BAGUA: Scaling up Distributed Learning with System Relaxations

Shaoduo Gan*, Jiawei Jiang†, Binhang Yuan
ETH Zürich, Switzerland
{firstname.lastname}@inf.ethz.ch

Xiangru Lian*, Rui Wang, Jianbin Chang,
Chengjun Liu, Hongmei Shi, Shengzhuo Zhang,
Xianghong Li, Tengxu Sun, Sen Yang,
Ji Liu
kadishou technology, China
admin@mail.xrlian.com;ji.liu.uwisc@gmail.com

ABSTRACT
Recent years have witnessed a growing list of systems for distributed data-parallel training. Existing systems largely fit into two paradigms, i.e., parameter server and MPI-style collective operations. On the algorithmic side, researchers have proposed a wide range of techniques to lower the communication via "system relaxations": quantization, decentralization, and communication delay. However, most, if not all, existing systems only rely on standard synchronous and asynchronous stochastic gradient (SG) based optimization, therefore, cannot take advantage of all possible optimizations that the machine learning community has been developing recently. Given this emerging gap between the current landscapes of systems and theory, we build BAGUA, a MPI-style communication library, providing a collection of primitives, that is both flexible and modular to support state-of-the-art system relaxation techniques of distributed training. Powered by this design, BAGUA has a great ability to implement and extend various state-of-the-art distributed learning algorithms. In a production cluster with up to 16 machines (128 GPUs), BAGUA can outperform PyTorch-DDP, Horovod and BytePS in the end-to-end training time by a significant margin (up to 2×) across a diverse range of tasks. Moreover, we conduct a rigorous tradeoff exploration showing that different algorithms and system relaxations achieve the best performance over different network conditions.

PVLDB Reference Format:
doi:10.14778/3503585.3503590

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/BaguaSys/bagua.

1 INTRODUCTION
The increasing performance of distributed machine learning systems has been one of the main driving forces behind the rapid advancement of machine learning techniques. From AlexNet [35] in 2012 to GPT-3 [11] in 2020, each leap in model quality is enabled by the growth of both the model size and the amount of data one can train a model with, along with a rapid increase in computations [47]. Behind this improvement are two major enabling factors: hardware accelerations (e.g., GPUs and TPUs) and the development of efficient and scalable distributed training algorithms [4, 7, 72, 73, 76]. It is not unfair to say that a scalable distributed training system is the cornerstone of modern deep learning techniques.

In this paper, we scope ourselves and focus on data parallel training, one of the most popular distributed training paradigms in which the data set is partitioned across different workers and the model fits into a single device. Not surprisingly, recently years have witnessed a growing list of systems for distributed data parallel training. Existing systems fit into two paradigms, following the seminal work done by Li et al. [38] on parameter server and Sergeyev et al. [56] on using MPI collective operations such as Allreduce. Both paradigms have enabled industrial-scale distributed training systems [47]: Adam (Microsoft) [13], early TensorFlow (Google) [3], Poseidon (Petuum) [77], Angel (Tencent) [32], and BytePS (ByteDance) [33] are based on parameter server, while PyTorch-DDP (Facebook) [39], Mariana (Tencent) [82], MALT (NEC Labs) [37], NCCL (NVIDIA) [2], and Horovod (Uber) [56] are based on MPI-style collective operations. These systems often involve joint efforts from machine learning, systems, and data management communities, and have been successful in making distributed training easier and more scalable.

On the theory and algorithm side, researchers have also been active in improving the performance of standard synchronous and asynchronous stochastic gradient (SG) based algorithms. Rightly noticing that a major system bottleneck is communication, researchers have proposed a range of techniques to lower the communication overhead mainly by "relaxing" certain aspects of the communication. Examples include (1) communication compression (e.g., quantization [4, 7, 73, 76], sparsification [5, 68, 70, 72], and error compensation [67]), (2) communication decentralization [34, 40, 42, 43, 64, 66], and (3) communication delay (e.g., LocalSGD [21, 44, 61, 69]) and asynchronization [43, 52, 60, 80, 81]. These techniques are optimized for different workloads and different network conditions. These techniques together hold promises to significantly decrease the communication overheads, in terms of both bandwidth and latency, or increase the tolerance to the existence of stragglers.

In this paper, we are motivated by one emerging gap between the current landscapes of systems and theory: Despite the recent advance of distributed learning theory and algorithm on system relaxations, most, if not all, existing systems only rely on standard synchronous and
asynchronous stochastic gradient (SG) based algorithms. The main consequence is that existing systems are not taking advantage of all possible optimizations that the machine learning community has been developing, and potentially many real-world applications can be further accelerated. In this paper, we ask: Can we further accelerate distributed learning systems with system relaxations for communications? If so, what is the right abstraction for this purpose?

In this paper, we present BAGUA, a communication library whose goal is to support state-of-the-art system relaxation techniques of distributed training. We made two technical contributions.

