EURECOM
Sophia Antipolis

MALCOM

Machine Learning for Communication Systems

Lecture 9
Distributed Machine Learning

Jaiprakash Nagar
(Slides: Marios Kountouris)

Spring 2024

So far…

- … we have talked about algorithms and techniques
- … we have seen ways to optimize their parameters

- *How should we implement our algorithms to best utilize our resources?*
- *Which is the right architecture for our ML algorithms?*
- *How about hardware? How does it affect the performance and the design options?*

Main Issues in Large-Scale ML (massive datasets):

- Storage
- Computation
- Communication (in distributed settings)

Goal: Speed up Machine Learning
Key Approach: Train at Scale

ImageNet Training

Submission Date	Model	Time to 93% Accuracy
Mar 2020	ResNet50-v1.5 *Apsara AI Acceleration(AIACC) team in Alibaba Cloud* source	0:02:38
May 2019	ResNet-50 *ModelArts Service of Huawei Cloud* source	0:02:43
Dec 2018	ResNet-50 *ModelArts Service of Huawei Cloud* source	0:09:22
Sep 2018	ResNet-50 *fast.ai/DIUx (Yaroslav Bulatov, Andrew Shaw, Jeremy Howard)* source	0:18:06
Mar 2018	ResNet50 *Google Cloud TPU* source	12:26:39
Jan 2018	ResNet50 *DIUX* source	14:37:59
Dec 2017	ResNet152 *ppwwyyxx* source	1 day, 20:28:27
Oct 2017	ResNet152 *Stanford DAWN* source	10 days, 3:59:59
Oct 2017	ResNet152 *Stanford DAWN* source	13 days, 10:41:37

https://dawn.cs.stanford.edu/benchmark/index.html

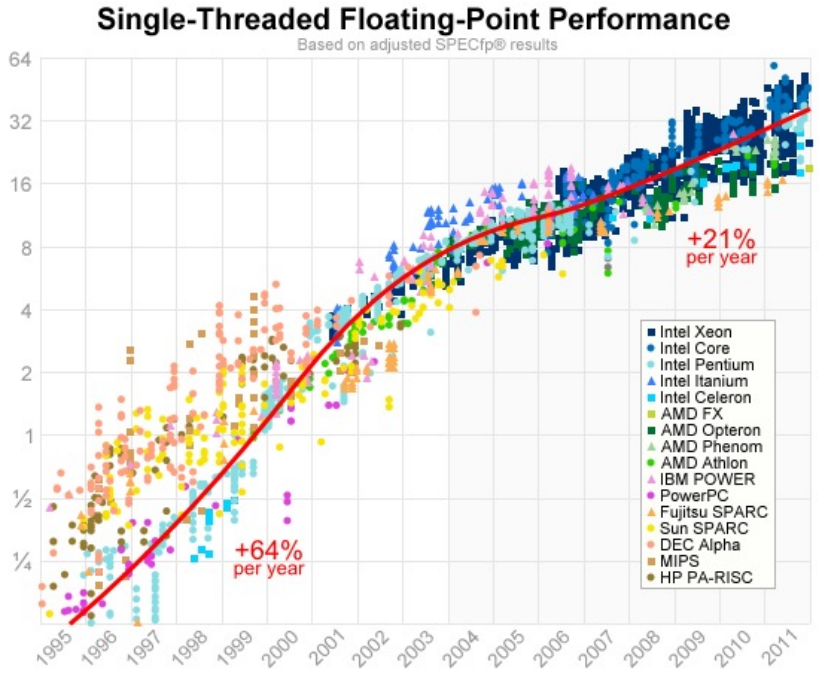
Computing Performance - the Good Old Days

G. Moore's "Law"

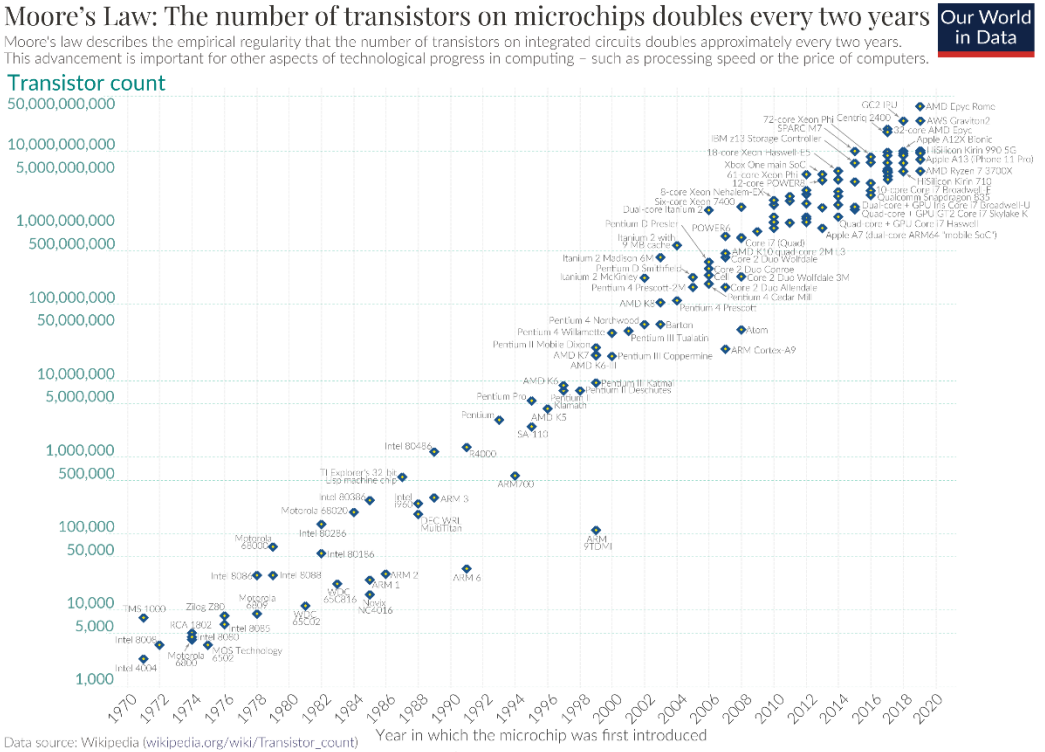
- Usually cast as "X doubles every 18-24 months".
- Q: What is X?
  - Computer performance
  - CPU Clock speed
  - Number of transistors per chip
  - One of the above, at constant cost?

~ 50 years of **exponential** computing performance growth

Remarkable but difficult to maintain!!



https://preshing.com/20120208/a-look-back-at-single-threaded-cpu-performance/



EURECOM
Sophia Antipolis

Computing Performance - the Good Old Days

Dennard Scaling Law

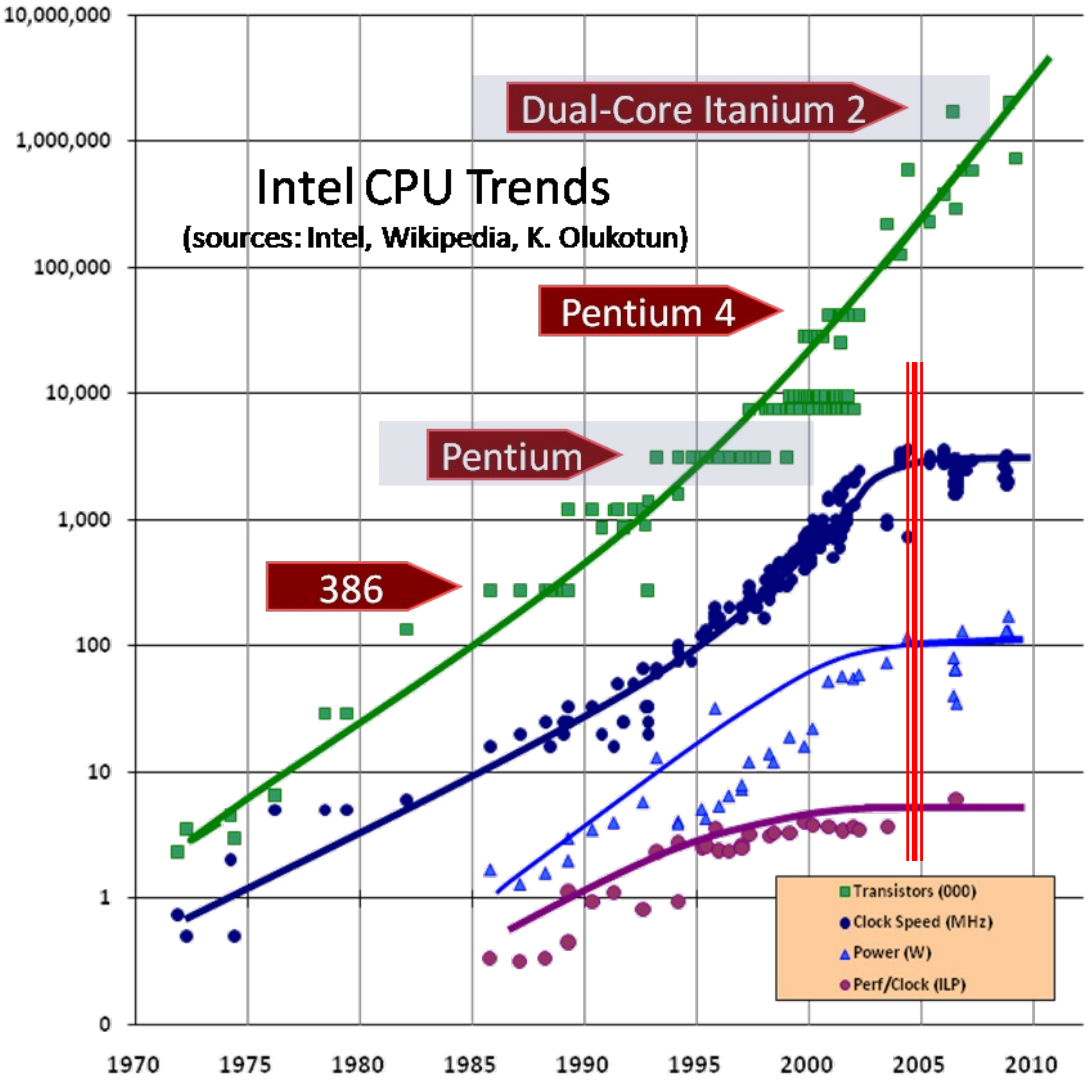
- *"When we double the number of transistors on the same surface, they use double the amount of power."* Right?
- Robert H. Dennard (1974): NO!!
- Dennard scaling: as transistors get smaller, power density (*power per mm*^{2}) stays constant.
- Voltage and current should be proportional to the linear dimensions of a transistor
- Thus, as transistors shrank, so did necessary voltage and current; power is proportional to the area of the transistor.

$$P = N_{\text{sw}} \cdot C \cdot F \cdot V^2$$

F: frequency, V: voltage, N_{sw}:number of bit switching

- Intuitively:
  - Smaller transistors ⇒ shorter propagation delay ⇒ faster frequency
  - Smaller transistors ⇒ smaller capacitance ⇒ lower voltage
- Capacitance C is related to area: so, as the size of the transistors shrunk, and the voltage was reduced, circuits could operate at higher frequencies at the same power

Computing Performance - the Good Old Days are Over



Free Performance Lunch is over!

(Herb Sutter 2005)

- Only transistor density still increasing exponentially
- All other trends start to flatten out

Power Wall

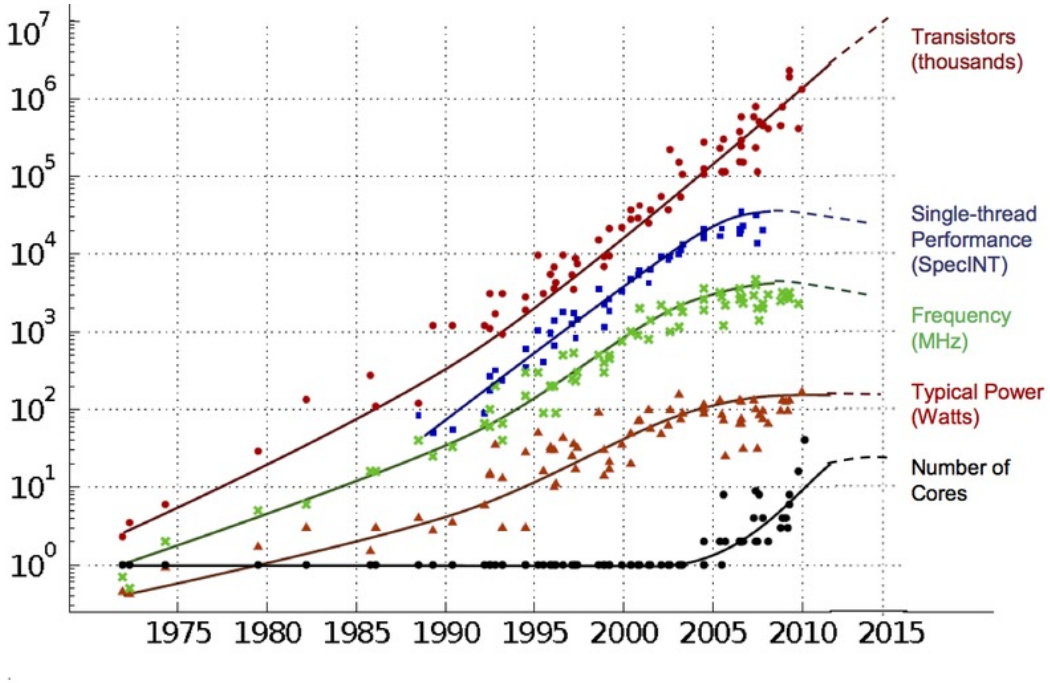
- End of Dennard scaling: no longer fixed power/area
- Dennard scaling ignored the "leakage current" and "threshold voltage"
- Too much heat to dissipate at high clock frequencies – chip will melt
- From 90nm and smaller, transistors would leak part of their power into the substrate of the processor ⇒ chip heats up
- Putting a sufficiently large # "leaky" transistors together, the chip overheats

But Moore's "Law" continues

- Single-thread (single-core) performance growth ended
  - Frequency scaling ended (Dennard scaling) ~ 4GHz
  - Instruction-level parallelism (perf/clock) scaling stalled
- *What do we do with all these transistors?*

- Performance improvements are now coming from the **increase in the number of cores** on a processor (ASIC)
- **#cores per chip doubles every 18 months *instead* of clock**

- Multiple (simple) cores at lower clock frequency are more power efficient



Original data collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond and C. Batten
Dotted line extrapolations by C. Moore

End of era of Sequential Computing … Time for **Parallel Computing**

Parallelism

- Transistor density still increasing exponentially
- Use the transistors to add more parallel units to the chip
  - *Increases throughput, but not speed*

Pros:

- Can continue to get speedups from added transistors
- Can even get speedups beyond a single chip or a single machine

Cons:

- Can't just sit and wait for things to get faster (lack of concern for systems efficiency)
- Need to work to get performance improvements
- Need to develop new frameworks and methods to parallelize automatically

- If we run in parallel on N copies of our compute unit, naively we would expect our program to run N times faster
- Does this always happen in practice?
- No! Why?

Amdahl's Law

- Let T_p be the time using p processors

$$\text{Speedup} \quad S = \frac{T_1}{T_p}$$

- Can $S > p$? (i.e., using p processors is more than p times faster than using 1 processor)
- YES! Think of a memory-bound computation and the use of cache

- Assume
  - f is the parallelizable portion of an algorithm.
  - There is a task that takes time τ on 1 process
  - A fraction f of that task will take time $\tau f/p$ on p processors
- What is the maximum possible speedup?

