# Distributed Deep Learning Systems

Thoai Nam

High Performance Computing Lab (HPC Lab)

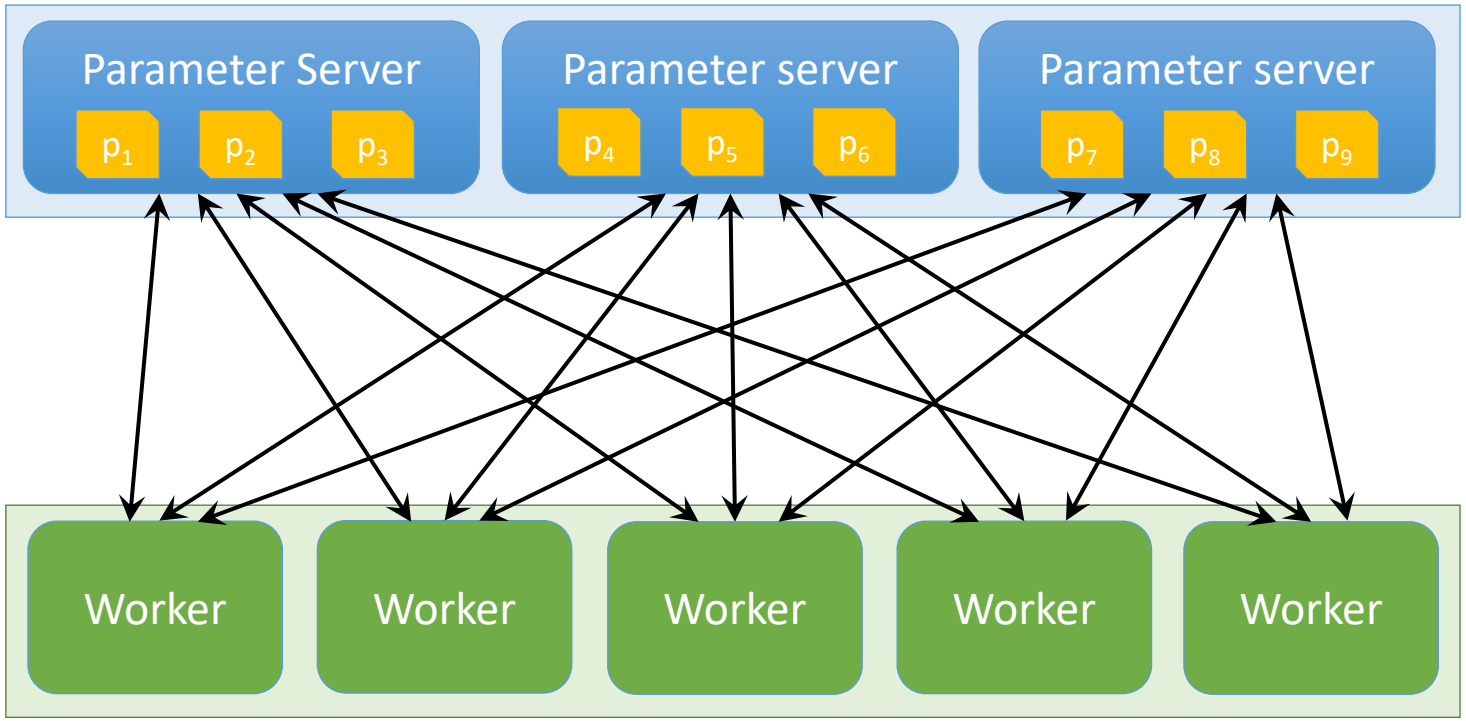
Faculty of Computer Science and Technology

HCMC University of Technology

# Parameter Server

# Parameter Server (PS)

- Model parameters are stored on PS machines and accessed via key-value interface (distributed shared memory)
- *Extensions*
  - Multiple keys (for a matrix); multiple "channels" (for multiple sparse vectors, multiple clients for same servers, …)
  - Push/pull interface to send/receive most recent copy of (subset of) parameters, blocking is *optional*
  - Can block until push/pulls with clock < $(t - \tau)$ complete




[Smola et al 2010, Ho et al 2013, Li et al 2014]

# Machine Learning (ML)

Wide array of problems and algorithms

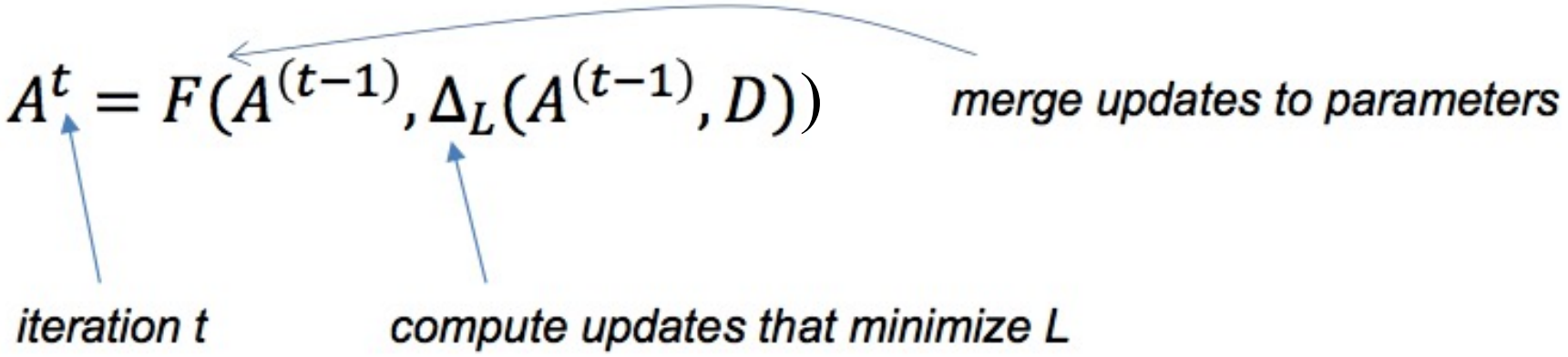
- Classification
  - Given labeled data points, predict label of new data point
- Regression
  - Learn a function from some (x, y) pairs
- Clustering
  - Group data points into “similar” clusters
- Segmentation
  - Partition image into meaningful segments
- Outlier detection

# Abstracting ML algorithms

- Can we find commonalities among ML algorithms?
- This would allow finding
  - Common abstractions
  - Systems solutions to efficiently implement these abstractions
- Some common aspects
  - We have a prediction model *A*
  - *A* should optimize some complex *objective function L*
  - ML algorithm does this by iteratively refining *A*

# High level view

- Notation
  - *D:* data
  - *A:* model parameters
  - *L:* function to optimize (e.g., minimize loss)
- Goal: Update *A* based on *D* to optimize *L*
- Typical approach: iterative convergence



# Distributed Deep Learning Systems (DDLS)

DDLSs train deep neural network models by utilizing the distributed resources of a cluster

- The massive parallel processing power of graphics processing units (GPUs) has been largely responsible for the recent successes in training deep learning models
- Increasingly larger and more complex deep learning models are necessary
- The disruptive trend towards big data has led to an explosion in the size and availability of training datasets for machine learning tasks
  - Training such models on large datasets to convergence can easily take weeks or even months on a single GPU
- Effective remedy to this problem is to utilize multiple GPUs to speed up training
- Scale-up approaches rely on tight hardware integration to improve the data throughput
  - These solutions are effective, but costly
  - Furthermore, technological and economic constraints impose tight limitations on *scaling up*
- DDLS aim at *scaling out* to train large models using the combined resources of clusters of independent machines

# Distributed SGD algorithm: all-reduce

- SGD (Stochastic Gradient Descend)

$$w_{t+1} = w_t - \alpha_t \cdot \frac{1}{B} \sum_{b=1}^{B} \nabla f_{i_{b,t}}(w_t),$$

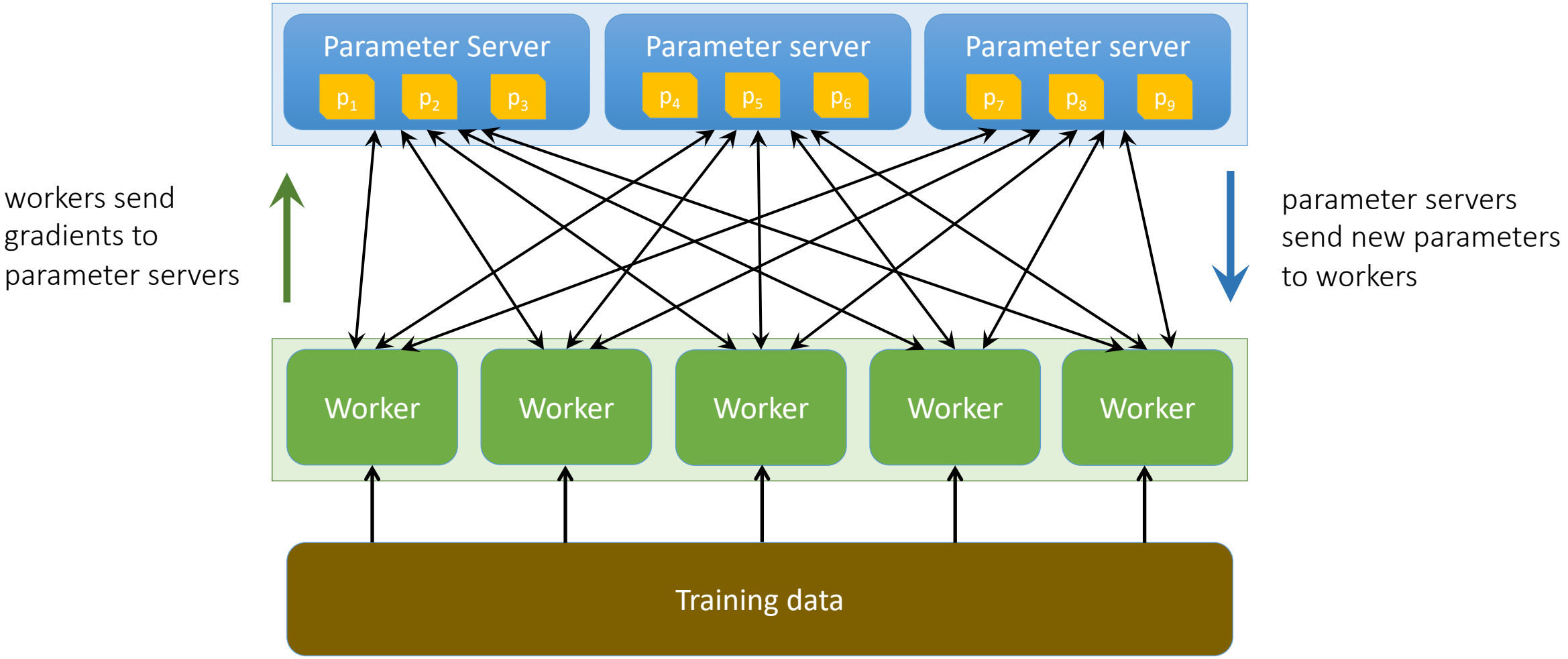
- M machines/mini-batches: B = M.B'

$$w_{t+1} = w_t - \alpha_t \cdot \frac{1}{M} \sum_{m=1}^{M} \frac{1}{B'} \sum_{b=1}^{B'} \nabla f_{i_{m,b,t}}(w_t)$$

**Algorithm 1** Distributed SGD with All-Reduce

**input:** loss function examples $f_1, f_2, \ldots$, number of machines $M$, per-machine minibatch size $B'$
**input:** learning rate schedule $\alpha_t$, initial parameters $w_0$, number of iterations $T$
**for** $m = 1$ **to** $M$ **run in parallel on machine** $m$
  **load** $w_0$ from algorithm inputs
  **for** $t = 1$ **to** $T$ **do**
    **select** a minibatch $i_{m,1,t}, i_{m,2,t}, \ldots, i_{m,B',t}$ of size $B'$
    **compute** $g_{m,t} \leftarrow \frac{1}{B'} \sum_{b=1}^{B'} \nabla f_{i_{m,b,t}}(w_{t-1})$
    **all-reduce** across all workers to compute $G_t = \sum_{m=1}^{M} g_{m,t}$
    **update model** $w_t \leftarrow w_{t-1} - \frac{\alpha_t}{M} \cdot G_t$
  **end for**
**end parallel for**
**return** $w_T$ (from any machine)

# Parameter server (PS)



**Algorithm 2** Asynchronous Distributed SGD with the Parameter Server Model

**input:** loss function examples $f_1, f_2, \ldots$, number of worker machines $M$, per-machine minibatch size $B'$
**input:** learning rate $\alpha$, initial parameters $w_0$, number of iterations per worker $T$
**for** $m = 1$ **to** $M$ **run in parallel on machine** $m$
  **load** $w_{m,0}$ from the parameter server
  **for** $t = 1$ **to** $T$ **do**
    **select** a minibatch $i_{m,1,t}, i_{m,2,t}, \ldots, i_{m,B',t}$ of size $B'$
    **compute** $g_{m,t} \leftarrow \frac{1}{B'} \sum_{b=1}^{B} \nabla f_{i_{m,b,t}}(w_{m,t-1})$
    **push** gradient $g_{m,t}$ to the parameter server
    **receive** new model $w_{m,t}$ from the parameter server
  **end for**
**end parallel for**
**run in parallel on param server**
  **initialize model** $w \leftarrow w_0$
  **loop**
    **receive** a gradient $g$ from a worker
    **update model** $w \leftarrow w - \alpha g$
    **send** $w$ back to the worker
  **end loop**
**end run on param server**
**return** $w_T$ (from any machine)

