Distributed Optimization for Machine Learning

Lecture 13 - Distributed Training: A Close Look at Data Parallelism

Tianyi Chen

School of Electrical and Computer Engineering
Cornell Tech, Cornell University

October 8, 2025

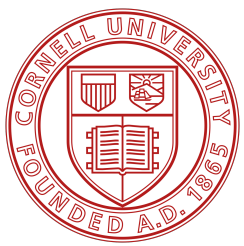
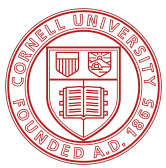
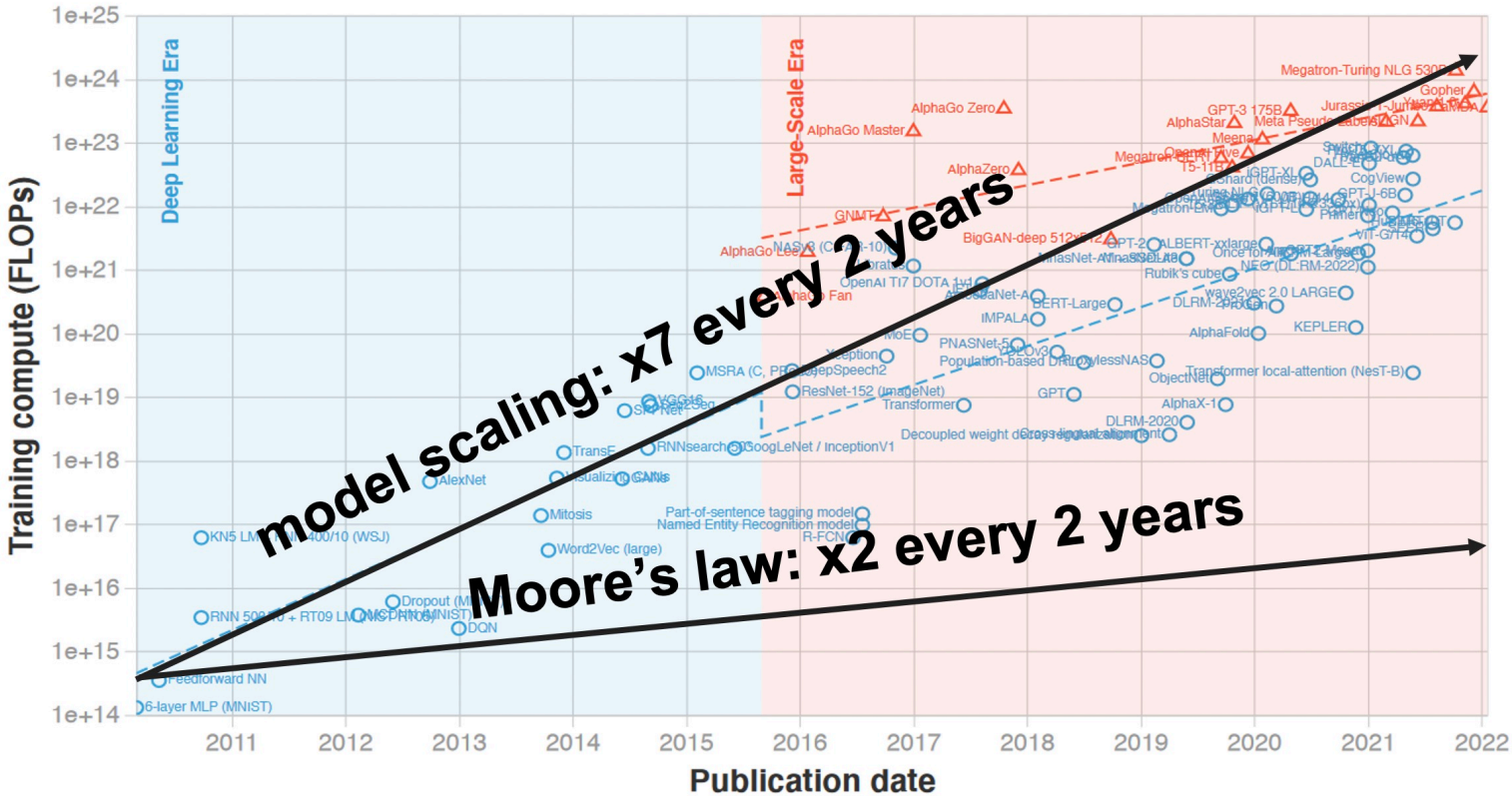


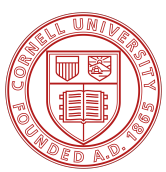
Table of Contents



Training large models requires large compute



Sevilla et al., "Compute trends across three eras of machine learning," **IJCNN 2022.**

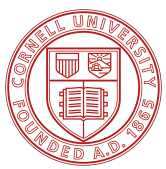


Why parallelize and distributed training?

Modern LLMs push **parameter count**, **context length**, and **training tokens** to extremes.

- **Memory**: weights + optimizer states + activations.
- **Throughput**: tokens/sec limited by compute and communication.

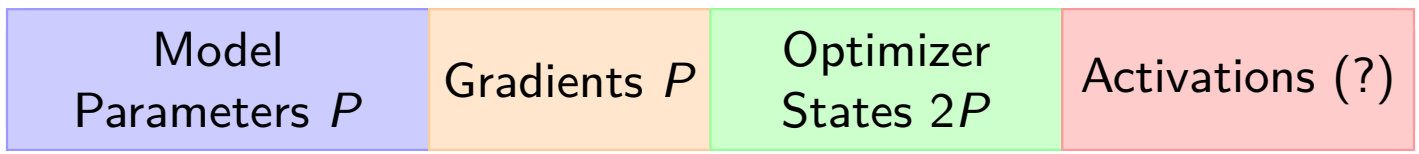
Larger scale + larger memory



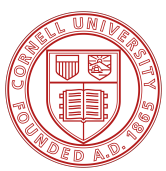
Memory decomposition of training LLMs

Back-of-the-envelope memory (dense FP16 training):

$$\textbf{Memory} = \underbrace{\text{Model}}_{\text{weights}} + \underbrace{\text{Gradients}}_{\text{backprop}} + \underbrace{\text{Optimizer States}}_{\text{e.g., Adam m,v}} + \underbrace{\text{Activations}}_{\text{forward/backward pass}}$$



Typical memory usage breakdown in training large models.



Memory comparison between GPT-3 vs LLaMA

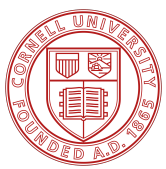
GPT-3 175B

- Weights (FP16): $175 \times 10^9 \cdot 2 \approx$ **350 GB**
- *Training state* ($\sim$16 B/param): $\approx$ **2.8 TB**
- Token storage (e.g., 300B tokens @2 B/token): $\sim$ *600 GB* raw text

LLaMA 65B

- Weights (FP16): $65 \times 10^9 \cdot 2 \approx$ **130 GB**
- *Training state* ($\sim$16 B/param): $\approx$ **1.04 TB**
- If trained on $\sim$1T tokens @2 B/token: $\sim$ *2 TB* raw text

Implication: A *single GPU* cannot hold full training state for 65B-175B dense models. We need **parallelism:** data, model (tensor/pipeline), or hybrids (3D).



Thousands of GPUs are needed to train LLMs

2 example models

GPT-3 (2020)

50,257 vocabulary size
2048 context length
175B parameters
Trained on 300B tokens

Model Name	n_{params}	n_{layers}	d_{model}	n_{heads}	d_{head}	Batch Size	Learning Rate
GPT-3 Small	125M	12	768	12	64	0.5M	6.0×10^{-4}
GPT-3 Medium	350M	24	1024	16	64	0.5M	3.0×10^{-4}
GPT-3 Large	760M	24	1536	16	96	0.5M	2.5×10^{-4}
GPT-3 XL	1.3B	24	2048	24	128	1M	2.0×10^{-4}
GPT-3 2.7B	2.7B	32	2560	32	80	1M	1.6×10^{-4}
GPT-3 6.7B	6.7B	32	4096	32	128	2M	1.2×10^{-4}
GPT-3 13B	13.0B	40	5140	40	128	2M	1.0×10^{-4}
GPT-3 175B or "GPT-3"	175.0B	96	12288	96	128	3.2M	0.6×10^{-4}

Table 2.1: Sizes, architectures, and learning hyper-parameters (batch size in tokens and learning rate) of the models which we trained. All models were trained for a total of 300 billion tokens.

