle this challenge, we adapt the power of two choices load balancing technique [42] to partial schedule synchronization by probing two random processes and determining the synchronization time based on the faster one. In a local view, we use two individual threads to execute computation and communication, through which cross-iteration training is enabled. In a heterogeneous cluster, the deterministic performance difference between machines can not be neglected. To utilize the flexibility of the traditional PS to achieve asynchronous updates, we combine PS with AllReduce architecture to implement a hierarchical synchronization protocol. The convergence analysis shows that the error is bounded and the statistical properties guarantee the convergence of deep neural network models. In a nutshell, we make the following technical contributions:

- We empirically study and demonstrate the effects of inherent workload imbalance and system heterogeneity. Based on the observations, we motivate the need for Non-Blocking AllReduce to relax the global synchronization barrier.
- We present a randomized sampling approach to deploy Non-Blocking AllReduce and detail the workflow to update parameter models using this strategy. Furthermore, we combine the traditional PS with AllReduce to implement hierarchical synchronization in heterogeneous clusters.
- We present the theoretical proof of convergence of asynchronous decentralized training and detail the system implementation to enable cross-iteration for Tensorflow using RNA.
- We implement RNA to perform comprehensive evaluations with various network models. It achieves nearly up to 1.8× speedup compared with existing state-of-the-art implementations.

The rest of this paper is organized as follows. Section 2 gives background and motivations on Non-Blocking AllReduce for distributed deep learning training. Section 3 describes the core design of RNA, and section 4 extends RNA into a large and heterogeneous cluster. Section 5 and 6 detail the convergence analysis and system implementation. Section 7 presents the experimental methodology, and Section 8 reports the evaluation results. Section 9 reviews related work. Section 10 concludes the paper.

2 BACKGROUND AND MOTIVATION

2.1 Distributed Deep Learning

In the training stage of deep learning, Stochastic Gradient Descent (SGD) [47] is adopted to minimize the loss function $f(x)$ over a data set S. In each iteration, parameters x are updated by $x \leftarrow x - \gamma \cdot \nabla_x f(x;\xi)$, where γ represents learning rate, and ξ represents a mini-batch of randomly sampled data from S. With the growing volume of data, it is popular to parallelize the training process in a distributed environment. Multiple parallelism schemes have been proposed recently to distribute the training process: data parallelism [49][34], model parallelism [9], hybrid parallelism [29][58] and pipeline parallelism [27][43]. Among them, data parallelism is the easiest one to be implemented without significant statistical efficiency loss compared with other approaches. Therefore, many popular deep learning frameworks such as TensorFlow [1], PyTorch [45] and MXNet [7] support this approach. Our paper focus on the data parallelism model.

For the data parallelism model, each node processes the randomly sampled input data independently and obtains gradients using the backpropagation algorithm [33]. At the end of each iteration, the obtained gradients from each node need to be gathered around so as to update the global parameter during the synchronization phase. The updated model will be applied in the next iteration and the distributed training process keeps this procedure until the model convergences. Synchronization is an essential part of parallelizing the training process and plays a critical role in achieving better scalability in a heterogeneous environment.

2.2 Existing Synchronization Approaches

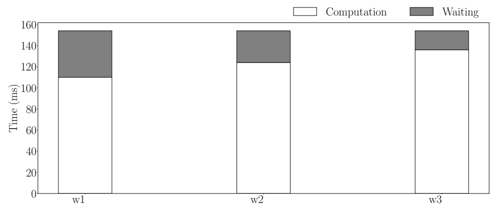
Parameter Servers (PS) is a well-known scheme for parallel SGD execution, which is also called as the centralized algorithm. Parameter Servers and multiple workers are launched in this typical setting. At each iteration, each independent worker obtains gradients based on the SGD and sends the data to the central PS. The central PS will send back the updated model to each work and continue the training process. However, this simple approach has a significant drawback because all the workers have to push/pull parameters from the centralized servers, leading to the communication hotspot. The communication hotspot limits the system scalability. Decentralized training is proposed to alleviate this issue.

Algorithm 1 The decentralized training process.

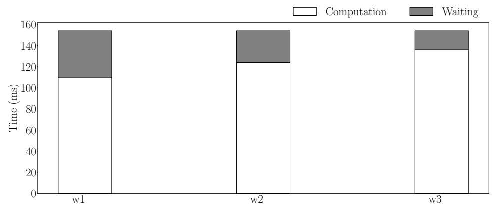
```
Require: A set of workers $M$; the communication topology $G$.
1: for worker $m_i \in M$ do
2:     compute the gradient $g_{k,i} = \nabla_{\varepsilon_{k,i}} f(x_{k,i}; \xi_{k,i})$
3:     average gradients using AllReduce $\overline{g}_k \leftarrow \sum_{i \in M} g_{k,i}$
4:     update parameters $x_{k+1,i} \leftarrow x_{k,i}$ - $\gamma_k \cdot \overline{g}_k$
5: end for
```

As shown in Algorithm 1, in a decentralized setting, there are no central parameter servers. Every worker in this scheme maintains a complete copy of the model parameters. At the end of an iteration, each node sends the obtained gradients to their out-going neighbors according to the communication topology, after which it applies the obtained gradients to the parameter. Ring All-Reduce [49] is one of the most popular implementation, which works in a scatter-and-gather way. In this setting, each worker only communicates with its neighboring sender and receiver in a fixed order during the synchronization phase, forming a logical ring. For a distributed system with N servers, in each step, the worker sends one portion of $\frac{1}{N}$ gradients to its left neighbor and meanwhile, it accepts another $\frac{1}{N}$ gradients from its right neighbor. It then averages the accepted gradients with its local portion, which is called as *Reduce* operations. The averaged portion will be transferred to its left neighbor in the next step during the scatter operations, and meanwhile this worker will accept another averaged part from its right neighbor. After $N-1$ steps, scatter operations are finished, and every worker owns a complete $\frac{1}{N}$ of gradients. Gather operations will be triggered then, which works similarly, but it does not require *Reduce* operation. In each step, each worker replaces its local portion of gradients with the accepted one from its right neighbor. After $N-1$ steps, each worker obtains the whole set of global gradients. It benefits from contention-free communication compared with PS strategy by abandoning the many-to-one communication protocol, which achieves the ideal parallelism within the theoretical upper bound. These procedures guarantee consistent convergence with the expense of introducing a blocking barrier. Workers will always have to wait for the slowest one to finish at each iteration. This means that this distinct communication pattern can only achieve its best performance in a **homogeneous** environment. This is a strict requirement, especially in a shared cloud environment.

To tolerate system heterogeneity, AD-PSGD [37] proposes a random synchronization mechanism to enable the point-to-point communication. Instead of synchronizing with the fixed neighbors specified by the communication topology, a worker performs an atomic model averaging with the randomly selected neighbor, regardless of whether they are in the same iteration or not. Even though the slow workers inevitably have staler parameters, the effects on the training of the global model can be minimized via the probabilistic solution. However, this strategy requires additional system overhead and manual efforts to generate a dynamic communication graph [39].



(a) VGG-16.

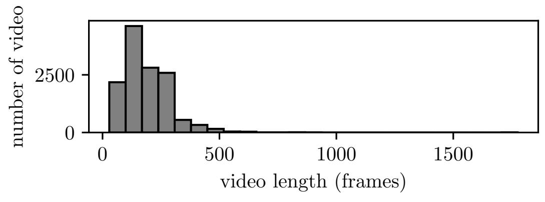


(b) ResNet-56.

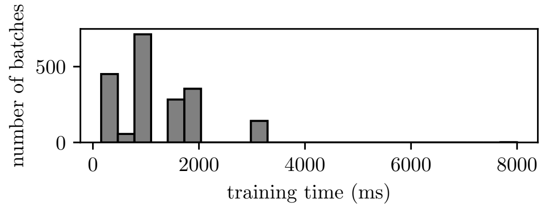
Figure 1: Training time breakdown with different system configurations.

2.3 Challenges and Motivation

2.3.1 Case Study. In a shared cluster, major bottlenecks for distributed deep learning training comes from the straggler problem, which is caused by various heterogeneities. In general, deterministic heterogeneity is relatively common because a large cluster is always configured with different hardware and dynamic capacities. Moreover, DL workloads running in a shared cluster always coexist with other data analytic workloads, which introduces transient slowdowns. The over-subscription of workloads in a cluster further harms the performance of DL training. To motivate our work, we build a toy cluster composing of three NVIDIA GeForce RTX 2080 Ti GPU on three nodes. One node 2 and 3 we inject 10ms and 40ms showdown, respectively. The machines are connected with 10Gb Ethernet. We run ResNet-56 [23] and VGG-16 [50] with the CIFAR-10 dataset [30]. We divide the time a worker spends in one training iteration into two parts: (i) the computation time, to carry out the forward propagation to produce output and backward propagation to obtain gradients; and (ii) the waiting time, including the time for exchanging gradients/parameters with the PS and the blocked time due to synchronization barrier (i.e., the time when the worker is not doing computation nor communication). From Figure 1, though w_1 can complete one iteration faster than other machines, e.g., nearly 2× faster in terms of computation, it has to wait for the slower machines to finish the propagation and then synchronize the obtained gradients with other nodes. This strict requirement for AllReduce

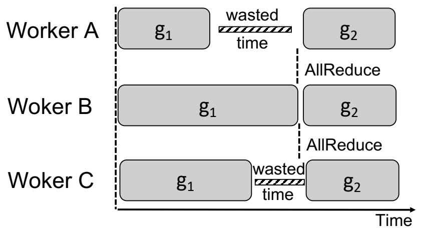


(a) Video length distribution.