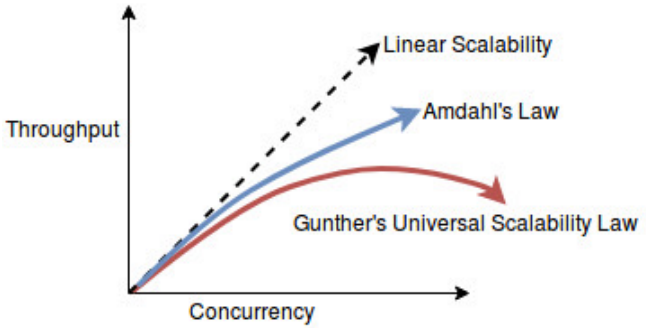
$$S(p) = \frac{\tau}{(1-f)\tau + \tau f/p} = \frac{1}{(1-f) + f/p} \xrightarrow{p\to\infty} \frac{1}{1-f}$$

- if $f = 0.99$, S =100
- Even if only 1% is non parallelizable, the max. performance is limited !

- Another measure $N_{1/2}$: how many processors are required to achieve half of the possible performance?
- For max. performance: $S(N_{1/2}) = \frac{1}{1-f} \Rightarrow N_{1/2} = \frac{f}{1-f}$ (e.g., $N_{1/2} = 9900$ if $f = 0.99$)

Amdahl's Law

- "Diminishing marginal returns" as we increase the parallelism
- Can never actually achieve a linear or super-linear speedup as the amount of parallel workers increases
- Not always true in practice - sometimes we do get super-linear speedup.
- In general: can run many heterogeneous machines in parallel in a cluster.



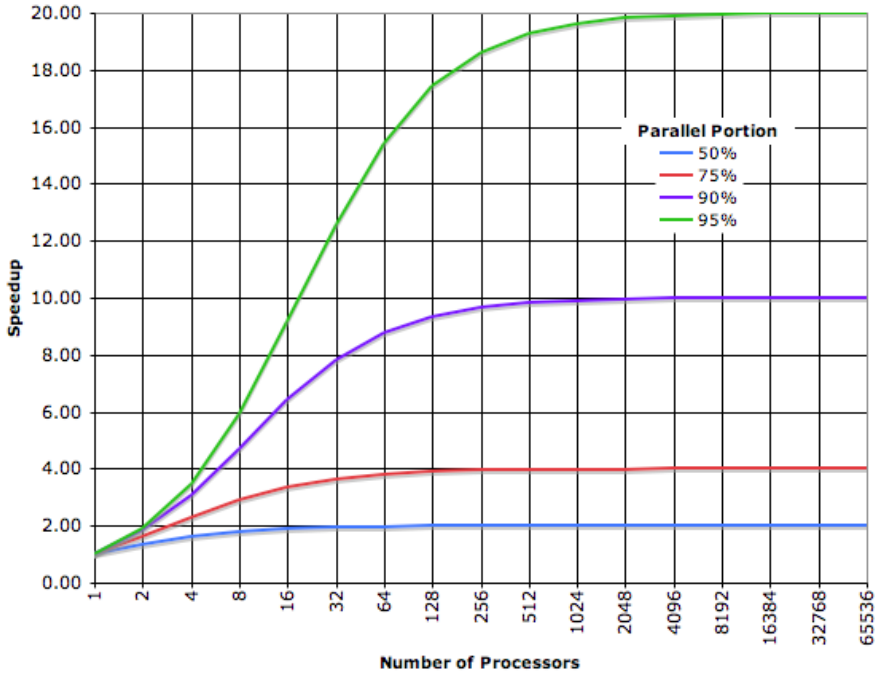
Gunther's Law: effect of coherency delay

CPUs

- Many parallel cores
- Deep parallel cache hierarchies – taking up most of the area
- Often many parallel CPU sockets in a machine

GPUs

- Can run way more numerical computations in parallel than a CPU
- Loads of lightweight cores running together



Sources of Parallelism

On CPUs

- Instruction-Level Parallelism
- Single-Instruction Multiple-Data (SIMD) / Vector parallelism
- Multi-core parallelism (significant cost to synchronize)
- Multi-socket parallelism / NUMA (Non-Uniform Memory Access)

On GPUs

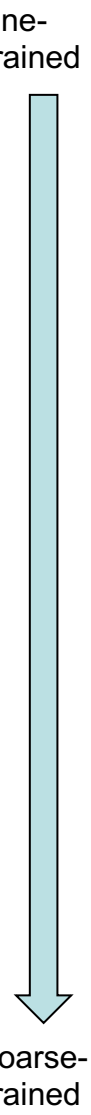
- Stream processing

On specialized accelerators and ASICs

- Various options, only limited by the available transistors

Distributed Setting

- Many workers communicate over a network
  - Possibly heterogeneous workers including CPUs, GPUs, and ASICs
- Usually no shared memory abstraction
  - Workers communicate explicitly through passing messages
- Latency much higher than all other types of parallelism
  - Often need fundamentally different algorithms to handle this



Modern Computation Paradigms

Smartphones

- ~11 TFLOPs

FLOP: Floating point operations per second
1 Peta (10^{15}) = 1000 Tera = 10^6 Giga

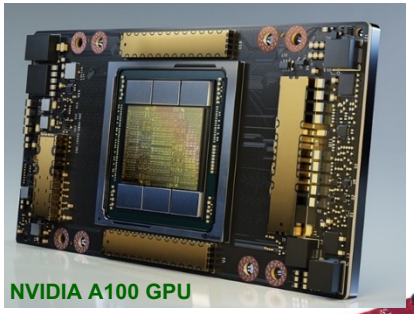
Large ("single") Computers

- 1.102 exaFLOPs
- See: www.top500.org

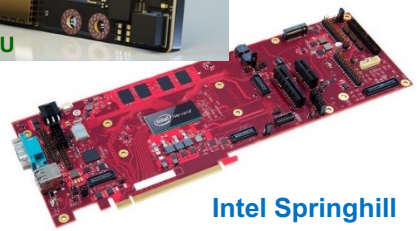
HPE Frontier OLCF-5

Specialized Hardware

- Focus on subset of operations
- GPU (Graphical Processing Unit), FPGA (Field Programmable Gated Array)
- ~10 TFLOPs



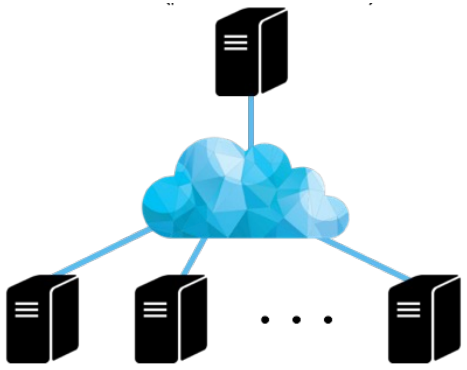
NVIDIA A100 GPU

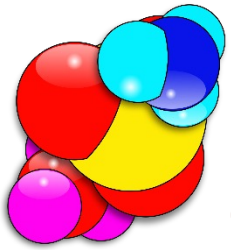


Intel Springhill

Distributed Computation

- 474 Peta FLOPS



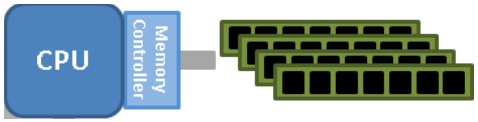


FOLDING @HOME

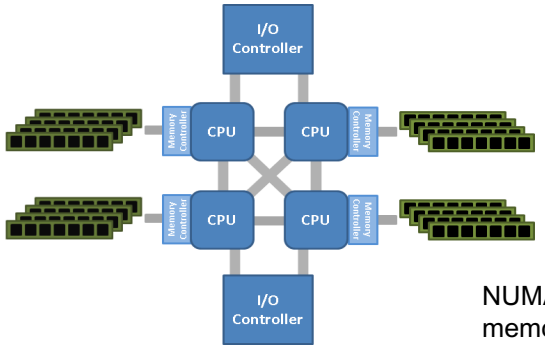
Speeding up ML

Parallel

Single machine, multicore

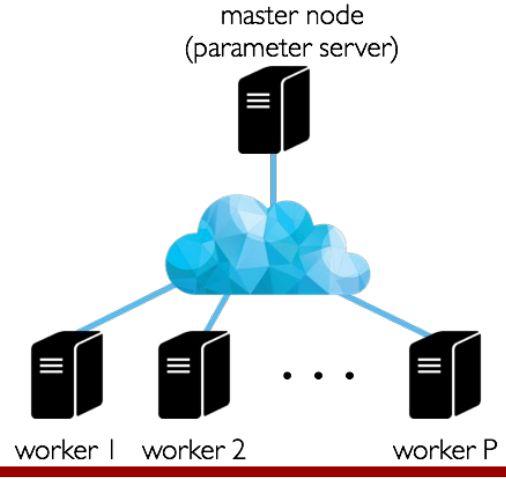


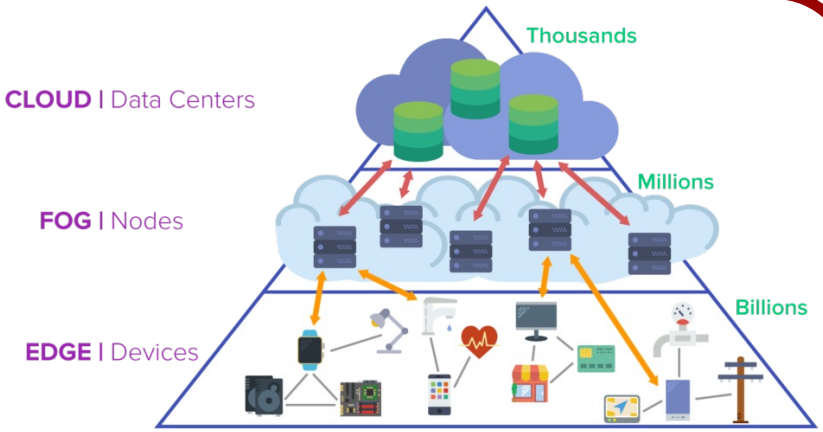
Multi-socket NUMA



NUMA: non-uniform memory access

Distributed





EURECOM
Sophia Antipolis

Parallel vs. Distributed

Parallel

- Single machine, many cores (usually up to100s)
- Shared memory

  (all cores have access to RAM)
- Communication costs to RAM: low

Distributed

- Many machines (usually up to1000s) connected via network
- Shared-nothing architecture

  (each node has its own resources)
- Communication costs: non-negligible

Scaling up vs. Scaling out

Scaling up vs. Scaling out

What we'd ideally like:

- Cores ∞
- Memory (RAM) ∞
- Communication Cost 0
- Cost to build 0

Feasible solutions:

- Scaling up: Getting the largest machine possible, with maxed out RAM
- Scaling out: Getting a bunch of machines, and linking them together

Scaling up

Pros

- RAM communication cheap (no network)
- Less implementation overheads
- Smaller power/smaller footprint

Cons

- 100s cores/machine = expensive
- Smaller fault tolerance
- Limited upgradability

Scaling out

Pros

- Much cheaper (especially on EC2)
- Can replace faulty parts
- Better fault tolerance (if it matters)

Cons

- Network bound
- Major implementation overheads
- Large power footprint

Distributed Machine Learning

Main idea: use multiple machines to do learning

Why distribute?

- Train more quickly
- Train models too large to fit on one machine
- Train when data are inherently distributed

Distributed ML/Computing

- Involves 2 or more machines collaborating on a single* task by communicating over a network.
- Requires explicit (i.e., written in software) communication among the workers.
- No shared memory abstraction! (Unlike parallelism on 1 machine)

- Key Principle: Overlapping **Computation** and **Communication**
- Communicating over the network results in high latency
- For best performance: workers still be doing useful work while communication is going on
  - rather than having to stop and wait for the communication to finish (stall)
  - asynchronous communication may help a lot here

Recap: GD vs. SGD in a Nutshell

Convergence rates (convexity)

- Full (batch) GD: For convex f, with L-Lipschitz gradient, under suitable stepsizes:
$$f(\boldsymbol{x}^t) - f(\boldsymbol{x}^*) = O(1/t)$$
- SGD: For convex f, under diminishing stepsizes (along with other conditions):
$$\mathbb{E}[f(\boldsymbol{x}^t)] - f(\boldsymbol{x}^*) = O(1/\sqrt{t})$$
(Here t can be a full iteration or a batch)

Convergence rates (strong convexity)

Under *strong convexity* assumptions on f (with parameter μ)

- Full GD: For strongly convex f, with L-Lipschitz gradient, for suitable stepsizes GD has a linear rate:
$$f(\boldsymbol{x}^t) - f(\boldsymbol{x}^*) = O(\rho^t) \quad \text{where } \rho < 1$$

- SGD: under strong convexity (plus other assumptions as before), SG sequence has sublinear rate:
$$\mathbb{E}[f(\boldsymbol{x}^t)] - f(\boldsymbol{x}^*) = O(1/t)$$

- Can we do better than sublinear convergence for the SG method?

Minibatch Gradient Descent

- GD: all examples at once $\boldsymbol{x}^{t+1} = \boldsymbol{x}^t - \eta_t \frac{1}{n} \sum_{i=1}^{n} \nabla f(\boldsymbol{x}^t; \boldsymbol{\xi}_i)$
- SGD: one example at a time $\boldsymbol{x}^{t+1} = \boldsymbol{x}^t - \eta_t \nabla f(\boldsymbol{x}^t; \boldsymbol{\xi}_{i_t})$
- Is it *really all or nothing*?

An intermediate approach

- **Minibatch GD**: calculate gradients over a subset of training points instead of just one

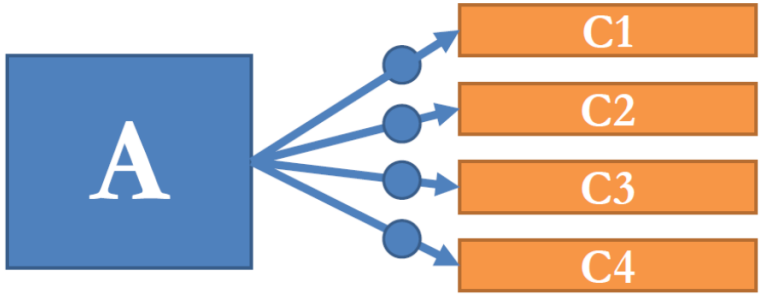
$$\boldsymbol{x}^{t+1} = \boldsymbol{x}^t - \eta_t \frac{1}{|B_t|} \sum_{i \in B_t}^{n} \nabla f(\boldsymbol{x}^t; \boldsymbol{\xi}_i)$$

where B_t is sampled uniformly from the set of all subsets of $\{1, \dots, n\}$ of size b.

