EURECOM
Sophia Antipolis

MALCOM

Machine Learning for Communication Systems

Lecture 10
Communication-efficient Edge AI/ML & Federated Learning

Jaiprakash Nagar
(Slides: Marios Kountouris)

Spring 2024

Communication-Efficient Edge AI Taxonomy

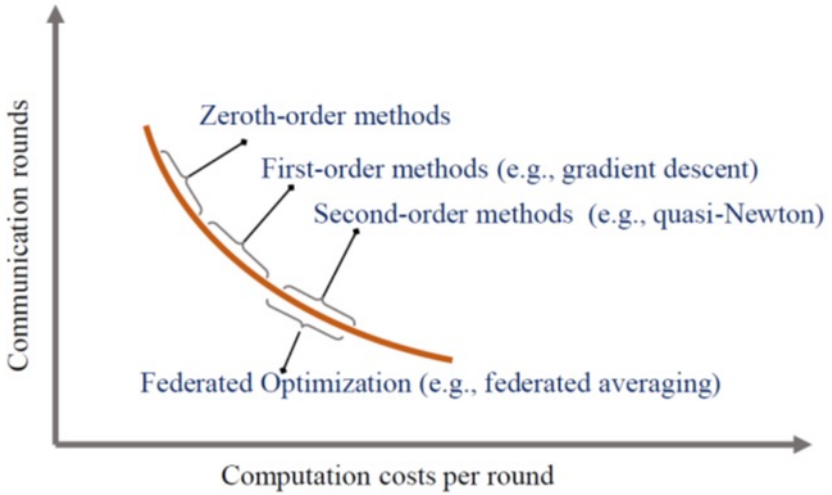
Communication-efficient Edge AI

Algorithmic level

- Zeroth order
- First order
- Second order
- Federated Optimization

System level

- Data partition based edge training
- Model partition based edge training
- Computation Offloading based edge inference
- Coded edge computing



Communication vs. Computation

$$Latency = \underbrace{\frac{D}{R} \cdot N}_{\text{communication}} + \underbrace{T \cdot N}_{\text{computation}}$$

D: total size of info per round, *R*: communication rate,
N: # comm. rounds, *T*: total computation time

Communication-efficient Algorithms

Zeroth-order Methods

- *Derivative-free* methods - only the function value is available
- Explicit gradient calculations may be computationally infeasible, expensive, or impossible.
- Classical techniques: use function difference information to approximate gradients (e.g. Kiefer-Wolfowitz type procedures)
  - Distributed ML: uplink transmission of function value scalar
  - RL: policy function learning without building a model
  - DNN: black box adversarial attacks.

- Optimization problem: $\min_{\theta \in \Theta} f(\theta) = \mathbb{E}_P[F(\theta; X)] = \int_{\mathcal{X}} F(\theta; X) dP(x)$

  where $\Theta \subseteq \mathbb{R}^d$ is a compact convex set, P is a distribution over the space $\mathcal{X}$, and for P - almost every $x \in \mathcal{X}$, the function $F(\cdot; x)$ is closed and convex

- Baseline approximation: for small non-zero scalar u and a vector $Z \in \mathbb{R}^d$, the directional derivative of $F(\theta; x)$ at point θ in the direction Z is approximated by

$$(F(\theta + uZ; x) - F(\theta; x))/u$$

Communication-efficient Algorithms

Zeroth-order Methods

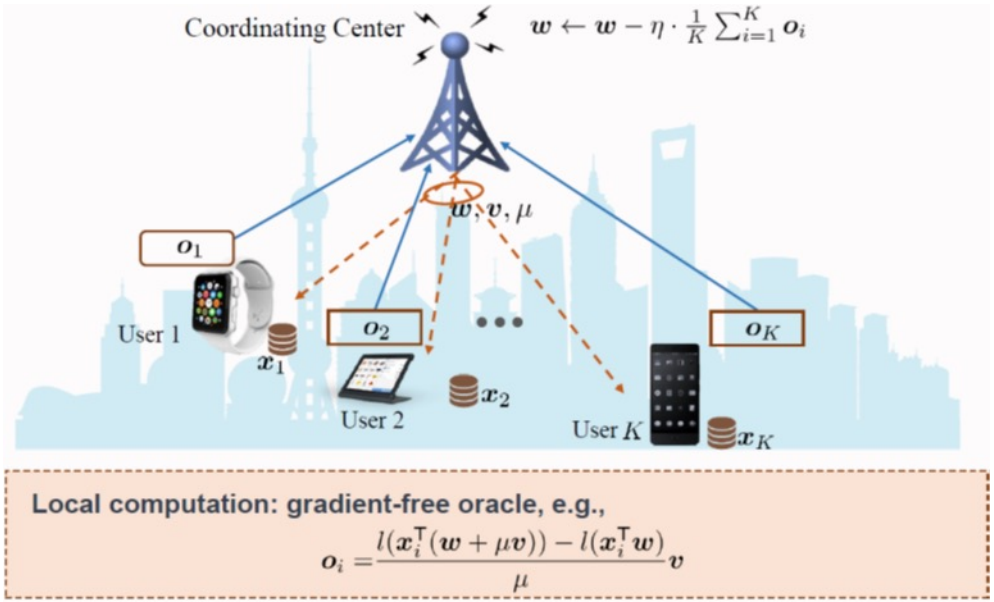
- Convergence rates: suffers a factor of at most $\sqrt{d}$ in the convergence rate over traditional stochastic gradient methods for d-dimensional convex optimization problems.

Distributed zeroth-order methods

- Objective: reduce number of per-device communications

- Some approaches:
  - At each iteration each device communicates with its neighbors with some probability (independent from others and the past), which decays to zero at a carefully tuned rate.

  - Exchange only quantized data information (even 1 bit)



Communication-efficient Algorithms

First-order Methods

- Most commonly used algorithms - mainly based on gradient descent methods
- Computational complexity per iteration scales with # data samples and model dimensions
- SGD: only one training sample is used to compute the gradient at each iteration.

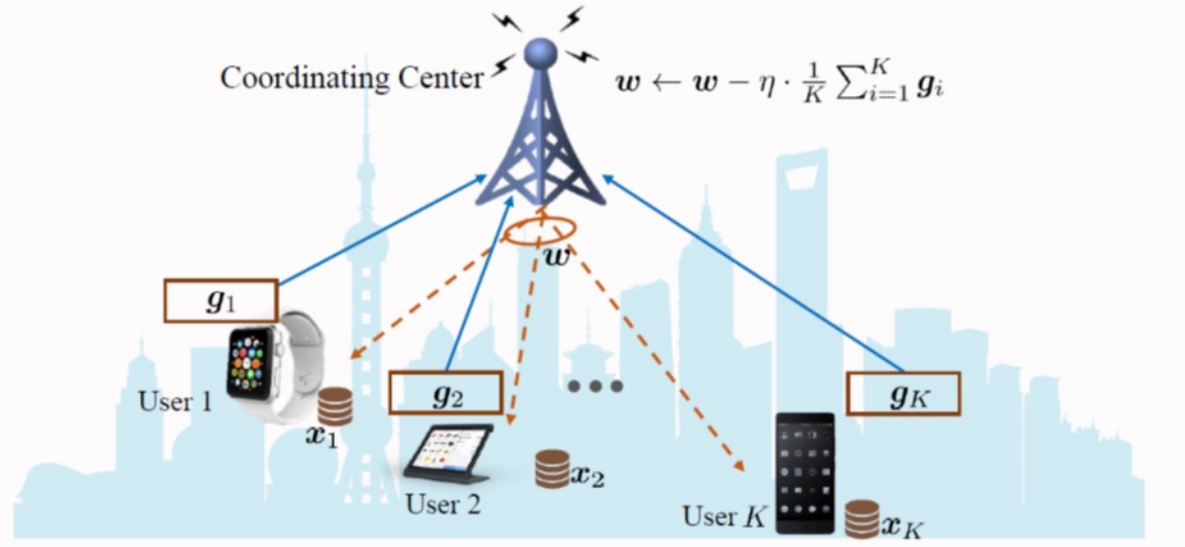
Two main approaches:

- *Communication round minimization*

  by accelerating the convergence rate of the learning algorithms

- *Communication overhead per round reduction*

  (e.g., gradient reuse, quantization, sparsification, sketching based compression)



Local computation: local gradient

$$g_i = l'(x_i^T w) x_i$$

EURECOM Sophia Antipolis

First-order Methods: Taxonomy of Distributed SGD

Four Dimensions

Communication Synchronization

- Synchronous
- Stale Synchronous
- Asynchronous
- Local SGD

Comp.- Comm. Parallelism

- Pipelining
- Scheduling

Distributed SGD

System Architectures

- Parameter Server
- All Reduce
- Decentralized/Gossip

Reduction techniques

- Quantization
- Sparsification
- Coding
- Compression

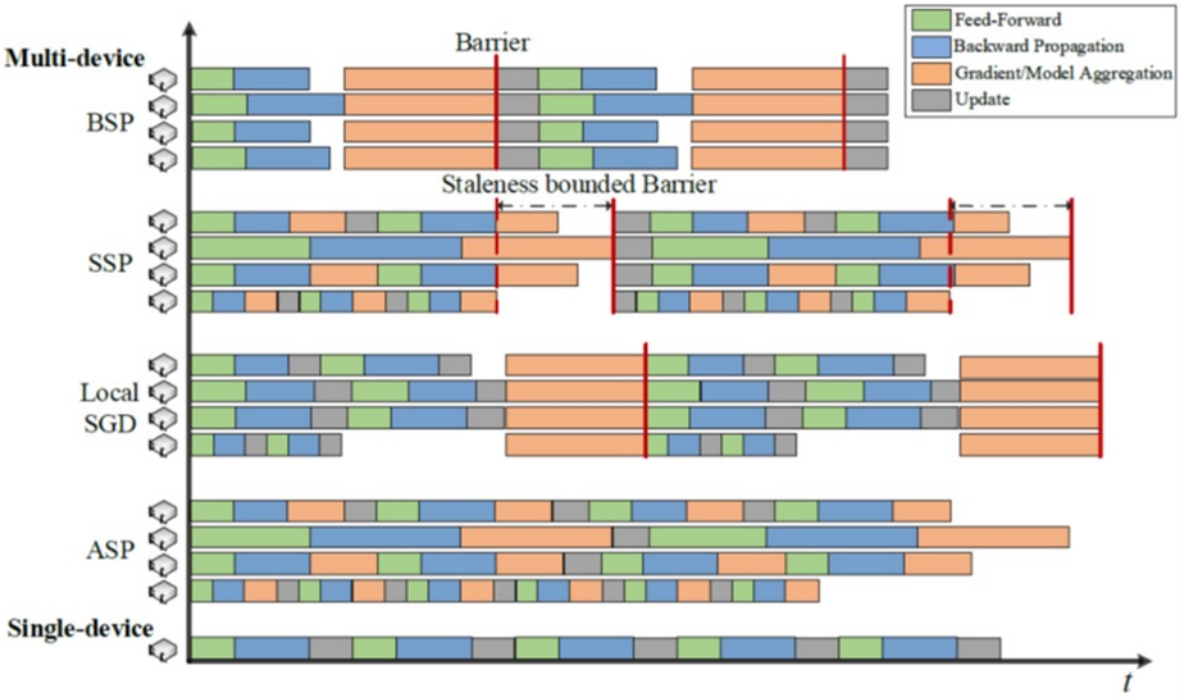
DistSGD: Communication Synchronization

- "Best" split depends on # workers and how frequently each worker needs to work with others

- **Synchronous:** every worker wait for all workers to finish transmitting all parameters in the current iteration, before the next training.

- **Stale Synchronous:** faster workers do more updates than slower ones. *staleness bounded barrier* to guarantee convergence: limits the iteration gap between fastest and slowest worker.

- **Asynchronous:** each worker transmits its gradient as soon as it's calculated. The PS updates the global model without waiting for the other workers.



Synch

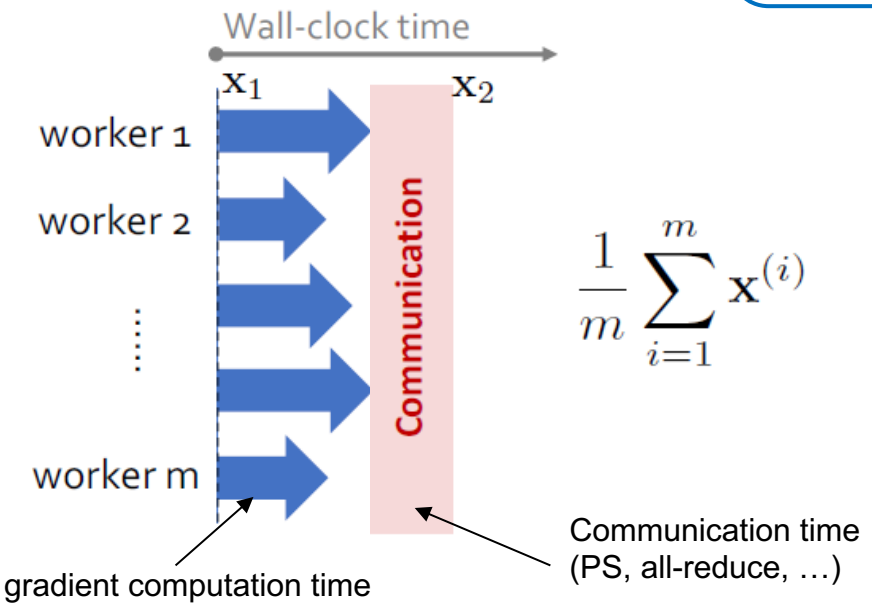
- **Local SGD:** each worker runs several or more iterations itself, and then PS averages all local models into the newest

Fully Synchronous SGD

1. Local stochastic gradients computation
2. Average local models across all nodes

Gradient at k-th iteration and i-th worker

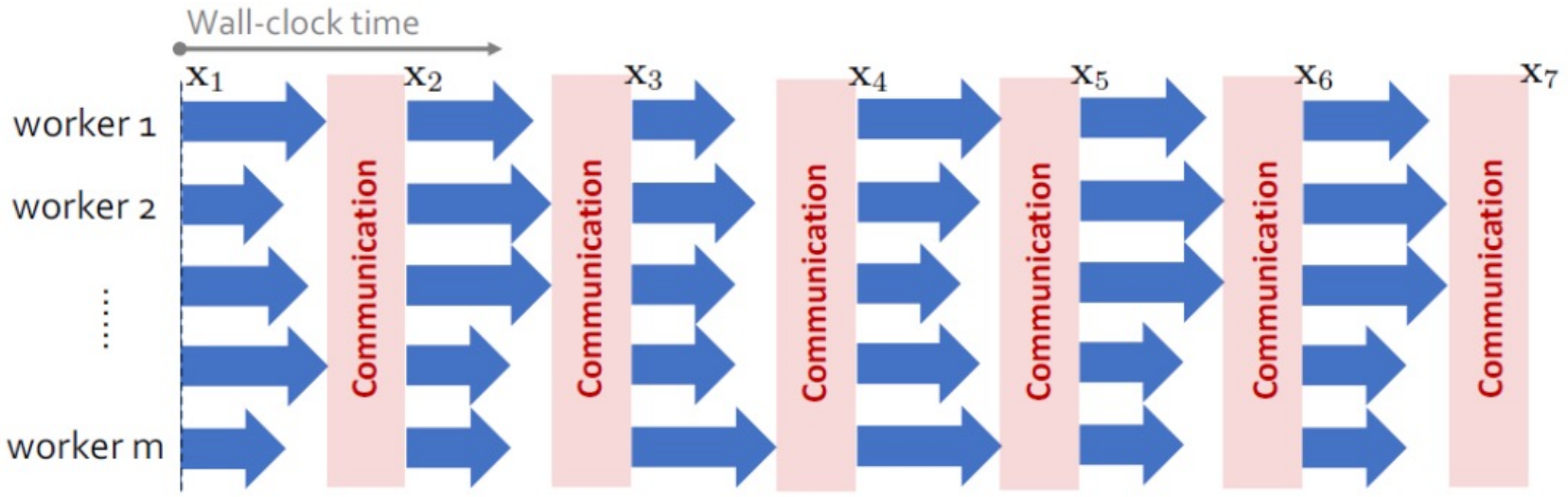
$$g(\mathbf{x}_k;\xi_k^{(i)}) = \frac{1}{|\xi_k^{(i)}|}\sum_{j\in\xi_k^{(i)}} \nabla f_j(\mathbf{x}_k)$$



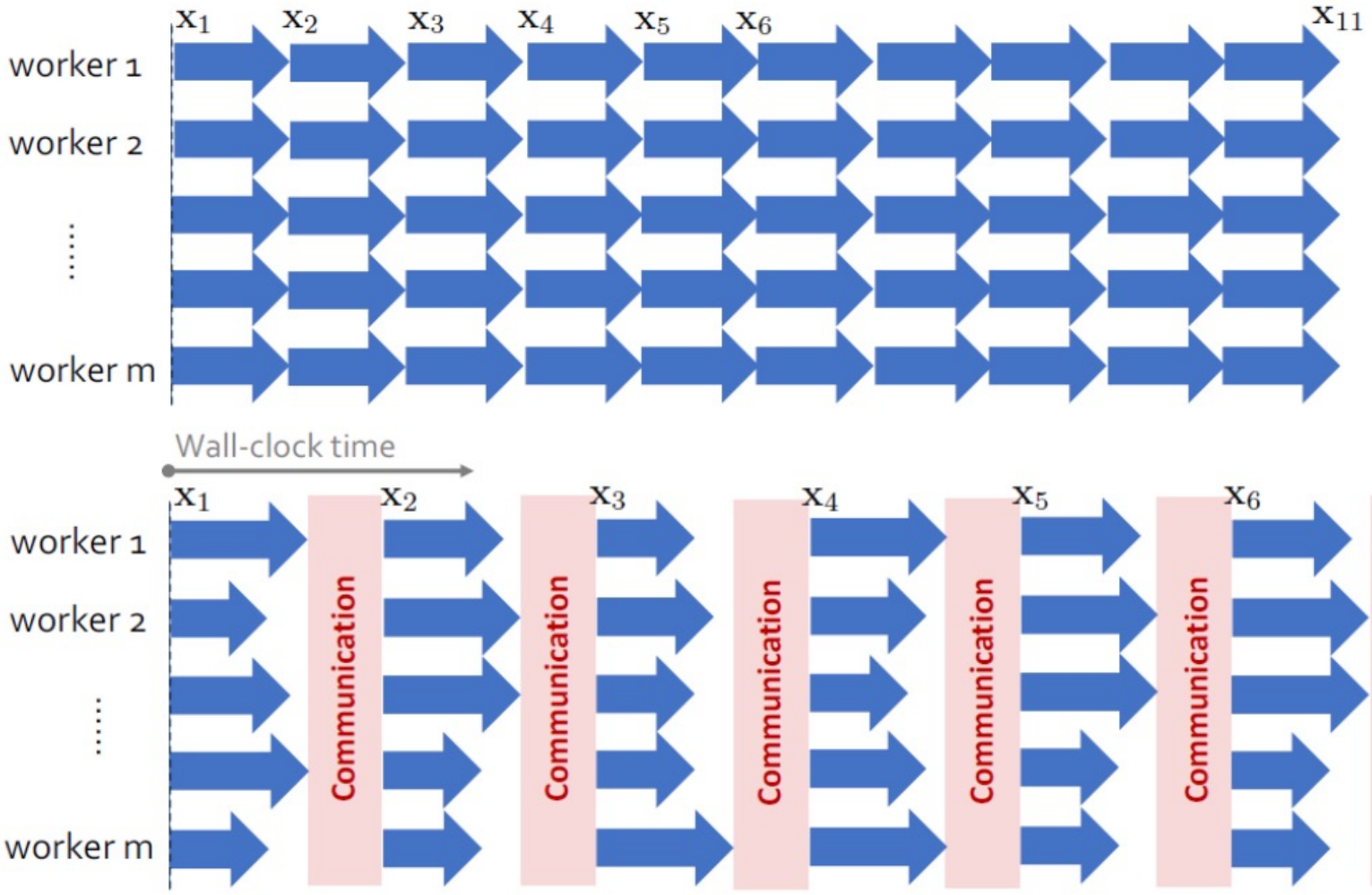
$$\frac{1}{m}\sum_{i=1}^{m}\mathbf{x}^{(i)}$$

Fully Synchronous SGD

1. Local stochastic gradients computation
2. Average local models across all nodes
3. Repeat the above steps until convergence



Fully Synchronous SGD



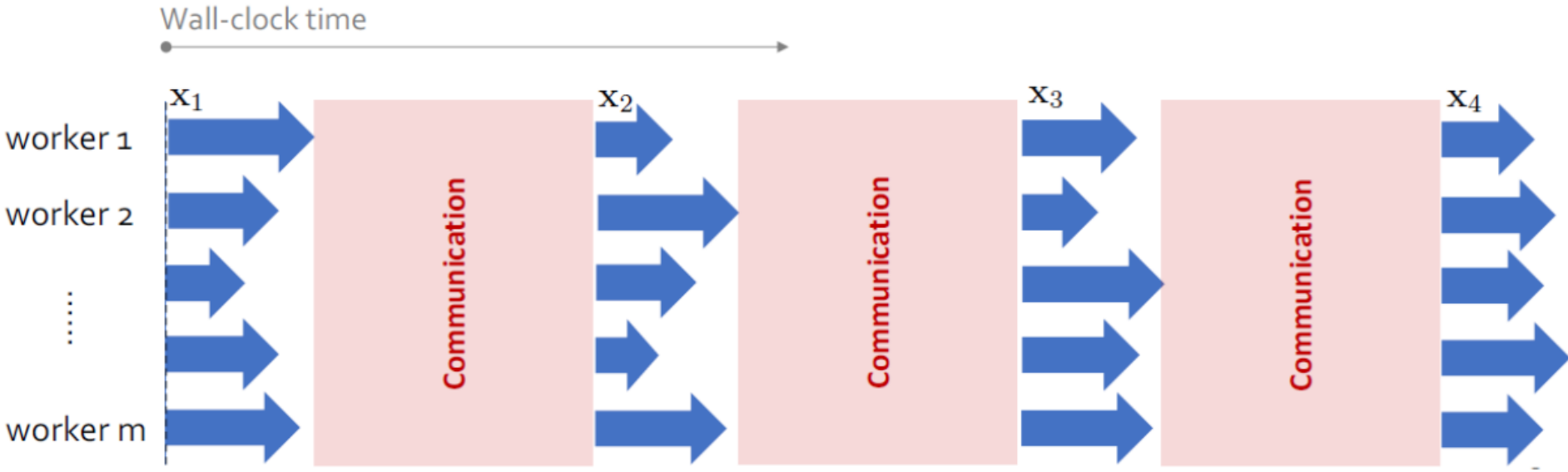
Ideal:
11 iterations

Practice:
7 iterations

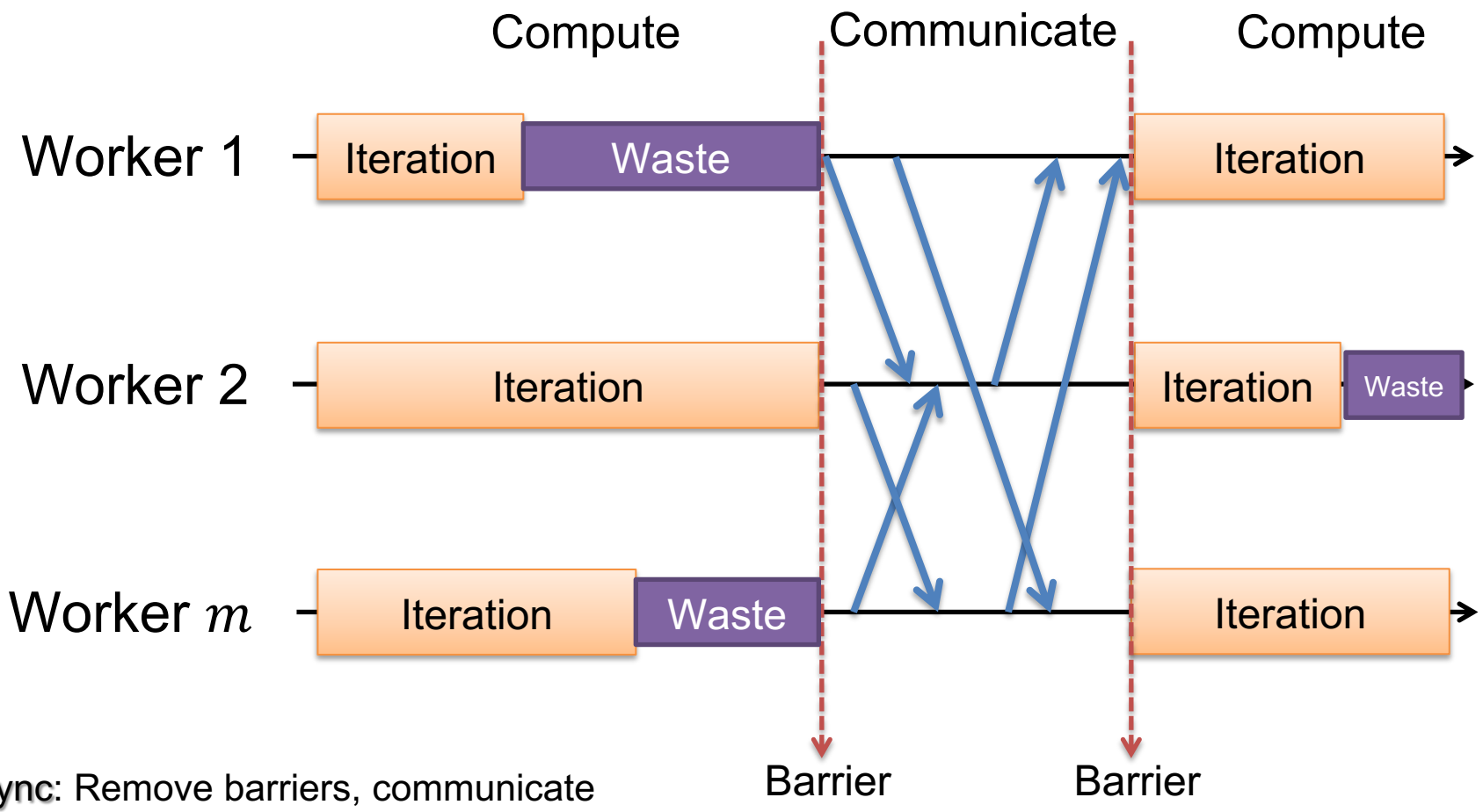
Fully Synchronous SGD in Deep Learning

In DNN training
communication time can be even larger than computation time

Need for communication-efficient distributed SGD



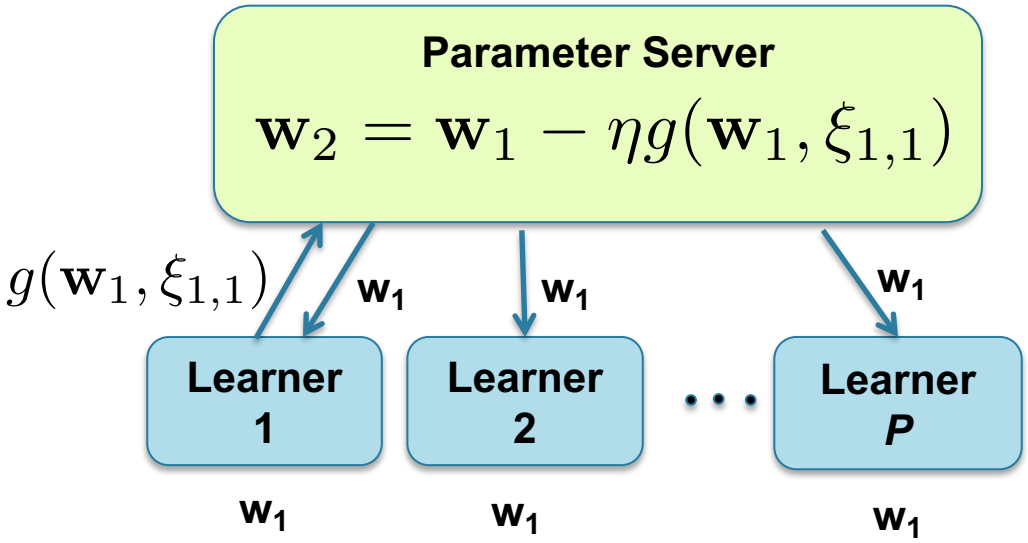
Asynchronous SGD



Async: Remove barriers, communicate and compute whenever each worker likes.