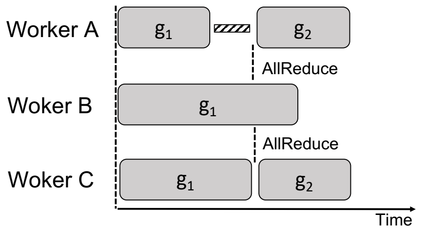


(b) Training time distribution.

Figure 2: Inherent load imbalance from training LSTM on UCF101.



(a) default AllReduce.



(b) Non-blocking AllReduce.

Figure 3: g_1 and g_2 represent gradients from iteration t_1 and t_2, respectively. The length of the bars represents the training time. Worker C is specified as the initiator.

operation degrades the system efficiency, slowing down the training progress. System heterogeneity is common, especially in the cloud environment. The straggler issue will be more severe on a large scale.

Load imbalance from the application also widely exists in the training of deep learning models. We use Inception V3 [53] to extract video features for UCF101 [52], which has 13,320 videos. Figure 2(a) summarizes the distribution of the length of video frames. The lengths of video range from 29 to 1776, with a mean value of 186 and a standard deviation of 97.7. We run a Long Short-Term Memory (LSTM) [26] model to demonstrate our observation. We configure the batch size as 32 and train the model on the GPUs with the same capability. Figures 2(b) illustrates the training time distribution over the 2,000 sampled batches in two epochs to train a 2,048-wide single-layer LSTM model on video frame features. Due to the network's recurrent structure, the computational overhead is proportional to the number of frames in the input video. The training time is distributed from 156 ms to 8000 ms, with a mean runtime of 1,219 ms and a standard deviation of 760 ms. These statistics above show that training an LSTM model for video classification exhibits an inherent load imbalance. Load imbalance is common, especially for dynamic neural networks [64]. Since sequences may have variable length, the cell function is executed for a different number of times for different sequences. In dynamic neural networks, some dependencies depend on input data or parameter values, which is dynamically determined, resulting in the runtime imbalance.

2.3.2 Non-Blocking Reduce Synchronization. The decentralized training follows the Bulk Synchronous Parallel (BSP) model, in which workers synchronize at the end of an iteration, i.e., barrier and proceed after the model parameters have been fully updated by all workers. However, from the above observations, we can learn that this synchronous execution lowers hardware efficiency since fast workers have to wait for stragglers to complete each iteration, wasting computing cycles, which can be illustrated in Figure 3(a). Bounded staleness [25] is an important technique to tolerate the temporary slowdown for centralized algorithms, allowing faster workers to advance to the next iteration within the bounded staleness. It is easy to implement asynchronous synchronization in a centralized scheme because each worker communicates with parameter servers directly and computes gradients independently. However, the distinct communication pattern of the Ring All-Reduce protocol makes it difficult to enforce such a technique directly. Because current implementations require that all the results should be within the same iteration to ensure consistency.

These observations motivate us to propose a Non-Blocking-Reduce mechanism to synchronize partial results without changing the communication graph. Inspired by the fact that the training process is robust concerning bounded errors, we propose to update the gradients without waiting for slower processes to reduce the delay. Instead of waiting for the slower nodes, the faster worker will Reduce the gradients partially from available workers. Figure 3 shows a simple example in a decentralized setting with three workers. Worker A, B, and C are at the iteration t_1 in the beginning. In default, when worker A completes the propagation, since the worker B and C are still in progress, it has to wait for the completion of the other two processes. It is until the slowest process, i.e., B, finishes the propagation, then the AllReduce is executed. The default strategy under-utilizes the resource because faster processes keep idle when waiting for slower processes. In a non-blocking setting, an initiator is randomly selected among these three processes. Suppose that C is selected. When process C finish propagation, the AllReduce operation is enforced, then worker A and C can advance to a new iteration. During the synchronization phase, worker B contributes a $Null$ value to maintain the communication graph. In this way, the waiting time for A is reduced while there is no idle time for C. The overall system efficiency is improved compared with the default strategy. In the next iteration t_2, if both worker A, B and C have available results, the AllReduce operation will synchronize the gradient g_1 from worker B with g_2 from worker A and C though they are in different versions. Through the Non-Blocking-Reduce mechanism, the strict blocking barrier can be relaxed, which can reduce the impact of stragglers. In a cluster with P processes, the probability that any process is specified as the initiator is equal to $\frac{1}{P}$, correspondingly on average, 50% of processes join the collective operation. In this way, the asynchronous synchronization might lower statistical efficiency than the synchronous implementation but it can trade statistical efficiency for system efficiency. However, the above example is too simple and straightforward with only two workers. In the following sections, we will discuss how to extend the Non-Blocking-Reduce mechanism to a cluster-wide scale and heterogeneous environments.

3 RANDOMIZED NON-BLOCKING ALLREDUCE

Traditional synchronous operations in the decentralized training such as AllReduce require a central scheduler to maintain a complete view of all processes. The synchronization operations can only be initiated after the slowest process finishes its computation. In this section, we introduce Randomized Non-blocking AllReduce (RNA), which takes a radically different approach: all processes operate in parallel, and the central scheduler does not maintain any progress state about training. The central scheduler relies on instantaneous progress information acquired from worker machines to initiate a synchronization. When the synchronization is triggered, RNA adopts weighted averaging to local accumulated results and dynamic scaling to the global aggregation to apply the obtained gradients.

3.1 Randomized Partial Collectives

The key to avoiding the "long-tail effects" for Ring AllReduce enforce partial collective operations, which force the slow processes to execute the synchronization. The main difference between the bulk synchronous parallel collective communication and the partial collective communication is when synchronization is triggered. For bulk synchronous parallel, the AllReduce is executed when all workers inform the central scheduler that they are ready to reduce the obtained gradients, e.g., *NEGOTIATE_ALLREDUCE* in Horovod. In this scenario, even a single delayed process affects the job's training time. In contrast to the synchronous mode, in MPI, there is a wait-free operation, which is called partial collective communication [16]. It forces the slow processes to execute the collective communication as soon as there is one process executing it. This process, called the initiator, is in charge of enforcing the others to join the collective communication. In partial collective communication, an *external* activation is allowed to enforce a process to execute the synchronization before it reaches the *internal* activation. The time when the synchronization should be initiated is a trade-off between system and algorithm efficiency. A simple and straightforward strategy is to select an initiator among processes randomly. When a process is elected as the initiator, if it has gradients ready to be reduced, an external activation is broadcast to all the other processes to join the collective operation, regardless of they have finished the propagation or not.

Ideally, random selection can guarantee that at least half of the processes on average can take part in the collective operation and contribute their gradients. For a cluster with N nodes, the probability that any process is selected as the synchronization initiator is $\frac{1}{N}$. Correspondingly, half of the processes on average have gradients ready for AllReduce before the selected initiator sends out the external activation. However, for a workload with a long tail distribution, e.g., as is illustrated in section 2.3.1, stragglers still have a high probability of slowing down the synchronization. Specifically, the expected waiting time is $\frac{1}{1-\rho}$, when there is workload in the queueing system [11], in which ρ represents the computational load.

3.2 Per-process Sampling

Inspired by the power of two choices load balancing techniques [42], RNA implements a power of two choices technique to improve the purely random selection of initiator of AllReduce. It provides low expected waiting time using a stateless, randomized approach. More precisely, For a fixed time T, using q choices, i.e., two here, the waiting time in an initially empty system over T is upper bounded by $\sum_{i=1}^{\infty} \rho^{\frac{q^i-q}{q-1}} - O(1)$, which improves the expected waiting time exponentially compared to the random strategy. The term $O(1)$ is obtained in an initially empty system over the first T units, which may depend on T. The central scheduler randomly selects two processes among the machines and sends a *probe* to each, where a probe is a lightweight RPC. The selected processes can only reply to the probe when they have ready gradients. As long as one of them responds to the central scheduler, the AllReduce operation is initiated. The probe is attached to the iteration identification to avoid the scheduling conflict. When the faster one is replied, another probe is expired. For example, if processes p_i and p_j are selected and finish propagation for the current iteration at the same time, they both reply to the probe. There are two cases: 1) response from p_i has been accepted, the probe for this iteration for p_j is expired; 2) no response has been accepted, faster p_i is accepted and, the probe identification is updated to the next iteration. Two-probe sampling can reduce the response time effectively compared with randomized approach. An additional number of probes cannot improve the performance but harm the performance because of the system overhead from sampling and messaging, which is detailed in Section 8.4.