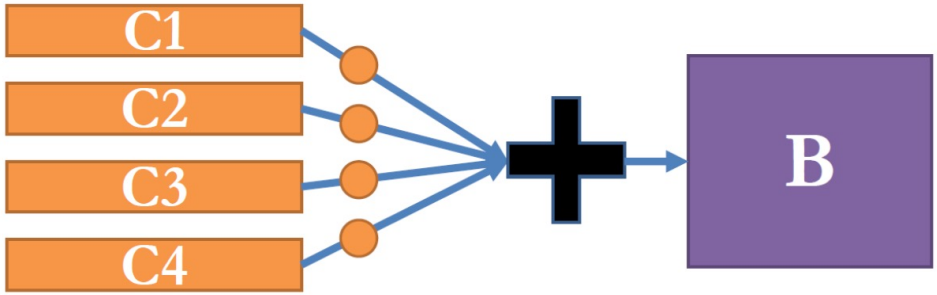
- The parameter b is the **batch size** (typically $b \ll n$)
- Not a descent method, but "closer" to one
- Less time to compute each update than GD (only needs to sum up b gradients, rather than n)
- But iterations more expensive than SGD

Basic patterns of distributed communication

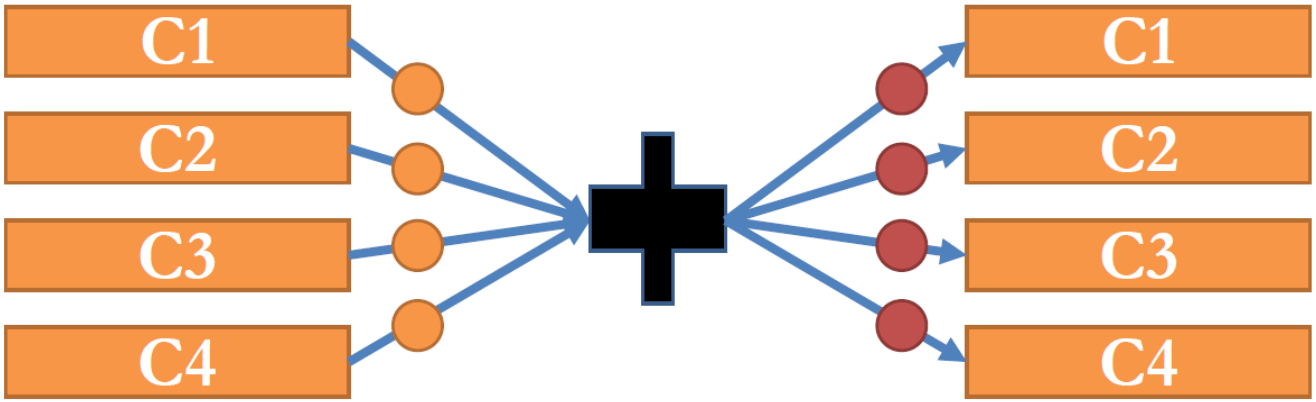
Broadcast: Machine A sends data to many machines



Reduce: Compute some reduction of data on multiple machines and materialize result on B

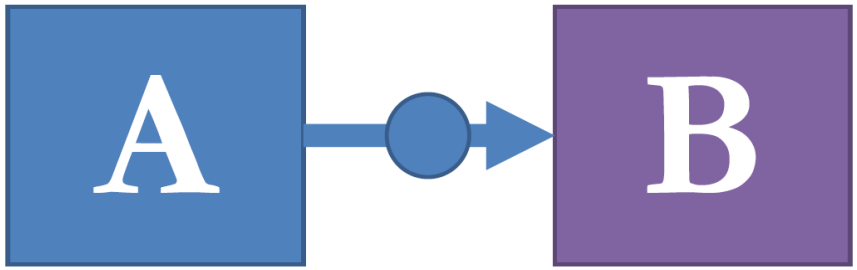


All-reduce: Compute some reduction of data on multiple machines and materialize result on all those machines.

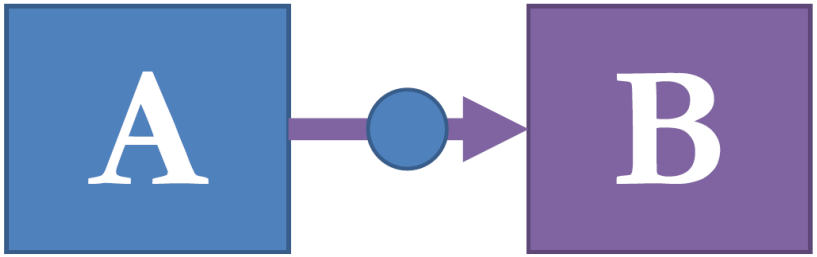


Basic patterns of distributed communication

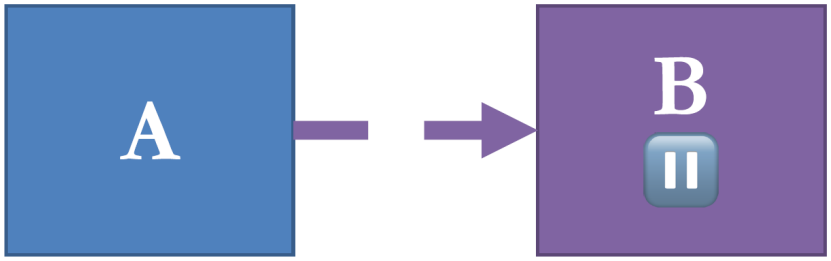
Push: Machine A sends some data to machine B



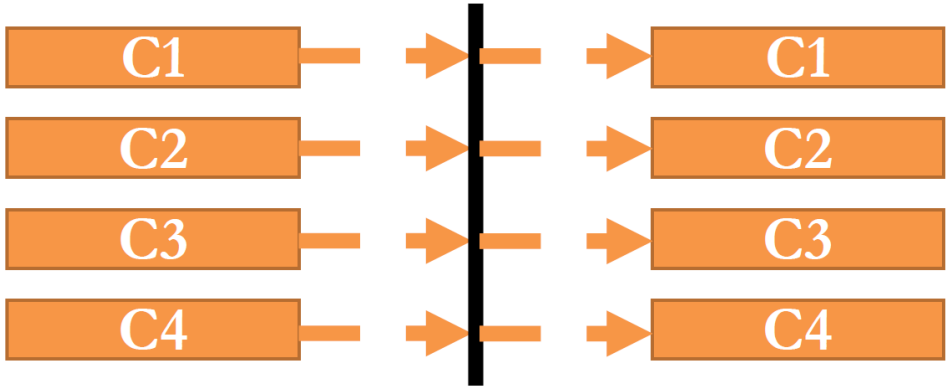
Pull: Machine A requests some data from machine B



Wait: Pause until another machine says to continue

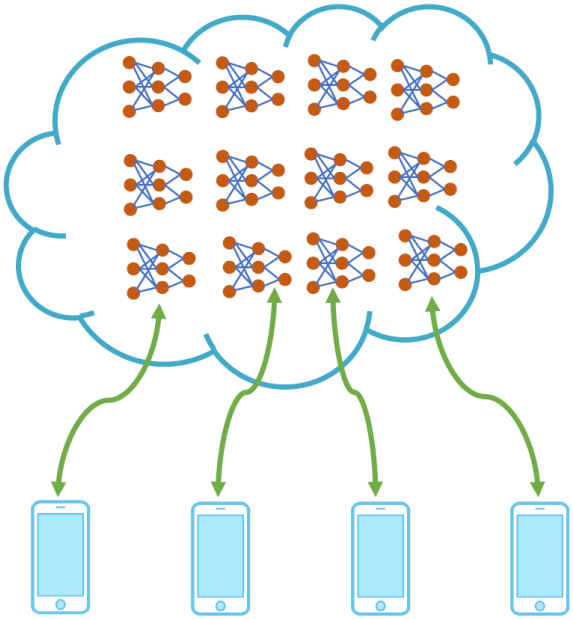


Barrier: Wait for all workers to reach some point in their code



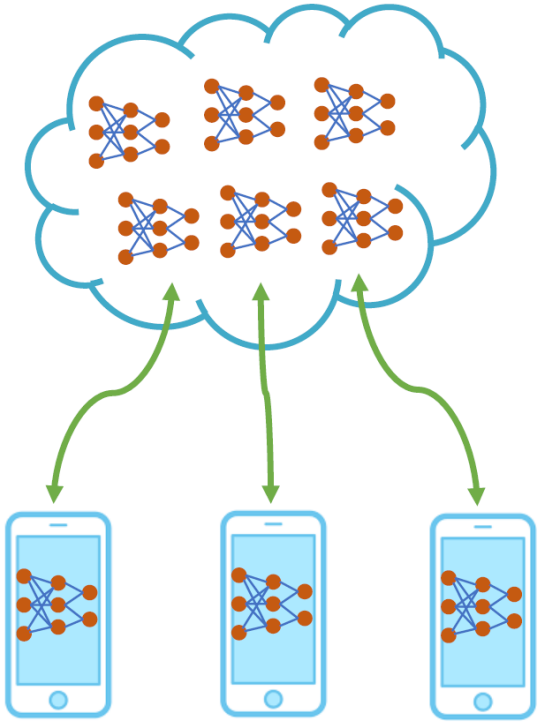
Distributed ML Models

Centralized



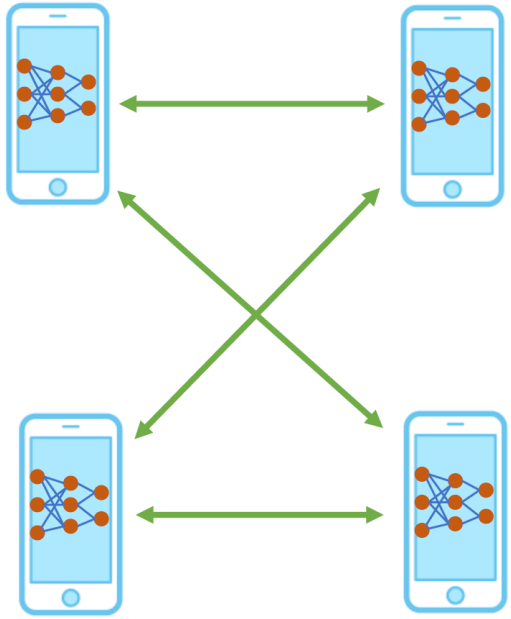
- ML in the cloud with “dumb” end-user devices
- All data is in the cloud/edge
- Inference and decision making in the cloud/edge
- No data privacy

Distributed



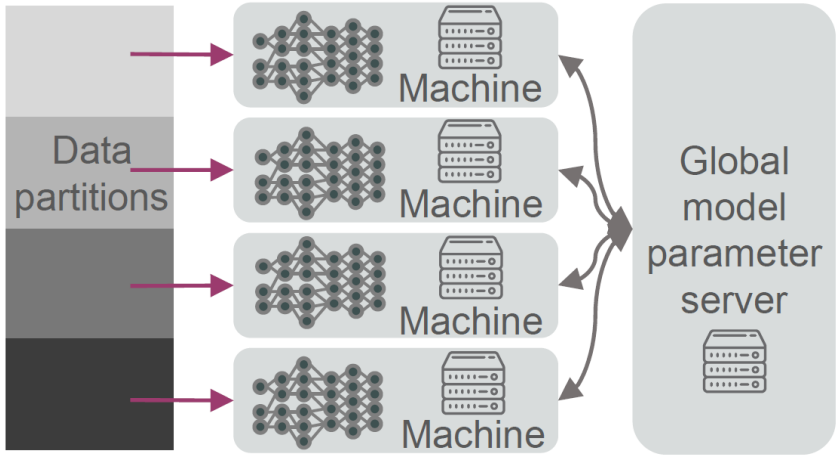
- ML in the cloud + on-user-device ML
- Only part of the data is in the cloud/edge
- Use the cloud/edge *but* smartly
- Privacy-preserving (e.g., federated learning)

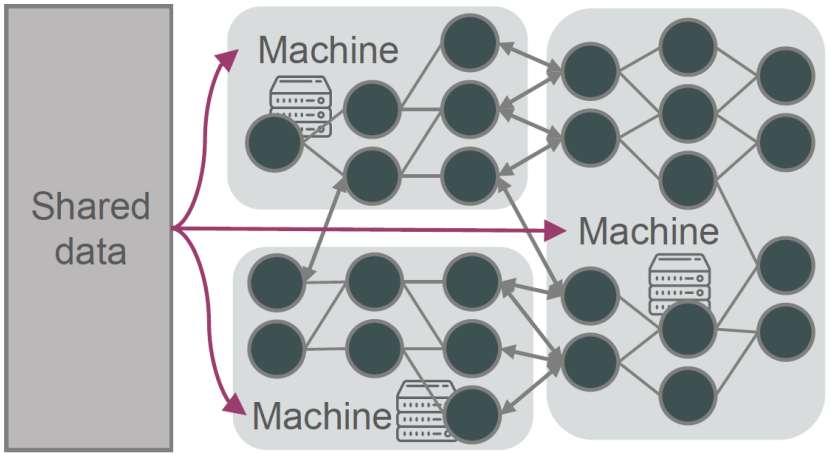
Decentralized



- No infrastructure (e.g., cloud) needed
- Data fully distributed
- Collaborative intelligence
- Privacy-preserving
  (sharing models instead of data)

Parallelism in Distributed Machine Learning





Data parallelism

- Replicate the model on each machine
- Divide data into multiple partitions
- Then respectively train copies of the model in parallel with their own allocated data samples.
- Efficiency of model training can be improved.
- *Most common approach*: typically preferable in terms of implementation, fault tolerance, and parallelizability

Model parallelism

- Replicate the data on each machine
- First split a large ML model into multiple parts
- Then feed data samples for training these segmented models in parallel.
- Improved training speed
- Resolve issues when the model is larger than the device memory.
- Attractive approach for massive models

How to Parallelize and Distribute SGD?

Parallelism within the Gradient Computation

- Compute the **gradient samples themselves** in parallel

$$x^{t+1} = x^t - \alpha \nabla f\left(x^t; y_{i_t}\right)$$

Issues:

- We run this so many times, we will need to synchronize a lot
- Computing $\nabla f\left(x^t; y_{i_t}\right)$ is cheap (even for deep networks) - $O(d)$

Typically used: instruction level parallelism, SIMD parallelism, model parallelism

Parallelism across Iterations

- Compute **multiple iterations** of SGD in parallel – Parallelize the outer loop

$$x^{t+1} = x^t - \alpha \nabla f\left(x^t; y_{i_t^{(1)}}\right)$$
$$x^{t+1} = x^t - \alpha \nabla f\left(x^t; y_{i_t^{(2)}}\right)$$
$$x^{t+1} = x^t - \alpha \nabla f\left(x^t; y_{i_t^{(3)}}\right)$$

Issues:

- all gradients computed on the same model
- Outer loop is sequential – need for fine-grained locking and frequent synchronization

Typically used: multi-core/multi-socket/cluster parallelism

Parallelism with Minibatching

- Parallelize across the **minibatch sum**

$$x^{t+1} = x^t - \alpha \frac{1}{B} \sum_{b=1}^{B} \nabla f(x^t; y_{i_{b,t}})$$

- All-Reduce, M workers, such that $B = M \cdot B'$

$$x^{t+1} = x^t - \alpha \frac{1}{M} \sum_{m=1}^{M} \frac{1}{B'} \sum_{b=1}^{B'} \nabla f(x^t; y_{i_{m,b,t}})$$

- Assign the computation of the sum when $m = 1$ to worker 1, the computation of the sum when $m = 2$ to worker 2, etc.
- After all gradients are computed, we perform the outer sum with an **all-reduce** operation

- After this **all-reduce**, the whole sum (which is essentially the minibatch gradient) will be present on all the machines

- So each machine can now update its copy of the parameters
- Since sum is same on all machines, the parameters will update in lockstep
- Statistically equivalent to sequential SGD!

Distributed Minibatch SGD with All-Reduce

Pros

- The algorithm is easy to reason about, since it's statistically equivalent to minibatch SGD
  - and we can use the same hyperparameters for the most part
- The algorithm is easy to implement
  - since all worker machines have the same role and runs on top of standard distributed computing primitives

Cons

- We're not overlapping computation and communication.
  - While the communication for the all-reduce is happening, the workers are idle
- The effective minibatch size is growing with the number of machines
  - If we don't want to run with a large minibatch size for statistical reasons, this can prevent us from scaling to large numbers of machines using this method
- Potentially requires lots of network bandwidth to communicate to all workers

Issues:

- Still run this so many times, we need to synchronize a lot
- Can have a tradeoff with statistical efficiency, since too much minibatching can harm convergence

Typically used: all types of parallelism

Training Examples in Distributed ML

Where do we get the training examples from?