Our first contribution is the system design of BAGUA, which provides a modular design for communications. BAGUA is a natural extension of the popular parameter server and Allreduce paradigms, inspired by the challenges of directly adapting these paradigms to support algorithms in Table 1 (See Section 3.3 for details). Specifically, we provide a collection of MPI-style collective operations to facilitate communication with different precision and centralization strategies. These primitives are flexible and modular enough to support many algorithms, illustrated in Table 1. Moreover, we also develop a simple automatic optimization framework that speeds up algorithms implemented within the BAGUA framework. The key behind this framework is automatic batching and scheduling of communications. Different from previous work such as Horovod [56] and BytePS [33], our optimization framework can be applied more widely beyond the standard SG based algorithm.

Our second contribution is an extensive empirical study centered around two hypotheses: (1) By supporting different system relaxation techniques, BAGUA is able to provide significant improvement for real-world applications and workloads with real-world infrastructure over existing systems; and (2) By supporting a diverse range of system relaxations, BAGUA is able to provide a scalable ML training over a diverse network conditions to allow a user picking different algorithms. To this end, we conduct a large-scale empirical study with both benchmark tasks and real-world applications running at Kwai Inc. On a cluster with up to 16 machines (128 GPUs in total, aggregated 2 petaFLOPS with Tensor Cores) we consider various network conditions following how V100 GPU machines ($\rho_1$.8xlarge, $\rho_3$.16xlarge, $\rho_3$.3xlarge, 24xlarge) are connected on AWS: 10Gbps, 25Gbps, and 100Gbps, with TCP/IP connections. BAGUA outperforms BytePS [33], Horovod [56], and PyTorch-DDP [39] by a significant margin (up to 2x for 10Gbps and up to 1.3x for 100Gbps) across a diverse range of tasks. Moreover, we conduct a rigorous tradeoff exploration showing that different algorithms and system relaxations achieve best performance over different network conditions. This illustrates the importance of providing this diverse cohort of algorithms to an end user.

There are several limitations of the current BAGUA system and we hope our efforts in building BAGUA can help and inspire future research in these exciting directions. First, BAGUA does not provide a principled way to help a user to automatically pick the most suitable system relaxations to apply. One exciting direction, after BAGUA provides the support for all these algorithms, is to understand how to build a principled auto-tuning system. Second, currently BAGUA only focuses on data parallelism and it is interesting future work to integrate other techniques such as model parallelism (e.g. [27, 29, 36, 49, 57, 71, 75]) and pipeline parallelism (e.g. [22, 24, 41, 48]) and to understand the system abstractions.

The rest of the paper is organized as follows. We start by a brief review of data parallel training and the optimization frameworks of existing systems in Section 2, acting as both the preliminaries and related work. We discuss the design and optimization of BAGUA in Section 3. We describe our experimental study in Section 4 and conclude in Section 5.

## 2 PRELIMINARY AND RELATED WORK

BAGUA is built on decades of research regarding distributed machine learning systems and algorithms. Plenty of them are from the database community [9, 10, 20, 28, 55]. We now summarize related work and discuss some in details to provide backgrounds and contexts. We refer the reader to [46] for the rigorous theoretical analysis of different system relaxation algorithms.

### 2.1 Data Parallel SG Based Algorithm

The cornerstone of distributed learning systems is the data-parallel stochastic gradient based (DP-SG) algorithms [38], which is the dominating algorithm that existing systems support and optimize for. Let $D$ be a dataset, $n$ is the number of workers, each worker $i$ holds its partition of the data $D^{(i)}$ and model replica at step $t$: $x_i^{(t)}$. Let $g_i^{(t)}$ be the stochastic gradient on worker $i$ at step $t$, a textbook DP-SG updates each local model replica, at worker $i$, as follows:

$$x_i^{(t+1)} = x_i^{(t)} - y \sum_{j=1}^{n} g_j^{(t)}$$

where $y$ is the learning rate. To make this happen, all machines need to exchange their local gradients $g_i^{(t)}$, aggregate, and broadcast to all machines. Naturally, this can be implemented by the standard Allreduce communication pattern.

When there are many workers or some potential stragglers, one can extend the above algorithm into its asynchronous counterpart. Instead of using the latest gradient at iteration $t$, we allow the access to some staled version:

$$x_i^{(t+1)} = x_i^{(t)} - y \sum_{j=1}^{n} g_j^{(i)}$$

where $i^{(j)}$ is the previous iteration at which the gradient of worker $j$ is computed, accessed by the worker $i$ at iteration $t$. In theory, linear speedup can be achieved by async-SGD [46].

### 2.2 Existing Distributed Learning Systems

Distributed learning systems have attracted intensive research over the last decade. Most existing systems, e.g., DistBelief [16], Angel [32], BytePS [33], and PyTorch-DDP [39], all focus on the optimization of the DP-SG algorithm or its asynchronous counterpart. There are two fundamental questions governing the design of these systems:

#### Table 1: System relaxation techniques. Async algorithms let workers communicate without waiting for other workers. Low precision algorithms compress the communication data. Decentralized algorithms remove the global data collection.