Asynchronous SGD

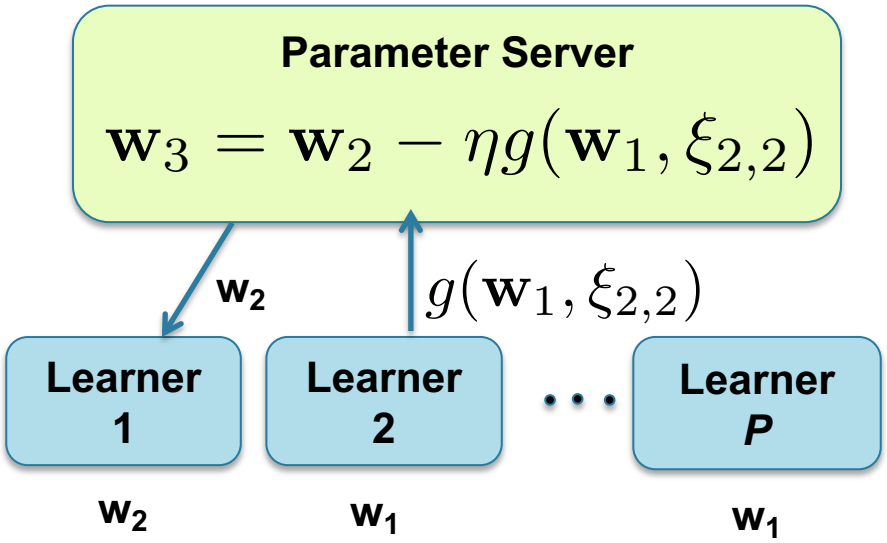
$$\mathbf{w}_{j+1} = \mathbf{w}_j - \eta g(\mathbf{w}_{\tau(j)}, \xi_j)$$



$\xi_{l,j}$ is the mini-batch of m samples for l-th learner

Asynchronous SGD

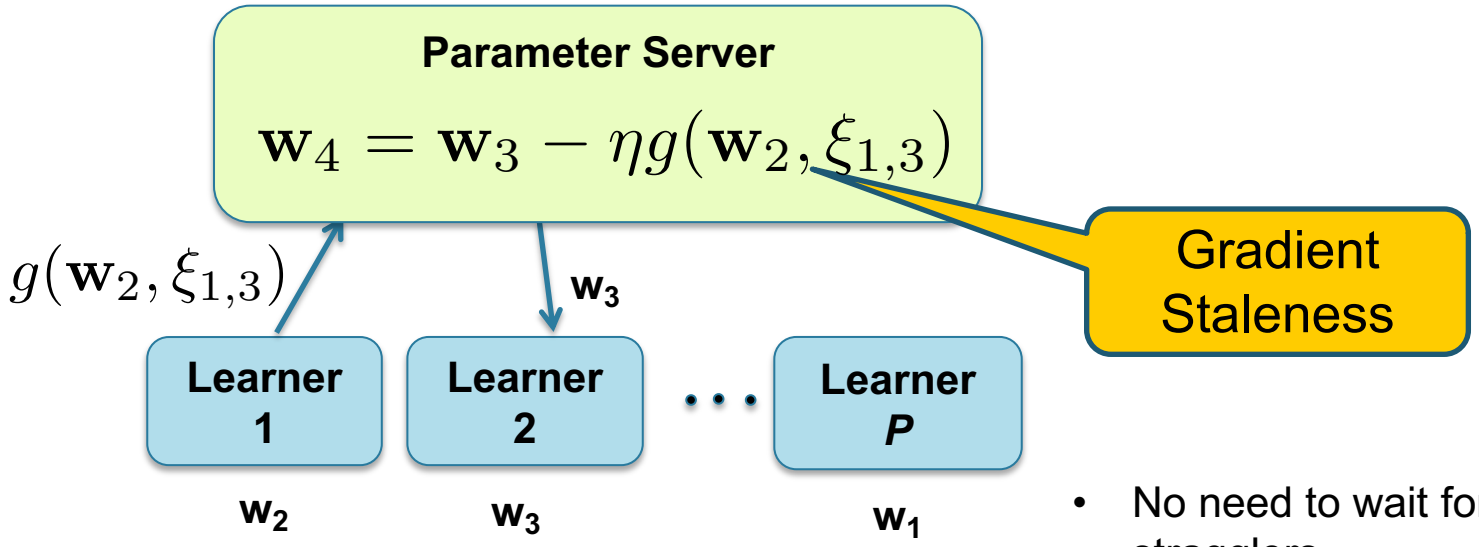
$$\mathbf{w}_{j+1} = \mathbf{w}_j - \eta g(\mathbf{w}_{\tau(j)}, \xi_j)$$



$\xi_{l,j}$ is the mini-batch of m samples for l-th learner

Asynchronous SGD

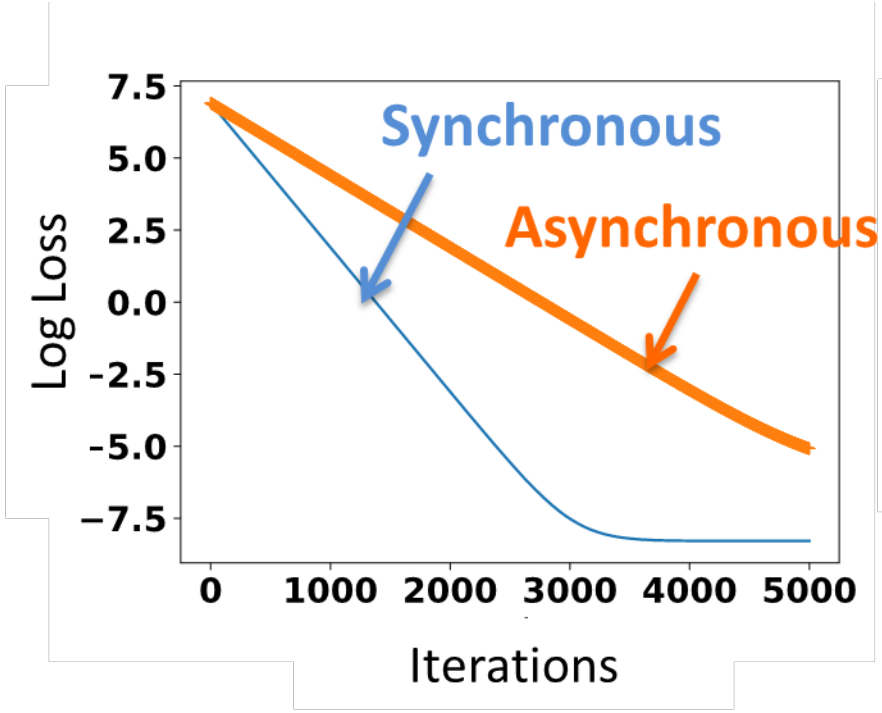
$$\mathbf{w}_{j+1} = \mathbf{w}_j - \eta g(\mathbf{w}_{\tau(j)}, \xi_j)$$

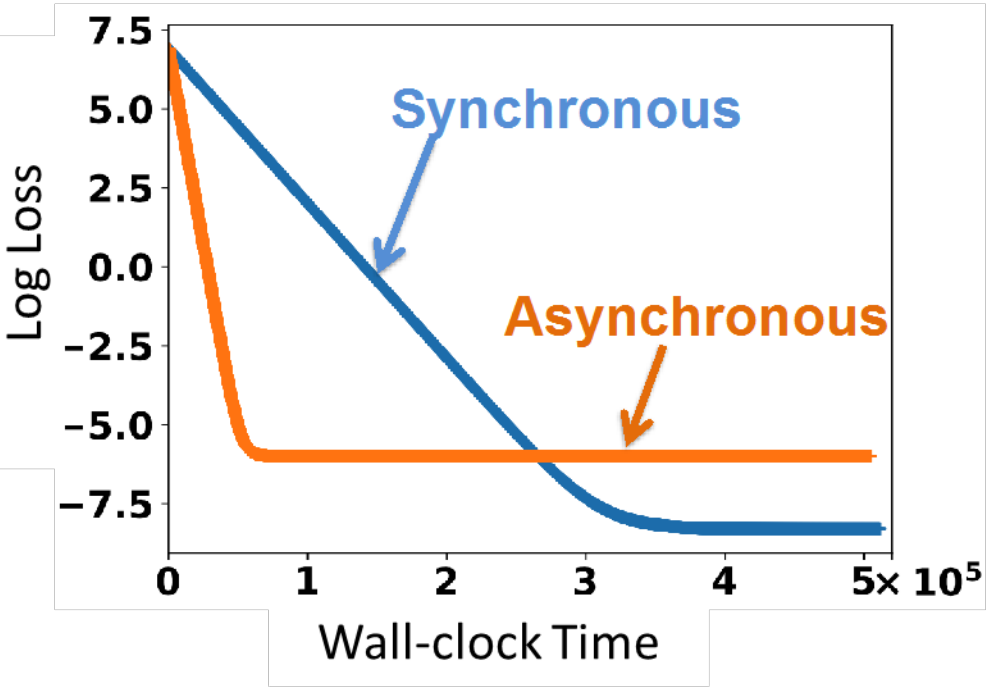


- No need to wait for stragglers
- Gradient Staleness can increase error

Synchronous vs. Asynchronous SGD

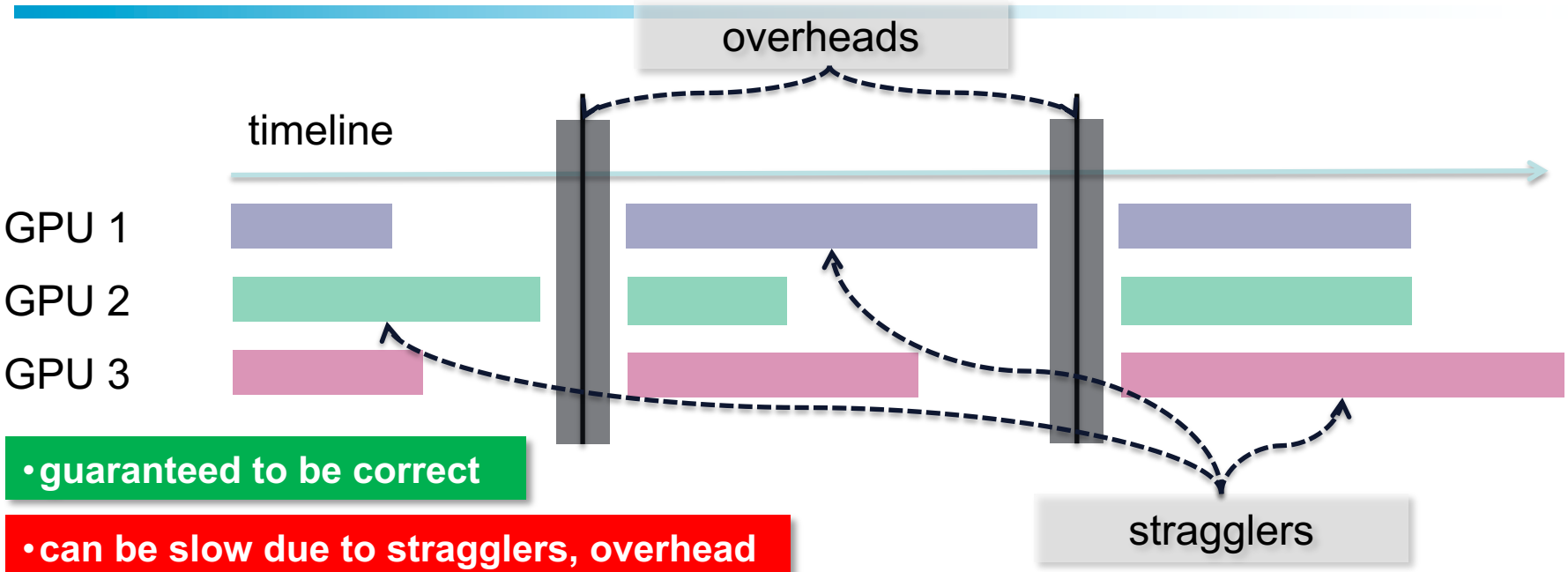
Performance Comparison





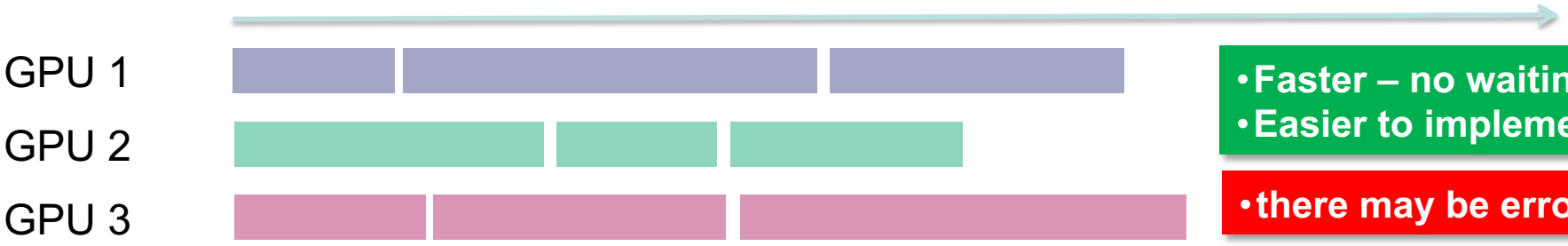
EURECOM
Sophia Antipolis

Synchronous vs. Asynchronous



- guaranteed to be correct
- can be slow due to stragglers, overhead

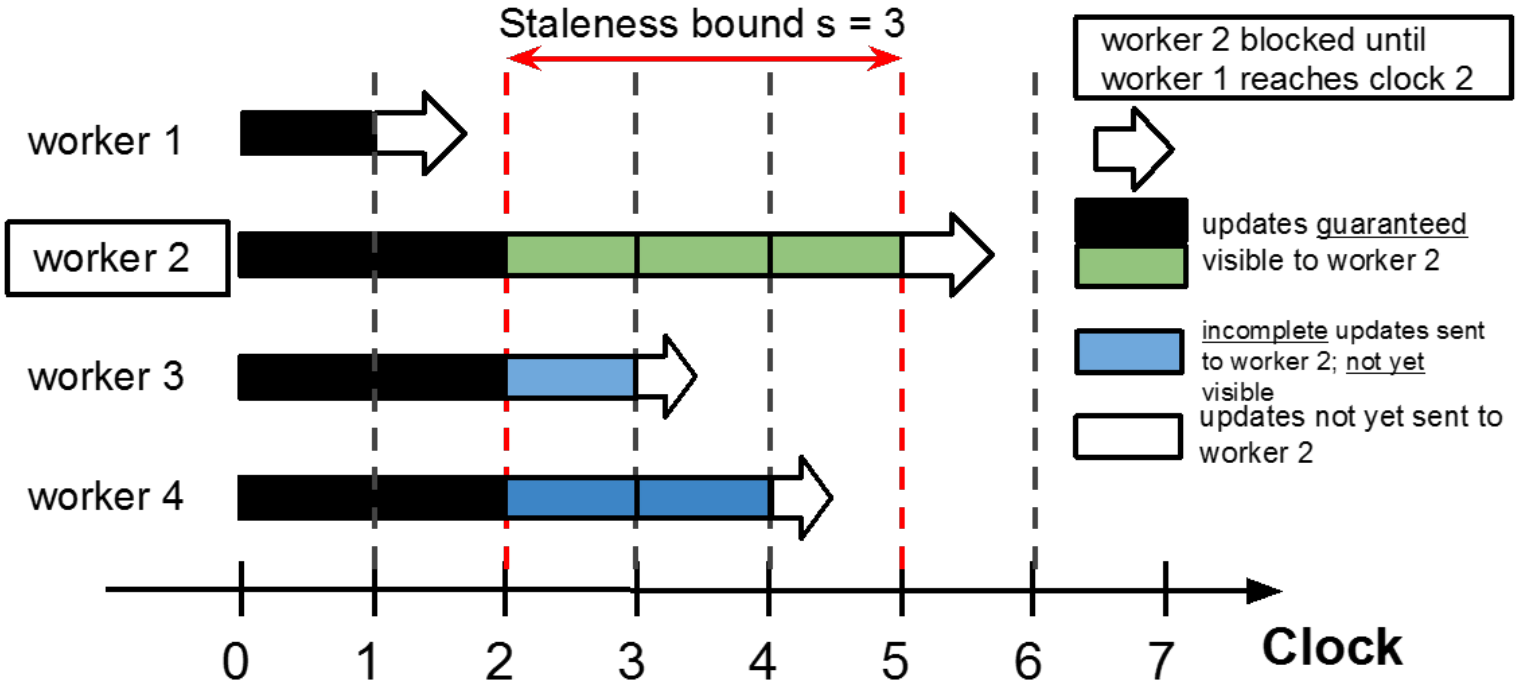
Asynchronous World



- Faster – no waiting
- Easier to implement
- there may be errors

Stale Synchronous Parallel (SSP)

- Between BSP and Async (subsumes both)



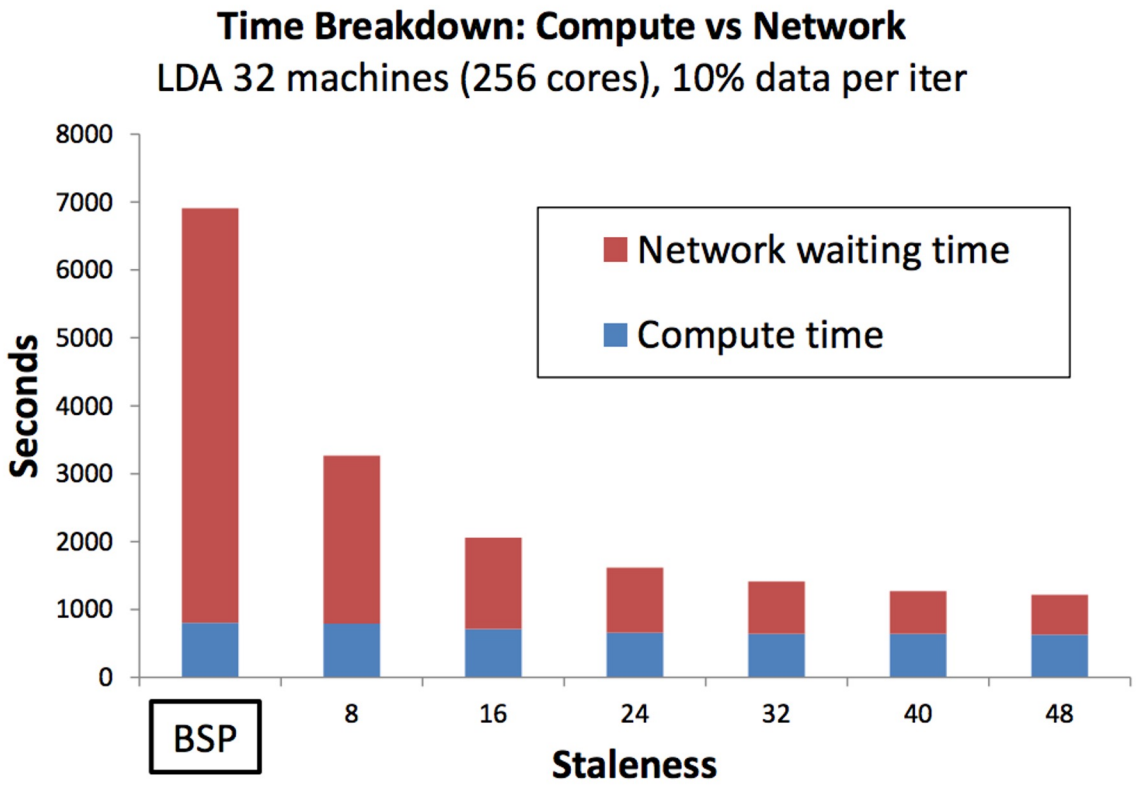
Black: all workers see these updates.

Blue: unspecified, workers may see it or not.

- Allow workers to run at their own pace as long as they are within an acceptable threshold of each other.
- Straggler-tolerant up to a certain threshold (staleness bound).
- Fastest/slowest threads not allowed to drift $> s$ clocks apart
- Efficiently implementable
- Allows communication and computation to happen concurrently.

Consistency

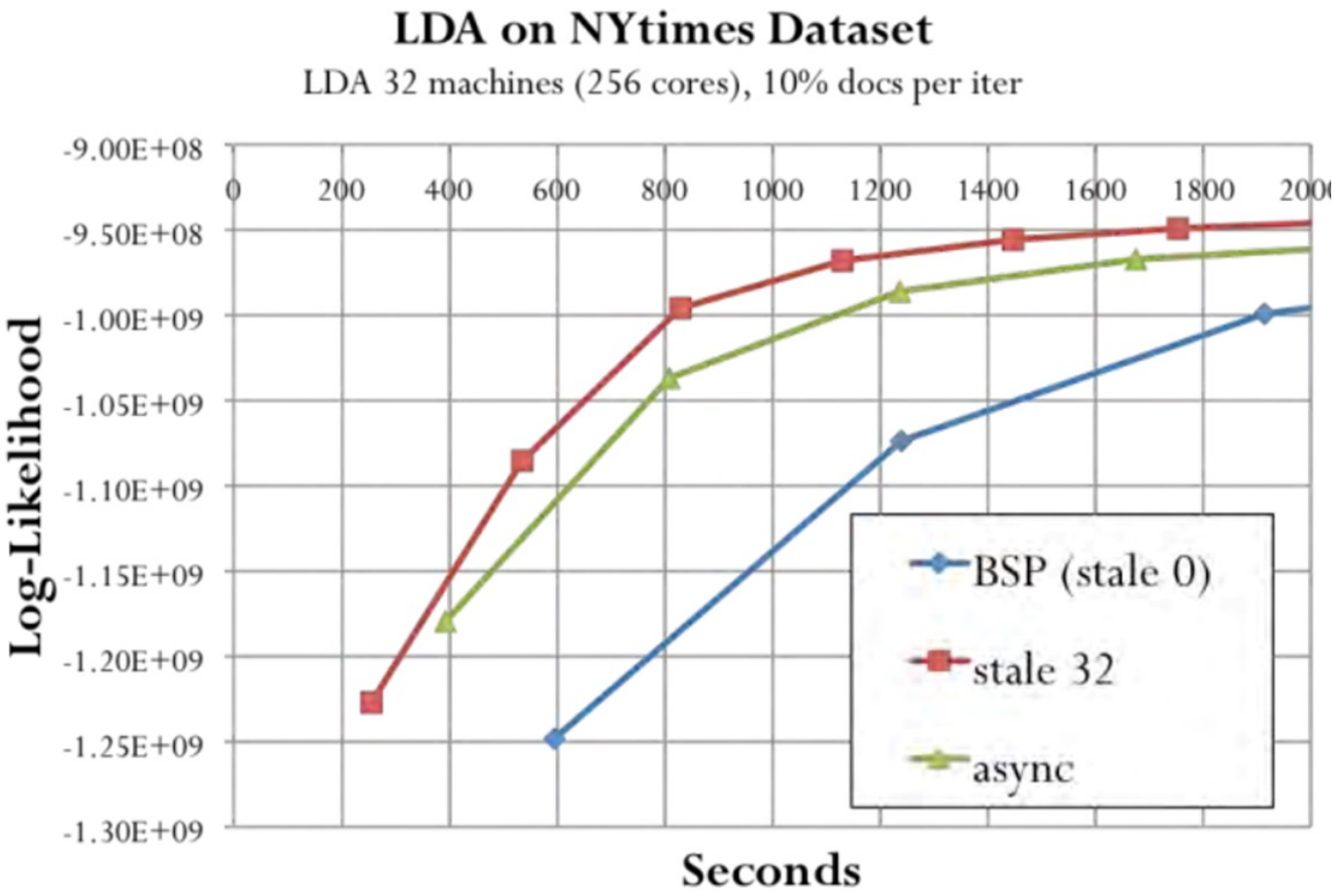
- Consistency: *how recent workers views of shared variables are.*



Strong consistency ⟶ Relaxed consistency

- BSP: consistent - Async: not consistent.
- Reduced network time with relaxed consistency.
- Suitable delay/staleness (SSP) gives big speed-up

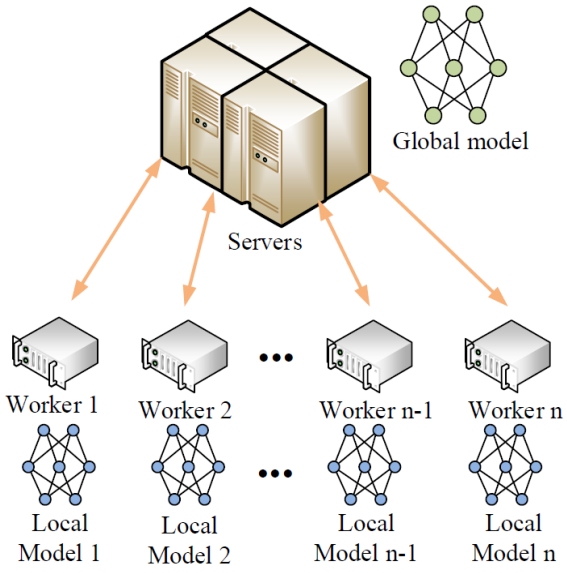
SSP Performance



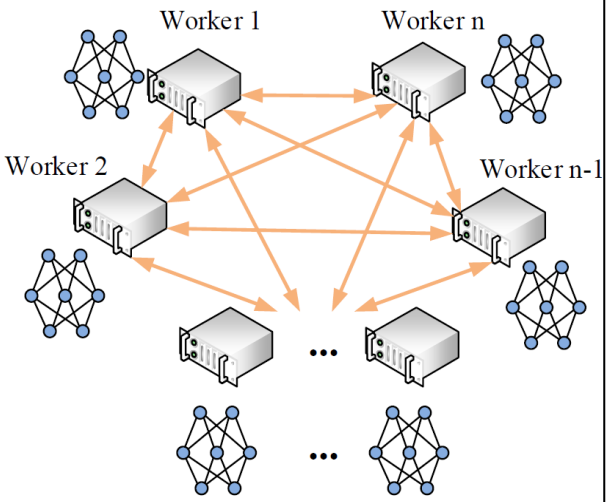
- SSP with correct staleness value is better than Async and BSP.
- SSP retains theoretical convergence guarantees.