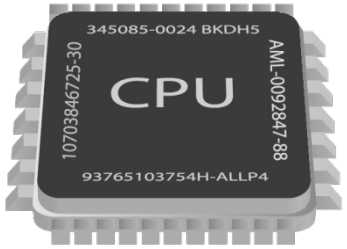
Training data servers

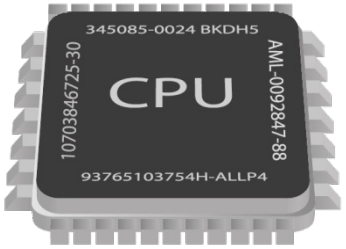
- Have one or more non-worker servers dedicated to storing the training examples (e.g., a distributed in-memory filesystem)
- The worker machines load training examples from those servers.

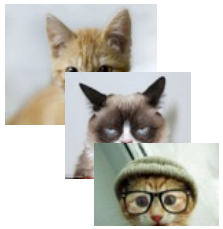
Partitioned dataset

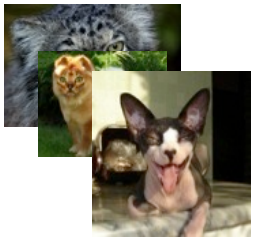
- Partition the training examples among the workers themselves and store them locally in memory on the workers.

Distribute the Effort









Several Issues

Theory

- Minibatch SGD vs. 1-sample SGD
- Many models weaker than 1
- Delays and Slow Nodes
- Communication Costs

Practice

- Barriers to entry / High Cost
- Implementation overhead
- Nontrivial choice of ML framework

Algos & Architectures

- Downpour SGD
- Hogwild!
- Delay-tolerant Algos for SGD
- TensorFlow
- Elastic Averaging SGD

Asynchronous Parallelism

- SGD is inherently a sequential algorithm
  - If we run it step-by-step: good convergence but slow
  - If we run it asynchronously: faster but suboptimal comm. among workers can lead to poor convergence

Limits of Parallel Performance

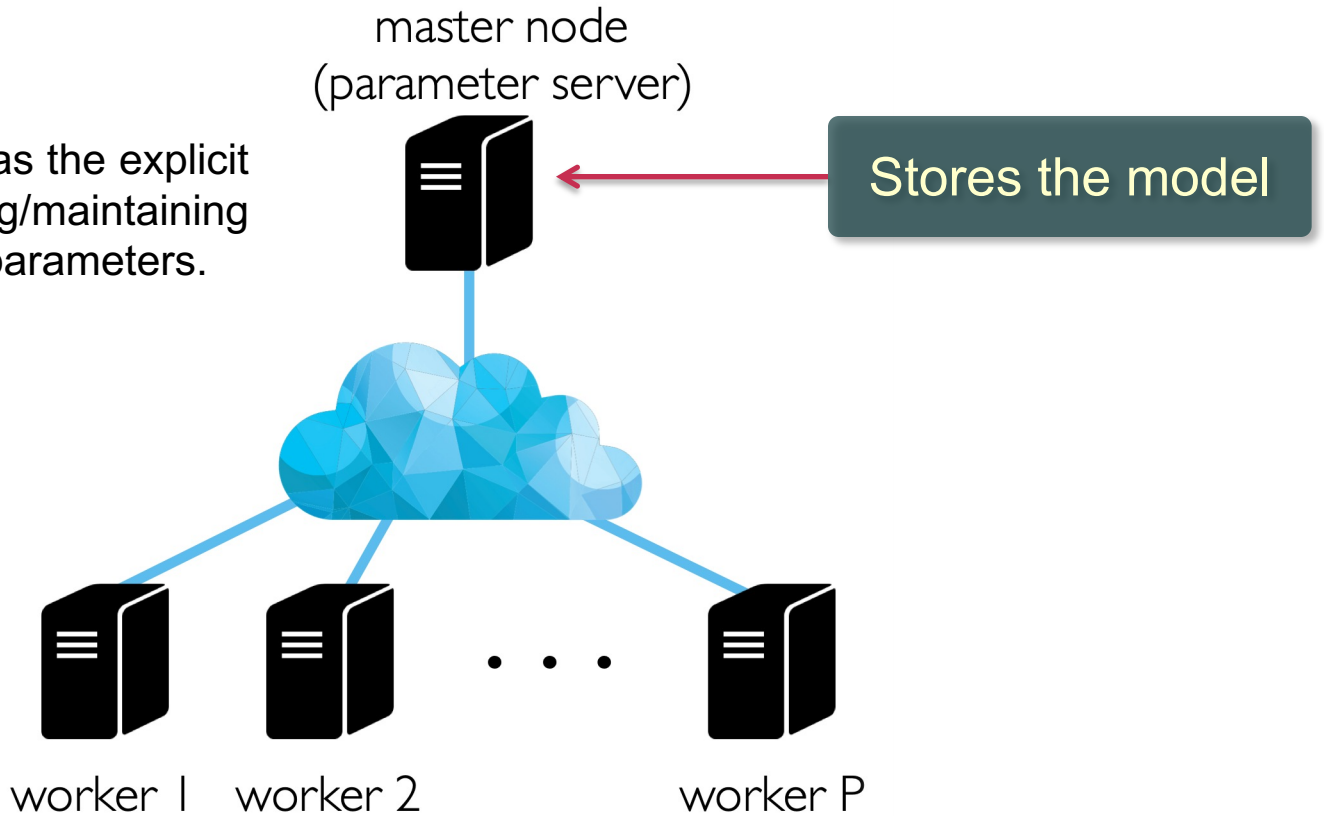
- **Synchronization**: need to synchronize to keep the workers aware of each other's updates to the model – otherwise can introduce errors

- Synchronization can be very expensive
  - Have to stop all the workers and wait for the slowest one
  - Have to wait for several round-trip times through a high-latency channel

- *How about just don't synchronize?*
  - Not synchronizing adds errors
  - But our methods are noisy (already) – maybe these errors are fine
  - If we don't synchronize, get almost perfect parallel speedup

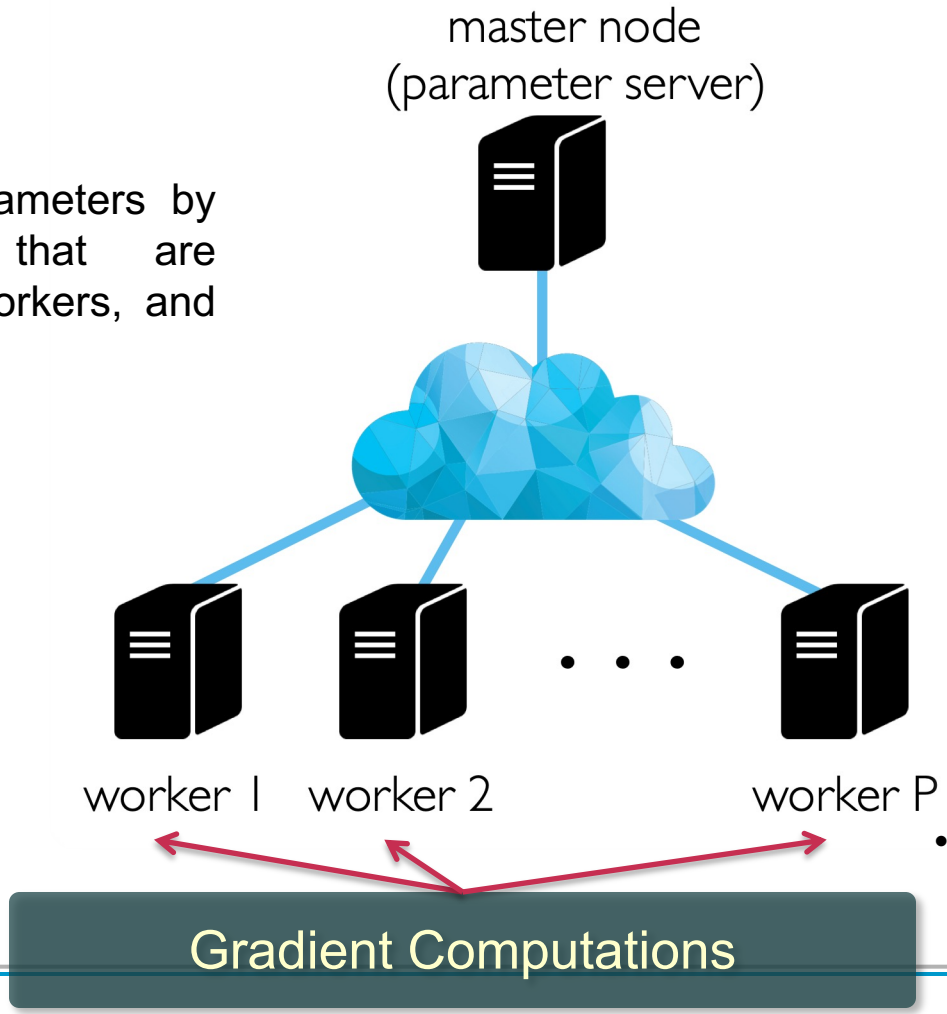
Parameter Server Model

A single machine (PS) has the explicit responsibility of storing/maintaining the current value of the parameters.



Parameter Server Model

PS updates its parameters by using gradients that are computed by the workers, and pushed to the PS

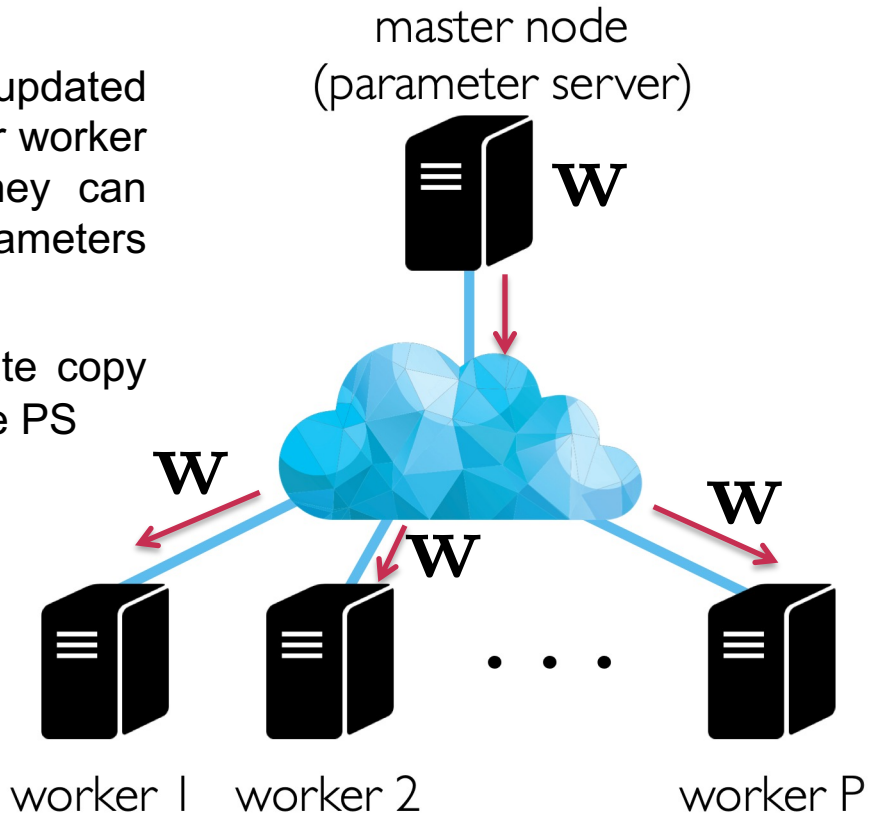


- Each worker computes gradients on minibatches of the data

Parameter Server Model

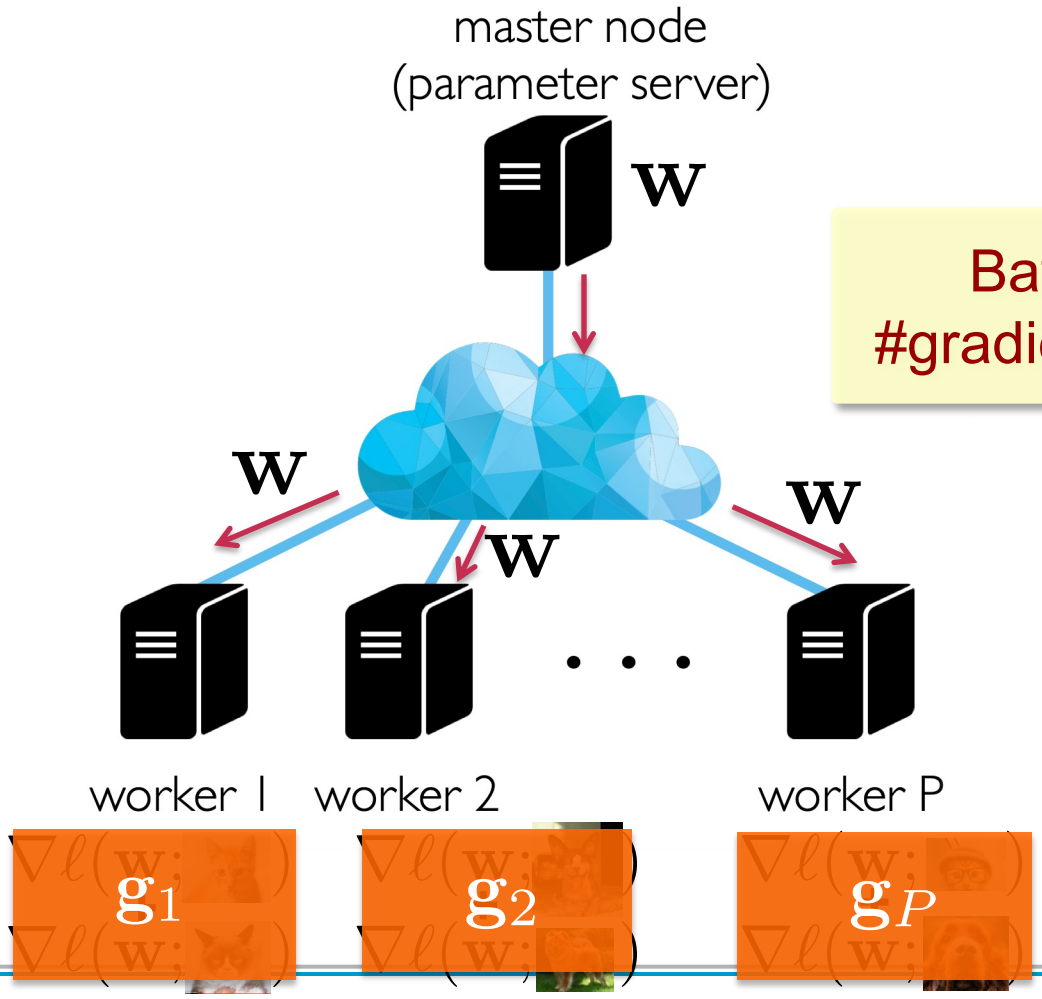
Periodically

- PS broadcasts its updated parameters to all other worker machines, so that they can use the updated parameters to compute gradients.
- Workers pull an update copy of the weights from the PS