Training: (rough order of magnitude to have in mind)

- O(1,000 - 10,000) V100 GPUs
- O(1) month of training
- O(1-10) $M

LLaMA (2023)

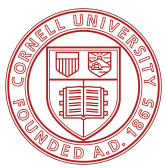
32,000 vocabulary size
2048 context length
65B parameters
Trained on 1-1.4T tokens

params	dimension	n heads	n layers	learning rate	batch size	n tokens
6.7B	4096	32	32	$3.0e^{-4}$	4M	1.0T
13.0B	5120	40	40	$3.0e^{-4}$	4M	1.0T
32.5B	6656	52	60	$1.5e^{-4}$	4M	1.4T
65.2B	8192	64	80	$1.5e^{-4}$	4M	1.4T

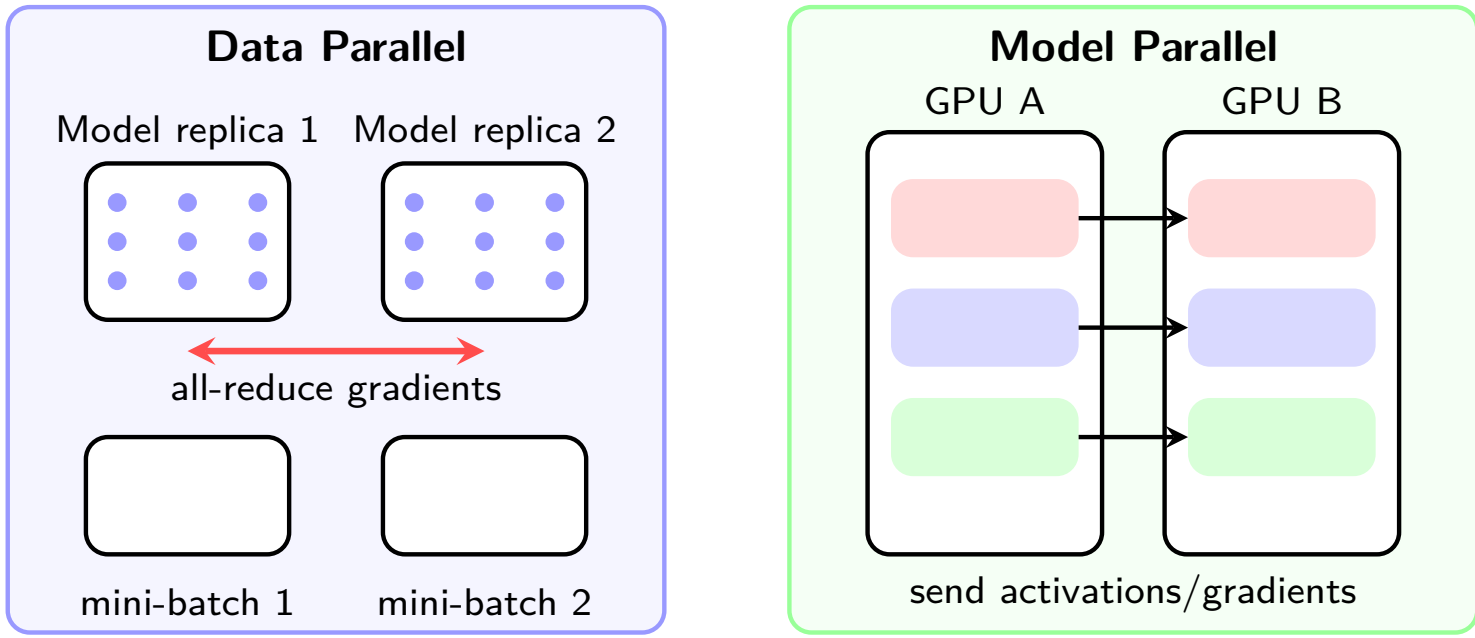
Table 2: **Model sizes, architectures, and optimization hyper-parameters.**

Training for 65B model:

- 2,048 A100 GPUs
- 21 days of training
- $5M

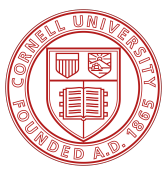


Two paths: Data parallel vs. model parallel



Data parallel: replicate the model; split the data; *consensus* at each step.

Model parallel: split the model across devices; exchange activations.



Why start with data parallel? Memory arithmetic

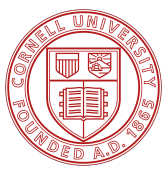
Let P be parameters, b bytes/param for *weights*, s bytes/param for *optimizer & master*, and A activation memory per batch.

$$\text{Per-GPU memory (DP)} \approx \frac{P \cdot b + P \cdot s}{1} + A \quad \text{(replicated across GPUs)}.$$

- If this **fits** in one GPU, add GPUs with DP to increase tokens/sec.
- If it **doesn't fit**, you must shard with model parallel across GPUs:

$$\text{Per-GPU memory (MP)} \approx \frac{P \cdot b + P \cdot s}{n_{\text{shards}}} + \text{per-stage activations}$$

- Combine with activation checkpointing to push the limit.

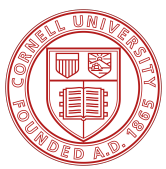


When do we need each?

- **Data Parallel**: model + optimizer *fit* on a single GPU, but you want more throughput
  - Near-linear speedup until communication (all-reduce) dominates.
  - Combine with compression, mixed precision, and large batch training.

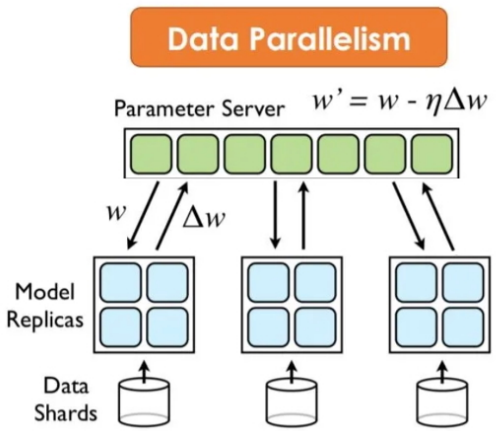
- **Model Parallel**: model *does not fit* (weights/activations/optimizer)
  - *Tensor parallel*: shard matrices within layers.
  - *Pipeline parallel*: shard layers across stages

Design target for different parallelisms: minimize *communication/compute* ratio while keeping memory under budget.

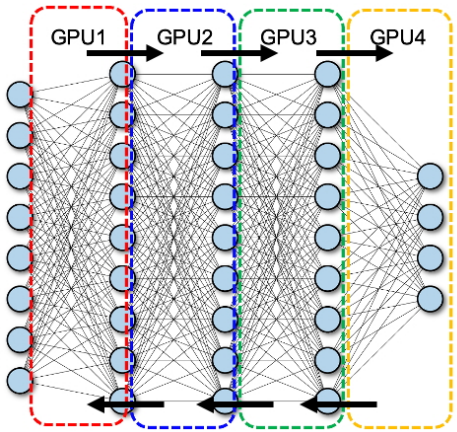


3 ways of parallelism used to train large models

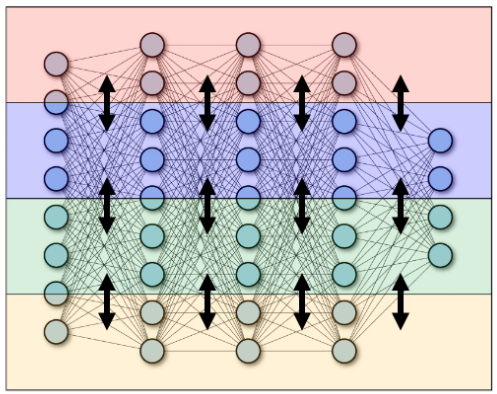
In today's industry, there are three dimensional-parallelism indicates 3 orthogonal parallel techniques used to train large models such as LLMs.