<table>
<thead>
<tr>
<th>Alg.</th>
<th>Sync.</th>
<th>Precision</th>
<th>Centralization</th>
</tr>
</thead>
<tbody>
<tr>
<td>[38]</td>
<td>Sync.</td>
<td>Full Prec.</td>
<td>Centralized</td>
</tr>
<tr>
<td>[34]</td>
<td>Sync.</td>
<td>Full Prec.</td>
<td>Centralized</td>
</tr>
<tr>
<td>[4, 62]</td>
<td>Sync.</td>
<td>Low Prec.</td>
<td>Centralized</td>
</tr>
<tr>
<td>[64, 46]</td>
<td>Sync.</td>
<td>Low Prec.</td>
<td>Centralized</td>
</tr>
<tr>
<td>[41]</td>
<td>Async.</td>
<td>Full Prec.</td>
<td>Centralized</td>
</tr>
<tr>
<td>-</td>
<td>Async.</td>
<td>Low Prec.</td>
<td>Decentralized</td>
</tr>
</tbody>
</table>
different ways to balance communication and aggregation by developing different systems, e.g., Horovod [29], BytePS, and PyTorch-DDP optimizes the execution for this communication pattern.

Figure 1: Illustration of Parameter Server and Allreduce.

(1) (Abstraction for Communications) How should one communicate and aggregate the gradient and model?

(2) (Optimizations) How should one optimize the end-to-end execution by balancing the communication and computation?

In terms of the abstraction for communications, existing systems fall into two paradigms: parameter server (PS) [14, 16, 25, 31, 38, 78] and Allreduce [12, 30, 56, 79]. Figure 1 illustrates these two paradigms. In a parameter server architecture, the model can be partitioned to shards and distributed to multiple nodes (we call these nodes “parameter servers”). During the training phase, workers periodically fetch the model from PS, calculate the gradients with local data and push the gradients to the PS, while the PS aggregates the gradients and updates the parameters. With an Allreduce paradigm, all the workers collaborate with their neighbors for model/gradient exchanges. A ring topology [53] is often adopted by existing systems for a two-phase communication: first, the paradigm partitions the model/gradient into n chunks (where n is the number of nodes), and use n rings with different starting and ending points to aggregate n chunks; second, the aggregation result of each chunk located in different nodes is broadcast through the ring.

After deciding on which communication paradigm to use, one key design is how to hide as much communication as possible during computation. This is often the core technical component of previous systems, e.g., Horovod [56], BytePS [33], and PyTorch-DDP [39]. These systems optimize the DP-SGD communication pattern by developing different ways to balance communication and computation. The key complexity roots from the fact that the training process of DP-SGD consists of delicate dependencies between different layers and their own (1) forward pass, (2) backward pass, (3) gradient synchronization, and (4) model update, phases. Figure 2 (Vanilla) illustrates a naive implementation of DP-SGD over a model with four layers. The system would communicate gradient (green) for each layer once its backward pass (blue) finishes, and update the model for all layers (pink) in one go once all their communications are finished. The system then starts the next forward pass (yellow).

PyTorch-DDP and Horovod are two Allreduce-based systems, and have specifically optimized this pipeline by overlapping the communication (Allreduce) with the backward pass and bucketing multiple gradients into one Allreduce operation. With overlapping, the Allreduce operations can take place in parallel with the computation of gradients. The Allreduce operation is only triggered when all gradients within a bucket are ready. The intuition of bucketing is that collective communications, like Allreduce, are more efficient on large tensors. After all Allreduce operations are finished, the model will be updated by the aggregated gradients.

BytePS, following the parameter server paradigm, has optimized this pipeline in a different way. BytePS partitions each gradient into small chunks with the identical size to conduct Push/Pull. BytePS overlaps Push/Pull with both backward and forward pass. It has a scheduler to maintain the communication order of gradient chunks. The principle is that parameters that are blocking the execution of the next forward pass will be prioritized for communication. Therefore, the forward pass of the next iteration could possibly be overlapped with the communication of the current iteration. In terms of asynchronous DP-SGD, BytePS supports it by allowing each worker updating the state of the server individually without waiting for other workers. Whereas PyTorch-DDP and Horovod do not support asynchronous communications since they rely on the Allreduce operator.

2.3 System Relaxations for Distributed DP-SG

While existing systems have been mainly focusing on synchronous and asynchronous DP-SG algorithm, the research community has developed a diverse set of techniques to further optimize for the different aspects of communications. These techniques often lead to different training algorithms, thus different communication patterns, as DP-SG. Given these differences, none of Horovod, BytePS, and PyTorch-DDP provides systematic support of these algorithms, as summarized in Table 1. The goal of BAGUA is to provide a flexible abstraction to support these diverse training algorithms with an automatic performance optimization framework without assuming a specific communication pattern such as the one of DP-SG. In order to reduce communication volumes, lossy communication compression methods are introduced, such as quantization [4, 7, 73, 76], sparsification [5, 68, 70, 72], sketching [26],
and error compensation [67]). In an attempt to get rid of the latency bottleneck, decentralized communication approaches are proposed [34, 40, 42, 43, 64, 66]. Additionally, localSGD is discussed to optimize for the number of communication rounds during training [21, 44, 61, 69]. To remove the synchronization barrier, some approach applies asynchronous update methods [50, 52, 60, 80, 81]. Lastly, it is worth to mention there are approaches that combines multiple strategies listed among [6, 8, 34, 43, 65].