3.3 Non-blocking AllReduce

The straightforward implementation of Ring AllReduce incurs high inefficiency when stragglers appear. The goal of RNA is to propose a communication primitive that can balance the efficiency between the system and algorithm. When the collective operation is initiated, the synchronization procedure involves updating the weights of contribution from each process. We use $w_{k,i} = 1$ to indicate that the process k at iteration i has gradients to be applied, otherwise, $w_{k,i} =$

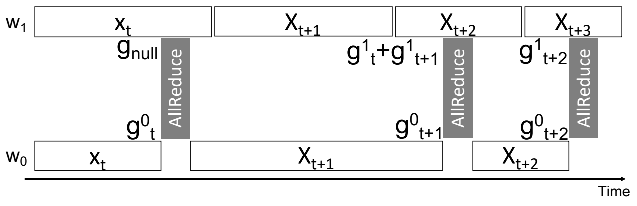


Figure 4: Non-blocking AllReduce: white bars represent computation processes, while grey bars represent communication processes. x_t represents the parameters being used for training, and g_t^0 represents the gradient from processes w_0 at time t.

Algorithm 2 Non-blocking AllReduce.

```
Require: A set of workers $M$; the communication topology $G$.
1: for worker $m_i \in M$ do
2:     compute the gradient $g_{k,i} = \nabla_{\varepsilon_{k,i}} f(x_{k,i}; \xi_{k,i})$
3:     obtain the weight for gradients $W = \frac{1}{\sum w_{k,i}}$
4:     average gradients using Non-blocking AllReduce $\overline{g}_k \leftarrow W \cdot \sum_{i \in M} g_{k,i}$
5:     update parameters $x_{k+1,i} \leftarrow x_{k,i}$ - $\gamma_k \cdot \overline{g}_k$
6: end for
```

0. Then the weight for each process is $W = \frac{1}{\sum w_{k,i}}$. Algorithm 2 illustrates the procedure of RNA. Specifically, according to the Linear Scaling Rule [21], RNA dynamically adjusts learning rate $\gamma_k = \sum w_{k,i} \cdot \gamma$ at each iteration. All other hyper-parameters (weight decay,etc.) are kept unchanged. The implementation of RNA can still leverage the benefit from Ring AllReduce that it can update the weights among processes in $O(M)$ time because it does not change the communication graph. RNA still follows the generalization of the conventional Ring AllReduce in deep learning training among all processes.

Figure 4 summarizes the working examples of RNA. RNA employs two threads to execute computation and communication. In our implementation, computation is done by GPU, while gradients synchronization is by CPU, i.e., MPI. In these examples, we assume that the process w_0 is always selected as the initiator. At iteration t, suppose that w_1 is slower than w_0 and w_1 has no available gradients when w_0 completes propagation. w_1 initiates the AllReduce without waiting for w_1. Since w_1 contributes a g_{null} gradient at this time, RNA adjusts the weight W and updates parameters correspondingly. At iteration $t + 1$, when w_0 triggers AllReduce operation since w_1 catches up with w_0 and has two gradients $g_{t+1,1}$ and $g_{t+2,1}$ available on the communication thread, the accumulated gradients are locally reduced and participate in the collective operations then. We should notice that $g_{t+2,1}$ is updated using the new parameters x_{t+1} while the gradients g_{t+1} uses stale parameter x_t. At iteration $t + 2$, w_1 is faster than the initiator. It does not wait for the completion of w_0 and continue the next iteration. When the initiator w_0 is ready, then AllReduce is performed to update the parameters using g_{t+2}^0 and g_{t+2}^1.

While in some extreme situations, the staleness might be more than two. RNA implements a weighted averaging to reduce the

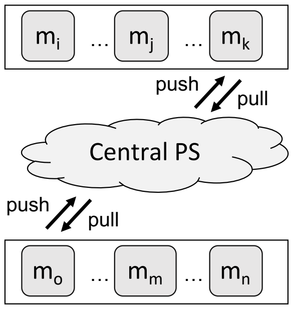


Figure 5: Hierarchical synchronization scheme: m_i represents the i-th worker.

accumulated locally. For work i at iteration k, the locally reduced gradient is $g' = \frac{\sum[t-(k-\tau)+1] \cdot g_t}{\sum[t-(k-\tau)+1]}$, in which g_t is the gradients obtained at iteration t and τ is the largest iteration gap among accumulated results. The weight of an update is linearly associated with its iteration. If some slower processes fall behind others severely, RNA follows the design of bounded staleness [25] to overwrite the stale data and only keep results within the bound.

4 HIERARCHICAL SYNCHRONIZATION IN HETEROGENEOUS CLUSTER

The objective of RNA is to leverage the randomized initiator to avoid the "long-tail effects." However, when this implementation is extended to a large and heterogeneous environment, the deterministic heterogeneity from hardware cannot be negligible. The mechanisms proposed so far are mainly effective in a homogeneous execution environment but do not help with slowdown situations. Slow workers who always fall behind others can enlarge the iteration gaps among workers gradually, resulting in lower accuracy. The best solution is to allow asynchronous synchronization. To achieve that, we combine the decentralized design with the traditional PS implementation, which can be illustrated in Figure 5. The hierarchical AllReduce first groups M machines into N groups, and uses three phases to do update parameters: firstly, each group executes AllReduce operation and updates parameter among the assigned machines in this group following the basic design of RNA. This procedure follows the basic randomized Non-blocking AllReduce; secondly, the averaged gradients among each group is applied to update models using parameter server. The updated parameters from each group is pushed to a central PS from the selected initiator to be averaged, the results are then pulled back to the initiator worker; thirdly, the selected initiator in each iteration executes a broadcast operation within the group to propagate the final result to every process. In this mode, each group can be regarded as a "node" in the traditional PS. In a large scale, it is easy to implement asynchronous synchronization because each group communicates with parameter servers directly and computes gradients independently.

Whether the hierarchical synchronization should be used or not, it depends on both the system performance and application behaviors. To determine whether to choose one or more AllReduce groups, we test a simple condition of $\zeta > \upsilon$, where ζ denotes the difference of the time between the fastest task and the slowest one,

Table 1: Notation.

$\|\|x\|\|^c$	the l_c norm of vector x
$E_\xi(\cdot)$	the expectation of variable ξ
x_k	the model parameter at k-th iteration
ξ	the sampled data from input
$f(\cdot)$	the target function for optimizing
$g(\cdot)$	the gradient function
τ_{ij}	the iteration gap between i, j-th machines
L	the Lipschitzian constant
σ^2	the bounded variance
K	the total number of iterations
$\mathbb{B}$	the parameter weights
x^*	the optimal parameters
η	staled iteration bound

and v is the average time of one iteration of all processes. If $\zeta > v$, we use a two-group configuration. During the group configuration, processes are ranked according to the processing time. The processes with processing time larger than v are regarded as a slower worker. Faster workers are defined in a similar way. Faster and slower workers are partitioned into two subsets. The partition-and-group procedures are recursively performed in each subset until $\zeta \leq v$ is satisfied inside the group.

It is worth noting that the proposed hierarchical synchronization is different from hierarchical AllReduce [28], which is mathematically equivalent to All-Reduce among all workers with acceleration brought by the hierarchical architecture. For RNA, workers end up with different weights after the synchronization for a different group. Compared with the default Ring AllReduce, the deterministic slowdown is avoided, making each group homogeneous. Compared with the traditional PS, the number of workers that communicate with a central server is reduced from M to N, in which $M \gg N$. As a result, the asynchronous synchronization among groups with varied capacities can mitigate network contention.

5 CONVERGENCE ANALYSIS

We next theoretically analyze the convergence rate of RNA. Important notation is summarized in Table1. Based on the weighted average gradients, we make the following assumptions for analysis.

Assumption 1. *For widely used stochastic gradient algorithms:*

- ***(Unbiased Gradient)****:The stochastic gradient $g(x;\xi)$ is unbiased: $E_\xi[g(x;\xi)] = \nabla f(x)$.*
- ***(Bounded Variance)****: The variance of stochastic gradient is bounded: $E_\xi(||g(x;\xi) - \nabla f(x)||^2) \leq \sigma^2, \quad \forall x$.*
- ***(Lipschitzian Gradient)****: The gradient function $\nabla f(\cdot)$ is Lipschitzian, that is to say $||\nabla f(x) - \nabla f(y)|| \leq L||x-y||, \quad \forall x, \forall y$.*

Assumption 2. ***Bounded delay****: the delay for updating the gradient value among all machines is bounded, which means* $\max \tau_{ij} \leq \eta$.

