Ako: Decentralised Deep Learning with Partial Gradient Exchange

Pijika Watcharapichat[1] Victoria Lopez Morales[1] Raul Castro Fernandez[2] Peter Pietzuch[1]
[1]Imperial College London [2]MIT
{pw610, v.lopez-morales, prp}@imperial.ac.uk, raulcf@csail.mit.edu

Abstract

Distributed systems for the training of *deep neural networks* (DNNs) with large amounts of data have vastly improved the accuracy of machine learning models for image and speech recognition. DNN systems scale to large cluster deployments by having *worker nodes* train many model replicas in parallel; to ensure model convergence, *parameter servers* periodically synchronise the replicas. This raises the challenge of how to split resources between workers and parameter servers so that the cluster CPU and network resources are fully utilised without introducing bottlenecks. In practice, this requires manual tuning for each model configuration or hardware type.

We describe *Ako*, a *decentralised* dataflow-based DNN system without parameter servers that is designed to saturate cluster resources. All nodes execute workers that fully use the CPU resources to update model replicas. To synchronise replicas as often as possible subject to the available network bandwidth, workers exchange *partitioned gradient updates* directly with each other. The number of partitions is chosen so that the used network bandwidth remains constant, independently of cluster size. Since workers eventually receive all gradient partitions after several rounds, convergence is unaffected. For the ImageNet benchmark on a 64-node cluster, Ako does not require any resource allocation decisions, yet converges faster than deployments with parameter servers.

Categories and Subject Descriptors C.2.4 [*Distributed Systems*]: Distributed applications; D.4.7 [*Organization and Design*]: Batch processing systems, Distributed systems

Keywords Deep Learning, Distributed Machine Learning, Decentralised Architecture

SoCC '16, October 05 – 07, 2016, Santa Clara, CA, USA.
© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ISBN 978-1-4503-4525-5/16/10...$15.00.
DOI: http://dx.doi.org/10.1145/2987550.2987586

1. Introduction

Deep neural networks (DNNs) [18, 22] have revolutionised the accuracy of machine learning models in many areas, including image classification [10, 20], speech recognition [17] and text understanding [6, 34]. This breakthrough was made possible by scalable distributed systems [5, 12, 24, 27, 35] that train DNNs in parallel on compute clusters, incorporating large volumes of training data in a manageable amount of time.

A common architecture for DNN systems takes advantage of data-parallelism [3, 28]: a set of *worker* nodes train model replicas on partitions of the input data in parallel; the model replicas are kept synchronised by a set of *parameter servers*—each server maintains a global partition of the trained model. Periodically workers upload their latest updates to the parameter servers, which aggregate them and return an updated global model.

While initial synchronisation strategies were synchronous [40], potentially limiting scalability due to straggling workers [7], the latest generation of DNN systems, such as Parameter Server [24], Project Adam [5], Singa [27] and Bösen [44], employs weaker consistency models between replicas: model replicas are exchanged *asynchronously* with parameter servers, while imposing an upper bound on divergence ("bounded staleness") [2, 19, 26, 49].

An open problem, which we explore experimentally in §2.3, is that DNN systems must balance the use of compute and network resources to achieve the fastest model convergence. In a typical deployment, the workers are compute-bound, and the parameter servers are network-bound [5]: if a deployment has too few parameter servers, the model may not converge or only converge slowly because model replicas are not synchronised frequently enough due to the limited network bandwidth between workers and parameter servers; with too many parameter servers, compute resources, which could rather be used for additional model replicas, are wasted, leading to slower convergence.

An optimal resource allocation for workers and parameter servers depends on many factors, including (i) the capabilities of the node hardware, (ii) the available network bandwidth, (iii) the size and model complexity, and (iv) the properties of the training data. In practice, users must decide on a resource allocation empirically for a given deployment, often resorting to a trial-and-error approach [47]. While such tuning may be feasible for large Internet companies with resource-rich and bespoke DNN systems [1, 5, 12, 45], it remains time-consuming and cumbersome. Recent work [47] models the performance of DNN systems offline, but suffers from poor accuracy. Co-locating workers and parameter servers [44] is also not a solution because it allocates resources without awareness of the performance impact (see §2.3).

Our goal is to design a DNN system that always utilises the full CPU resources and network bandwidth of a cluster. To avoid having to split resources between workers and parameter servers, we devise an architecture that does not have parameter servers but only homogenous workers.

The challenge is that such a *decentralised* DNN system must still (i) be *scalable* in terms of its network usage for model synchronisation and (ii) be *efficient* with a fast convergence time, i.e. comparable to designs with parameter servers. In particular, all-to-all synchronisation of model replicas between workers does not scale due to its quadratic communication cost [23, 50]. Prior work [23, 41] proposed indirect synchronisation topologies in which traffic is relayed by workers, but this impacts convergence time due to the higher latency [41].

We describe **Ako**,[1] a decentralised DNN system in which homogeneous workers train model replicas and synchronise directly with each other in a peer-to-peer fashion. Despite not having parameter servers, Ako does not sacrifice scalability or efficiency due to two features:

Scalable decentralised synchronisation (§3). Ako adopts a new approach to synchronise model replicas termed *partial gradient exchange*: each worker periodically computes an updated model gradient, partitions the gradient according to the available network bandwidth, and exchanges the partitions with other workers. Since a worker receives a different gradient partition from other workers in each synchronisation round, partial gradient exchange can maintain convergence as workers eventually receive the complete model gradient with bounded delay.

Decoupled CPU and network use (§4). An Ako worker decouples its use of CPU resources for model training from the use of network bandwidth for replica synchronisation: parallel compute tasks train the model replica as fast as possible, generating model gradient updates. Gradient updates are accumulated asynchronously by separate network tasks when awaiting transmission. Before transmission, the accumulated model gradients are partitioned so that synchronisation traffic fully saturates the network bandwidth while maintaining a constant latency.

[1] "to learn" in the Maori language

In §5, we evaluate a prototype version of Ako implemented using the SEEP stateful distributed dataflow platform [14]. For the ImageNet benchmark, we show that Ako, without requiring any resource configuration on the cluster, exhibits a 25% lower time-to-convergence than a hand-tuned deployment with parameter servers. It also results in better model convergence in comparison to TensorFlow [1] and Singa [27]. Ako achieves this by having a higher hardware efficiency through the full utilisation of cluster resources, thus compensating for its lower statistical efficiency due to only exchanging partial gradients.

2. Resource Allocation in DNN Systems

Next we describe the process for training DNNs (§2.1) and how it can be parallelised with a parameter server architecture (§2.2). We then show empirically how deploying such a system on a compute cluster is challenging: to reduce the training time, it is necessary to find the right assignment of CPU and network resources to workers and parameter servers (§2.3).

2.1 A primer on deep learning

Deep neural networks (DNNs) achieve the highest accuracy among machine learning models for pattern recognition and classification problems [20, 35], but they are computationally expensive to train. DNNs feature multiple levels of representation, through the composition of non-linear simple modules, and thus can express higher-level, abstract concepts. A DNN is represented as a set of *neurons* organised in *layers*, whose outputs are used as input for successive layers. It typically has one input and one output layer with multiple intermediate hidden layers.

The function calculated by each neuron based on an input x can be expressed as:

$$o_j(x) = \varphi\left(\sum_{i=1}^{k} w_{ji} \cdot x_i + b\right) \tag{1}$$

where x is the input vector of size k to the neuron, w_j is the set of input weights for the neuron representing a given problem-specific feature, b is a bias parameter, and φ is an activation function.

The goal of training a DNN is to find the weights w_{ji} for all the neurons that produce the correct response for a given set of data points. For simplicity, we will refer to all the weights in the DNN as w. For complex problems, the size of w is on the order of millions [5, 8, 12].