To illustrate the difference of communication patterns between these advanced training algorithms and vanilla DP-SG, we take the example of QSGD [4] and Decentralized Low-precision SGD [64]. Figure 3 illustrates execution pipelines and communication patterns of DP-SG, QSGD and decentralized low-precision SGD. Compared with DP-SG, the execution components of the pipeline and their dependencies can be changed in the advanced algorithms. For example, the component “Quantization” required by both algorithms doesn’t even exist in the DP-SG, and the “model update” component in Decentralized low-precision SGD needs to happen before the communication. Since these advanced algorithms cannot fit into the DP-SG communication pattern, it is challenging for systems born for DP-SG to handle these algorithms.

3 SYSTEM DESIGN

The goal of BAGUA is to support advanced training algorithms beyond DP-SG. To achieve this, we revisit the two fundamental questions governing the design of previous systems, without assuming the pattern of DP-SG:

1. (Abstraction for Communications) How should one communicate and aggregate the gradient and model? In BAGUA, beyond parameter server and Allreduce, we design a collection of MPI-style collective operations to facilitate communications with different precision and centralization strategies.

2. (Optimizations) How should one optimize the end-to-end execution by balancing the computation and communication? In BAGUA, we develop a simple, but effective, automatic optimization framework which can be applied to optimize the execution of an algorithm implemented within BAGUA.

These two design decisions enable the flexibility and efficiency of BAGUA— to implement a new advanced algorithm with system relaxation (e.g., 1-bit Adam [63] or Decentralized SGD [42]), in BAGUA, a developer does not need to worry about manually balancing communications with computations; instead, she can specify, at a high-level, the logical semantics and BAGUA will automatically optimize its execution. In this section, we first provide a high-level system overview, followed by the descriptions of these primitives and their implementations, and then the simple, but effective, optimization framework in BAGUA.

3.1 System Overview

The goal of BAGUA is to facilitate the development of efficient and scalable distributed training algorithms that takes advantage of system relaxations. As illustrated in Figure 4, there are three players: an end-user, an optimization algorithm, and the BAGUA runtime.

From an end-user’s perspective, using BAGUA is very similar to use as PyTorch or TensorFlow for training on a single machine, with minimal changes to their existing code. The end-user should provide: (1) a neural network model that needs to train, specified as a graph in PyTorch, and (2) a stream of data examples. The end-user then specifies the training algorithm to use, e.g., QSGD [4] (training with communication compression), 1-bit Adam [63], or DecentralizedSGD [42], together with the information of the training infrastructure such as the number of machines and whether one should use MPI or NCCL for communication.

The core of BAGUA is a training algorithm, implemented by developers using the communication primitives and abstractions provided by BAGUA. An algorithm takes as input a neural network, provided by the end-user, and equips it with an algorithm-specific communication function. Specifically, the developer of an algorithm achieves this by registering this communication function as hooks at different stages of execution. One example is to register one hook after the backward computation of each layer. The communication function contains the core logic of a training algorithm, which has the following signature:

\[ f((x_1, g_1) \ldots (x_n, g_n)) \mapsto (x'_1, g'_1) \ldots (x'_n, g'_n) \]

where \((x_i, g_i)\) are the current model \((x_i)\) and gradient \((g_i)\) on the \(i\)th machine and \((x'_i, g'_i)\) are the updated model and gradient on the \(i\)th machine. To implement a communication function, the developer of an algorithm assumes an MPI-like execution model. The key difference is that the developer is equipped with not only the standard communication primitives in MPI (e.g., Allreduce), but also a set of communication primitives provided by BAGUA. These primitives support system relaxations such as compressed communications with error compensation, or decentralized communications.

When implementing the communication function in BAGUA, the developer provides a batched version of such a function, taking as input a set of layers. This allows BAGUA to later batch the communications automatically and optimize for its overlapping with the computations. When BAGUA invokes this function, it will rearrange parameters of all layers into consecutive memory space and also pass in a flattened version of these layers, treat them as a single variable. An algorithm developer can decide whether her algorithm can use this flattened version to avoid conducting communication for every layer by communicating once for all the layers.

During the runtime, each invocation to the communication function (which is triggered by the registered hooks) is registered with BAGUA, which equips BAGUA a global view of the workload to enable automatic scheduling and batching. The key technical contribution of BAGUA is to automatically apply a series of optimizations for communications. To make this happen, the core of BAGUA is the execution optimizer, which runs in two phases.

1. Profiling Phase. During the first forward/backward pass of the gradient descent computation, BAGUA keeps a log of all invocations of communication functions, executes them without any
optimizations. It then automatically: (1. **Bucketing**) groups layers into different buckets, whose communication will happen all at once; (2. **Flattening**) rearranges all the models and gradients of all layers in the same group into consecutive memory spaces to achieve better locality; (3. **Scheduling**) schedules when to conduct the communication of each bucket, overlapping with computations.

2. **Execution Phase.** For the rest forward/backward passes of the gradient decent computation, **Bagua** will conduct execution over an automatically optimized version of the model. By default, **Bagua** conducts one communication per bucket.