System Architectures

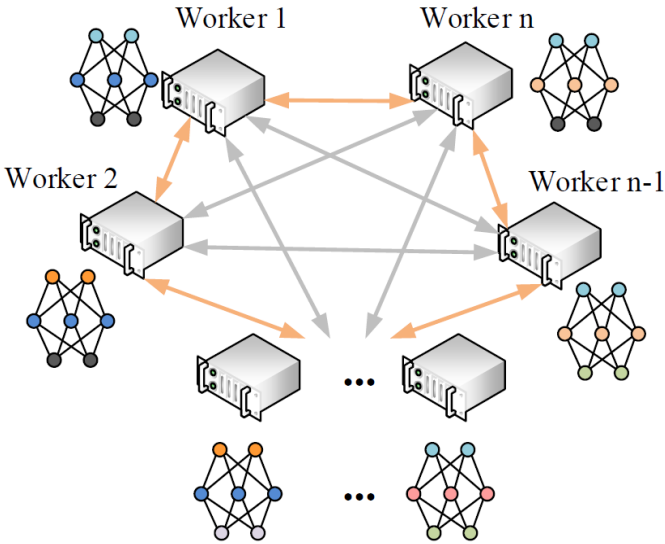
Model/Gradient aggregation



Parameter Server



All-to-All
(All Reduce)



Decentralized / Gossip

System Architectures & Synchronization

Architecture	Synchronization	Model consistency	Communication Frequency	Communication Congestion	Convergence
PS	BSP	high	high	high	easy
	SSP	normal	high	normal	normal
	ASP	low	high	low	difficult
	Local SGD	normal	low	high	difficult
All-Reduce	BSP	high	high	low	easy
	SSP	-	-	-	-
	ASP	-	-	-	-
	Local SGD	normal	low	low	difficult
Gossip	BSP	low	high	low	normal
	SSP	-	-	-	-
	ASP	low	high	low	difficult
	Local SGD	low	low	low	difficult

- Model consistency measures how local models are different from others
- Communication frequency measures how frequently workers communicate with others
- Communication congestion measures how heavy the traffic of the central node is
- BSP: synchronous / SSP: stale synchronous / ASP: asynchronous

Communication-Efficient First-order Methods

Communication Round Minimization

- Accelerating mini-batch SGD:
  - use the largest possible mini-batch that does not hurt the sample complexity
- Variance Reduction techniques:
  - each device collects spatially distributed data and devices are allowed to communicate only with direct neighbors.
- How to tackle statistical heterogeneity of data? (hinders the fast convergence)

Communication Bandwidth Minimization

- Target: reduce the size of local updates from each device, thereby reducing the overall communication cost.
- 3 representative techniques: gradient reuse, gradient quantization, and gradient sparsification

Gradient Reuse:

- Key observation: gradients of some devices vary slowly between two consecutive communication rounds
- Lazily Aggregated Gradient (LAG):
  - use outdated gradients of these devices at the fusion center
  - some devices upload nothing during a communication round $\Rightarrow$ communication overhead per round significantly reduced

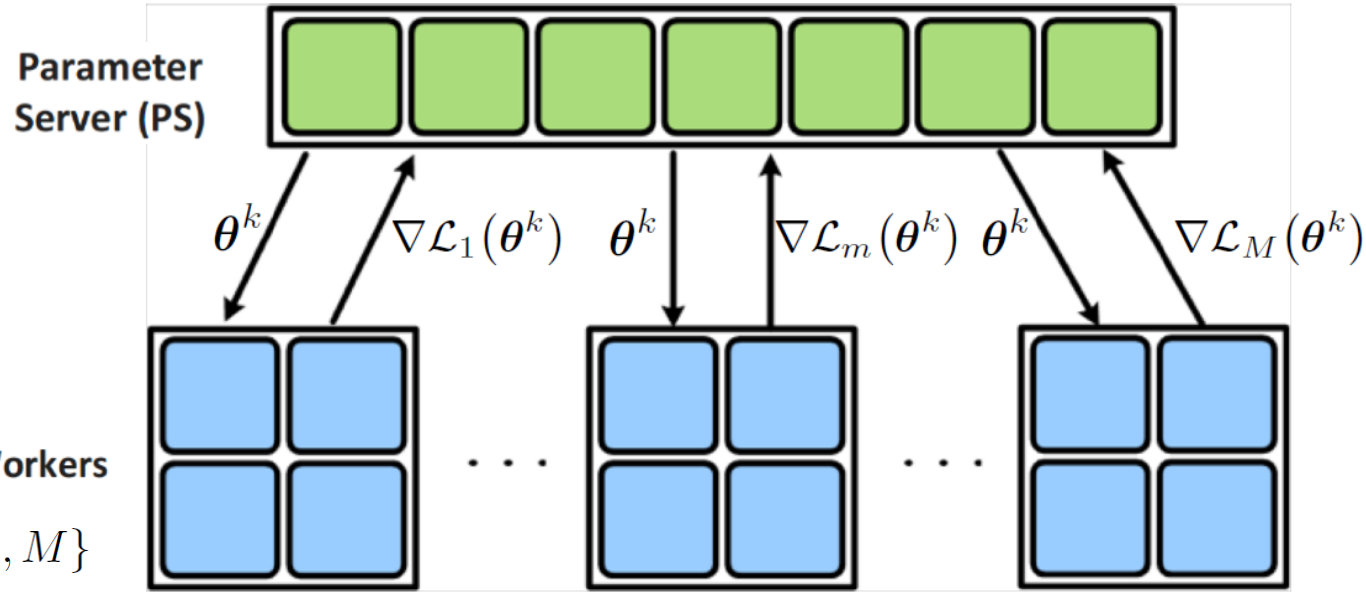
Lazily Aggregated Gradients

Gradient Reuse

$$\underset{\boldsymbol{\theta}\in\mathbb{R}^d}{\text{minimize}}\ \mathcal{L}(\boldsymbol{\theta}) \quad \text{with} \quad \mathcal{L}(\boldsymbol{\theta}) := \sum_{m\in\mathcal{M}} \mathcal{L}_m(\boldsymbol{\theta})$$

Minimization of a sum of smooth loss functions distributed among multiple devices

$$\boldsymbol{\theta}^{k+1} = \boldsymbol{\theta}^k - \alpha \sum_{m\in\mathcal{M}} \nabla\mathcal{L}_m(\boldsymbol{\theta}^k)$$



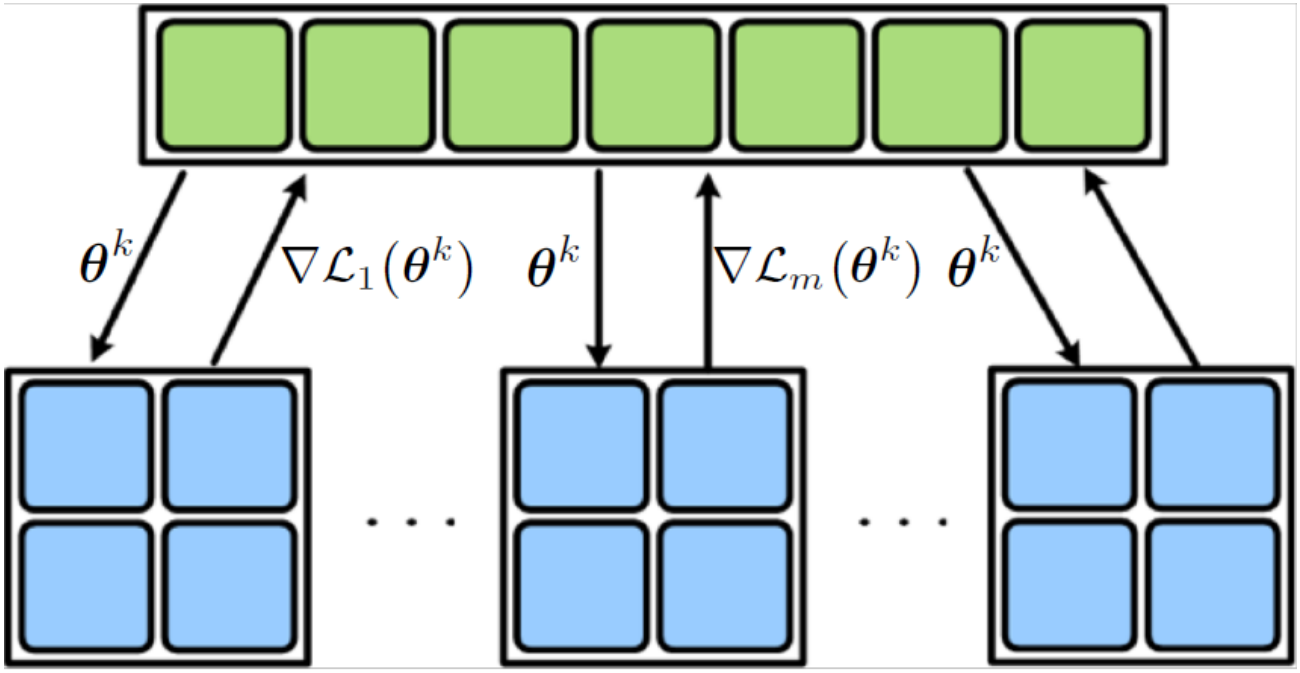
$\mathcal{M} := \{1, \ldots, M\}$

- Per iteration communication overhead for *M* uploads (one per worker)

Lazily Aggregated Gradients

Fresh gradient Old gradient

$$\boldsymbol{\theta}^{k+1} = \boldsymbol{\theta}^k - \alpha \sum_{m \in \mathcal{M}^k} \nabla \mathcal{L}_m(\boldsymbol{\theta}^k) - \alpha \sum_{m \in \mathcal{M}/\mathcal{M}^k} \nabla \mathcal{L}_m(\hat{\boldsymbol{\theta}}_m^{k-1})$$

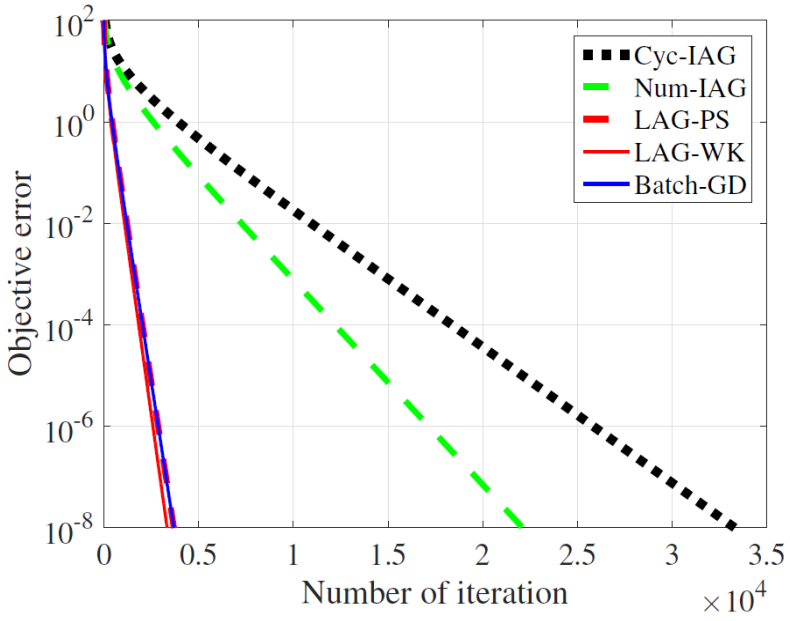


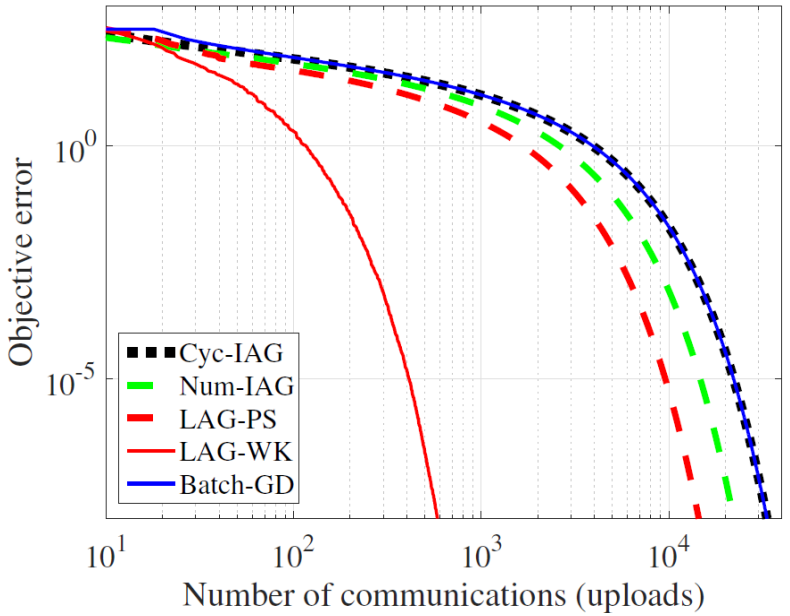
- Select a subset of workers $\mathcal{M}^k \subseteq \mathcal{M}$ to upload
- Remaining workers in $\mathcal{M}/\mathcal{M}^k$ do not upload

Lazily Aggregated Gradient: Performance

- LAG enjoys the same convergence rate as GD for loss $\mathcal{L}(\boldsymbol{\theta})$ convex, strongly convex and for local loss $\mathcal{L}_m(\boldsymbol{\theta})$ nonconvex smooth.
- Heterogeneous distributed datasets (local objectives): LAG requires smaller # uploads (lower communication cost, e.g. 1/*M*) for achieving a given accuracy than GD, e.g. 1/*M*.

Performance – Logistic Regression (real datasets, M=9 workers)





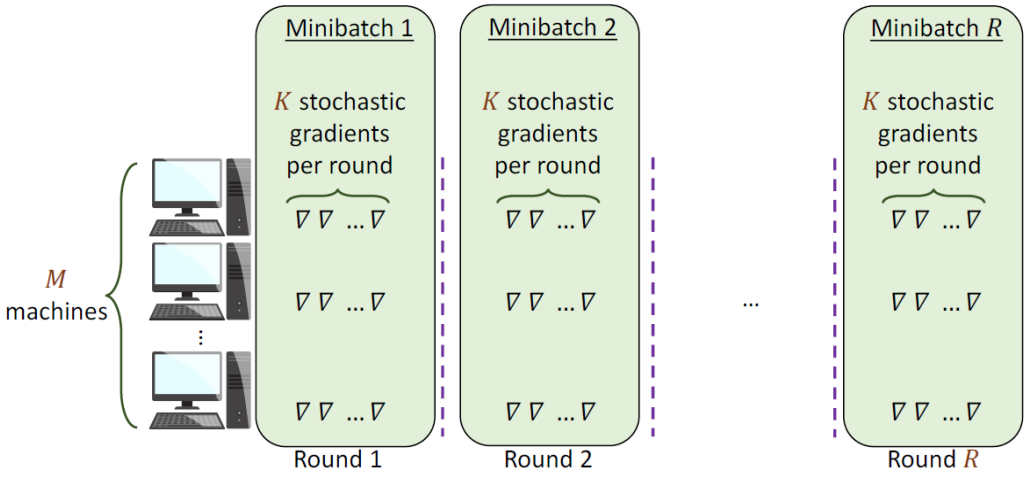
Cyc-/Num-IAG: cyclic/non-uniform update of incremental aggregated gradient

- LAG needs same number of iterations but fewer uploads

Periodic Averaging SGD (PASGD)

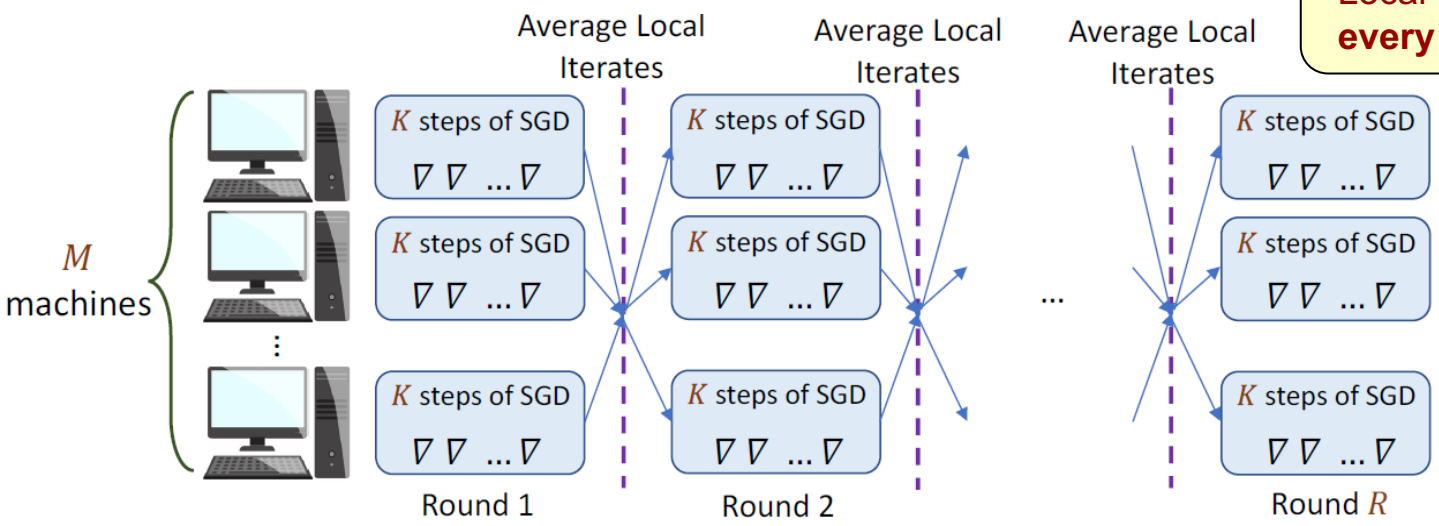
Mini-batch SGD

$R = T/K$ steps of SGD with mini-batch size KM



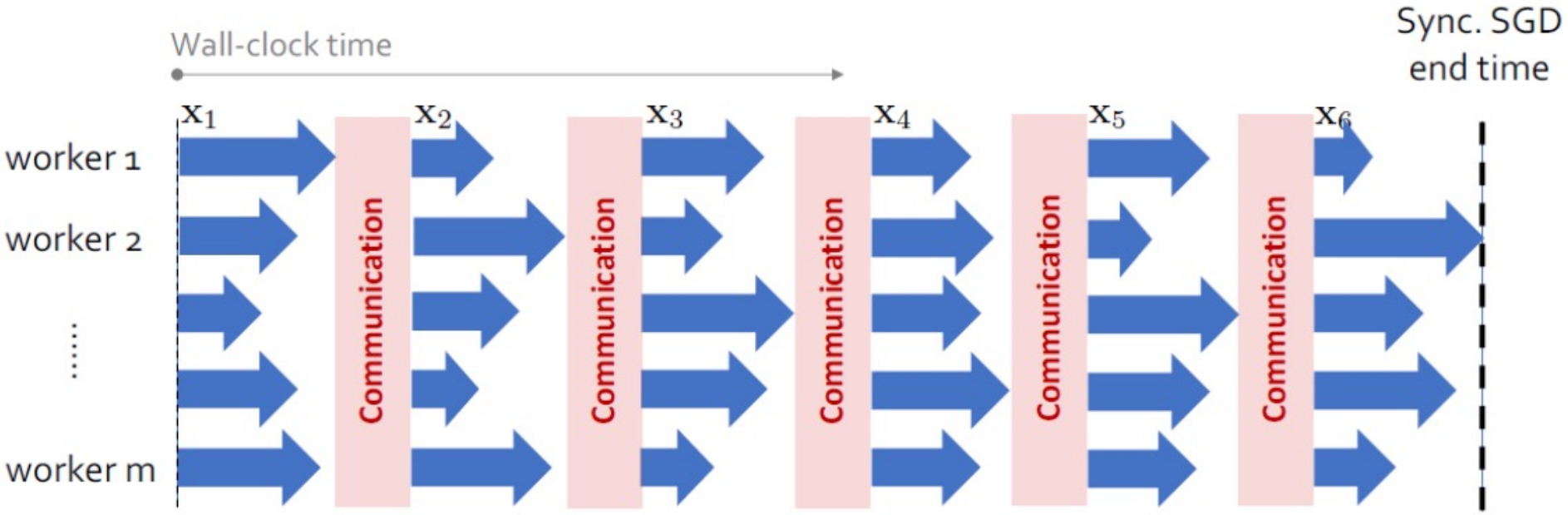
PASGD / Local SGD

- Workers perform ***K*** local updates
- Local models are averaged **after every *K* local steps**



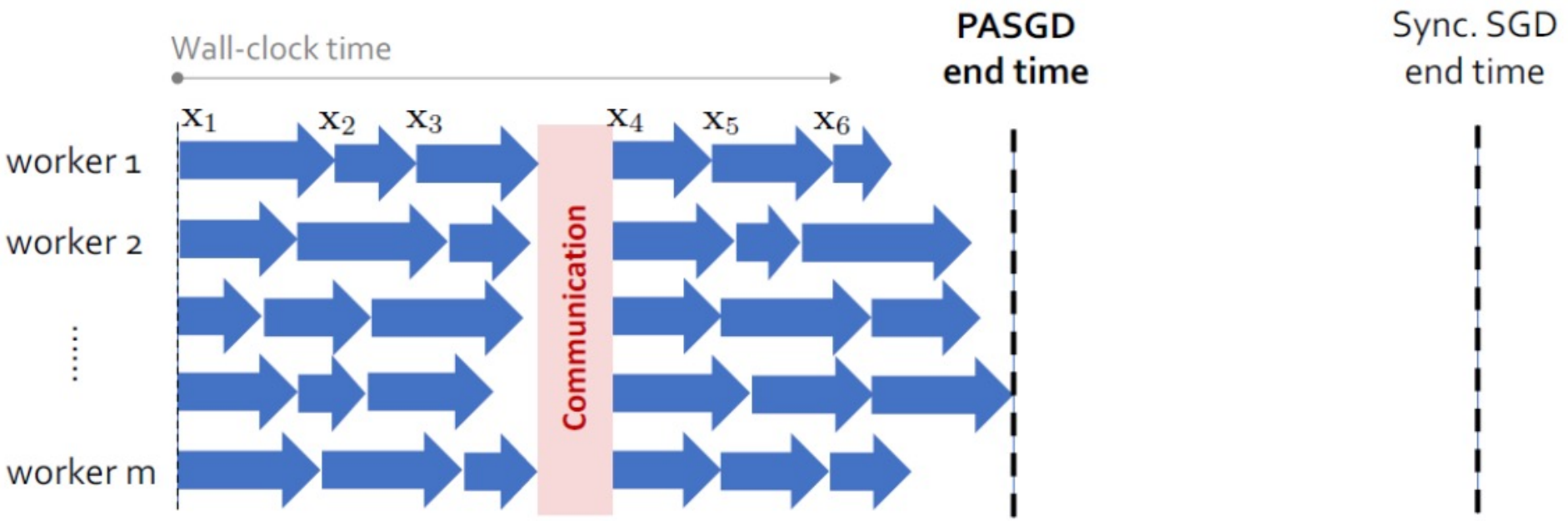
- Performs very well empirically
- Best of both worlds?
- Makes KR updates (vs. only R for mini-batch SGD)
- Leverages parallelism (unlike Single-Machine SGD)

Periodic Averaging / Local SGD



Periodic Averaging / Local SGD

- Communication delay is reduced by *K* times

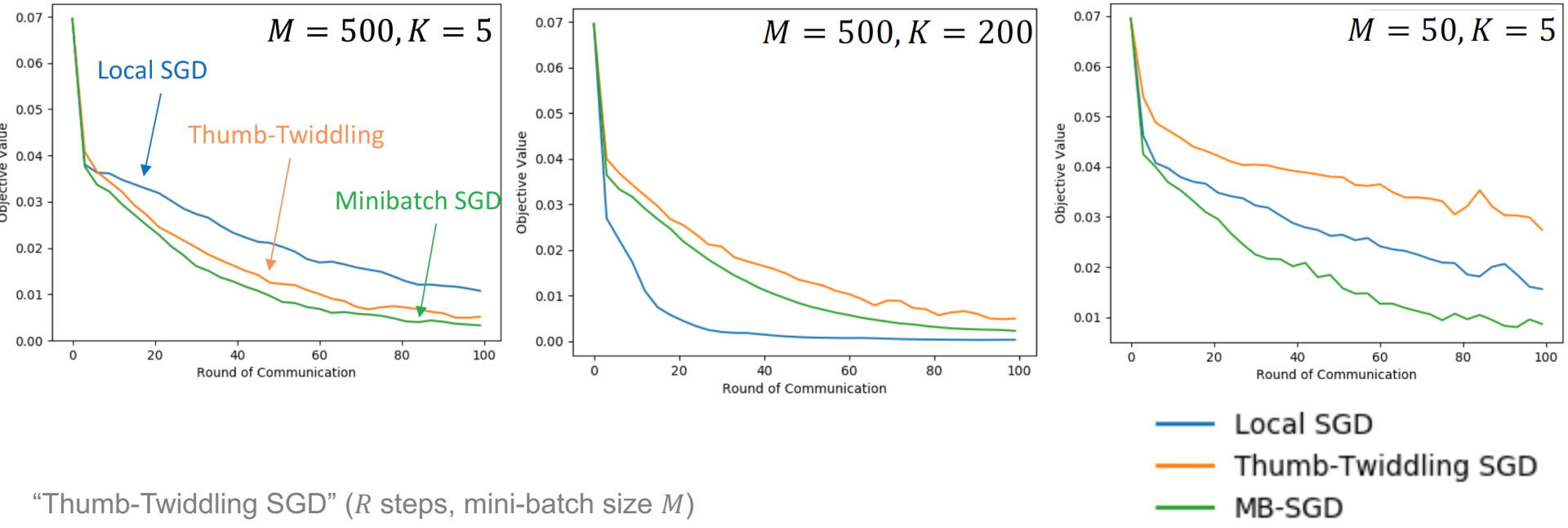


Local SGD vs. Mini-batch SGD

- Local SGD is neither strictly better nor strictly worse than mini-batch SGD!!
- Better than mini-batch SGD if $K = \Omega(R)$
- Worse than mini-batch SGD if $K = O(\sqrt{R})$

Performance Comparison

Logistic regression experiment

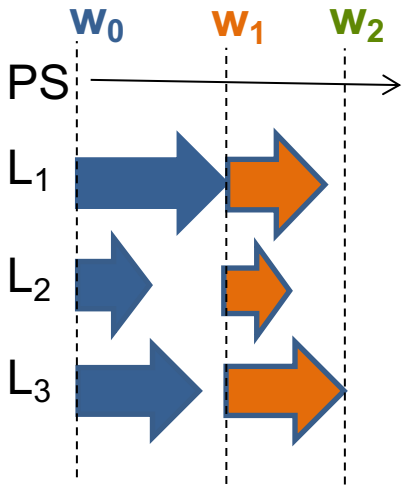


"Thumb-Twiddling SGD" (R steps, mini-batch size M)

Expected Runtime of SGD Variants

Sync variants

Fully Sync-SGD



K-sync SGD

K-batch-sync SGD

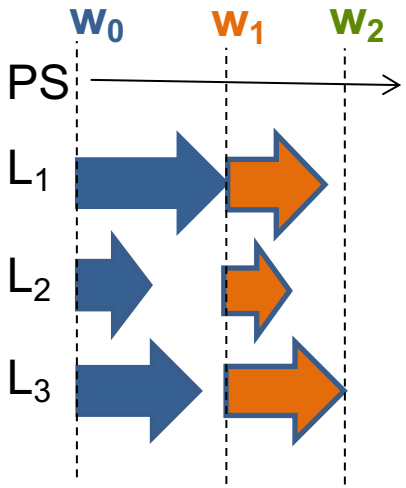
- Fully Sync-SGD: we wait for all learners to complete computation.
- Once one step is complete, model weights are then combined and distributed to all learners, and the next batch of training begins.
- Inefficient: learners are idle for most of the time

https://arxiv.org/abs/1803.01113

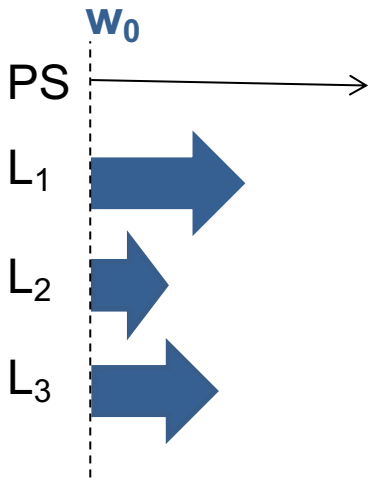
Expected Runtime of SGD Variants

Sync variants

Fully Sync-SGD



K-sync SGD



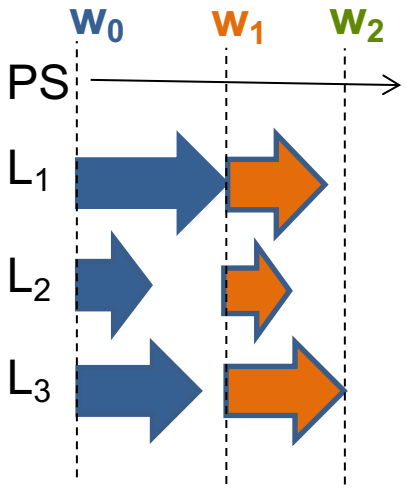
K-batch-sync SGD

- K Sync-SGD: we can wait for “K” of them.

