Predicting Throughput of Distributed Stochastic Gradient Descent

Zhuojin Li, Marco Paolieri, Leana Golubchik, Sung-Han Lin, and Wumo Yan

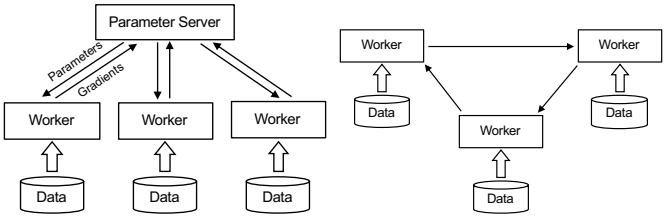
Abstract—Training jobs of deep neural networks (DNNs) can be accelerated through distributed variants of stochastic gradient descent (SGD), where multiple nodes process training examples and exchange updates. The total throughput of the nodes depends not only on their computing power, but also on their networking speeds and coordination mechanism (synchronous or asynchronous, centralized or decentralized), since communication bottlenecks and stragglers can result in sublinear scaling when additional nodes are provisioned. In this paper, we propose two classes of performance models to predict throughput of distributed SGD: *fine-grained models*, representing many elementary computation/communication operations and their dependencies; and *coarse-grained models*, where SGD steps at each node are represented as a sequence of high-level phases without parallelism between computation and communication. Using a PyTorch implementation, real-world DNN models and different cloud environments, our experimental evaluation illustrates that, while fine-grained models are more accurate and can be easily adapted to new variants of distributed SGD, coarse-grained models can provide similarly accurate predictions when augmented with ad hoc heuristics, and their parameters can be estimated with profiling information that is easier to collect.

Index Terms—Distributed Machine Learning, Stochastic Gradient Descent, Performance Prediction, Scalability, PyTorch.

1 Introduction

In recent years, deep learning [17] has achieved breakthrough results in many domains, including computer vision, speech recognition, and natural language processing; notably, to improve accuracy on increasingly difficult tasks, deep neural networks (DNNs) with more parameters and with computationally expensive training have been proposed [10], [32], [33], [34], [12]. At the same time, to reduce training times, deep learning has embraced hardware accelerators (including GPUs, FPGAs and ASICs [16]) to "scale up" and distributed training algorithms to "scale out". However, finding the best configuration (e.g., number of workers) for scale-out usually requires exhaustively profiling on the performance of every possible scenario, which is extremely time-consuming and not scalable for a large number of jobs.

One difficulty in estimating performance of distributed machine learning is that each training algorithm and architecture can exhibit distinct scalability. The prevalent approach for distributed training is *data-parallel stochastic gradient descent* (SGD), where multiple worker nodes train a local copy of the same DNN model using different shards of data; model updates are then shared between workers, using either a centralized or decentralized architecture, as illustrated in Fig. 1. To coordinate workers, two main strategies exist: in *Sync-SGD*, a worker can proceed to the next SGD step only after all other workers have completed the current step and exchanged model updates (i.e., workers start each step synchronously, using up-to-date copies of the model); in *Async-SGD*, workers can start local SGD steps independently of each other, using copies of the DNN model where only some of the updates have been applied. Async-SGD can achieve higher *job throughput* (total examples processed, per second, by all workers assigned to a training job) than Sync-SGD, because workers are never idle while waiting for stragglers [5]; nonetheless, accuracy of trained DNN models is more reliable for Sync-SGD, which is equivalent to single-worker training with larger batch size [3], [9], [11].



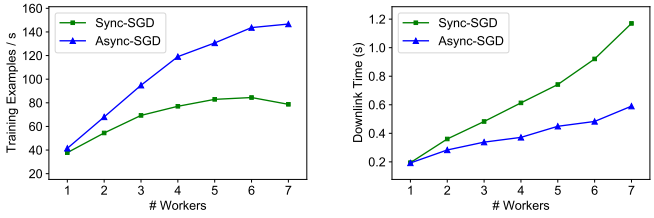
(a) Centralized (Parameter Server) (b) Decentralized (Ring AllReduce)

Figure 1: Centralized and Decentralized Architectures

Both Sync-SGD and Async-SGD can be implemented using either a centralized or decentralized architecture [2]. A popular centralized architecture is the *parameter server architecture* of Fig. 1a, where one or multiple parameter server nodes hold the global version of the DNN model: before each SGD step, workers pull the latest model parameters (downlink phase), independently run SGD on a batch of training examples (computation phase), and send gradients back to the parameter server (uplink phase); then, the parameter server updates the global model with the received gradients (update phase). In contrast, decentralized architectures commonly use the ring network topology illustrated in Fig. 1b: after an SGD step, workers perform an AllReduce operation to exchange and aggregate their updates.

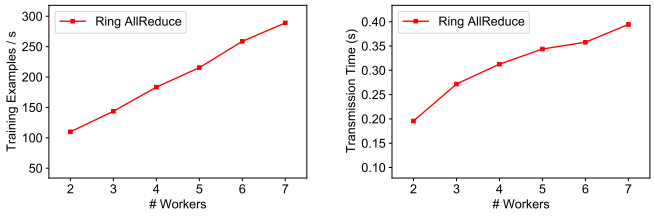
- *Zhuojin Li, Marco Paolieri, Leana Golubchik, and Wumo Yan are with the University of Southern California, Department of Computer Science, 941 Bloom Walk, Los Angeles, CA 90089, USA. E-mail:* {zhuojinl,paolieri,leana,wumoyan}@usc.edu
- *Sung-Han Lin is with Meta, 1 Hacker Way, Menlo Park, CA 94025, USA. E-mail:* sunghanl@fb.com*. This work was completed prior to the author joining Meta.*

Submitted to the IEEE Transactions on Parallel and Distributed Systems.



(a) Throughput (b) Downlink Time

Figure 2: Training ResNet-152 (batch size of 32 examples) on AWS `p3.2xlarge` with PS architecture



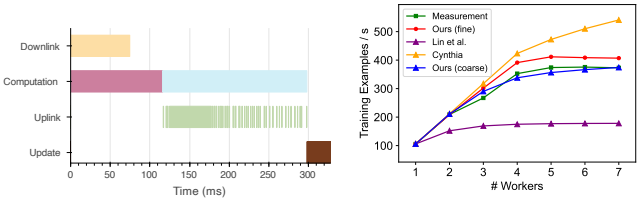
(a) Throughput (b) Transmission Time

Figure 3: Training ResNet-152 (batch size of 32 examples) on AWS `p3.2xlarge` with Ring AllReduce architecture

Fig. 2a shows the throughput measured for a training job of the ResNet-152 image classification DNN [10], using one parameter server and an increasing number of AWS `p3.2xlarge` worker instances, each equipped with an Nvidia V100 GPU and 10Gbps network. For both Sync-SGD and Async-SGD, throughput initially scales linearly but then starts saturating at 5 workers (Sync-SGD) and 6 workers (Async-SGD) due to a network bottleneck at the parameter server: Fig. 2b shows that the average time to transmit the up-to-date model (in each training step) increases with the number of workers; as expected, Async-SGD can achieve higher throughput. In contrast, with the decentralized architecture of Fig. 1b, workers using an AllReduce operation can balance network traffic across links: Fig. 3a shows that the throughput of decentralized Sync-SGD can scale linearly as the number of worker nodes grows, since the transmission time of Ring AllReduce operations (i.e., the time to exchange data among all workers) converges to a constant [24], as shown in Fig. 3b [1]; while introduced in [20], decentralized Async-SGD is not presented because most decentralized architectures use Sync-SGD [8], [4], [13].

To predict scaling characteristics of distributed SGD, most existing works [27], [26], [21], [37] propose *coarse-grained* analytical models, which partition each SGD step into a strict *sequence of communication and computation phases* (downlink, compute, uplink, update). In fact, machine learning frameworks such as TensorFlow and PyTorch define communication and computation *operations at a much lower level of granularity*, with dependencies that allow, once satisfied, to *overlap their execution*. For example, Fig. 4a shows the trace of an SGD step with overlaps between communication and

1. We observe higher-than-expected transmission time with 7 workers, due to the significant influence from background traffic on the cloud; this does not, however, substantially affect the job throughput.



(a) Timeline of a Training Step (b) Comparison of Approaches

Figure 4: Training ResNet-50 (batch size of 64 examples) on AWS `p3.2xlarge` with Async-SGD

computation: the worker starts the feedforward phase of the computation (red) as soon as it receives the first DNN layer during the downlink operation from the parameter server (yellow); similarly, as soon as the backward phase of the computation (cyan) is completed for a DNN layer, its uplink transmission to the parameter server (green) is initiated. To capture parallelism within an SGD step, our preliminary work [19] proposes a *fine-grained* model where tensor operations are executed as soon as their dependencies are satisfied, and the gradients of a tensor can be transmitted in parallel with the computation of other tensors.

Fig. 4b compares our throughput predictions with existing approaches for a training job of ResNet-50 (batch size 64) on AWS `p3.2xlarge`. As illustrated, existing coarse-grained models (yellow, purple) can noticeably mispredict job throughput. Our fine-grained model of Async-SGD introduced in [19] (red) analyzes dependencies between tensor operations (i.e., GPU events such as computation and data transmission) in the computation graph: profiling traces collected on a single worker are then used for a discrete-event simulation estimating throughput with multiple workers. While more accurate than coarse-grained models, this approach is computationally more expensive and requires fine-grained profiling data, which can be difficult to obtain with some ML frameworks (e.g., currently TensorFlow v2.4 [1] and PyTorch v1.7 [23] provide no support for profiling distributed training) and may incur measuring errors due to profiling overhead of GPUs and other hardware accelerators.