Convergence bound: AllReduce spreads the reduced gradients. With the Non-Blocking Reduce mechanism, fast machines use the staled gradient values from slow ones to update its parameters, while slow machines will utilize gradients from future iterations results obtained from faster ones. We uniformly represent the mixed-version gradients for these two conditions as $G\left(x_{k+\tau_{kj}}, \xi_{k+\tau_{kj}}\right)$, where τ_{kj} is the iteration gap to the j-th machine. The positive τ_{kj} represents gradients from the fast worker and, the negative one is from a slow one. Based on two assumptions above, we obtain the following convergence bound:

Theorem 5.1. *The step length sequence $\{\gamma_k\}_{k=1,\ldots,K}$ in algorithm satisfies*

$$\sum_{k=1}^{K}\left(\gamma_k^2\left(\frac{L}{2} + L^2\mathbb{B}\eta\sum_{\kappa=1}^{\eta}\gamma_{k+\kappa}\right) - \frac{\gamma_k}{2\mathbb{B}}\right) \leq 0. \tag{1}$$

We have the following convergence rate for the training:

$$\sum_{k=1}^{K}\gamma_k E||\nabla f(x_k)||^2 \leq \frac{2\left(f(x_1) - f(x^*)\right)}{\mathbb{B}} + \sum_{k=1}^{K}\left(\gamma_k^2 L + 2L^2\mathbb{B}\gamma_k\sum_{j=k-T}^{k-1}\gamma_j^2\right)\sigma^2. \tag{2}$$

The convergence rate is bounded, which satisfies the same convergence properties as the asynchronous parameter server approach [62].

Independent staleness: With the guarantee of the convergence bound, we further analyze that the convergence rate is independent of the staled parameters η after a sufficient number of iterations. We let the step length γ_k as a constant value, and we can obtain the corollary:

Theorem 5.2. *The delay parameter η is bounded by:*

$$\frac{4\mathbb{B}L(f(x_1) - f(x^*))}{\sigma^2}(\eta+1)^2 \leq K. \tag{3}$$

We set the step length γ_k to be a constant γ:

$$\gamma = \sqrt{\frac{f(x_1) - f(x^*)}{\mathbb{B}LK\sigma^2}}. \tag{4}$$

after substituting the upper bound:

$$\gamma L + 2L^2\mathbb{B}\gamma^2\eta \leq \frac{1}{2\mathbb{B}(\eta+1)} + \frac{\eta}{2\mathbb{B}(\eta+1)^2} \tag{5}$$

$$= \frac{2\eta+1}{2\mathbb{B}(\eta+1)^2} = \frac{1}{2\mathbb{B}}\frac{2\eta+1}{(\eta+1)^2} \tag{6}$$

$$\leq \frac{1}{2\mathbb{B}} \tag{7}$$

According to theorem 1, we set the step length γ_k to be a constant γ. Then we can obtain the following convergence rate:

$$\sum_{k=1}^{K}\gamma E||\nabla f(x_k)||^2 \leq \frac{2\left(f(x_1) - f(x^*)\right)}{M} + \sum_{k=1}^{K}\left(\gamma^2 L + 2L^2 M\gamma\sum_{j=k-T}^{k-1}\gamma^2\right)\sigma^2 \tag{8}$$

which is equivalent to:

$$\frac{1}{K}\sum_{k=1}^{K}E||\nabla f(x_k)||^2 \leq 4\sqrt{\frac{\left(f(x_1) - f(x^*)\right)L\sigma^2}{\mathbb{B}K}}. \tag{9}$$

Discussion According to the Theorem 5.2, we can find that when the K is large enough, $\frac{4\mathbb{B}L(f(x_1)-f(x^*))}{\sigma^2}(\eta+1)^2$ is greater than $O(\eta^2)$. We make the following conclusion regarding the bound. First, because we analyze non-convex objectives, for a given sequence of learning rates, the algorithm will converge to a point of negligible gradient. Convergence can be achieved by this algorithm asymptotically. Second, The convergence rate can be achieved by $O(\frac{1}{\sqrt{\mathbb{B}K}})$. The maximum delay and the number of "missing" gradients per iteration can be minimized. It can be concluded that the convergence rate is guaranteed while it requires the additional performance cost of synchronization with the slower convergence. Since the communication cost is also independent of nodes according to the analysis in [36], our optimization techniques can achieve bandwidth-optimal performance with partial synchronization, which will be further empirically confirmed by evaluations in Section 8.

6 IMPLEMENTATION DETAILS

RNA is implemented using C++11 and Python on top of Horovod. We implement the key functionality of partial AllReduce in the package *controller*, which serves as the coordinator to initiate and perform the AllReduce to average gradients among processes. *Controller* resides in the root node in the cluster, which serves as a centralized mechanism to decide the time to execute the decentralized AllReduce. A plugin is developed for TensorFlow to enable the cross-iteration feature. As for the hierarchical synchronization, we follow PS-lite [17] to implement asynchronous communication, which provides flexible and high-performance operations such as zero-copy *push* and *pull*.

Controller. We implement partial AllReduce using MPI only now because RNA separates communication from computation so as to avoid the resource contention for GPU, removing barriers from communication. In default, the controller records the count of received tensors at each iteration. Only if it is equal to the number of participated processes, then the synchronization is initiated. Then AllReduce is performed by Open MPI to synchronize the obtained gradients. RNA selects one process as the initiator based on the randomized algorithm for each iteration. When the initiator has tensors ready for reduction, the AllReduce is performed by the background MPI. As for other processes, RNA will *sum* the gradients locally if there are multiple available tensors, which are from different iterations. As for stragglers, each process allocates a *null* gradient, i.e., *null* tensor in TensorFlow, as the input by default, whose size and shapes are exactly identical to each other among processes. After each AllReduce operation, the input gradients are overwritten by a *null* gradient so as to avoid using outdated gradients. When the AllReduce is completed, RNA returns a callback with the *iterationID* via *EnqueueTensorAllreduce*. The *iterationID* records the step of synchronization. Also, the returned output gradients overwrite the previous results on each worker. Note that all of the above input and output gradients are cached on the CPU memory. NCCL [44] can be applied to synchronize gradients among remote GPUs, which requires additional GPU memory buffer among GPUs to cache input and output gradients individually.

TensorFlow plugin. In default, Horovod wraps TensorFlow optimizer in DistributedOptimizer, which is an opt-in graph optimization module. It supports altering runtime behavior of graph execution, such as instrumenting OpKernel implementation, adding and removing data, or control dependencies of graph nodes. The TensorFlow plugin goes through the data-flow graph to obtain gradients before its execution at each iteration. To enable cross-iteration training, we create two TensorFlow ops. The *WriteOp* caches the obtained gradient on the CPU memory. If there is a *null* tensor, it will be replaced by the new input. If there are input tensor waiting for reduction, it will be accumulated. To avoid being block by the default *allreduce*(), a new kernel, i.e., *ReadOp*, is created. It first checks if there is a new output tensor available, according to the *iterationID*. If yes, the new gradients are copied into the GPU memory to replace the local gradients via PCIe. Otherwise, local gradients are used to fit the deep learning model. If RNA with hierarchical synchronization, the *get_weight*() will write the parameters to CPU buffers after *apply_gradient*() is finished. Then the updated tensors are pulled back. RNA overrides the *set_weight*() API that is inherited from Tensorflow *optimizer* to use the averaged gradients from CPU to continue training. With these two new kernels, the computation of TensorFlow does not necessarily wait for the completion of the communication.

Hierarchical synchronization. A parameter server (PS) is a logically separate device that stores global parameters and provides a key-value interface to workers. Generally, PS approach has the following phases: (1) each worker computes the gradients using its local sampled data and sends them to PS (push); (2) central server aggregates the gradients across workers and updates its parameters (update); (3) Workers synchronize parameters with PS (pull). Following the logic of ps-lite, a *notify_ready* value is returned from the callback by *get_weight*(). The PS only executes the parameter summation, i.e., model averaging, which require additional CPU resources. Fortunately, modern CPU are good at summation operation due to the highly optimized AVX instructions [38]. The additional computation on the CPU will not be the bottleneck. Then *PSPushPull*() is called to perform a *push* and *pull* operation on the output tensors sequentially. Only the selected initiator serves as the "node" and triggers the *PSPushPull*() operation. We applied the default *wait*() API to lock the variables on each worker. After the gradients are aggregated and being updated at the central server, the updated parameters will be pulled back to the initiator. The pulled back parameters are written to the CPU buffer. RNA notifies MPI to broadcast the new tensors among the group via *broadcast*() to overwrite the output tensor with the same *iterationID*, and then the parameters are unlocked. The *set_weight*() API in TensorFlow uses the updated parameters for propagation. The hierarchical synchronization is executed asynchronously across all processes periodically. We leave the frequency tuning as our future work.