DNNs are trained using several epochs, i.e. doing multiple passes over the data. Typically, the weights w are computed by the *backpropagation* algorithm with gradient descent [30], which updates w iteratively:

$$w_{t+1} = w_t - \eta \cdot \nabla F(w) \tag{2}$$

where w_t are the current weights in iteration t, η is the learning rate and $\nabla F(w)$ are the calculated gradients over a

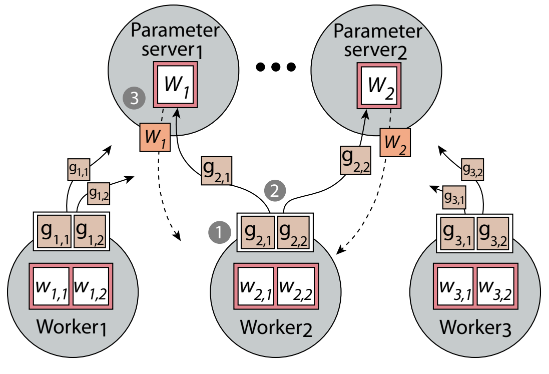


Figure 1: DNN architecture with parameter servers

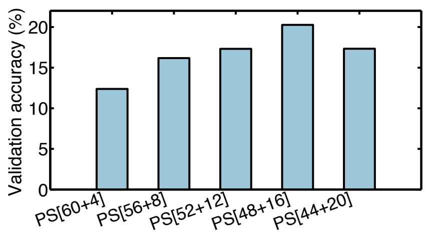


Figure 2: Accuracy for different worker/parameter server allocations in cluster

cost function, generally the error. For the remainder of the paper, we will refer to $\nabla F(w)$ just as the *gradients* g. To compute the gradients, forward propagation first obtains the output layer activations, whose error towards the real output is used then for the backward propagation.

For gradient descent, there are several possibilities when the weight updates can be applied. In stochastic learning, each propagation of a data point is followed immediately by a weight update. For efficiency, a common technique is to use *mini-batches*, i.e. use stochastic learning with more than one data point at a time.

2.2 DNN systems with parameter servers

DNNs with high accuracy require millions of features, organised into tens of layers, and they must be trained by iterating repeatedly over gigabytes or terabytes of data [5, 35]. To have an acceptable *time-to-convergence*, i.e. the training time required to reach a given accuracy, they must execute on compute clusters, ranging from tens to thousands of machines [5, 12, 35].

A scalable approach for training DNNs is to use a *parameter server* architecture [24], as shown in Fig. 1. The training data is split across *worker nodes*. Each worker calculates the gradients g in parallel over its data partition, and refines a *model replica*, i.e. its own weights w, according to the backpropagation algorithm (step 1). In this example, the model at worker j contains two weight parameters, $w_{j,1}$ and $w_{j,2}$, whose gradients are represented as $g_{j,1}$ and $g_{j,2}$, respectively.

After a mini-batch is processed, a worker only has a model w that is updated with its own gradient from the local data. To obtain a global model of *all* data, W, the workers synchronise their models through *parameter servers*. The parameter servers update the global model W by aggregating the local gradients g (step 2), and return a new global model W to the workers (step 3). To prevent the communication from becoming a bottleneck, each parameter server is responsible for a disjoint part of the model W_n, only managing the weights of the corresponding layers and/or neurons.

Previous research has shown that the global model can be updated asynchronously—with each worker synchronising independently—and still converge [5, 12]. To avoid divergence between workers, systems include *staleness* thresholds, i.e. an upper bound on how many model updates can be missed by a worker [2, 19, 26, 49].

In a distributed setting, a DNN system must therefore optimise two different performance aspects to reach the fastest time-to-convergence. It must achieve:

(i) high **hardware efficiency**, which is the time to complete a single iteration. With more workers, a system increases parallelism and thus reduces iteration time; and

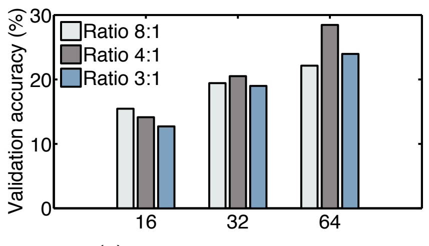
(ii) high **statistical efficiency**, which is the improvement in the model per iteration. For this, workers must synchronise their model replicas as often as possible to maximise global information gain. Since the synchronisation frequency is limited by the available network bandwidth, DNN systems scale out via multiple parameter servers to improve statistical efficiency.

There is a trade-off between hardware efficiency and statistical efficiency when allocating resources for workers and parameter servers in a fixed-sized compute cluster: assigning more machines to workers improves hardware efficiency, but it reduces statistical efficiency unless more parameter servers are added; conversely, more parameter servers increase the network bandwidth for model synchronisation. This permits more frequent model updates, thus improving statistical efficiency, but if the parameter servers take resources away from the workers, hardware efficiency is reduced.

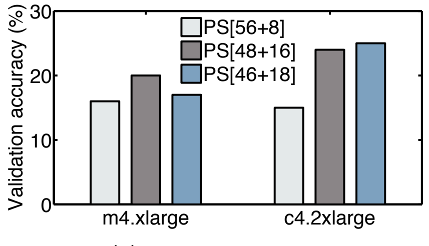
In practice, modern distributed deep learning systems [1, 4, 5, 27] require such decisions on resource allocation [47]. For a given resource budget, typically the majority of machines execute as workers while the rest form a group of distributed parameter servers, together maintaining the global model. For example, TensorFlow [1] supports a typical resource split resulting in several worker machines that synchronise with the centralised group of parameter servers; Singa [27, 42, 43] supports even more advanced cluster configurations with multiple parameter server groups.

2.3 Resource allocation problem

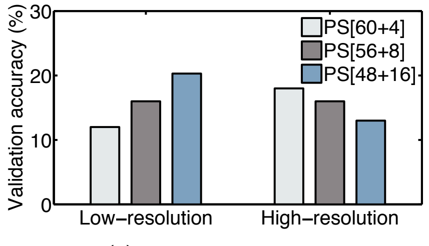
An optimal resource allocation for workers and parameter servers should result in the fastest time-to-convergence. Next we show experimentally that the best allocation depends on many factors, including the cluster size, the hardware capabilities, and the training data.



(a) Different cluster sizes



(b) Different hardware



(c) Different workloads

Figure 3: Effect of system and workload changes on best resource allocation

We deploy a DNN system with parameter servers on a 64-machine cluster, training a model for the ImageNet benchmark (see §5.1 for details). Fig. 2 shows the accuracy after one hour of training for different resource allocations between workers and parameter servers on the cluster. The result shows that the best accuracy is achieved for an allocation of 48 workers and 16 parameter servers. Note that the extreme allocations that emphasise hardware efficiency (60 workers) or statistical efficiency (20 servers) both exhibit a 38.9% and 14.5% worse accuracy, respectively, than the best allocation.

Cluster size. The ratio of the optimal allocation between workers and parameter servers changes with different cluster sizes. Fig. 3(a) shows the accuracy for three allocation ratios (3:1, 4:1 and 8:1) as the cluster size changes. While a 8:1 ratio (i.e. 2 parameter servers) yields the best accuracy for a 16-machine deployment, this is not the case when the cluster size increases: with a 32-machine or 64-machine cluster, a ratio of 4:1 achieves the best accuracy for this workload.

Hardware. The best allocation also depends on the machine hardware. In Fig. 3(b), we show the accuracy of the ImageNet DNN model for two different hardware configurations ("m4.xlarge" and "c4.2xlarge" VMs) on a 64-machine Amazon EC2 deployment. For the slower "m4.xlarge" VMs, an allocation of 16 parameter servers gives the highest accuracy of 20%, but, with the faster "c4.2xlarge" VMs, 18 parameter servers achieve 25%.

Workload. In practice, input data changes, e.g. when new types of training data become available, which also affects the optimal allocation. Next, we vary the training data for the ImageNet benchmark between low-resolution (100×100 pixel) and high-resolution (200×200 pixel) images. Fig. 3(c) shows that the low-resolution images achieve the highest accuracy with 48 workers and 16 parameter servers. This, however, turns out to be the worst allocation for the high-resolution images, which perform best with 60 workers and only 4 servers.

Co-located deployment. A heuristic is to colocate each worker with a parameter server on the same machine [44]. Such an approach, however, does not solve the problem: as we show in §5.2, it exhibits worse convergence due to the large number of global model partitions. In addition, it still

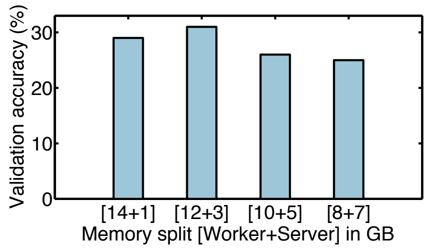


Figure 4: Effect of memory allocation with co-location

requires a decision on how to share the resources on each machine: besides allocating CPU threads, the memory of the machine must be shared.

