HPC16 4th Presentation

Yohei Tsuji

Tokyo institute of technology,

Dept. of Mathematical and Computing Science, Matsuoka Lab.

Selected Papers

- P. Watcharapichat, V. L. Morales, R. C. Fernandez, P. Pietzuch.
- ***Ako: Decentralised Deep Learning with Partial Gradient Exchange***.
- SoCC '16 Proceedings of the Seventh ACM Symposium on Cloud Computing.

Index

1. Introduction
2. Resource Allocation in DNN Systems
3. Partial Gradient Exchange
4. Ako Architecture
5. Evaluation
6. Related work
7. Conclusion

Index

§1 Introduction

- DNNs in distributed systems

- A common architecture for DNN systems takes advantage of data-parallelism which a set of *workers* train model replicas.
- By using *parameter servers*, model replicas are kept synchronised.

- DNN systems employing parameter server must balance the use of compute and network resources to achieve fastest model.
  - However, an optimal resource allocation depends on many factors, and users must decide it empirically, by trial-and-error approach.

§1 Introduction
- Described system

- Goal is to design a DNN system that always utilises the full CPU resources and network bandwidth of a cluster.

- Paper describe ***Ako***, a decentralised DNN system.
  - Homogeneous workers train model replicas without parameter server.
  - Synchronise directly with each other in a peer-to-peer fashion.

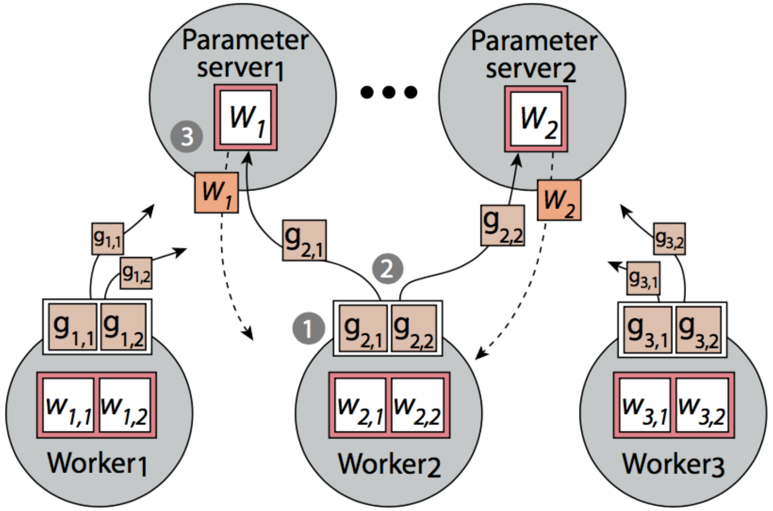
Index

1. Introduction
2. Resource Allocation in DNN Systems
3. Partial Gradient Exchange
4. Ako Architecture
5. Evaluation
6. Related work
7. Conclusion

§2 Resource Allocation in DNN Systems

- DNN systems with parameter servers

- A scalable approach for training DNNs is to use *parameter server*
  1. The training data is split across *worker*.
  2. Each worker calculate the gradient over its data partition.
  3. Worker sends the local gradient g to parameter servers.
  4. Parameter servers aggregate the gradients and update the global model W.
  5. And return the new model W to the workers.



For more detail about parameter server architecture, read [24, 2, 19, 26, 49].

§2 Resource Allocation in DNN Systems

- DNN systems with parameter servers

- To reach fastest time-to-convergence, DNN systems must achieve:
    1. High hardware efficiency,
        - Which is time to complete a single iteration.
    2. High statistical efficiency,
        - Which is the improvement in the model per iteration.

- There is a trade-off between these two aspects.
    - In practice, modern distributed DNN systems require such decision on resource allocation.

§2 Resource Allocation in DNN Systems

- Resource allocation problem

- The best allocation should result in fastest time-to-convergence.
  - However, the best allocation depends on many factors which make prediction difficult.

- This difficulty can be checked through some experiments.
  - Deployed a DNN system with parameter servers on 64-machines, training a model for **ImageNet** benchmark (explained later).

§2 Resource Allocation in DNN Systems

- Resource allocation problem

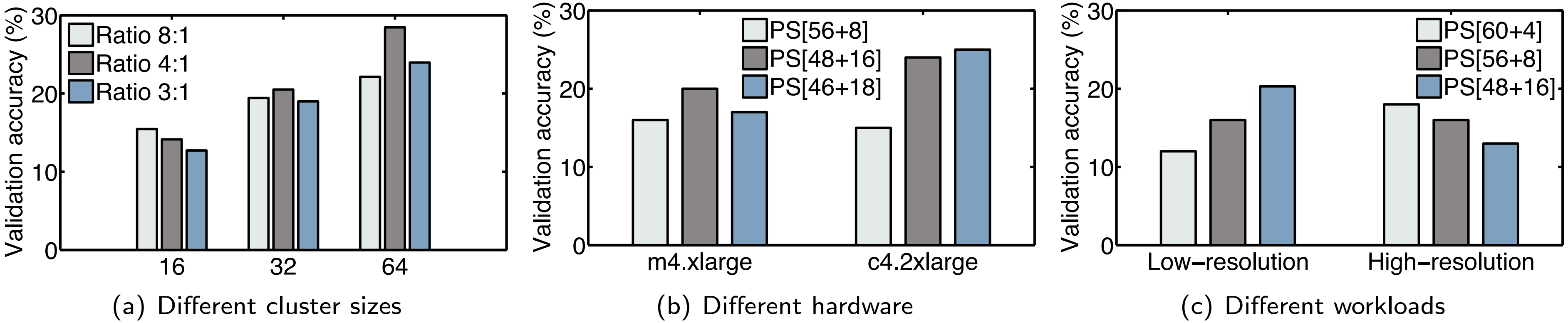
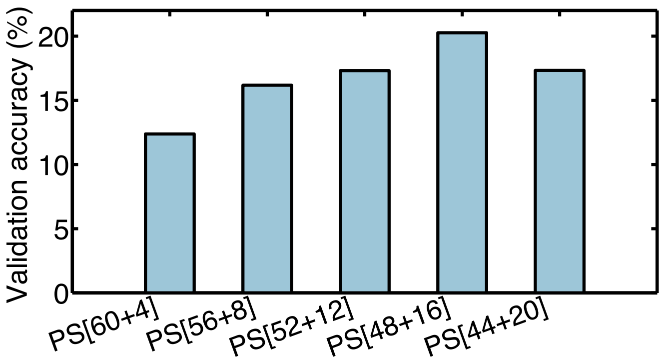


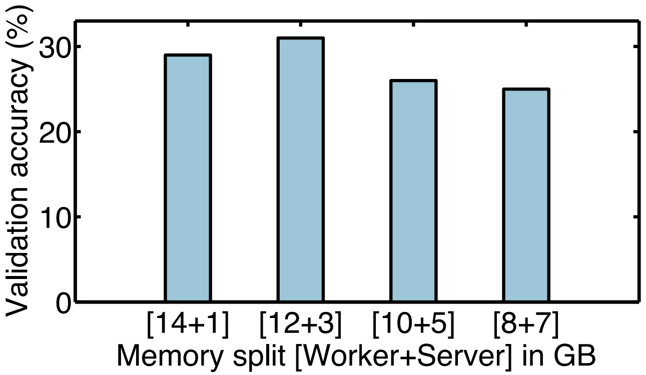
Figure 3: Effect of system and workload changes on best resource allocation

- Accuracy with different (a) **cluster size**, (b) **hardware**, and (c) **workloads**.
- In (b), comparing “m4.xlarge” and “c4.2xlarge” VMS on a 64-machine Amazon EC2 deployment.
- In (c), low-resolution is 100x100 pixels, and high-resolution is 200x200 pixels.