Data parallelism



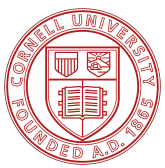
Pipeline parallelism



Tensor parallelism

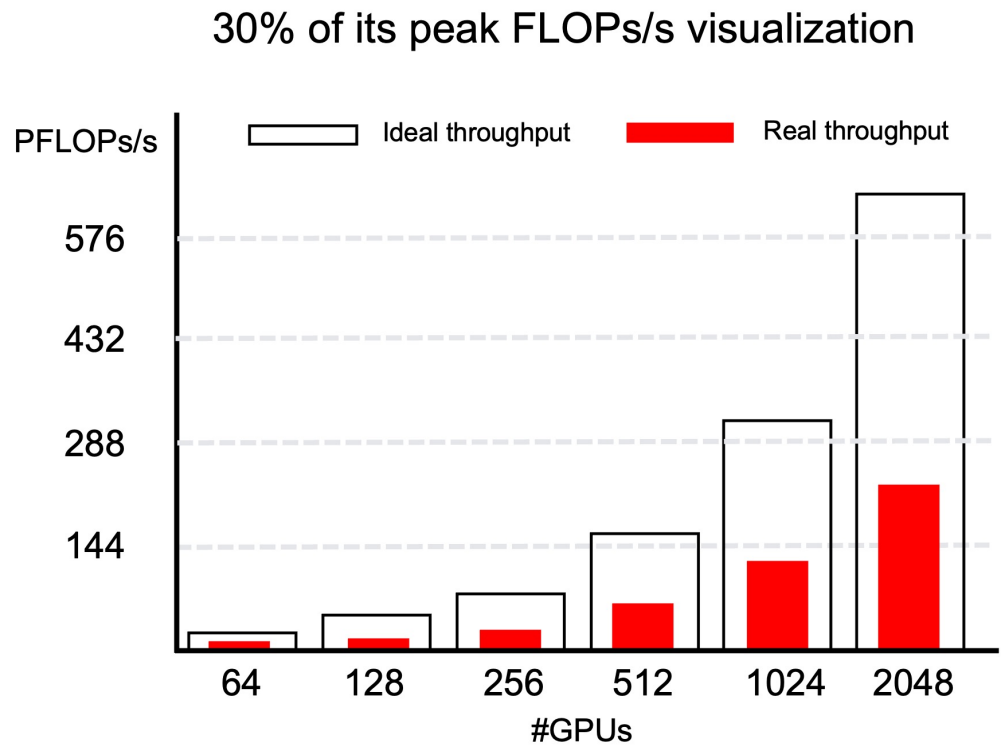
model parallelism

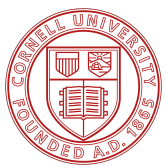
This lecture mainly focuses on **data parallelism**.



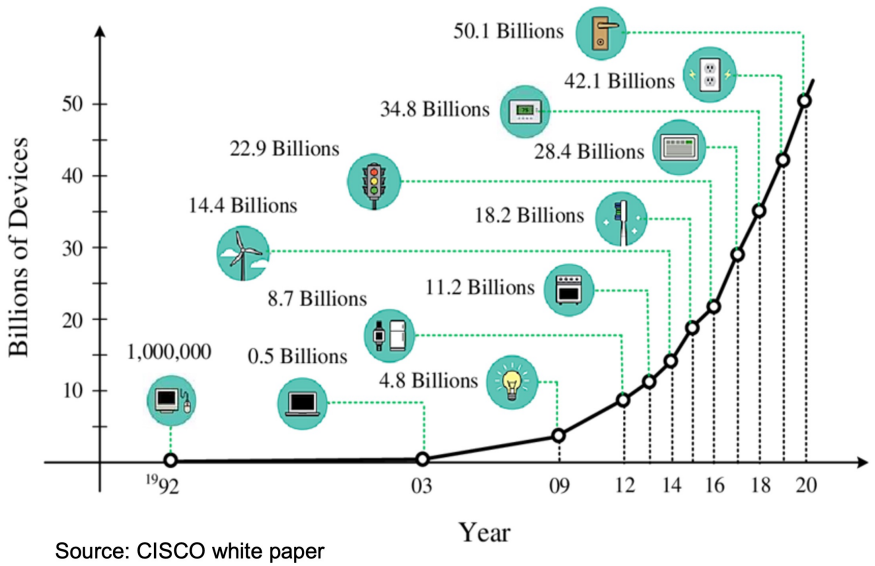
Distributed training is extremely challenging

- The communication overhead and GPU idle time hamper scalability
- Each GPU can only achieve 30%-55% of its peak compute power
- The system achieves 30% scalability - inefficient





Distributed training beyond data centers



Source: CISCO white paper

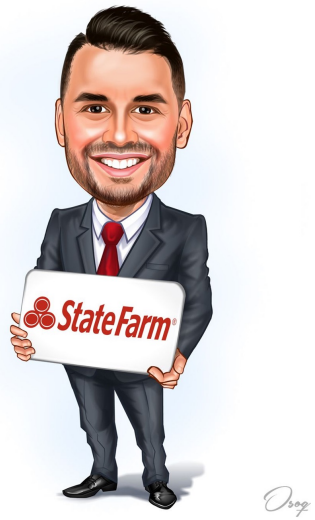
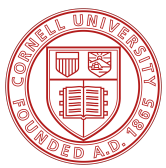


Table of Contents

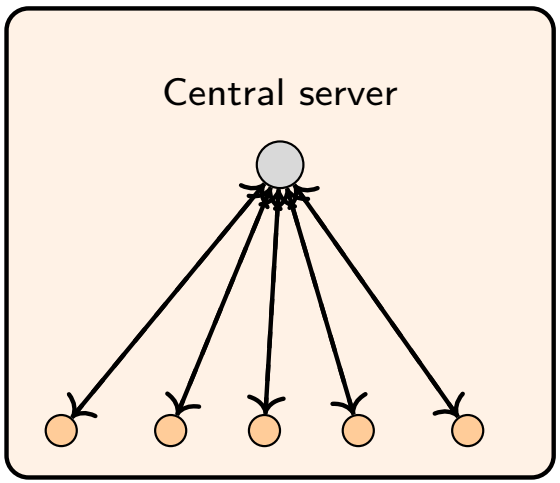


Optimization formulation of data parallelism

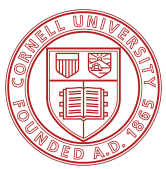
A network of n nodes (here are GPUs) collaborate to solve:

$$\min_{\mathbf{x}\in\mathbb{R}^d} f(\mathbf{x}) = \frac{1}{n}\sum_{i=1}^{n} f_i(\mathbf{x}), \quad \text{where } \boxed{f_i(\mathbf{x}) = \mathbb{E}_{\xi_i\sim D_i}[F(\mathbf{x};\xi_i)]}$$

- Each component $f_i : \mathbb{R}^d \to \mathbb{R}$ is local and private to node i.
- Random variable ξ_i denotes local data following distribution D_i.
- D_i may be different $\Rightarrow$ **data heterogeneity**.



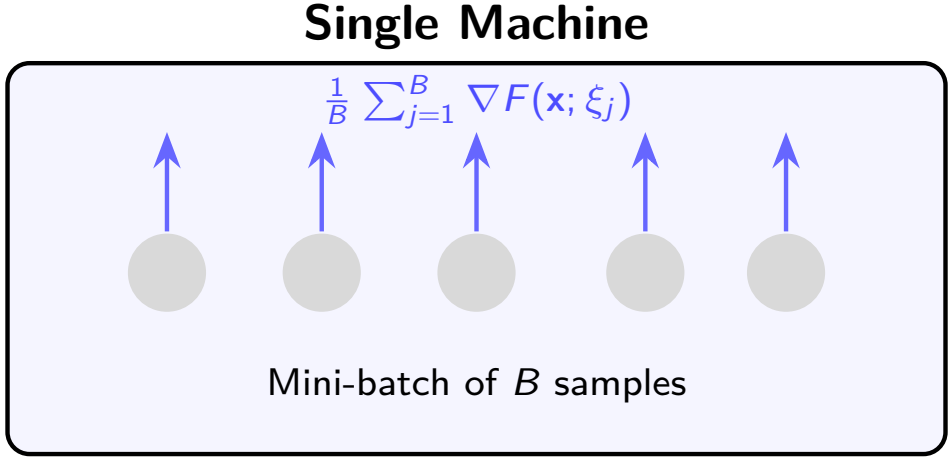
Local data on nodes



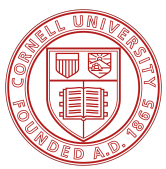
Mini-batch SGD: Averaging within one machine

Key idea: Instead of updating per sample, average gradients over a mini-batch of samples.

$$\mathbf{x}^{k+1} = \mathbf{x}^k - \frac{\eta}{B} \sum_{j=1}^{B} \nabla F(\mathbf{x}^k; \xi_j^k)$$



Effect: Reduces variance and stabilizes convergence.