7 EVALUATION SETUP

7.1 Testbed Setup

We use a local cluster to evaluate the performance of the proposed RNA model and implemented mechanism. Table 2 lists the hardware configurations of the machines in the cluster. These machines are connected with EDR Infiniband. All nodes in this cluster run Ubuntu Server 16.04 with MPI 4.0.1, Python 3.7, CUDA 10.1, cuDNN 7.6.0, gcc 8.1.0, g++ 8.1.0, TensorFlow 2.1.

Table 2: The hardware configuration of the physical cluster.

Processor	GPU Model	Num.
Intel 3.2GHz Xeon E5-2667 v3	2 × Nvidia Tesla K80 GPUs	4
Intel 2.60GHz Xeon Silver 4112	8 × NVIDIA GTX-1080Ti	2
Intel 3.2GHz Xeon Bronze 3104	2 × Nvidia GTX-2080Ti	4

As for the dynamic system heterogeneity, we follow the experiment setting as Hop [40] to inject delays to simulate the system heterogeneity. Each worker is slowed down randomly in each iteration, where n is the number of workers.

7.2 Deep learning models and datasets

To evaluate the performance of RNA and compare it with other works, we train three kinds of neural network models on real datasets, including image classification, machine translation, and video processing.

7.2.1 image classification. ResNet50 is a convolutional neural network that is 50 layers deep used for image classification. We train ResNet50 [23] model over ImageNet dataset [48], which contains 1,281,167 images to be classified into 1,000 classes. The model contains 25,559,081 parameters. Momentum optimizer is used with $momentum = 0.9$ and $weight\_decay = 5 * 10^{-5}$. The initial learning rate is 0.125 and decays to its 0.1× on epochs 30, 60, 80. The batch size is 128.

VGG16 [50] is a communication-intensive network with thirteen convolution layers of a 3×3 filter with a stride 1. It is a pretty large network, and it has more than 138 million parameters. We train VGG16 on dataset CIFAR-10 [30], whose evaluation setup is batch size: 128, learning rate: 0.1, momentum: 0.9, weight decay: 10^{-4}.

7.2.2 machine translation. Transformers [57] are developed to solve the problem of neural machine translation, which transforms an input sequence to an output sequence. We train Transformer on WMT17 dataset [19], which is an English to German translation dataset. The initial learning rate is set to be 2.0. The model has 61,362,176 trainable parameters. While training the model, we use the varying input length. The samples in the training dataset typically consist of sentences in various lengths. Thus the computation overhead varies with the length of the input and output sentences, leading to unbalance training time.

7.2.3 video processing. RNN [41] is a class of artificial neural networks where connections between nodes form a directed graph along a temporal sequence. LSTM is a kind of recurrent neural networks that are intimately related to input sequences and lists. We train a single, 4096-wide LSTM layer, followed by a 1024 Dense layer, with some dropout in between on UCF101 [52]. The model has 34,663,525 parameters. UCF101 has 13,320 videos from 101 action categories, which gives the largest diversity in terms of actions and with the presence of large variations in camera motion, object appearance, etc. We also use the varying input length to train the model, which is linearly associated with the length of videos. The input batch size is 128.

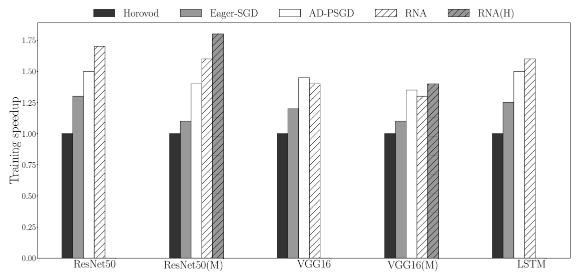


Figure 6: Training speedup by RNA compared to Horovod, eager-SGD, and AD-PSGD. "M" represents the mixed heterogeneity. "H" means that RNA is configured with hierarchical synchronization.

7.3 Approaches and Performance Metrics

We compare the performance of RNA with three other synchronization models: Horovod [49], AD-PSGD [37], and eager-SGD [35]. Horovod is selected as the state-of-the-art baseline, which significantly outperforms many other implementations of All-Reduce. To achieve better performance, NCCL is configured to achieve better AllReduce speed. The Tensor Fusion is also enabled for better network utilization. Tensor Fusion can reduce the overhead when performing AllReduce operations on gradients by avoiding frequent initialization. AD-PSGD is implemented in TensorFlow by randomly selecting communication neighbors. It uses grpc to communicate parameters between nodes. eager-SGD proposes solo and majority collective communication to implement partial AllReduce. Since in a heterogeneous environment, it happens that few processes are always faster than the others. The solo collective communication may negatively impact the convergence because of the staled update from slower processes. So we only implement the majority collective communication as the baseline.

We use the time it takes for the model to achieve the target loss as the metric of performance. We also measure the number of iterations and averaged per-iteration time to analyze the effect of our optimizations. We further use the validation dataset to validate the accuracy of the obtained models. As for Transformers, we conduct fixed-time experiments to compare the throughput of a different solution.

8 EXPERIMENTAL EVALUATION

8.1 Training speedup and convergence

We evaluate the training speed of ResNet50, VGG16, and LSTM with different system configurations. Because ResNet50 and VGG16 are balanced workloads after being preprocessed, we introduce system delay randomly, which ranges from 0 to 50ms, on each process. To evaluate the performance of the hierarchical synchronization mechanism, we simulate a cluster with mixed heterogeneity by dividing the machines into two groups, A and B. For group B, higher system delay is injected, which ranges from 50 to 100ms randomly. Based on the design principle of hierarchical synchronization, two ring communication graphs are formed, and one central PS is created to coordinate the parameter synchronization. We train these three models on corresponding datasets. The goal of training is to

Table 3: The final training accuracy for different neural networks. ResNet and VGG represent ResNet50 and VGG16, respectively.

approach	neural networks				
	ResNet	ResNet(H)	VGG	VGG(H)	LSTM
Horovod	78%	79%	93.4%	93.2%	88.2%
eager-SGD	76.2%	75.8%	92.8%	92.2%	87.5%
AD-PSGD	70.8%	68%	86.8%	87.6%	78.8%
RNA	78.2%	77.8%	92.6%	92.4%	87.8%

minimize the loss value. We use Keras EarlyStopping [56] to check whether the loss is no longer decreasing at the end of every epoch. The patience is set to ten, i.e., the training process is terminated if the loss cannot be decreased within ten iterations.

From Figure 6, it can be learned that RNA outperforms eager-SGD for ResNet50, VGG16, and LSTM. Compared with the state-of-the-art Horovod, RNA can achieve training speedup for ResNet50, VGG16, and LSTM by 1.7×, 1.4×, and 1.6×. The performance improvement demonstrates that the randomized per-process approach can mitigate the impact of dynamic heterogeneity probabilistically. Specifically, when training ResNet50 in a cluster with a higher degree of heterogeneity, the training speedup brought by eager-SGD and RNA with hierarchical synchronization are decreased, which are from 1.3× to 1.1× and from 1.7× to 1.5×, respectively. However, the RNA with hierarchical synchronization shows stable performance improvement, which is 1.8× and 1.4× for ResNet50 and VGG16, respectively. It demonstrates the probabilistic approach cannot handle the deterministic slowdown, i.e., group B's machines are slower than A by 50ms on average at each iteration. The hierarchical synchronization mechanism can avoid mixed heterogeneity. It can also be noticed that the performance of AD-PSGD is higher than RNA for VGG-16. It is because VGG16 has a larger neural network, which makes the communication a dominating factor. Specifically, RNA requires extra memory copy between CPU and GPU. However, from Table 3, we can see that AD-PSGD achieves the lowest accuracy compared with the other three approaches when the training is terminated. Horovod, eager-SGD, and RNA can achieve high accuracy.

Specifically, the convergence curve for LSTM using different approaches is shown in Figure 7. Though AD-PSGD reaches the stopping criteria for training earlier than Horovod, i.e., shorter execution time, it sacrifices the model accuracy. Compared with Horovod, RNA lowers the training time from 8,200 ms to 5200ms, leading to nearly 1.6× speedup. Eager-SGD can achieve similar accuracy with Horovod and RNA, but its throughput is lower than RNA, resulting in longer execution time. Overall, RNA can both speed up the training process and guarantee good model accuracy.