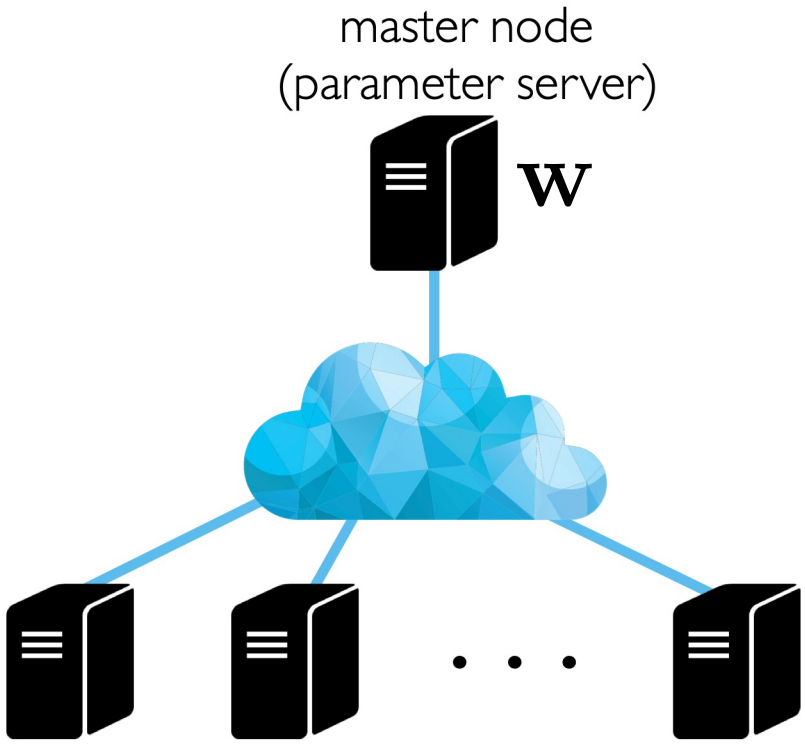
All this can be done synchronously or asynchronously

Parameter Server Model



Batch size B:
#gradients / iteration

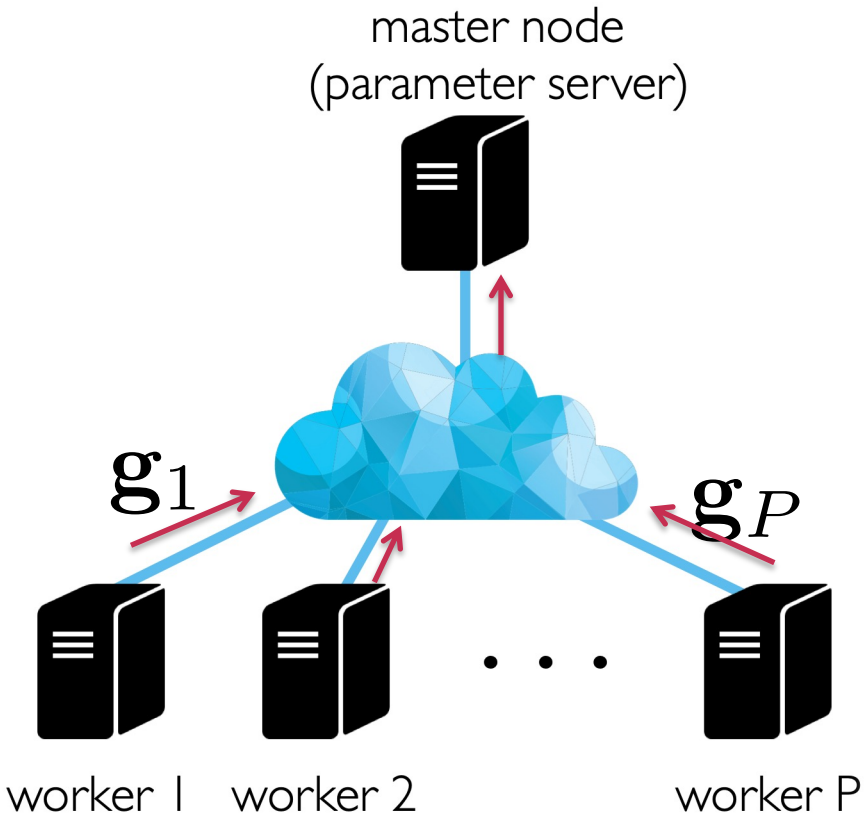
Parameter Server Model



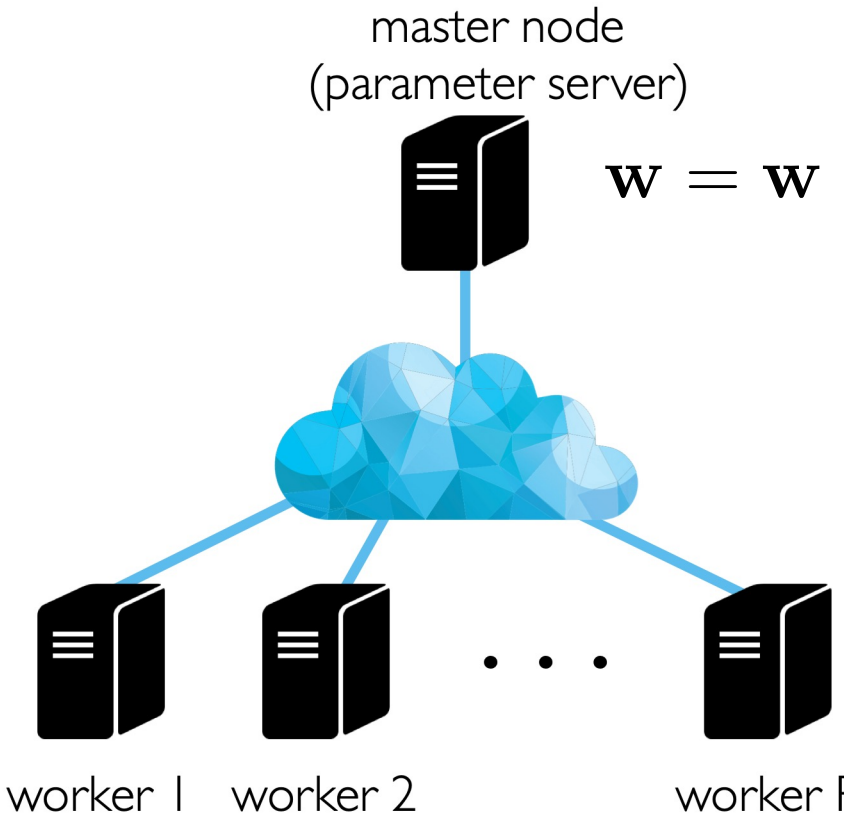
$$\mathbf{g}_1 = \sum_{i \in \mathcal{S}_1} \nabla \ell((\mathbf{x}_i, y_i); \mathbf{w})$$

$$\mathbf{g}_P = \sum_{i \in \mathcal{S}_P} \nabla \ell((\mathbf{x}_i, y_i); \mathbf{w})$$

Parameter Server Model



Parameter Server Model



$$\mathbf{w} = \mathbf{w} - \gamma \sum_{i=1}^{P} \mathbf{g}_i$$

Repeat distributed iterations until happy with the model

Parameter Server Model

Two options when learning with a PS

Synchronous distributed training

- Similar to All-reduce, but with gradients summed on a central parameter server
- Still equivalent to sequential minibatch SGD

Asynchronous distributed training

- Compute and send gradients & add them to the model as soon as possible
- Broadcast updates whenever they are available

Multiple Parameter Servers

- Use multiple PS machines when the parameters are too many for a single PS to handle
- We partition the parameters among the multiple PSs
  - Each server is only responsible for maintaining the parameters in its partition
  - When a worker wants to send a gradient, it will partition that gradient vector and send each chunk to the corresponding PS
  - Later, the worker will receive the corresponding chunk of the updated model from that PS machine
- This lets us scale up to very large models!

Other Ways to Distribute

Decentralized learning

- Idea: learn without any central coordination
  - No parameter server; each worker has its own copy of the model
- Workers update by doing the following:
  - Run an SGD update step using an example stored on that worker,
  - Average its current model with the model(s) of some other workers, usually its neighbors in some sparse graph
- This limits total communication
- Sometimes called a gossip algorithm

Local SGD

- Many parallel workers update their own copy of the model by running SGD steps using their own local data
- Periodically the workers average by taking an all-reduce step
  - Like all-reduce SGD, but the all-reduce happens less frequently than at every SGD iteration
- Can generalize better than large-batch SGD
  - *"Don't use large minibatches, use local SGD" ICLR 2020*

Tradeoffs in Distributed Computing/ML

Distributed systems have fundamental tradeoffs

Communication vs. Computation

- Parallelizing a sequential algorithm ⇒ all computing nodes need to synchronize their states ⇒ communication cost
- Parallelization reduces *computation time* but incurs non-negligible *communication overhead*.

Example: minimize objective function $f: \Theta \to \mathbb{R}$ wrt $w \in \Theta$

- SGD starts from a fixed initial parameter w_0.
- At iteration *t*, it takes a noisy gradient vector g_t such that $\mathbb{E}[g_t] = \nabla f(w_t)$, then the algorithm updates the vector by $w_{t+1} = w_t - \gamma_t g_t$

- Parallelize the update formula:
  - at the *t*-th iteration, every machine computes an independent copy of the noisy gradient based on local data
  - SD: average of local noisy gradients $g^{(t)} = \frac{1}{P}\sum_{i=1}^{P} g_i^{(t)}$ ($g_i^{(t)}$: noisy gradient on the i-th machine)
- Computing $g^{(t)}$ requires aggregating the local information on *P* machines ⇒ synchronization
- All-reduce communication rounds:
  - $O(P)$ time for star networks
  - $O(\log P)$ time for fully connected networks

Tradeoffs in Distributed Computing/ML

Communication vs. Accuracy

- To achieve a high statistical accuracy, communication is an essential to aggregate useful information across machines (separate locations) $\Rightarrow$ communication cost.
- Practical concern for high-dimensional data

Example: compute the rank of matrix $X \in \mathbb{R}^{n \times d}$

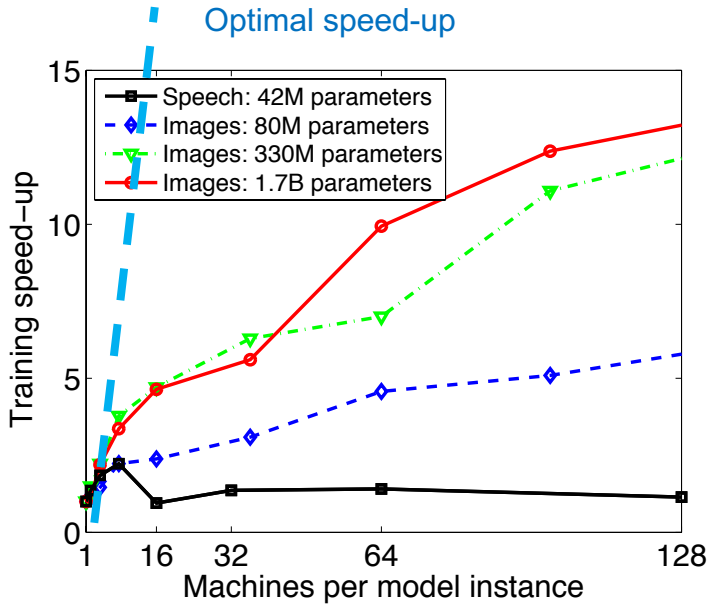
- Computation: result can be encoded in $O(\log d)$ bits
- Communication cost: e.g., $O(d^2)$ (too expensive for large d)
- Each i-th machine $X_i \in \mathbb{R}^{n_i \times d}$.
- Simple algorithm: (1) each machine computes the covariance matrix $X_i^T X_i$
  (2) sends it to the master node
  (3) master node aggregates the local covariance matrices to compute $X^T X$ and then the rank.

Algorithmic Efficiency vs. Ease of Implementation

- In distributed algorithms, many decisions (data partitioning, communication protocols, conflict management, fault tolerance, etc.) must be correctly made to build an efficient implementation.

How does distributed ML perform?

- Convergence speed of minibatch SGD?
- How can we measure speed-ups?
- Communication is expensive ⇒ how often do we average?
- How do we choose the right model?
- What happens with delayed nodes (stragglers)?
- Fault tolerance? Robustness?
- …



- *Reality ~ 100x worse than optimal…Why so slow?*
- *Two factors control run-time*

Time to accuracy ϵ =
[time per data pass]
× [#passes to accuracy ϵ]

How to analyze Parallel Algorithms

- Main measure of performance

$$\text{Speedup} = \frac{\text{Time of serial } \mathcal{A} \text{ to accuracy } \epsilon}{\text{Time of parallel } \mathcal{A} \text{ to accuracy } \epsilon}$$

Example: Gradient Descent

Serial

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \gamma \cdot \frac{1}{n} \sum_{i=1}^{n} \nabla f_i(\mathbf{x}_k)$$

Parallel

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \gamma \cdot \frac{1}{n} \left(\sum_{i_1=1}^{P} \nabla f_{i_1}(\mathbf{x}_k) + \ldots + \sum_{i_P=n-P+1}^{n} \nabla f_{i_P}(\mathbf{x}_k) \right)$$

- Convergence is invariant of allocation
- Both algorithms reach to same accuracy after *T* iterations
- Speedup is independent of convergence rate
- **Embarrassingly Parallel $O(\#cores)$ speedup // Not true for minibatch SGD**

How to evaluate Minibatch SGD?

- Measure of performance

$$\text{worst case speedup} = \frac{\text{bound on \#iter of SGD to } \epsilon}{\text{bound on \#iter of Parallel SGD to } \epsilon}$$

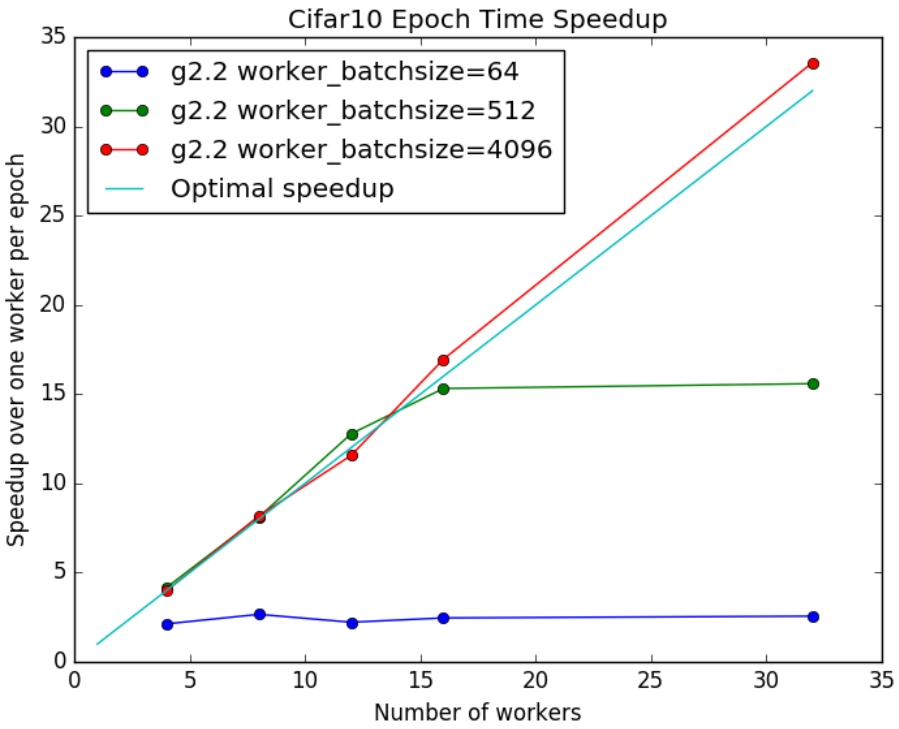
- How to evaluate **run-time**?