Parallel SGD: compute locally, communicate globally

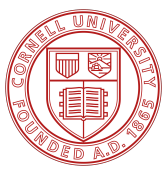
$$\min_{\mathbf{x}\in\mathbb{R}^d} \; f(\mathbf{x}) \;=\; \frac{1}{n}\sum_{i=1}^{n} f_i(\mathbf{x}), \qquad \text{where } f_i(\mathbf{x}) = \mathbb{E}_{\xi_i\sim D_i}[F(\mathbf{x};\xi_i)]\,.$$

PSGD

$$g_i^k \;=\; \nabla F\left(\mathbf{x}^k;\, \xi_i^k\right) \qquad \text{(Local compt.)}$$

$$\mathbf{x}^{k+1} \;=\; \mathbf{x}^k \;-\; \frac{\eta}{n}\sum_{i=1}^{n} g_i^k \qquad \text{(Global comm.)}$$

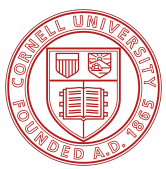
- Each node i samples mini-batch ξ_i^k and computes $\nabla F\left(\mathbf{x}^k;\xi_i^k\right)$.
- All nodes synchronize (i.e., *globally average*) to update $\mathbf{x}$.



Mini-batch vs. Parallel SGD implementation

Aspect	**Mini-Batch SGD**	**Parallel SGD**
Computation	One machine averages over B samples	n workers compute on separate batches
Synchronization	Implicit (within memory)	Explicit (via Allreduce or parameter server)
Effective batch size	B	$n \times B_{\text{local}}$
Noise reduction	By sampling multiple data	By averaging across workers
Communication	None	Required per iteration
Data heterogeneity	N/A (one dataset)	Possible ($D_i \neq D_j$)

Key: Parallel SGD = Mini-batch SGD distributed across multiple nodes.



From consensus to Ring-AllReduce

Recall: In consensus averaging, all workers iteratively mix local states

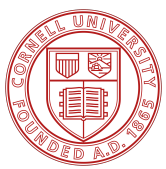
$$\mathbf{x}_i^{k+1} = \sum_j W_{ij}\mathbf{x}_j^k, \quad \text{where } \mathbf{W} \text{ is doubly stochastic.}$$

Goal: Reach *consensus average* $\mathbf{x}_i^k \to \bar{\mathbf{x}} = \frac{1}{n}\sum_{i=1}^n \mathbf{x}_i$.

Connection to distributed training

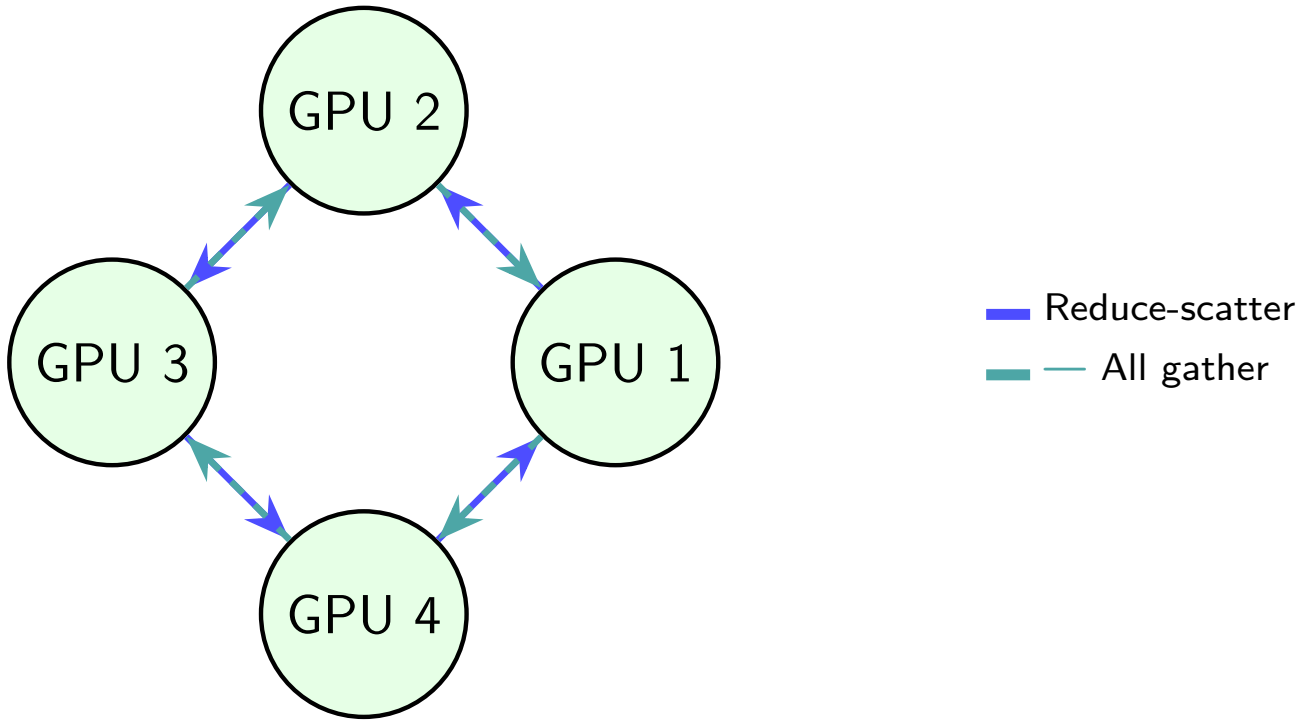
In practice, when synchronizing gradients or parameters across GPUs, we perform the same averaging - but via the **Ring-AllReduce** protocol.

Consensus theory $\Rightarrow$ *Efficient synchronization in deep learning clusters.*

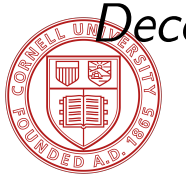


Globally average in practice: Ring-AllReduce

Idea: All GPUs share gradients in a ring to compute the global average. No central server; synchronization is peer-to-peer.



Key takeaway: Each GPU eventually holds $\bar{g} = \frac{1}{n}\sum_i g_i$ and updates $\mathbf{x}$.
Decentralized synchronization (common in datacenter cluster training).



Ring-AllReduce Phase 1: Reduce-scatter

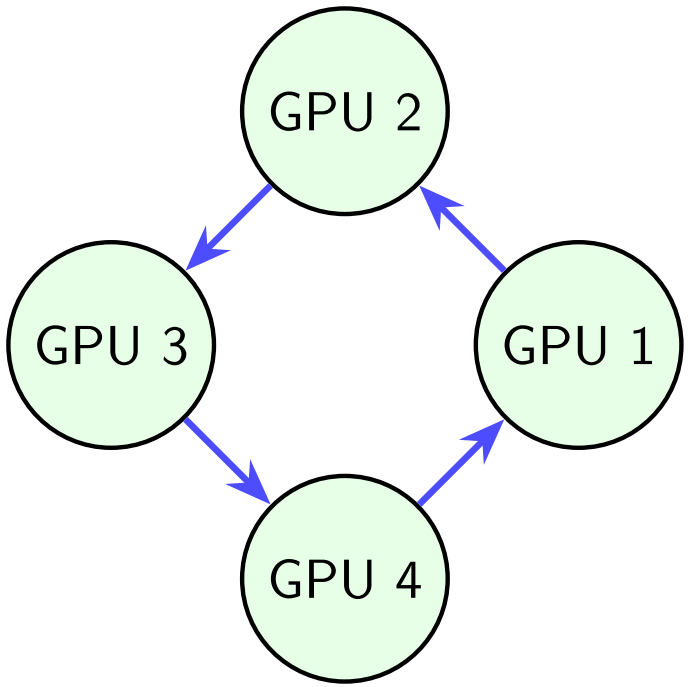
Goal: Compute partial sums of gradient chunks in a distributed way.