Fig. 4 shows the accuracy achieved on a co-located 32-machine deployment after one hour of training with different memory allocations between the workers and the parameter servers. While each machine has 16 GB of RAM, an allocation of 3 GB for the parameter server and 12 GB to the workers results in the highest accuracy (31%); an equal memory split achieves the worst accuracy of 25%.

3. Partial Gradient Exchange

Instead of using parameter servers to synchronise the model updates produced by workers, we adopt a *decentralised* synchronisation scheme in which workers communicate directly with each other, without intermediate nodes. This avoids the challenge of having to decide on a resource split between workers and parameter servers.

A strawman solution for decentralised synchronisation is to use *all-to-all* communication between the workers, but this does not scale: n workers would require $O(n^2)$ network bandwidth for the synchronisation, but worker bandwidth only grows linearly with cluster size, $O(n)$.

To reduce the bandwidth requirement of decentralised synchronisation, workers could propagate model updates *indirectly*, i.e. with some workers relaying updates [41]. Such an approach, however, degrades statistical efficiency because it suffers from higher synchronisation latency, and it requires typically additional all-to-all full model exchanges [23].

We thus want to design a new decentralised synchronisation approach that (i) *scales* near linearly with the cluster size, and (ii) also exhibits *high statistical efficiency* that matches current parameter-server-based approaches.

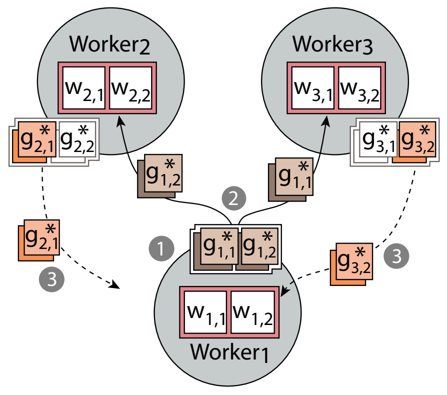
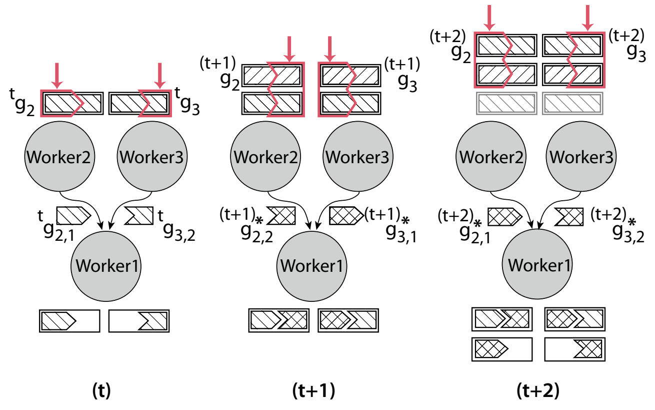


Figure 5: Decentralised DNN architecture with partial gradient exchange



${}^tg_2$ ${}^tg_3$
${}^{(t+1)}g_2$ ${}^{(t+1)}g_3$
${}^{(t+2)}g_2$ ${}^{(t+2)}g_3$
Worker2
Worker3
${}^tg_{2,1}$ ${}^tg_{3,2}$
${}^{(t+1)}g^*_{2,2}$ ${}^{(t+1)}g^*_{3,1}$
${}^{(t+2)}g^*_{2,1}$ ${}^{(t+2)}g^*_{3,2}$
Worker1
(t)
(t+1)
(t+2)

Figure 6: Accumulation of gradient partitions

3.1 Partial gradient exchange algorithm

We describe a new approach called *partial gradient exchange* that reduces the communication cost of gradient synchronisation. In partial gradient exchange, workers create disjoint *partitions* of the full gradient update. In a single round, a worker sends only one partition to each other worker, with the remaining partitions transmitted in subsequent rounds. While gradient partitions await transmission, workers update them locally as new gradients become available, which ensures that model updates are propagated with low latency.

Partial gradient exchange can thus control the network usage based on the size of the gradient partitions, while still maintaining high statistical efficiency without centralised parameter servers.

Gradient partitioning. Fig. 5 illustrates the synchronisation procedure with partial gradient exchange. In step 1, each worker j processes the m data points in its mini-batch and creates a local gradient g_j. Each worker then accumulates the gradient with any previously-unsent local gradients g_j^* (see below), and partitions it into p disjoint *gradient partitions*, $(g^*_{j,1},\dots,g^*_{j,p})$ where, in this example, p is equal to 2.

Each gradient partition, $g^*_{j,i}$, $i \in [1\dots p]$, is sent to other workers in a round-robin fashion (step 2). If p is equal to the number of workers, each worker receives a different gradient partition; if p is smaller, multiple workers receive the same partition; and if p is larger, only a subset of all partitions is exchanged in a single round.

Since all workers perform the partial gradient exchange concurrently, the other workers also share their gradient partitions (step 3). The received gradient partitions are applied to the local model w_j by updating the weights. We refer to the above three steps as a *synchronisation round*.

Gradient accumulation. A single synchronisation round does not send the complete gradient for a mini-batch to all workers. Instead, it takes p synchronisation rounds to transfer the complete gradient at time t. To avoid discarding unsent gradients while new gradients are calculated continuously, gradient partitions are accumulated across synchronisation rounds. By doing this, workers eventually receive complete gradient information from others after some delay.

Fig. 6 shows how gradients are accumulated. At time t, each worker j computes its local gradient ${}^tg_j$, which is partitioned. The worker checks if there are previously-generated gradients, and as there are none, each partition is sent directly to the other workers.

In the next synchronisation round $t+1$, each worker produces a new gradient ${}^{(t+1)}g_j$. Since there exist previous gradients ${}^tg_j$, they are accumulated through addition, ${}^{(t+1)}g_j^* = {}^tg_j + {}^{(t+1)}g_j$, before being partitioned and sent to the other workers. After this synchronisation round, the gradients ${}^tg_j$ computed at time t have been transmitted to the other workers, thus completing the t mini-batch.

In the following synchronisation rounds, the process is analogous, limiting the accumulation to the last p generated gradients to avoid the transmission of already-sent gradients. This can be seen in Fig. 6 for the synchronisation round $t+2$, in which only the last $p=2$ generated gradients, ${}^{(t+1)}g_j$ and ${}^{(t+2)}g_j$, are accumulated.

Since the communication is asynchronous, accumulated gradient partitions may not be received in their expected synchronisation rounds. Although this introduces staleness in the local model, it does not compromise convergence, as we explain in §3.3.

Algorithm. We formalise partial gradient exchange in Alg. 1. Each worker executes two functions, `generateGradients` and `updatePartialModel`, asynchronously.

The `generateGradients` function computes the local gradient ${}^tg_j$ from the training data d and the local weights ${}^tw_j$ (line 4). It then updates the local weights (line 5) and accumulates the last p-generated gradients in an incremental fashion (line 6). After that, a worker creates (line 7) and disseminates (line 10) the gradient partitions to the other workers. This process is repeated until convergence is reached (line 2): the procedure is stopped when the val-

Algorithm 1: Partial gradient exchange

```
1  function generateGradients (j, d, t, $\eta$, $\tau$)
       input : worker index $j$, mini-batch data points $d$,
               gradient computation timestamp $t$, learning
               rate $\eta$, staleness bound $\tau$
2      while ¬converged do
3          if $c_j \le min\,(s_{j,1}, \ldots, s_{j,n}) + p + \tau$ then
4              ${}^{t}g_j \leftarrow$ computeGradient $({}^{t}w_j, d)$
5              ${}^{(t+1)}w_j \leftarrow {}^{t}w_j + \eta \cdot {}^{t}g_j$
6              ${}^{t}g_j^* \leftarrow {}^{(t-1)}g_j^* + {}^{t}g_j - {}^{(t-p)}g_j$
7              $({}^{t}g_{j,1}^*, \ldots, {}^{t}g_{j,p}^*) \leftarrow$ partitionGrad $({}^{t}g_j^*, p)$
8              for $i = 1 \ldots n$ in parallel do
9                  $k \leftarrow i \bmod p$
10                 sendGradient $(i, {}^{t}g_{j,k}^*)$
11             $c_j \leftarrow c_j + 1$

12 function updatePartialModel (j, i, $g_{j,p}$, $\eta$)
       input : receiver worker index $j$, origin worker index $i$,
               gradient partition $g_{j,p}$, learning rate $\eta$
13     $w_{j,p} \leftarrow w_{j,p} + \eta \cdot g_{j,p}$
14     $s_{j,i} \leftarrow s_{j,i} + 1$
```

idation accuracy has not improved after a fixed number of evaluation rounds.