# Distributed Deep Learning Systems (DDLS)

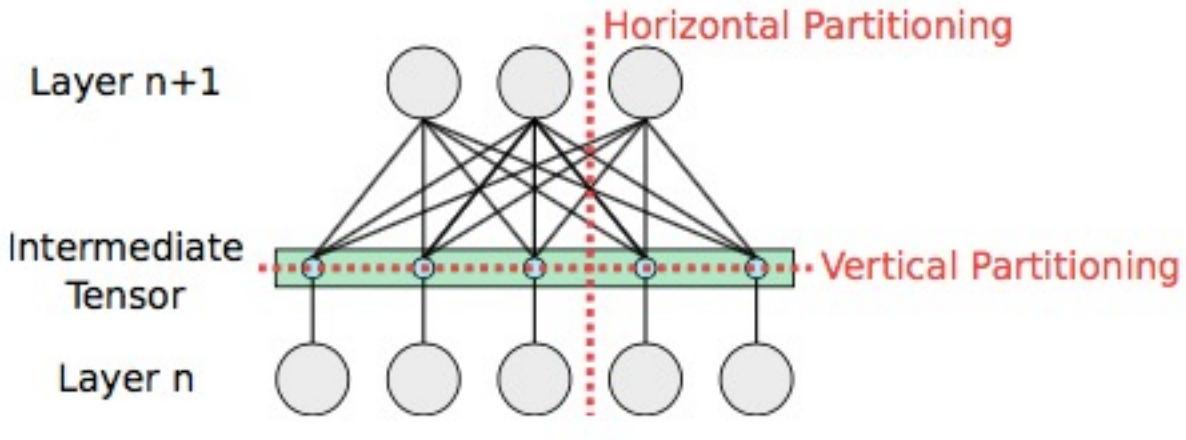
Matthias Langer, Zhen He, Wenny Rahayu, Yanbo Xue, Distributed Training of Deep Learning Models: A Taxonomic Perspective, IEEE Transactions on Parallel and Distributed Systems, 2020, Volume: 31, Issue: 12, Pages: 2802-2818, 10.1109/TPDS.2020.3003307

# How to parallelize?

- How to execute the algorithm over a set of workers?
- *Data-parallel* approach
  - Partition data *D*
  - All workers share the model parameters *A*
- *Model-parallel* approach
  - Partition model parameters *A*
  - All workers process the same data *D*

# Model parallelism

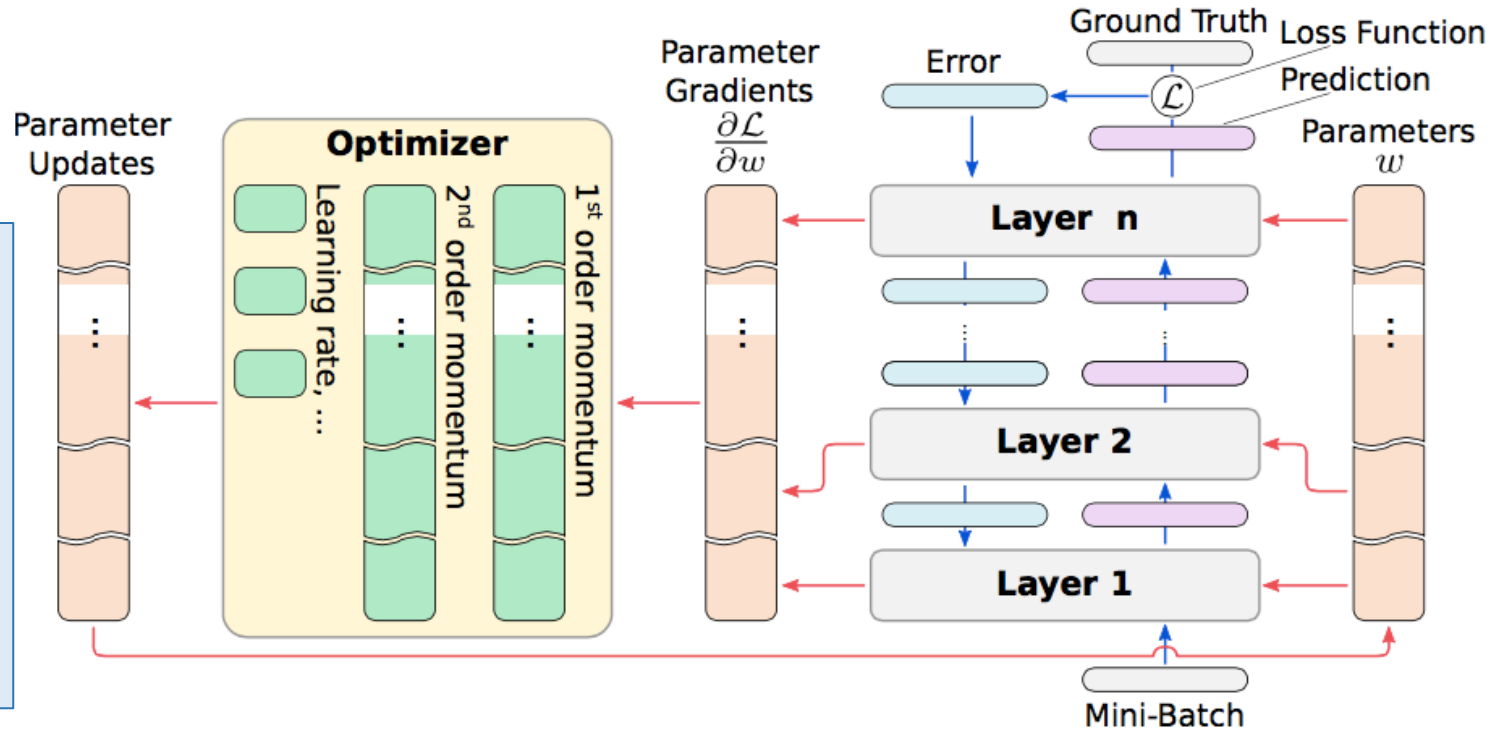
- The model is split into partitions, which are then processed in separate machines
- Model partitioning can be conducted either by applying splits between neural network layers (=*vertical partitioning*) or by splitting the layers (=*horizontal partitioning*)




- *Vertical partitioning* can be applied to any deep learning model because the layers themselves are unaffected
- *Horizontal partitioning*: the layers themselves are partitioned

# Data parallelism

① Each worker downloads the current model
② Each worker performs backpropagation using its assignment of data in parallel
③ The respective results are aggregated and integrated to parameter server in order to form a new model.




Distinct data flow cycles in deep learning models during training (→ = gradient computation cycle; → = model update / optimization cycle)

- Increasing the overall sample throughput rate by replicating the model onto multiple machines, where backpropagation can be performed in parallel, to gather more information about the loss function faster
- Most transformations applied to a specific training sample in deep neural networks do not involve data from other samples
- Sum of per-parameter gradients computed using subsets $(x^o, \cdots, x^n)$ of a mini-batch $(x)$ matches the per-parameter gradients for the entire input batch:

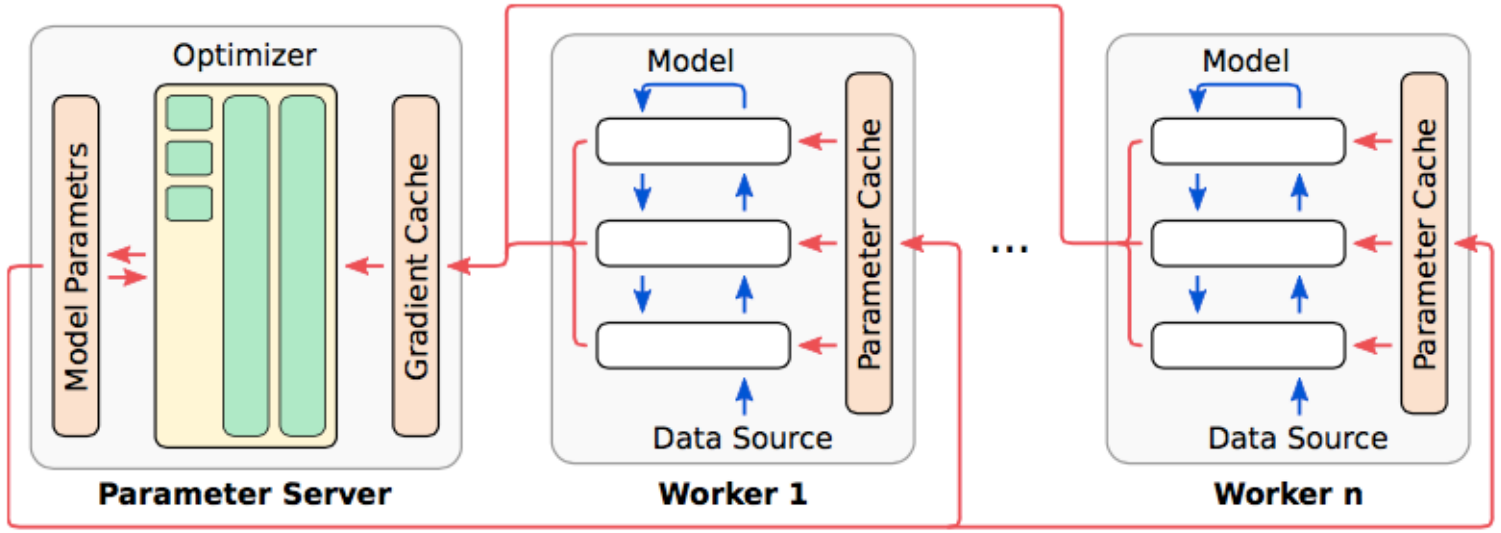
$$\partial \mathcal{L}(x; w)/\partial w = \partial \mathcal{L}(x^o; w)/\partial w + \ldots + \partial \mathcal{L}(x^n; w)/\partial w$$

# Centralized & Decentralized optimization

- *Centralized optimization*
  - The *optimization cycle* is executed in a central machine, while the *gradient computation* code is replicated onto the remaining cluster nodes
- *Decentralized optimization*
  - *Both cycles* are replicated in each cluster node and some form of synchronization is realized that allows the distinct optimizers to act cooperatively

# Centralized optimization

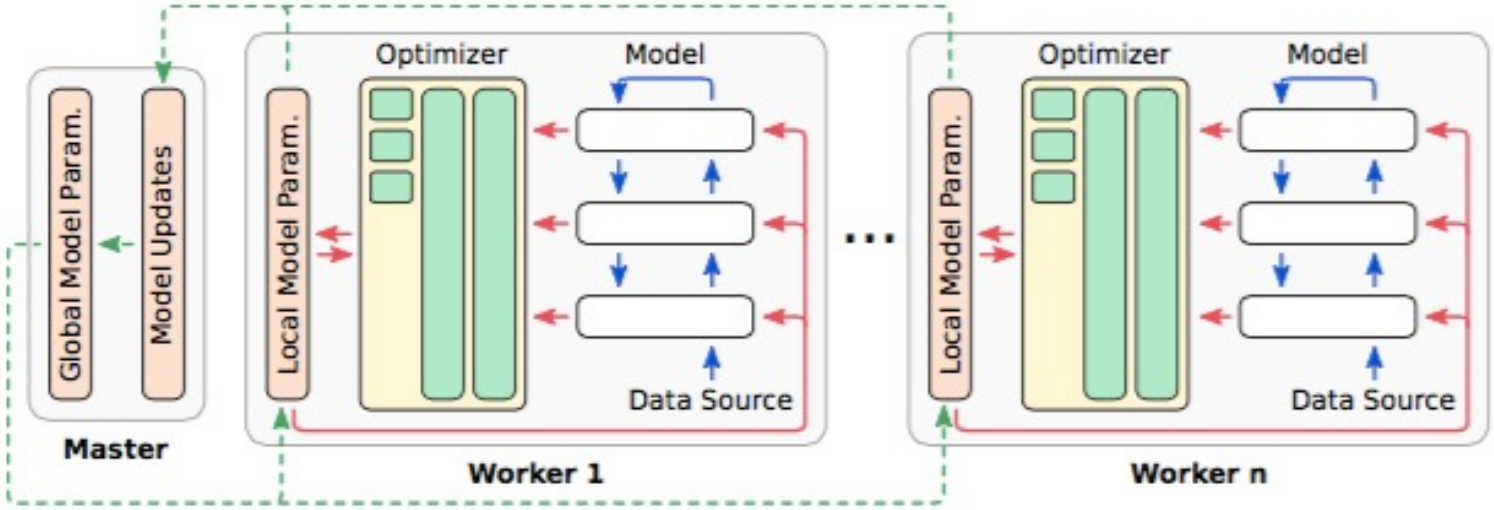
- A single optimizer instance (often called *parameter server*) is responsible for updating a specific model parameter
- Parameter servers depend on the gradients computed by workers that perform backpropagation (*workers*)
- Depending on whether computations across workers are scheduled synchronously or asynchronously, this can have different effects on the optimization.