§2 Resource Allocation in DNN Systems

- Resource allocation problem





- Accuracy with different worker and parameter server **allocation** (left), and **memory allocation** in co-located parameter server (right).
  - Both are accuracy after one hour training.
- In co-located [44], worker and parameter server are located on same node.

Index

§3 Partial Gradient Exchange

- Adopting decentralised synchronisation scheme

- Instead of using parameter server, the author adopts a decentralised synchronisation scheme.
  - which workers communicate directly with each other, without intermediate nodes.

- Some decentralised solutions are…
  - All-to-All communication
  - Relaying updates
- However, these are not “good” as parameter server.

§3 Partial Gradient Exchange
- Partial gradient exchange algorithm

- A new decentralised synchronisation approach called ***partial gradient exchange***.
  - In this approach, worker sends only one partition to each other worker.

- For each worker, there are three steps which refer as ***synchronisation round***.
  - Calculating & accumulating local gradient
  - Partitioning local gradient
  - Sending local gradient

§3 Partial Gradient Exchange

- Partitioning gradients at synchronization round t

$$^{(t)}\mathbf{g}_j$$

Creates the local gradients from (a part of) data points in mini-batch.

$$\Downarrow$$

$$^{(t)}\mathbf{g}_j^*$$

Accumulates the gradient with previous-unsent local gradients.

$$^{(t)}\mathbf{g}_j^* \leftarrow {}^{(t)}\mathbf{g}_j + {}^{(t-1)}\mathbf{g}_j + {}^{(t-2)}\mathbf{g}_j + \cdots$$

To do so, worker needs to store previous local gradients some how.

$$\Downarrow$$

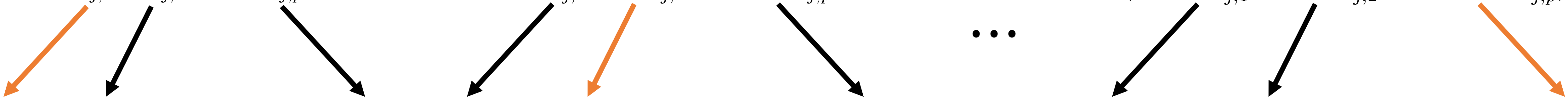
$$\left({}^{(t)}\mathbf{g}_{j,1}^*, {}^{(t)}\mathbf{g}_{j,2}^*, \cdots, {}^{(t)}\mathbf{g}_{j,p}^*\right)$$

Partitions the accumulated gradient into p disjoint ***gradient partitions***.

§3 Partial Gradient Exchange

- Sending gradients at synchronization round t

$$({}^{(t)}\mathbf{g}^*_{j,1}, {}^{(t)}\mathbf{g}^*_{j,2}, \cdots, {}^{(t)}\mathbf{g}^*_{j,p}) \quad ({}^{(t+1)}\mathbf{g}^*_{j,1}, {}^{(t+1)}\mathbf{g}^*_{j,2}, \cdots, {}^{(t+1)}\mathbf{g}^*_{j,p}) \quad \cdots \quad ({}^{(t+p-1)}\mathbf{g}^*_{j,1}, {}^{(t+p-1)}\mathbf{g}^*_{j,2}, \cdots, {}^{(t+p-1)}\mathbf{g}^*_{j,p})$$



Sending to other workers

- Sends each gradient partitions to other workers in round-robin fashion.
- It takes p synchronization rounds to send complete gradient which calculated at synchronization round t.

§3 Partial Gradient Exchange

- Accumulating gradients

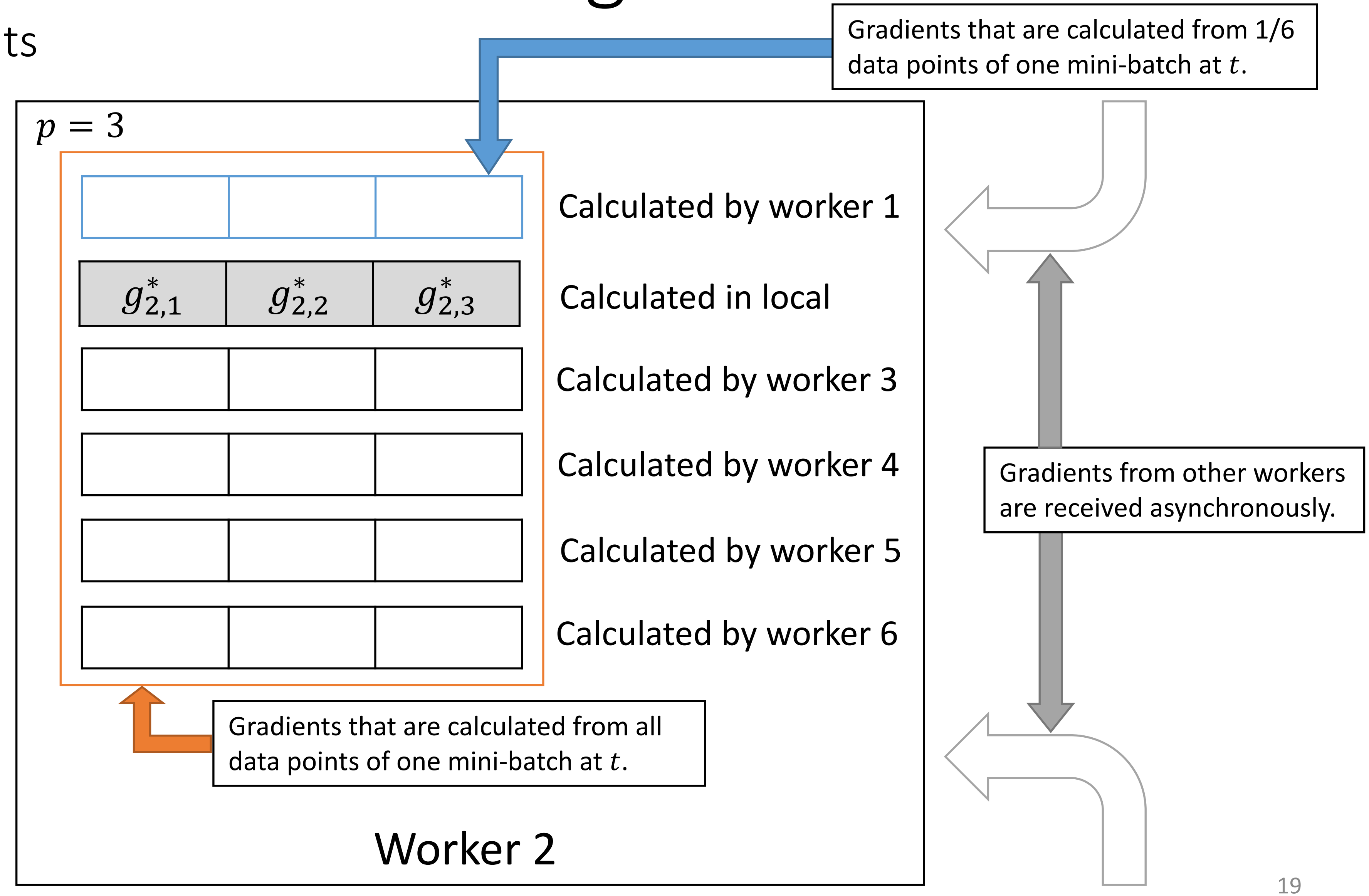
- According to the previous slide, only last p gradients are needed to be accumulated.
  - Thus, the relational expression will be:

$$\begin{cases} {}^{(1)}\mathbf{g}_j^* \leftarrow {}^{(1)}\mathbf{g}_j & \\ {}^{(t)}\mathbf{g}_j^* \leftarrow {}^{(t-1)}\mathbf{g}_j^* + {}^{(t)}\mathbf{g}_j & \text{if } 2 \leq t \leq p \\ {}^{(t)}\mathbf{g}_j^* \leftarrow {}^{(t-1)}\mathbf{g}_j^* + {}^{(t)}\mathbf{g}_j - {}^{(t-p)}\mathbf{g}_j & \text{if } t > p \end{cases}$$

  - Subtracting ${}^{(t-p)}\mathbf{g}_j$ is needed to avoid sending already-sent gradient partitions.
  - This improves the training quality when compared with no accumulation.

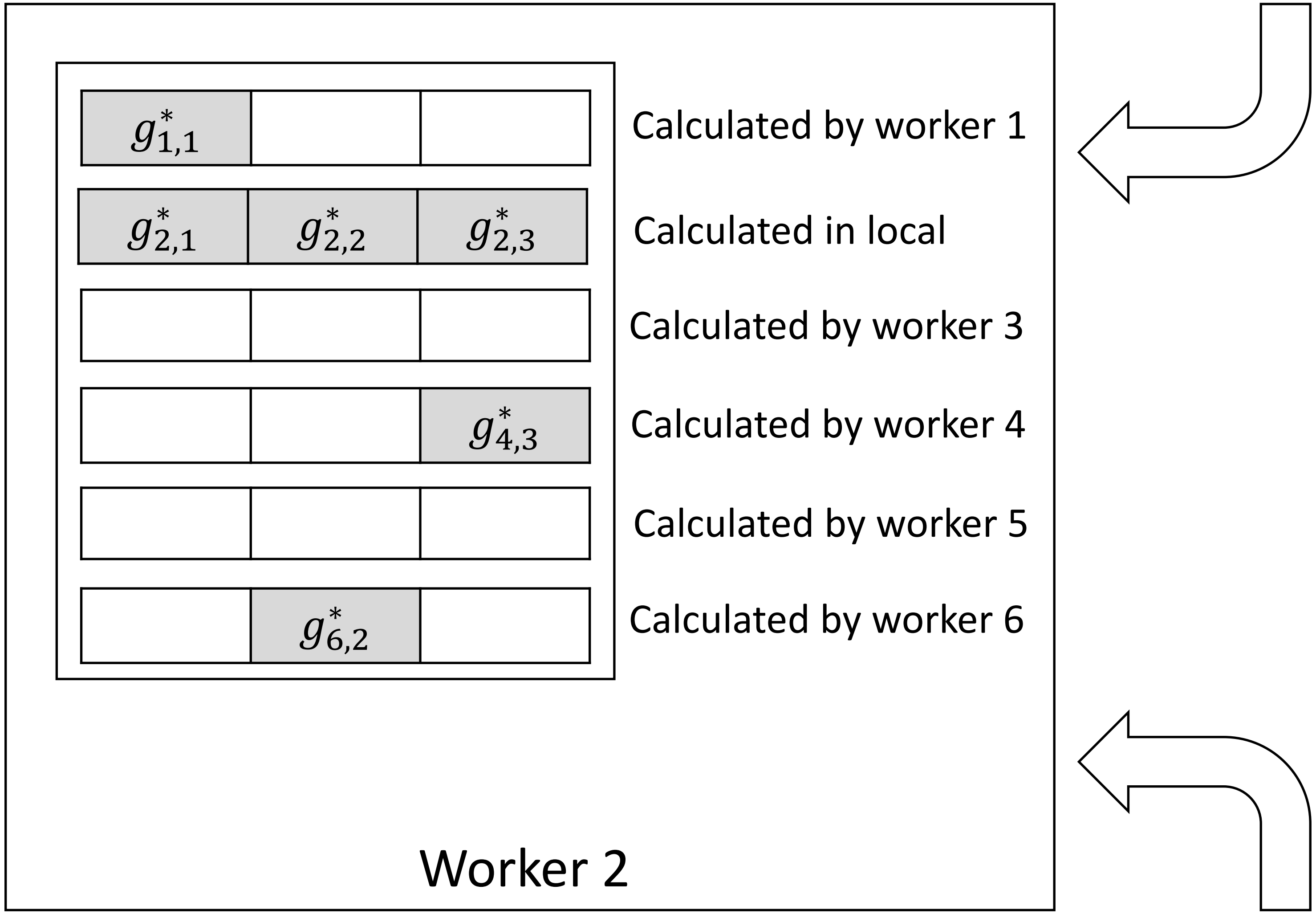
§3 Partial Gradient Exchange

- Receiving gradients



§3 Partial Gradient Exchange

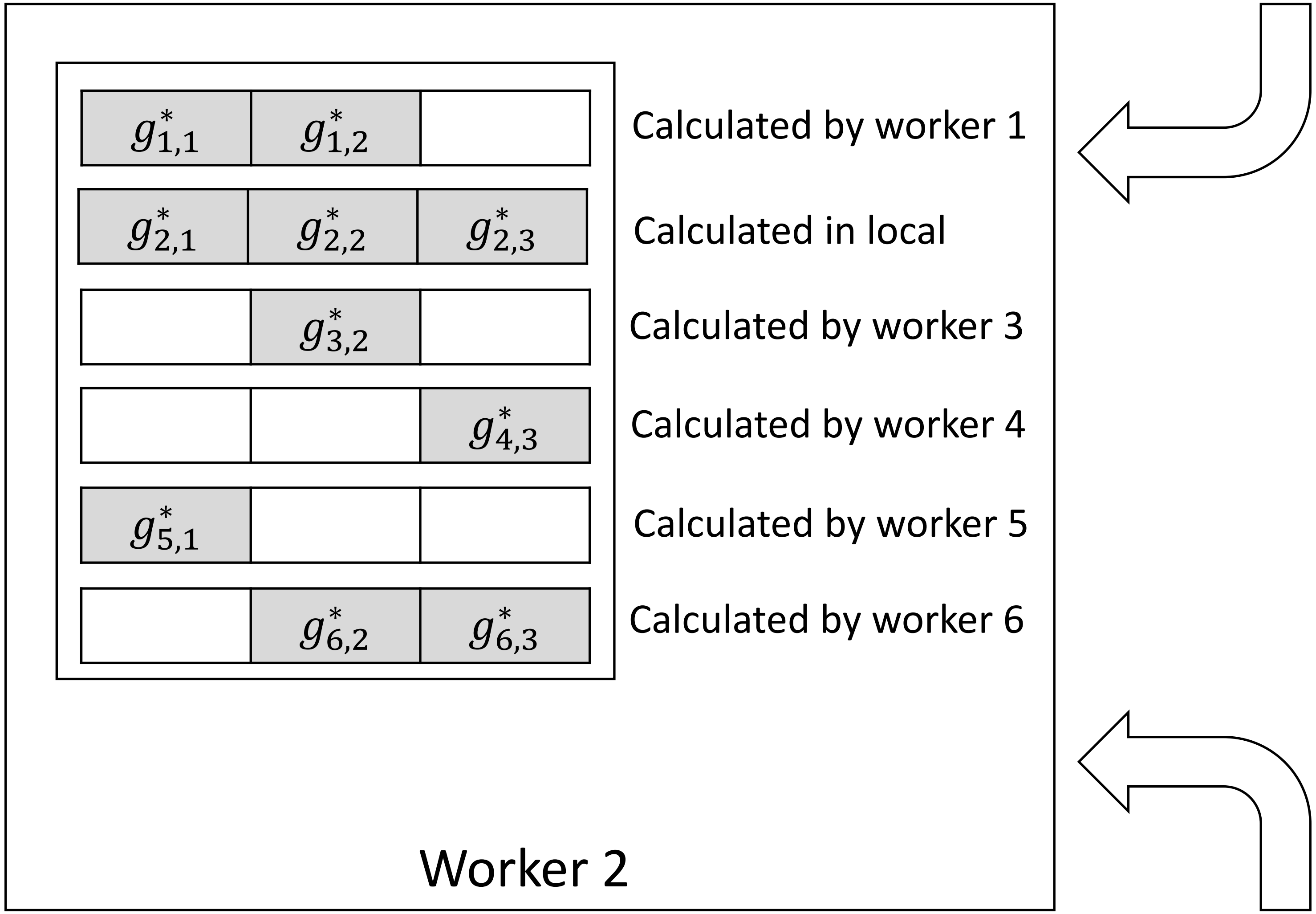
- Receiving gradients (cont.)



synchronization round : t

§3 Partial Gradient Exchange

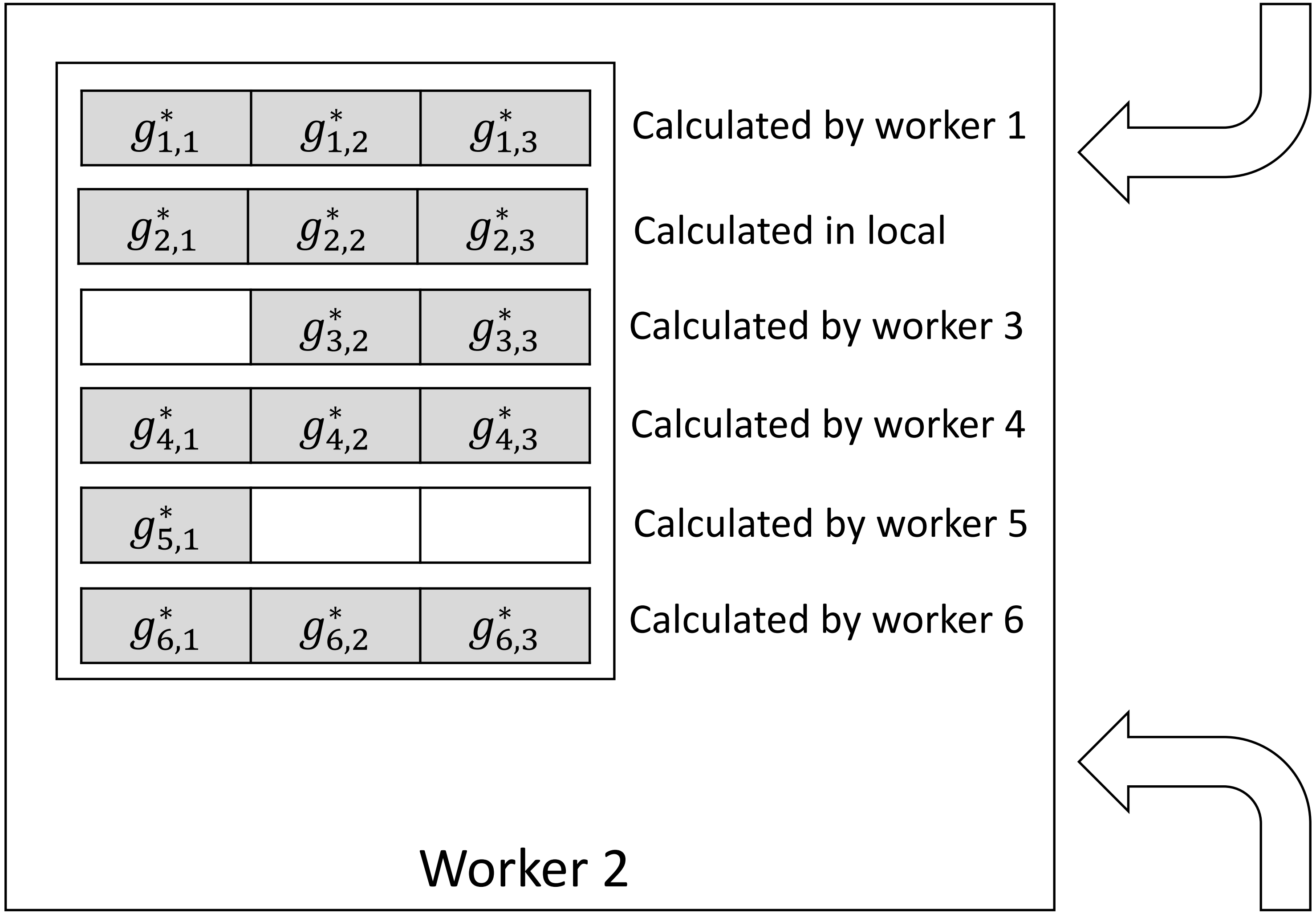
- Receiving gradients (cont.)



synchronization round : $t+1$

§3 Partial Gradient Exchange

- Receiving gradients (cont.)



synchronization round : $t + 2$

§3 Partial Gradient Exchange

- Receiving gradients (cont.)