8.2 Validation on models

We further test the accuracy of the obtained ResNet50, VGG16 and LSTM models, which are summarized in Table 4. From the number of executed iterations, we can see that the state-of-the-art Horovod is bottlenecked by the throughput because of dynamic system heterogeneity or inherent imbalance. Although AD-PSGD requires fewer iterations to converge to minimize the loss value, the

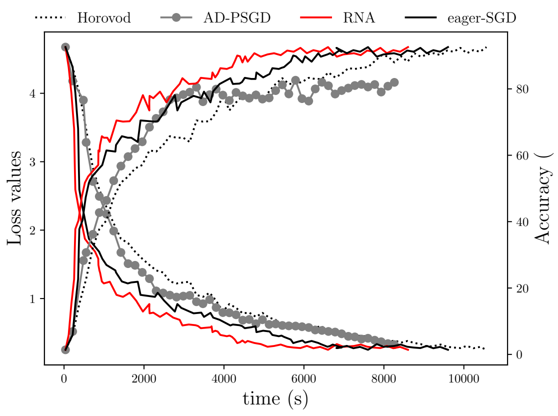


Figure 7: Convergence curve in terms of loss value and training accuracy for LSTM. Each point is collected at the end of one epoch.

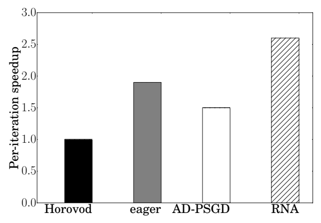
Table 4: The validation accuracy for different neural networks.

models	approaches	# of iterations	top-1 acc.	top-5 acc.
ResNet50	Horovod	42200	76.2%	93.2%
	eager-SGD	49800	74.8	91.2%
	AD-PSGD	38800	68.8	88.6%
	RNA	52400	75.9%	92.6%
VGG16	Horovod	1080	92.5%	-
	eager-SGD	1360	91.8%	-
	AD-PSGD	880	82.8%	-
	RNA	1420	92.2%	-
LSTM	Horovod	8120	68.2%	94.8%
	eager-SGD	9600	66.8%	94.6%
	AD-PSGD	7800	60.6%	90.1%
	RNA	9660	66.5%	95.2%

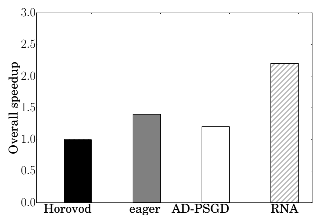
execution time of each iteration is severely affected by the synchronization overhead. And AD-PSGD achieves the lowest validation accuracy compared with other approaches. For LSTM and ResNet50, RNA takes advantage of asynchronous execution to allow more iteration in a fixed duration, leading to higher throughput. RNA can achieve higher training throughput than Eager-SGD because it can efficiently reduce the response time at each iteration. Both eager-SGD and RNA can obtain higher model accuracy than AD-PSGD. From these results, we can learn that RNA has significant convergence speed improvement compared with the state-of-the-art approach, AllReduce in Horovod. While compared with AD-PSGD, higher model accuracy is guaranteed.

8.3 Throughput comparison

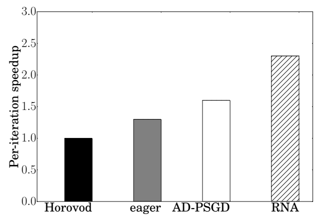
To evaluate the throughput, we train Transformers in both homogeneous and heterogeneous clusters. The homogeneous cluster has two nodes configured with eight NVIDIA GeForce GTX-1080Ti for each. In the homogeneous environment, the high variance of the input sentence length incurs imbalance training time among



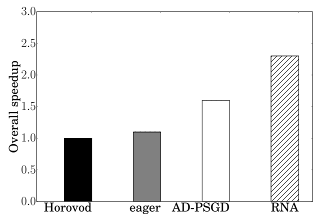
(a) Homogeneous.



(b) Homogeneous.



(c) Heterogenous.



(d) Heterogenous.

Figure 8: Per-Iteration Speedup and Overall Speedup comparison among Horovod, Eager-SGD, AD-PSGD and RNA in both homogeneous and heterogeneous environment.

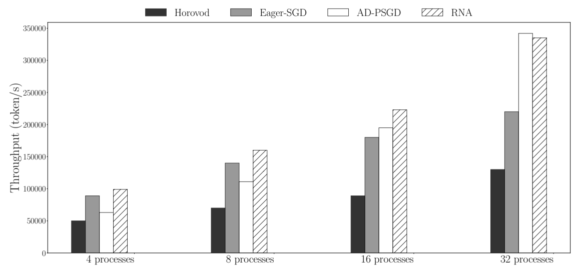


Figure 9: Throughput comparison between different approaches for Transformer on WMT17.

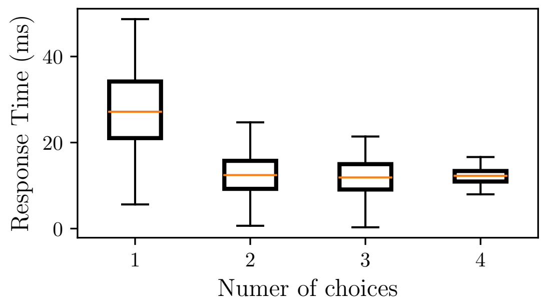


Figure 10: Effect of number of choices on response time. Whiskers depict 5-th and 95-th percentiles; boxes depict median, 25-th, and-75th percentiles.

processes. In the heterogeneous environment, we inject the additional dynamic slowdown to evaluate heterogeneity tolerance. In the experiment, we set the batch size to 4,096 tokens. The per-iteration speedup, and the overall speedup is shown in Figure 8. Horovod is selected as the baseline since it strictly follows the BSP model. The per-iteration time is the training time required for each iteration, which is averaged from one epoch. The overall speedup is the convergence time over the baseline. From Figure 8(a), we can see that RNA achieve the highest per-iteration speedup over Horovod, which is nearly 2.6× in a homogeneous environment, while eager-SGD and AD-PSGD can accomplish that by 1.9× and 1.4×, respectively. The reduction in the per iteration time results in less waiting time between iterations.More tokens have been processed by RNA within a fixed time duration compared with other approaches, leading to higher throughput. To obtain the same loss value of 2.0, RNA achieves 2.2× the overall speedup over Horovod on the execution time, as is illustrated in Figure 8(b), while eager-SGD and AD-PSGD achieve that by 1.4× and 1.2×, respectively. In a heterogeneous environment, as is shown in Figure 8(c) and 8(d), eager-SGD suffers from the random slowdown, whose per-iteration speedup drops from 1.9× to 1.3×. However, both AD-PSGD and RNA can achieve stable speedup, which is 1.6× and 2.3×, respectively, in terms of overall speedup. Combined with these two results, we can learn that RNA achieves a better balance between statistical efficiency and system efficiency. It requires more iterations while ignoring the staled contribution to gain significant speedup in per iteration time, leading to overall execution time speedup.

We further evaluate the scalability of RNA with other approaches on Transformers by varying the number of GPU processes. From Figure 9 shows that RNA and eager-SGD almost achieve the highest and similar throughput on a 4-processes scale. With the increased number of processes, both the AD-PSGD and RNA achieves higher throughput than Horovod and eager-SGD. When the number of processes is increased, RNA performs better scalability than Eager-SGD and Horovod. AD-PSGD also shows superior performance in terms of scalability. In particular, we notice when the number of processes is increased to 32, AD-PSGD is has a little higher throughput than RNA. Because Transformer networks mostly consist of tensor contractions implemented as batched matrix products. As a result, Transformer requires more straightforward computation compared with the computation-intensive Convolution and causes dominated communication overhead due to the significant number of parameters, leaving less space for optimization. However, compared with AD-PSGD, we notice that RNA can reach 24 on BLEU score while AD-PSGD can only obtain 22. This result shows that RNA can ensure higher accuracy compared with AD-PSGD. In terms of throughput, RAN can achieve better scalability compared with eager-SGD. These results show that the relaxed synchronization in RNA does not sacrifice high accuracy compared to state-of-the-art solutions while ensuing the training throughput.

8.4 Sensitivity analysis

The number of choices to approximate the behavior of the system could affect performance [42]. We design a microbenchmark to evaluate the performance of the per-process sampling approach. The simulated cluster has 100 nodes. We simulate the unbalanced workload by injecting tasks to each process with randomized skewness, which ranges from 10 to 50ms. At each iteration, we randomly

Table 5: The transmission cost in RNA.