- The blue process → computes per-parameter gradients based on the current model parameters by applying backpropagation on mini-batches drawn from the training data
- The optimization cycle → consumes these gradients to determine model parameter updates.

# Decentralized optimization (1)

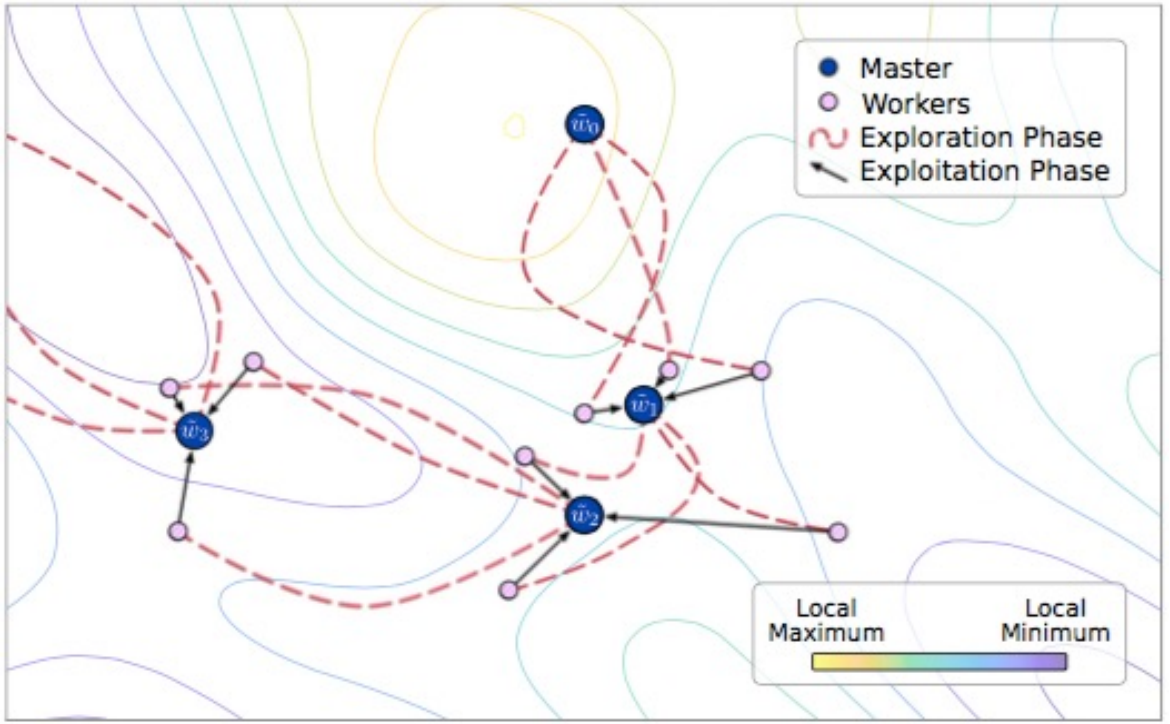
- Each worker independently probes the loss function to find gradient descent trajectories to minima that have good generalization properties
- To arrive at a *better* joint model, some form of arbitration is necessary to bring the different views into alignment.




- In this figure, we assume the existence of a dedicated master node, which processes the individual parameter adjustments suggested by the workers and comes up with a new global model state that is then shared with them
- The blue process → computes per-parameter gradients based on the current model parameters by applying backpropagation on mini-batches drawn from the training data
- The optimization cycle → consumes these gradients to determine model parameter updates.

# Decentralized optimization (2)

- Multiple independent entities concurrently try to solve a similar but not exactly the same problem
  - The loss function in deep learning is usually non-trivial
  - Find different descent trails more appealing and converge towards different local minima
- Over time, the workers diverge and eventually arrive at incompatible models
  - models that cannot be merged without destroying the accumulated information
  - DDLS that rely on decentralized optimization have to take measures to limit divergence




➢ The master and all workers start from the same model state $w_o$ on the yellow plateau (=high loss)

➢ *Exploration phase* (→)
  - The workers iteratively evaluate the loss function using different mini-batches and independently update their local models

➢ *Exploitation phase* (→)
  - Each worker shares its model updates with the master node, which in turn merges the updates to distill latent parameter adjustments that have worked better on average across the investigated portion of the training dataset

➢ A revised new global state $w_1$ is then shared with the workers, which use it as the starting point of the next exploration phase

# Synchronous & Asynchronous scheduling

- *Synchronous* systems
  - In bulk synchronous (or simply *synchronous*) systems, computations across all workers occur simultaneously
  - Global synchronization barriers ensure that individual worker nodes do not progress until the remaining workers have reached the same state
- *Asynchronous* systems
  - *Asynchronous* systems take a more relaxed approach to organizing collaborative training and avoid delaying the execution of a worker to accommodate other workers (i.e. the workers are allowed to operate at their own pace)
- *Bounded asynchronous* systems
  - A hybrid approach between two above archetypes
  - They operate akin to centralized asynchronous systems, but enforce rules to accommodate workers progressing at different paces
  - The workers operate asynchronously with respect to each other, but only within certain *bounds*

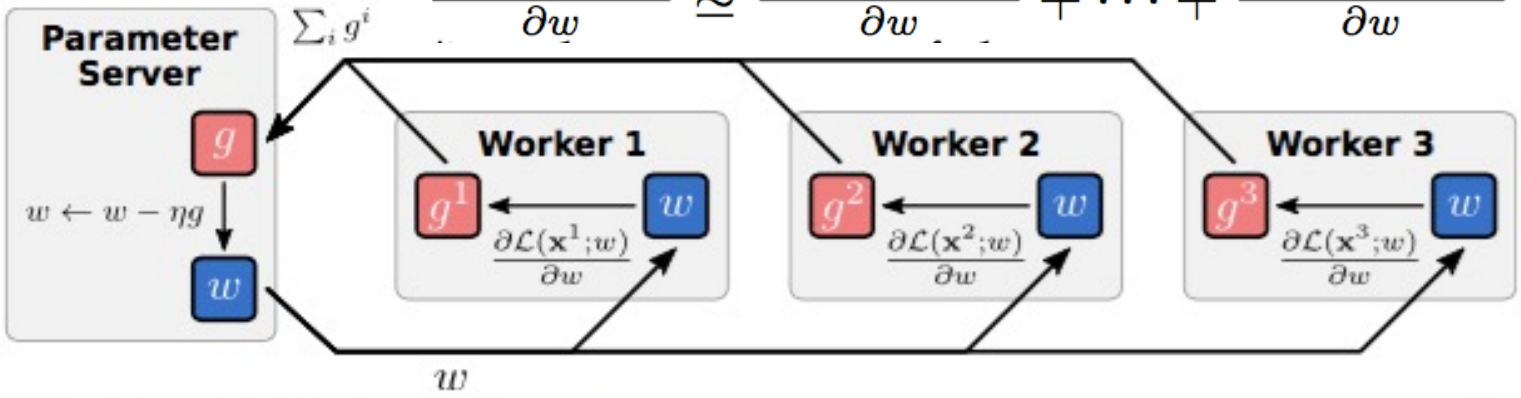
# Centralized Synchronous systems (1)

- Centralized systems
  - Model training is split between the workers (=gradient computation) and the parameter servers (=model update)
- Synchronization:
  - Training cannot progress without a full parameter exchange between the parameter server and its workers
  - The parameter server is dependent on the gradient input to update the model
  - The workers are dependent on the updated model in order to further investigate the loss function
  - The cluster as a whole cyclically transitions between phases, during which all workers perform the same operation

# Centralized Synchronous systems (2)

- Each training cycle begins with the workers *downloading* new model parameters ($w$) from the parameter server
- Workers locally sample a training mini- batch ($x \sim \mathcal{D}^i$) and *compute* per-parameter gradients ($g^i$)
- Workers *share* their gradients with the parameter server
- The parameter server *aggregates* the gradients from all workers and injects the aggregate into an optimization algorithm to *update* the model.

$$\frac{\partial \mathcal{L}(\mathbf{x}\sim\mathcal{D};w)}{\partial w} \approx \frac{\partial \mathcal{L}(\mathbf{x}\sim\mathcal{D}^1;w)}{\partial w} + \cdots + \frac{\partial \mathcal{L}(\mathbf{x}\sim\mathcal{D}^n;w)}{\partial w}$$




**PARAMETER SERVER PROGRAM**

**Require:** initial model $\tilde{w}_0$, learning rate $\eta$, number of workers $n$

1: **for** $t \leftarrow 0, 1, 2, \ldots$ **do**
2: Broadcast $\tilde{w}_t$
3: Await gradients $g_t^i$ from all workers
4: $\tilde{w}_{t+1} \leftarrow \tilde{w}_t - \eta \sum_{i=1}^{n} g_t^i$
5: **end for**

**PROGRAM OF THE $i^{\text{th}}$ WORKER**

**Require:** training data source $\mathcal{D}^i$

1: **for** $t \leftarrow 0, 1, 2, \ldots$ **do**
2: Await $\tilde{w}_t$
3: Sample mini-batch $\mathbf{x} \sim \mathcal{D}^i$
4: $g_t^i \leftarrow \frac{\partial \mathcal{L}(\mathbf{x};\tilde{w}_t)}{\partial \tilde{w}_t}$
5: Send $g_t^i$ to parameter server
6: **end for**

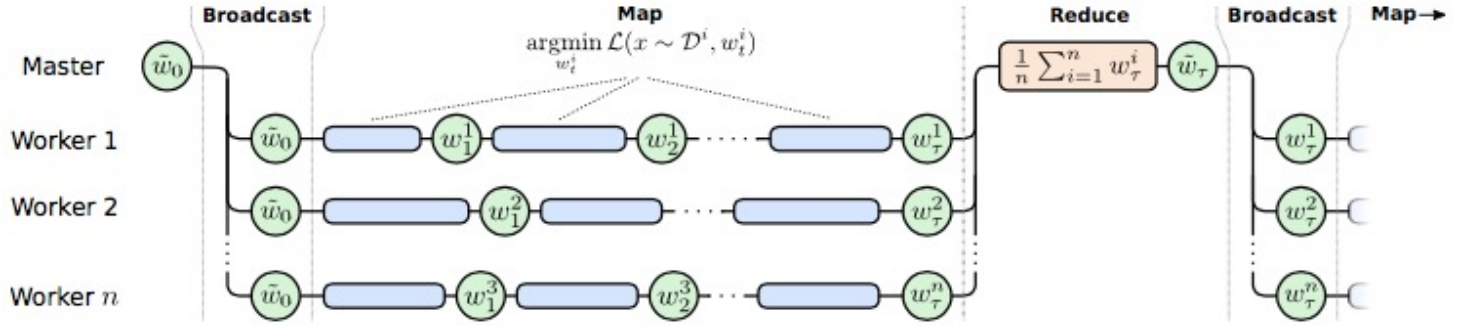
# Centralized Synchronous systems (3)

- Assuming appropriate and sufficient random sampling, larger mini-batches may represent the training distribution better [31]
- Optimizing a model using mini-batches with a large coverage of the training distribution tend to get trapped in sharp minima basins of the loss function

- The next training step can only be conducted once all workers have completed their assigned task and submitted gradients
  - A majority of cluster machines always has to wait for stragglers [34]
- Relaxed solutions
  - The training set is (1) large enough, (2) reasonably well-balanced, and (3) sufficiently randomly distributed among the workers
  - Minor portions of the training data are absent, such that this requirement can be relaxed
    - Ending training epochs once 75% of all training samples have been processed [23]
    - Over-provision by allocating more workers and ending each gradient aggregation phase once a quorum has been reached [17]

# Decentralized Synchronous systems (1)

- Rely on decentralized optimization independently conduct model training in each worker
- Works do not exchange parameters to further model training, but rather to share the independent findings from each worker with the rest of the cluster to determine descent trajectories with good generalization properties
- Workers operate in phases separated by global synchronization barriers




**MASTER PROGRAM**

**Require:** initial model state $\tilde{w}_0$, number of workers $n$

1: $t \leftarrow 0$
2: **loop**
3: Broadcast $\tilde{w}_t$
4: Await models $w^i_{t+\tau}$ from all workers
5: $\tilde{w}_{t+\tau} \leftarrow \frac{1}{n} \sum_{i=1}^{n} w^i_{t+\tau}$
6: $t \leftarrow t + \tau$
7: **end loop**

**PROGRAM RUN BY THE $i^{\text{th}}$ WORKER**

**Require:** training data source $\mathcal{D}^i$, learning rate $\eta$

1: $t \leftarrow 0$
2: **loop**
3: Await $\tilde{w}_t$
4: $w^i_t \leftarrow \tilde{w}_t$
5: **for** $u \leftarrow t+1, t+2 \ldots, t+\tau$ **do**
6: Sample mini-batch $\mathbf{x} \sim \mathcal{D}^i$
7: $w^i_{u+1} \leftarrow w^i_u - \eta \frac{\partial \mathcal{L}(\mathbf{x}; w^i_u)}{\partial w^i_u}$
8: **end for**
9: Send $w^i_{t+\tau}$ to parameter server
10: $t \leftarrow t + \tau$
11: **end loop**

The decentralized synchronous system SparkNet [10]

# Decentralized Synchronous systems (2)

- The initial model parameters $(\tilde{w}_0)$ are distributed among the workers to initialize the local models $(w^i)$

## Exploration phase

Each worker

- Randomly sample mini-batches from their locally available partition of the training dataset
- Determine per-parameter gradients and adjust their model to minimize the loss function ($\mathcal{L}$)
- This process is repeated $\boldsymbol{\tau}$ times, during which each worker independently trains its local model in isolation

- Due to the different properties of the mini-batches, each worker eventually arrives at a slightly better (w.r.t. $\mathcal{L}$), but different model
- The master node acts as a synchronization conduit

## Exploitation phase

- The worker models $(w_\tau^i)$ are merged to form a new joint model $(\tilde{w}_{t+\tau})$

# Decentralized Synchronous systems (3)

- Keeping the local optimizers running for too long will result in reduced convergence performance or even setbacks if the worker models diverge too far
  - Limit the amount of independent exploration steps ($\tau$) [10, 25]

  => small $\tau$
- The best rate of convergence for a given model can typically be achieved if $\tau$ is rather small ($\tau \leq 10$) [8, 10].