- Each GPU splits its gradient into n parts:

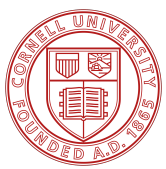
$$g_i = [g_i^{(1)}, g_i^{(2)}, \ldots, g_i^{(n)}].$$

- GPUs pass and sum chunks around ring.
- After $n-1$ steps, each GPU holds one summed chunk $\sum_i g_i^{(k)}$.

Analogy: Everyone edits one page of an essay.



Result: Each GPU stores $\frac{1}{n}$ of the total gradient sum.



Ring-AllReduce Phase 2: All-Gather

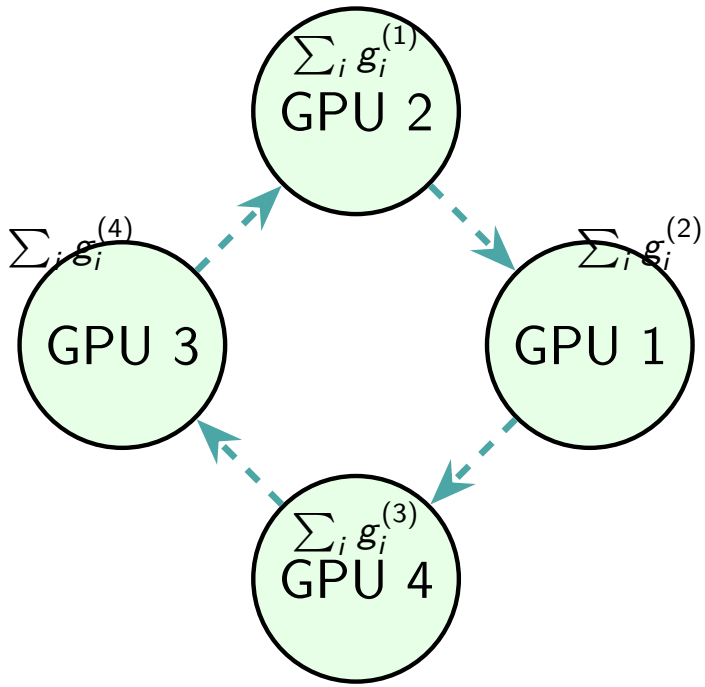
Goal: Share the summed chunks so that all GPUs obtain the full averaged gradient.

- Each GPU holds one chunk $\sum_i g_i^{(k)}$.
- GPUs circulate these chunks around ring.
- After $n-1$ passes, all GPUs have all chunks:

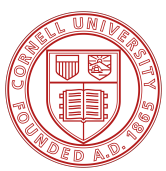
$$[\sum_i g_i^{(1)}, \sum_i g_i^{(2)}, \ldots, \sum_i g_i^{(n)}].$$

- Each divides by n to get $\bar{g} = \frac{1}{n}\sum_i g_i$.

Analogy: Everyone exchanges their completed page to rebuild the full essay.

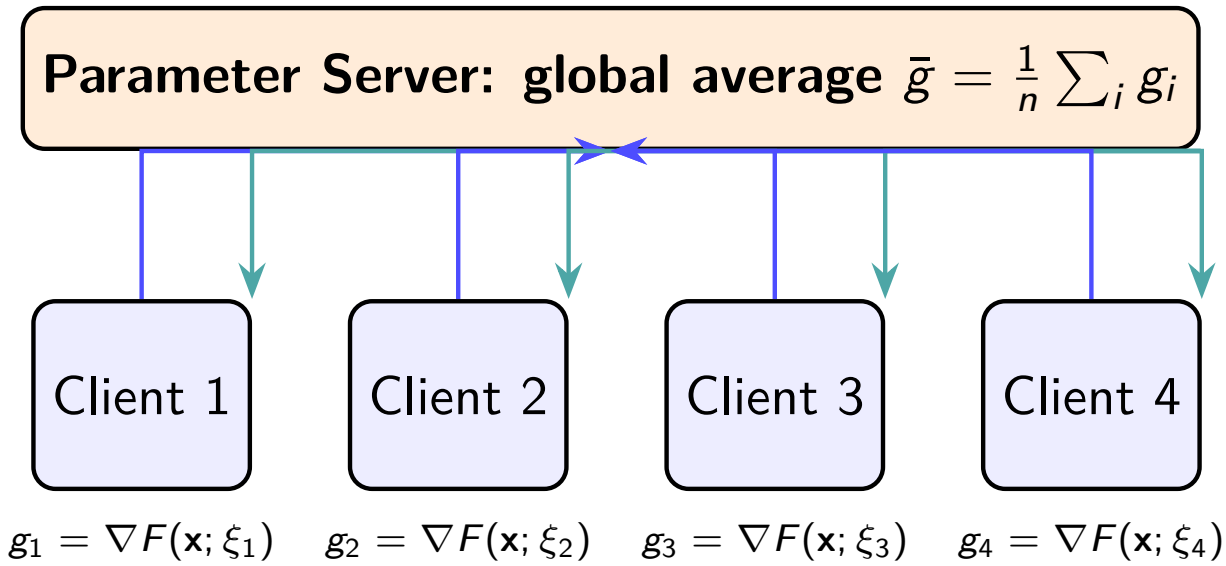


Result: Each GPU has the full averaged gradient $\bar{g}$ and updates $\mathbf{x}$.



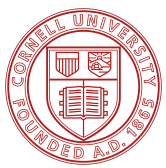
Globally average in practice: Parameter server

Idea: Each client computes local gradients, sends them to a central server, which averages and broadcasts the updated global model.



All clients synchronize per iteration

Key takeaway: A single global node (server) performs the averaging and distributes $\mathbf{x}^{k+1}$. *Common in Federated Learning.*



Parallel SGD convergence rate

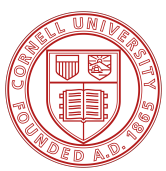
Assumptions:

- F: μ-strongly convex, L-smooth
- $g(\mathbf{x};\xi)$ is an unbiased estimate of $\nabla F(\mathbf{x})$, with bounded variance.

Theorem 1 (Parallel SGD under smooth and convex loss)

Let f be L-smooth and $\eta_k \equiv \eta = 1/\sqrt{K}$. Then the following holds

$$\frac{1}{K}\sum_{k=1}^{K}\mathbb{E}[||\nabla f(\mathbf{x}^k)||_2^2] \leq \mathcal{O}\left(\frac{1}{K} + \frac{\sigma}{\sqrt{nK}}\right)$$



Linear speedup in parallel SGD

Theorem 1 (Parallel SGD under smooth and convex loss)

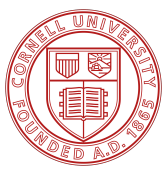
Let f be L-smooth and $\eta_k \equiv \eta = 1/\sqrt{K}$. Then the following holds

$$\frac{1}{K}\sum_{k=1}^{K}\mathbb{E}[||\nabla f(\mathbf{x}^k)||_2^2] \leq \mathcal{O}\left(\frac{1}{K} + \frac{\sigma}{\sqrt{nK}}\right)$$

This implies that to achieve an ϵ-accurate solution, Parallel SGD needs

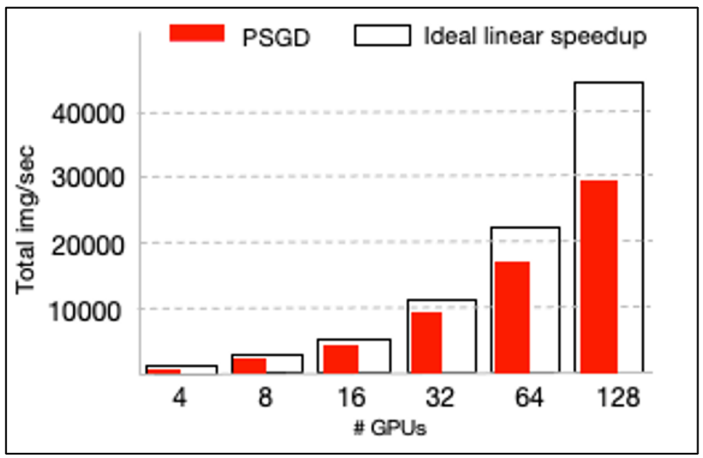
$$\frac{\sigma}{\sqrt{nK}} \leq \epsilon \quad \Longrightarrow \quad K \geq \frac{\sigma^2}{n\epsilon^2} \quad \text{iterations,}$$

which decreases linearly with number of nodes n; called *linear speedup*!