3.2 **Communication Primitives**

One key component of **Bagua** is a collection of communication primitives. All these operators follow an execution model similar to MPI, which take as input $n$ tensors $x_1...x_n$ (which can store parameter, gradient, etc.), each at a different worker, and outputs new data products $x'_1...x'_n$, each at a different worker:

$$op(x_1...x_n) \Rightarrow x'_1...x'_n$$

**Centralized, Full Precision.** **Bagua** provides a simple primitive, $\text{C}_{FP\_S}$, which provides the same functionality as the standard Allreduce operator. Specifically:

$$\text{C}_{FP\_S}(x_1...x_n) \Rightarrow x'_1...x'_n \quad \forall i \in [n], \quad x'_i = \sum_j x_j$$

We use this notation to express that, the effect of the $\text{C}_{FP\_S}$ operator is to calculate the sum of all local replicas, $\sum_j x_j$, and make it accessible to all workers.

**Centralized, Low Precision.** Communication compression has attracted intensive interests recently, given that many deep neural networks are tolerant to aggressive lossy compression of its gradient [4, 5, 7, 26, 67, 68, 70, 72, 73, 76]. **Bagua** provides the $\text{C}_{LP\_S}$ primitives for this purpose. Specifically:

$$\text{C}_{LP\_S}(x_1...x_n, \delta_1...\delta_n, \epsilon_1...\epsilon_n) \Rightarrow x'_1...x'_n, \delta'_1...\delta'_n, \epsilon'_1...\epsilon'_n$$

$$\forall i \in [n], \quad x'_i = Q\left(\sum_j Q(x_j - \delta_j) - \epsilon_i\right)$$

where $Q$ is the lossy compression function, specified by the developer and $\text{C}_{LP\_S}$ supports a general form of communication compression with error compensation [65, 67]. Note that setting $\delta_i$ and $\epsilon_i$ to None will disable error compensation and gives

$$\text{C}_{LP\_S}(x_1...x_n, \text{None}, \text{None}) \Rightarrow x'_1...x'_n \quad \forall i \in [n], \quad x'_i = Q\left(\sum_j Q(x_j)\right)$$

Intuitively, $\delta_i$ and $\epsilon_i$ keep the error caused by last iterations' compression. The convergence efficiency introduced by error compensated methods is quite robust to the compression. This technique is especially helpful when the compression function is relatively aggressive (e.g., top-K compression [5, 62]).

**Decentralized, Full Precision.** **Bagua** also supports decentralized communication, which gets rid of the latency bottleneck for model synchronization — instead of synchronizing among all $n$ workers in the cluster, each worker only sends the update to its neighbors. For example, according to a ring-based topology, the neighbors of a worker include its immediate left and immediate right workers in the ring. Formally, **Bagua**’s decentralized full precision communication primitive $\text{D}_{FP\_S}$ can be formalized as below:

$$\text{D}_{FP\_S}(x_1...x_n) \Rightarrow x'_1...x'_n \quad \forall i \in [n], \quad x'_i = \sum_{j \in N(i)} x_j$$

where $N(i)$ is the set of workers that are neighbors of worker $i$. Note that $N(i)$ is an input to $\text{D}_{FP\_S}$, which can be a deterministic function (e.g., fixed ring topology) or a randomized function.

**Decentralized, Low Precision.** **Bagua** also provides the primitive $\text{D}_{LP\_S}$ for decentralized low precision communication:

$$\text{D}_{LP\_S}(x_1...x_n) \Rightarrow x'_1...x'_n \quad \forall i \in [n], \quad x'_i = \sum_{j \in N(i)} Q(x_j)$$

Regarding the asynchronous algorithms, the current version of **Bagua** does not provide any asynchronous version of these primitives, instead, it supports asynchronous algorithms using these synchronous primitives as follows. An algorithm can implement two concurrent threads, one deals with computation and another deals with communications. These two threads do not wait for each other. This provides an implementation of many asynchronous algorithms [15, 43, 80], summarized in Table 1. It can also enable implementations for LocalSGD [44] and model averaging [74]. It is interesting to further explore the benefits of providing asynchronous version of primitives, which we leave as future work.

3.3 **Comparisons with PS and Allreduce**

We see **Bagua** as a **natural extension** of two existing paradigms, i.e., parameter server and Allreduce. The design of **Bagua**’s communication primitives is inspired by challenges that we faced when trying to implement all eight different communication patterns in Table 1 directly using these paradigms, discussed as follows.

**Parameter Server.** Considering the centralized, low precision communication pattern ($\text{C}_{LP\_S}$), we found it would be unnatural to implement such a pattern by directly using the put/get abstraction provided by a parameter server. The fundamental challenge is that the error compensation step requires us to keep a state (to remember the error) on the server side and continue to conduct accumulation and quantization for each get request. The $\text{C}_{LP\_S}$ primitive is an extension of the parameter server abstraction for this purpose. A more fundamental problem is to support decentralized algorithms ($\text{D}_{FP\_S}$) using the parameter server abstraction — we have to explicitly specify the communication topology. The $\text{D}_{FP\_S}$ primitive is an extension for this purpose.