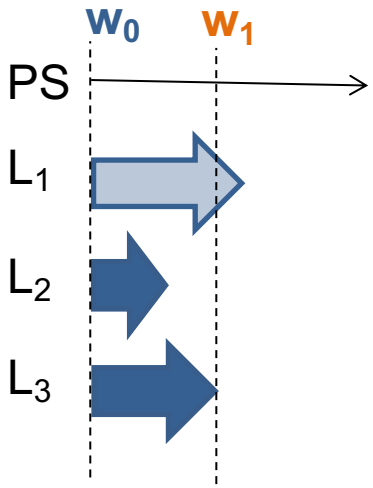
Expected Runtime of SGD Variants

Sync variants

Fully Sync-SGD



K-sync SGD



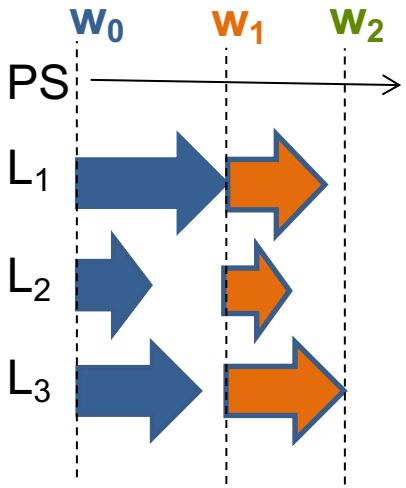
K-batch-sync SGD

- K Sync-SGD: we can wait for “K” of them.

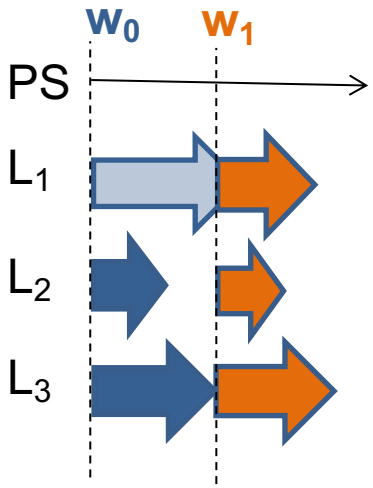
Expected Runtime of SGD Variants

Sync variants

Fully Sync-SGD



K-sync SGD



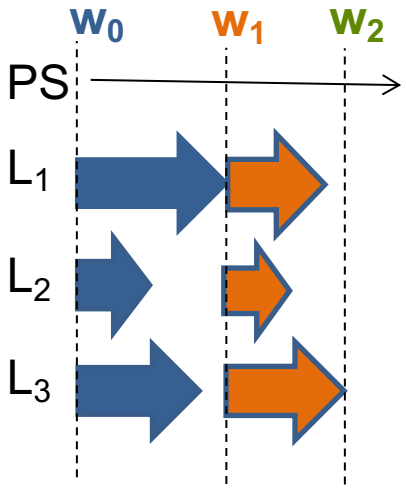
K-batch-sync SGD

- K Sync-SGD: we can wait for “K” of them.

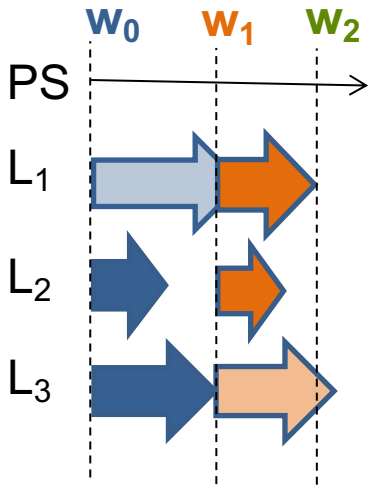
Expected Runtime of SGD Variants

Sync variants

Fully Sync-SGD



K-sync SGD

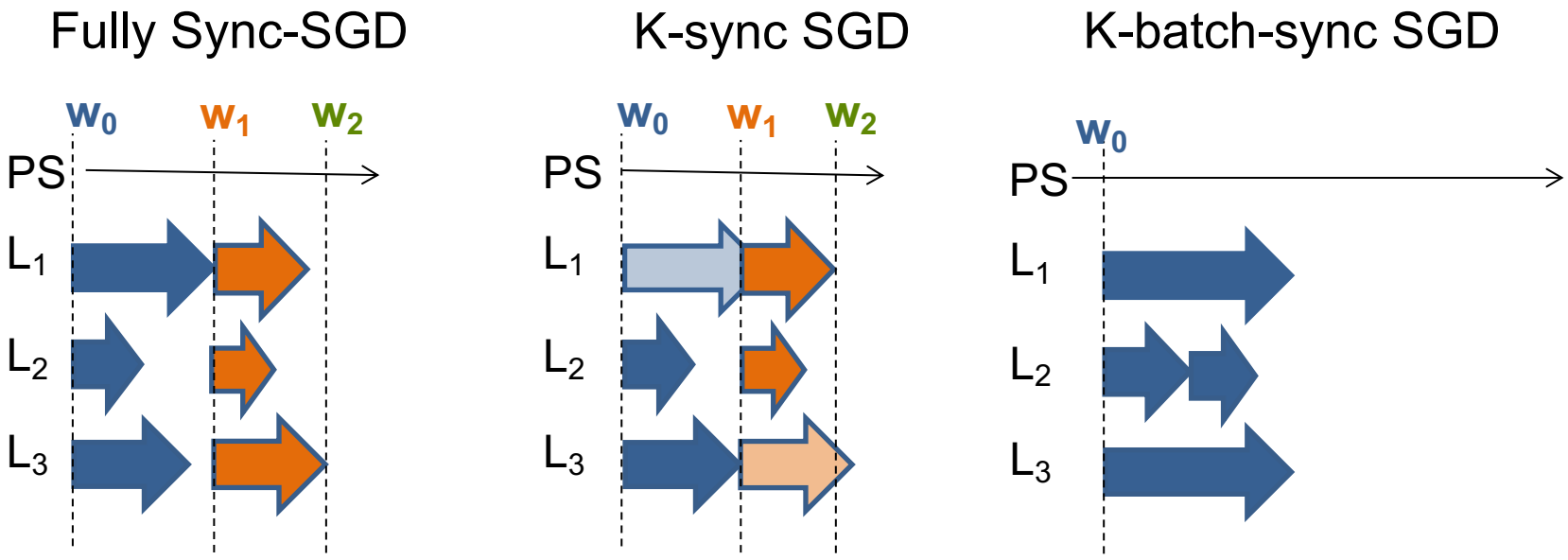


K-batch-sync SGD

- K Sync-SGD: we can wait for “K” of them.
- Some learners still have to wait until K of them have completed execution, and we are throwing away the work we did at N-K learner

Expected Runtime of SGD Variants

Sync variants

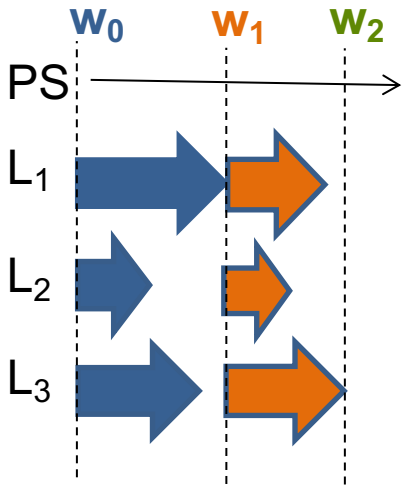


- K-batch-sync-SGD: update at the end of K batches of data, instead of K workers.

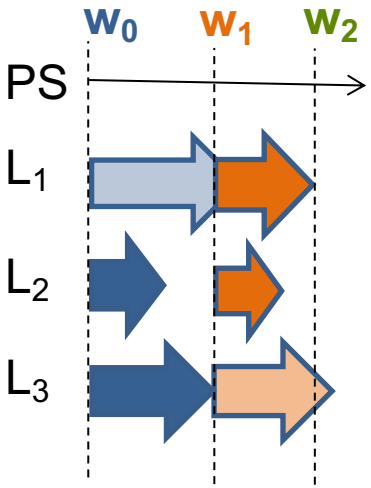
Expected Runtime of SGD Variants

Sync variants

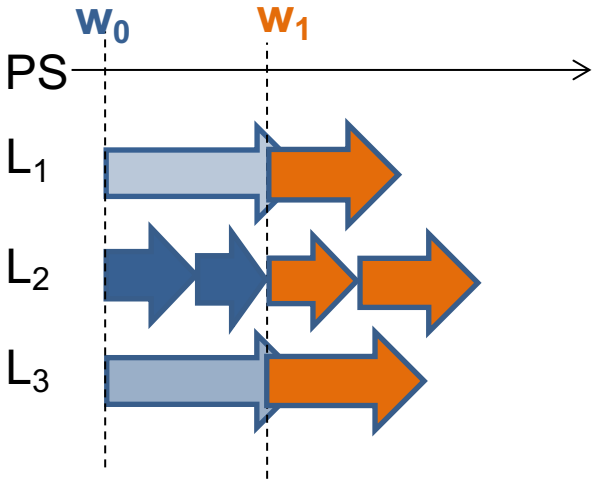
Fully Sync-SGD



K-sync SGD



K-batch-sync SGD

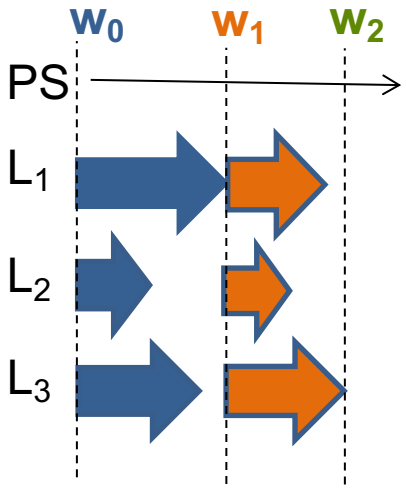


- K-batch-sync-SGD: update at the end of K batches of data, instead of K workers.

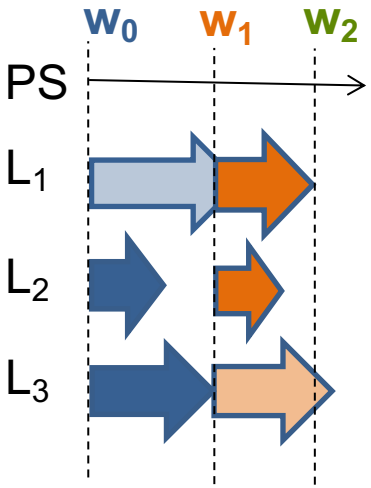
Expected Runtime of SGD Variants

Sync variants

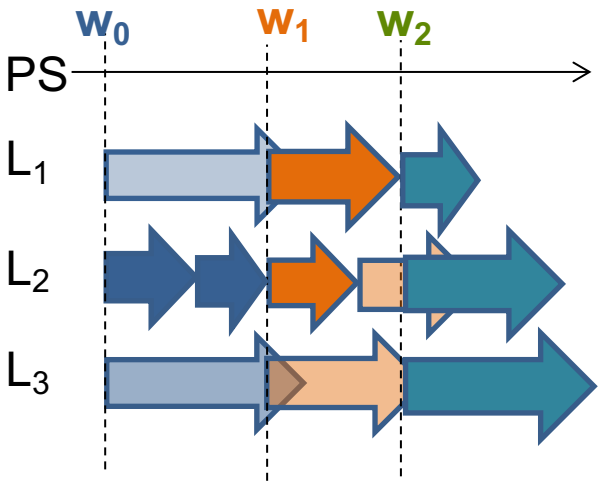
Fully Sync-SGD



K-sync SGD

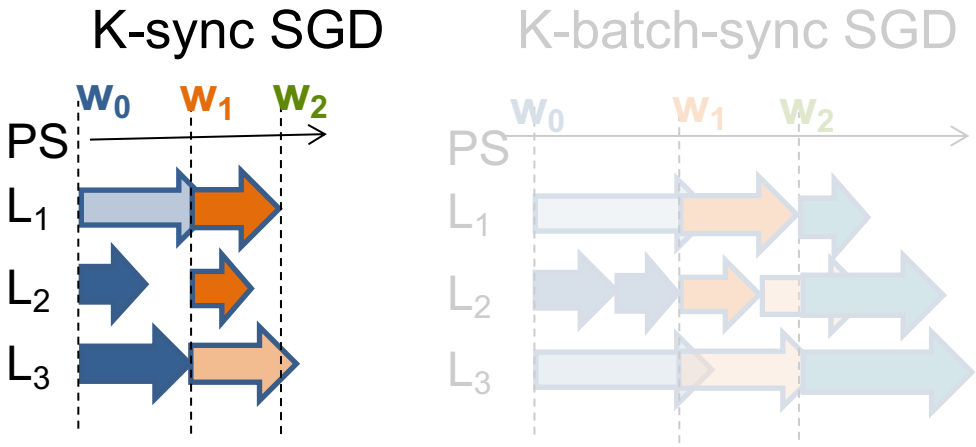


K-batch-sync SGD



$$\boldsymbol{w}_{j+1} = \boldsymbol{w}_j - \frac{\eta}{K} \sum_{l=1}^{K} g(\boldsymbol{\xi}_{l,j}, \boldsymbol{w}_j)$$

Expected Runtime of SGD Variants



K-sync SGD:

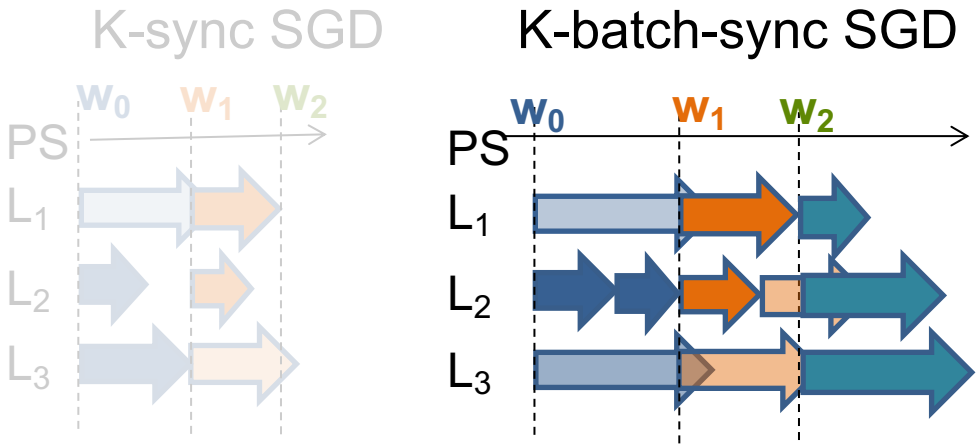
Expected Runtime per iteration: $\mathbb{E}[T] = \mathbb{E}[X_{K:P}]$

where $X_{K:P}$ is the K-th statistic of $X_1, X_2, \ldots X_P$.

Special Case: Exponential $X_l \sim \exp(\mu)$

$$\mathbb{E}[T] = \left(\frac{\log \frac{P}{P-K}}{\mu}\right)$$

Expected Runtime of SGD Variants



K-batch-sync SGD:

Expected Runtime per iteration: In general not tractable

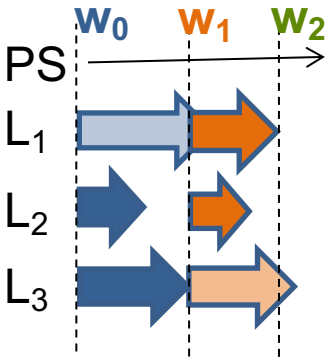
Special Case: Exponential $X_l \sim \exp(\mu)$

$$\mathbb{E}[T] = \left(\frac{K}{P\mu}\right)$$

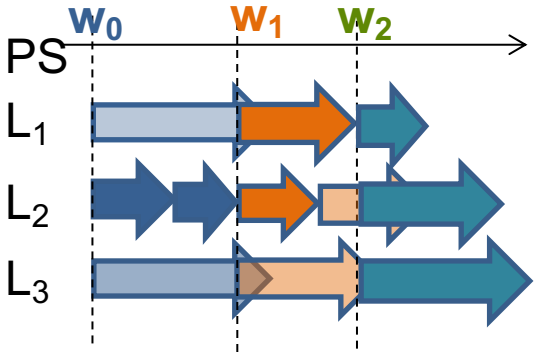
Key idea: Every gradient push – minimum of *P* exponentials

Expected Runtime of SGD Variants

K-sync SGD



K-batch-sync SGD



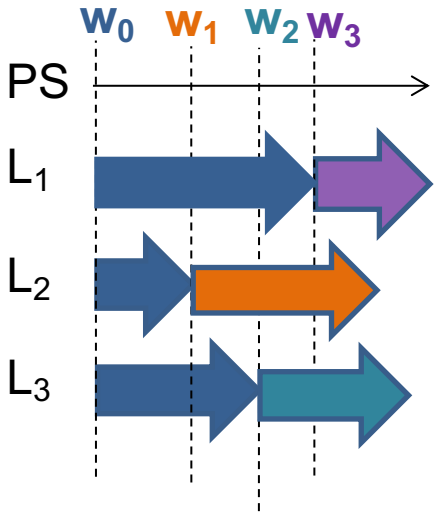
$$\mathbb{E}[T] = \mathbb{E}[X_{K:P}] \approx \frac{1}{\mu} \log \frac{P}{P-K}$$

$$\mathbb{E}[T] = \frac{K}{\mu P}$$

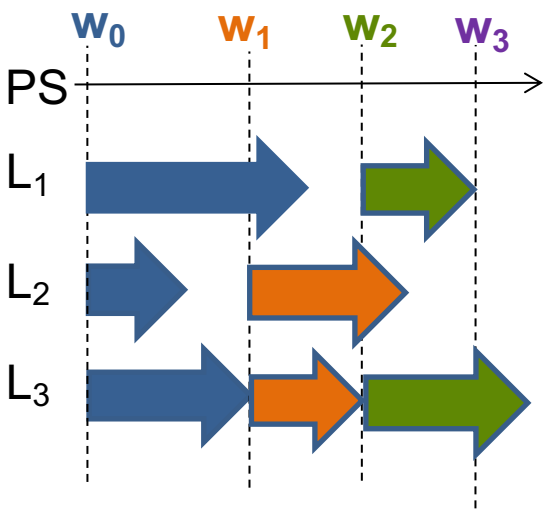
Expected Runtime of SGD Variants

Async variants

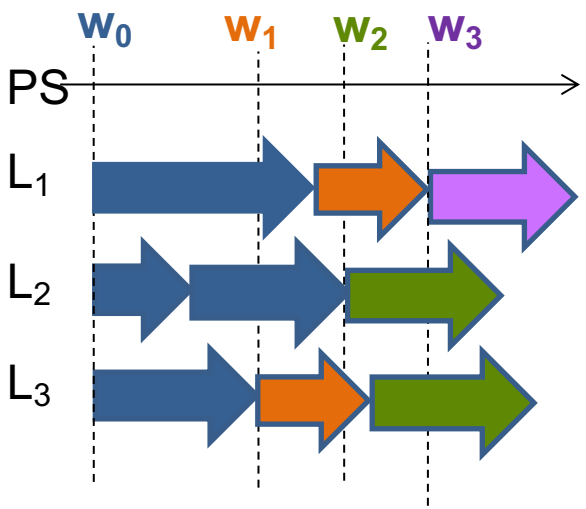
Async SGD



K-Async SGD



K-Batch Async SGD

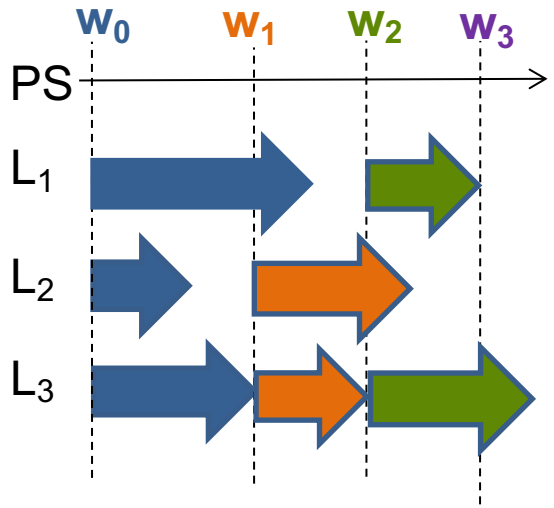


$$\boldsymbol{w}_{j+1} = \boldsymbol{w}_j - \frac{\eta}{K}\sum_{l=1}^{K} g\left(\boldsymbol{\xi}_{l,j}, \boldsymbol{w}_{\tau(l,j)}\right)$$

Expected Runtime of SGD Variants

Async variants

K-Async SGD

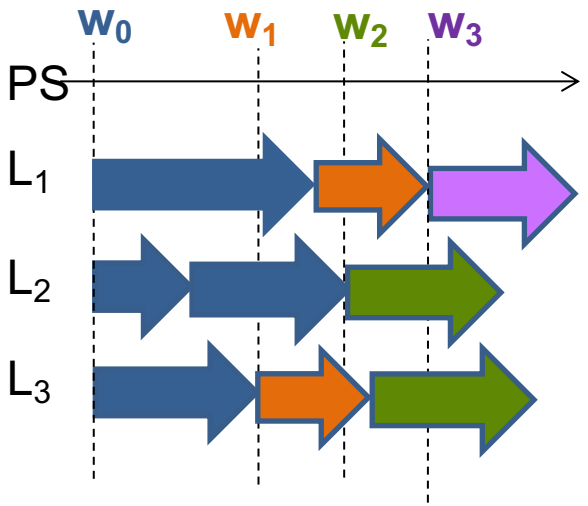


K-async SGD:
Expected Runtime per iteration: In general not tractable
Special Case: New-Longer-Than-Used X_l

$$\mathbb{E}[T] \leq \mathbb{E}[X_{K:P}]$$

$$\underbrace{\mathbb{E}[T]}_{K\text{-th statistic of remaining times } Y_l} \leq \underbrace{\mathbb{E}[X_{K:P}]}_{K\text{-th statistic of } X_l}$$