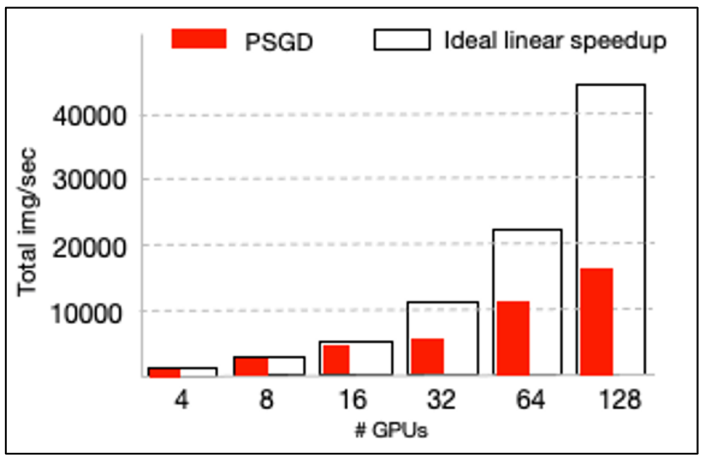


Parallel SGD may not achieve actual linear speedup

- Cannot achieve ideal linear speedup due to communication overhead
- Larger communication-to-computation ratio $\rightarrow$ worse performance

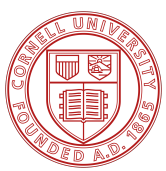


Small comm.-to-compt. ratio



Large comm.-to-compt. ratio

- Ring-all reduce is used in each experiment!
- How can we reduce the communication overhead in Parallel SGD?



Methodologies to save communication

- Global average incurs $O(n)$ comm. overhead; proportional to network size n **[Decentralized communication]**
- Each node interacts with the server at every iteration; proportional to iteration numbers **[Lazy communication]**
- Each node sends a full model (or gradient) to the server; proportional to dimension d **[Compressed communication]**
- and more (asynchronous communication; robust communication against Byzantine nodes, etc.)

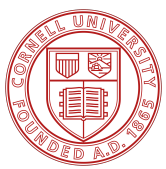
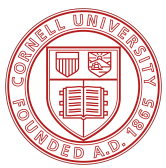


Table of Contents

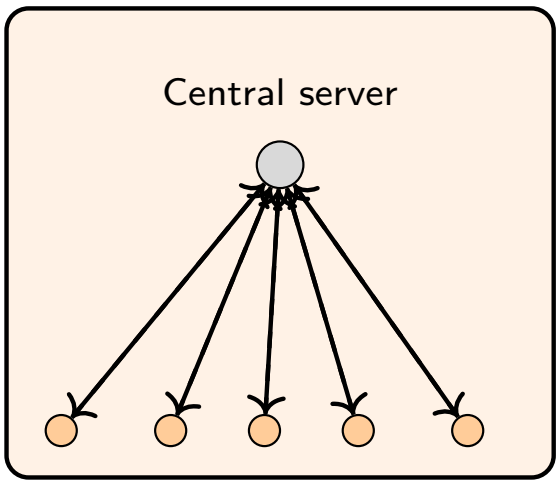


Optimization formulation of data parallelism

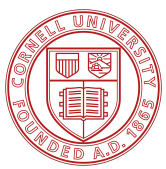
A network of n nodes (such as mobile devices) collaborate to solve:

$$\min_{\mathbf{x}\in\mathbb{R}^d} f(\mathbf{x}) = \frac{1}{n}\sum_{i=1}^{n} f_i(\mathbf{x}), \quad \text{where } \boxed{f_i(\mathbf{x}) = \mathbb{E}_{\xi_i\sim D_i}[F(\mathbf{x};\xi_i)]}$$

- Each component $f_i : \mathbb{R}^d \to \mathbb{R}$ is local and private to node i.
- Random variable ξ_i denotes local data following distribution D_i.
- D_i may be different $\Rightarrow$ **data heterogeneity**.

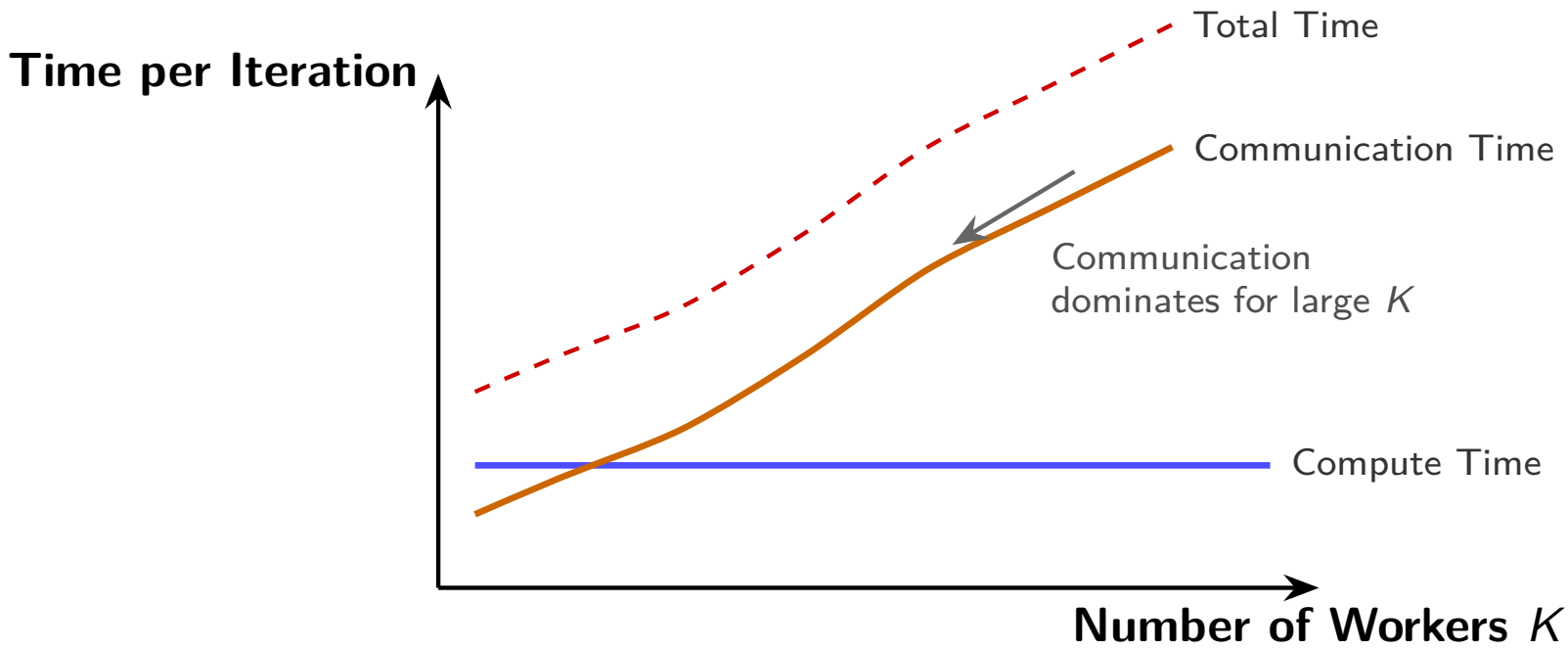


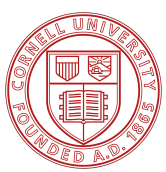
Local data on nodes



Communication bottleneck in Parallel SGD

- In Parallel SGD, workers **synchronize after every step**.
- Comm. dominates runtime when n is large or network is slow.