To summarize, performance prediction on distributed SGD is challenging, because a diversity of distributed training scenarios (e.g., DNNs, hardware platforms, optimized algorithm implementations) can exhibit distinct scalability. Existing coarse-grained models ignore the parallelism within a training step, resulting in poor predictions when communication and computation overlap; the fine-grained model in our preliminary work [19] is computationally expensive and only targets a specific implementation in TensorFlow. A detailed review of related work is given in Section 5.

In this paper, we conduct a comprehensive study on both coarse-grained and fine-grained performance models for distributed SGD, and systematically evaluate their extendability across a broad range of scenarios. Specifically, we make the following contributions.

- We propose heuristics to address the loss of information in our coarse-grained model of Async-SGD [21] due to ignoring parallelism between computation and communication operations. Our improved coarse-grained models (blue curve in Fig. 4b) produce throughput predictions with accuracy similar to fine-

grained models, but they require only limited profiling information and can be evaluated very quickly.
- We apply our fine-grained Async-SGD model of [19] to the machine learning framework PyTorch. Our evaluation shows that this fine-grained model can be successfully adapted to different underlying communication libraries (Gloo and NCCL), achieving predictions with average errors from 2.8% to 5.8%.
- We propose new models, coarse-grained and fine-grained, for Sync-SGD with centralized and decentralized architectures. These models account for stragglers due to the unequal split of network bandwidth among workers, and they can predict job throughput with average errors from 2.7% to 4.8%.
- We perform an extensive evaluation of coarse-grained and fine-grained prediction models using a PyTorch implementation of Sync-SGD and Async-SGD, for several real-world DNNs, and on multiple cloud environments (CloudLab [6] and AWS) with heterogeneous computing units (Xeon E5-2630 CPUs, Nvidia V100 GPUs, and Nvidia T4 GPUs). We highlight the results for GPU clusters in Table 1 to illustrate that both approaches can achieve low errors in throughput predictions, once they account for parallelism between computation and communication.

2 Asynchronous SGD

In this section, we present coarse-grained and fine-grained models for Async-SGD with parameter server architecture. In Async-SGD, each worker independently pulls the global model from the parameter server, performs a local SGD step, and pushes the resulting gradients of each DNN layer back to the parameter server, where they are applied to the global model. Workers compete for downlink and uplink networking with the parameter server, through an intermittent communication pattern: to predict network transmission rates, our models estimate the mean number of workers concurrently transmitting or receiving data from the parameter server. In addition, we account for heterogeneous processing rates at the workers, which can be provisioned independently and with different types of hardware accelerators.

Notably, through a heuristic correction, our coarse-grained model can also address distributed SGD implementations where fine-grained communication and computation operations overlap; this occurs in several real-world implementations, where a worker can start the feedforward computation of a DNN layer as soon as the downlink of that layer and the feedforward of the previous layer have completed; similarly, a worker can start the uplink of the gradients of a layer as soon as its backpropagation computation has completed, as illustrated in Fig. 4a.

2.1 Coarse-grained Model

In our coarse-grained model of Async-SGD with parameter server architecture, each SGD step completed by a worker includes a simple sequence of operations: downlink, compute, uplink, update. While compute operations use local resources, the rest of the operations use resources shared

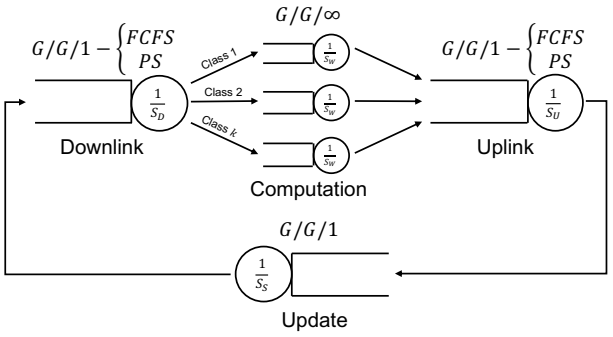


Figure 5: Queueing Model of Async-SGD

with other workers. To model resource sharing, we adopt the queueing model illustrated in Fig. 5, where each worker $k \in \{1, ..., K\}$ has exactly one task circulating through the queueing stations: after a *local computation* on a dedicated station (modeled as a $G/G/\infty$ queue, since it serves at most one task), the task is routed to the *uplink*, *update*, and *downlink* stations (modeled as $G/G/1$ queues), which are shared with tasks of other workers. While the scheduling model at the update station is always *processor sharing* (PS, i.e., n tasks are served in parallel with rate $1/n$), we solve the model using either a First Come First Serve (FCFS) or PS model at the downlink and uplink stations: when network utilization is low, FCFS is an appropriate model because transmissions of different workers tend to use the links exclusively until completion (TCP congestion control mechanisms favor the node that is already transmitting); when network utilization is high, multiple workers share the network long enough to reach similar bandwidth proportions, and this is accurately modeled by a PS model.

The notation for our queueing model is summarized in Table 2. Let $T_k^{l,\text{PS}}$ and $T_k^{l,\text{FCFS}}$ denote the mean response times at station $l \in \{k, U, S, D\}$ calculated with a PS or FCFS model, respectively, for the task of worker k. For this queueing model, the response times $T_k^{l,\text{PS}}$ and $T_k^{l,\text{FCFS}}$ can be obtained recursively from the mean service times S_k^l using exact or approximate *mean value analysis* [29], [28], respectively. The recursion is on the population vector $\vec{n} = (n_1, ..., n_K)$, where $n_k \in \{0, 1\}$ is the population of each class $k = 1, ..., K$:

$$T_k^{l,\text{PS}}(\vec{n}) = S_k^l\Big(1 + \sum_{j=1}^{K} N_j^l(\vec{n} - \vec{e}_k)\Big) \tag{1}$$

$$T_k^{l,\text{FCFS}}(\vec{n}) = S_k^l + \sum_{j=1}^{K} S_j^l\Big(N_j^l(\vec{n} - \vec{e}_k) - \frac{1}{2}\rho_j^l(\vec{n} - \vec{e}_k)\Big) \tag{2}$$

In these equations, ρ_j^l and N_j^l are the utilization and mean number of tasks of worker j at station l, and $\vec{e}_k$ is a vector with k-th component equal to 1 and other components equal to 0. The solution starts with $n_k = 1$ for all k and evaluates $T_k^{l,\text{PS}}(\vec{n})$ or $T_k^{l,\text{FCFS}}(\vec{n})$ for all $\vec{n} \in \{0, 1\}^K$; at each step of the recursion, $N_j^l(\vec{n} - \vec{e}_k)$ and $\rho_j^l(\vec{n} - \vec{e}_k)$ are evaluated through the identities (from Little's Law)

$$N_k^l(\vec{n}) = X_k^l(\vec{n})\, T_k^l(\vec{n})$$
$$\rho_k^l(\vec{n}) = X_k^l(\vec{n})\, S_k^l$$

	Async-SGD				Sync-SGD		
	Basic	Overlap	Heterogeneous	Heterogeneous with overlap	Basic	Overlap	Decentralized
Coarse	4.7% (11.1%)	4.7% (15.2%)	4.8% (16.0%)	3.8% (11.4%)	3.2% (10.0%)	4.1% (10.5%)	5.0% (11.6%)
Fine	5.4% (15.0%)	4.4% (11.4%)	3.5% (11.1%)	3.0% (9.6%)	5.2% (11.9%)	4.7% (10.2%)	2.5% (9.3%)

Table 1: Overview: Average (Maximum) Errors of Coarse-grained and Fine-grained Models on AWS GPU Clusters

K	Number of workers
$k \in \{1, ..., K\}$	Workers (task classes of queueing model)
$L(k) \coloneqq \{k, U, S, D\}$	Stations visited by class k: k-th worker node, uplink (U), parameter server (S), downlink (D)
S_k^l	Service time of class k at station l
$\vec{n} \coloneqq (n_1, ..., n_K)$	Number of tasks, for each class
$N_k^l(\vec{n})$	Mean number of tasks of class k at station l
$X_k^l(\vec{n})$	Mean throughput of class k at station l
$T_k^l(\vec{n})$	Mean response time of class k at station l
$\rho_k^l(\vec{n})$	Class k utilization at station l

Table 2: Notation

using the mean throughput $X_k^l(\vec{n}) = n_k / \left(\sum_{l \in L(k)} T_k^l(\vec{n})\right)$. In our queueing model, the transmission times T_K^U and T_K^D increase as the number of workers K in the system grows.

Finally, once we evaluate $T_k^{l,\text{PS}}$ and $T_k^{l,\text{FCFS}}$ in Eqs. (1) and (2), the throughput of a distributed SGD job using K workers is given by:

$$X^{\text{FCFS}}(K) = \sum_{k=1}^{K} \frac{1}{T_k^k + T_k^{U,\text{FCFS}} + T_k^{S,\text{PS}} + T_k^{D,\text{FCFS}}} \quad (3)$$

$$X^{\text{PS}}(K) = \sum_{k=1}^{K} \frac{1}{T_k^k + T_k^{U,\text{PS}} + T_k^{S,\text{PS}} + T_k^{D,\text{PS}}} \quad (4)$$

where $X^{\text{PS}}(K)$ and $X^{\text{FCFS}}(K)$ are the job throughput calculated with a PS or FCFS model, respectively. Our PS and FCFS models (for high and low network utilization, respectively), are combined as:

$$X_{job}(K) = \begin{cases} X^{\text{FCFS}}(K) & \text{if } \rho^{\text{FCFS}} \leq \rho_T, \\ X^{\text{PS}}(K) & \text{otherwise.} \end{cases} \quad (5)$$

When network utilization ρ^{FCFS} is less than the threshold ρ_T, the FCFS solution $X^{\text{FCFS}}(K)$ is used; when ρ^{FCFS} is greater than ρ_T, the PS solution $X^{\text{PS}}(K)$ is used.