**Allreduce.** Similarly, using AllReduce operators in MPI also faces challenges when implementing a centralized, low precision communication pattern. It is not clear how to keep track of the quantization error made by AllReduce at each communication step and compensate it in the next iteration. Similarly, it can also be unnatural to support the decentralized communication pattern using existing operators in MPI.

3.4 **Implementations of Primitives**

For the centralized primitives, **Bagua** adopts a ScatterReduce communication pattern [79]. Specially, the target tensor is divided into $n$ partitions, where $n$ is the number of workers. The $i$-th worker is responsible for aggregating the $i$-th partition. Since the underlying communication library NCCLU does not provide a ScatterReduce primitive, we implement this primitive using the basic **send** and
recv NCCL operators. Each worker 1) partitions local tensor, 2) sends partitions to corresponding workers, 3) receives responsible partitions from other workers, 4) merges received partitions, and 5) sends merged partition to other workers. ScatterReduce communication pattern can take advantage of the aggregated bandwidth of all workers (like Allreduce), and support compression techniques (unlike Allreduce). The low precision primitive C_LP_S leverages the ScatterReduce communication to incorporate two phases of compression. Note that, the compression and decompression procedures can be combined with error compensation technique to reduce information loss (see semantics in Section 3.2).

Unlike centralized training, workers in decentralized training only communicates with one or a few peers. BAGUA engineers two mechanisms to allocate peers — ring and random. The ring strategy gives successive ranks to workers and organizes all workers as a ring. Alternatively, the random strategy randomly chooses a peer for each worker. The low precision primitive D_LP_S uses the same peer selection and communication procedure as D_FP_S. The difference is that D_LP_S uses the compression function \( Q \) to compress the tensor.

### 3.5 BAGUA Optimization Framework

The central component of BAGUA is its execution optimizer. The goal of BAGUA’s execution optimizer is to automatically schedule and optimize the computations and communications during the training. We explore the following techniques in BAGUA.

Overlapping communication and computation is one central optimization to speedup distributed DP-SG. BAGUA automatically analyzes the computation graph that includes the in-place tensor operations and tensor communication primitives. Compared with existing systems, BAGUA considers more sophisticated scheduling. In vanilla DP-SG, the optimization can only hide the Allreduce communications inside the gradient computation, BAGUA is responsible for scheduling additional elements, such as compression/de-compression and the model update computations specified by the optimization algorithms (E.g., Figure 3).

In order to boost the efficiency of communication and parallel computation, fusing small tensors into buckets is an essential step — frequently calling the communication primitives to transfer small fragments of parameters is far from ideal in terms of fully utilizing the network bandwidth. Although the bucketing trick is adopted in both Horovod and PyTorch-DDP, their schema simply considers the Allreduce operation as the cost in the heuristic. By contrast, BAGUA needs to consider beyond the Allreduce. Once we split the computation graph into buckets, BAGUA conducts fusion over the buckets. BAGUA would carefully align parameters within a bucket into a continuous memory space. Then this flatten view of the parameters is leveraged for all the executions. For example, the low-precision compression/decompression lambda is directly applied over a flatten view of the bucket instead of individual tensors; the SG based optimizer for model update is also conducted at the level of buckets (Apex [1] also uses a similar optimization). This flatten view can utilize the parallelism of GPUs more effectively.

Last but not least, the communication of BAGUA can be conducted hierarchically. This is particularly useful when dealing with the heterogeneity in network connections, e.g., the bandwidth between GPUs within a server is much higher than the bandwidth between servers. Therefore, BAGUA communicates hierarchically in two levels: intra-node and inter-node, and optimizes the communication primitives based on this abstraction. For example, the centralized low-precision primitive C_LP_S can be optimized as first aggregating tensors within the node without compression, then performing inter-node aggregation over the leader workers with compression, and finally letting each leader worker broadcast aggregated data within the node. Notice that this optimization can potentially change the semantics of the communication primitives.

### 4 EVALUATION

We conduct extensive experimental study around three hypotheses:

- **BAGUA is able to provide significant performance improvements over state-of-the-art systems in terms of end-to-end training time and scalability, over realistic industrial-scale infrastructure.**
- **Different algorithms that BAGUA support provide benefits for different models and datasets under different network conditions. It is thus important for BAGUA to support all these algorithms.**
- **BAGUA’s automatic execution optimizer effectively optimizes the execution of various distributed training algorithms.**

#### 4.1 Experimental Setting

**Infrastructure.** All experiments are conducted on 16-GPU instances, each of which is equipped with 8 NVIDIA V100 32GB GPUs interconnected by NVLink. We consider three different network conditions following how V100 GPU machines (p3.8xlarge, p3.16xlarge, p3dn.24xlarge) are connected on AWS: 10Gbps, 25Gbps, and 100Gbps, with TCP/IP connections. The default bandwidth we are using is 100Gbps without specification.

**Competing Systems.** We compare the performance of BAGUA with three SOTA systems. PyTorch-DDP [39], Pytorch’s default solution of distributed data parallelism learning. Horovod [56], a distributed learning framework developed by Uber. BytePS [33], a distributed learning platform developed by ByteDance. Both PyTorch-DDP and Horovod relies on MPI Allreduce for communication while BytePS uses parameter servers. Horovod and Pytorch-DDP also supports fp16 gradient compression via the fp16 support in NVIDIA NCCL, which we also compare with.