The `updatePartialModel` function is executed when an accumulated gradient partition is received by a worker. It updates the local model $w_{j,p}$ asynchronously (line 13).

3.2 Number of gradient partitions

The number of gradient partitions p impacts the statistical efficiency of partial gradient exchange. There is a trade-off: when p is small, a worker exchanges large gradient partitions, synchronising local models more quickly but requiring more network bandwidth; when p is large, a worker uses less bandwidth but requires more synchronisation rounds to receive a full mini-batch gradient update.

The best choice of p therefore depends on the available network bandwidth, and workers can use a cost model to select p when training begins: let m be the local model size, and n the number of workers, the amount of data to send the full gradient update to all workers is $m(n{-}1)$. With partial gradient exchange, the amount becomes $\frac{m}{p}(n{-}1)$ as only one partition is sent to each worker.

Assuming a rate γ at which workers compute new gradient partitions, which is profiled during system start-up, partial gradient exchange requires a bandwidth usage of $\frac{\gamma m(n-1)}{p}$ to be sustainable, i.e. have a constant transmission delay. Given an available bandwidth B at each worker (e.g. 1 Gbps), and assuming full-bisection bandwidth, the workers thus select the partition number p as:

$$p = \left\lceil \frac{\gamma m(n-1)}{B} \right\rceil \tag{3}$$

3.3 Bounding staleness

The gradients computed by each worker may use weights from previous mini-batches, which introduces *staleness* [39]. This staleness has two sources: (i) a simple delay due to the asynchronous updates to the local models [3] because a worker computes new gradients without receiving updates from all the other workers; and (ii) a distributed aggregated delay [2] because a worker only completes a mini-batch after it has received all p gradient partitions, requiring multiple synchronisation rounds.

To guarantee convergence, partial gradient exchange thus imposes a staleness bound, analogous to the stale synchronous parallel (SSP) model [7]: it limits the generation of new local gradients when a worker has advanced in the computation further than τ compared to all other workers (line 3 in Alg. 1). To do so, each worker j maintains multiple *staleness clocks* $s_{j,n}$, one for each other worker n, and a *local staleness clock* c_j. The local clock is incremented after each produced gradient (line 11); the other workers' staleness clocks are incremented when partial updates are received (line 14).

As there is no global model state in partial gradient exchange, for workers to check the staleness bound and identify the least progressed worker, they must maintain the clock information of other worker to compute the clock differential. The staleness bound check also depends on the number of partitions p because p synchronisation rounds are necessary to fully propagate a model. Therefore the used staleness bound is $p + \tau$ (line 3). Since updates are incorporated into the local model as soon as they are available, partial gradient exchange reduces empirical staleness, similar to the eager stale synchronous parallel model [11], leading to faster convergence.

4. Ako Architecture

We describe the architecture and implementation of Ako, a decentralised DNN system that uses partial gradient exchange for synchronisation. To combine parallelism for model training with low communication latency for synchronisation, the architecture of an Ako worker follows a stateful distributed dataflow model [14]: as shown in Fig. 7, execution is broken into a series of short, data-parallel *tasks*. Tasks can update in-memory state and exchange data with each other, and also over the network.

We first summarise Ako's design goals (§4.1) and then give implementation details (§4.2).

4.1 Design goals

(1) Full utilisation of CPU and network resources. We want each worker to fully utilise all CPU cores for training the local model (for highest hardware efficiency), while also saturating the available network bandwidth for synchronisation (for highest statistical efficiency). Fig. 7 shows how workers decouple *compute tasks* that train the local model

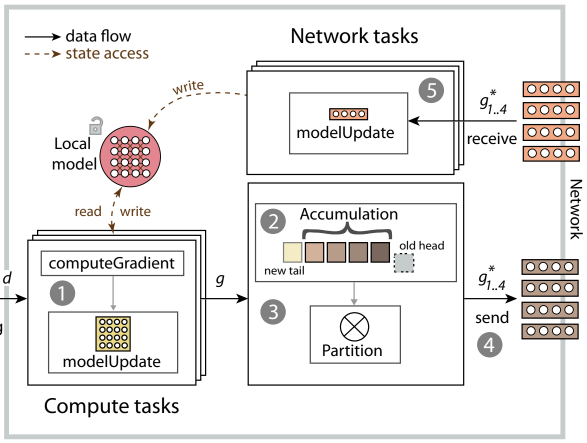


Figure 7: Architecture of an Ako worker

from *network tasks* that exchange gradients, thus utilising all resources.

(2) Low model contention. Both compute and network tasks must access the local model without contention. Workers represent the model as a list of matrices, each corresponding to a layer. Tasks can therefore modify distinct components independently.

(3) Low-latency synchronisation. Since the statistical efficiency depends on the latency of the gradient exchange, workers must exchange gradients with low delay. For this, workers use parallel network tasks to prepare and transmit updates as soon as they are generated.

(4) Support for complex processing pipelines. Real-world DNN systems include multiple pre- and post-processing steps, such as image resizing or model validation. Ako's dataflow-based architecture means that it is easy to extend with custom processing tasks.

4.2 Architecture and implementation

Next we describe the implementation of an Ako worker. As shown by the numbered steps in Fig. 7, a worker uses a pool of compute tasks to (1) compute gradients, and a pool of network tasks to (2) accumulate gradients, (3) partition gradients, (4) send gradients to other workers, and (5) receive gradients from other workers.

(1) Gradient computation. Compute tasks train the model in parallel. Each compute task has exclusive access to a partition of the input data as well as lock-free access to the local model.

The gradients generated by different parallel compute tasks must be aggregated at the end of a mini-batch in order to update the local model. After aggregating its generated gradient, a compute task checks if the mini-batch is finished, and if so, updates the model. Note that this occurs concurrently with other compute tasks reading the model. While such lock-free concurrent access leads to inconsistencies, it does not degrade statistical efficiency significantly [29].

Workers control the staleness across all distributed model replicas in a decentralised fashion. Each worker maintains a *staleness counter* that represents the difference in the gradient production of all workers: the counter is increased for each locally-generated gradient and decreased for each partial gradient update received from other workers. To control staleness, a compute task does not generate new gradients when its staleness counter is higher than the staleness threshold τ (see §3.3).

(2) Gradient accumulation. The computed gradients at the end of a mini-batch are accumulated by a pool of network tasks (see §3.1). Each worker maintains (i) the accumulated gradient from the previous round and (ii) an *accumulation queue*, which stores the last p generated gradients ordered by their creation round.

Every time a gradient is produced, a network task updates the accumulated gradient by adding the new generated gradient to the accumulated gradient and subtracting the old ${}^{(t-p)}g_j$ gradient at the head of the accumulation queue. The queue is then updated, removing its head and adding the new gradient to its tail.

(3) Gradient partitioning. Before the accumulated gradients are sent over the network, a network task partitions them using range-partitioning, with the position index in the internal representation of the convolutional kernels and fully-connected layer neurons as the partitioning key (see §3.1). The number of partitions is calculated according to the cost model from §3.2.

(4) Gradient sending. After that, a network task sends the gradient partitions, tagged by the partitioning range, to other workers. Each worker has a unique identifier, and the partitions are sent round-robin. After the number of synchronisation rounds equals the number of partitions, complete gradients have been received by all workers.

(5) Gradient receiving. Concurrently, workers receive gradient partitions from other workers, which must be merged to achieve convergence. Network tasks apply the gradients immediately to the corresponding parts of the local model without locking. Given that updates are applied at a fine granularity, tasks are likely to update different parts of the DNN model, which reduces the probability of lost updates.

4.3 Fault tolerance

When a worker fails, it loses its partial model, the *staleness counter* and the contents of the *accumulation queue*. In addition, since it can no longer contribute gradients to the other workers, the failed worker can disrupt the other workers' *staleness counters* if it takes time to recover. Next we describe how Ako handles worker failures.

Ako's workers rely on *checkpointing* to save their partial models and the staleness counters, similar to SEEP [14] or TensorFlow [1]. Gradient exchanges that have not yet been applied before the failure, as well as the content of the accumulation queue can be safely discarded due to the stochastic nature of the training process [13]. This simplifies failure recovery at the cost of additional iterations to achieve convergence. Finally, SEEP's master node also notifies workers of failures so that failed workers are removed from the staleness counters. Entries to the counters are re-added when workers recover.