- Since the communication is asynchronous, accumulated gradient partitions may not be received in their expected synchronisation rounds.
  - Expected to be received in p synchronisation rounds.
  - Although this introduce staleness in the local model, it does not compromise convergence (mentioned later).

§3 Partial Gradient Exchange

- Algorithm

- Each worker executes two functions, `generateGradients` and `updatePartialModel`, **asynchronously**.
  - c_j, $s_{j,i}$, and τ are used for bounding staleness (mentioned later).
  - The `updatePartialModel` function is executed when an gradient partition is received by a worker.

Algorithm 1: Partial gradient exchange

function `generateGradients` *(j, d, t, η, τ)*
  input : worker index j, mini-batch data points d, gradient computation timestamp t, learning rate η, staleness bound τ
  while $\neg$`converged` **do**
    if $c_j \leq \mathit{min}\,(s_{j,1}, \ldots, s_{j,n}) + p + \tau$ **then**
      ${}^{t}g_j \leftarrow$ `computeGradient` $({}^{t}w_j, d)$
      ${}^{(t+1)}w_j \leftarrow {}^{t}w_j + \eta \cdot {}^{t}g_j$
      ${}^{t}g_j^* \leftarrow {}^{(t-1)}g_j^* + {}^{t}g_j - {}^{(t-p)}g_j$
      $({}^{t}g_{j,1}^*, \ldots, {}^{t}g_{j,p}^*) \leftarrow$ `partitionGrad` $({}^{t}g_j^*, p)$
      for $i = 1 \ldots n$ ***in parallel*** **do**
        $k \leftarrow i \bmod p$
        `sendGradient` $(i, {}^{t}g_{j,k}^*)$
      $c_j \leftarrow c_j + 1$

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

§3 Partial Gradient Exchange

- Deciding the number of gradient partitions (*p*)

- The number of gradient partitions p impacts the statistical efficiency.
  - Workers can use cost model to select p when training begins:

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

  - Where, m is the local model size, n is the number of the workers, γ is the rate which workers compute new gradient partitions[?], and B is the given available full-bisection bandwidth.

§3 Partial Gradient Exchange

- Deciding the number of gradient partitions (p) (cont.)

- The reason of this cost model
  - The amount of data to send the full gradient is $m(n-1)$ per worker.
  - With partial gradient exchange, it is $m(n-1)/p$.
  - And only γ of the whole workers need to communicate, thus $\gamma m(n-1)/p$.
  - And this $\gamma m(n-1)/p$ is the required bandwidth usage of partial gradient exchange.
  - This means,

$$B = \frac{\gamma m(n-1)}{p}$$

Assuming system has a $m \times m$ network with bandwidth B.

  - Therefore, integer p will be represented as:

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

§3 Partial Gradient Exchange

- Bounding staleness

- The gradients computed by each worker may use weights from previous mini-batch, which introduces ***staleness***.
- To guarantee convergence, Ako imposes a **staleness bound** τ.
  - Limits the generation of new local gradients when a worker has advanced in the computation further than τ compared to all other workers.
  - To do so, each worker j maintain,
    - *Staleness clock* $s_{j,i}$ for each other worker i.
    - *Local staleness clock* c_j.
  - As p synchronisation rounds are necessary to fully propagate model, staleness bound is $p + \tau$.

§3 Partial Gradient Exchange

- Bounding staleness (cont.)

Algorithm 1: Partial gradient exchange

function `generateGradients` (j, d, t, η, τ)
  input : worker index j, mini-batch data points d, gradient computation timestamp t, learning rate η, staleness bound τ
  while $\neg$`converged` **do**
    if $c_j \leq min\,(s_{j,1}, \ldots, s_{j,n}) + p + \tau$ **then**
      ${}^{t}g_j \leftarrow$ `computeGradient` $({}^{t}w_j, d)$
      ${}^{(t+1)}w_j \leftarrow {}^{t}w_j + \eta \cdot {}^{t}g_j$
      ${}^{t}g_j^* \leftarrow {}^{(t-1)}g_j^* + {}^{t}g_j - {}^{(t-p)}g_j$
      $({}^{t}g_{j,1}^*, \ldots, {}^{t}g_{j,p}^*) \leftarrow$ `partitionGrad` $({}^{t}g_j^*, p)$
      for $i = 1 \ldots n$ ***in parallel*** **do**
        $k \leftarrow i \bmod p$
        `sendGradient` $(i, {}^{t}g_{j,k}^*)$
      $\boxed{c_j \leftarrow c_j + 1}$

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

Local staleness bound is incremented after one synchronisation round ended.

Staleness bound for worker i is incremented when partial gradient is received.

Index

1. Introduction
2. Resource Allocation in DNN Systems
3. Partial Gradient Exchange
4. Ako Architecture
5. Evaluation
6. Related work
7. Conclusion

§4 Ako Architecture

- Implementation of the Ako architecture

- The Ako architecture follows a stateful distributed dataflow model.
- Execution is broken into a series of short ***tasks***.
  - Compute tasks have one work
    - Gradient computation
  - Network tasks have four works
    - Gradient accumulation
    - Gradient partitioning
    - Gradient sending
    - Gradient receiving

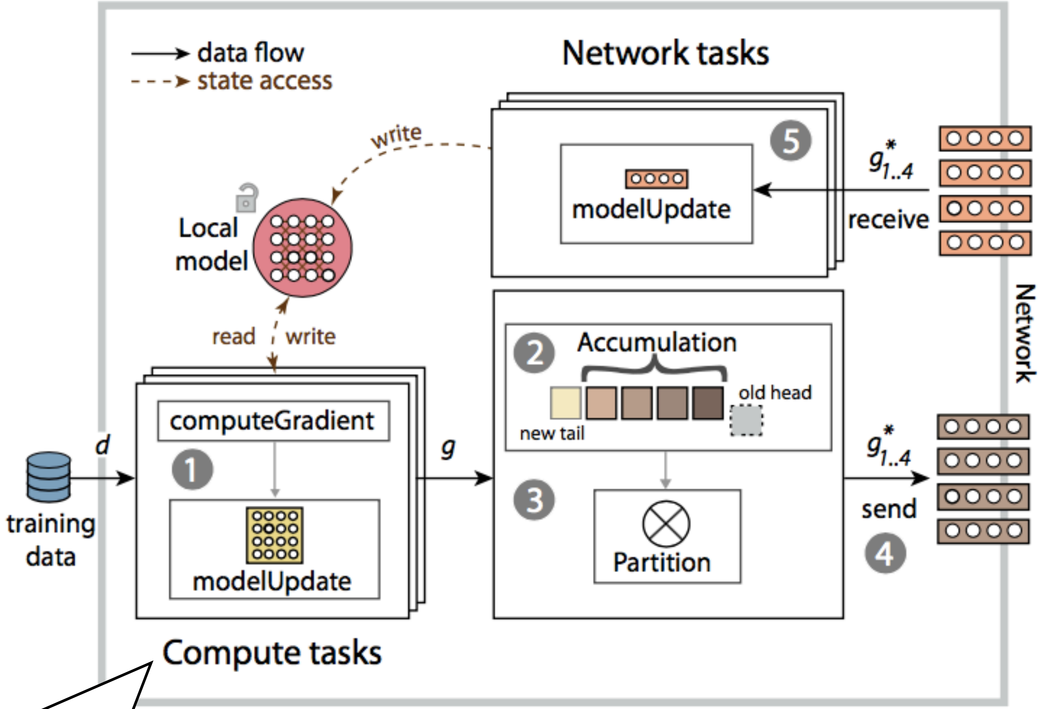


Figure 7: Architecture of an Ako worker

Computation is parallel.
Each task has exclusive
access to the local model.