**Datasets and Tasks.** We use five learning tasks, covering both standard benchmarks and production datasets at Kwai Inc: (1) Image: (ImageNet [17], VGG16 [59]); (2) Text: (SQuAD [54], BERT-LARGE fine-tune [18]); (3) Text: (Kwai Dataset, BERT-BASE fine-tune [18]); (4) Speech: (AISHELL-2 [19], Transformer); (5) Image+Text: (Kwai Dataset, LSTM [23]+AlexNet [35]). Table 2 summarizes the model size and FLOPs.

#### Table 2: Model Characteristics

<table>
<thead>
<tr>
<th>Model</th>
<th>VariG</th>
<th>BERT-LARGE</th>
<th>BERT-BASE</th>
<th>Transformer</th>
<th>LSTM+AlexNet</th>
</tr>
</thead>
<tbody>
<tr>
<td># Parameters</td>
<td>113.3M</td>
<td>302.2M</td>
<td>36.6M</td>
<td>66.5M</td>
<td>126.8M</td>
</tr>
<tr>
<td># FLOPs</td>
<td>31G</td>
<td>232G</td>
<td>22G</td>
<td>149G</td>
<td>97.12G</td>
</tr>
</tbody>
</table>

#### BAGUA Algorithms.

We implemented six algorithms in BAGUA. Allreduce, the standard DP-SG algorithm, implemented with C_FP_S primitive. QSGD [4], a quantized (8-bit) DP-SG algorithm, implemented with C_LP_S primitive without error compensation. 1-bit Adam [63], a quantized (1-bit) distributed learning algorithm, implemented with by C_LP_S primitive with error compensation. Decen-32bits, a decentralized training algorithm with the random
We will see, there is no one-size-fits-all algorithm in (in terms of latency and throughput), and worker heterogeneities. (Async) can make the communication totally independent of the 10Gbps bandwidth, following the setup of V100 instance B3.8xlarge.

4.2 End-to-end Comparison with SOTAs

We first conduct end-to-end experiments over the network with 10Gbps bandwidth, following the setup of V100 instance p3.8xlarge on AWS. In Figure 5, we compare Bagua with Horovod-16bits, which is the best of all the baseline systems in this setting. We report the results of Bagua Allreduce and the best algorithm in Bagua we select for each task by considering both the accuracy and efficiency. As we can see, Bagua can be up to 2x faster than the best competing system, while still guaranteeing the same model accuracy. Bagua Allreduce is the slowest one in this case because it has the largest amount of data transition. These results also reflect the effectiveness and necessity of supporting various algorithms.

There are multiple reasons behind the speedup of Bagua. The most direct one comes from the communication-efficient algorithms. For example, Bagua (QSGD-8bits and 1-bit Adam) can compress the data communication much more aggressively than Horovod-16bits, Bagua (Decentralized 32bits and 8bits) could have much less communication connections than Allreduce-based systems, and Bagua (Async) can make the communication totally independent of the computation. Therefore, the overall communication overhead is highly reduced by Bagua. Besides, Bagua has a well-optimized execution pipeline for the computation and communication, including tensor flattening and bucketing, memory management, overlapping and so on. All the algorithms implemented in Bagua are automatically benefiting from these system optimizations.

4.3 Trade-off of Bagua Algorithms

By supporting a diverse collection of algorithms, Bagua provides users flexibility to accommodate different tasks, network conditions (in terms of latency and throughput), and worker heterogeneities. As we will see, there is no one-size-fits-all algorithm in Bagua that is always optimal for all these situations. Instead, we see an interesting trade-off space of Bagua algorithms.

Convergence. The convergence behavior of different algorithms heavily depends on the tasks; thus, it is important to support a diverse cohort in Bagua. Figure 6 illustrates the convergence behavior of different algorithms. Taking VGG16 as an example and treating Bagua (Allreduce) as the baseline algorithm, QSGD and Async can have almost the same convergence rate, whereas Decen-32bits and Decen-8bits show some drop of accuracy. 1-bit Adam algorithm can’t converge on VGG16 and the training loss explodes after a few epochs because an important assumption of 1-bit Adam regarding the gradients variance cannot hold in this task. For BERT-LARGE, most algorithms can converge in a similar rate as Allreduce, except Async algorithm that suffers an obvious gap. For LSTM+AlexNet, Decen-32bits, Decen-8bits and Async can converge as Allreduce does, the performance of QSGD is degraded, and 1-bit Adam diverges again. The above results verify that different training algorithms show diverse convergence behaviours and there is NO algorithm that can beat others across all workloads. Unfortunately, given a specific task, Bagua currently cannot choose the best algorithm for users. Although a prior work [45] has theoretically given the convergence bounds of different training algorithms, it is still unsolved which algorithm of them achieves the best empirical convergence rate without actually running them. Later, we will provide some guidelines for uses based on our empirical study.