2.1.1 Correction for Overlapping Operations

In many real-world implementations of Async-SGD, there are fine-grained computation and communication operations, e.g., one operation for each DNN layer or for each DNN tensor within a layer. These fine-grained operations have dependencies on each other, but they can often be executed in parallel, as illustrated by Fig. 4a: for example, downlink and computation stages can overlap because the feedforward computation of a DNN layer can start as soon as its downlink and previous feedforward operations have completed, while other downlink operations are still in progress; similarly, the gradient uplink of a DNN layer can start as soon as its backpropagation has completed, while the backpropagation of other layers is still in progress. These overlaps are more noticeable when many workers share network resources to communicate with the parameter server: in this case, workers transmit at reduced rates and downlink/uplink operations take longer, so that an increased fraction of the computation overlaps with communication operations. When these overlaps are ignored, job throughput can be severely mispredicted.

To account for overlaps between downlink and feedforward computation, and between backpropagation and uplink, we propose a heuristic correction to our model: we profile feedforward and backpropagation times S_k^F and S_k^B, respectively; we solve our hybrid model in Eq. (5) to estimate the response times of downlink and uplink queues (T_k^D and T_k^U, respectively); then, we replace the computation time $S_k^W = S_k^F + S_k^B$ in our model with

$$S_k^W = \max\left(0, S_k^F - T_k^D\right) + \max\left(0, S_k^B - T_k^U\right) \quad (6)$$

and we solve our model again to evaluate job throughput. This corrected model assumes that the entire downlink and uplink operations (with durations T_k^D and T_k^U) overlap with feedforward and backpropagation, respectively.

2.1.2 Profiling

In our preliminary work [21], the durations of downlink and uplink operations were measured directly, by monitoring network traffic in a profiling job with a single parameter server and a single worker node; in turn, the duration of computation and update operations were estimated as the time between transmissions. However, these estimated can be inaccurate due to overlaps in communication and computation, and to intermittent gradient transmissions, as illustrated in Fig. 4a.

To address this problem, we estimate the mean service time of downlink and uplink stations (without contention with other workers) as

$$S_k^D = S_k^U = \frac{M}{B}$$

where M is the size of the DNN model and B is the total network bandwidth.

For the computation time S_k^W, we separately profile the feedforward S_k^F and backpropagation S_k^B times for each type of worker k, to apply the correction of Eq. (6). Note that our model allows workers with heterogeneous computing power, since we individually profile computation times for each type of worker node; instead, the downlink/uplink times S_k^D and S_k^U, and the mean service time S_k^S at the parameter server, are the same for all workers (i.e., they are independent of k).

2.2 Fine-grained Model

Instead of modeling centralized Async-SGD as a sequence of four phases, our fine-grained model represents an SGD step as a directed acyclic graph where nodes denote the execution of communication or computation *operations*, and directed edges are *dependencies* between operations. The granularity of individual operations depends on the type of profiling information: for example, using low-level GPU profiling, computation operations correspond to the execution of GPU kernels (e.g., a GEMM function); if profiling collects execution times for each DNN layer, a computation operation represents the time for forward or backward propagation of a layer. Similarly, communication operations can represent individual tensors (weights during downlink and gradients during uplink), or entire DNN layers.

This fine-grained model provides an abstraction for simulating SGD steps with multiple workers, by accounting for reduced transmission rates due to the sharing of network resources. We adopt a PS model to share network resources among multiple workers: when n workers are sending or receiving data from the parameter server, transmission times recorded during single-worker profiling are multiplied by n.

Our earlier work [19] applied this approach to the distributed SGD implementation of TensorFlow 1.13, collecting low-level GPU profiling information. In this paper, we adapt the approach to predict throughput for the PyTorch framework; since the current PyTorch (v1.7) does not support the collection of profiling traces for distributed SGD, we built our own profiler to record, for each DNN layer, the execution times of communication and computation operations. Notably, our work shows that a fine-grained approach can be applied to a broader class of machine learning frameworks, including those without support for distributed profiling; in addition, we confirm that profiling information collected for each layer has sufficient granularity to characterize overlaps between communication and computation, and that low-level GPU profiling is not required.

2.2.1 Profiling

Using traces collected on a single worker, we profile two types of operations: (1) computation operations, namely forward or backward propagation of DNN layers at a worker, or the aggregation of gradients at the parameter server; (2) communication operations, which include transmission of parameters (during downlink) or gradients (during uplink) between the worker and the parameter server. For each profiled operation op, we record:

- $op.res \in \{\text{DOWNLINK}, \text{WORKER}, \text{UPLINK}, \text{PS}\}$, the resource used by op: DOWNLINK represents the network link used to transmit the DNN parameters from the parameter server to workers; WORKER represents the compute resources at workers, used for the forward or backward propagation; UPLINK represents the network link used to transmit the gradients from the workers to the parameter server; PS represents resources at the parameter server, used to update the global model with the received gradients.
- For each communication operation op (i.e., when $op.res \in \{\text{DOWNLINK}, \text{UPLINK}\}$), we record the size of the transmitted data as $op.size$.

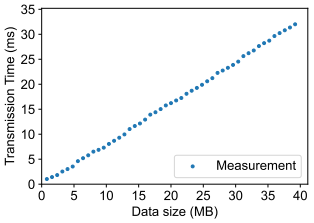


Figure 6: Point-to-point Transmission Time for Different Data Sizes on AWS 10Gbps Network

- For each computation operation op (i.e., when $op.res \in \{\text{WORKER}, \text{PS}\}$), we record the measured duration as $op.dur$.
- For each operation op, we record the operations that it depends on, as $op.waiting\_for$, and the operations that depend on op, as $op.dependent\_ops$. For example, the feedforward computation operation of a DNN layer depends on: (1) the communication operation transmitting that layer, and (2) the feedforward computation operation of the previous layer.

The current PyTorch v1.7 supports point-to-point communication through the Gloo backend [7]. We benchmarked the performance of this backend on AWS EC2 instances with 10Gbps networking; the results in Fig. 6 show that transmission time is a linear function of data size, for a wide range of input sizes. Therefore, in the following we use a simple linear model (data size divided by network bandwidth) to estimate transmission times.

2.2.2 Simulation Algorithm

We present a simulation algorithm (Algorithm 2.1) to generate N-step synthetic traces with W workers from the S-step profiling trace collected using a single worker. For each operation, $op.remaining$ represents the remaining amount of work to complete the operation: for communication operations, $op.remaining$ is initialized as $op.size/B$; for computation operations, it is initialized as $op.dur$. For each worker $w \in W$ and resource $r \in \{\text{DOWNLINK}, \text{WORKER}, \text{UPLINK}, \text{PS}\}$, we use a queue $scheduler[w, r]$ to store all operations ready to run (i.e., without pending dependencies). Q refers to the queue of operations in progress: for each worker w and resource r, only one operation can be in Q at a time. During each iteration, we remove from Q the operation op with minimum remaining time (Line 11), subtract its remaining time from other operations in Q (Line 15), and update the dependencies of operations waiting for op (Line 18). Finally, we add another operation to Q (Line 28), if the same worker has other operations requesting the same resource and without unsatisfied dependencies.

This algorithm supports heterogeneous compute nodes, provided that we collect profiling traces on each type of compute nodes to estimate $op.dur$. Overlaps of communication and computation are simulated by executing operations in parallel once their dependencies are satisfied. For example, forward propagation of a layer can start when the worker has received its parameters (from a downlink operation) and forward propagation of the previous layer has completed; similarly, an uplink communication can start as soon as backward propagation of a layer has completed.