§4 Ako Architecture

- Implementation of the Ako architecture (cont.)

- Gradient computation
  - Local computation is in parallel and each task has exclusive access to a partition of the local model.
  - When the gradient computation is at the end of the mini-batch, the computed (local) gradients are aggregated and updates the (local) model.
  - Update occurs concurrently with other compute task reading the (local) model.

- Gradient accumulation
  - The computed gradients at the end of a mini-batch are accumulated by a pool of network task.

§4 Ako Architecture
- Implementation of the Ako architecture (cont.)

- Gradient partitioning
  - Before sending the gradients, it is partitioned using range-partitioning.
- Gradient sending
  - Send the gradient partitions, tagged by the partitioning range, to other workers in round-robin.
  - After p rounds, complete gradients have been sent to all workers.
- Gradient receiving
  - Concurrently, workers receive gradient partitions from other worker.
  - Network task apply the gradients immediately without locking.

§4 Ako Architecture
- Fault tolerance

- Ako uses checkpointing for fault tolerance.
  - Each worker saves their local models and the staleness counter.
  - Similar to SEEP [14] and TensorFlow [1].
  - SEEP's master node notifies the other workers and let them remove the staleness counter.
    - Counters are re-added when worker recover.

Index

1. Introduction
2. Resource Allocation in DNN Systems
3. Partial Gradient Exchange
4. Ako Architecture
5. Evaluation
6. Related work
7. Conclusion

§5 Evaluation

- Setting-up datasets and DNNs

- Datasets
  - MNIST
  - ImageNet
- DNNs
  - 3 convolutional (with max-pooling) and 2 fully-connected layers.
    - For MNIST, 10/20/100 convolutional kernels (filters) with 200 neurons
    - For ImageNet 32/64/256 convolutional kernels (filters) with 800 neurons.
- Prior to training
  - Datasets are partitioned evenly across the workers.
  - The model parameters are initialized using warm-start.

§5 Evaluation

- Setting-up systems

- Ako vs. PS[w+p] (parameter server) vs. Al-to-All
  - All are implemented on top of the **SEEP** stateful distributed data platform with the same optimizations.
- Ako vs. TF (TensorFlow) vs. SG (Singa)
  - For TF and SG, asynchronous **Downpour algorithm** architecture is used to train DNNs.

- Staleness bound τ
  - Decided according to the used data set and DNN models.
  - As a heuristic, τ is increased proportionally to the # of used workers.

§5 Evaluation

- Short intro of MNIST and ImageNet

- MNIST
  - Dataset of handwritten digits (0 to 9).
  - 60,000 training sets, and 10,000 test sets.
  - Each image has 28x28 pixels which have 0 to 255 value.
  - http://yann.lecun.com/exdb/mnist/

- ImageNet
  - Dataset of images that illustrate synonym set (synset) nouns.
  - More than 14,000,000 images that have been indexed.
  - http://image-net.org/

§5 Evaluation

- Short intro to SEEP and Downpour SGD

- SEEP [14] (http://lsds.doc.ic.ac.uk/projects/SEEP)
  - An experimental parallel data processing system developed by LSDS.
  - Handles large scale stream data processing in cloud architectures with stateful operator.

- Downpour SGD [12]
  - Asynchronous SGD algorithm on parameter server deployment.
  - Using AdaGrad learning rate.

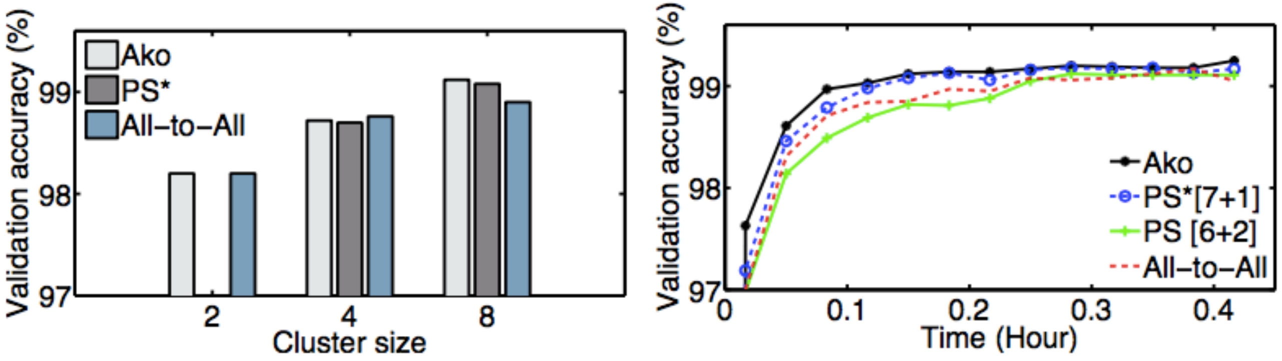
§5 Evaluation
- Performance metrics & cluster hardware

- Validation of the DNN models
  - Based on top-1 accuracy with the validation data, not the top-5.

- Hardware environment
  1. For MNIST, 16-machine cluster with 4-core Intel Xeon E3-1220 3.1GHz CPUs with 8GB RAM and 1Gbps Ethernet
  2. For ImageNet, 64-machine Amazon EC2 cluster with "m4.xlarge" Intel Xeon instances, each with 4 vCPU cores at 2.4GHz and 16GB RAM

§5 Evaluation

- Results of convergence and scalability (MNIST)

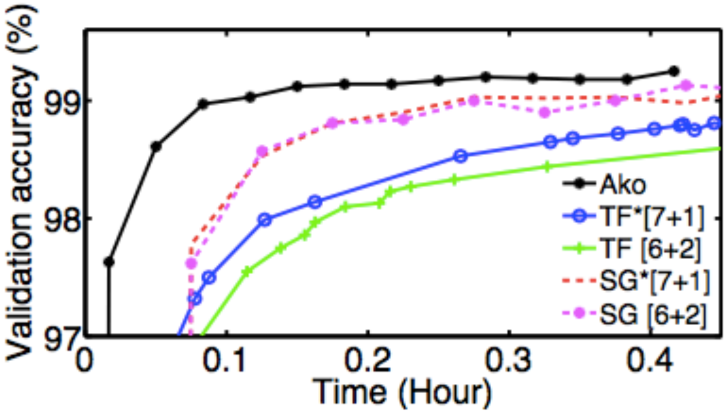


(a) Accuracy after 10 minutes of training (b) Convergence with 8 machines

- Fig. (a) shows that Ako achieves similar convergence as PS*.
  - PS*[1+3] for 4 machines and PS*[7+1] for 8 machines.
- Fig. (b) shows that Ako achieves similar convergence as PS* and converges faster than All-to-All.
  - All-to-All is not “too bad” since the data that need to communicate is not too large.