- $\tau$ determines how much time should be spent on improving the local models versus synchronizing the states across machines
- To make the best use of the cluster GPUs

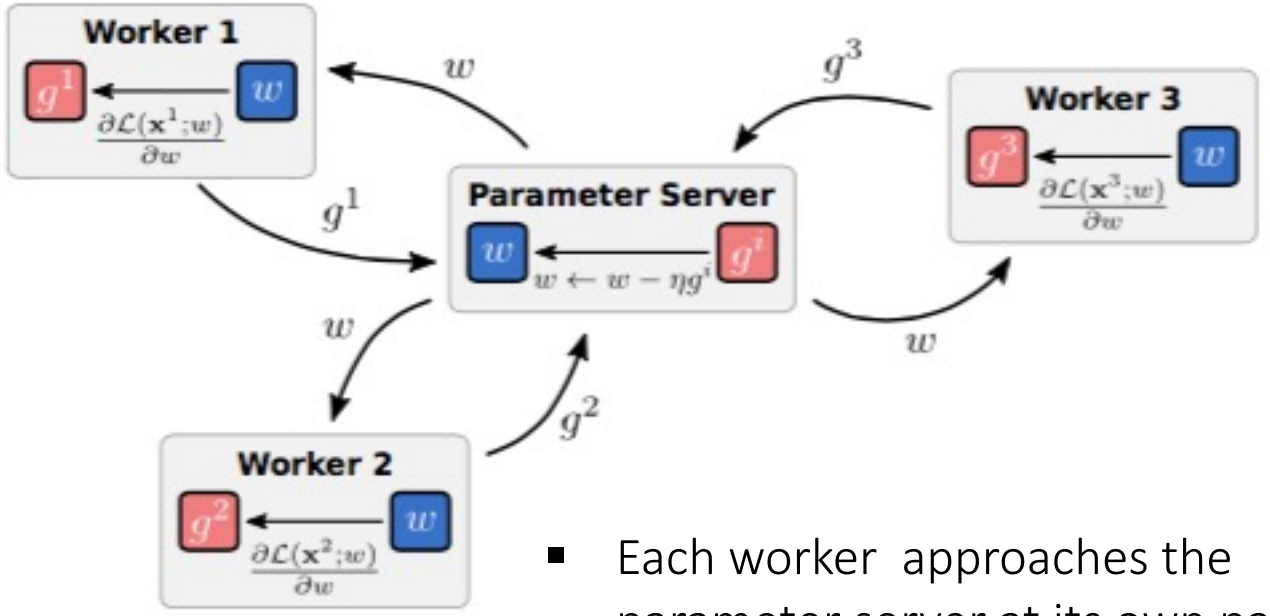
  => large $\tau$
  - Often lead to sub-optimal convergence rates [11]

- ➢ Any choice of $\tau$ represents the dilemma of finding a balance between harnessing the benefits from having more computational resources and the need to limit divergence among workers
- ➢ Practically motivated suggestions such as to aim for a 1:5 computation-to-computation ratio (≈83.3% GPU utilization; [10]) may serve as a starting point for hyper-parameter search and to determine whether efficient decentralized optimization is possible at all using a certain configuration.

# Centralized Asynchronous systems (1)

- *Asynchronous systems*
  - Each worker acts alone
- *Centralized systems*
  - Each worker shares its gradients with the parameter server once a mini-batch has been processed
- Centralized Asynchronous DDLS [14], [17], [19], [21], [35]
- Instead of waiting for other workers to reach the same state, the parameter server eagerly injects received gradients into the optimization algorithm to train the model
  - Each update of the global model is only based on the gradient input from a single worker
  - Similar to the eager aggregation mechanisms
- Instead of discarding the results from all remaining workers and losing the invested computational resources, each worker is allowed to simply continue using its locally cached stale version of the model parameters

# Centralized Asynchronous systems (2)




**PARAMETER SERVER PROGRAM**

**Require:** initial model state $\tilde{w}_0$, learning rate $\eta$

1: $\tilde{w} \leftarrow \tilde{w}_0$
2: Distribute $\tilde{w}$
3: **for** $t \leftarrow 0, 1, 2, \ldots$ **do**
4: **if** received gradients $(g^i_{t-\delta})$ from worker $i$, with a delay of $\delta$ steps **then**
5: $\tilde{w} \leftarrow \tilde{w} - \eta g^i_{t-\delta}$
6: Send $\tilde{w}$ to worker $i$
7: **end if**
8: **end for**

**PROGRAM OF THE $i^{\text{th}}$ WORKER**

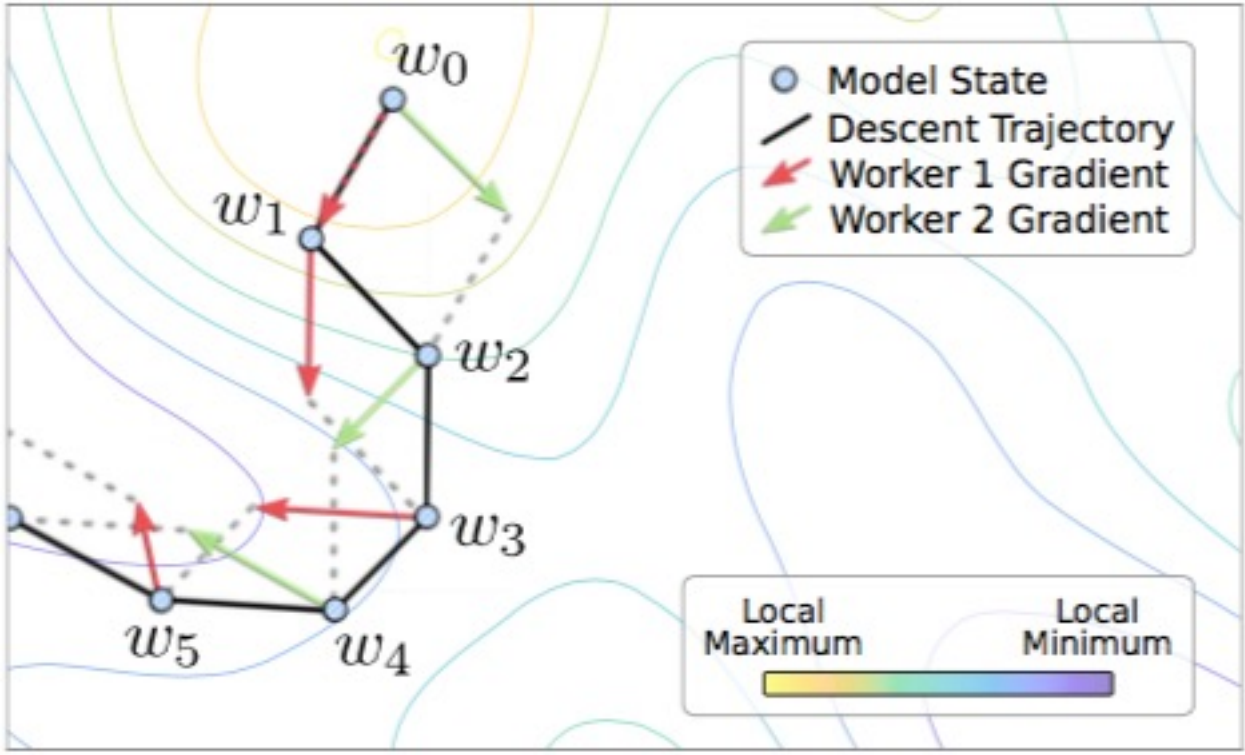
**Require:** training data source $\mathcal{D}^i$

1: **for** $t^i \leftarrow 0, 1, 2, \ldots$ **do**
2: Await of current parameter server model $\tilde{w}$
3: $w^i \leftarrow \tilde{w}$
4: Sample mini-batch $\mathbf{x} \sim \mathcal{D}^i$
5: $g^i \leftarrow \frac{\partial \mathcal{L}(\mathbf{x};w^i)}{\partial w^i}$
6: Send $g^i$ to parameter server
7: **end for**

- Each worker approaches the parameter server at its own pace to offer gradient, after which the global model is updated immediately, and request updated model parameters
- Each worker maintains a separate parameter exchange cycle with the parameter server
- There is no interdependence between workers, situations where straggler nodes delay the execution of other workers cannot happen.

# Centralized Asynchronous systems (3)

- For this system to work, choosing the results from one worker over another must not introduce a bias that significantly changes the shape of the loss function
- Thus, on average, the mini-batches sampled by each worker have to mimic the properties of the training distribution reasonably well
- At any point in time, only a single worker is in possession of the most recent version of the model
  - Other workers only possess stale variants that represent the state of the parameter server during their last interaction with it
  - Any gradients that they produce are relevant to the shape of the loss function around that stale model representation




- ✧ In the figure, both workers start from the same model state ($w_0$), but draw different mini-batches from the same distribution
- ✧ *Worker 1*: send gradients($w_0$), PS($w_0$->$w_1$), receive $w_1$
- ✧ *Worker 2*: send gradients($w_0$), PS($w_1$->$w_2$), receive $w_2$
- ✧ *Worker 1*: send gradients($w_1$), PS($w_2$->$w_3$), receive $w_3$
- ✧ *Worker 2*: send gradients($w_2$), PS($w_3$->$w_4$), receive $w_4$

➢ Staleness has serious implications on model training [37]
  - In a cluster with multiple asynchronous workers, the next worker to exchange parameters is usually stale by some amount of update steps

# Centralized Asynchronous systems (4)

- The fair scheduling is undesirable in practice
  - The slowest machine would hold back faster machines,
  - which is exactly the situation that asynchronous systems try to avoid
- Gradients from severely eclipsed workers can confuse the parameter server's optimizer
  - Can setback training or even destroy the model
- To avoid compounding delays
  - The parameter server typically places workers that indicated their readiness to upload gradients in a priority queue based on their staleness [17], [19], [42]
- To protect against adverse influences from severe stragglers, some systems allow defining conditions that must be fulfilled before queued requests can be processed by the parameter server
- These conditions typically take the form of either *a value* or *delay bound*.

# Bounded Asynchronous systems

- *Value bounds*
  - The parameter server maintains a copy of all versions of the model currently in use across the cluster
    - $w_{t-\delta}$: currently known by the slowest worker
    - $w_t$: he most recent model
  - $w_t - w_{t-\delta}$ is the amount of change in transit that is currently not known by the slowest worker
  - If a worker triggers an update that leads to a violation of some value bound (i.e. $\|w_t - w_{t-\delta}\|_\infty \geq \Delta_{max}$ ), it is delayed until the value bound condition holds again
  - Choosing a reliable metric and limit for a value bound can be difficult [38]
    - The magnitude of future model updates is largely unknown
    - Adjustment during training.

- *Delay bounds*

(e.g. the Stale Synchronous Parallel [35])

  - Each worker (i) maintains a separate clock ($t^i$)
  - Whenever a worker submits gradients to the parameter server, $t^i$ is increased
  - If the clock of a worker differs from that of the slowest worker by more than $s$ steps, it is delayed until the slow worker has caught up
  - If a worker downloads the current global model it is ensured that this model includes all local updates and may also contain updates from other workers within a range of $[t^i-s, t^i+s-1]$ update steps.