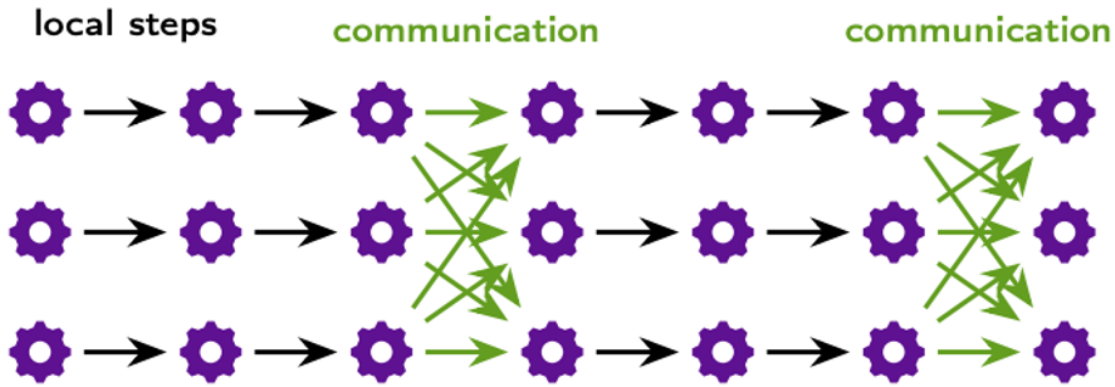
Idea: Local updates before synchronization

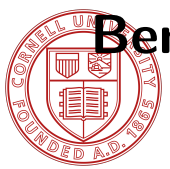
Key idea: Each node i performs several SGD steps before averaging.

$$\mathbf{x}_i^{(s+1)} = \mathbf{x}_i^{(s)} - \eta\, g_i^{(s)}, \quad s = 0, \ldots, \tau - 1$$

where $\mathbf{x}_i^{(0)} = \mathbf{x}^k$. After every τ steps:

$$\mathbf{x}^{k+1} = \frac{1}{n} \sum_{i=1}^{n} \mathbf{x}_i^{(\tau)}$$





Benefit: Reduces communication by a factor of τ.

Mini-batch SGD vs. Local SGD

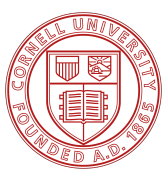
Mini-batch or Parallel SGD:

$$\mathbf{x}^{t+1} = \mathbf{x}^t - \eta_t \frac{1}{n\tau} \sum_{i=1}^{n} \sum_{s=1}^{\tau} \nabla F(\mathbf{x}^t; \xi_i^{t,s})$$

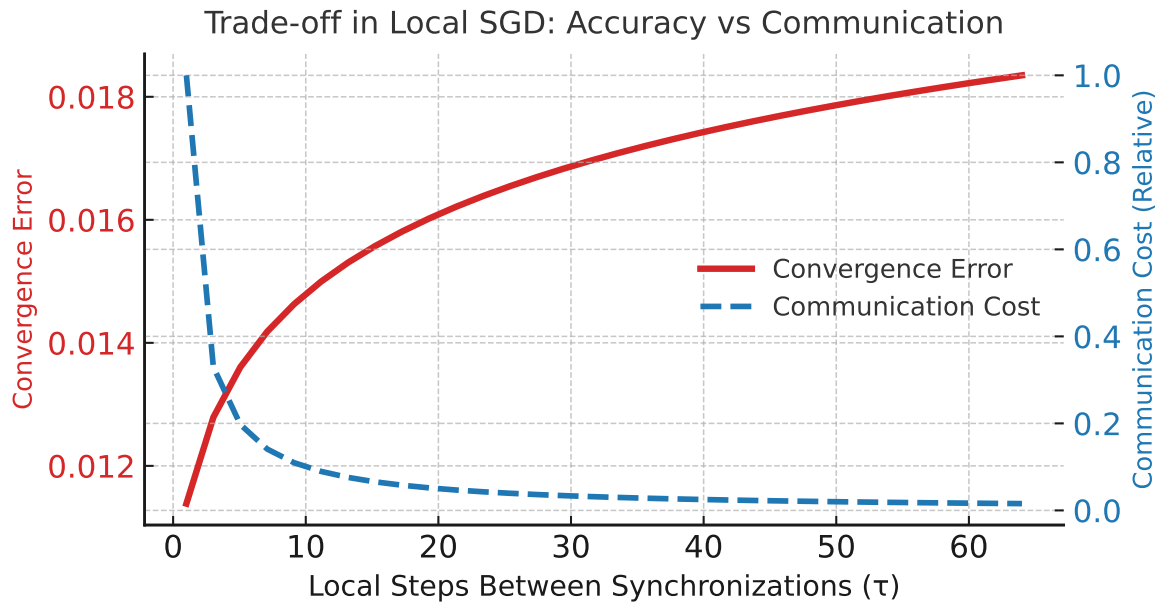
Local SGD:

$$\mathbf{x}_i^{t+1} = \begin{cases} \mathbf{x}_i^t - \eta_t \nabla F(\mathbf{x}_i^t; \xi_i^t), & t \bmod \tau \neq 0 \\ \frac{1}{n} \sum_{i=1}^{n} \left(\mathbf{x}_i^t - \eta_t \nabla F(\mathbf{x}_i^t; \xi_i^t) \right), & t \bmod \tau = 0 \end{cases}$$

Method	Mini-batch SGD	Local SGD
# Comm. rounds	K	K
Batch size	$n\tau$	n
# Model updates	K	τK
# Gradient calcs	$n\tau K$	$n\tau K$

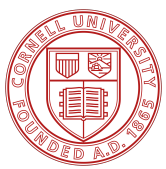


Communication vs. computation trade-off



$$\text{Runtime per iteration} = \text{Compute} + \frac{1}{\tau}\text{Comm.}$$

Insight: Increasing τ improves efficiency but risks model drift.



Why Local SGD works under homogeneous data?

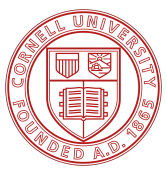
If $f(\mathbf{x})$ is convex and all workers start synchronized ($\mathbf{x}_i^t = \bar{\mathbf{x}}^t$):

$$f(\bar{\mathbf{x}}^{t+\tau}) \leq f(\bar{\mathbf{x}}^t) - (\text{averaged descent term}).$$

Thus, synchronization preserves global descent.

$$f(\bar{\mathbf{x}}^{t+\tau}) \leq \frac{1}{n}\sum_{i=1}^{n} f_i(\mathbf{x}_i^{t+\tau}) \leq f(\bar{\mathbf{x}}^t) - (\text{progress}).$$

Not true in general if f_i differ across workers (non-i.i.d.).



Quadratic objectives: analytical insight

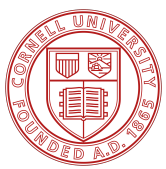
For local quadratic objectives $f_i(\mathbf{x}) = \frac{1}{2}\mathbf{x}^\top A_i \mathbf{x} - \mathbf{b}_i^\top \mathbf{x}$:

$$\mathbf{x}_i^{k+1} = \mathbf{x}_i^k - \eta_k \nabla f_i(\mathbf{x}_i^k).$$

Averaging yields:

$$\mathbb{E}[\bar{\mathbf{x}}^{k+1} | \mathcal{F}^k] = \bar{\mathbf{x}}^k - \eta_k \frac{1}{n} \sum_{i=1}^n \nabla f_i(\mathbf{x}_i^k) \approx \bar{\mathbf{x}}^k - \eta_k \nabla f(\bar{\mathbf{x}}^k).$$

Hence, Local SGD mimics global descent dynamics.