§5 Evaluation

- Results of convergence and scalability (MNIST)



(c) Comparison with TensorFlow and Singa

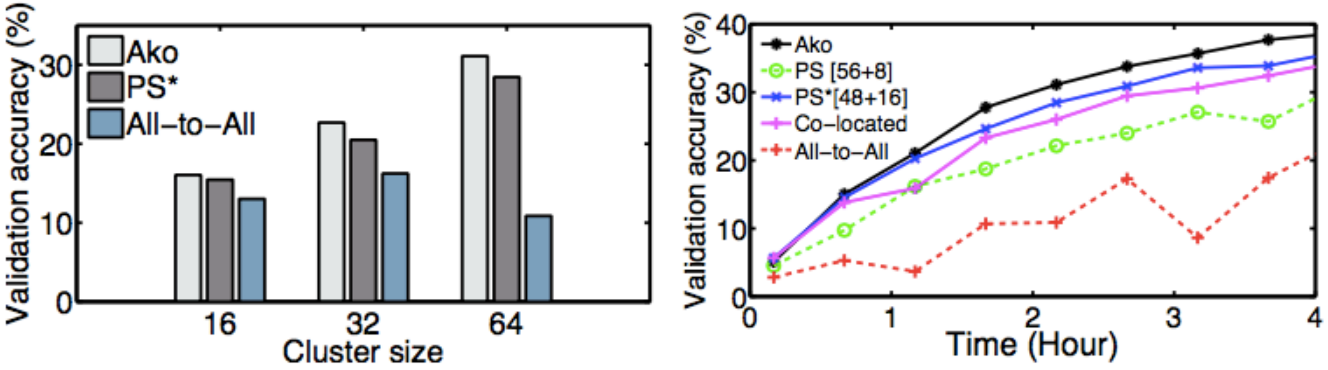
Dataset	Accuracy	TensorFlow	All-to-All	Ako
MNIST	99%	> 20 min	14 min	7 min
ImageNet	30%	3.3 h	> 4 h	1.5 h

Table 1: Time to reach target validation accuracy

- Fig. (c) shows that Ako converges faster than both TF and SG.
- From table 1, it takes Ako 7 minutes and TF* more than 20 minutes to achieve validation accuracy of 99%.
  - Author speculates this difference is caused from synchonisation under downpour SGD.

§5 Evaluation

- Results of convergence and scalability (ImageNet)



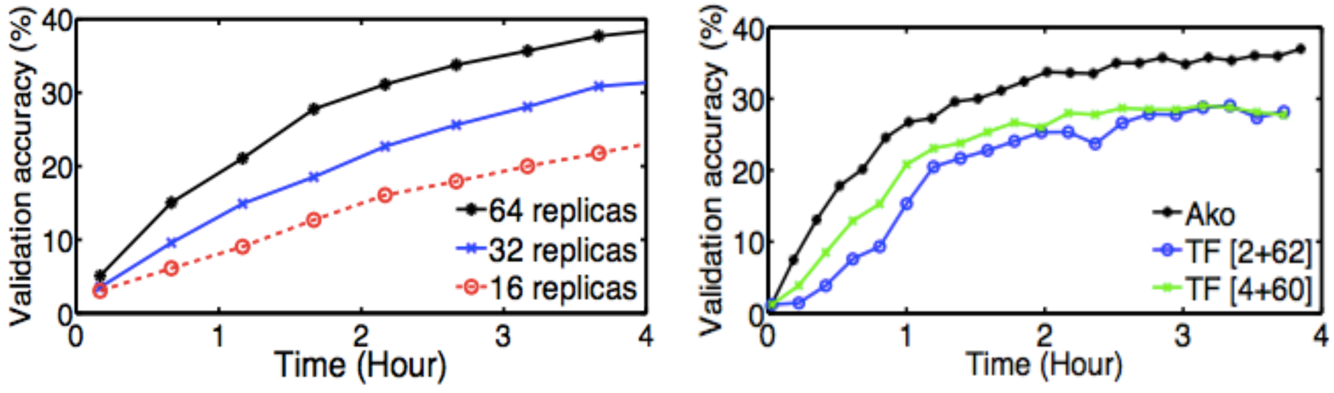
(a) Accuracy after 2-hours of training

(b) Convergence with 64 machines

- Fig. (a) shows that Ako achieves a higher validation accuracy than PS*, and with more machines, Ako and PS* convergence improves.
  - Any Ako worker can be used for validation, as difference between them are negligible.
- Fig. (b) shows that Ako requires less training time than PS*.
  - As Ako has more worker nodes than PS has.

§5 Evaluation

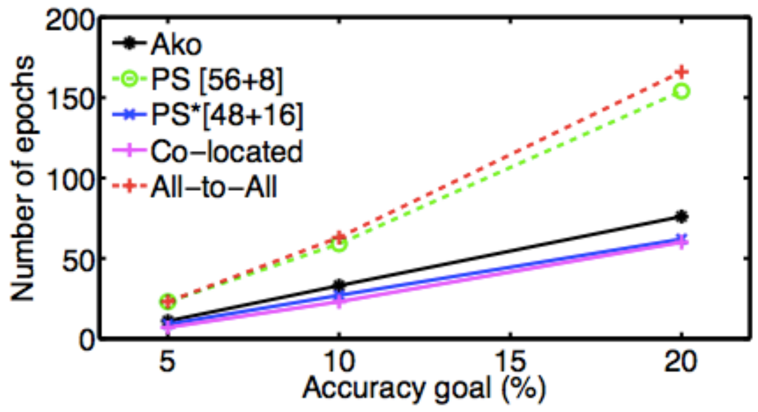
- Results of convergence and scalability (ImageNet)



(c) Convergence with different cluster sizes (a) Comparison with TensorFlow

- Fig. (c) shows that Ako scales gracefully.
  - Ako keeps the communication cost constant with p.
- Fig. (a) shows that Ako achieves higher accuracy from the begging of training.