Time to accuracy ϵ =
[time per data pass]
$\times$ [#passes to accuracy ϵ]

- **Minibatch SGD vs. serial SGD?**
- **Convergence after *T* gradient computations?**

Why so slow?

- **Speed-up (time for data pass) becomes better with larger *B***
- **Communication is the Bottleneck**
  - beyond ~10 machines, network overhead becomes dominant

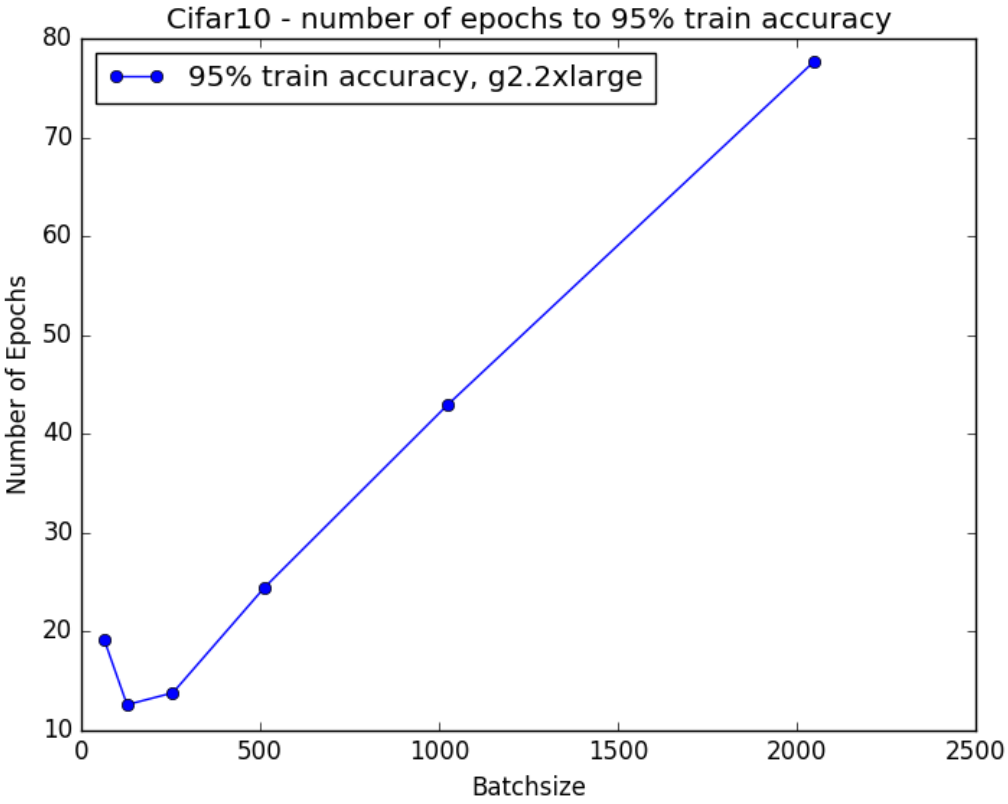


Time per pass:
time for
dataset_size/batch_size

Bigger Batch
⇒ Less Communication
(smaller time per epoch)

How about Larger Batches?

- If small batch is bad ⇒ maximize it
- Number of passes to ϵ accuracy: worse with larger B



Large Batch
⇒ worse training error
(more passes to accuracy ϵ)

Large Batch
⇒ worse generalization

Convergence of Minibatch SGD

- System perspective: **large** batch **is better**
- Statistical perspective: **small** batch **is better**

- **Q:** *Are large batches fundamentally bad?*

- Main question: How does minibatch SGD compare to1-sample SGD?

- Idea: compare T iterations of 1-sample SGD with T/B iterations of minibatch SGD

- Expectation: updating the global model every B gradients is not a big issue (under some assumptions).

Minibatch SGD - Setup

- Supervised learning - Empirical Risk Minimization

Model
$w \in \mathcal{W} \subseteq \mathbb{R}^d$

$$F(w) = \frac{1}{n}\sum_{i=1}^{n} f(w, z_i) := \frac{1}{n}\sum_{i=1}^{n} f_i(w)$$

$$w^* = \arg\min_{w \in \mathcal{W}} F(w)$$

- *Differentiable loss functions*
- *β–smooth*
- *λ–strongly convex*
- *μ–Polyak-Lojasiewicz*
  $\frac{1}{2}\|\nabla F(w)\|_2^2 \geq \mu(F(w) - F(w^*))$

Minibatch SGD

$$w_{(k+1)B} = w_{kB} - \gamma \sum_{i=kB}^{(k+1)B-1} f_{s_i}(w_{kB})$$

Single iteration of 1-sample SGD

- Progress in a single iteration:

$$\mathbb{E}\|w_1 - w^*\|^2 = \mathbb{E}\|w_0 - \gamma \nabla f_{s_1}(w_0) - w^*\|^2$$
$$= \mathbb{E}\|w_0 - w^*\|^2 - 2\gamma \mathbb{E}\langle \nabla f_{s_1}(w_0), w_0 - w^* \rangle + \gamma^2 \mathbb{E}\|\nabla f_{s_1}(w_0)\|^2$$
$$= \mathbb{E}\|w_0 - w^*\|^2$$
$$- 2\gamma \mathbb{E}\langle \nabla f(w_0), w_0 - w^* \rangle + \gamma^2 \mathbb{E}\|\nabla f_{s_1}(w_0)\|^2$$

- When minibatch SGD “works”, we expect B times more progress!

Single iteration of *B*-sample SGD

- Progress in a single minibatch iteration:

$$\|w_B - w^*\| = \|w_0 - w^* - \gamma \sum_{i=1}^{B} \nabla f_{s_i}(w_0)\|^2$$

$$= \|w_0 - w^*\|^2$$

$$- 2\gamma \left\langle \sum_{i=1}^{B} \nabla f_{s_i}(w_0), w_0 - w^* \right\rangle + \gamma^2 \left\| \sum_{i=1}^{B} \nabla f_{s_i}(w_0) \right\|^2$$

- How does it compare in expectation with 1-sample SGD?

Single iteration of *B*-sample SGD

- Progress is equal to:

$$-2\gamma\mathbb{E}\left\langle\sum_{i=1}^{B}\nabla f_{s_i}(w_0), w_0 - w^*\right\rangle + \gamma^2\mathbb{E}\left\|\sum_{i=1}^{B}\nabla f_{s_i}(w_0)\right\|^2$$

$$= 2B\gamma\mathbb{E}\left\langle\nabla f(w_0), w_0 - w^*\right\rangle + \gamma^2\mathbb{E}\left\|\sum_{i=1}^{B}\nabla f_{s_i}(w_0)\right\|^2$$

- Variance is equal to:

$$\mathbb{E}\left\|\sum_{i=1}^{B}\nabla f_{s_i}(w_0)\right\|^2 = \mathbb{E}\left(\sum_{i=1}^{B}\|\nabla f_{s_i}(w_0)\|^2 + \sum_{i=1}^{B}\sum_{j=1,j\neq i}^{B}\langle\nabla f_{s_i}(w_0), \nabla f_{s_j}(w_0)\rangle\right)$$

$$= B\cdot\mathbb{E}\|\nabla f_{s_1}(w_0)\|^2 + \sum_{i=1}^{B}\sum_{j=1,j\neq i}^{B}\mathbb{E}\langle\nabla f_{s_i}(w_0), \nabla f_{s_j}(w_0)\rangle$$

$$= B\cdot\mathbb{E}\|\nabla f_{s_1}(w_0)\|^2 + B(B-1)\mathbb{E}\|\nabla f(w_0)\|^2$$

Minibatch SGD vs. SGD Progress

- **1-sample SGD Progress:**

$$-2\gamma\mathbb{E}\left\langle \nabla f(w_0), w_0 - w^* \right\rangle + \gamma^2 \mathbb{E}\left\| \nabla f_{s_1}(w_0) \right\|^2$$

- ***B*-sample SGD Progress:**

$$-B \cdot \left(2\mathbb{E}\gamma \left\langle \nabla f(w_0), w_0 - w^* \right\rangle - \gamma^2 \mathbb{E}\left\| \nabla f_{s_1}(w_0) \right\|^2 \right.$$

$$\left. - (B-1)\gamma^2 \mathbb{E}\left\| \nabla f(w_0) \right\|^2 \right)$$

- "Extra variance" term directly controlled by B
- Simple idea: Set B so that the extra term is smaller than $\mathbb{E}\left\| \nabla f_{s_1}(w_0) \right\|^2$

Minibatch SGD vs. SGD Progress

- **If** $B = \delta \dfrac{\mathbb{E}\|\nabla f_{s_1}(w_0)\|^2}{\mathbb{E}\|\nabla f(w_0)\|^2}$

We get

$$-B \cdot \left(2\mathbb{E}\gamma \left\langle \nabla f(w_0), w_0 - w^* \right\rangle - (1+\delta)\gamma^2 \mathbb{E}\|\nabla f_{s_1}(w_0)\|^2 \right)$$

- **1- sample SGD gets:**

$$-2\gamma\mathbb{E}\left\langle \nabla f(w_0), w_0 - w^* \right\rangle + \gamma^2 \mathbb{E}\left\|\nabla f_{s_1}(w_0)\right\|^2$$

***B* times more progress! (approximately)**

Gradient Diversity

- **Gradient Diversity:** $$\Delta(w) = \frac{\sum_{i=1}^{n} \|\nabla f_i(w)\|^2}{\|\sum_{i=1}^{n} \nabla f_i(w)\|^2} = \frac{\sum_{i=1}^{n} \|\nabla f_i(w)\|^2}{\sum_{i=1}^{n} \|\nabla f_i(w)\|^2 + \sum_{i \neq j} \langle \nabla f_i(w), \nabla f_j(w) \rangle}$$

- **Key Result**

Let $w_{k \cdot B}$ be a fixed model, and let $w_{(k+1) \cdot B}$ denote the model after a mini-batch iteration with batch-size $B = \delta n \cdot \Delta(w) + 1$. Then, we have

$$\mathbb{E}\{ \|w_{(k+1)\cdot B} - w^*\|^2 \mid w_{kB}\} \leq \mathbb{E}\|w_{k\cdot B} - w^*\|^2 - B\left(2\gamma \langle \nabla f(w_{k\cdot B}), w_{k\cdot B} - w^* \rangle - (1+\delta)\gamma^2 M^2(w_{k\cdot B})\right)$$

$$M^2(w) = \frac{1}{n}\sum_{i=1}^{n} \|\nabla f_i(w)\|^2$$

- True for both global, local min, and critical points
- Universal lemma (no assumptions)

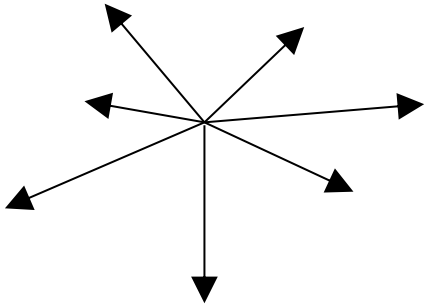
- **Corollary:** If B proportional to grad diversity ($B = \delta n \cdot \Delta(w) + 1$), we are good!!!

Gradient Diversity

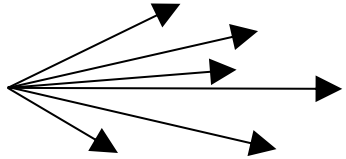
- **Measures similarity between gradients**
  - Big Diversity: Larger batches ⇒ better speedups
  - Small Diversity: Smaller Batches ⇒ worse speedup

- **Minibatch SGD:** If batch size is beyond a threshold (proportional to grad diversity) ⇒ worse accuracy
- Gradient diversity controls the distributed performance

- Examples:
  1. All gradients are orthogonal, Diversity = 1
  2. All gradients identical, Diversity = 1/*n*