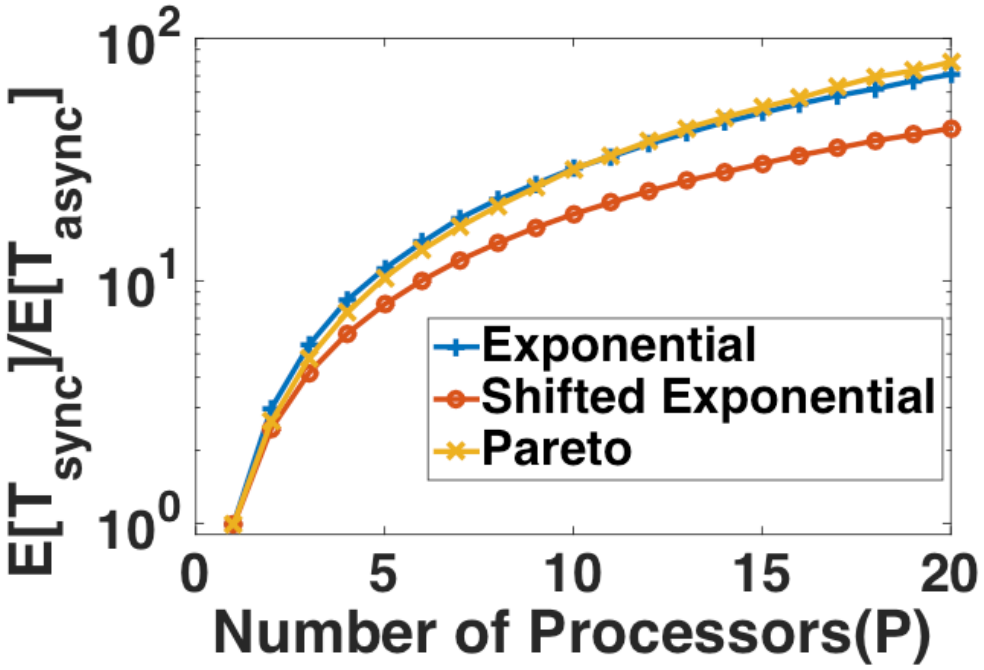
K-Batch Async SGD



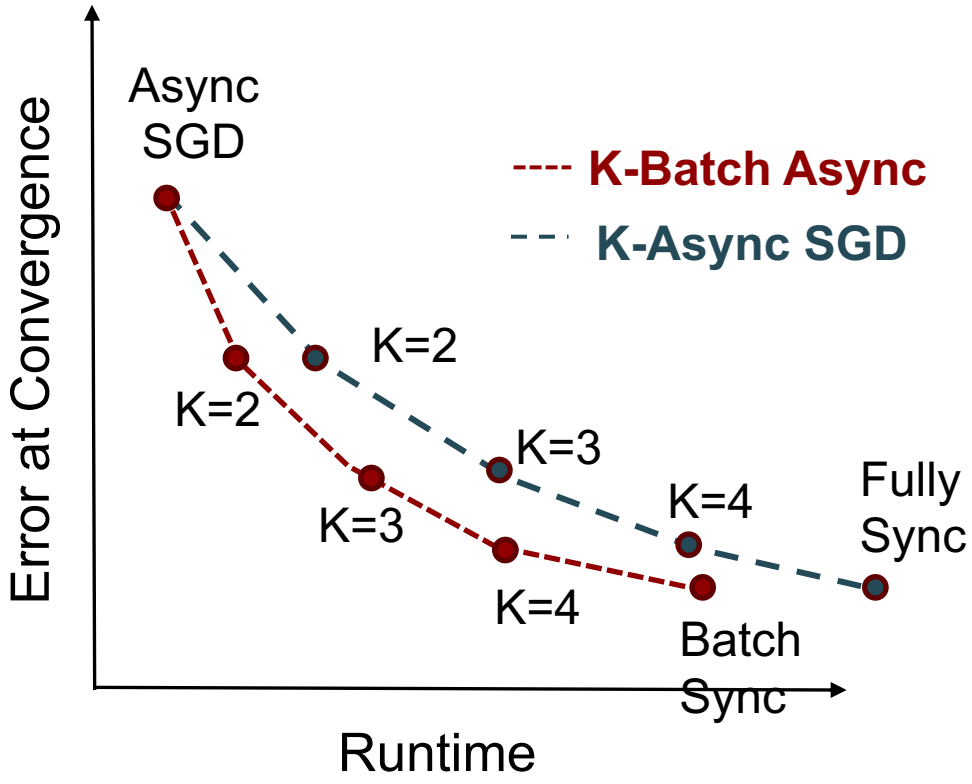
K-batch-async SGD:
Expected Runtime per iteration: $\mathbb{E}[T] = \frac{K}{P}\mathbb{E}[X]$

Comparison of Expected Runtime

$$\frac{\mathbb{E}[T_{Sync}]}{\mathbb{E}[T_{Async}]} = P\frac{\mathbb{E}[X_{P:P}]}{\mathbb{E}[X]} \overset{X_l \sim \exp(\mu)}{\approx} P \log P$$



Error – Runtime Tradeoff

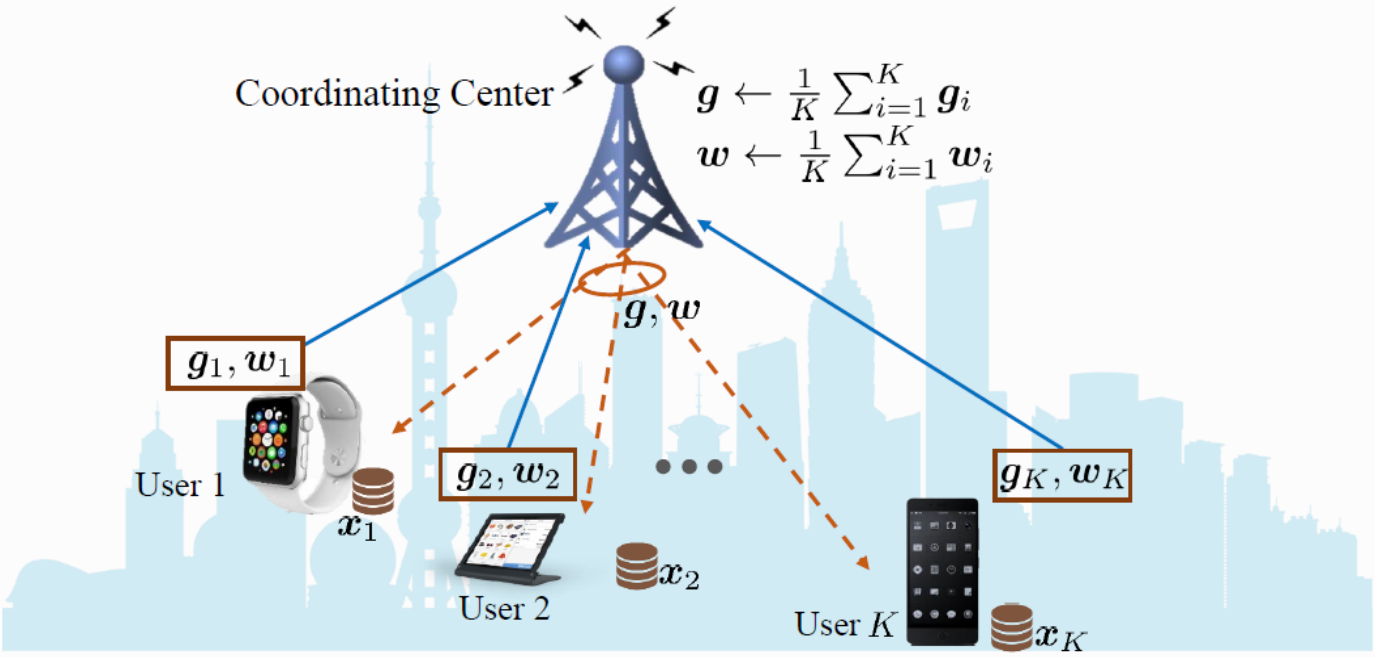


https://arxiv.org/abs/1803.01113

Communication-efficient Algorithms

Second-order Methods

- Exploit second order curvature information
- Exact second-order methods require the computation, storage and even communication of a Hessian matrix (huge overhead)



Two main approaches

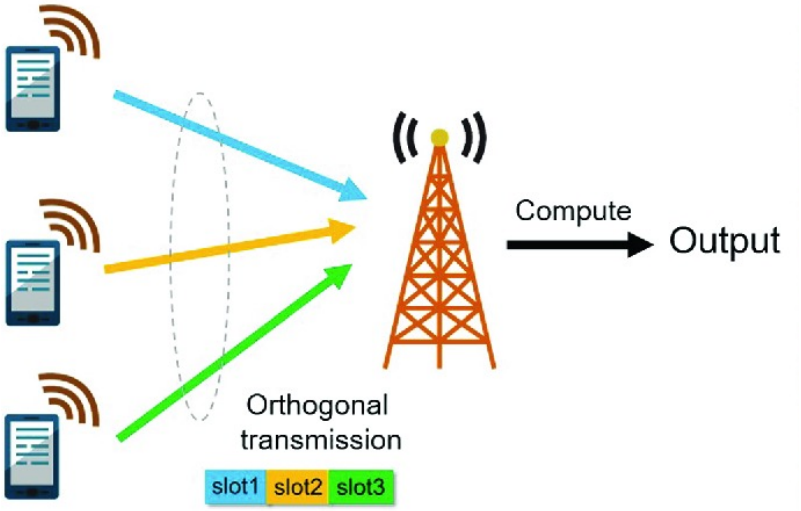
- Maintain a global approximated inverse Hessian matrix in the central node
- Solve a second-order approximation problem locally at each device.

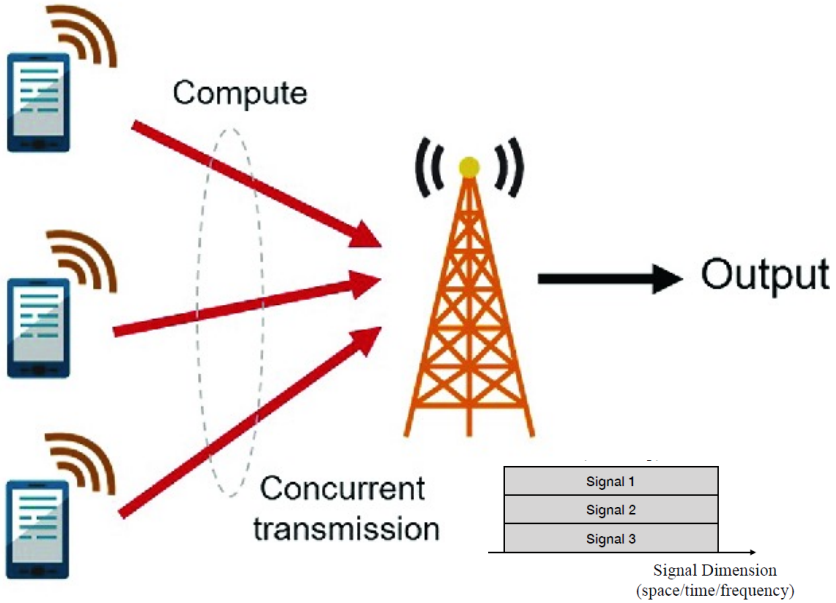
Local computation: Approximate Newton-type update

$$g_i = l'(x_i^\top w)x_i, w_i^{(t)} = \arg\min_w [l(x_i^\top w) - (g_i^{(t-1)} - \eta g^{(t-1)})^\top w + \frac{\mu}{2}\|w - w^{(t-1)}\|_2^2]$$

Communication-Efficient ML Systems

Fast Aggregation with Over-the-Air Computation





Comm. – Computation Separation

- Transmit/Receive, then Compute
- Orthogonal uplink access
- Non-Orthogonal access: interference cancelation at Rx

Over-the-Air Computation (AirComp)

- Wireless shared medium
- Exploit signal superposition property
- Analog transmissions?
- Synchronization!!!

Over-the-Air Computation

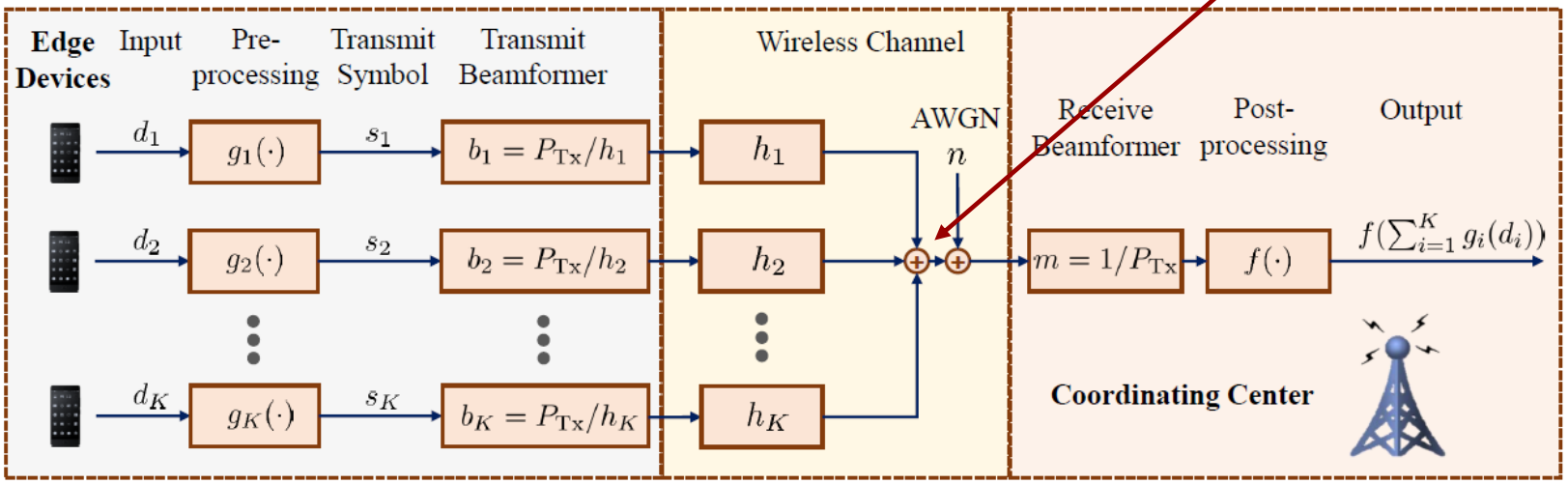
Key idea

- Interference can be harnessed for computing functional values instead of being canceled
  $\Rightarrow$ efficient radio resource utilization

Main steps

- Pre-processing the input data at each device
- Decode directly the sum of all preprocessed data
- *Desired function* is obtained by post-processing the summation

Main issue: Synchronization



Function Computation

What kind of functions can we compute over the air?

Nomographic functions

Definition: Let $\mathbb{A}$ be any metric space and $K \geq 2$. Then, a function $f : \mathbb{A}^K \to \mathbb{R}$ for which there exist functions $\{\varphi_i \in \mathrm{F}(\mathbb{A})\}_{i=1}^{K}$ and $\psi \in \mathrm{F}(\mathbb{R})$ such that f can be represented in the form

$$f(x_1, \dots, x_K) = \psi\left(\sum_{i=1}^{K} \varphi_i(x_i)\right)$$

is called nomographic function.

$\mathrm{F}(\mathbb{A}^\ell)$: space of every function $f : \mathbb{A}^\ell \to \mathbb{R}$

- Examples: arithmetic mean, weighted sum, geometric mean, polynomial, Euclidean norm
- $\{\varphi_i \in \mathrm{F}(\mathbb{A})\}_{i=1}^{K}$: preprocessing functions, ψ: post-processing function

Kolmogorov – Arnold representation (superposition) theorem [1957]
every multivariate continuous function can be represented
as a superposition of continuous functions of one variable

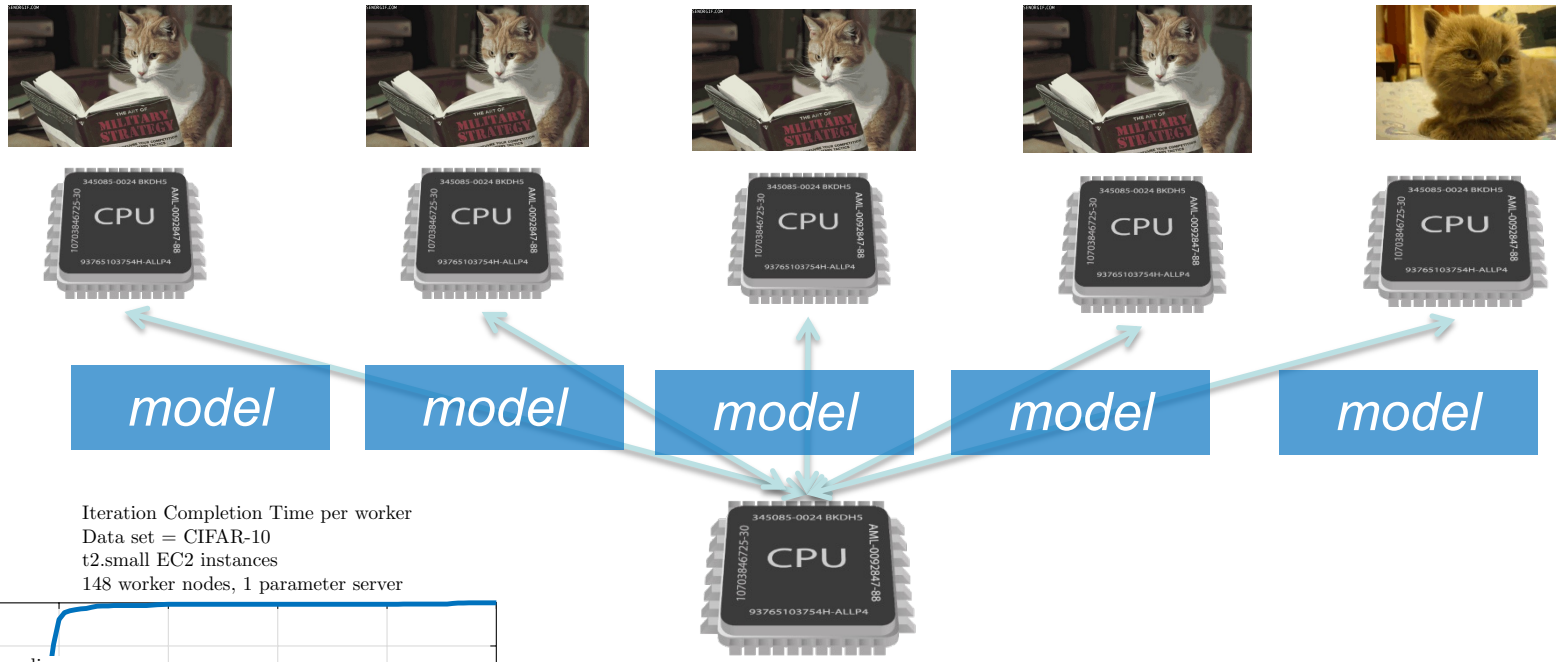
- Lorentz's refinement implication: every continuous function of *K* variables is computable by a wireless (**extended**) MAC with continuous preprocessing functions using a 1-way computation network

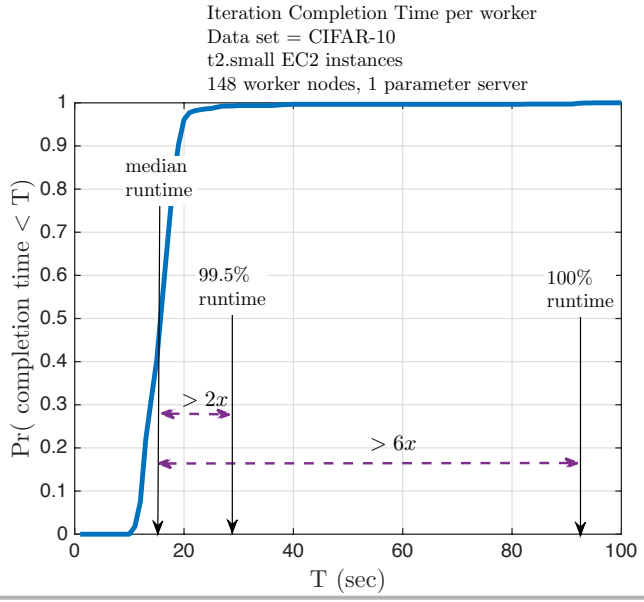
Conventional vs. ML-driven Communication

Comm. Techniques	Conventional	ML-driven
Multiple Access	Decoupling messages from users	Computing functions of distributed data
Resource Allocation	Maximize sum rate or reliability	Fast acquisition (importance aware tx)
Signal Encoding	Optimal tradeoff between rate and distortion/reliability	Latency minimization while preserving learning accuracy

Straggler Mitigation

Straggling Learners





How to deal with stragglers?

Stragglers

- Ideal compute time per node ~ O(total_time/P)

- But there is a lot of randomness:
  - Network/Comm Delays
  - Node/HW Failures
  - Resource Sharing

- What if time per node is a random variable: $X = \text{constant} + \text{Exp}(\lambda)$

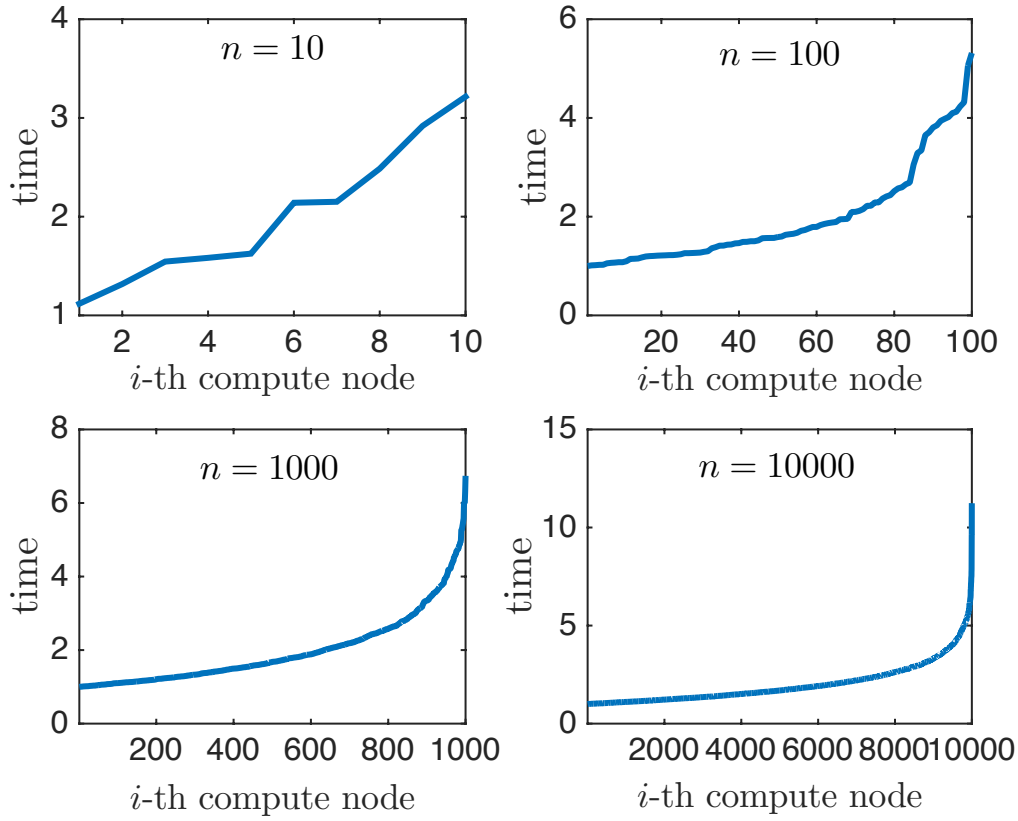
The expected runtime of the i fastest nodes

$$\mathbb{E}\{X_{(i)}\} = 1 + \frac{1}{\lambda} \sum_{n-i+1}^{n} \frac{1}{i}$$

Slowest node is $\log(n)$ times slower than fastest

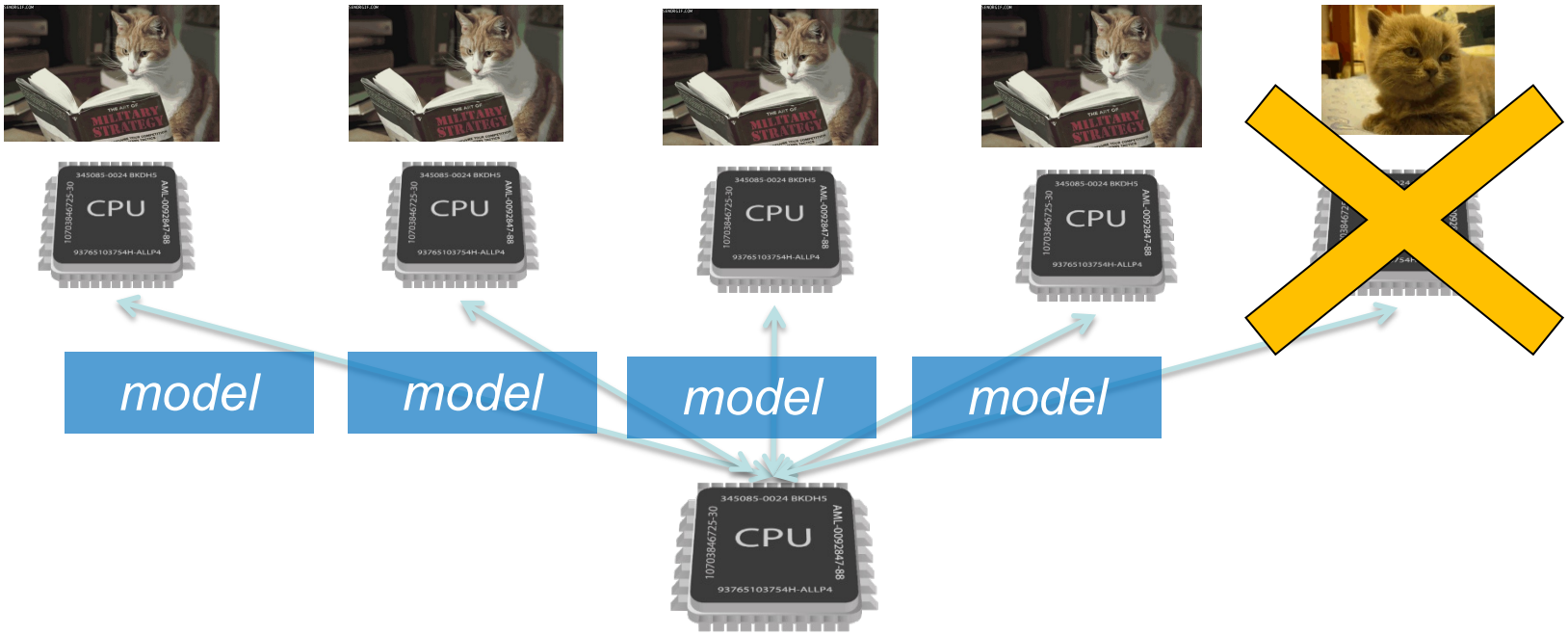
Simulation Results

- $X(t) = 1 + \text{Exp}(0.5)$



Straggler issue:
leads to slower mini-batch SGD implementations

Straggler Termination



- How many nodes to wait for before moving on to next iteration?
- How much does the error affect us?

Straggler Termination: Expected Savings

- Define the runtime of each node: $X = 1 + Y$, where $Y \sim \text{Exp}(\lambda)$

The expected runtime of one node is

$$\mathbb{E}\{X\} = 1 + \frac{1}{\lambda}$$

- Define $X_{(k)}$ to be the *k*-th fastest runtime

The expected runtime of the *k* fastest nodes

$$\mathbb{E}\{X_{(k)}\} = 1 + \frac{1}{\lambda} \sum_{j=n-k+1}^{n} \frac{1}{i} = 1 + O\left(\frac{1}{\lambda} \log\left(\frac{n}{n-k}\right)\right)$$

- Proof: **HW** ☺ *Hint:* $X_{(k)} - X_{(k-1)} \sim \text{Exp}(\lambda(n-k+1))$

Straggler Termination: Expected Savings

- If we wait for the k fastest workers, the runtime is

$$\mathbb{E}\{X_{(k)}\} = 1 + O\left(\frac{1}{\lambda}\log\left(\frac{n}{n-k}\right)\right)$$

- If we wait for all the runtime is

$$1 + O\left(\frac{1}{\lambda}\log\left(n\right)\right)$$

- Speedup factor: $\dfrac{\log(n)}{\log(n/(n-k))}$ X *faster*

How much faster did distributed SGD become?

Straggler Termination: Expected Savings

- Say we wait for $k = (1-\varepsilon)n$ of all nodes
- Each iteration becomes $\frac{\log(n)}{\log(1/\epsilon)}$ X faster
- But we have to perform ~ $1/\epsilon$ more iterations
- E.g.: terminating 50% of nodes $\Rightarrow$ 2x more iterations, 3.3$\cdot \log(n)$ x faster

Can we do better?