# Decentralized Asynchronous systems (1)

- The workers act independently and continue to explore the loss function based on a model that is detached from the master's current state
- The workers cannot replace their model parameters upon completing a parameter exchange with the master node
- Instead, workers have to merge the respective asynchronously gathered information

- Combining master $(\tilde{w})$ and worker models $(w^i)$ in such a setting is to apply linear interpolation as

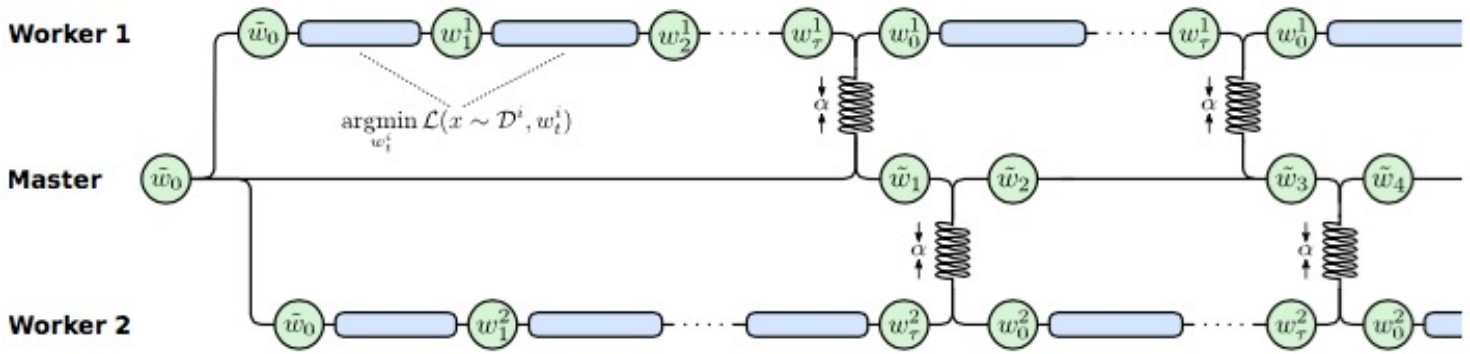
$$w^i \leftarrow w^i - \alpha(w^i - \tilde{w})$$
$$\tilde{w} \leftarrow \tilde{w} + \beta(w^i - \tilde{w}) \quad \text{for } \{\alpha, \beta\} \in \left[0, \frac{1}{2}\right] \text{ and } \alpha \geq \beta \quad (1)$$

([11], [25], [29], [30], [42], [43])

  - The worker model is displaced towards the master model's state at a rate of α times their relative distance
  - The master model is displaced in the opposite direction at a rate of β
  - This operation is equivalent to temporarily extending the loss function with the squared $l^2$-norm of the difference between both models.

# Decentralized Asynchronous systems (2)

- The decentralized asynchronous system *Elastic Averaging SGD* (EASGD) [30]
- Once τ iterations have been completed by a worker
  - The master node's current model $\tilde{w}$ is downloaded and the penalization term $\delta^i$ is computed and applied to the local model
  - Then $\delta^i$ is transferred to the master node, which applies the inverse operation to $\tilde{w}$ (α=β)
- A symmetric force (elastic symmetric) between each worker and the master node that equally attracts both models




**MASTER PROGRAM**

**Require:** initial model state $\tilde{w}_0$

$\tilde{w} \leftarrow \tilde{w}_0$
**loop**
  **if** received download request from worker $i$ **then**
    Upload $\tilde{w}$ to worker $i$
  **end if**
  **if** received $\delta^i$ from worker $i$ **then**
    $\tilde{w} \leftarrow \tilde{w} + \delta^i$
  **end if**
**end loop**

**PROGRAM RUN BY THE $i^{\text{th}}$ WORKER**

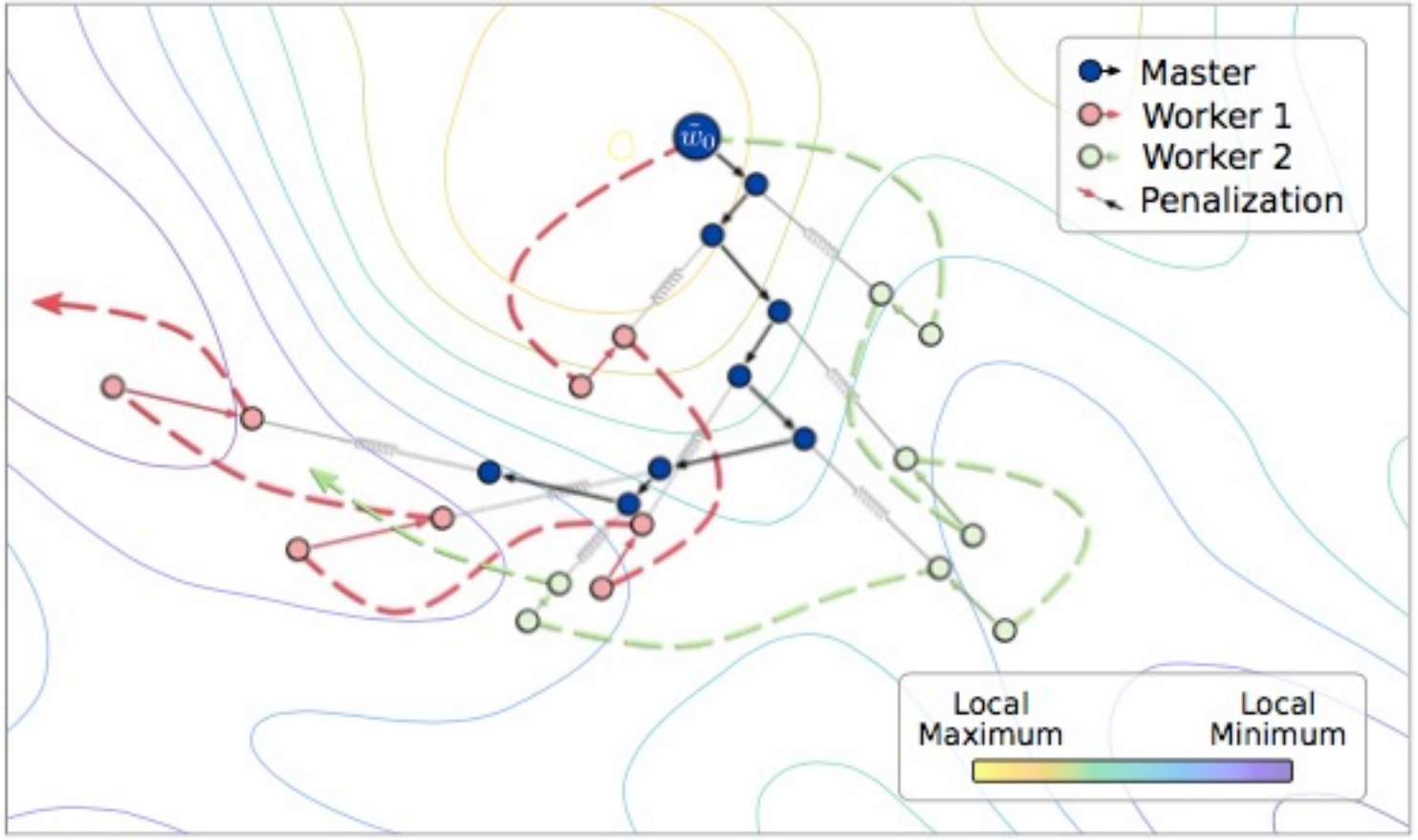
**Require:** training data source $\mathcal{D}^i$, learning rate $\eta$, penalization factor $\alpha$, parameter sharing interval $\tau$, initial model state $\tilde{w}_0$

$w^i \leftarrow \tilde{w}$
**for** $t^i \leftarrow 1, 2, \ldots$ **do**
  $w \leftarrow w^i$
  **if** $t^i \mod \tau = 0$ **then**
    Download $\tilde{w}$ from master
    $\delta^i \leftarrow \alpha(w - \tilde{w})$
    Upload $\delta^i$ to master
    $w^i \leftarrow w^i - \delta^i$
  **end if**
  Sample mini-batch $\mathbf{x} \sim \mathcal{D}^i$
  $w^i \leftarrow w^i - \eta \frac{\partial \mathcal{L}(\mathbf{x}, w)}{\partial w}$
**end for**

The decentralized asynchronous system EASGD [30]

# Decentralized Asynchronous systems (3)

- The individual models (workers & Master) are evolving side-by-side in parallel
- Note that there is no direct interaction between workers
- Stability is maintained by the penalization coefficients (α and β) in combination with the length of isolated learning phases (τ)
- The optimizer hyper-parameters, α, β and τ, are inter-dependent and must be weighted carefully for each training task and cluster setup to constrain how far individual workers can diverge from the master and one another




- The communication demand with the master node scales roughly linear with the number of workers
  - To avoid congestion induced delays due to network I/O bandwidth limitations at the master, τ must be scaled accordingly [11]
  - long phases of isolated training can severely hamper convergence due to the increasing incompatibilities in the models.

# Communication Patterns (CP)

# CP in centralized systems

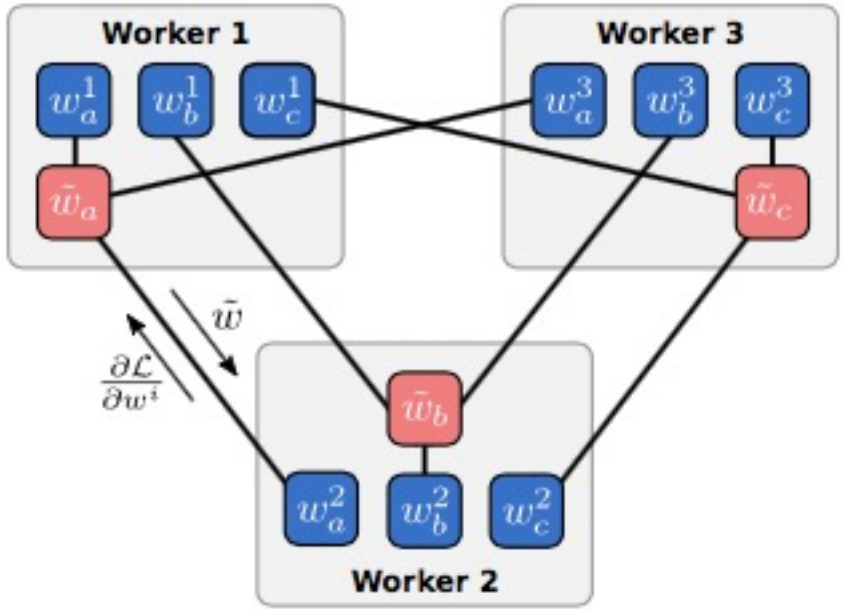
- The parameter server a bottleneck
  - 1 parameter server & n workers: sending & receiving $n \cdot \|w\|$ parameters
- In bulk-synchronous systems
  - Parameter up- and downloads occur sequentially, implying a communication delay of at least $2nT_w + (n-1)R_w + U_w$ in theory, where $T_w$, $R_w$ and $U_w$ respectively denote the time required to *transmit*, *reduce* and *update* $\|w\|$ parameters (i. e. the model)
- Efficient collective communication
  - Binomial tree [15]
    - ✧ Lower bound: $2\lceil \log_2(n+1) \rceil T_w + \lceil \log_2(n) \rceil R_w + U_w$
  - Scatter-reduce/broadcast algorithm [44]
    - ✧ Lower bound: $(2 + 2\frac{n-1}{n})T_w + \frac{n-1}{n}R_w + U_w$
- Asynchronous communication
  - The minimum communication delay per worker: $2T_w + U_w$
  - A full parameter exchange with all workers: $nT_w + U_w$
    - ✧ Parameter exchange requests from individual workers can be overlapped if n > 1

# Parameter server (1)

- If the parameter server is a bottleneck, it is highly desirable to distribute this role [21]
- Most gradient-descent-based optimization algorithms can be executed independently for each model parameter, which permits almost arbitrary slicing
  - In practice this freedom is limited by the overheads incurred from peering with each additional endpoint to complete a parameter exchange [23]
  - Asynchronous systems where congestion-free $k : n$ communication (i.e. between $k$ parameter servers and $n$ workers) is more difficult to realize because the workers operate largely at their own pace [25]
- Additional limitations apply if training depends on hyper-parameter schedules that must be coordinated across parameter servers, or if reproducibility is desired [21]