Algorithm 2.1: Simulation

```
StartRandomStep(S, Q, w, active)
step = SampleWithReplacement(S)
for op in step.copy()    // each step starts with downlinks
  if op.res == DOWNLINK
    scheduler[w, DOWNLINK].add(op)
// scheduler stores worker operations, which are ready to run
Q.add( scheduler[w, DOWNLINK].remove_op() )
active[DOWNLINK].append(w)

GenerateTrace(S, W)
Q = ∅    // set of operations in progress
trace = Trace()    // empty simulated trace
active = {DOWNLINK : [], UPLINK : []}
for w in W    // setup for each worker
  completed_steps[w] = 0
  for r in {DOWNLINK, WORKER, UPLINK, PS}
    scheduler[w, r] = Scheduler(r)    // empty scheduler
  StartRandomStep(S, Q, w,active)  // first downlink
while Q ≠ ∅
  sort Q by x.remaining / Share(x.worker, x.res, active)
  next = Q.remove_min()
  w, r = next.worker, next.res
  eta = next.remaining / Share(w, r, active)
  trace.add(w, r, next, eta)
  for operation x in Q    // update remaining work
    x.remaining -= eta × Share(x.worker, x.res, active)
  // dependent ops can be assigned to scheduler if ready
  for d in next.dependent_ops
    d.waiting_for.remove(next)
    if d.waiting_for == ∅    // no other dependency
      if scheduler[w, d.res] != ∅  // w already using d.res
        scheduler[w, d.res].add(d)  // just queue d
      else    // start running operation d
        scheduler[w, d.res].add(d)
        active[d.res].append(w) // w becomes active
        Q.add( scheduler[w, d.res].remove_op() )
  if scheduler[w, r] != ∅    // w has more ops to run on r
    Q.add( scheduler[w, r].remove_op() )
  else    // no more operations of w to run on r
    active[r].remove(w)    // become inactive
    if scheduler[w, i] == ∅ ∀i    // no more pending ops
      completed_steps[w] += 1    // step is over
      if completed_steps[w] < N
        if MODE == ASYNC
          StartRandomStep(S, Q, w,active)
        else if MODE == SYNC and Q == ∅
          // start next step after all workers complete
          for w' in W
            StartRandomStep(S, Q, w', active)
return trace

Share(w, r, active)        // fraction of r assigned to worker w
if r in {DOWNLINK, UPLINK}
  if LINK_MODEL == FCFS
    // the first worker obtains full bandwidth
    if active[r].front() == w return 1 else return 0
  else if LINK_MODEL == PS
    return 1/active[r].length    // equally sharing
  else if LINK_MODEL == ALL_REDUCE
    return 1    // full bandwidth
else return 1    // processing is independent for each worker
```

3 Synchronous SGD

In this section, we present coarse-grained and fine-grained models for Sync-SGD, where all workers start each SGD step at the same time, using an up-to-date copy of the DNN model. When a parameter server architecture is used, the parameter server transmits the current DNN model to the workers (in parallel) and waits to receive gradients; only after gradients from all workers have been received and applied to the DNN model, the parameter server starts the next SGD step. When a decentralized architecture is used, workers compute gradients and then exchange them with each other, usually through an AllReduce operation on a ring topology (to reduce network traffic). After each worker has received and applied updates from all the other workers, it proceeds to the next SGD step.

A problem common to both of the above architectures is that of *straggler workers*: an SGD step can take longer for a worker due to the variability in the performance of computation and communication resources. For example, heterogeneous workers process training examples at different rates; or, in a parameter server architecture, uplink and downlink operations can receive varying and unequal shares of network bandwidth to the parameter server. Fig. 7 illustrates this phenomenon with a trace where the downlink operation starts at the same time for all the workers, but terminates earlier for worker 1; similarly, uplink phases of workers 0 and 2 start at a similar time, but the uplink of worker 2 is faster, even though the same amount of data is transmitted by each worker. As a consequence, workers 1 and 2 must wait for worker 0 before starting the next SGD step. It is also worth noting that, similarly to Fig. 4a, downlink and uplink operations of Sync-SGD can also overlap with the forward and backward propagation, respectively. All of these aspects are addressed by our models of Sync-SGD.

3.1 Coarse-grained Model

We now describe the details of our coarse-grained analytical model for Sync-SGD, under both centralized and decentralized architectures. The notation for our coarse-grained model is summarized in Table 3.

3.1.1 Parameter Server Architecture

K	Number of workers
M	Model size
B	Network Bandwidth
T_D	Duration of Downlink phase
T_U	Duration of Uplink phase
T_F	Duration of Feed-forward
T_B	Duration of Back-propagation
T_C	Duration of Computation phase, including T_F and T_B
T_S	Duration of Update phase on parameter server

Table 3: Notation

In Sync-SGD, workers start each SGD step at the same time, with the same sequence of downlink, compute, uplink, and model update phases. With a parameter server architecture, as the number of workers increases, the duration of the compute phase is the same, but downlink and uplink take

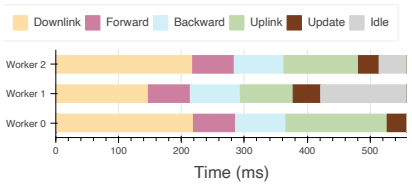
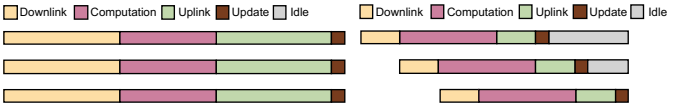


Figure 7: Trace of Sync-SGD with 3 workers



(a) PS (b) FCFS

Figure 8: Bandwidth Sharing Models in Sync-SGD

longer because more workers share the network resources to/from the parameter server. Assuming a uniform split of network resources (PS model), a simple model for the duration T_{PS} of a Sync-SGD step with one parameter server and K workers is then

$$\begin{aligned} T_{PS} &= T_D + T_C + T_U + T_S \\ &= \left(K \cdot \frac{M}{B}\right) + (T_F + T_B) + \left(K \cdot \frac{M}{B}\right) + T_S \end{aligned} \quad (7)$$

where M is the model size, B is the total network bandwith to/from the parameter server, and T_F, T_B, T_S are the computation times for the forward, backward, and model update phases. With this model, illustrated in Fig. 8a, the execution of each phase proceeds synchronously at the different workers.

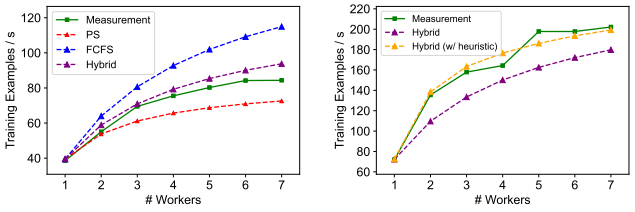
In fact, in cloud environments with background traffic, network bandwidth can be split unevenly among workers, and this model can severely underestimate job throughput; this is illustrated in Fig. 9a, where we compare the predictions of this model (red line) with actual measurements (green line) in a cluster of AWS `p3.2xlarge` instances training ResNet-152. Fig. 8b shows an extreme case of uneven sharing, where different workers have continuous access to the entire network bandwidth for the duration of their transmissions (FCFS model): the downlink phase "offsets" the execution of SGD phases at the different workers, which can then transmit their gradients at different times, without contention for network resources. In this case, a model for the duration T_{FCFS} of a Sync-SGD step with one parameter server and K workers is

$$T_{FCFS} = \left(K \cdot \frac{M}{B}\right) + (T_F + T_B) + \left(\frac{M}{B}\right) + T_S\,. \quad (8)$$

This extreme case is also unlikely to happen in practice, and leads to overestimating job throughput, as illustrated in Fig. 9a (blue line). Instead, we find that a linear combination of these models provides reliable estimates of job throughput; in particular, we adopt a simple average:

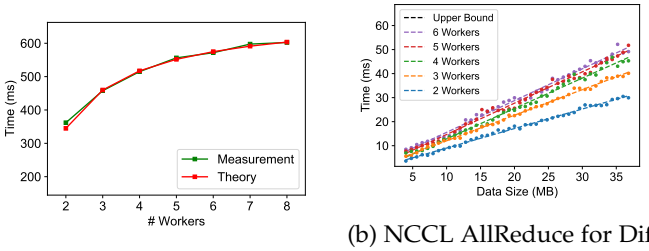
$$\begin{aligned} T_{\text{Hybrid}} &= \frac{T_{PS} + T_{FCFS}}{2} \\ &= \left(\frac{KM}{B}\right) + (T_F + T_B) + \left(\frac{(K+1)M}{2B}\right) + T_S \end{aligned} \quad (9)$$

Fig. 9a illustrates that this model (purple line) is very close to the measured throughput (green line).



(a) Non-overlap (b) Overlap

Figure 9: Training ResNet-152 (batch size of 32 examples) on AWS `p3.2xlarge` with centralized Sync-SGD



(a) Gloo AllReduce for 40MB Data on 1Gbps Network

(b) NCCL AllReduce for Different Data Sizes on 10Gbps Network

Figure 10: Performance of AllReduce with Different Backends

Finally, to account for the effects of overlapping communication and computation, we follow a similar approach to Section 2.1.1, where a maximum overlap of communication and computation is assumed; for Sync-SGD, from Eq. (9) we obtain our final model for the duration of each SGD step:

$$T_{\text{Hybrid}}^{\text{Overlap}} = \max\left(\frac{KM}{B}, T_F\right) + \max\left(\frac{(K+1)M}{2B}, T_B\right) + T_S \quad (10)$$

Fig. 9b shows that this modified model (yellow line) achieves more accurate predictions than the basic model of Eq. (9) (purple line) for Sync-SGD with overlapping communication and computation.

3.1.2 *Decentralized Architecture*

For Ring AllReduce update operations over homogeneous networks, communication time is estimated in [24] as $2(K-1)T/K$, where K is the number of participants and T represents the time required to transmit data between two workers. The intuition behind the formula is that each Ring Allreduce operation of data size M on K processors consists of (1) a ReduceScatter operation, which requires each processor to send data of size M/K for $(K-1)$ times, and (2) an AllGather operation, which requires another transmissions of M/K data for $(K-1)$ times on each processor. As the number of participants K increases, communication time approaches the upper bound $2T$. We validate the performance of the Ring AllReduce implementation using two different backends available in PyTorch[2]: Gloo [7] in our CPU cluster (Fig. 10a), and NCCL [22] in our GPU cluster (Fig. 10b). In both cases, we find this model to be an accurate estimate of transmission times.

2. MPI [31] is another optional backend which requires rebuilding PyTorch from source; here we choose Gloo and NCCL as they are recommended by the PyTorch documentation for communication between CPUs and GPUs, respectively.