Network Conditions. Among the set of algorithms that have similar convergence behavior as Allreduce, their relative performance is governed by the underlying network conditions: latency and bandwidth. We vary these two factors and illustrate the epoch time in Figure 7 (We show BERT-LARGE, but other tasks have similar profile). Algorithms that conduct communication compression outperform others when the bandwidth is relatively low; whereas decentralized algorithms outperform others when the latency is relatively high. We see when the network gets slower, the gap between Bagua and other systems becomes even larger.

Worker Heterogeneity. We also simulate a heterogeneous cluster by manually degrading the Applications Clocks of one GPU. Specifically, we set the frequency of Graphics from 1290MHz to 585MHz. As shown in Table 3, when there are stragglers in the system, asynchronous algorithms outperform a synchronous one in terms of epoch time, which is also consistent with previous observations [51].

Insights and Guidelines. These results justify the fundamental motivation of Bagua: at the algorithmic level, there is no algorithm.

Table 3: Epoch time (s) with one straggler GPU

<table>
<thead>
<tr>
<th>Network</th>
<th>VGG16</th>
<th>Bert-large</th>
<th>Bert-base</th>
<th>Transformer</th>
<th>LSTM+AlexNet</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bagua (Async)</td>
<td>48</td>
<td>57</td>
<td>421</td>
<td>185</td>
<td>128</td>
</tr>
<tr>
<td>Bagua (1-bit Adam)</td>
<td>131</td>
<td>154</td>
<td>762</td>
<td>411</td>
<td>333</td>
</tr>
<tr>
<td>Bagua (QSGD)</td>
<td>132</td>
<td>154</td>
<td>908</td>
<td>420</td>
<td>316</td>
</tr>
<tr>
<td>Bagua (Decen-32bits)</td>
<td>138</td>
<td>164</td>
<td>885</td>
<td>452</td>
<td>324</td>
</tr>
<tr>
<td>Bagua (Decen-8bits)</td>
<td>136</td>
<td>165</td>
<td>800</td>
<td>455</td>
<td>320</td>
</tr>
</tbody>
</table>
We now validate the effectiveness of the Bagua optimization framework. As described in Section 3.5, the optimization framework consists of three main optimizations: \( O \): Overlapping between the training computation and Bagua execution; \( F \): Fusion and Flattening of tensors; \( H \): Hierarchical Communications.

We first apply Bagua to the standard DP-SG algorithm and compare with PyTorch-DDP, Horovod, and BytePS, as illustrated in Table 4. Different from these systems that manually optimize specifically for DP-SG, Bagua automatically optimizes for an algorithm that is implemented within its framework. We see that Bagua achieves similar, and sometimes better, performance, illustrating the effectiveness of Bagua’s optimization framework.

Second, we show that all three optimizations are crucial for the end-to-end performance of Bagua, and the benefits of them can vary significantly from task to task. We conduct an ablation study and Table 5 illustrates the result (\( X=0 \) means the optimization \( X \) is tuned off). We see different optimizations are important for different workloads. In principle, \( O \) can overlap as much operations as possible, \( F \) can make communications of small tensor more efficient, and \( H \) accelerates intra-node GPU communications. These intuitions can be observed from the empirical results. For communication intensive workloads (e.g., VGG-16), hierarchical communication improves the performance significantly. For models with many small tensors (e.g., BERT-LARGE) and decentralized communication patterns (e.g., LSTM+AlexNet), fusion and overlapping play a larger role. An exception is the asynchronous algorithm (Transformer) because its communication is completely independent with the computation, therefore, it doesn’t effect the epoch time.

5 CONCLUSION

We propose Bagua, a communication framework whose design goal is to support various distributed training algorithms with system relaxations, powered by a new system design and a simple but effective optimization framework. We conduct empirical study to illustrate the end-to-end performance of Bagua and to provide a systematic tradeoff study of different training algorithms.

ACKNOWLEDGMENTS

CZ and the DS3Lab gratefully acknowledge the support from the Swiss National Science Foundation (Project Number 200021_184628, and 197485), Innosuisse/SNF BRIDGE Discovery (Project Number 40B2-0_187132), European Union Horizon 2020 Research and Innovation Programme (DAPHNE, 957407), Botnar Research Centre for Child Health, Swiss Data Science Center, Alibaba, Cisco, eBay, Google Focused Research Awards, Kuaishou Inc., Oracle Labs, Zurich Insurance, and the Department of Computer Science at ETH Zurich.
REFERENCES


Young, et al 4447–4458.

Learning tralized training over decentralized data. In Hanlin Tang, Xiangru Lian, Ming Yan, Ce Zhang, and Ji Liu. 2018. D2: Decentralized SGD with Memory.

SGD with Memory.

International Conference on Learning Representations.

Long Li, Xiangru Lian, Ji Liu, Ce Zhang, and Yuxiong He. 2021. 1-bit Adam: SGD with Memory.


Zhengyuan Zhou, Panayotis Mertikopoulos, Nicholas Bambos, Peter Glynn, YinYu Ye, Li-Jia Li, and Li Fei-Fei. 2018. Distributed asynchronous optimization with unbounded delays: How slow can you go?. In International Conference on Machine Learning. PMLR, 5979–5979.