5. Evaluation

The goals of our experimental evaluation are (i) to explore Ako's scalability and convergence compared to parameter server architectures (§5.2); (ii) to observe its statistical (§5.3) and hardware efficiency (§5.4); (iii) to explain the efficiency results by examining Ako's resource utilisation (§5.5); and (iv) to investigate the impact of gradient partitions (§5.6) and accumulation (§5.7) in partial gradient exchange on training performance.

5.1 Experimental set-up

Datasets and DNNs. We train DNN models on two well-known datasets for visual classification: (i) MNIST [21] contains 28×28 pixel grey-scale images of handwritten digits with 10 classes. The dataset has 60,000 training images, and 10,000 images for testing; (ii) ImageNet [31] has more than 14 million high-resolution images divided into 1000 classes, each with a ground truth label. We randomly select 100 classes to obtain a subset of approximately 120,000 images as this reduces the convergence time in experiments.

We train DNNs with 3 convolutional (interleaved with max-pooling) and 2 fully-connected layers: for MNIST, we use 10/20/100 convolutional kernels with 200 neurons, while the ImageNet model consists of 32/64/256 kernels and 800 neurons. Prior to training, the datasets are partitioned evenly across the workers. The model parameters are initialised using warm-start [12].

System comparisons. We compare (i) Ako with partial gradient exchange to (ii) an architecture with distributed parameter servers (PS) and (iii) a decentralised architecture with all-to-all communication between workers (All-to-All). To be comparable, all approaches are implemented on top of the SEEP stateful distributed dataflow platform [14] with the same optimisations. To put the absolute performance of Ako into perspective, we also compare against a CPU-based distributed TensorFlow (TF) and Singa (SG) deployments on the same hardware.

For Ako and All-to-All, all machines in the cluster execute workers. PS support an arbitrary number of machines to act as parameter servers, each maintaining a model partition that asynchronously synchronises with workers. We explore different allocations of workers and parameter servers: we denote a deployment with w workers and p parameter servers as PS[w+p], and mark the best configuration, as determined empirically through exhaustive search, with an asterisk (PS*[w+p]).

For PS, we also consider a co-located deployment [44] in which each worker executes a parameter server (PS[w+w]). As there are two processes sharing memory on each machine, we manually choose an allocation that yields the best training performance.

For TF and SG, we use a parameter server architecture to train DNNs with the asynchronous Downpour algorithm [12]. The global model parameters in TF are represented as a set of TF *variables*, i.e. persistent mutable tensors, which can be assigned to different nodes to scale the parameter servers. In general, a DNN layer is defined as one variable in the TensorFlow computation graph [36–38], and, at runtime, variables are assigned to parameter servers using a round-robin strategy [15].

Similar to prior work [9, 11, 19], we empirically set the staleness bound τ according to the used dataset and DNN model. As a heuristic, we increase τ proportionally to the number of used workers.

Performance metrics. We validate the DNN models based on the top-1 accuracy with the corresponding validation data. We measure the model training performance in terms of convergence and quantify training progress based on the validation accuracy over time. In addition, we collect the number of epochs required to achieve a predefined accuracy goal for assessing the statistical efficiency of different approaches.

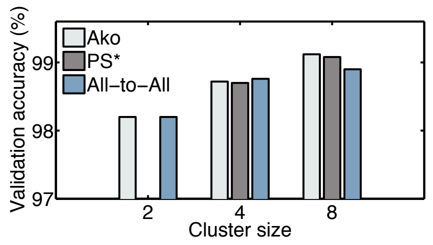
Cluster hardware. Based on the workload, we use two clusters with different sizes and hardware capabilities: (i) we train with the MNIST dataset and conduct the hardware utilisation study on a private 16-machine cluster with 4-core Intel Xeon E3-1220 CPUs at 3.1 Ghz with 8 GB of RAM and 1 Gbps Ethernet; (ii) for ImageNet, we use a 64-machine Amazon EC2 cluster with "m4.xlarge" Intel Xeon instances, each with 4 vCPU cores at 2.4 Ghz and 16 GB of RAM.

5.2 What is the convergence and scalability?

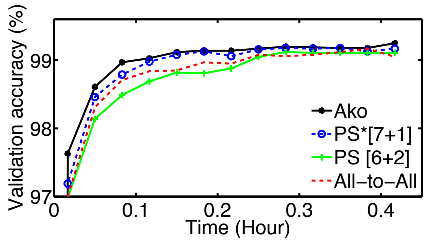
We evaluate the convergence speed of Ako in comparison to the other approaches on both the MNIST and ImageNet datasets. We also explore the scalability by training each of the models with different cluster sizes.

MNIST. We train a DNN for the MNIST dataset and vary the cluster size between 2, 4 and 8 machines. We collect the validation accuracy over 20 minutes of training. For the PS approach on 4 machines, we use a PS*[3+1] deployment; on 8 machines, we consider PS*[7+1] and PS[6+2]. We pick the best configuration to compare to the other approaches.

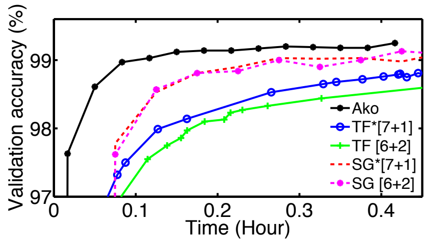
After 10 minutes of training, Fig. 8(a) shows that Ako achieves a similar convergence to PS* and slightly better convergence than All-to-All with 8 machines. Fig. 8(b) shows the convergence over time on the 8-machine cluster across the three approaches (with different PS allocations). The plot confirms the results from Fig. 8(a): Ako exhibits similar con-



(a) Accuracy after 10 minutes of training

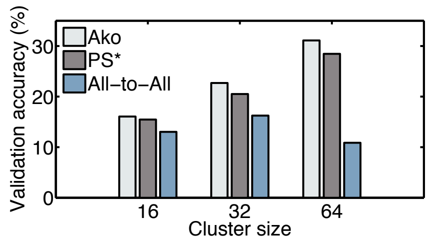


(b) Convergence with 8 machines

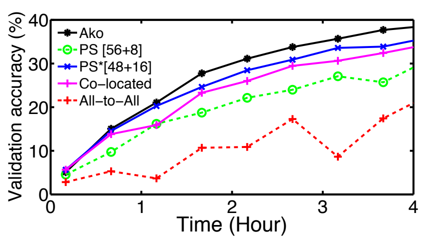


(c) Comparison with TensorFlow and Singa

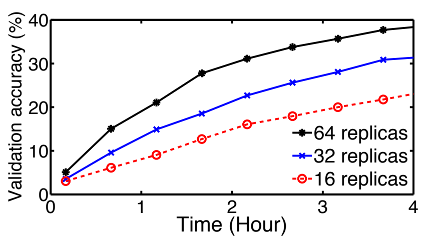
Figure 8: Model convergence with different cluster sizes (MNIST)



(a) Accuracy after 2-hours of training



(b) Convergence with 64 machines



(c) Convergence with different cluster sizes

Figure 9: Model convergence with different cluster sizes (ImageNet)

Dataset	Accuracy	TensorFlow	All-to-All	Ako
MNIST	99%	> 20 min	14 min	7 min
ImageNet	30%	3.3 h	> 4 h	1.5 h

Table 1: Time to reach target validation accuracy

vergence to the best allocation for PS, which is PS*[7+1], and converges faster than the All-to-All approach. Synchronisation in Ako and PS does not require as much communication as for All-to-All, whose convergence is affected by the higher synchronisation delay.

With 8 machines, we also compare Ako to distributed TensorFlow and Singa when training under the same DNN workload. We deploy two parameter server configurations for both TensorFlow and Singa, TF*[7+1], TF[6+2], SG*[7+1] and SG[6+2], with training performed using asynchronous Downpour [12].

Fig. 8(c) shows that Ako converges faster than both configurations of TF and SG. Also summarised in Table 1, it takes Ako 7 minutes and TF* more than 20 minutes to achieve a target of validation accuracy of 99%. We speculate that the difference in convergence speeds between Ako and the two systems results from the synchronisation under Downpour, which allows workers to process an entire mini-batch before updating the global model.

ImageNet. Next we train a more complex DNN with the ImageNet dataset on 16, 32 and 64 machines. We collect the validation accuracy after 2 hours and also observe convergence over time.