Since workers exchange and combine gradients with an AllReduce operation at the end of each training step, AllReduce implements the uplink, update and downlink phases in Sync-SGD with a decentralized architecture. A model of the duration T of each SGD step is then:

$$T = (T_F + T_B) + 2(K-1)M/(KB)\,. \tag{11}$$

3.2 Fine-grained Model

We now extend our fine-grained model of Async-SGD to Sync-SGD for both centralized and decentralized architectures, by modifying the communication mechanism and link sharing models.

3.2.1 Parameter Server Architecture

For the parameter server architecture, we include a synchronization barrier in the simulation (Line 36) to ensure that the next step starts only when all pending operations of the current step have completed (i.e., when Q is empty). In addition, similarly to Section 3.1.1, we repeat the simulation using two network sharing models: PS, where all workers share the network bandwidth equally (Line 5); and FCFS, where the next worker transmission receives the entire bandwidth, until completion (Line 2). After running the simulation with both sharing models, we use the average of their throughput estimates as our prediction.

3.2.2 Decentralized Architecture

In Sync-SGD with a decentralized architecture, all workers exchange gradients at the end of each training step with a Ring AllReduce operation. To predict throughput of this architecture, we modify our fine-grained simulation of Section 2.2: in addition to the synchronization barrier at Line 36, we skip all downlink operations (using $op.remaining$ equal to 0) and replace the duration of each uplink operation with that of an AllReduce operation with

$$op.remaining = \frac{2(K-1)}{K} \cdot \frac{op.size}{B}$$

where K is the number of workers and B is the network bandwidth, which is entirely available to the worker (Line 8) due to the ring topology.

4 Results

In this section, we validate the accuracy of the Async-SGD and Sync-SGD models described in Sections 2 and 3, respectively, using our implementation of distributed SGD based on PyTorch 1.7.

4.1 Experimental Setup

Table 4 summarizes our experimental setup: (1) CPU cluster with CloudLab d430 instances, (2) GPU cluster with AWS `p3.2xlarge` instances, (3) GPU cluster with AWS `g4dn.4xlarge` instances, and (4) GPU cluster with AWS `p3.16xlarge` instances. To validate our methods across a variety of DNNs, we select neural networks from a set of model families, including Inception networks [33], [32], ResNets [10], DenseNets [12] and EfficientNets [34]. Fig. 11 reports the model sizes and computation times (in a SGD step with batch size 32 on an Nvidia V100 GPU) of the neural networks in our experiments. For each scenario of distributed SGD, we train these neural network models using different batch sizes (indicated as L in Figs. 12 to 22) for 100 steps, and we measure job throughput as the total number of examples per second across all workers assigned to the training job.

3. Bandwidth for single-flow traffic is limited to 10Gbps.

	Cluster 1	Cluster 2	Cluster 3	Cluster 4
Type	CPU-only	single GPU	single GPU	multi-GPU
Node	CloudLab d430	AWS `p3.2xlarge`	AWS `g4dn.4xlarge`	AWS `p3.16xlarge`
CPU	Intel Xeon E5-2630 v3 (8-core)	Intel Xeon E5-2686 v4 (8 vCPUs)	custom Intel Cascade Lake (16 vCPUs)	Intel Xeon E5-2686 v4 (64 vCPUs)
GPU	—	NVIDIA Tesla V100	NVIDIA Tesla T4	8x NVIDIA Tesla V100
Network	1 Gbps	10 Gbps	10 Gbps	25Gbps[3]

Table 4: Experimental Setup

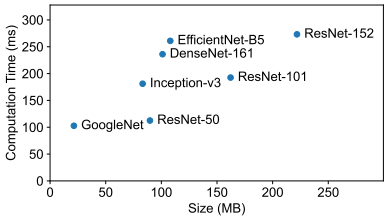


Figure 11: Performance Benchmark of Neural Network Models with Batch Size 32 on Nvidia Tesla V100 GPUs

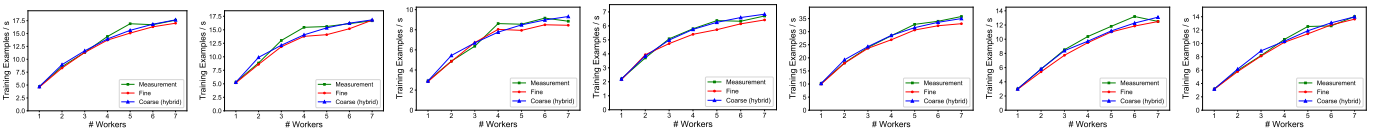
For the fine-grained model, we collect layer-level traces from 100 single-worker steps for each scenario (DNN model, batch size, node instance), inserting timestamps before and after the computation and transmission of every layer to denote its operation and to track dependencies between operations. In addition, we record the size of the parameters in each neural network layer in order to evaluate the communication time in the simulation (Section 2.2.1). For the coarse-grained model, we measure the time of feedforward and backpropagation stages, and the size of entire neural network models (Section 2.1.2).

4.2 Async-SGD

We now evaluate the prediction errors of our fine-grained and coarse-grained models of Async-SGD in three scenarios: homogeneous nodes without and with overlaps between communication and computation operations, as well as heterogeneous nodes.

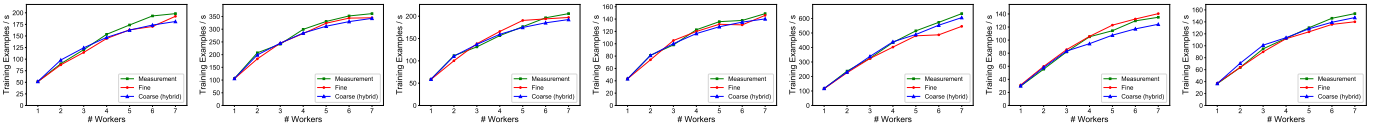
4.2.1 Basic Model

First, we evaluate fine-grained and coarse-grained models of Async-SGD without overlapping communication and computation. Here, we set the threshold ρ_T of our coarse-grained model to 0.5 for the CloudLab CPU cluster and 0.6 for the AWS GPU clusters. The selection of ρ_T was carried out through a one-time preliminary profiling. For each platform



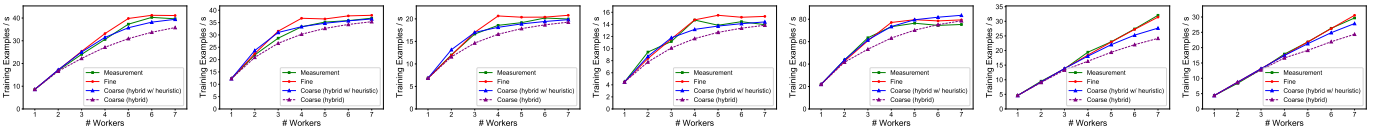
(a) Inception-v3, L=16 (b) ResNet-50, L=16 (c) ResNet-101, L=16 (d) ResNet-152, L=16 (e) GoogleNet, L=8 (f) EfficientNetB5, L=16 (g) DenseNet-161, L=16

Figure 12: Async-SGD without Overlapping Communication and Computation on Homogeneous CPUs



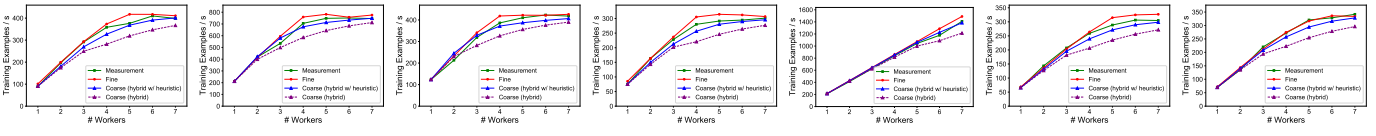
(a) Inception-v3, L=16 (b) ResNet-50, L=32 (c) ResNet-101, L=32 (d) ResNet-152, L=32 (e) GoogleNet, L=16 (f) EfficientNetB5, L=16 (g) DenseNet-161, L=16

Figure 13: Async-SGD without Overlapping Communication and Computation on Homogeneous GPUs



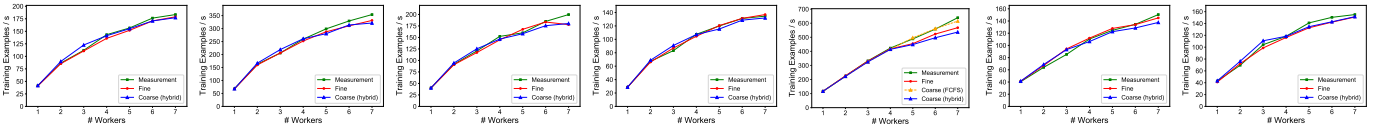
(a) Inception-v3, L=32 (b) ResNet-50, L=32 (c) ResNet-101, L=32 (d) ResNet-152, L=32 (e) GoogleNet, L=16 (f) EfficientNetB5, L=32 (g) DenseNet-161, L=32

Figure 14: Async-SGD with Overlapping Communication and Computation on Homogeneous CPUs