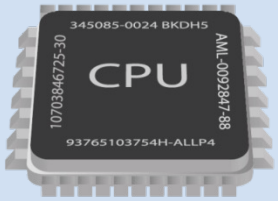
Replication & Backup Workers

- What if we used some redundancy??

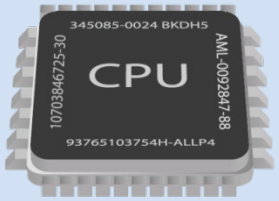
$\nabla f_1(w)$

$\nabla f_2(w)$

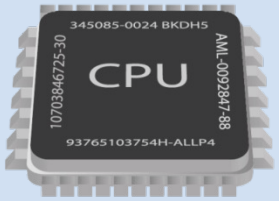
$\nabla f_3(w)$



$\nabla f_1(w)$



$\nabla f_2(w)$



$\nabla f_3(w)$

- **No loss in accuracy**
- **Seems wasteful**

Replication: Expected Savings

- For every gradient i, the time that the PS will receive it is a RV

$$X_i = \min \text{ time of all nodes computing } i$$

- X_i distribution replicating r-times: $X_i \sim 1 + \text{Exp}(r \cdot \lambda)$

- The expected runtime of r-replication

$$1 + \frac{1}{r\lambda} O(\log(n))$$

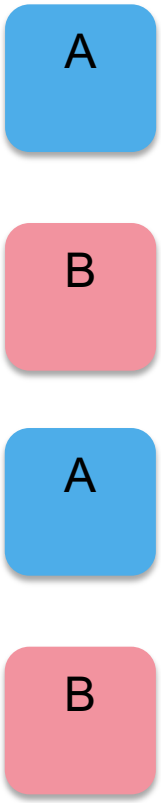
Comparison of expected times

- Waiting for all nodes: $1 + \frac{1}{\lambda} O(\log(n))$
- Waiting for the $(1-\varepsilon)n$ fastest: $1 + \frac{1}{\lambda} O\left(\log(1/\epsilon)\right)$
- Replicating r times: $1 + \frac{1}{r\lambda} O(\log(n))$

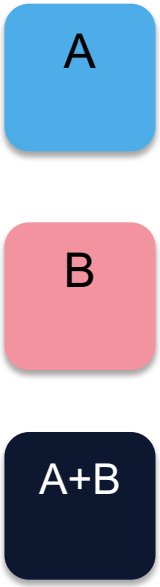
Either lose information OR compute too much
Can we do better?

Coding Theory to the Rescue

Replication vs. Codes



Can recover A and B from any 1 erasure

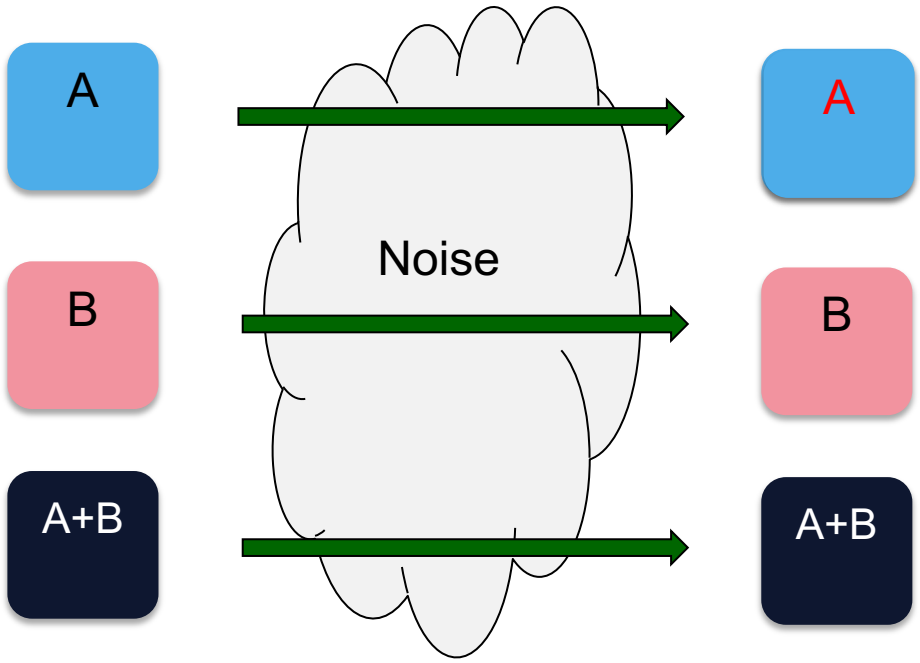


Can also recover A, B from any 1 erasure

How much redundancy for r erasures?

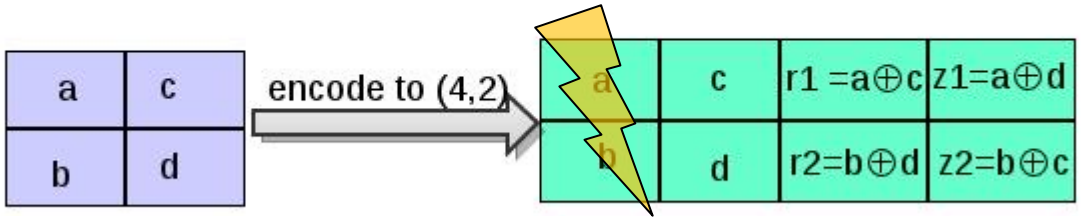
Replication: factor of r Codes: factor of $1 + r/n$

Coding Theory to the Rescue

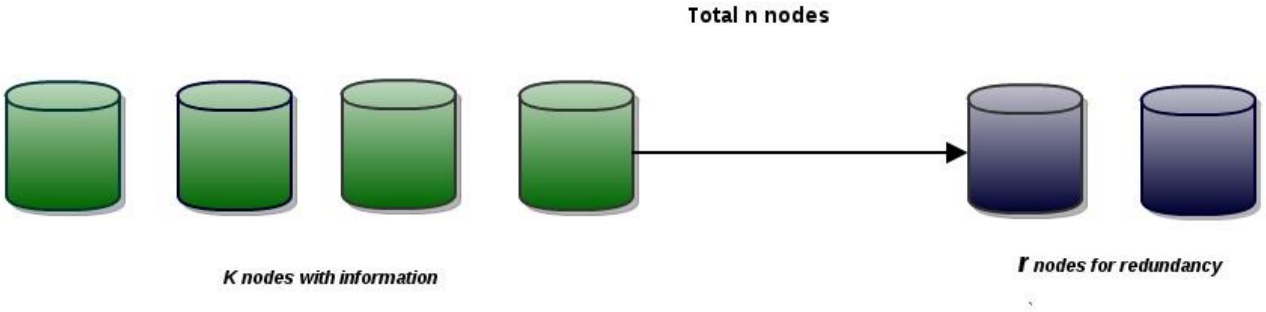


- **Coding**: system of rules to convert information into another form of representation
- Encoding: process converting information from a source into symbols for communication or storage.
- Decoding: the reverse process, converting code symbols back into a form that the recipient understands.

MDS Codes in a Nutshell



- 4 elements in 2 nodes encoded into 4 nodes of which 2 nodes are used for redundancy.
- To recover a, elements c and $a \oplus c$ are used.
- To recover b, elements c and $c \oplus b$ are used.
- Out of the 6 surviving elements, only 3 are used to recover lost information.
- Rebuilding ratio is equal to 3/6=1/2

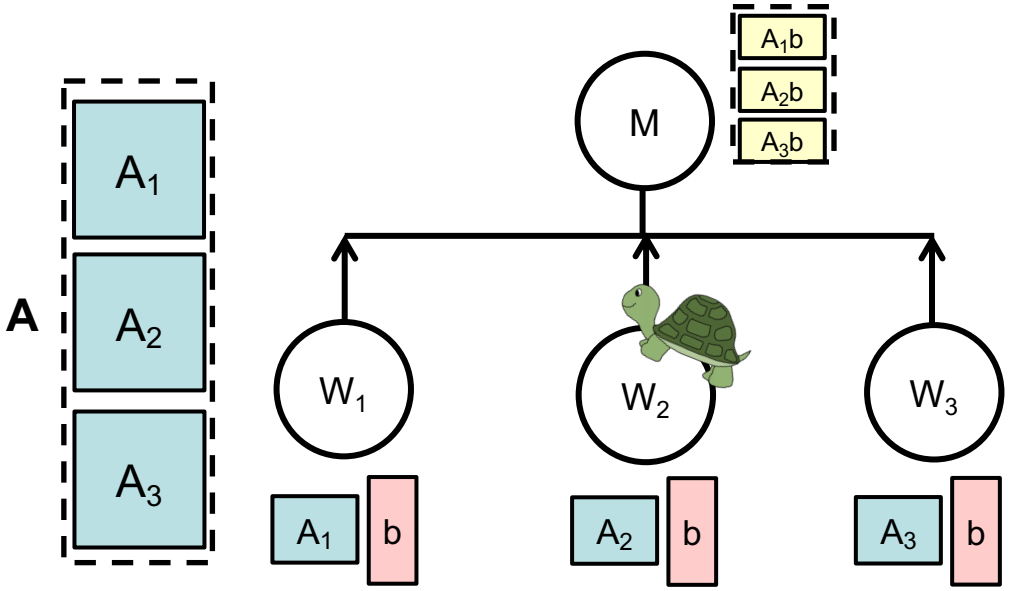


- (n, k)- MDS code: n is the total # nodes of which
  k is #nodes containing information
  $r = n - k$ nodes as redundancy

Distributed Matrix-Vector Multiplication

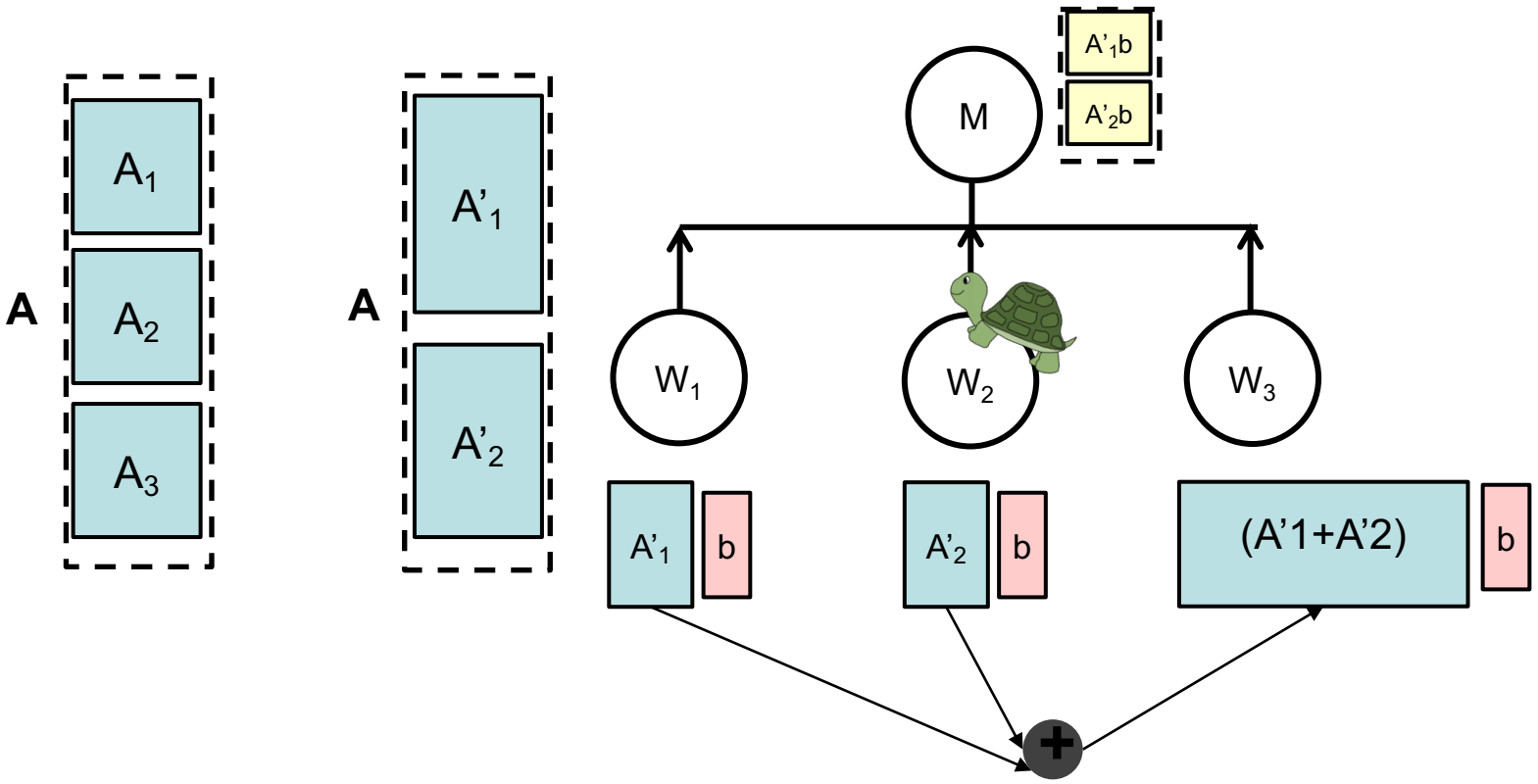
- **Task:** multiply A and b
- **Distributed setup:** 3 workers, 1 master

$$\mathbf{A}\times\mathrm{b} = \begin{pmatrix} A_1 \\ A_2 \\ A_3 \end{pmatrix} \times \mathrm{b} = \begin{pmatrix} A_1\times\mathrm{b} \\ A_2\times\mathrm{b} \\ A_3\times\mathrm{b} \end{pmatrix}$$



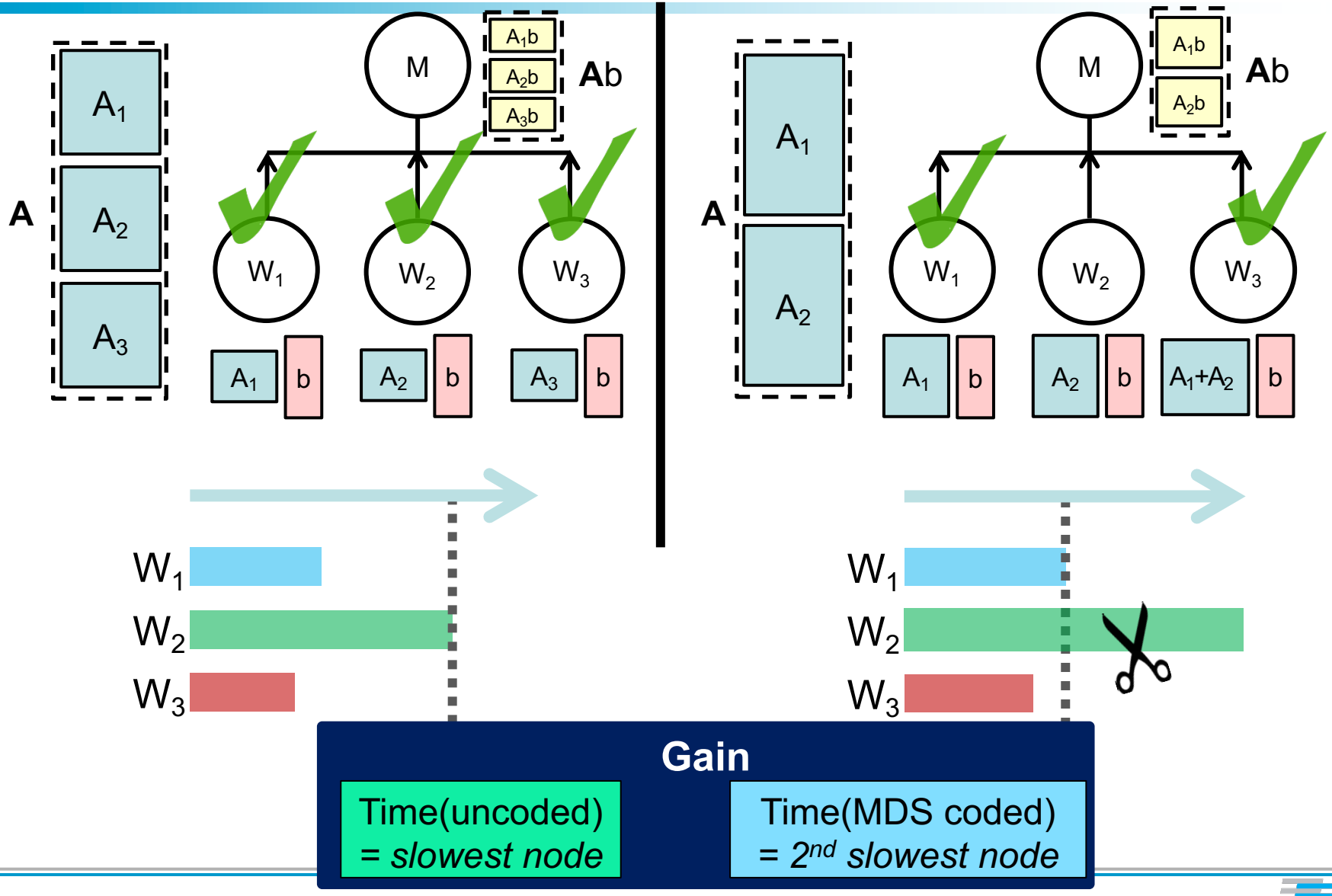
Uncoded Algorithm

Coded Matrix Multiplication



Coded Algorithm

MDS-Coded Matrix Multiplication



Comparison of expected times

- Waiting for all nodes: $1 + \frac{1}{\lambda} O(\log(n))$
- Waiting for the $(1-\varepsilon)n$ fastest [incur loss]: $1 + \frac{1}{\lambda} O\left(\log(1/\epsilon)\right)$
- Replicating r times: $1 + \frac{1}{r\lambda} O(\log(n))$
- $((1+\varepsilon)n, n)$–MDS code [no loss]: $1 + \frac{1}{\lambda} O\left(\log(1/\epsilon)\right)$

Coded Computation for Linear Operations

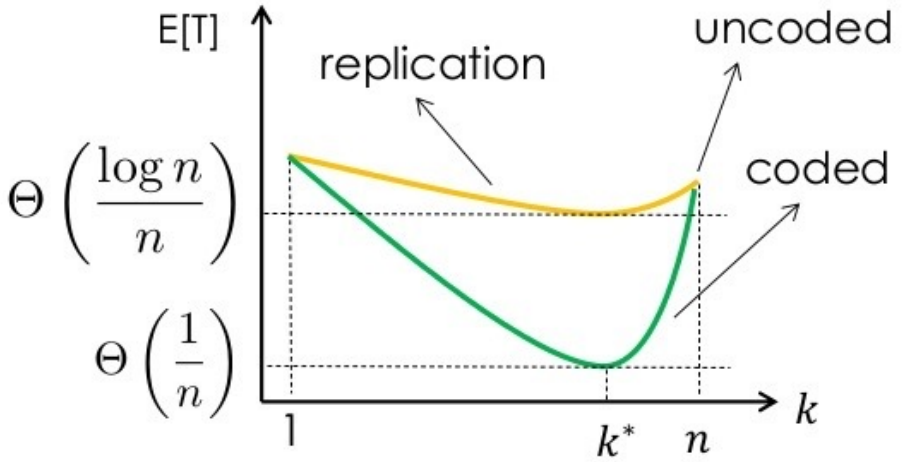
Assumptions:

- n workers
- k-way parallelizable: $f = g(f_1, f_2, \ldots f_k)$
- **Computing time of $\sum_{i=1}^{k} a_i f_i$ = constant + exponential RV (i.i.d.)**
- Average computing time is proportional to $\frac{1}{k}$

Theorem:

$$\mathrm{E}[T_{\text{uncoded}}] = \Theta\left(\frac{\log n}{n}\right) \qquad \mathrm{E}[T^{\star}_{\text{replication}}] = \Theta\left(\frac{\log n}{n}\right)$$

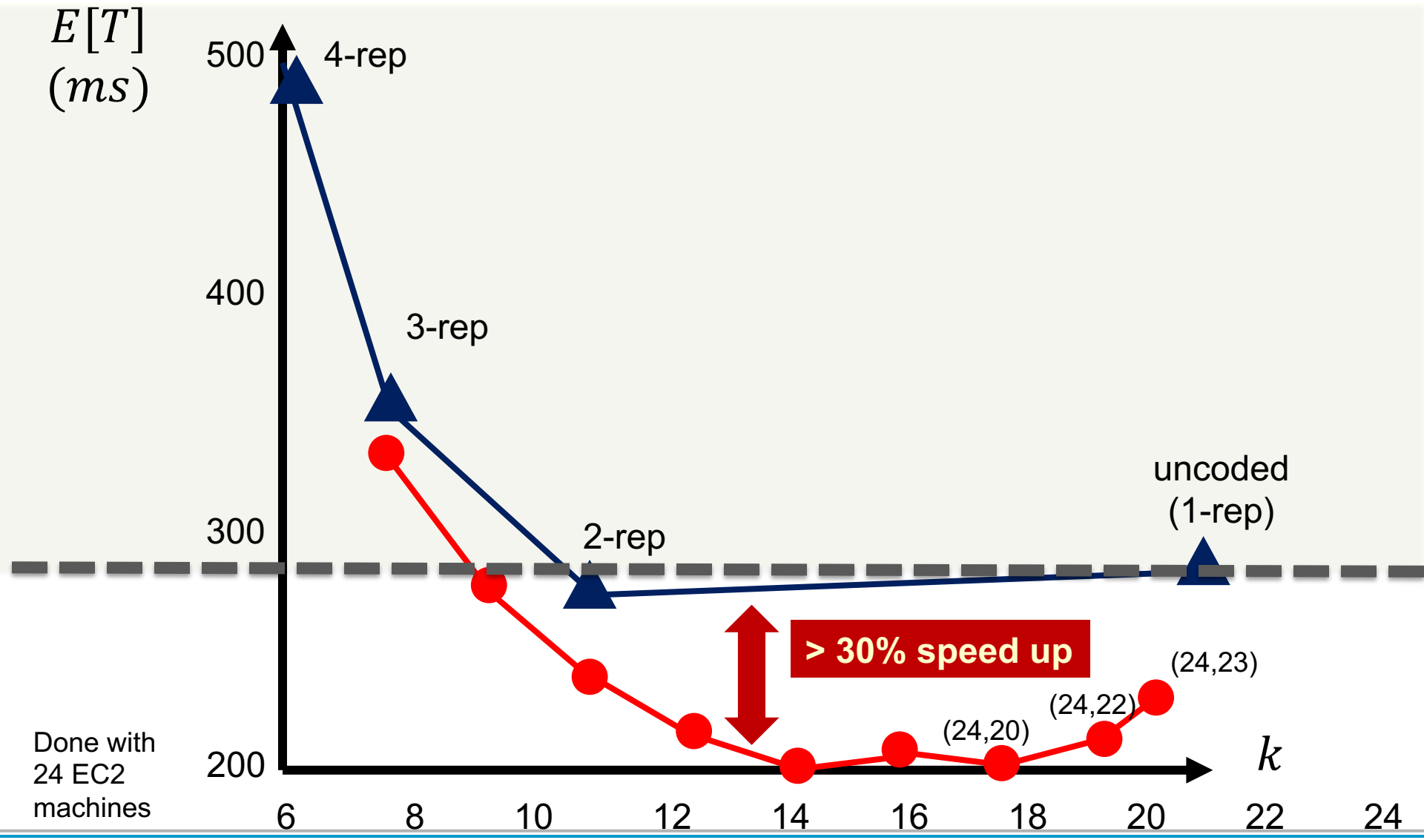
$$\mathrm{E}[T^{\star}_{\text{MDS-coded}}] = \Theta\left(\frac{1}{n}\right)$$



Can be applied to:

- Linear regression
- Fixed point type algorithms (SVD, PageRank, etc.)