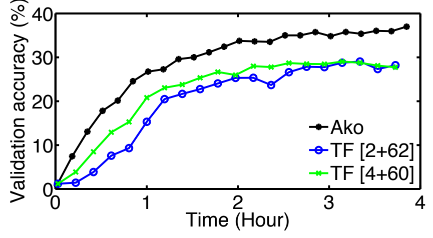
Fig. 9(a) shows that, after 2 hours of training, Ako achieves a higher validation accuracy than PS* on 32 and 64 machines. In contrast, the All-to-All approach converges more slowly due to its high synchronisation cost. With more machines, convergence improves for Ako and PS but not for All-to-All. PS claims a larger fraction of the machines for parameter servers, whereas Ako makes use of all machines as workers, speeding up convergence.

Any Ako worker can be used for validation as differences between them are negligible. After training on the 16-machine cluster with a fixed number of epochs, the average accuracy across workers is 20.2%, with a variance of 0.15%.

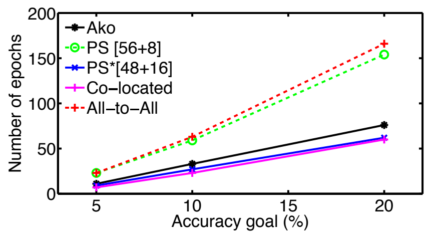
Fig. 9(b) explores convergence over time on 64 machines with different resource configurations. Overall, Ako requires less training time than PS* to reach the same accuracy after the early phase of learning: Ako has 64 workers to finish each epoch, but PS* has only 48 workers as the other machines are used as parameter servers. The difference in the number of iterations between the two strategies grows as training continues, which leads to different convergence rates.

Other configurations for PS, including (i) too few parameter servers (PS[56+8]) and (ii) co-located parameter servers (PS[64+64]), exhibit even worse convergence: the network contention at the parameter servers in PS[56+8] prohibits a tight synchronisation among workers; gathering all the global model partitions across 64 servers in PS[64+64] also causes additional delay.

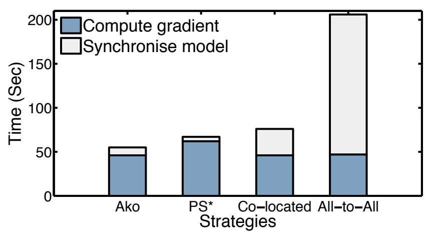
Fig. 9(c) shows the convergence over time for Ako with different cluster sizes. Given that partial gradient exchange selects a number of gradient partitions that keeps the communication cost constant, Ako scales gracefully. With 64 machines, each worker partitions gradients into 20 partitions before sending them to the other workers. The communication cost thus remains constant, avoiding bottlenecks that would increase the synchronisation delay.



(a) Comparison with TensorFlow

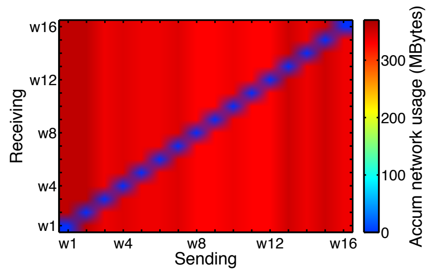


(b) Epoch number for given accuracy goal

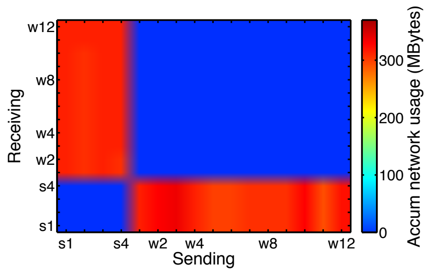


(c) Break-down of epoch time

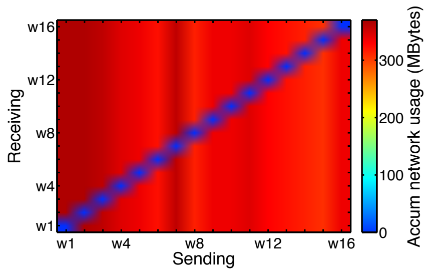
Figure 10: Experiment with 64 replicas (ImageNet)



(a) Ako



(b) PS



(c) All-to-All

Figure 11: Average network usage with 16 machines (ImageNet)

Next we compare Ako with distributed TensorFlow on the same 64-machine cluster training ImageNet. We consider two TensorFlow configurations, TF[62+2] and TF[60+4], because our DNN has five layers, each of which is defined as a TensorFlow variable in the graph.

Fig. 10(a) shows that Ako exhibits competitive convergence from the beginning of training. To get to a validation accuracy of 30%, Ako takes 1.5 hours while TF takes more than 3 hours (see Table 1). When converged, Ako also achieves a higher validation accuracy. Our experiment demonstrates that the performance of Ako is en par with that of a standard deployment of a state-of-the-art distributed deep learning platform.

5.3 What is the statistical efficiency?

Next we assess how progressive epochs under partial gradient exchange contribute to the convergence for ImageNet on 64 machines. We define three validation accuracy goals (5%, 10% and 20%), and observe the number of epochs required to achieve them.

Fig. 10(b) shows that the PS approach requires the fewest passes over the training data for a given accuracy when the number of server nodes is high enough for the centralised parameter synchronisation: 16 servers for 48 workers in PS*[48+16], and 64 servers for 64 workers in the co-location case (PS[64+64]).

Ako requires extra epochs for the same accuracy, making it less statistically efficient than PS—gradients are partitioned before exchange, which means that workers receive incomplete gradients but with low latency; only after multiple rounds, they obtain complete gradients.

With too few parameter servers (PS[56+8]), the efficiency of the parameter server approach declines and becomes the same as that of All-to-All, which suffers from diverging models due to insufficient synchronisation.

5.4 What is the hardware efficiency?

Given that Ako trains the DNN model for ImageNet faster than PS*, the best distributed parameter server configuration, while having lower statistical efficiency, it must have higher hardware efficiency. To confirm this, we collect the time per epoch across the approaches together with their break-down into (i) gradient computation and (ii) model synchronisation time.

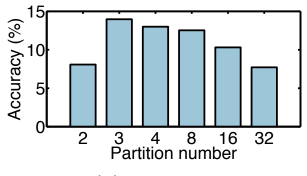
Fig. 10(c) shows that Ako has a shorter epoch time than PS, regardless of its configuration; as expected, the decentralised All-to-All approach takes the longest time.

All approaches except All-to-All spend most of their epoch time on gradient computation. Compared to the other approaches, the optimal PS*[48+16] configuration requires longer to do this because it has only 48 workers for processing. In fact, its gradient computation time alone exceeds Ako's overall epoch time.

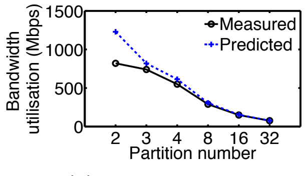
Comparing PS* to the co-located set-up (PS[64+64]), the later takes more time for the model synchronisation. This is due to the fact that its workers must synchronise with all 64 machines to obtain a new version of the model, prolonging the epoch time compared to PS*[48+16].

5.5 What is the resource utilisation?

We now investigate how Ako utilises the CPU resources and network bandwidth of the cluster compared to the other approaches. We deploy the ImageNet DNN model with Ako on



(a) Accuracy



(b) Network usage

Figure 12: Effect of gradient partitions

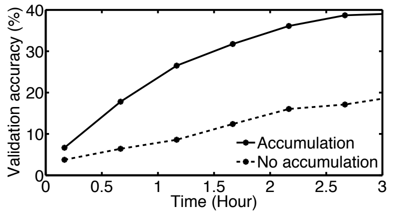


Figure 13: Benefit of gradient accumulation

our 16-machine cluster, and compare to the best PS*[12+4] configuration and the All-to-All approach. We measure the average CPU utilisation across the cluster and the accumulated network bandwidth usage between all machine pairs.

The average CPU usage of the workers for Ako, PS*[12+4] and All-to-All is 87%, 84% and 85%, respectively; the distributed parameter servers have an average utilisation of 17%, underutilising their CPU resources. Updating the global model parameters does not require as much CPU resources as the matrix multiplications and convolutions performed by the workers.