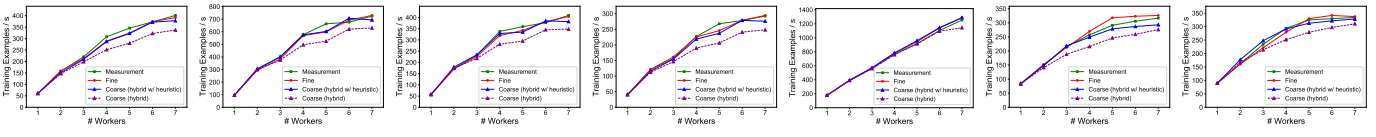
(a) Inception-v3, L=32 (b) ResNet-50, L=64 (c) ResNet-101, L=64 (d) ResNet-152, L=64 (e) GoogleNet, L=32 (f) EfficientNetB5, L=32 (g) DenseNet-161, L=32

Figure 15: Async-SGD with Overlapping Communication and Computation on Homogeneous GPUs



(a) Inception-v3, L=16 (b) ResNet-50, L=32 (c) ResNet-101, L=32 (d) ResNet-152, L=32 (e) GoogleNet, L=16 (f) EfficientNetB5, L=16 (g) DenseNet-161, L=16

Figure 16: Async-SGD without Overlapping Communication and Computation on Heterogeneous GPUs



(a) Inception-v3, L=32 (b) ResNet-50, L=64 (c) ResNet-101, L=64 (d) ResNet-152, L=64 (e) GoogleNet, L=32 (f) EfficientNetB5, L=32 (g) DenseNet-161, L=32

Figure 17: Async-SGD with Overlapping Communication and Computation on Heterogeneous GPUs

(AWS and CloudLab), we ran a job with 1 to 7 workers to select the value of ρ_T from $\{0.1, 0.2, ..., 1.0\}$ with best prediction accuracy on the preliminary profiling. Intuitively, we use a larger ρ_T for the GPU clusters on AWS because their background traffic can prevent the equal share of network bandwidth (modeled by the PS discipline), while we find networking to be more stable on CloudLab.

Fig. 12 depicts our predictions for Async-SGD on a CPU cluster (i.e., CloudLab d430 instances), where the fine-grained model achieves an average (maximum) error of 5.2% (10.8%), and the coarse-grained model achieves an average (maximum) error of 3.9% (11.8%). In addition, Fig. 13 shows the corresponding results on a GPU cluster (i.e., AWS `p3.2xlarge`), where the average (maximum) error is 5.4% (15.0%) for the fine-grained model and 4.7% (11.1%) for the coarse-grained model.

In this scenario, both coarse-grained and fine-grained models achieve similarly accurate predictions across all neural networks and batch sizes on CPU and GPU platforms. The worst case for the fine-grained model is in Fig. 13e for 6 and 7 workers, where we observe that data transmissions interleave with each other, while they overlap in our simulations, increasing the contention for network bandwidth and resulting in lower predicted throughput. This discrepancy is due to small model size of GoogleNet, which can be transmitted in just 17 ms using the AWS 10 Gbps network, a time interval too short for TCP/IP congestion control to adjust bandwidth sharing among workers.

4.2.2 *Overlapping Communication and Computation*

Next, we validate fine-grained and coarse-grained models of Async-SGD with overlapping communication and computation, and highlight the importance of applying the proposed heuristic in the case of the coarse-grained model.

In the CPU cluster experiment depicted in Fig. 14, the average (maximum) error is 4.3% (11.9%) for the fine-grained model and 4.0% (13.7%) for the coarse-grained model. In the GPU cluster experiment depicted in Fig. 15, the average (maximum) error is 4.4% (11.4%) for the fine-grained model and 4.7% (15.2%) for the coarse-grained model.

Note that, when using the coarse-grained model without the proposed heuristic (curve *"Coarse (hybrid)"* in Figs. 14 and 15), the average (maximum) error increases to 9.5% (24.6%) in the CPU-based experiments and to 11.6% (21.4%) in the GPU-based experiments. For most mispredictions, the model underestimates throughput because, when the number of workers increases, communication operations are longer and overlap with a more significant portion of computations; since the model assumes a sequential execution of these operations, predicted throughput is lower than our measures in the real system. After applying Eq. (6), the coarse-grained model achieves similar accuracy to the fine-grained model.

In contrast, the coarse-grained model shows lower accuracy in Fig. 15c for 2 workers: in this case, the estimated network utilization ρ^{FCFS} in our model is 0.53, which is lower than the threshold $\rho_T = 0.6$ for the AWS GPU cluster, so a FCFS model is adopted. However, measurements show that data transmissions partially overlap rather than interleave, and thus the network behaves more similarly to a PS discipline in the real system.

4.2.3 Heterogeneous Nodes

We validate our models of Async-SGD on heterogeneous compute nodes with AWS `p3.2xlarge` (V100 GPUs) and `g4dn.4xlarge` (T4 GPUs) instances, where V100 achieves approximately twice the performance (in FLOPS) of T4 [15], [14]. As shown in Fig. 16, without overlap between communication and computation, the average (maximum) error of the fine-grained model is 3.5% (11.1%) and that of the coarse-grained model is 4.8% (16.0%). Also, as depicted in Fig. 17, in the scenario with overlapping communication and computation, the average (maximum) error is 3.0% (9.6%) for the fine-grained model and 3.8% (11.4%) for the coarse-grained model. As before, the coarse-grained model not accounting for the overlap underestimates throughput, resulting in an average (maximum) error of 10.5% (23.0%). These results indicate that our coarse-grained and fine-grained models are both extendable to heterogeneous computing environments and achieve similarly accurate predictions across different neural network models.

In Fig. 16e, the coarse-grained model shows lower accuracy for 6 and 7 heterogeneous workers (11.0% and 16.1% error). Similarly to the case of homogeneous workers (Fig. 13e, 15.0% and 13.8% error for 6 and 7 workers, respectively), we observed that data transmissions interleave in the measurements, while our models incorrectly adopt the PS model. We attribute these mispredictions to the short transmission time of GoogleNet, which is not sufficient for the TCP/IP congestion control to adjust bandwidth sharing among workers. To motivate our interpretation, Fig. 16e includes predictions of the coarse-grained model with FCFS discipline (yellow), which achieves better accuracy.

4.3 Sync-SGD

We next evaluate the prediction errors of our fine-grained and coarse-grained models of Sync-SGD on both centralized and decentralized settings.

4.3.1 Centralized Setting

Figs. 18 and 19 depict predictions for centralized Sync-SGD on AWS `p3.2xlarge` instances. Both models achieve similarly accurate predictions: without overlapping communication and computation, the average (maximum) error is 5.2% (11.9%) for the fine-grained model and 3.2% (10.0%) for the coarse-grained model. In the case of overlapping communication and computation, the average (maximum) error is 4.7% (10.2%) for the fine-grained model and 4.1% (10.5%) for the coarse-grained model. Moreover, our heuristic significantly improve predictions in the case of overlaps, from an average (maximum) error of 18.3% (28.2%) as depicted by the curve *"Coarse (hybrid)"* in Fig. 19.

4.3.2 Decentralized Setting

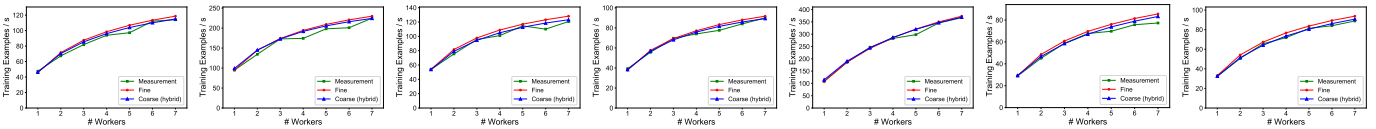
For the setting of decentralized Sync-SGD, we first evaluate our implementation using the Gloo backend on the CPU cluster, as shown in Fig. 20. Compared to the centralized setting where a network bottleneck can occur at the parameter server, the decentralized setting balances network traffic due to the AllReduce collective operations. Thus, the throughput of decentralized Sync-SGD grows almost linearly as the number of worker increases. In this case, the average (maximum) error of the fine-grained model is 2.3% (8.8%) and that of the coarse-grained model is 2.7% (12.8%).

Next, we evaluate the performance of the *DistributedDataParallel* [18] library of PyTorch v1.7 with the NCCL backend on the GPU cluster, without overlapping communication and computation. Results in Fig. 21 illustrate that the average (maximum) prediction error of the fine-grained model is 2.5% (9.3%) and that of the coarse-grained model is 5.0% (11.6%). Both models achieve accurate throughput predictions across all neural networks on the CPU and the GPU platforms.

4.4 Multi-GPU Experiments

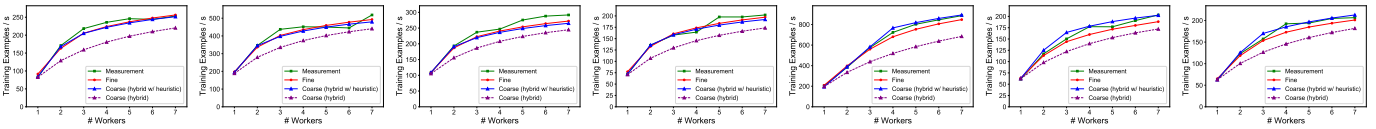
We also evaluate the prediction error of our models on AWS `p3.16xlarge` multi-GPU instances, which are configured with 8 Nvidia Tesla V100 GPUs and 25 Gbps network. Following [8], after all GPUs on a machine complete a training step, we first perform an AllReduce operation to aggregate gradients across all GPUs on the machine; then one GPU on the machine either pushes the aggregated gradients to the parameter server (centralized setting) or performs another AllReduce operation with other machines (decentralized setting); later, the GPU receiving the updated model broadcasts it to other GPUs on the same machine.