Experimental Results



EURECOM
Sophia Antipolis

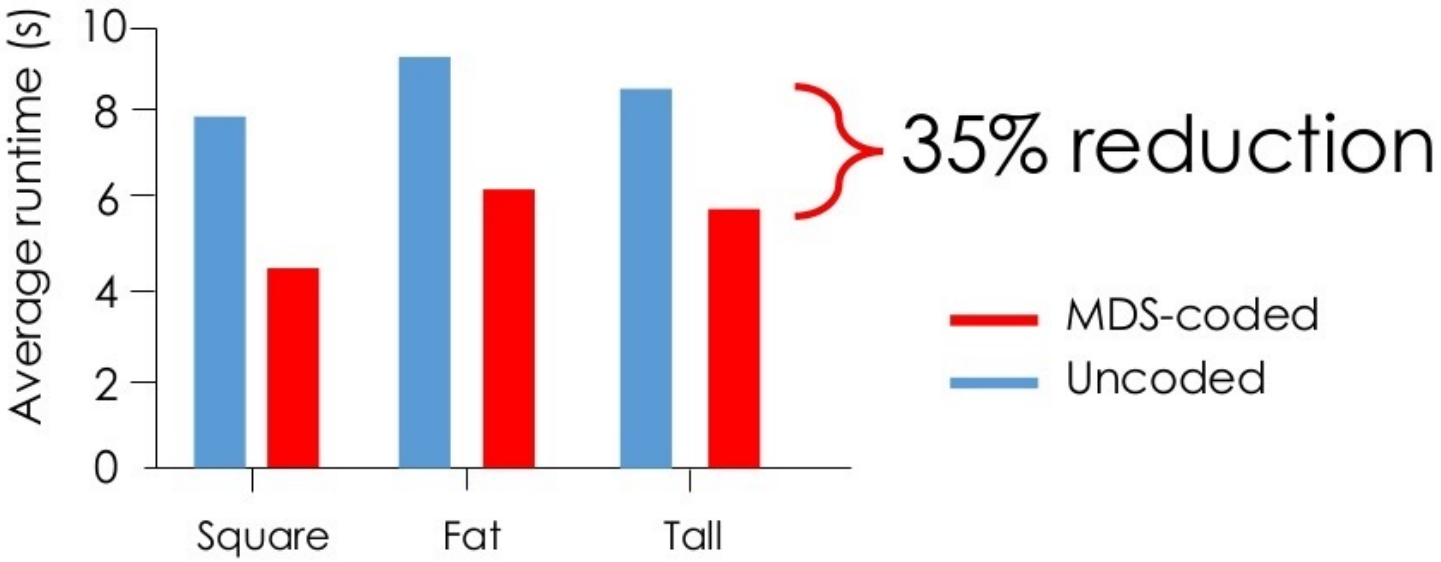
Distributed Linear Regression

Gradient descent for linear regression = Iterative matrix multiplication

$$\theta^{(t+1)} = \theta^{(t)} - \alpha A^T (A\theta^{(t)} - y)$$

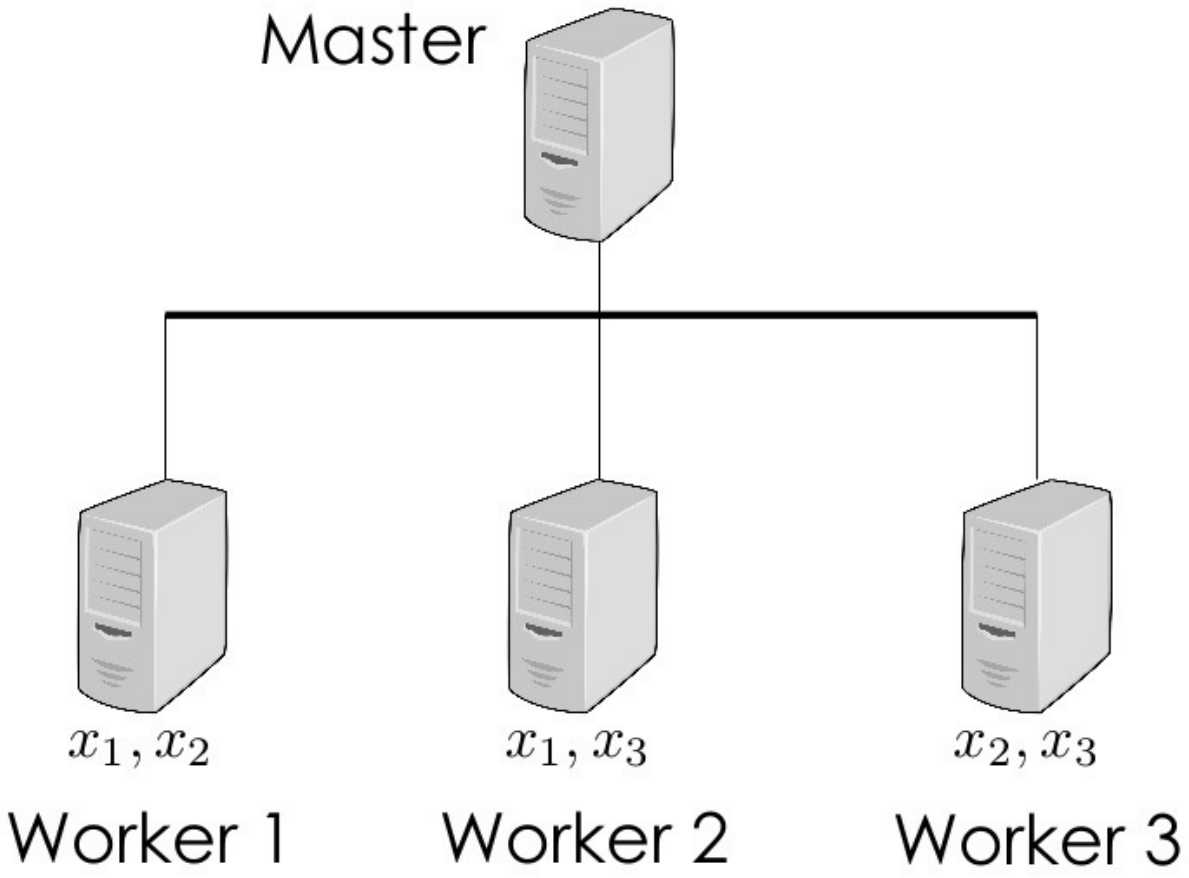
Coded Gradient Descent

= Gradient Descent + Coded Matrix Multiplication



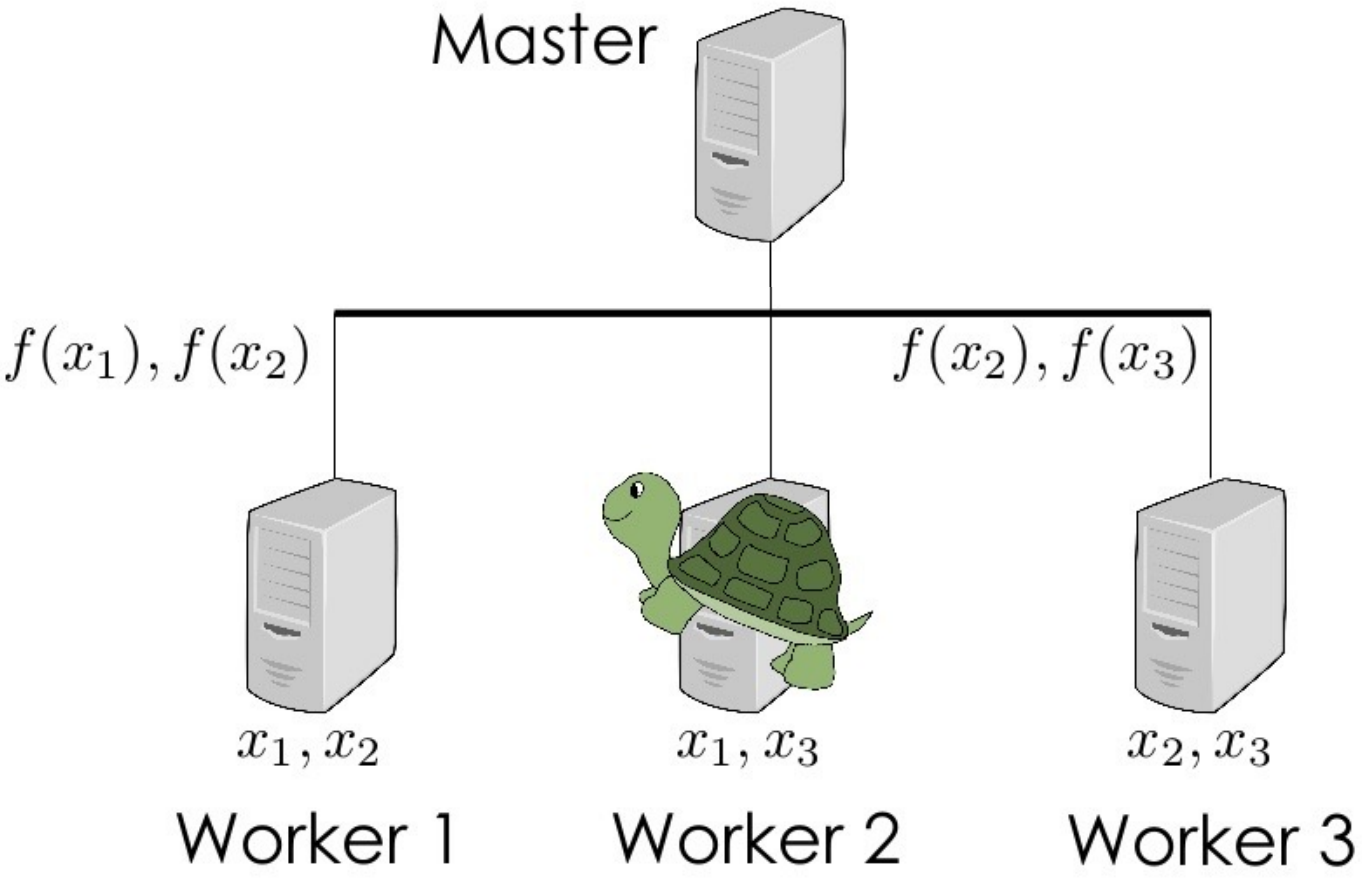
Gradient Coding

Goal: $f(x_1) + f(x_2) + f(x_3)$



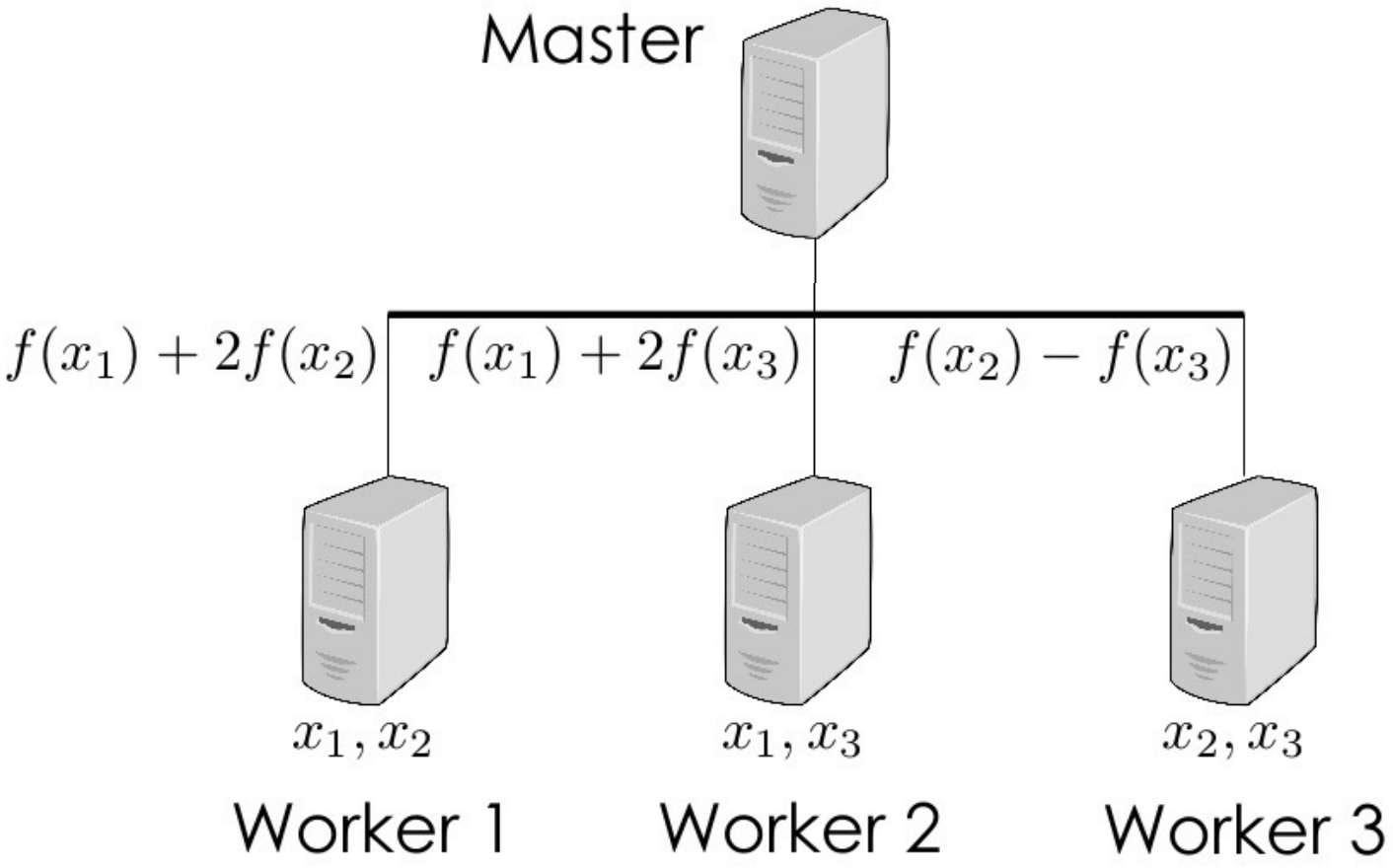
Gradient Coding

Goal: $f(x_1) + f(x_2) + f(x_3)$



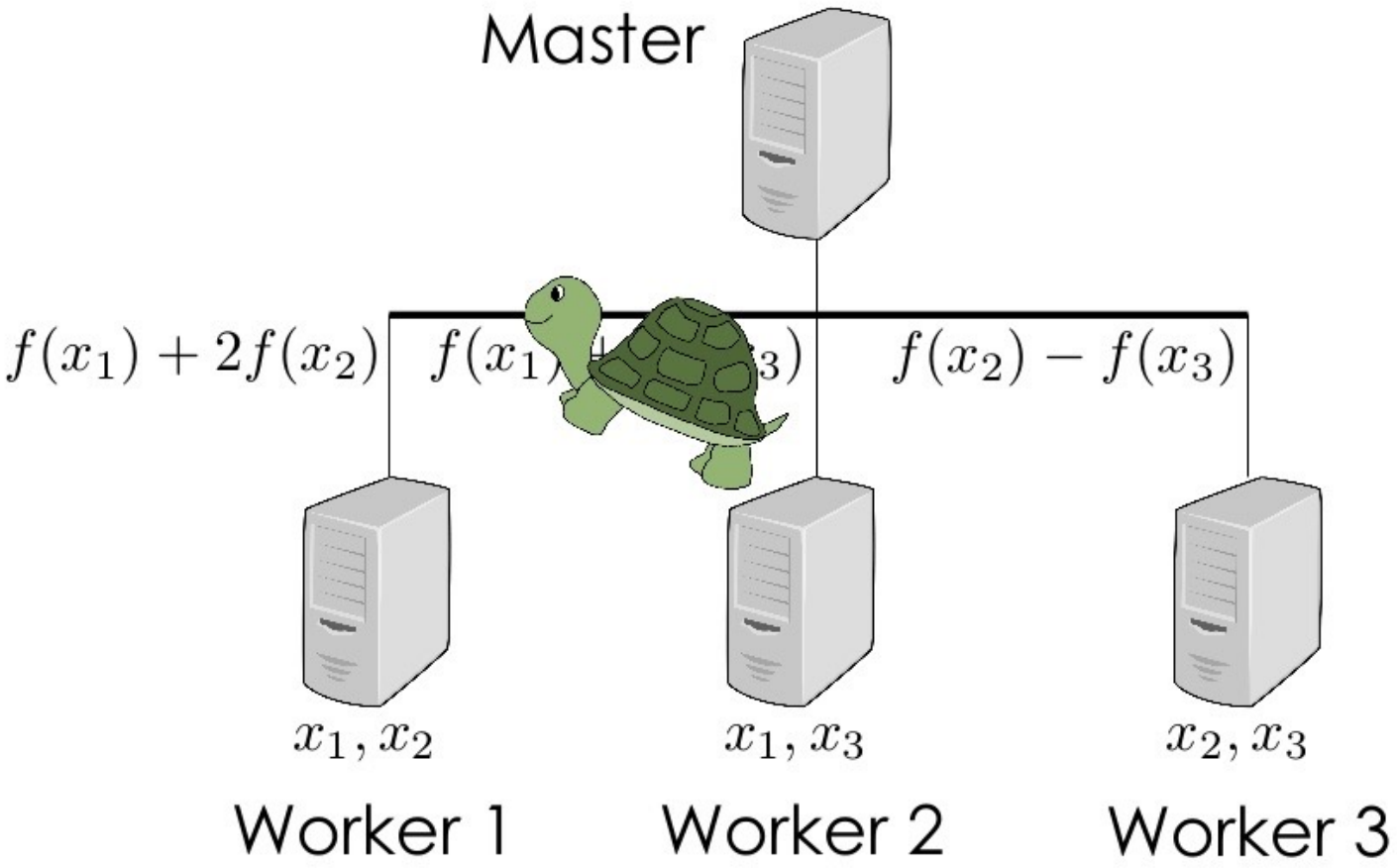
Gradient Coding

Goal: $f(x_1) + f(x_2) + f(x_3)$



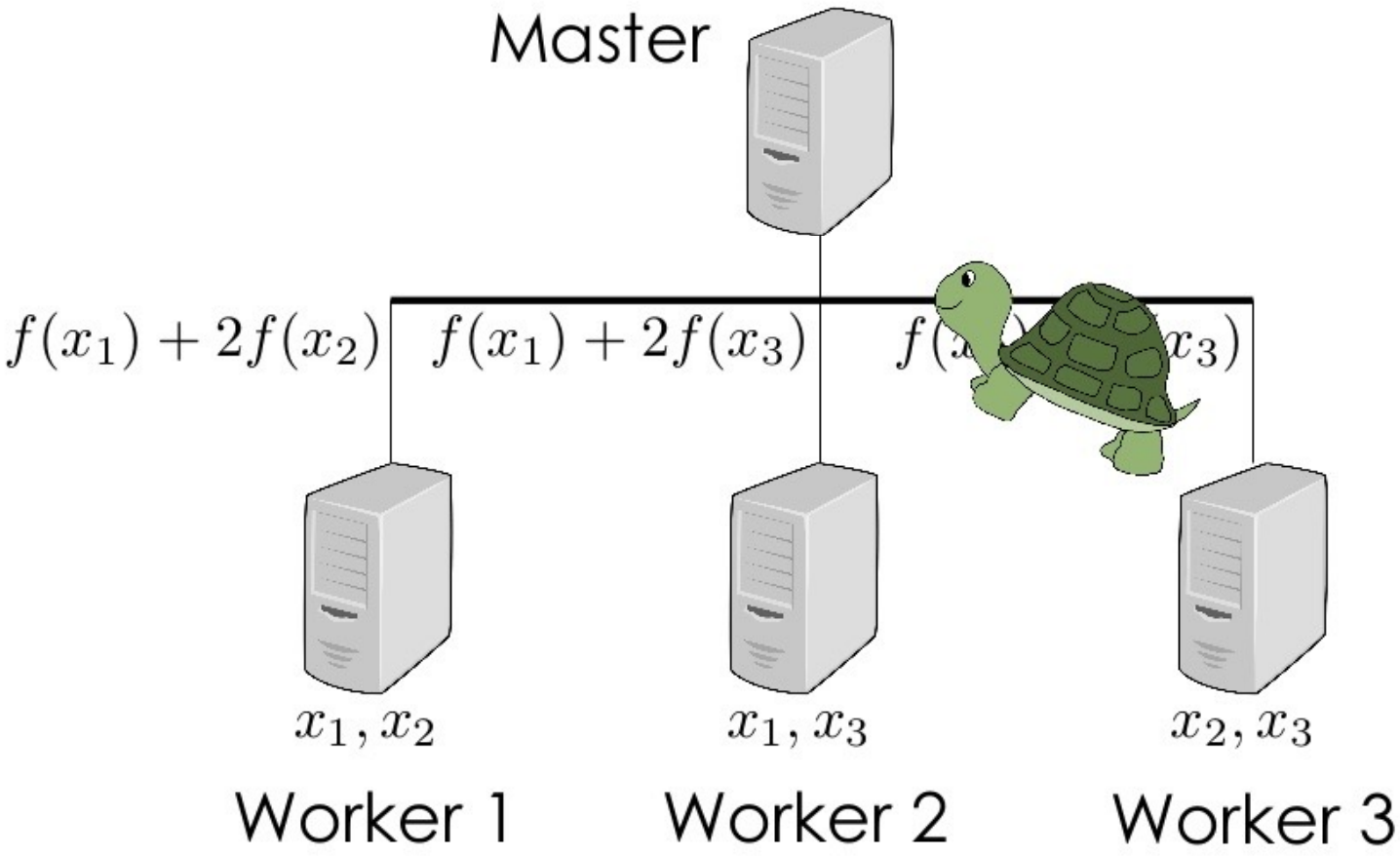
Gradient Coding

Goal: $f(x_1) + f(x_2) + f(x_3)$



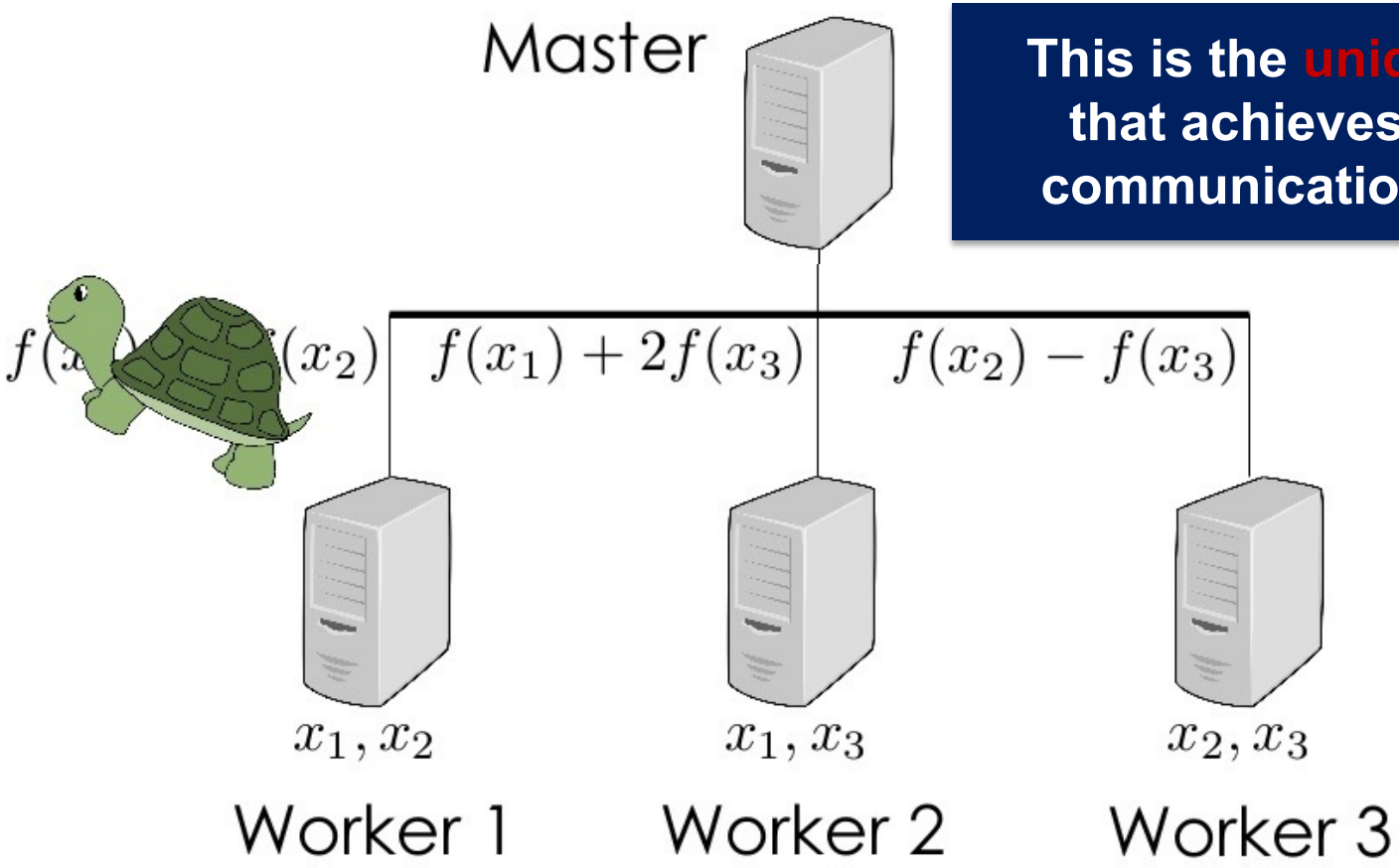
Gradient Coding

Goal: $f(x_1) + f(x_2) + f(x_3)$



Gradient Coding

Goal: $f(x_1) + f(x_2) + f(x_3)$



This is the unique solution that achieves minimum communication overhead

EURECOM
Sophia Antipolis

Federated Learning

What is Federated Learning?

- In a nutshell, broadly speaking…. FL = Privacy-Preserving Collaborative ML keeping the (training) data decentralized

- Instead of making predictions in the cloud, distribute the model, make predictions on device

Key Features:

- On-device datasets: end users keep raw data locally (privacy by default)

- On-device training: end-user devices perform training on a shared model

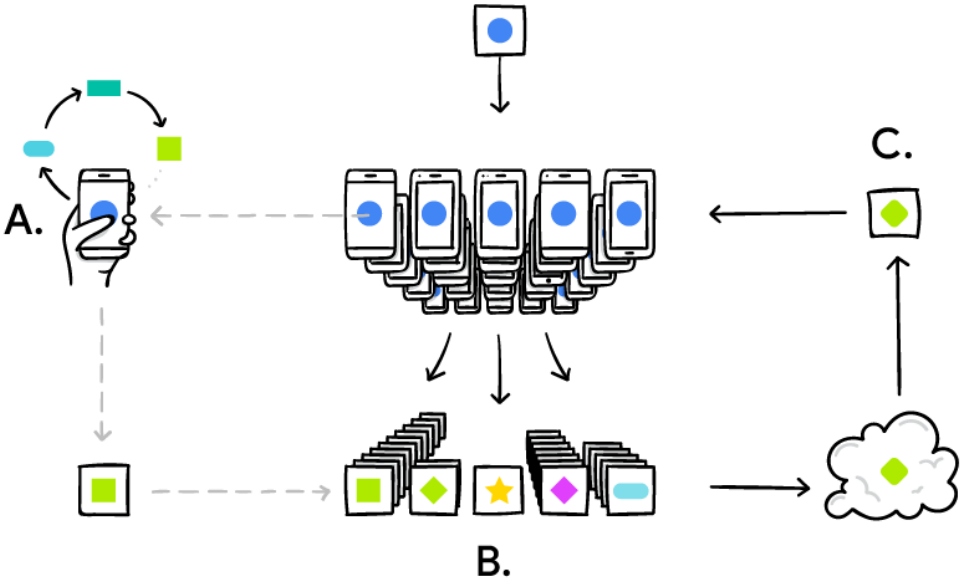
- Federated learning: a shared global model is trained via federated computation

- Federated computation: server collects trained weights from end users and update the shared model (i.e., coordinates a fleet of participating devices to compute aggregations of devices' private data)

Basic Federated Learning Operation

Iterate the following process till convergence

- **At devices**: perform local training, send the trained weights to aggregator
- **At aggregator**: average the model, redistribute to the devices



(A) device (e.g. phone) personalizes the model locally, based on its user usage (local dataset)

(B) many users' updates are aggregated

(C) to form a consensus change to the shared model

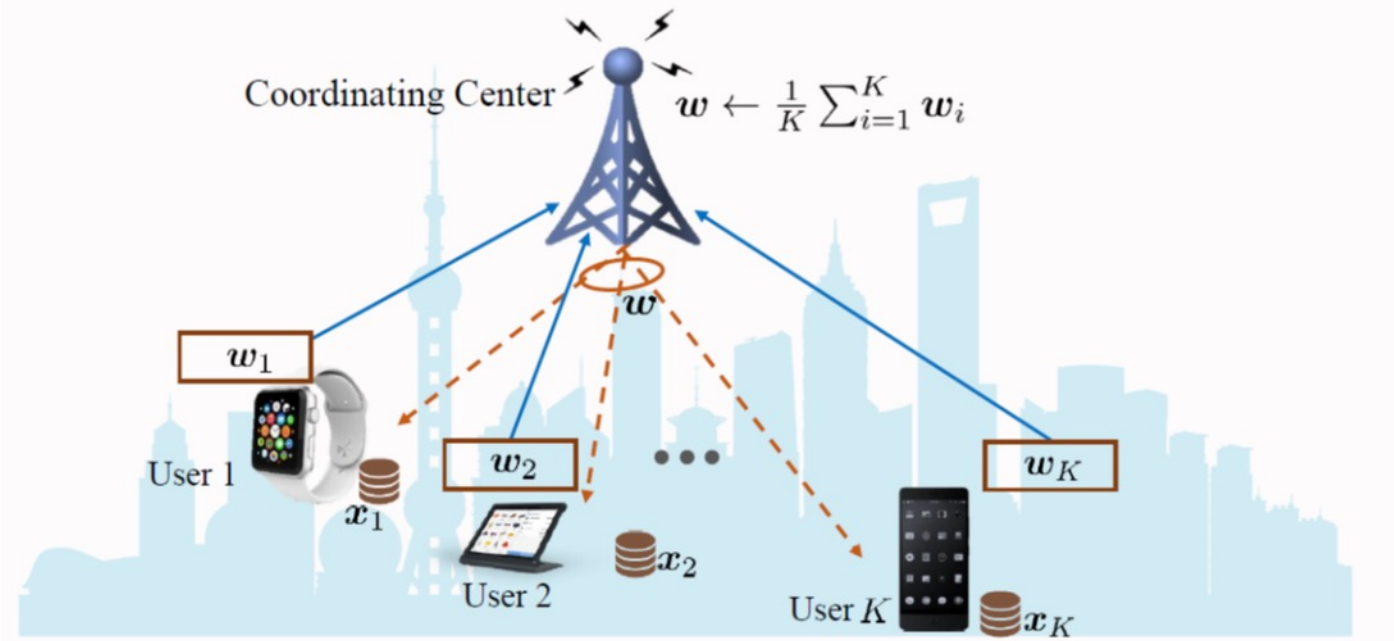
… and the procedure is repeated

We want the final model to be:

- as good as the centralized solution (ideally)
- at least better than what each party can learn on its own

Federated Optimization

- Iteratively perform a local training algorithm (e.g., multiple steps of SGD) based on the dataset at each device
- Aggregate the local updated models, i.e., compute the average (or weighted average) of the local updated model parameters.