Fig. 11 shows the accumulated network usage in MBs. For Ako, the network usage between all machine pairs is high, fully saturating the network while still achieving a low synchronisation delay. All-to-All also fully saturates the network, but suffers from a high delay due to the introduced queuing. PS* has a much lower network usage, with substantial network bandwidth between workers left unused. The peak network usage of Ako is lower than that of All-to-All because it partitions gradients before exchange according to its cost model. Ako's workers exchange gradients as frequently as possible while respecting the capacity limit of the network. In a cloud environment in which the network bandwidth depends on the type of VM, the selected number of gradient partitions will be such as to maximise the usage of the available bandwidth.

While gradient partitioning avoids network contention, the gradient accumulation queue must be maintained to ensure the complete propagation of gradients. Although the size of the queue leads to higher memory usage and depends on the model size, the additional memory usage is typically acceptable: many well-known models such as GoogleNet [35] and AlexNet [20] are on the order of MBs because they do not require high numeric precision [16].

5.6 What is the effect of gradient partitions?

Next we investigate the effectiveness of how Ako chooses the number of gradient partitions according to its cost model (§3.2). We execute the ImageNet DNN model with Ako on our 16-machine cluster across different partition numbers, and measure the validation accuracy and the workers' bandwidth usage.

Fig. 12(a) shows that 3 gradient partitions yield the highest accuracy, which is the partition number predicted by the cost model. With only 2 partions, the partition size becomes larger. This is beneficial for the learning progress because it improves statistical efficiency—it takes fewer rounds to exchange the complete gradient. However, the gradient exchange has higher latency due to network contention, which increases the divergence of the local models during training. Having more than 3 partitions also reduces the accuracy because the information exchange between the workers becomes less effective, which reduces the statistical efficiency.

Fig. 12(b) compares the measured and predicted network usage according to the cost model. For almost all partition numbers, the measured bandwidth usage is close to the predicted one. The difference is largest with 2 partitions because the predicted usage by the model goes beyond the 1 Gbps bandwidth available in the cluster.

5.7 What is the benefit of gradient accumulation?

In partial gradient exchange, the accumulation of gradient updates ensures the eventual completeness of gradients sent to workers. We conduct an experiment to evaluate how this improves training quality on the ImageNet DNN model with 8 machines. We measure the validation accuracy of Ako with and without gradient accumulation.

Fig. 13 shows the convergence over time. Without accumulation, the resulting model exhibits worse accuracy over time with slower convergence: the best accuracy on validation is only around 20%, while gradient accumulation improves this to nearly 40%. If workers do not receive complete gradient updates, the statistical efficiency of the system is reduced.

6. Related work

DNN systems with parameter servers. Scalable deep neural network (DNN) systems, such as DistBelief [12], TensorFlow [1], Project Adam [5], Singa [27, 42, 43], Poseidon [48] and SparkNet [25], speed up the distributed training of DNN models using parameter servers. To avoid network bottleneck while synchronising multiple workers with a centralised global model, parameter servers are scaled to gain higher accumulative bandwidth [24].

TensorFlow [1] expresses DNNs as dataflow graphs that train under a parameter server architecture. For distributed parameter servers, TensorFlow uses a round-robin strategy that assigns different DNN layers to parameter server nodes. This assignment can lead to imbalances in the computational load and network utilisation among servers because the model is partitioned at a relatively coarse granularity. In addition, the scaling of parameter servers can be limited by the number of mutable tensors (i.e. variables) in graph.

While scaling parameter servers relieves network contention, there are further techniques to reduce network communication, including data compression [1, 24] and filtering [24]. In Bösen [44], messages are prioritised—ones that lead to more significant model progress are transmitted first. In exchange for more efficient network usage, such techniques, however, increase CPU utilisation.

Poseidon [48] uses Bösen's parameter server architecture with a hybrid synchronisation model: workers establish direct connections to offload network communication from the parameter server. While Poseidon reduces the size of model updates by exchanging sufficient factors [46] for dense fully-connected layers, its communication cost can grow quadratically with respect to the number of nodes. In general, distributed parameter servers can adopt a hybrid architecture to increase the accumulative bandwidth at the server side, but this further complicates resource allocation decisions.

Singa [27, 42, 43] is a deep learning platform that supports multiple partitioning and synchronisation schemes, thus enabling users to easily use different training regimes at scale. Through the concept of *logical groups* of execution units, Singa supports complex cluster topologies, e.g. one with multiple server groups. This flexibility allows users to tune configurations for given problems and the available cluster resources, but requires them to make configuration decisions empirically. Ako is a step towards removing some of this configuration complexity.

To address resource allocation issues in DNN systems trained with parameter servers, Yan et al. [47] propose to model system performance and then search the possible configuration space. This allows users to estimate system scalability with different parallelisation and synchronisation strategies. It is unclear, however, how bounded staleness can be included in the modelling and affects its accuracy. In addition, such an offline approach may struggle to account for dynamic effects such as stragglers, leading to inaccurate predictions.

Rather than assigning servers and workers to different machine groups, Bösen [44] uses collocation, i.e. each node contains both servers and workers. This type of allocation requires gradients to be aggregated to different model partitions across nodes before redistributing the updated model parts back to all workers. Although this approach maximises the accumulative network bandwidth for parameter servers, the two-step all-reduce procedure adds substantial delays when the system scales, even under bounded staleness.

DNN systems without parameter servers. Several DNN systems make use of decentralised training to simplify the deployment in distributed environments while maximising data parallelism. Since direct communication between nodes can grow quadratically when adding more workers, this requires scalable synchronisation strategies.

Wang et al. [41] propose a decentralised protocol for parameter sharing with custom synchronisation topologies. Network contention is reduced by having a subset of workers relay gradients to the rest of the topology. As a result, model updates propagate more slowly, resulting in a higher convergence time.

Similarly, in MALT [23], workers exchange gradients with a subset of workers selected by a Halton sequence. Due to the synchronisation delay, this suffers from slower convergence, especially for complex neural network models. In contrast, workers in Ako always exchange partial model updates with all others.

In CNTK [32, 33], gradients are aggregated across all nodes, followed by model redistribution. To reduce the bandwidth requirement for densely-connected speech DNNs, gradient values are quantised aggressively before being exchanged over the network. To ensure model convergence, the resulting quantisation error must be added to the gradients in the following rounds, compensating for the inaccuracy. By representing gradients as 1-bit values, the work shows that there is little negative impact on model accuracy as the quantisation error feedback is treated as type of delayed update. In a similar fashion, delayed updates in Ako result from gradient partitioning.

Mariana [50] synchronises multiple GPGPU workers in a linear topology. By sending gradients and model updates synchronously via adjacent workers, there is only minimal network contention, but this approach increases synchronisation delay due to the larger hop count.

Deep Image [45] uses a custom-built supercomputer with GPGPUs for DNN training. Workers are responsible for individual model partitions, and exchange updates through a butterfly network topology. In contrast, Ako does not partition the model, but instead performs full gradient synchronisation over multiple rounds.

7. Conclusions

To achieve the best performance, distributed DNN systems must fully utilise the cluster CPUs for model training and the cluster bandwidth for model synchronisation. Today's DNN architectures, however, rely on distributed parameter servers, which makes it hard for users to allocate cluster resources to them in an optimal way, i.e. without introducing compute or network bottlenecks.

We described Ako, a decentralised DNN system that does not require parameter servers, yet scales to large deployments with competitive performance. Ako achieves this through a new scalable synchronisation approach, *partial gradient exchange*, in which gradient updates propagate to all workers but only using constant bandwidth by sending partitions. We showed experimentally that an implementation of Ako with asynchronous compute and network tasks has better performance on a fixed-sized cluster than one with parameter servers.

References

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale Machine Learning on Heterogeneous Systems. In *White paper,* `http://tensorflow.org/`, 2015.

[2] A. Agarwal and J. C. Duchi. Distributed Delayed Stochastic Optimization. In *Advances in Neural Information Processing Systems (NIPS)*, 2011.

[3] A. Ahmed, M. Aly, J. Gonzalez, S. Narayanamurthy, and A. J. Smola. Scalable Inference in Latent Variable Models. In *ACM International Conference on Web Search and Data Mining (WSDM)*, 2012.

[4] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. In *Arxiv preprint arXiv:1512.01274*, 2015.

[5] T. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman. Project Adam: Building an Efficient and Scalable Deep Learning Training System. In *USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2014.

[6] K. Cho, B. van Merrienboer, Ç. Gülçehre, F. Bougares, H. Schwenk, and Y. Bengio. Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. In *Arxiv preprint arXiv:1406.1078*, 2014.