For the Parameter Server architecture, since AWS limits the bandwidth of each flow to 10 Gbps on `p3.16xlarge` instances, we observe that the downlink times of one and two workers in Sync-SGD remain the same, because network bandwidth is not fully utilized (Fig. 23); as we deploy more than three workers, the downlink time increases because multiple transmissions overlap, sharing the available bandwidth of 25 Gbps. In Figs. 22a to 22d, our coarse-grained models overestimate throughput by assuming that the entire



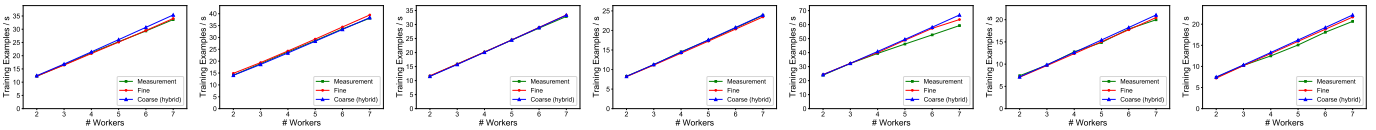
(a) Inception-v3, L=16 (b) ResNet-50, L=32 (c) ResNet-101, L=32 (d) ResNet-152, L=32 (e) GoogleNet, L=16 (f) EfficientNetB5, L=16 (g) DenseNet-161, L=16

Figure 18: Centralized Sync-SGD without Overlapping Communication and Computation on GPUs



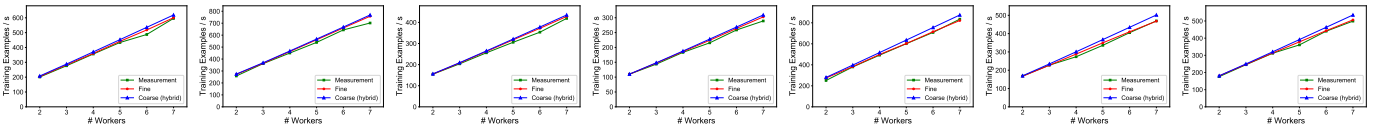
(a) Inception-v3, L=32 (b) ResNet-50, L=64 (c) ResNet-101, L=64 (d) ResNet-152, L=64 (e) GoogleNet, L=32 (f) EfficientNetB5, L=32 (g) DenseNet-161, L=32

Figure 19: Centralized Sync-SGD with Overlapping Communication and Computation on GPUs



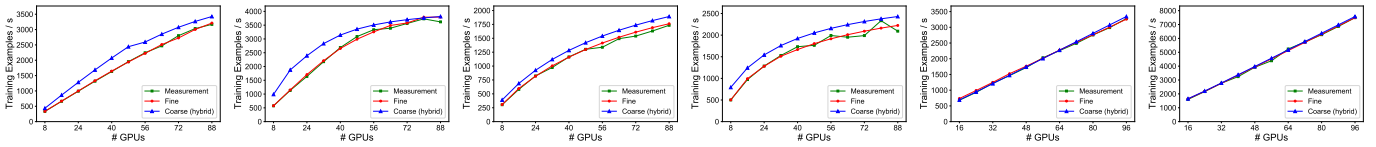
(a) Inception-v3, L=16 (b) ResNet-50, L=16 (c) ResNet-101, L=32 (d) ResNet-152, L=32 (e) GoogleNet, L=8 (f) EfficientNetB5, L=16 (g) DenseNet-161, L=16

Figure 20: Decentralized Sync-SGD with Gloo AllReduce Operations on CPUs



(a) Inception-v3, L=16 (b) ResNet-50, L=32 (c) ResNet-101, L=32 (d) ResNet-152, L=32 (e) GoogleNet, L=16 (f) EfficientNetB5, L=32 (g) DenseNet-161, L=32

Figure 21: Decentralized Sync-SGD with PyTorch *DistributedDataParallel* on GPUs



(a) Inception-v3, L=128, Centralized Async-SGD (b) VGG-11, L=128, Centralized Async-SGD (c) Inception-v3, L=128, Centralized Sync-SGD (d) VGG-11, L=128, Centralized Sync-SGD (e) ResNet152, L=128, Decentralized Sync-SGD (f) VGG-19, L=128, Decentralized Sync-SGD

Figure 22: Multi-GPU Experiments on AWS `p3.16xlarge`

bandwidth of 25 Gbps is available to individual flows (under either a PS or FCFS model); as a result, prediction errors can be significant, with average (maximum) 19.3% (65.7%) for Async-SGD and 13.6% (27.8%) for Sync-SGD. However, inaccurate predictions in Fig. 22b occur only with a small number of workers; in these cases, our traces show that data transmissions almost interleave, but each transmission uses only 10 Gbps (instead of the entire 25 Gbps assumed by our model) due to the bandwidth limitation of single flows on AWS. As the number of workers grows, prediction accuracy improves. In the case of fine-grained models, after we modify the SHARE procedure of Algorithm 2.1 to limit the bandwidth of individual transmission flows to 10 Gbps, we achieve accurate predictions, with average (maximum) errors 1.8% (4.8%) for Async-SGD and 3.0% (7.4%) for Sync-SGD. The results highlight that fine-grained models can be adapted more easily to new networking platforms, by adjusting the bandwidth assignment used in the simulation.

For decentralized Sync-SGD, since the NCCL backend can open a number of sockets to fully utilize the network capacity, both coarse-grained and fine-grained models obtain good predictions in Figs. 22e and 22f, with average (maximum) errors of 1.9% (7.9%) and 1.7% (4.3%), respectively.

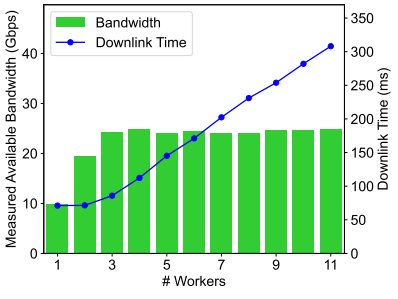


Figure 23: Available Bandwidth and Downlink Time on AWS 25Gbps Network of the Experiment in Fig. 22c

5 Related Work

Several performance models of ML jobs exist in the literature. Learning-based models like Optimus [26] and Ernest [36] propose blackbox models for performance prediction of large-scale ML jobs. Their blackbox models substantially rely on historical information of ML jobs to learn model parameters, and thus their performance is significantly affected when training data is limited. In contrast, our approach only uses profiling information from a single worker, and produces

predictions for an arbitrary number of workers. Moreover, learning-based models mainly focus on a specific scenario (where their training data was collected), without providing insights on how to adapt the model when the system or coordination mechanism change (e.g., in the case of overlapping communication and computation).

In addition to learning-based models, analytical models for Sync-SGD have been proposed in the literature with different levels of granularity. Coarse-grained analytical models of Sync-SGD in Paleo [27] and [25] predict computation time based on FLOP counts or instruction cycles of neural network layers, respectively. Even when these analytical models accurately describe specific algorithms for convolutional layers (e.g., GEMM, FFT), they may not be applicable to new distributed SGD algorithms of the constantly evolving ML frameworks. On the other hand, an accurate analytical model of the computation time can be easily integrated into our coarse-grained and fine-grained models, by replacing the profiled computation time with the analytical estimate. In addition, these coarse-grained models omit layer-level behavior, resulting in less accurate predictions in the case of overlapping communication and computation.

[30] adopts comprehensive measurements on every component of a Sync-SGD step, with the purpose of identifying bottlenecks in an SGD step under various settings, such as multiple ML frameworks and inter-node connections (e.g., TCP over InfiniBand or NCCL on GPUDirect). Similarly to our fine-grained model, the authors compare the communication time of every layer with the backward computation time of the previous layer, to estimate the overlap of communication and computation; however, they measure the duration of every activity of the model, instead of predicting job throughput with respect to the number of workers, which is the focus of our work.

While the majority of analytical approaches focus on Sync-SGD, modeling Async-SGD is more difficult because communication patterns between parameter servers and workers are more complex and can change over time. Cynthia [37], the closest work to our paper, discusses both Sync-SGD with overlaps between communication and computation, and Async-SGD without overlaps. However, for the estimation of communication time, it assumes a PS model for Sync-SGD, while we illustrate the necessity of combining FCFS and PS disciplines in the case of unequal network bandwidth shares in Section 3.1; furthermore, the assumption of constant communication time for Async-SGD only works for lower network utilization, where transmissions by multiple workers tend to completely interleave with each other (Section 2.1). Fig. 4b compares Cynthia with our models for a training job of ResNet-50 on AWS `p3.2xlarge` instances, where Cynthia mispredicts throughput after network saturation.

Some works propose performance models for other communication mechanisms; for example, [35] studies the scalability of machine learning algorithms based on MapReduce framework Apache Spark. The authors present a coarse-grained model for Sync-SGD with two communication protocols for gradient distribution and aggregation in Spark. Gradient distribution uses a tree topology, where communication time is estimated from the logarithm of the number of workers; gradient aggregation is performed on a square root number of nodes at first, and then extended to the remaining nodes. However, the proposed models are restricted to MapReduce, which is rarely used for DNN training. Notably, our models can be extended to MapReduce computations by using a modified estimate of communication time.

6 Conclusions

We proposed both coarse-grained and fine-grained models to predict the scaling characteristics of distributed SGD. Fine-grained models can be adapted more easily to variants of communication mechanisms used by workers to exchange model updates, but they require ML framework profilers and may incur profiling overhead when measuring low-level GPU information. In contrast, parameters of coarse-grained models can be obtained more easily through simple profiling, but these models are less accurate in some scenarios due to missing layer-level information.