# Parameter server (2)

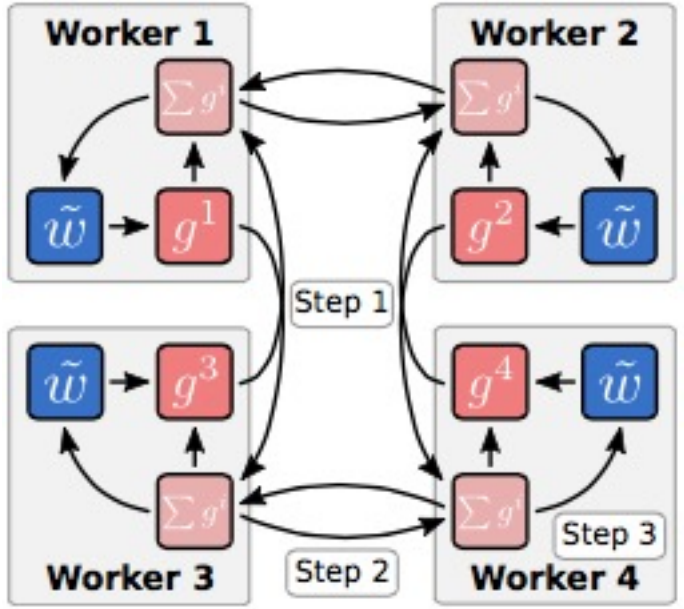
- A popular variant of the multi-parameter-server-approach is to migrate the parameter server role into the worker nodes [17], [19], [24], [27]
- Such that all nodes are workers, but also act as parameter servers (i. e. $k = n$)
- Each worker is responsible for maintaining and updating $1/n$ the global model parameters
- The external communication demand of each node is reduced to $2\frac{n-1}{n}\|w\|$
  - The locally maintained model partition does not have to be exchanged via the network



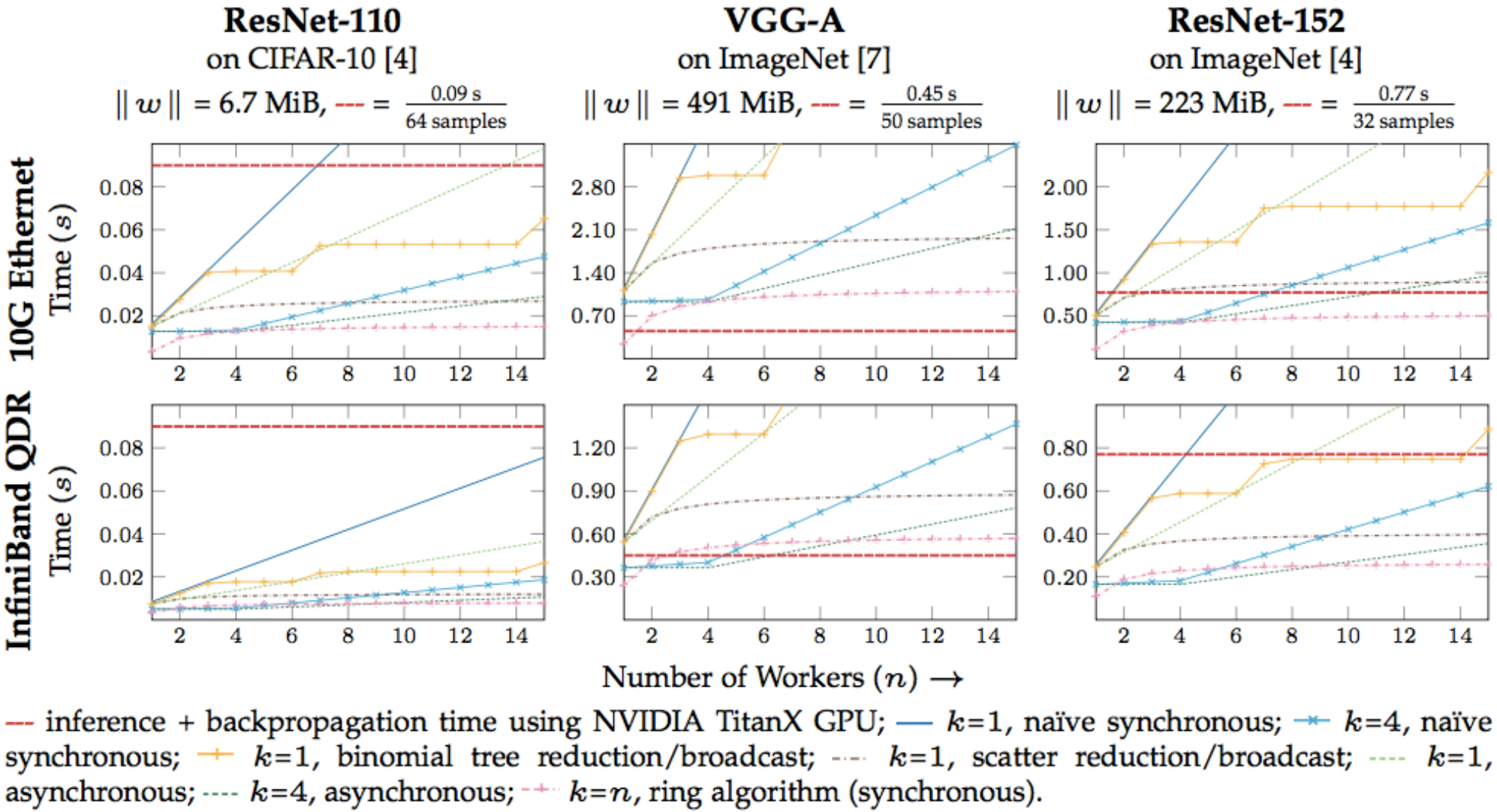

- It can be beneficial in homogeneous cluster setups
- Any node-failure requires a complete reorganization of the cluster [12].

# Parameter server (3)

- The entire parameter server function is implemented in each worker
- The workers synchronously compute gradients, which are shared between machines using
  - a collective *all-reduce* operation
    - ✧ lower bound delay = $\lceil \log_2(n) \rceil (T_w + R_w) + U_w$
  - a ring algorithm
    - ✧ lower bound delay = $2\frac{n-1}{n} T_w + \frac{n-1}{n} R_w + U_w$




- Each machine uses the thereby locally accumulated identical gradients to step an equally parameterized optimizer copy, which in turn applies exactly the same update
- Not only robust to node failures, but also makes adding and removing nodes trivial.

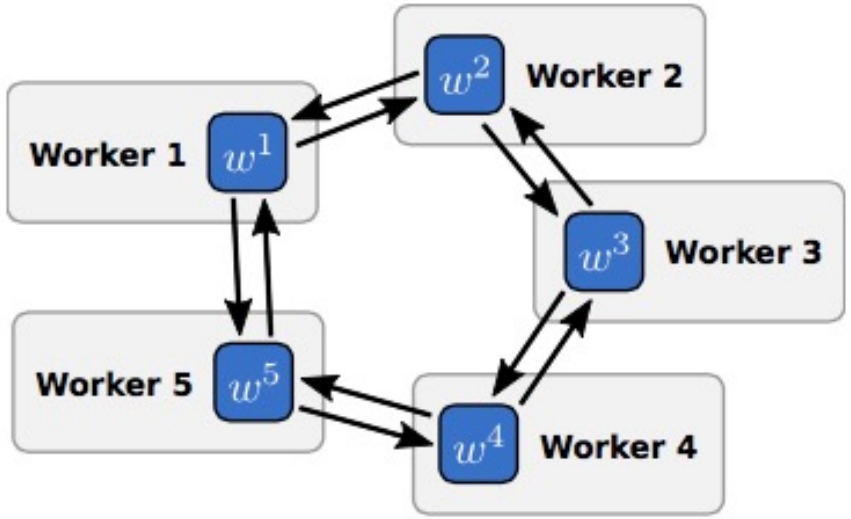



Lower bound communication delays when training various models in different cluster setups in Ethernet and InfiniBand environments, assuming Rw ≈ 10 GiB/s and Uw ≈ 2 GiB/s in an ideal scenario with no latency or competing I/O requests that need arbitration

# CP in decentralized systems

- Assuming isolated training phases of $\tau$ cycles, the communication demand of each decentralized worker per local compute step is only $\frac{1}{\tau}\|w\|$
- Decentralized systems typically maintain a higher computation hardware utilization, even with limited network bandwidth, which can make training large models possible in spite of bandwidth-constraints
- Scaling out to larger cluster sizes may still result in the master node becoming a bottleneck
  - it is possible to split the master's role in decentralized systems to reduce communication costs like in centralized systems
- Each machine is a self-contained independent trainer
  => decentralized DDLS have many options for organizing parameter exchanges

# CP in decentralized systems: D-PSGD [26]




- A ring-like structure
- After each training cycle, each worker sends its model parameters to its neighbors and also integrates the models it receives from them
- The more hops two workers are away from each other, the further they may diverge
- Because the ring is closed, all workers project the same distance-attenuated force on each other which is crucial for stability

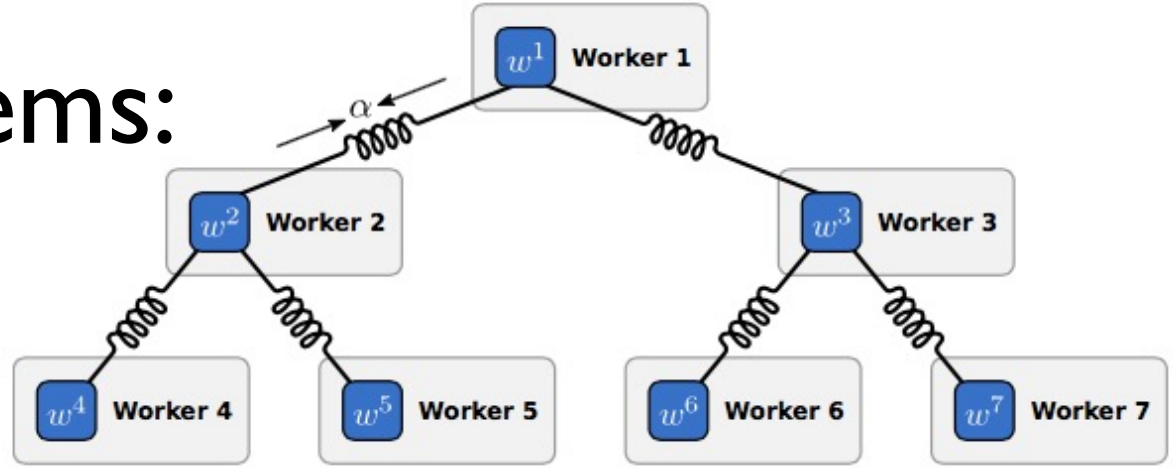
**Require:** training data source $\mathcal{D}^i$, initial model state $\tilde{w}_0$, learning rate $\eta$, neighbor nodes $N^{i-1}$ and $N^{i+1}$, merge weights $\alpha^{i-1,i,i+1}$

1: $w_0^i = \tilde{w}_0$
2: **for** $t \leftarrow 0, 1, 2, \ldots$ **do**
3: Send $w_t^i$ to neighbor nodes $N^{i-1}, N^{i+1}$
4: Sample mini-batch $\mathbf{x} \sim \mathcal{D}^i$
5: $g^i \leftarrow \frac{\partial \mathcal{L}(\mathbf{x}; w_t^i)}{\partial w_t^i}$
6: Await $w_t^{i-1}$ and $w_t^{i+1}$ from neighbor nodes
7: $w_{t+1}^i \leftarrow \frac{1}{\sum_{j=i-1}^{i+1} \alpha^j} \left( \sum_{j=i-1}^{i+1} \alpha^j w_t^j \right) - \eta g^i$
8: **end for**

D-PSGD [26]

# CP in decentralized systems: TreeEASGD[25]

- A hierarchical tree-based communication pattern
  - Avoids bandwidth-related limitations
- Each worker implements two parameter exchange intervals
  - At every $\tau^i_{\text{up}}$ cycles, they share their current model parameters with the respective upstream node
  - At every $\tau^i_{\text{down}}$ cycles, they share their current model parameters with all adjacent downstream nodes
- The degree of exploration is controlled by separately adjusting the up- and downstream parameter exchange frequencies (i.e. $\tau^i_{\text{up}}$ and $\tau^i_{\text{down}}$) for each worker based on its depth in the hierarchy




**Require:** training data $\mathcal{D}^i$, initial model state $\tilde{w}_0$, learning rate $\eta$, penalization factor $\alpha$, parameter exchange intervals $\tau^i_{\text{up}}$ ($\infty$ for root node) and $\tau^i_{\text{down}}$ ($\infty$ for leaf nodes), upstream node $N_{\text{up}}$, downstream nodes $\{N^1_{\text{down}}, \ldots\}$

$w^i = \tilde{w}_0$
**for** $t \leftarrow 1, 2, \ldots$ **do**
  **if** $t \mod \tau^i_{\text{up}} = 0$ **then**
    Send $w^i$ to worker $N_{\text{up}}$
  **end if**
  **if** $t \mod \tau^i_{\text{down}} = 0$ **then**
    **for** $N \leftarrow N^0_{\text{down}}, N^1_{\text{down}}, \ldots$ **do**
      Send $w^i$ to worker $N$
    **end for**
  **end if**
  **if** received some model $w$ from any up- or downstream worker **then**
    $w^i \leftarrow w^i + \alpha(w - w^i)$
  **end if**
  Sample mini-batch $\mathbf{x} \sim \mathcal{D}^i$
  $w^i \leftarrow w^i - \eta \frac{\partial \mathcal{L}(\mathbf{x}; w^i)}{\partial w^i}$
**end for**