§5 Evaluation
- Results of statistical efficiency

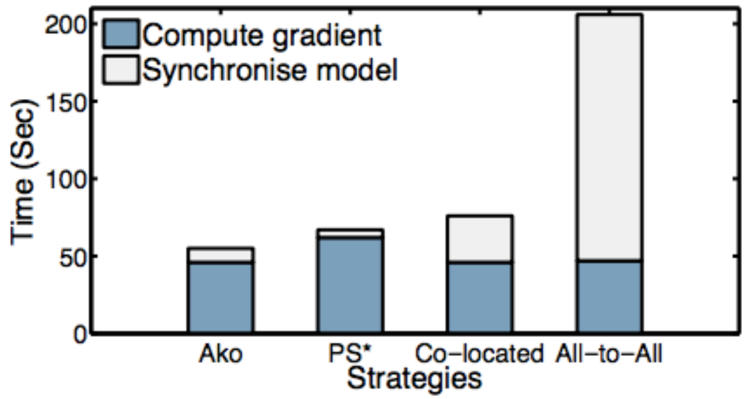


(b) Epoch number for given accuracy goal

- Number of epochs to achieve 5, 10, 15, 20% accuracy in ImageNet.
- Fig. shows that the PS approach requires the fewest passes.
  - Ako requires extra epochs, which is less statistically efficient than PS.
  - Workers receive incomplete gradients but with low latency.

§5 Evaluation

- Results of hardware efficiency



(c) Break-down of epoch time

- Collected time per epoch with two aspects.
- Fig. shows that Ako has shorter epoch time than PS.

§5 Evaluation
- Results of resource utilisation

- Average CPU resource utilisations on 16-machines were
  - Worker of Ako: 87%
  - Worker of PS*[12+4]: 84%, parameter server of PS*[12+4]: 17%
  - Worker of All-to-All: 85%

§5 Evaluation

- Results of resource utilisation

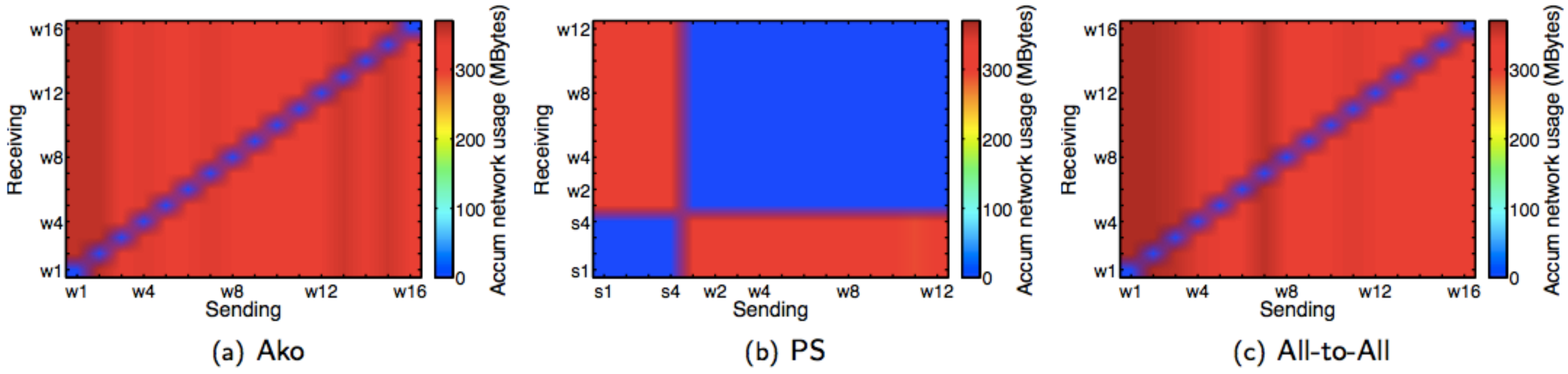
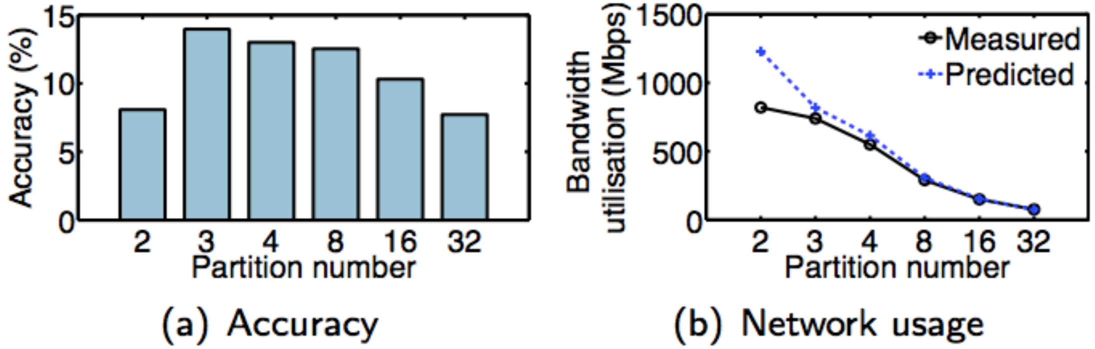


Figure 11: Average network usage with 16 machines (ImageNet)

- Fig. shows the accumulated network usage utilisation in MBs.
  - For Ako, usage is high while still achieving a low synchronisation delay.
  - For PS*, worker-worker networks are unused.
  - All-to-All also saturates the network, but suffers from a high delay.

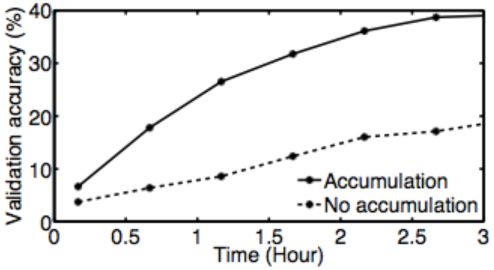
§5 Evaluation

- Effectiveness of gradient partitions and accumulation



(a) Accuracy (b) Network usage

Figure 12: Effect of gradient partitions



- Fig. (a) shows how partition number effects accuracy in Ako.
- Fig. (b) shows how partition number effects bandwidth usage in Ako.
- Right fig. shows how accumulation of gradient effects accuracy in Ako.
  - Without accumulation, workers do not receive complete gradients, make the statistical efficiency low.

Index

1. Introduction
2. Resource Allocation in DNN Systems
3. Partial Gradient Exchange
4. Ako Architecture
5. Evaluation
6. Related work
7. Conclusion

§6 Related Work
- DNN systems with parameter servers

- DistBelief [12]
- TensorFlow [1]
- Project Adam [5]
- Singa [27, 42, 43]
- Poseidon [48]
- SparkNet [25]
- Bösen [44]
- Yan et al. [47]

§6 Related Work
- DNN systems without parameter servers

- Wang et al. [41]
- MALT [23]
- CNTK [32, 33]
- Mariana [50]
- Deep Image [45]

Index

1. Introduction
2. Resource Allocation in DNN Systems
3. Partial Gradient Exchange
4. Ako Architecture
5. Evaluation
6. Related work
7. Conclusion

§7 Conclusions

- To achieve the best performance, distributed DNN systems must fully utilise the system resources.
- This paper described Ako, a decentralised DNN system that does not use parameter servers.
- In the experiment of Ako implementation on a fixed-size cluster, it achieved better performance than one with parameter servers.