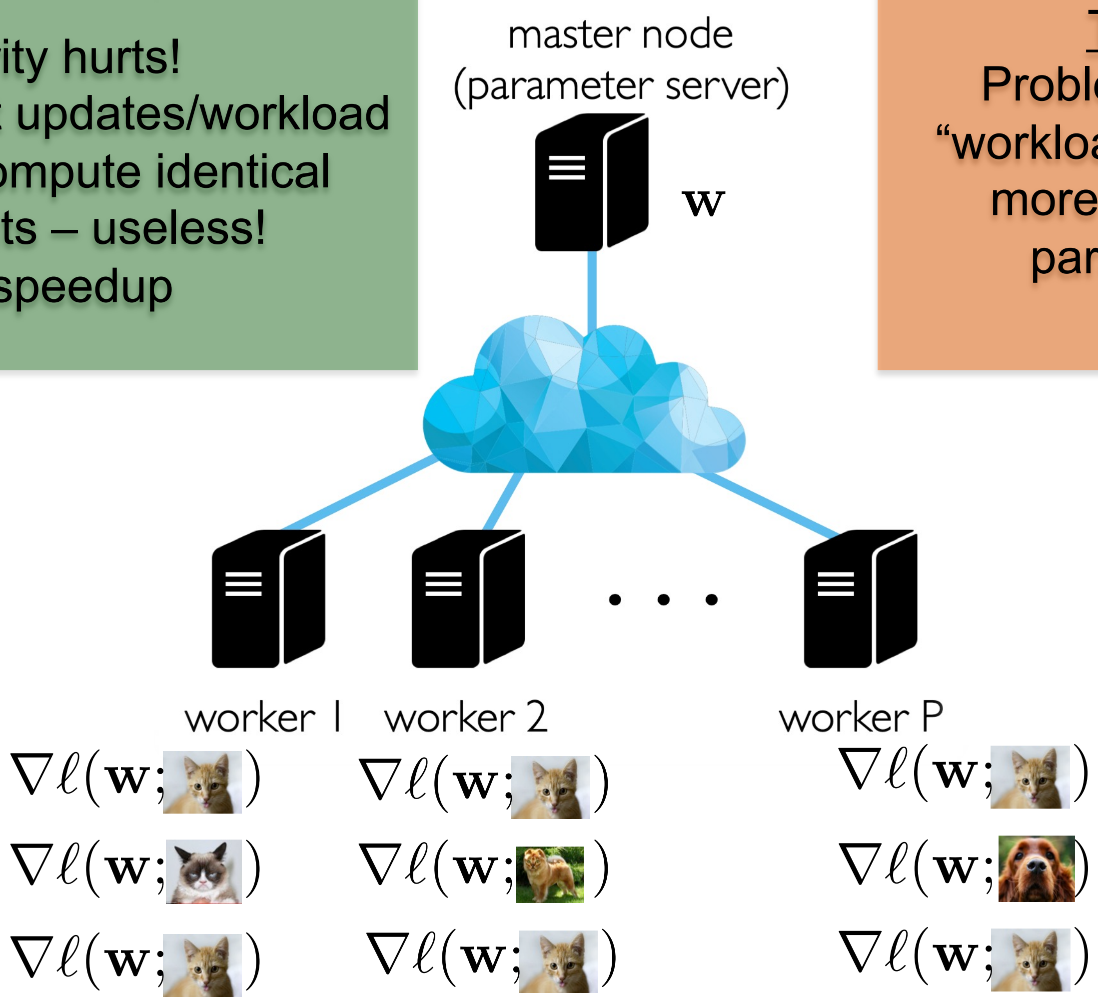
Large gradient diversity



Small gradient diversity

Large Batches bring Redundancy

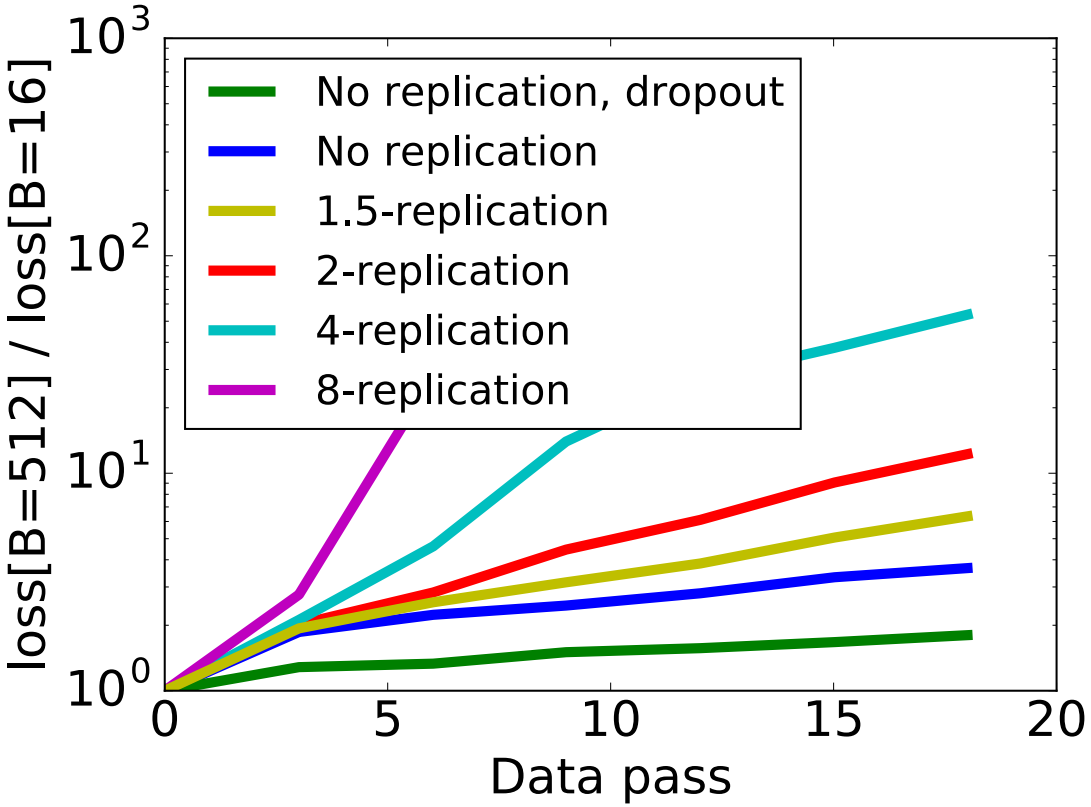
Similarity hurts!
"similar" gradient updates/workload
$\Rightarrow$ workers compute identical gradients – useless!
$\Rightarrow$ no speedup



<u>Theorem</u>
Problems with high "workload diversity" are more amenable to parallelization.

Gradient Diversity in Experiments

CIFAR-10 (CUDA ConvNet)



- **Smaller diversity ⇒ slower convergence**
- **Mechanisms to increase diversity?**
- **Many questions:** generalization? delayed nodes? how often do we average? …

How to increase diversity?

- We want to maximize

$$\mathrm{B}^*(\mathbf{w}) \leq n \cdot \frac{\sum_i \|\nabla \ell(\mathbf{w}; \mathbf{z}_i)\|^2}{\sum_i \|\nabla \ell(\mathbf{w}; \mathbf{z}_i)\|^2 + \sum_{i \neq j} \langle \nabla \ell(\mathbf{w}; \mathbf{z}_i), \ell(\mathbf{w}; \mathbf{z}_j) \rangle}$$

- Possible ways:
  - **Data pre-processing** (similar to preconditioning)
  - **Dropout** (e.g., gradient sparsification)
  - Design **deep architectures that suppress interference**

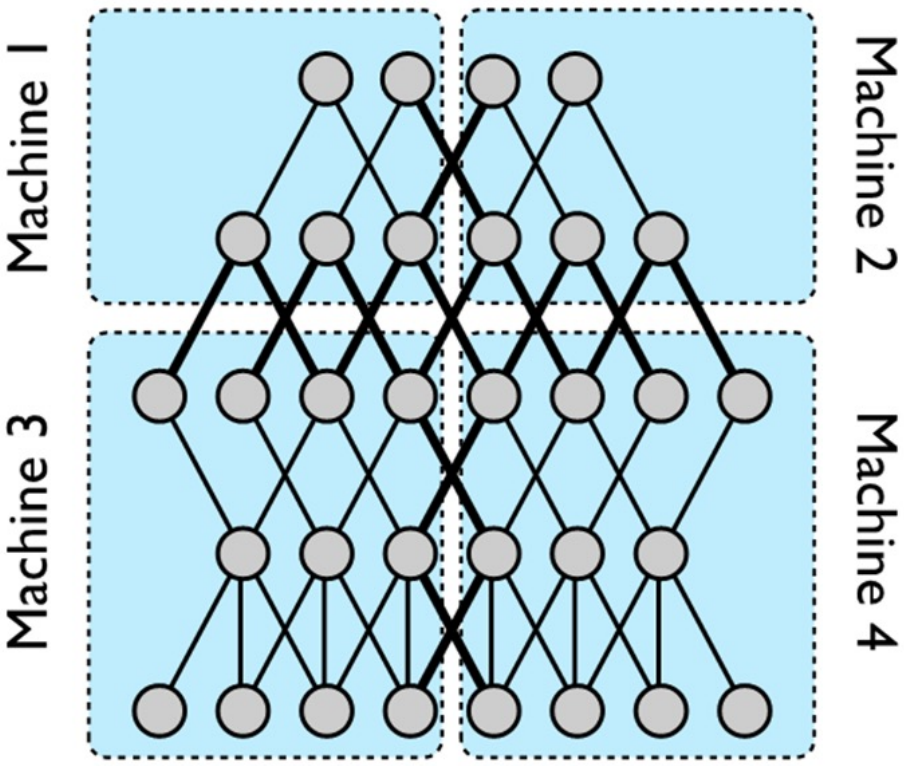
Large Scale Distributed Deep Networks

- Limitations of GPU
  - Speed-up is small when the model does not fit in the GPU memory
  - Data and parameter reduction are not attractive for large scale ML problems

How to train a deep network with billions of parameters using tens of thousands of CPU cores?

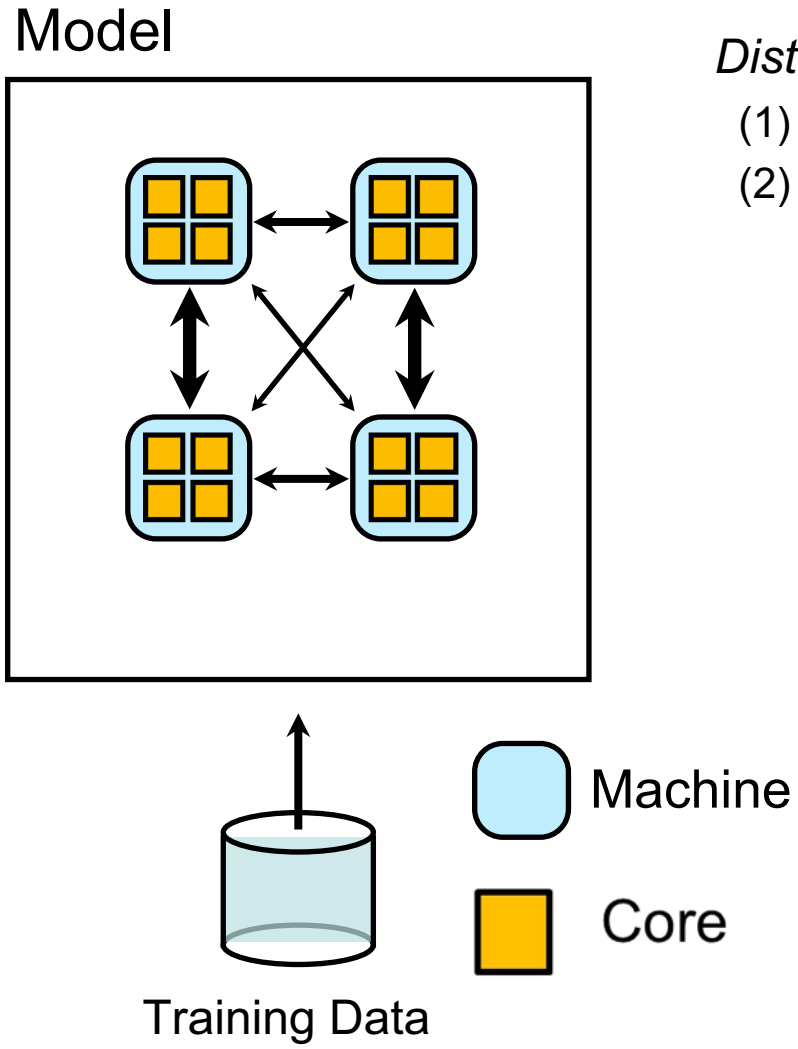
- **DistBelief** (Google Brain 2011)
  - Predecessor of Tensorflow
  - Software framework using computing clusters with 1000s of machines to train large models
  - Clusters are utilized to asynchronously compute gradients
  - Not require the problem to be either convex or sparse
  - Two main methods
    - (1) Downpour SGD
    - (2) Sandblaster L-BFGS
  - Two-level parallelism
    - (1) Model Parallelism
    - (2) Data Parallelism

Model Parallelism



- A 5 layer DNN with local connectivity is partitioned across 4 machines.
- Only nodes with edges that cross partition boundaries (**thick** lines) need to have their state transmitted between machines.
- Even in cases where a node has multiple edges crossing a partition boundary, its state is only sent to the machine on the other side of that boundary once.
- Within each partition, computation for individual nodes will thebe parallelized across all available CPU cores.

Model Parallelism



DistBelief enables model parallelism

(1) Across machines via message passing (**blue** box)

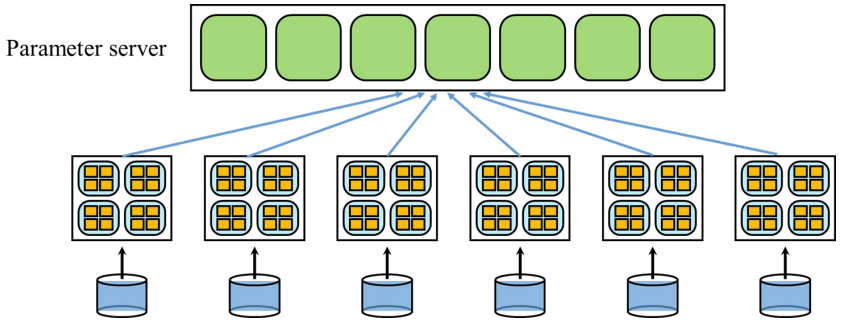
(2) Within a machine via multithreading (**orange** box)

- Performance benefits of distributing DNNs across multiple machines depends on
  - connectivity structure
  - computational needs of the model.
- Models with a large number of parameters or high computational demands typically benefit from access to more CPUs and memory…

… up to the point where communication costs dominate

Downpour SGD

- A variant of asynchronous SGD
  - leverages adaptive learning rates and supports a large number of model replicas
- Divide training data into a number of subsets and run a copy of the model on each of these subsets
- Communicate gradient updates through a centralized parameter server (PS)

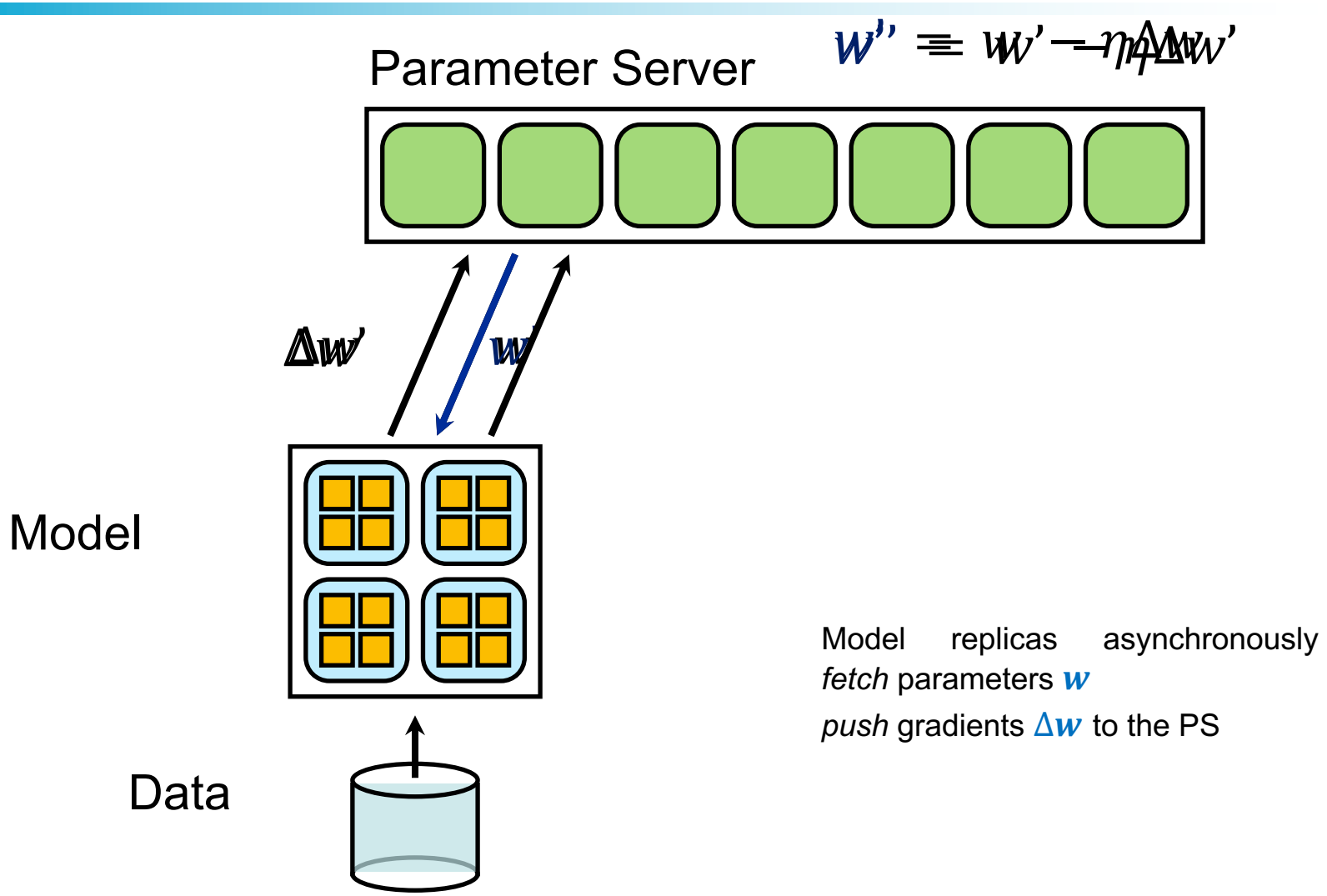


- Two asynchronous aspects: *model replicas* and *parameter shards* run independently

Issues:

- Each model replica computes its gradients based on slightly outdated parameters
- No guarantee that at any given moment each shard of the PS has undergone the same number of updates
- Subtle inconsistencies in the timestamps of parameters
- Little theoretical grounding for the safety of these operations, but it works!