Local computation: multiple steps of (stochastic) gradient descent

$$\boldsymbol{w}^{[0]} = \boldsymbol{w}, \boldsymbol{g}_i^{[j]} = l'(\boldsymbol{x}_i^{\mathsf{T}}\boldsymbol{w}^{[j]})\boldsymbol{x}_i, \boldsymbol{w}_i^{[j+1]} = \boldsymbol{w}_i^{[j]} - \eta \boldsymbol{g}_i^{[j]} \text{ for } j = 0, \cdots, L-1$$

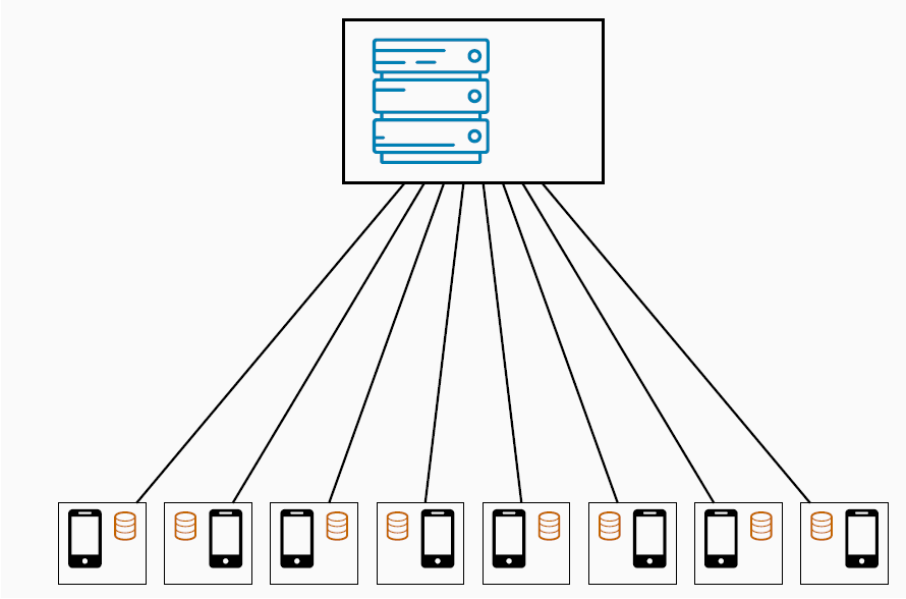
Federated Learning vs. Distributed Learning

Characteristics of **federated learning** vs. traditional distributed learning

- Data locality and distribution
  - **massively decentralized, naturally arising (non-IID) partition**
  - Data is siloed, held by a small number of coordinating entities
  - system-controlled (e.g. shuffled, balanced)
- Data availability
  - **limited availability, time-of-day variations**
  - almost all data nodes always available
- Addressability
  - **data nodes are anonymous and interchangeable**
  - data nodes are addressable
- Node statefullness
  - **stateless (generally no repeat computation)**
  - statefull
- Node reliability
  - **unreliable (~10% failures)**
  - reliable
- Wide-area communication pattern
  - **hub-and-spoke topology**
  - peer-to-peer topology (fully decentralized)
  - none (centralized to one datacenter)
- Distribution scale
  - **massively parallel (1e9 data nodes)**
  - single datacenter
- Primary bottleneck
  - **communication**
  - computation

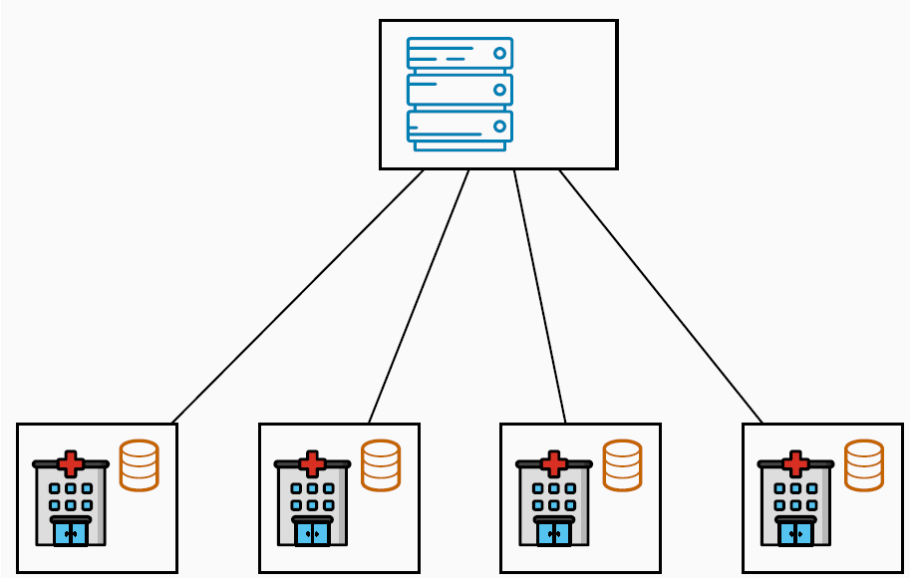
Federated Learning Variants

Cross-device FL



- Massive number of parties (up to 10^{10})
- Small dataset per party (could be size 1)
- Limited availability and reliability
- Some parties may be malicious

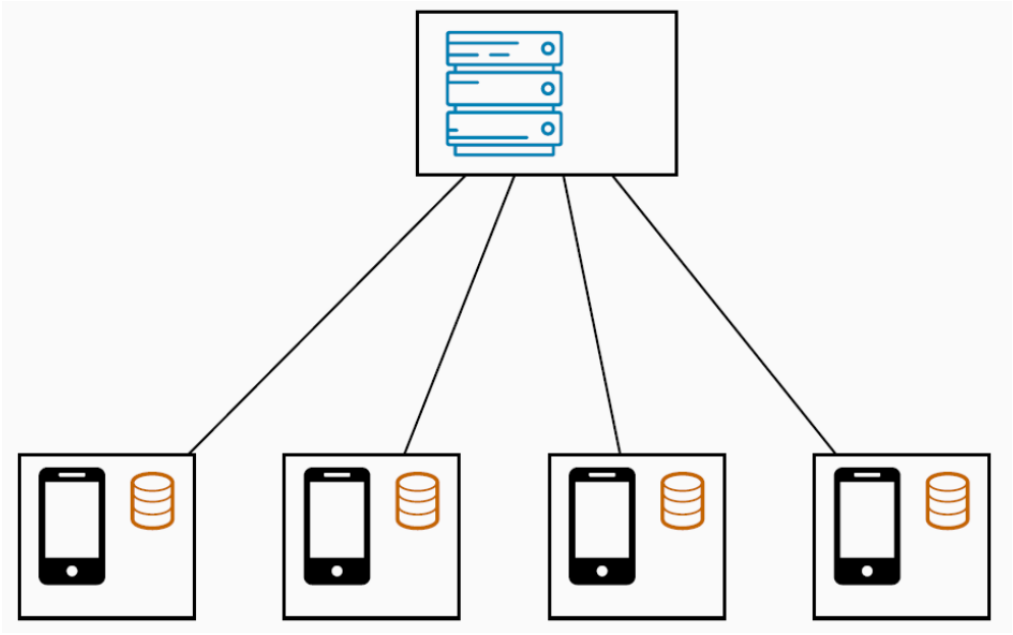
Cross-silo FL



- 2 - 100 parties
- Medium to large dataset per party
- Reliable parties, almost always available
- Parties are typically honest

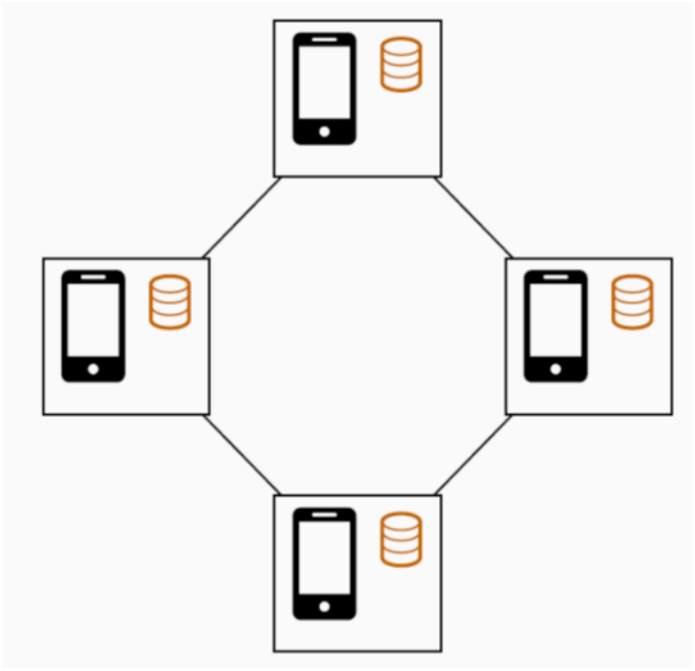
Federated Learning Variants

Server-orchestrated FL



- Server-client communication
- Global coordination, global aggregation
- Server is a single point of failure and may become a bottleneck

Fully decentralized FL



- Device-to-device communication
- No global coordination, local aggregation
- Naturally scales to a large number of devices

Federated Learning Challenges

- Massively Distributed
  - Training data is stored across a very large number of devices
- Limited Communication
  - Only a handful of rounds of unreliable communication with each devices
- Unbalanced Data
  - Some devices have few examples, some have orders of magnitude more
- Highly Non-IID Data
  - Data on each device reflects one individual's usage pattern
- Unreliable Compute Nodes
  - Devices go offline unexpectedly; expect faults and adversaries
- Dynamic Data Availability
  - The subset of data available is non-constant, e.g. time-of-day vs. country

FL over Wireless Networks

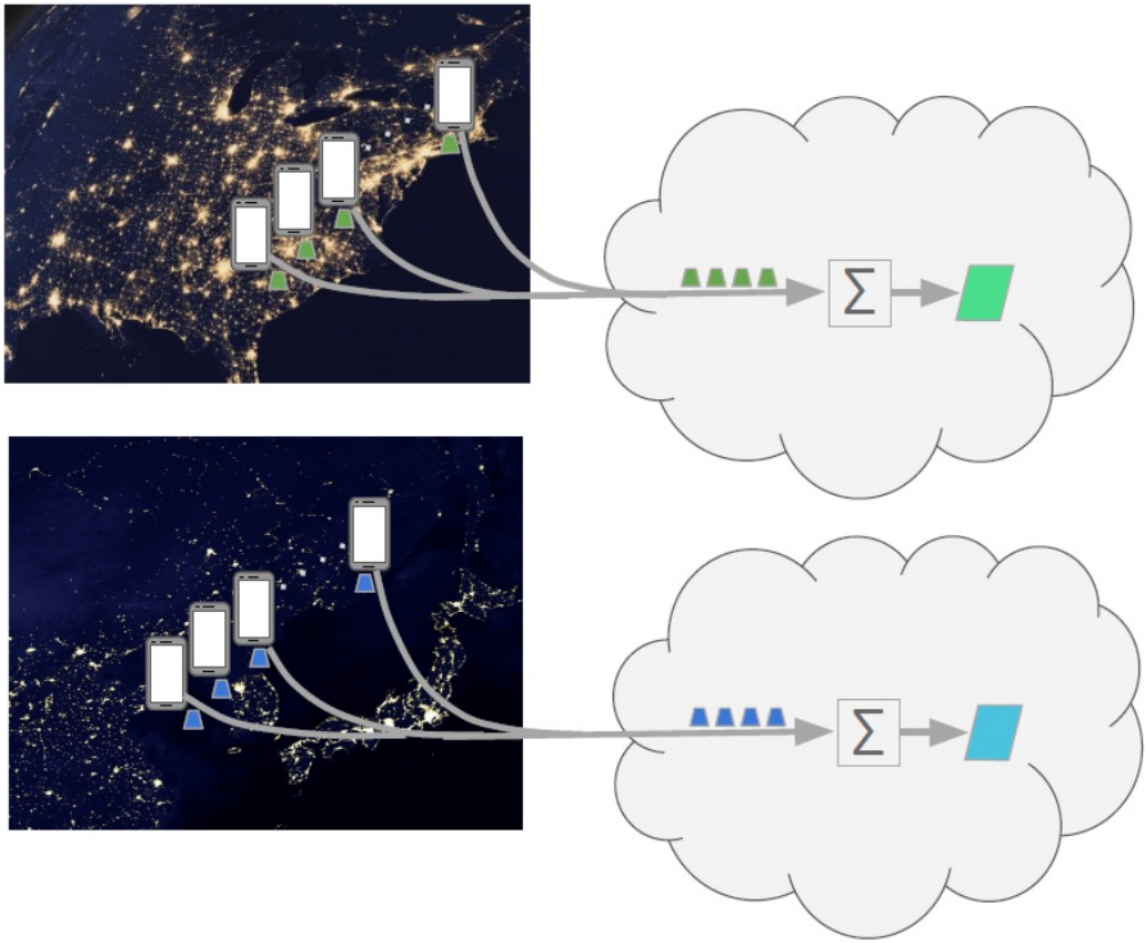
- Communication medium is shared and resource-constrained
  - Only a limited number of devices can be selected in each update round
  - Transmissions are not reliable due to interference

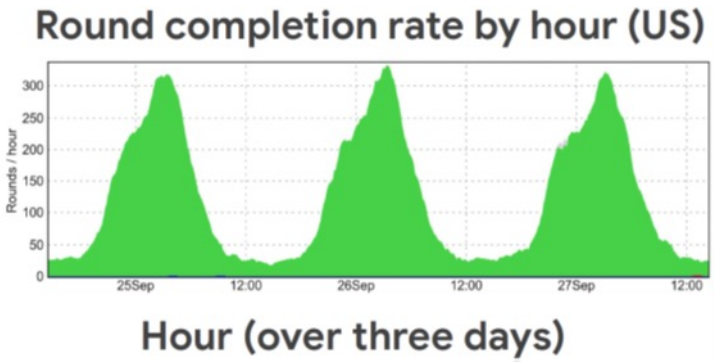
Typical orders-of-magnitude

- 100-1000s of users per round
- 100-1000s of rounds to convergence
- 1-10 minutes per round

Constraints in the federated setting

- Each device reflects one users' data.
- So no one device is representative of the whole population.
- Devices must idle, plugged-in, on WiFi to participate.
- Device availability correlates with both geo location *and* data distribution.





H. Eichner, et al. Semi-Cyclic Stochastic Gradient Descent. ICML 2019.

Federated Averaging

- We consider a set of K parties (clients)
- Each party k holds a dataset $\mathcal{D}_k$ of n_k points
- Let $\mathcal{D} = \mathcal{D}_1 \cup \cdots \cup \mathcal{D}_K$ be the joint dataset and $n = \sum_k n_k$ the total number of points
- We want to solve problems of the form $\min_{\theta \in \mathbb{R}^p} F(\theta; \mathcal{D})$ where:

$$F(\theta; \mathcal{D}) = \sum_{k=1}^{K} \frac{n_k}{n} F_k(\theta; \mathcal{D}_k) \quad \text{and} \quad F_k(\theta; \mathcal{D}_k) = \sum_{d \in \mathcal{D}_k} f(\theta; d)$$

- $\theta \in \mathbb{R}^p$ are model parameters (e.g., weights of a logistic regression or neural network)
- This covers a broad class of ML problems formulated as empirical risk minimization

Federated Averaging

Algorithm FedAvg (server-side)

Parameters: client sampling rate ρ

initialize θ

for each round $t = 0, 1, \ldots$ **do**

  $\mathcal{S}_t \leftarrow$ random set of $m = \lceil \rho K \rceil$ clients

  for each client $k \in \mathcal{S}_t$ in parallel **do**

    $\theta_k \leftarrow \text{ClientUpdate}(k, \theta)$

  $\theta \leftarrow \sum_{k \in \mathcal{S}_t} \frac{n_k}{n} \theta_k$

Algorithm ClientUpdate(k, θ)

Parameters: batch size B, number of local steps L, learning rate η

  for each local step $1, \ldots, L$ **do**

    $\mathcal{B} \leftarrow$ mini-batch of B examples from $\mathcal{D}_k$

    $\theta \leftarrow \theta - \frac{n_k}{B} \eta \sum_{d \in \mathcal{B}} \nabla f(\theta; d)$

  send θ to server

- For $L = 1$ and $\rho = 1$, it is equivalent to classic parallel SGD: updates are aggregated and the model synchronized at each step
- For $L > 1$: each client performs multiple local SGD steps before communicating

Federated Averaging Algorithm

Server

Until Converged:

1. Select a random subset (e.g. 1000) of the (online) clients

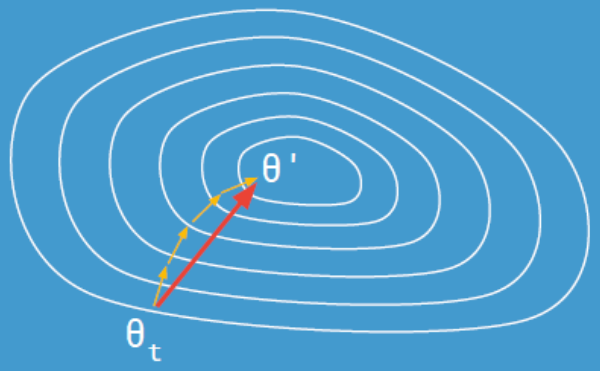
2. In parallel, send current parameters θ_t to those clients

Selected Client k

1. Receive θ_t from server.

2. Run some number of minibatch SGD steps, producing θ'

3. Return $\theta'-\theta_t$ to server. **Potential bottleneck**



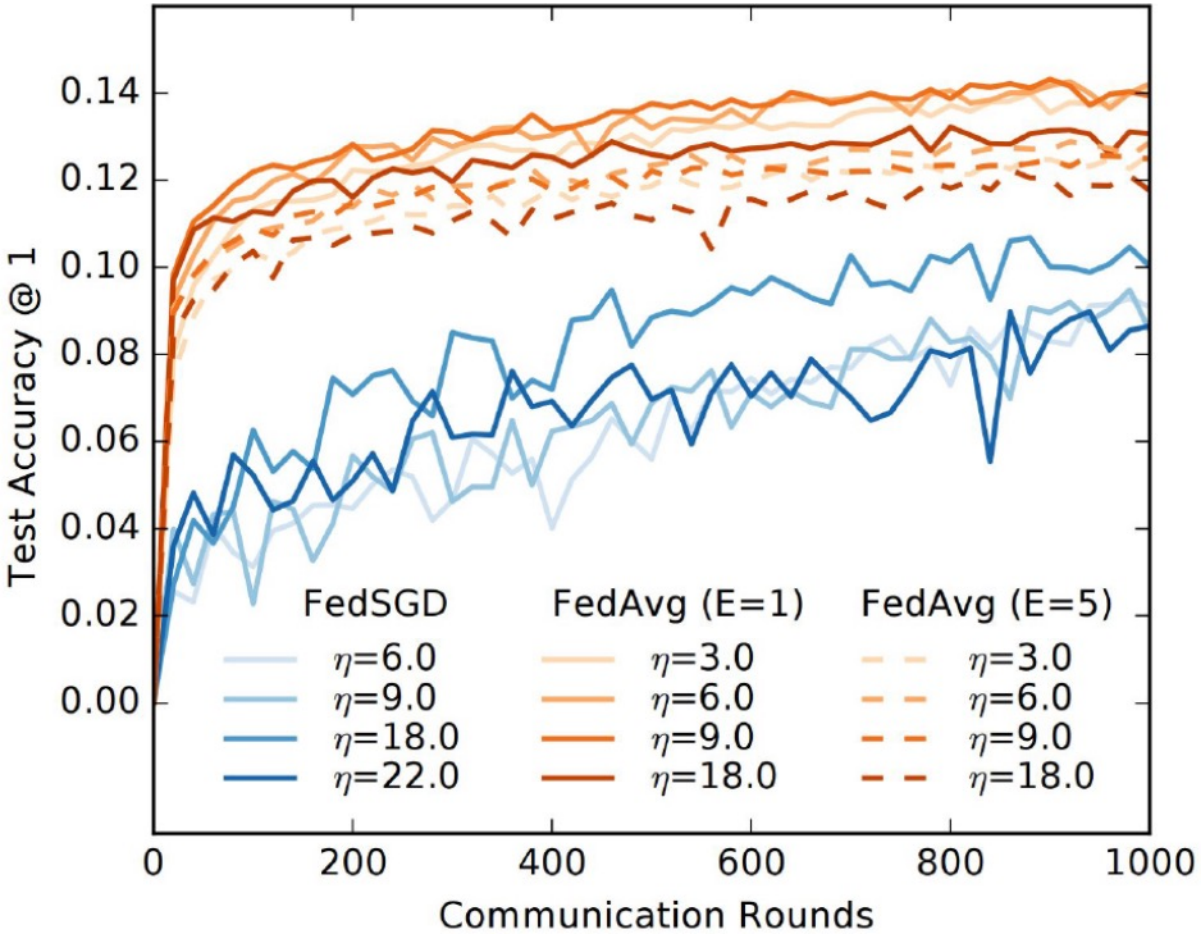
3. $\theta_{t+1} = \theta_t$ + data-weighted average of client updates

H. B. McMahan, *et al.* **Communication-Efficient Learning of Deep Networks from Decentralized Data.** AISTATS 2017

Federating Averaging Performance

Large-scale LSTM for next-word prediction

Dataset: Large Social Network, 10m public posts, grouped by author.



Rounds to reach 10.5% Accuracy

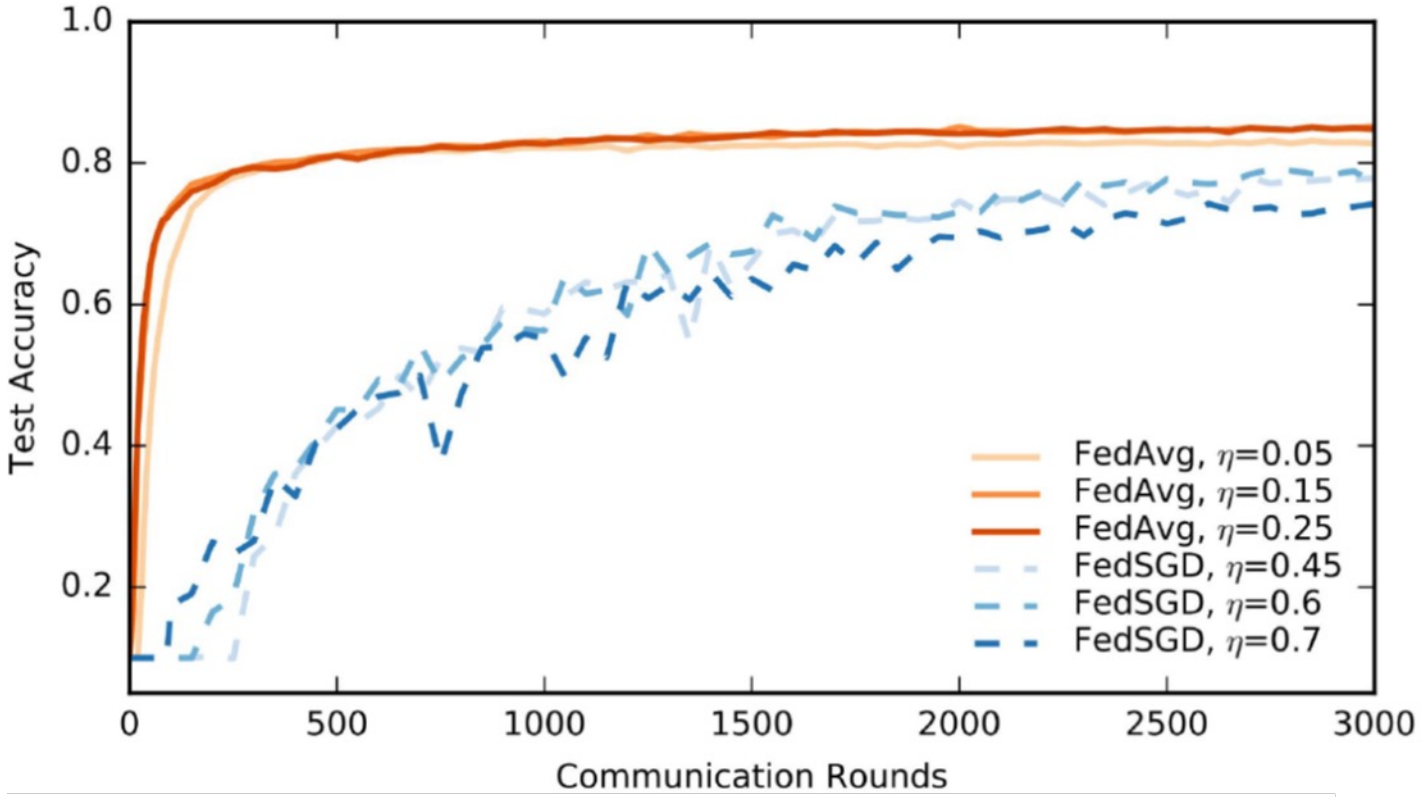
FedSGD	820
FedAvg	35

23x decrease in communication rounds

H. B. McMahan, et al. *Communication-Efficient Learning of Deep Networks from Decentralized Data*. AISTATS 2017

Federating Averaging Performance

CIFAR-10 convolutional model



Updates to reach 82%

SGD	31,000
FedSGD	6,600
FedAvg	630

49x decrease in communication (updates) vs SGD

(IID and balanced data)

ACC.	80%		82%		85%	
SGD	18000	(—)	31000	(—)	99000	(—)
FEDSGD	3750	(4.8×)	6600	(4.7×)	N/A	(—)
FEDAVG	280	(64.3×)	630	(49.2×)	2000	(49.5×)

H. B. McMahan, et al. *Communication-Efficient Learning of Deep Networks from Decentralized Data*. AISTATS 2017