DL application	ResNet50	LSTM	VGG16	Transformers
Extra cost	6.2%	3.8%	23%	18%

select a number of processes as the probes. When the fastest one among probes finishes execution, the computation proceeds to the next round. We run the synthetic workload for 100 iterations and obtain the response time for each iteration, which is shown in the Figure 10. The figure demonstrates that using one more oversampling probe could significantly improve performance compared to selecting initiator randomly, which reduces median response time by more than 2.4× compared to random sampling from 28ms to 12ms on average. Furthermore, the deviation of execution time for each iteration is smaller than using random selection, i.e., choice of one. The figure also demonstrates an interesting observation: a low probe ratio negatively impacts performance because it does not oversample enough to find a faster process. However, additional oversampling does not improve performance due to increased messaging. As illustrated in Section 5, the convergence rate trade-off more synchronization costs to gain overall execution time speedup. With this observation, to reduce the response time at each iteration efficiently, we use a probe ratio of 2 to implement our per-sampling approach.

8.5 System overhead

Compared with Horovod, to achieve asynchronous training, RNA firstly aggregates obtained gradients locally by writing the data from GPU to CPU memory. After the AllReduce operation, RNA needs to read the reduced results from CPU memory, which incurs extra transmission cost (i.e., overhead) because of the memory copy. Table 5 measures the transmission cost percentage in the execution time of three jobs using RNA. The transmission time for ResNet50, LSTM, VGG16 and Transformers accounts for the execution for one iteration for 6.2%, 3.8%, 23%, and 18%, respectively. We can see that the overhead for VGG16 and Transformers is more significant than that for the other jobs since these two have a larger number of parameters. Overall, the cost is much smaller compared to the performance improvement by RNA. But the transmission overhead is bottlenecked by the bandwidth of PCIe between CPU and GPU. And this communication overhead does not increase if we scale out the cluster because the transmission is executed locally. Overall, the additional transfer overhead is much smaller compared to the performance improvement brought by RNA. For neural networks with a larger model, we can optimize the performance by layer-wise overlapping between GPU and CPU.

9 RELATED WORK

A variety of solutions [63] have been proposed to overcome the straggler problem for distributed deep learning. Redundant execution [2][66] is commonly used to mitigate stragglers in the traditional data analytics platform. The main idea is to launch speculative execution on multiple machines. Recently, backup worker [6] is proposed in distributed learning systems to overcome the stragglers problem. However, the redundant execution introduces non-negligible overhead from data communication. Firstly, in ring All-Reduce, a more restrictive communication pattern makes it impossible to implement these techniques, e.g., backup works. Secondly, the redundant execution is not fruitful to handle the randomized system heterogeneity and inherent load imbalance.

Adaptive tuning strategy solves stragglers by matching the amounts of task loads to their respective capacities in a heterogeneous environment. FlexMap [8] launches elastic map tasks with dynamic input block sizes, and PIKACHU [20] is proposed to adjust the reduce task size elastically based on the system heterogeneity. However, all these works only focus on the traditional BSP scheme. Advanced approaches are proposed for deep learning systems. For example, R^2SP [5] is proposed to tune the batch size adaptively and FlexPara [60] partitions parameters to provision adaptive tasks to match the varying capacity. However, these works do not fundamentally solve the problem because the severe and continuous slowdown of some workers will eventually drag down other workers and the whole training. In the operating system, work-stealing is a classical method to achieve load balancing among workers, improving system-wide performance. The concept of work-stealing is to move workloads from slower workers to the faster ones, e.g., FlexRR [22]. Skewtune [31] is proposed to mitigate skewness in the data analytics platform, which waits for idle workers to steal work from those tasks with the greatest remaining processing time. Considering the large overhead from communication, these approaches cannot be directly applied to deep learning systems. Recently, relaxed synchronization is proposed to exhibit the strict need for synchronization on the BSP model. SSP [12][61] enables processes to execute the training independently and allows fast workers to advance a bounded number of iterations ahead of slow workers. A-BSP [59] is proposed to aggressively synchronize parameters by applying the partial updates from slower workers. But all these approaches target on the centralized PS architecture.

Taking advantage of the robustness of deep learning training process, AD-PSGD [37] is first to explore the fundamental algorithm level solution to allow asynchronous synchronization in a decentralized setting. In AD-PSGD, gradient updates are only sent to limited (random) neighborhoods using gossip algorithms. However, it requires extra overhead to perform atomic parameter averaging. Otherwise, it will suffer from deadlocks issues due to scheduling conflicts. Furthermore, the implementation is limited to a certain type of AllReduce graph. Similar approaches such as Cutout [15] and Dropout [4] propose random errors and omissions into the training process to improve generalization of network models. Hop [40] introduces a generic solution to overcome heterogeneity for decentralized training protocol, which proposes a queue-based synchronization mechanism to enable bounded staleness. However, maintaining a bunch of queues and tokens for each worker in a large cluster incurs communication overhead and delay. Furthermore, Hop accumulates gradients for faster workers. Due to the bounded iteration gap, some workers' severe and continuous slowdown will eventually drag down other workers and the entire training.

Prague [39] and Eager-SGD [35] are more related to our approach, which proposes a new communication primitive to allow partial workers to synchronize parameters quickly. Specifically, Prague offers both static and dynamic group scheduling to construct a new group randomly during the runtime to avoid conflicts. However, this approach is based on system profiling information, whose decision might not be optimal for dynamic neural networks such as RNN and LSTM. Moreover, it requires a careful group scheduling at each iteration to avoid the synchronization conflicts. It also introduces additional system overhead to form the communication graph, which harms the training throughput. Eager-SGD proposes solo and majority collective communication to implement an asynchronous decentralized SGD. Solo collective communication could sacrifice model accuracy because it advances the synchronization aggressively. As is illustrated in the evaluation results, the majority cannot ensure the performance because it does not oversample enough to avoid the slower processes. Furthermore, in a heterogeneous environment, eager-SGD still suffers from a deterministic slowdown, which cannot be avoided by the randomized approach.

SGP [3] adopts a gossip algorithm called PushSum for approximate distributed averaging, which allows for much more loosely coupled communications to achieve efficient distributed training in a high-latency or high-variability environment. SGP does not use global collective communication primitives. Alternatively, each process only communicates with its neighbors. However, all the processes need to finish the current iteration before going to the next. SGP is robust to communication-constrained settings. Compared to SGP, our work is robust to load imbalance. RNA can relax the strict synchronization to tolerate computation straggler because of the feature of asynchrony. Both eager-SGD and RNA only require $O(1)$ step to globally propagate the lo-cal update. However, in SGP, each process propagates its local update using $O(logP)$ steps. Zero/DeepSpeed [46] presents a set of optimizations to reduce memory redundancy in distributed training, by partitioning parameter weights, activations, and optimizer state separately, and it can scale models to 170 billion parameters. Compared with Zero, RNA is straggler tolerant and is orthogonal to their approach.

10 CONCLUSION

In this paper, we discuss and tackle the challenging straggler problem caused by imbalanced training load in the Ring All-Reduce protocol. The imbalance can be from the dynamic system heterogeneity itself or inherent workload. We propose a new synchronization mechanism, RNA, to implement a straggler-tolerant and BSP-compatible AllReduce to improve distributed deep learning performance. The key idea is that RNA allows partial processes to synchronize their gradients without waiting for slower ones. RNA can address performance issues in AllReduce using probabilistic approach, including the straggler problem caused by dynamic system heterogeneity and asymmetric workloads incurred by imbalance input data. We have performed comprehensive evaluations with various DL applications in different environments while providing convergence proof for the asynchronous gradient descent algorithm. Our experiment results on three representative deep learning applications, including image classification, machine translation, and video processing, show the proposed solution can achieve 1.8× speedup over the state-of-the-art implementation, i.e., Horovod, and 1.3× speedup over AD-PSGD.

11 ACKNOWLEDGMENTS

The authors would like to thank the shepherd Jim Dowling and the anonymous reviewers for their helpful suggestions and comments. This work was supported by the NSF grant CCF-1908843.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: A system for large-scale machine learning. In *Proc. of USENIX OSDI*.

[2] Ganesh Ananthanarayanan, Ali Ghodsi, Scott Shenker, and Ion Stoica. 2013. Effective straggler mitigation: Attack of the clones. In *Proc. of USENIX NSDI*.

[3] Mahmoud Assran, Nicolas Loizou, Nicolas Ballas, and Mike Rabbat. 2019. Stochastic gradient push for distributed deep learning. In *International Conference on Machine Learning*. PMLR, 344–353.

[4] Jimmy Ba and Brendan Frey. 2013. Adaptive dropout for training deep neural networks. In *Advances in neural information processing systems*. 3084–3092.

[5] Chen Chen, Wei Wang, and Bo Li. 2019. Round-robin synchronization: Mitigating communication bottlenecks in parameter servers. In *Proc. of IEEE INFOCOM*.

[6] Jianmin Chen, Xinghao Pan, Rajat Monga, Samy Bengio, and Rafal Jozefowicz. 2016. Revisiting distributed synchronous SGD. *arXiv preprint arXiv:1604.00981* (2016).

[7] 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. *arXiv preprint arXiv:1512.01274* (2015).