TreeEASGD [25]

# CP in decentralized systems: GoSGD[45]

- Workers and can peer with each other to exchange parameters by implementing a sum-weighted gossip protocol
- Each worker ($i$) defines a variable $\alpha^i$ that is initialized equally such that $\sum_i \alpha^i = 1$
- After each local update, a random Bernoulli distribution is sampled to decide whether a parameter exchange should be done
  - The probability ($\mathfrak{p}$) determines the average communication interval
- $\alpha^i$ is halved and sent along with the current model parameters ($w^i$)to the destination worker, which in turn replaces its local model with the weighted average based on its own and the received $\alpha$ value

**Require:** training data source $\mathcal{D}^i$, initial model state $\tilde{w}_0$, learning rate $\eta$, number of cluster nodes $n$, parameter exchange probability $p$

1: $\alpha^i = \frac{1}{n}$
2: $w^i = \tilde{w}_0$
3: **loop**
4: **if** received parameters $\langle w^j, \alpha^j \rangle$ from worker $j$ **then**
5: $w^i \leftarrow \frac{\alpha^i}{\alpha^i+\alpha^j} w^i + \frac{\alpha^j}{\alpha^i+\alpha^j} w^j$
6: $\alpha^i \leftarrow \alpha^i + \alpha^j$
7: **end if**
8: Sample mini-batch $\mathbf{x} \sim \mathcal{D}^i$
9: $w^i \leftarrow w^i - \eta \frac{\partial \mathcal{L}(\mathbf{x}; w^i)}{\partial w^i}$
10: **if** $s \sim$ Bernoulli$(p) = 1$ **then**
11: $\alpha^i \leftarrow \frac{\alpha^i}{2}$
12: Send $\langle w^i, \alpha^i \rangle$ to randomly chosen worker.
13: **end if**
14: **end loop**

GoSGD [45]

- Workers that shared their state recently are weighted down in relevance because the information they collected about the loss function has become more common knowledge (=*gossip*) among other workers
- The variance of staleness within the cluster is minimized
- By setting the learning rate ($\eta$) to zero, all workers asymptotically approximate a consensus model.

# DDLS tools (1)

- *DistBelief* (Google; [21])
  - A centralized asynchronous DDLS with support for multiple parameter servers
- *Project Adam* (Microsoft; [23])
  - Took a similar approach by moving gradient computation steps into the parameter servers for some neural network layers
  - Organize the parameter servers in a Paxos cluster to establish high availability
- *Petuum* [16]
  - Imposing delay bounds to control the staleness of asynchronous workers can improve the rate of convergence
- *Parameter Server* [14]
  - Formalizing the processing of deep learning workloads to establish hybrid parallelism and integrate with a general machine learning architecture
- *TensorFlow* (Google; [17]) and *MXNet* (Apache Foundation; [19])
  - Modern descendants of DistBelief that improve upon previous approaches by introducing new concepts, such as defining backup workers
  - Optimizing model partitioning using self-tuning heuristic models, and improving scalability by allowing hierarchical parameter servers to be configured
- *CaffeOnSpark* (Yahoo; [24]) and *BigDL* (Intel; [27])
  - The opposite approach and focus on easy integration with existing data analytics systems and commodity hardware environments by implementing centralized synchronous data-parallel model training on top of Apache Spark
  - Accommodate the frequent communication needs of such systems, they use sophisticated communication patterns to implement a distributed parameter server

# DDLS tools (2)

- *SparkNet* [10],
  - A decentralized synchronous DDLS that replicates Caffe-solvers using Apache Spark's map-reduce API to realize training in commodity cluster environments
  - As part of the popular Java DDLS *deeplearning4j*
  - The restriction to synchronous execution is often considered as a major downside by this approach
- *MPCA-SGD* [11]
  - Improve upon SparkNet
  - Extend the basic Spark-based approach by overlapping computation and communication to realize quasi-asynchronous training and extrapolates the recently observed descent trajectory to cope with staleness
- The data-parallel optimizer of *PyTorch* (Facebook; [47])
  - Implement a custom interface to realize synchronous model training using collective communication primitives
  - Either one or all workers act as parameter server (all-reduce approach)
- *EASGD* [30]
  - Retain the idea of limited isolated training phases but imposes fully asynchronous scheduling
  - Having a single master node can become a bottleneck as the cluster grows larger
- *D-PSGD* [26], *TreeEASGD* [25] and *GoSGD* [45]
  - Approaches to further scale out decentralized optimization by distributing the master function
- *COTS HPC* [9] and *FireCaffe* [15]
  - Optimized for HPC and GPU supercomputer environments, where they have been shown to achieve unparalleled performance for certain applications.

| *DDLS Name* (a-z) | *Parallelism* (Model / Data) | *Optimization* | *Scheduling* | *Parameter Exchange* | *Topology* | *Remarks* |
|---|---|---|---|---|---|---|
| **BigDL** [27] | DP only | central | sync. | scatter-red. | distributed PS (always $k = n$) | Each worker acts as a parameter server for $\frac{1}{n}$ of the model (cf. Section 3.4.1). Distributed parameter exchanges are realized via the Spark block manager. |
| **CaffeOnSpark** [24] | DP only | central | sync. | scatter-red. | distributed PS (always $k = n$) | Parameter exchange realized via RDMA using repeated invocations of MPI functions. Equivalent implementations are available for **Caffe2** and **Chainer**. |
| **COTS HPC** [9] | MP only | central | sync. | – | distrib. array abstraction | Model layers partitioned along tensor dimensions and distributed across cluster. Fine-grained access is managed via a low-level array abstraction. |
| **D-PSGD** [26] | DP only | decentral | sync. | 2:1 reduce | closed ring | Each node exchanges parameters with only its neighbors on the ring (cf. Section 3.4.2). |
| **DistBelief** [21] | MP + DP | central | async. | ad hoc | distrib. PS | Model partitions spread across dedicated parameter server nodes (cf. Section 3.4.1). |
| **EASGD** [30] | DP only | decentral | async. | ad hoc | single master | Decentralized asynchronous system as discussed in Section 3.3.5. Reactive adjustment of hyper-parameters can speedup training [42]. |
| **FireCaffe** [15] | DP only | central | sync. | binom. tree | single PS | Simplistic centralized synchronous system as discussed in Section 3.3.1. |
| **GoSGD** [45] | DP only | decentral | soft-bounded async. | ad hoc | p2p mesh | No dedicated master node. Parameter exchanges between any two workers realized via sum-weighted randomized gossip protocol as discussed in Section 3.4.2. |
| **MPCA-SGD** [11] | DP only | decentral | soft-bounded async. | binom. tree | dedicated master node | Model updating and sharing updates are decoupled. Penalization occurs as a part of the model's cost function. Staleness effects are dampened using an extrapolation mechanism. |
| **MXNet** [19] | MP + DP | central | bounded async. | scatter-reduce *async.*: ad hoc | distributed PS (default $k = n$) | Supports various advanced parameter server configurations, including but not limited to hierarchical multi-stage proxy servers (cf. Section 3.4.1). |
| **Parameter Server** [14] | MP + DP | central | bounded async. | reduce *async.*: ad hoc | distrib. PS | Model partitions spread redundantly across parameter server group. Workers organized in model parallelism enabled groups. One worker per group can act as a proxy server. |
| **Petuum** [16] | MP + DP | central | bounded async. | ad hoc with eager scatter | distrib. PS | Pioneered the use of delay bounds to control staleness (cf. Section 3.3.4). Average model staleness is further reduced through the eager distribution of model parameters. |
| **Project Adam** [23] | MP + DP | central | async. | ad hoc | distrib. PS | Dedicated parameter server group that is managed as a Paxos cluster. Hybrid parallelism realized through transferring gradient computation for fully connected layers into PS. |
| **PyTorch** [47] | MP + DP | central | sync. | all-reduce | single PS or replicated PS | Model parallelism capabilities were added recently with version 1.4.0. Can only use either synchronous data-parallelism or model parallelism. |
| **SparkNet** [10] | DP only | decentral | sync. | reduce | dedicated master node | Decentralized synchronous implementation as discussed in Section 3.3.2. Realized using Spark map-reduce. Production-grade re-implementation present in **deeplearning4j**. |
| **TensorFlow** [17] | MP + DP | central | bounded async. | scatter/all-red. *async.*: ad hoc | distributed PS (default $k = n$) | Supports single and multi parameter server setups, as well as all-reduce-based approaches. By default, each worker acts as a parameter server for a portion of the model. |
| **TreeEASGD** [25] | DP only | decentral | bounded async. | ad hoc | tree | All nodes are workers and form a tree. Each worker only exchanges parameters with its immediate up- and downstream neighbors (cf. Section 3.4.2). |

# Parallelism

- DP is more frequently supported than MP
  - Decentral optimization is based on the concept of sparse communication between independent trainers; Realizing cross-machine MP in such systems is counter-intuitive
  - New modeling and training techniques [2][4] allow utilizing the available parameter space more efficiently, while technological improvements in hardware allow processing increasingly larger models
  - Not every model can be partitioned evenly across a given number of machines, which leads to the under-utilization of workers [20]
- If a model fits well into the GPU memory, the resource requirements of the backpropagation algorithm can often be regulated reasonably well by adjusting the mini-batch size in DP systems,
  - Some DDLS discourage using cross-machine MP in favor of DP, which is less susceptible to processing time variations

# Optimization

- A trend towards decentralized systems in research
- Centralized DDLS
  - Minor improvements, such as tailored optimization techniques [2], [36], [39]
  - The development of domain-specific compression methods [13]
- Centralized DDLS dominate industry usage and application research, although centralized and decentralized DDLS offer similar convergence guarantees [26]
  - Centralized approaches are generally better understood and easier to use
  - Most popular and industry- backed deep learning frameworks (PyTorch, TensorFlow, MXNet, etc.) contain centralized DDLS implementations that are mature, highly optimized and work tremendously well as long as parameter exchanges do not dominate the overall execution [11], [48]

# Scheduling

- Centralized asynchronous methods
  - Cope better with performance deviations and have the potential to yield a higher hardware utilization
  - But introduce new challenges such as concurrent updates and staleness => some DDLS support synchronous and asynchronous modes of operation
- Centralized bounded asynchronous DDLS can always simulate synchronous and asynchronous scheduling
  - If a delay bound is used, $s = 0$ is identical to synchronous, while $s = \infty$ results in fully asynchronous behavior
- Decentralized DDLS
  - Some decentralized DDLS define a simple threshold ($\tau$) to limit the amount of exploration per training phase
  - Others take a more dynamic approach to cope better with bandwidth limitations, which is indicated using the term *soft-bounded*

# Parameter Exchange mechanism

- Binomial tree methods
  - Scale worse than scattering operations, but are preferable in high latency environments because less individual connections between nodes are required [44]
- Collective operation
  - Common, but not necessarily the only parameter exchange method available
  - Some synchronous DDLS implement several collective operations and switch between them to maximize efficiency

# Topology

- Centralized DDLS
  - The current state-of-the-art in centralized DDLS for small clusters is the synchronous all-reduce-based approach
  - Large and heterogeneous setups can be utilized efficiently using hierarchically structured asynchronous communication patterns [19]
- Decentralized DDLS
  - Heavily structured communication protocols [25], [26], boosting techniques [11], as well as relatively unstructured methods [45] have been reported to offer better convergence rates than naïve implementations

# Right technique

- The non-linear non-convex nature of deep learning models in combination with the abundance of distributed methods opens up a large solution space [2]
- Although frequently done, comparing DDLS based on processing metrics such as GPU utilization or training sample throughput is not useful in practice
  - Such performance indicators can easily be maximized by increasing the batch-size, allowing more staleness or extending exploration phases, which does not necessarily equate to faster training or yield a better model
- Benchmark is not enough
  - Well-established deep learning benchmarks like DAWN- Bench [48] propose comparing the end-to-end training performance by measuring quality metrics (e.g. time to ac- curacy x%)
  - However, optimal configurations w.r.t. quality metrics are usually highly task dependent and may vary as the training progresses [42]

# Benchmark tools

- The collection and quantitative study of the performance of DDLS using standardized AI benchmarks is becoming increasingly important and can provide guidance regarding what configurations work well in practice
- DAWNBench [48]
  - Strong emphasis on distributed implementations, but focuses only on a few workloads
- MLPerf [49]
  - Expand the scope and defines stricter test protocols to establish better comparability
- Deep500 [50]: a new benchmark tool
  - Focus on gathering more information by defining metrics and measurement points along the training pipeline
- AIBench [51]
  - Aim at covering many machine learning applications like recommendation systems, speech recognition, image generation, image compression, text-to-text translation, etc.