[7] J. Cipar, Q. Ho, J. K. Kim, S. Lee, G. R. Ganger, G. Gibson, K. Keeton, and E. Xing. Solving the Straggler Problem with Bounded Staleness. In *14th Workshop on Hot Topics in Operating Systems (HotOS)*, 2013.

[8] A. Coates, B. Huval, T. Wang, D. Wu, B. Catanzaro, and N. Andrew. Deep Learning with COTS HPC Systems. In *International Conference on Machine Learning (ICML)*, 2013.

[9] H. Cui, J. Cipar, Q. Ho, J. K. Kim, S. Lee, A. Kumar, J. Wei, W. Dai, G. R. Ganger, P. B. Gibbons, G. A. Gibson, and E. P. Xing. Exploiting Bounded Staleness to Speed Up Big Data Analytics. In *USENIX Annual Technical Conference (ATC)*, 2014.

[10] Y. L. Cun, B. Boser, J. S. Denker, R. E. Howard, W. Habbard, L. D. Jackel, and D. Henderson. Handwritten Digit Recognition with a Back-propagation Network. In *Advances in Neural Information Processing Systems (NIPS)*, 1990.

[11] W. Dai, A. Kumar, J. Wei, Q. Ho, G. Gibson, and E. Xing. High-Performance Distributed ML at Scale through Parameter Server Consistency Models. In *Conference on Artificial Intelligence (AAAI)*, 2015.

[12] J. Dean, G. S. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, and A. Y. Ng. Large Scale Distributed Deep Networks. In *Advances in Neural Information Processing Systems (NIPS)*, 2012.

[13] S. Dudoladov, C. Xu, S. Schelter, A. Katsifodimos, S. Ewen, K. Tzoumas, and V. Markl. Optimistic Recovery for Iterative Dataflows in Action. In *ACM International Conference Management of Data (SIGMOD)*, 2015.

[14] R. C. Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch. Making State Explicit for Imperative Big Data Processing. In *USENIX Annual Technical Conference (ATC)*, 2014.

[15] Google. TensorFlow: Assign variables to parameter servers. `https://www.tensorflow.org/versions/r0.8/api_docs/python/train.html#replica_device_setter`, 2015.

[16] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan. Deep Learning with Limited Numerical Precision. In *Arxiv preprint arXiv:1502.02551*, 2015.

[17] G. Hinton, L. Deng, D. Yu, G. Dahl, A. rahman Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. Sainath, and B. Kingsbury. Deep Neural Networks for Acoustic Modeling in Speech Recognition. *Signal Processing Magazine*, 2012.

[18] G. E. Hinton, S. Osindero, and Y.-W. Teh. A Fast Learning Algorithm for Deep Belief Nets. *Neural Computation*, 18(7): 1527–1554, 2006.

[19] 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. In *Advances in Neural Information Processing Systems (NIPS)*, 2013.

[20] A. Krizhevsky, I. Sutskever, and G. E. Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems (NIPS)*, 2012.

[21] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-Based Learning Applied to Document Recognition. In *Intelligent Signal Processing*, 2001.

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

[23] H. Li, A. Kadav, E. Kruus, and C. Ungureanu. MALT: Distributed Data-parallelism for Existing ML Applications. In *10th ACM European Conference on Computer Systems (EuroSys)*, 2015.

[24] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su. Scaling Distributed Machine Learning with the Parameter Server. In *USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2014.

[25] P. Moritz, R. Nishihara, I. Stoica, and M. I. Jordan. SparkNet: Training Deep Networks in Spark. In *Arxiv preprint arXiv:1511.06051*, 2015.

[26] A. Nedic, D. P. Bertsekas, and V. S. Borkar. Distributed Asynchronous Incremental Subgradient Methods. In *Studies in Computational Mathematics*, 2001.

[27] B. C. Ooi, K.-L. Tan, S. Wang, W. Wang, Q. Cai, G. Chen, J. Gao, Z. Luo, A. K. Tung, Y. Wang, Z. Xie, M. Zhang, and K. Zheng. SINGA: A Distributed Deep Learning Platform. In *ACM International Conference on Multimedia (MM)*, 2015.

[28] R. Power and J. Li. Piccolo: Building Fast, Distributed Programs with Partitioned Tables. In *USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2010.

[29] B. Recht, C. Re, S. Wright, and F. Niu. Hogwild: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent. In *Advances in Neural Information Processing Systems (NIPS)*, 2011.

[30] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning Representations by Back-propagating Errors. In *Cognitive Modeling*, 1986.

[31] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.

[32] F. Seide, H. Fu, J. Droppo, G. Li, and D. Yu. 1-bit Stochastic Gradient Descent and its Application to Data-parallel Distributed Training of Speech DNNs. In *INTERSPEECH*, 2014.

[33] F. Seide, H. Fu, J. Droppo, G. Li, and D. Yu. On Parallelizability of Stochastic Gradient Descent for Speech DNNs. In *Acoustics, Speech and Signal Processing (ICASSP)*, 2014.

[34] I. Sutskever, O. Vinyals, and Q. V. Le. Sequence to Sequence Learning with Neural Networks. In *Arxiv preprint arXiv:1409.3215*, 2014.

[35] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going Deeper with Convolutions. In *Arxiv preprint arXiv:1409.4842*, 2014.

[36] TensorFlow. Code example 1. `https://github.com/tensorflow/tensorflow/blob/r0.8/tensorflow/models/image/mnist/convolutional.py`, 2016.

[37] TensorFlow. Code example 2. `https://github.com/dsindex/tensorflow/blob/master/mlp_mnist_dist.py`, 2016.

[38] TensorFlow. Code example 3. `https://github.com/tensorflow/models/blob/master/autoencoder/autoencoder_models/Autoencoder.py`, 2016.

[39] D. Terry. Replicated Data Consistency Explained Through Baseball. In *Communications of the ACM*, 2013.

[40] L. G. Valiant. A Bridging Model for Parallel Computation. In *Communications of the ACM*, 1990.

[41] M. Wang, H. Zhou, M. Guo, and Z. Zhang. A Scalable and Topology Configurable Protocol for Distributed Parameter Synchronization. In *5th Asia-Pacific Workshop on Systems (APSys)*, 2014.

[42] W. Wang, G. Chen, A. T. T. Dinh, J. Gao, B. C. Ooi, K.-L. Tan, and S. Wang. SINGA: Putting Deep Learning in the Hands of Multimedia Users. In *ACM International Conference on Multimedia (MM)*, 2015.

[43] W. Wang, G. Chen, H. Chen, T. T. A. Dinh, J. Gao, B. C. Ooi, K. Tan, and S. Wang. Deep Learning At Scale and At Ease. In *Arxiv preprint arXiv:1603.07846*, 2016.

[44] J. Wei, W. Dai, A. Qiao, Q. Ho, H. Cui, G. R. Ganger, P. B. Gibbons, G. A. Gibson, and E. P. Xing. Managed Communication and Consistency for Fast Data-parallel Iterative Analytics. In *ACM Symposium on Cloud Computing (SoCC)*, 2015.

[45] R. Wu, S. Yan, Y. Shan, Q. Dang, and G. Sun. Deep Image: Scaling up Image Recognition. In *Arxiv preprint arXiv:1501.02876*, 2015.

[46] P. Xie, J. K. Kim, Y. Zhou, Q. Ho, A. Kumar, Y. Yu, and E. P. Xing. Distributed Machine Learning via Sufficient Factor Broadcasting. In *Arxiv preprint arXiv:1409.5705*, 2015.

[47] F. Yan, O. Ruwase, Y. He, and T. Chilimbi. Performance Modeling and Scalability Optimization of Distributed Deep Learning Systems. In *ACM International Conference on Knowledge Discovery and Data Mining (KDD)*, 2015.

[48] H. Zhang, Z. Hu, J. Wei, P. Xie, G. Kim, Q. Ho, and E. Xing. Poseidon: A System Architecture for Efficient GPU-based Deep Learning on Multiple Machines. In *Arxiv preprint arXiv:1512.06216*, 2015.

[49] M. Zinkevich, J. Langford, and A. J. Smola. Slow Learners are Fast. In *Advances in Neural Information Processing Systems (NIPS)*, 2009.

[50] Y. Zou, X. Jin, Y. Li, Z. Guo, E. Wang, and B. Xiao. Mariana: Tencent Deep Learning Platform and Its Applications. *Proc. VLDB Endow.*, 7(13):1772–1777, August 2014.