[8] Wei Chen, Jia Rao, and Xiaobo Zhou. 2017. Addressing performance heterogeneity in mapreduce clusters with elastic tasks. In *Proc. of IEEE IPDPS*.

[9] Adam Coates, Brody Huval, Tao Wang, David Wu, Bryan Catanzaro, and Ng Andrew. 2013. Deep learning with COTS HPC systems. In *Proc. of ICML*. 1337–1345.

[10] Ronan Collobert, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel Kuksa. 2011. Natural language processing (almost) from scratch. *Journal of machine learning research* (2011).

[11] Robert B Cooper. 1981. Queueing theory. In *Proceedings of the ACM'81 conference*. 119–122.

[12] Henggang Cui, James Cipar, Qirong Ho, Jin Kyu Kim, Seunghak Lee, Abhimanu Kumar, Jinliang Wei, Wei Dai, Gregory R Ganger, Phillip B Gibbons, et al. 2014. Exploiting bounded staleness to speed up big data analytics. In *Proc. of USENIX ATC*.

[13] Henggang Cui, Hao Zhang, Gregory R Ganger, Phillip B Gibbons, and Eric P Xing. 2016. Geeps: Scalable deep learning on distributed gpus with a gpu-specialized parameter server. In *Proc. of ACM Eurosys*.

[14] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, Quoc V Le, et al. 2012. Large scale distributed deep networks. In *Advances in neural information processing systems*.

[15] Terrance DeVries and Graham W Taylor. 2017. Improved regularization of convolutional neural networks with cutout. *arXiv preprint arXiv:1708.04552* (2017).

[16] Salvatore Di Girolamo, Pierre Jolivet, Keith D Underwood, and Torsten Hoefler. 2015. Exploiting offload enabled network interfaces. In *Proc. of HOTI*. IEEE, 26–33.

[17] Distributed (Deep) Machine Learning Community. 2014. PS-lite. https://github.com/dmlc/ps-lite.

[18] Jeffrey L Elman. 1990. Finding structure in time. *Cognitive Science* (1990), 179–211.

[19] EMNLP 2017. 2017. WMT17. http://www.statmt.org/wmt17/.

[20] Rohan Gandhi, Di Xie, and Y Charlie Hu. 2013. {PIKACHU}: How to Rebalance Load in Optimizing MapReduce On Heterogeneous Clusters. In *Proc. of USENIX ATC*.

[21] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. 2017. Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677* (2017).

[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 *Proc. of ACM SoCC*.

[23] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proc. of IEEE CVPR*.

[24] Geoffrey Hinton, Li Deng, Dong Yu, George Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Brian Kingsbury, et al. 2012. Deep neural networks for acoustic modeling in speech recognition. *IEEE Signal processing magazine* (2012).

[25] Qirong Ho, James Cipar, Henggang Cui, Seunghak Lee, Jin Kyu Kim, Phillip B Gibbons, Garth A Gibson, Greg Ganger, and Eric P Xing. 2013. More effective distributed ml via a stale synchronous parallel parameter server. In *Proc. of*

NeurIPS.

[26] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* (1997), 1735–1780.

[27] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. 2019. Gpipe: Efficient training of giant neural networks using pipeline parallelism. In *Advances in Neural Information Processing Systems*. 103–112.

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

[29] Zhihao Jia, Matei Zaharia, and Alex Aiken. 2018. Beyond data and model parallelism for deep neural networks. *arXiv preprint arXiv:1807.05358* (2018).

[30] Alex Krizhevsky, Geoffrey Hinton, et al. 2009. *Learning multiple layers of features from tiny images*. Technical Report. Citeseer.

[31] YongChul Kwon, Magdalena Balazinska, Bill Howe, and Jerome Rolia. 2012. Skewtune: mitigating skew in mapreduce applications. In *Proc. of ACM SIGMOD*.

[32] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep learning. *nature* (2015).

[33] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* (1998), 2278–2324.

[34] 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 *Proc. of USENIX OSDI*.

[35] 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 *Proc. of ACM PPoPP*. 45–61.

[36] 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 *Advances in Neural Information Processing Systems*.

[37] Xiangru Lian, Wei Zhang, Ce Zhang, and Ji Liu. 2017. Asynchronous decentralized parallel stochastic gradient descent. *arXiv preprint arXiv:1710.06952* (2017).

[38] Chris Lomont. 2011. Introduction to intel advanced vector extensions. *Intel white paper* 23 (2011).

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

[40] Qinyi Luo, Jinkun Lin, Youwei Zhuo, and Xuehai Qian. 2019. Hop: Heterogeneity-aware Decentralized Training. In *Proc. of ACM ASPLOS*.

[41] Tomáš Mikolov, Martin Karafiát, Lukáš Burget, Jan Černockỳ, and Sanjeev Khudanpur. 2010. Recurrent neural network based language model. In *Eleventh annual conference of the international speech communication association*.

[42] Michael Mitzenmacher. 2001. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems* (2001), 1094–1104.

[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 *Proc. of ACM SOSP*. 1–15.

[44] NVIDIA DEVELOPER COMMUNITY. 2017. NVIDIA Collective Communications Library (NCCL). https://developer.nvidia.com/nccl.

[45] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. PyTorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*. 8024–8035.

[46] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2019. Zero: Memory optimization towards training a trillion parameter models. *arXiv preprint arXiv:1910.02054* (2019).

[47] Herbert Robbins and Sutton Monro. 1951. A stochastic approximation method. *The annals of mathematical statistics* (1951), 400–407.

[48] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. 2015. Imagenet large scale visual recognition challenge. *International journal of computer vision* (2015), 211–252.

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

[50] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[51] Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D Manning, Andrew Y Ng, and Christopher Potts. 2013. Recursive deep models for semantic compositionality over a sentiment treebank. In *Proc. of EMNLP*. 1631–1642.

[52] Khurram Soomro, Amir Roshan Zamir, and Mubarak Shah. 2012. UCF101: A dataset of 101 human actions classes from videos in the wild. *arXiv preprint arXiv:1212.0402* (2012).

[53] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alexander A Alemi. 2017. Inception-v4, inception-resnet and the impact of residual connections on learning. In *Thirty-first AAAI conference on artificial intelligence*.

[54] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. Going deeper with convolutions. In *Proc. of IEEE CVPR*.

[55] Christian Szegedy, Alexander Toshev, and Dumitru Erhan. 2013. Deep neural networks for object detection. In *Advances in neural information processing systems*.

[56] TensorFlow Community. 2017. TensorFlow EarlyStopping. https://www.tensorflow.org/python/tf/keras/callbacks.

[57] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in neural information processing systems*. 5998–6008.

[58] Minjie Wang, Chien-chin Huang, and Jinyang Li. 2019. Supporting very large models using automatic dataflow graph partitioning. In *Proc. of EuroSys*. 1–17.

[59] Shaoqi Wang, Wei Chen, Aidi Pi, and Xiaobo Zhou. 2018. Aggressive synchronization with partial processing for iterative ml jobs on clusters. In *Proc. of ACM Middleware*. ACM, 253–265.

[60] Shaoqi Wang, Wei Chen, Xiaobo Zhou, Sang-Yoon Chang, and Mike Ji. 2019. Addressing Skewness in Iterative ML Jobs with Parameter Partition. In *Proc. of IEEE INFOCOM*.

[61] Jinliang Wei, Wei Dai, Aurick Qiao, Qirong Ho, Henggang Cui, Gregory R Ganger, Phillip B Gibbons, Garth A Gibson, and Eric P Xing. 2015. Managed communication and consistency for fast data-parallel iterative analytics. In *Proc. of ACM Socc*.

[62] Zhang Wei, Suyog Gupta, Xiangru Lian, and Liu Ji. 2016. Staleness-aware Async-SGD for Distributed Deep Learning. In *International Joint Conference on Artificial Intelligence*.

[63] Donglin Yang and Dazhao Cheng. 2020. Efficient GPU Memory Management for Nonlinear DNNs. In *Proc. HPDC*. 185–196.

[64] Yuan Yu, Martín Abadi, Paul Barham, Eugene Brevdo, Mike Burrows, Andy Davis, Jeff Dean, Sanjay Ghemawat, Tim Harley, Peter Hawkins, et al. 2018. Dynamic control flow in large-scale machine learning. In *Proc. of EuroSys*. 1–15.

[65] Joe Yue-Hei Ng, Matthew Hausknecht, Sudheendra Vijayanarasimhan, Oriol Vinyals, Rajat Monga, and George Toderici. 2015. Beyond short snippets: Deep networks for video classification. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 4694–4702.

[66] Matei Zaharia, Andy Konwinski, Anthony D Joseph, Randy H Katz, and Ion Stoica. 2008. Improving MapReduce performance in heterogeneous environments.. In *Proc. of OSDI*.