# Criteria of DDLS

| | $RLBD_1$ | $RBM_2$ | $OptLR_3$ | $COCC_4$ | $CAHP_5$ | $RSOI_6$ | $ERB_7$ |
|---|---|---|---|---|---|---|---|
| **MP** | | ✓ | higher | + | +++ | + | trivial |
| **MP + mini-batch pipelining** | | ✓ | lower | ++ | ++++ | ++ | medium |
| **DP + central + synchronous** | | | higher | + | + | + | easy |
| **DP + central + asynchronous** | ✓ | | lower | +++ | ++ | +++ | hard |
| **DP + decentral + synchronous** | ✓ | | likely lower | + | ++ | ++ | easy |
| **DP + decentral + asynchronous** | ✓ | | | ++ | +++ | +++ | hard |

1 **R**equires **L**arge **B**alanced **D**ataset; 2 **R**equires **B**alanced **M**odel; 3 **Opt**imal **L**earning **R**ate as cluster grows; 4 **C**omplexity due to **O**verlapping **C**omputation and **C**ommunication (e. g. growing staleness with cluster size); 5 **C**omplexity due to **A**dditional **H**yper-**P**arameters (number, stability, entanglement, etc.); 6 **R**esilience to **S**poradic **O**utside **I**nfluences; 7 difficulty to **E**stablish **R**eproducible **B**ehavior

# Future research directions

- Using decentralized optimization techniques in conjunction with P2P model sharing [45]
  - An interesting area of research for certain IoTs or automotive applications
- A comprehensive analysis of different distributed approaches in real-life scenarios would be helpful to many practitioners
  - An actual cluster setups the situation is usually more complex due to competing workloads
  - Most works in distributed deep learning restrict themselves to ideal test scenarios
- Efficiently realizing distributed training in heterogeneous setups is a largely un-tackled engineering problem
  - An investment commodity, clusters are often not replaced, but rather extended
- A structured quantitative analysis of the results from DDLS benchmarks could be interesting for many practitioners

# Reference (1)


1. Y. LeCun, Y. Bengio, and G. E. Hinton, “Deep Learning,” *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.
2. J. Schmidhuber, “Deep Learning in Neural Networks: An Overview,” *Neural Networks*, vol. 61, pp. 85–117, 2015.
3. A. Krizhevsky, I. Sutskever, and G. E. Hinton, “ImageNet Classifi- cation with Deep Convolutional Neural Networks,” *Adv. in Neural Information Processing Systems*, vol. 25, pp. 1097–1105, 2012.
4. K. He, X. Zhang, S. Ren, and J. Sun, “Deep Residual Learning for Image Recognition,” *Proc. IEEE Conf. on Computer Vision and Pattern Recognition*, pp. 770–778, 2016.
5. S. Shi, Q. Wang, P. Xu *et al.*, “Benchmarking State-of-the-Art Deep Learning Software Tools,” *arXiv CoRR*, vol. abs/1608.07249, 2016.
6. N. Shazeer, A. Mirhoseini, K. Maziarz *et al.*, “Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer,” *Proc. 5th Intl. Conf. on Learning Representations*, 2017.
7. K. Simonyan and A. Zisserman, “Very Deep Convolutional Net- works for Large-Scale Image Recoginition,” *Proc. 3rd Intl. Conf. on Learning Representations*, 2015.
8. M. Langer, “Distributed Deep Learning in Bandwidth- Constrained Environments,” Ph.D. dissertation, La Trobe University, 2018.
9. A. Coates, B. Huval, T. Wang, D. J. Wu, B. C. Catanzaro, and A. Y. Ng, “Deep Learning with COTS HPC Systems,” *Proc. 30th Intl. Conf. on Machine Learning*, pp. 1337–1345, 2013.
10. P. Moritz, R. Nishihara *et al.*, “SparkNet: Training Deep Networks in Spark,” *Proc. 4th Intl. Conf. on Learning Representations*, 2016.
11. M. Langer, A. Hall *et al.*, “MPCA SGD - A Method for Distributed Training of Deep Learning Models on Spark,” *IEEE Trans. on Parallel and Distributed Systems*, vol. 29, no. 11, pp. 2540–2556, 2018.
12. K. Zhang, S. Alqahtani, and M. Demirbas, “A Comparison of Distributed Machine Learning Platforms,” *Proc. 26th Intl. Conf. on Computer Communications and Networks*, 2017.
13. T.Ben-NunandT.Hoefler,“DemystifyingParallelandDistributed Deep Learning: An In-Depth Concurrency Analysis,” *ACM Com- put. Surv.*, vol. 52, no. 4, 2019.

# Reference (2)


14. M.Li,D.G.Andersen,A.J.Smola*etal.*,"CommunicationEfficient Distributed Machine Learning with the Parameter Server," *Adv. in Neural Information Processing Systems*, vol. 27, pp. 19–27, 2014.
15. F.N.Iandola,K.Ashraf*etal.*,"FireCaffe:Near-LinearAcceleration of Deep Neural Network Training on Compute Clusters," *Proc. IEEE Conf. on Computer Vision and Pattern Recognition*, 2015.
16. E. P. Xing, Q. Ho, W. Dai, J. K. Kim, J. Wei, S. Lee *et al.*, "Petuum - A New Platform for Distributed Machine Learning on Big Data," *IEEE Trans. on Big Data*, vol. 1, no. 2, pp. 49–67, 2015.
17. M. Abadi, P. Barham, J. Chen *et al.*, "TensorFlow: A System for Large-Scale Machine Learning," *Proc. 12th USENIX Symp. on Operating Systems Design and Implementation*, pp. 265–283, 2016.
18. Q. V. Le, "Building High-Level Features Using Large Scale Unsu- pervised Learning," *Proc. IEEE Intl. Conf. on Acoustics, Speech and Signal Processing*, pp. 8595–8598, 2013.
19. T. Chen, M. Li, Y. Li *et al.*, "MXNet: A Flexible and Efficient Ma- chine Learning Library for Heterogeneous Distributed Systems," *Proc. 29th Conf. on Neural Information Processing Systems*, 2015.
20. Y. Huang, Y. Cheng, A. Bapna, O. Firat, M. X. Chen *et al.*, "GPipe: Efficient Training of Giant Neural Networks using Pipeline Paral- lelism," *arXiv CoRR*, vol. abs/1811.06965, 2018.
21. J. Dean, G. S. Corrado, R. Monga, K. Chen, M. Devin *et al.*, "Large Scale Distributed Deep Networks," *Adv. in Neural Information Processing Systems*, pp. 1223–1231, 2012.
22. S. Ioffe and C. Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift," *Proc. 32nd Intl. Conf. on Machine Learning*, vol. 37, pp. 448–456, 2015.
23. T. Chilimbi, Y. Suzue *et al.*, "Project Adam: Building an Efficient and Scalable Deep Learning Training System," *Proc. 11th USENIX Symp. on OS Design and Implementation*, pp. 571–582, 2014.

# Reference (3)


24. A. Feng, J. Shi, and M. Jain, “CaffeOnSpark Open Sourced for Distributed Deep Learning on Big Data Clusters,” 2016. [Online]. Available: http://yahoohadoop.tumblr.com/post/139916563586/ caffeonspark- open- sourced- for- distributed- deep.
25. S. Zhang, “Distributed Stochastic Optimization for Deep Learn- ing,” Ph.D. dissertation, New York University, 2016.
26. X.Lian,C.Zhang,H.Zhang,C.-J.Hsieh*etal.*,“CanDecentralized Algorithms Outperform Centralized Algorithms? A Case Study for Decentralized Parallel Stochastic Gradient Descent,” *Adv. in Neural Information Processing Systems*, vol. 30, pp. 5330–5340, 2017.
27. J. Dai, Y. Wang, X. Qiu, D. Ding, Y. Zhang *et al.*, “BigDL: A Distributed Deep Learning Framework for Big Data,” *Proc. ACM Symposium on Cloud Computing*, pp. 50–60, 2019.
28. I. J. Goodfellow, O. Vinyals, and A. M. Saxe, “Qualitatively Char- acterizing Neural Network Optimization Problems,” *Proc. 3rd Intl. Conf. on Learning Representations*, 2015.
29. H. R. Feyzmahdavian, A. Aytekin *et al.*, “An Asynchronous Mini- Batch Algorithm for Regularized Stochastic Optimization,” *Proc. 54th IEEE Conf. on Decision and Control*, pp. 1384–1389, 2015.
30. S. Zhang, A. Choromanska, and Y. LeCun, “Deep learning with Elastic Averaging SGD,” *Adv. in Neural Information Processing Systems*, vol. 28, pp. 685–693, 2015.
31. N. S. Keskar, D. Mudigere, J. Nocedal *et al.*, “On Large-Batch Train- ing for Deep Learning: Generalization Gap and Sharp Minima,” *Proc. 5th Intl. Conf. on Learning Representations*, 2017.
32. I. Sutskever, J. Martens, G. Dahl *et al.*, “On the Importance of Initialization and Momentum in Deep Learning,” *Proc. 30th Intl. Conf. on Machine Learning*, vol. 28, no. 3, pp. 1139–1147, 2013.
33. D. Kingma and J. Ba, “Adam: A Method for Stochastic Optimiza- tion,” *Proc. 3rd Intl. Conf. on Learning Representations*, 2015.
34. J. Chen, X. Pan *et al.*, “Revisiting Distributed Synchronous SGD,” *Proc. 5th Intl. Conf. on Learning Representations*, 2017.
35. Q.Ho,J.Cipar,H.Cui,J.K.Kim*etal.*,“MoreEffectiveDistributed ML via a Stale Synchronous Parallel Parameter Server,” *Adv. in Neural Information Processing Systems*, vol. 26, pp. 1223–1231, 2013.

# Reference (4)


36. R. Tandon, Q. Lei, A. Dimakis, and N. Karampatziakis, "Gradient Coding: Avoiding Stragglers in Distributed Learning," *Proc. 34th Intl. Conf. on Machine Learning*, vol. 70, pp. 3368–3376, 2017.
37. A. Agarwal and J. C. Duchi, "Distributed Delayed Stochastic Op- timization," *Adv. in Neural Information Processing Systems*, vol. 24, pp. 873–881, 2011.
38. W. Dai, A. Kumar, J. Wei, Q. Ho *et al.*, "High-Performance Distributed ML at Scale Through Parameter Server Consistency Models," *Proc. 29th Conf. on Artificial Intelligence*, pp. 79–87, 2015.
39. I. Mitliagkas, C. Zhang *et al.*, "Asynchrony Begets Momentum, With an Application to Deep Learning," *Proc. 54th Allerton Conf. on Communication, Control, and Computing*, pp. 997–1004, 2017.
40. S.Zheng,Q.Meng,T.Wang,W.Chen,N.Yu*etal.*,"Asynchronous Stochastic Gradient Descent with Delay Compensation," *Proc. 34th Intl. Conf. on Machine Learning*, pp. 4120–4129, 2017.
41. F. Niu, B. Recht, C. Re´, and S. J. Wright, "Hogwild!: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent," *Adv. in Neural Information Processing Systems*, vol. 24, pp. 693–701, 2011.
42. H. Kim, J. Park, J. Jang, and S. Yoon, "DeepSpark: Spark-Based Deep Learning Supporting Asynchronous Updates and Caffe Compatibility," *arXiv CoRR*, vol. abs/1602.08191, 2016.
43. X. Lian, Y. Huang, Y. Li, and J. Liu, "Asynchronous Parallel Stochastic Gradient for Nonconvex Optimization," *Adv. in Neural Information Processing Systems*, vol. 28, pp. 2737–2745, 2015.
44. R. Thakur, R. Rabenseifner, and W. Gropp, "Optimization of Collective Communication Operations in MPICH," *Intl. Jrnl. of High Performance Computing Applications*, vol. 19, pp. 49–66, 2005.
45. M. Blot, D. Picard, M. Cord, and N. Thome, "Gossip Training for Deep Learning," *arXiv CoRR*, vol. abs/1611.09726, 2016.
46. S. Boyd, A. Ghosh *et al.*, "Randomized Gossip Algorithms," *IEEE Trans. on Information Theory*, vol. 52, no. 6, pp. 2508–2530, 2006.
47. N. Ketkar, "Deep Learning with Python: A Hands-on Introduc- tion," *ISBN: 978-1-4842-2766-4*, pp. 195–208, 2017.

# Reference (5)


48. C. Coleman, D. Narayanan, D. Kang, T. Zhao *et al.*, “DAWNBench: An End-to-End Deep Learning Benchmark and Competition,” *NIPS ML Systems Workshop*, 2017.
49. P. Mattson, C. Cheng, C. Coleman *et al.*, “MLPerf Training Bench- mark,” *arXiv CoRR*, vol. abs/1910.01500, 2019.
50. T. Ben-Nun, M. Besta, S. Huber, A. N. Ziogas, D. Peter, and T. Hoefler, “A Modular Benchmarking Infrastructure for High- Performance and Reproducible Deep Learning,” *Proc. 33rd Intl. Parallel and Distributed Processing Symposium*, pp. 66–77, 2019.
51. W. Gao, F. Tang, L. Wang, J. Zhan, C. Lan, C. Luo, Y. Huang, C. Zheng *et al.*, “AIBench: An Industry Standard Internet Service AI Benchmark Suite,” *arXiv CoRR*, vol. abs/1908.08998, 2019.