We thoroughly validated our models in various scenarios of distributed SGD, including synchronous and asynchronous strategies, centralized and decentralized architectures, overlapping communication and computation, unequal network bandwidth distribution and heterogeneous platforms. Experimental results highlight that our heuristics effectively improve the accuracy of coarse-grained models in the case of overlapping communication and computation, and that both fine-grained and coarse-grained models can achieve accurate predictions in all distributed SGD scenarios.

As future directions, we plan to extend our models to other implementations of distributed SGD, using dedicated interconnect technologies such as GPUDirect and NVLink, as well as other communication backends such as MPI. We also plan to explore multi-NIC configurations to increase bandwidth and improve scalability of centralized approaches.

Acknowledgements

This work was supported in part by the NSF CCF-1763747 and the NSF CNS-1816887 awards.

References

[1] M. Abadi et al. TensorFlow: A system for large-scale machine learning. In *OSDI'16*, pages 265–283, 2016.
[2] T. Ben-Nun and T. Hoefler. Demystifying parallel and distributed deep learning: An in-depth concurrency analysis. *ACM Computing Surveys (CSUR)*, 52(4):1–43, 2019.
[3] J. Chen, R. Monga, S. Bengio, and R. Józefowicz. Revisiting distributed synchronous SGD. *CoRR*, abs/1604.00981, 2016.
[4] M. Cho, U. Finkler, S. Kumar, D. Kung, V. Saxena, and D. Sreedhar. Powerai ddl. *arXiv preprint arXiv:1708.02188*, 2017.
[5] J. Dean and L. A. Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, 2013.
[6] D. Duplyakin, R. Ricci, A. Maricq, G. Wong, J. Duerig, E. Eide, L. Stoller, M. Hibler, D. Johnson, K. Webb, et al. The design and operation of CloudLab. In *2019 USENIX*, pages 1–14, 2019.
[7] Gloo. Collective communications library. https://github.com/facebookincubator/gloo, 2017.
[8] P. Goyal, P. Dollár, R. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He. Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv:1706.02677*, 2017.
[9] S. Hadjis, C. Zhang, I. Mitliagkas, and C. Ré. Omnivore: An optimizer for multi-device deep learning on cpus and gpus. *CoRR*, abs/1606.04487, 2016.
[10] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *CVPR'16*, pages 770–778, 2016.

[11] Q. Ho, J. Cipar, H. Cui, S. Lee, J. K. Kim, P. B. Gibbons, G. A. Gibson, G. Ganger, and E. P. Xing. More effective distributed ML via a stale synchronous parallel parameter server. *Advances in neural information processing systems*, 26:1223–1231, 2013.

[12] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger. Densely connected convolutional networks. In *IEEE CVPR*, pages 4700–4708, 2017.

[13] X. Jia, S. Song, W. He, Y. Wang, H. Rong, F. Zhou, L. Xie, Z. Guo, Y. Yang, L. Yu, et al. Highly scalable deep learning training system with mixed-precision: Training imagenet in four minutes. *arXiv preprint arXiv:1807.11205*, 2018.

[14] Z. Jia, M. Maggioni, J. Smith, and D. P. Scarpazza. Dissecting the NVidia Turing T4 GPU via microbenchmarking. *arXiv preprint arXiv:1903.07486*, 2019.

[15] Z. Jia, M. Maggioni, B. Staiger, and D. P. Scarpazza. Dissecting the NVIDIA Volta GPU architecture via microbenchmarking. *arXiv preprint arXiv:1804.06826*, 2018.

[16] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *ISCA'17*, pages 1–12, 2017.

[17] Y. LeCun, Y. Bengio, and G. Hinton. Deep Learning. *Nature*, 521(7553):436–444, 2015.

[18] S. Li et al. PyTorch distributed: Experiences on accelerating data parallel training. *arXiv preprint arXiv:2006.15704*, 2020.

[19] Z. Li, W. Yan, M. Paolieri, and L. Golubchik. Throughput prediction of asynchronous SGD in TensorFlow. In *ICPE'20*, pages 76–87. ACM, 2020.

[20] X. Lian, W. Zhang, C. Zhang, and J. Liu. Asynchronous decentralized parallel stochastic gradient descent. In *International Conference on Machine Learning*, pages 3043–3052. PMLR, 2018.

[21] S.-H. Lin, M. Paolieri, C.-F. Chou, and L. Golubchik. A model-based approach to streamlining distributed training for asynchronous SGD. In *MASCOTS'18*, pages 306–318, 2018.

[22] NCCL. Nvidia collective communication library. https://developer.nvidia.com/nccl, 2016.

[23] A. Paszke et al. PyTorch: An imperative style, high-performance deep learning library. *arXiv preprint arXiv:1912.01703*, 2019.

[24] P. Patarasuk and X. Yuan. Bandwidth optimal all-reduce algorithms for clusters of workstations. *Journal of Parallel and Distributed Computing*, 69(2):117–124, 2009.

[25] Z. Pei, C. Li, X. Qin, X. Chen, and G. Wei. Iteration time prediction for CNN in multi-GPU platform: Modeling and analysis. *IEEE Access*, 7:64788–64797, 2019.

[26] Y. Peng, Y. Bao, Y. Chen, C. Wu, and C. Guo. Optimus: an efficient dynamic resource scheduler for deep learning clusters. In *EuroSys'18*, pages 3:1–3:14, 2018.

[27] H. Qi, E. R. Sparks, and A. Talwalkar. Paleo: A performance model for deep neural networks. In *ICLR'2017*, 2017.

[28] M. Reiser. A queueing network analysis of computer communication networks with window flow control. *IEEE transactions on Communications*, 27(8):1199–1209, 1979.

[29] M. Reiser and S. S. Lavenberg. Mean-value analysis of closed multichain queuing networks. *JACM*, 27(2):313–322, 1980.

[30] S. Shi, Q. Wang, and X. Chu. Performance modeling and evaluation of distributed deep learning frameworks on gpus. In *DASC/PiCom/DataCom/CyberSciTech'18*, pages 949–957, 2018.

[31] M. Snir, W. Gropp, S. Otto, S. Huss-Lederman, J. Dongarra, and D. Walker. *MPI–the Complete Reference*, volume 1. MIT press, 1998.

[32] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In *CVPR'15*, pages 1–9, 2015.

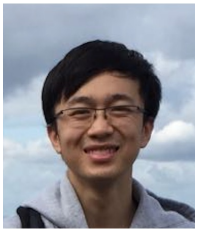
[33] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna. Rethinking the inception architecture for computer vision. In *CVPR'16*, pages 2818–2826, 2016.

[34] M. Tan and Q. Le. Efficientnet: Rethinking model scaling for convolutional neural networks. In *International Conference on Machine Learning*, pages 6105–6114. PMLR, 2019.

[35] A. Ulanov, A. Simanovsky, and M. Marwah. Modeling scalability of distributed machine learning. In *ICDE'17*, pages 1249–1254, 2017.

[36] S. Venkataraman, Z. Yang, M. Franklin, B. Recht, and I. Stoica. Ernest: Efficient performance prediction for large-scale advanced analytics. In *NSDI'16*, pages 363–378, 2016.

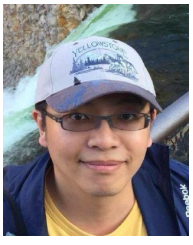
[37] H. Zheng, F. Xu, L. Chen, Z. Zhou, and F. Liu. Cynthia: Cost-efficient cloud resource provisioning for predictable distributed deep neural network training. In *ICPP'19*, pages 86:1–86:11, 2019.



Zhuojin Li is a PhD student at the University of Southern California, Los Angeles, USA. He received his M.S. in Computer Engineering from the University of Southern California (2020) and his B.S. in Computer Science from Peking University, China (2018). His research interests focus on performance evaluation and modeling of large-scale machine learning systems.

Marco Paolieri is a Senior Research Associate at the University of Southern California, Los Angeles, USA. He received his Ph.D. in Computer Science, Systems, and Telecommunications (2015) and his M.S. in Computer Engineering (2011) from the University of Florence, Italy. His research interests focus on stochastic modeling and quantitative evaluation of performance and reliability in concurrent and distributed systems.

Leana Golubchik is a Stephen and Etta Varra professor of computer science and ECE systems with the University of Southern California, Los Angeles, CA. Prior to that, she was with the University of Maryland and Columbia University. Her research interests are broadly in the design and evaluation of large scale distributed systems including hybrid clouds and their applications in data analytics, QoS-based design of P2P and multimedia systems, and reliability of software architectures. She is a past chair of ACM SIGMETRICS and a member of the IFIP WG 7.3.



Sun-Han Li received his Ph.D. degree from the University of Southern California (USC), Los Angeles, CA, in 2017. He has been a performance analyst at NetApp, USA, from October 2017 to January 2021; he is now a Performance and Capacity Engineer at Facebook. His main research interests include the performance modeling, analysis, and design of large-scale distributed systems, including cloud computing systems, peer-to-peer networking, and multimedia systems. He is a member of the IEEE and ACM.

Wumo Yan received his B.S. in Electrical Engineering and Computer Science from National Tsing Hua University in 2017. He is a PhD student in Computer Science at the University of Southern California, Los Angeles, CA. His current research focuses on distributed machine learning systems.