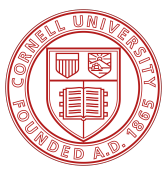
Local SGD improves efficiency in quadratic setting

Theorem 2 (Local SGD under smooth and convex loss)

The error bound for Local SGD with τ local updates equals the bound for Mini-batch SGD with batch size n and $K\tau$ rounds:

$$\epsilon_{\text{L-SGD}} := \frac{1}{K}\sum_{k=1}^{K}\mathbb{E}[||\nabla f(\mathbf{x}^k)||_2^2] = \mathbf{\Theta}\left(\frac{1}{K\tau} + \frac{\sigma}{\sqrt{nK\tau}}\right)$$

- More local updates τ always help convergence.
- Mini-batch SGD: $\epsilon_{\text{MB-SGD}} = \mathbf{\Theta}\left(\frac{1}{K} + \frac{\sigma}{\sqrt{nK\tau}}\right)$
- Therefore, Local SGD *can be better* given the same computation budget.



Performance for general convex objectives*

Upper and lower bounds for Local SGD:

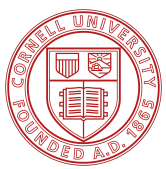
$$\text{Upper:} \qquad \epsilon_{\text{L-SGD}} = \mathcal{O}\left(\frac{\sigma^{2/3}}{K^{2/3}\tau^{1/3}} + \frac{\sigma}{\sqrt{nK\tau}}\right)$$

$$\text{Lower:} \qquad \Omega\left(\frac{\sigma^{2/3}}{K^{2/3}\tau^{2/3}} + \frac{\sigma}{\sqrt{nK\tau}}\right)$$

Mini-batch SGD: $\quad \boldsymbol{\Theta}\left(\frac{1}{K} + \frac{\sigma}{\sqrt{nK\tau}}\right)$

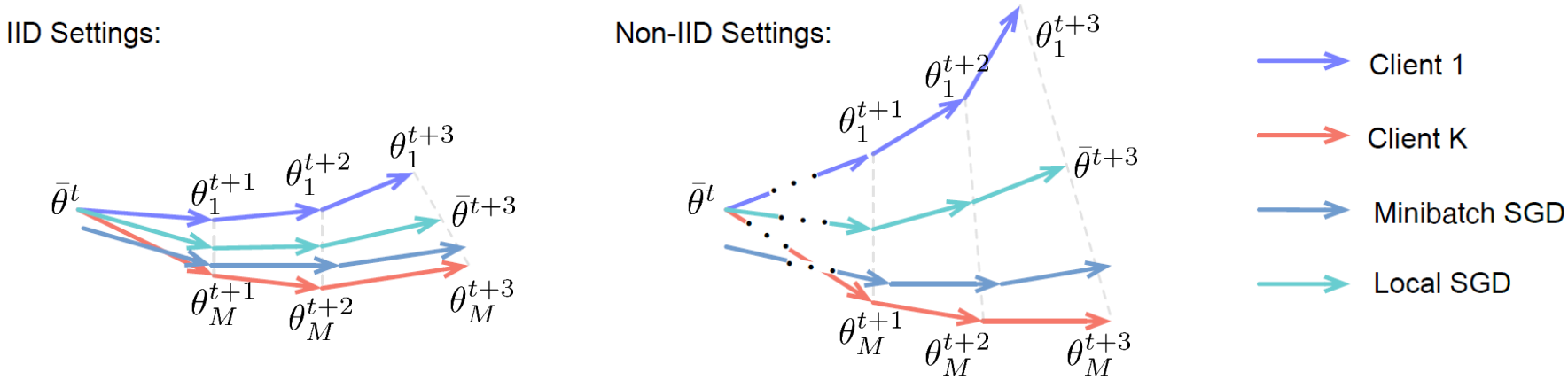
Local SGD better when $K \lesssim \tau$, worse when $K \gtrsim \tau$.

Woodworth et al. "Is Local SGD Better than Mini-batch SGD?", *ICML 2020*

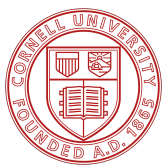


Why local SGD may fail under heterogeneous data?

In heterogeneous (non-i.i.d.) data settings, local gradients are misaligned

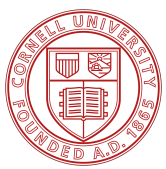


- Local updates diverge due to heterogeneous data (Γ^2).
- Need additional assumptions to control gradient dissimilarity.
- Larger $\tau \Rightarrow$ greater deviation from the global model.

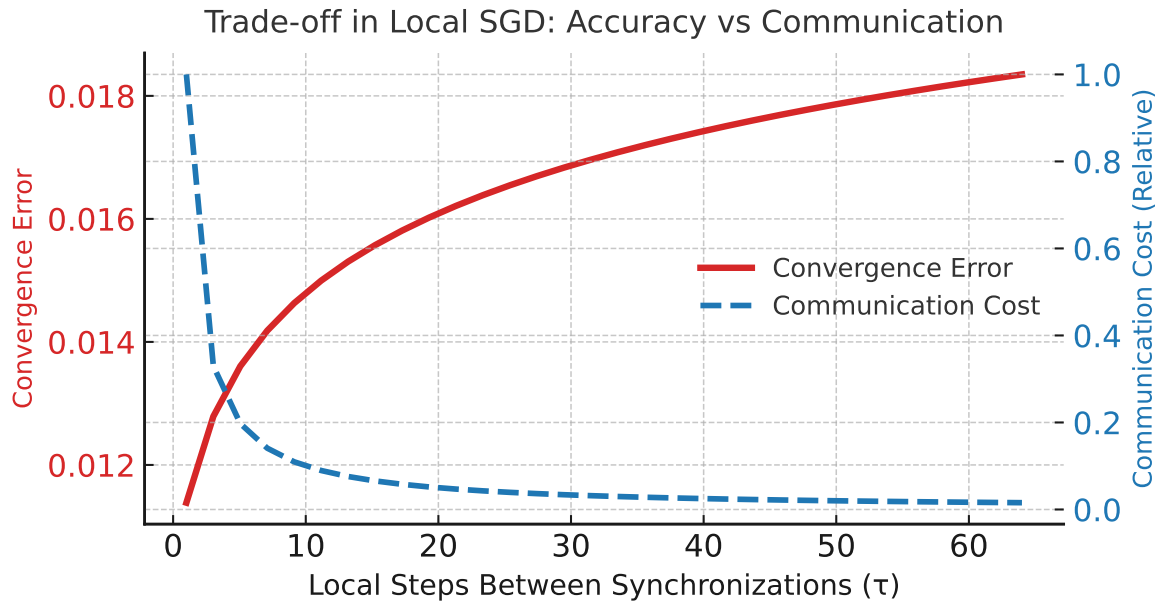


Summary: Parallel SGD vs local SGD

Aspect	**Parallel SGD**	**Local SGD**
Communication	Every iteration	Every τ iterations
Local computation	1 gradient step	τ local steps
Speed	Communication-limited	Compute-efficient
Convergence rate	Stable	Slower ($(\eta^2\tau\Gamma^2)$ bias)
Best for	Data centers, i.i.d. data	Federated settings

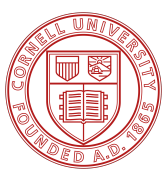


Takeaway: Communication - accuracy trade-off



- $\tau = 1$: fully synchronized (Parallel SGD)
- $\tau > 1$: fewer syncs $\Rightarrow$ faster but drift grows
- Choose τ based on network bandwidth and data heterogeneity

Rule of thumb: $\tau^* \propto \sqrt{\frac{c_{\text{comm}}}{c_{\text{comp}}}}$



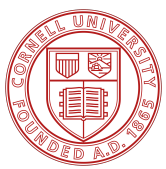
When to use Local SGD?

Recommended if:

- Communication cost $c_{\text{comm}} \gg c_{\text{comp}}$
- Data across workers are relatively homogeneous
- Occasional synchronization suffices for convergence

Avoid if:

- Highly non-i.i.d. data (strong gradient heterogeneity)
- Models are unstable to small parameter changes



Recap and fine-tuning

- What we have talked about today?

  $\Rightarrow$ **Data parallelism** enables scaling model training across multiple workers by distributing data and synchronizing models through averaging.

  $\Rightarrow$ **Parallel SGD** achieves global model updates via synchronization.

  $\Rightarrow$ **Local SGD** relaxes synchronization - maintaining learning efficiency in homogeneous data; drift apart in heterogeneous settings.

Welcome anonymous survey!