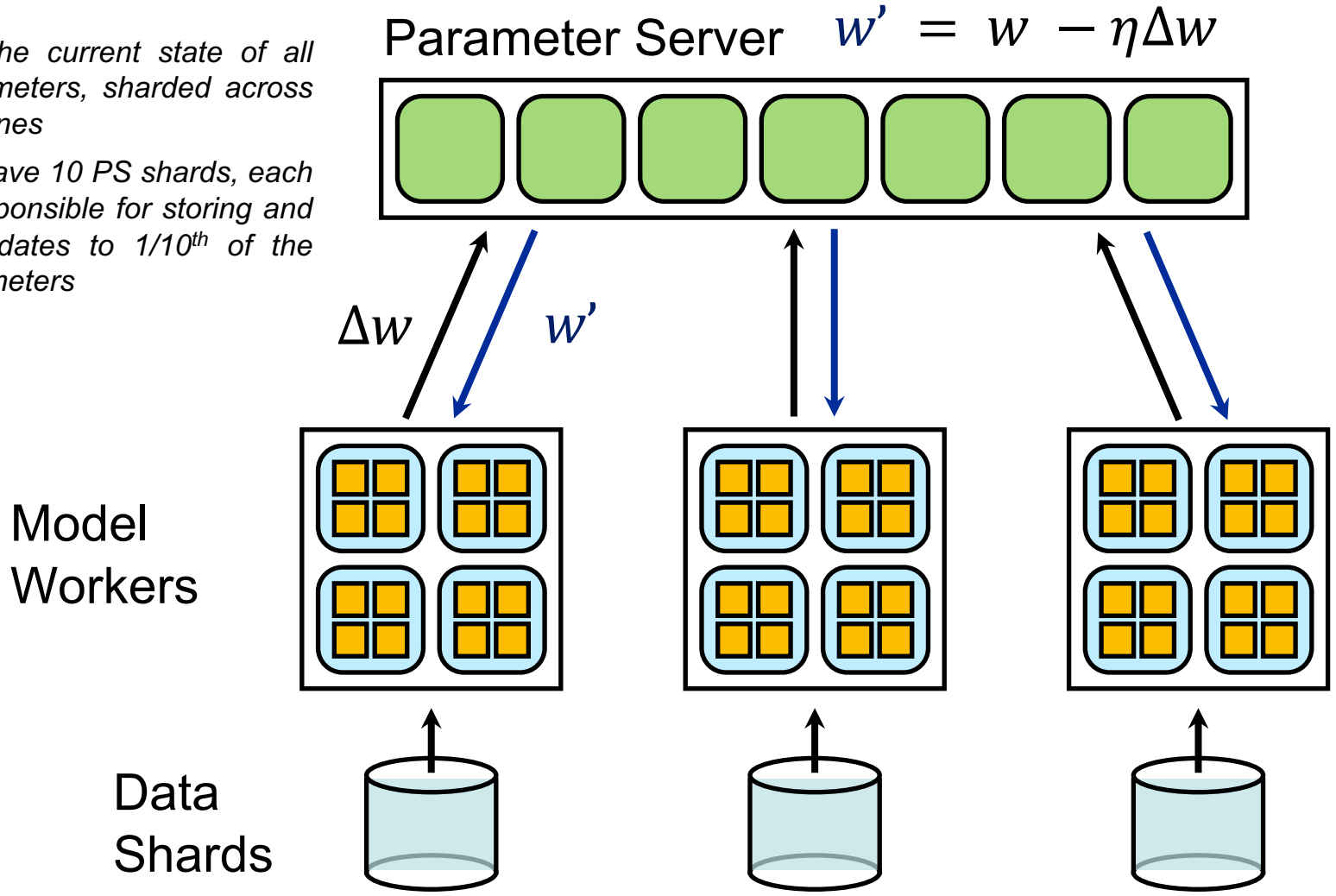
Downpour SGD



Model replicas asynchronously
fetch parameters w
push gradients Δw to the PS

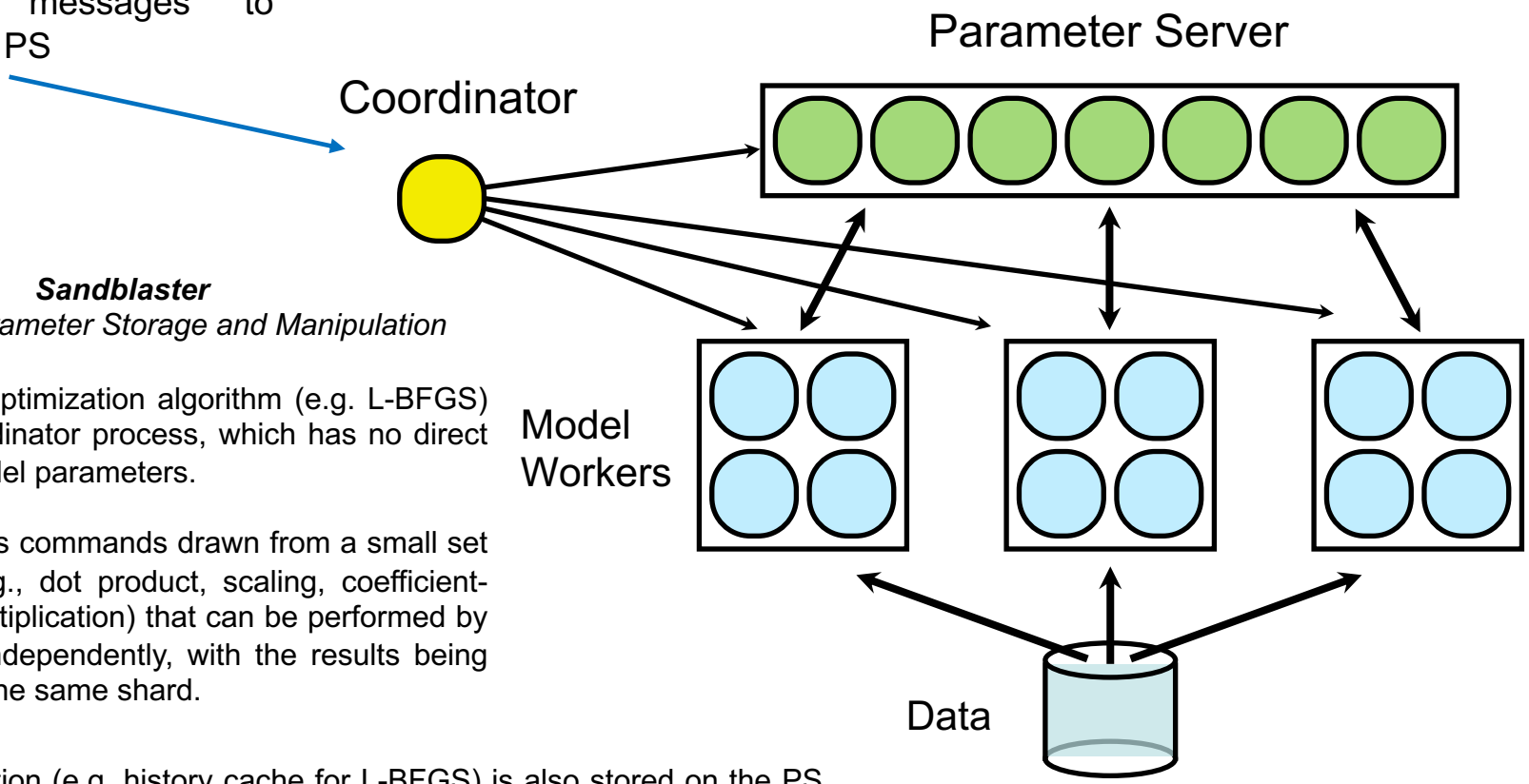
Downpour SGD

- *PS keeps the current state of all model parameters, sharded across many machines*
- *E.g., if we have 10 PS shards, each shard is responsible for storing and applying updates to 1/10th of the model parameters*



Sandblaster L-BFGS

Batch optimization orchestration via a single 'coordinator' that sends small messages to replicas and the PS



Sandblaster
Distributed Parameter Storage and Manipulation

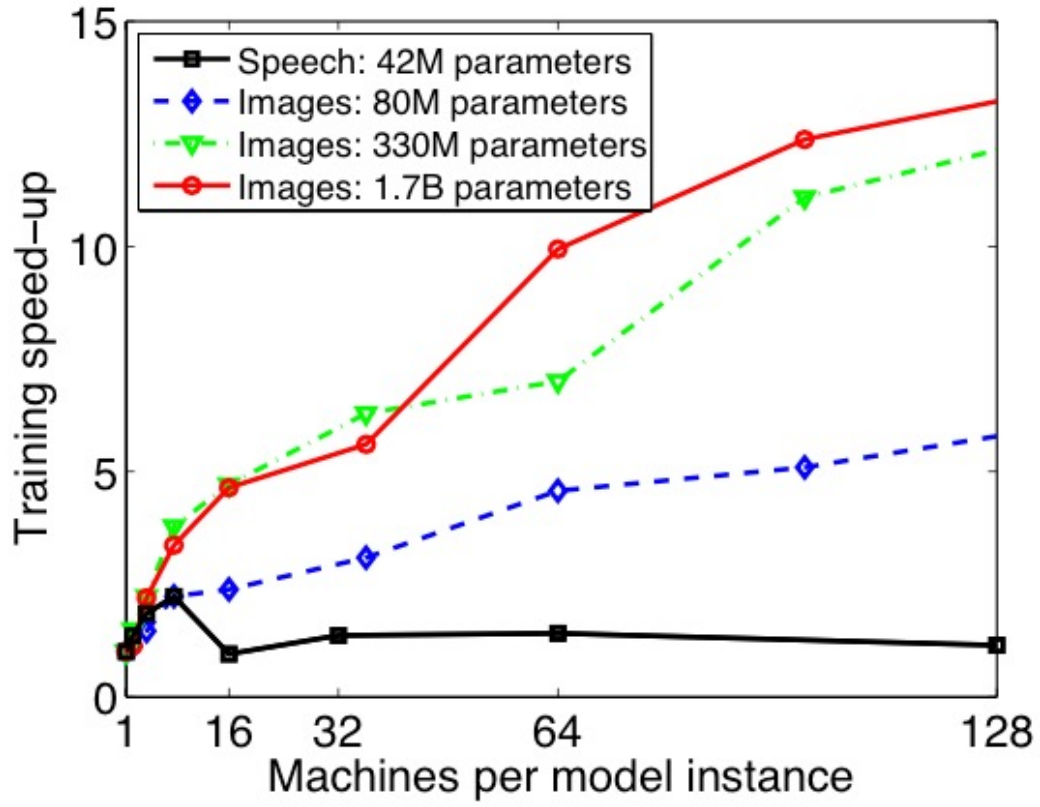
The core of the optimization algorithm (e.g. L-BFGS) resides in a coordinator process, which has no direct access to the model parameters.

Coordinator issues commands drawn from a small set of operations (e.g., dot product, scaling, coefficient-wise addition, multiplication) that can be performed by each PS shard independently, with the results being stored locally on the same shard.

Additional information (e.g. history cache for L-BFGS) is also stored on the PS shard on which it was computed

⇒ large models (billions of parameters) can run without overhead of sending all parameters and gradients to a single central server

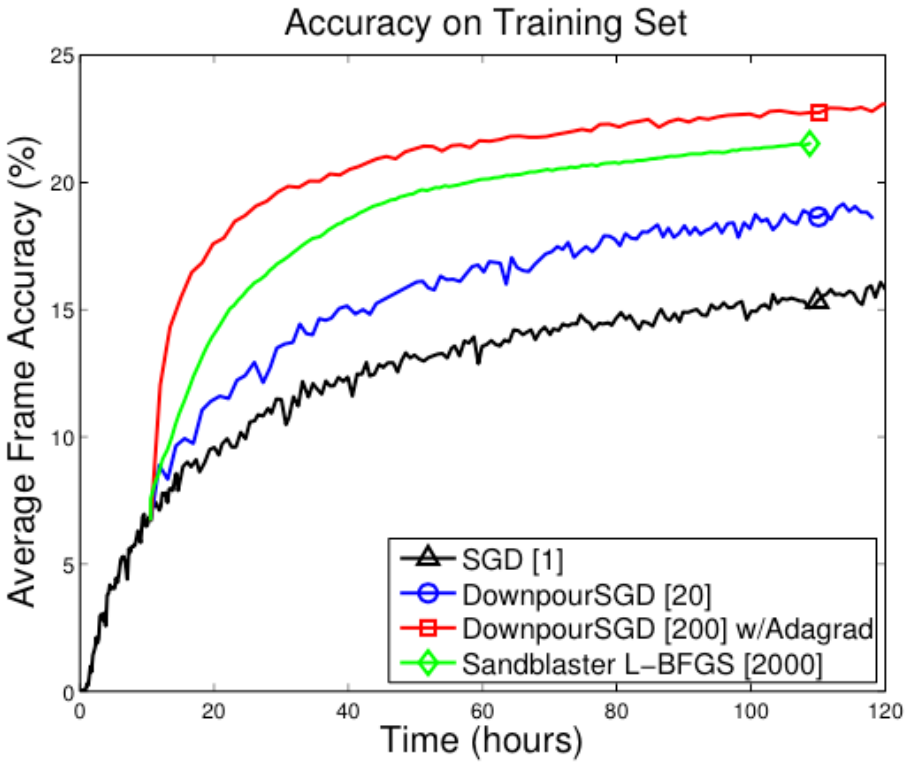
Training Speed-up Performance



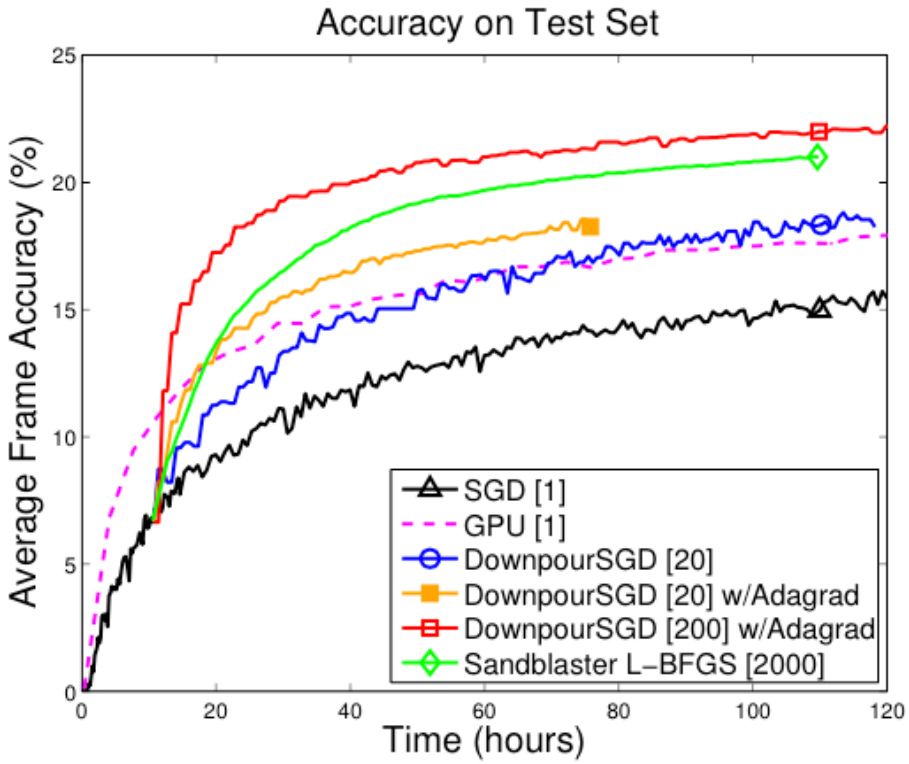
Models with more parameters benefit more from the use of additional machines than do models with fewer parameters.

"Large Scale Distributed Deep Networks" [Dean et al., NIPS 2012]

Accuracy Performance



Training accuracy (on a portion of the training set) for different optimization methods.



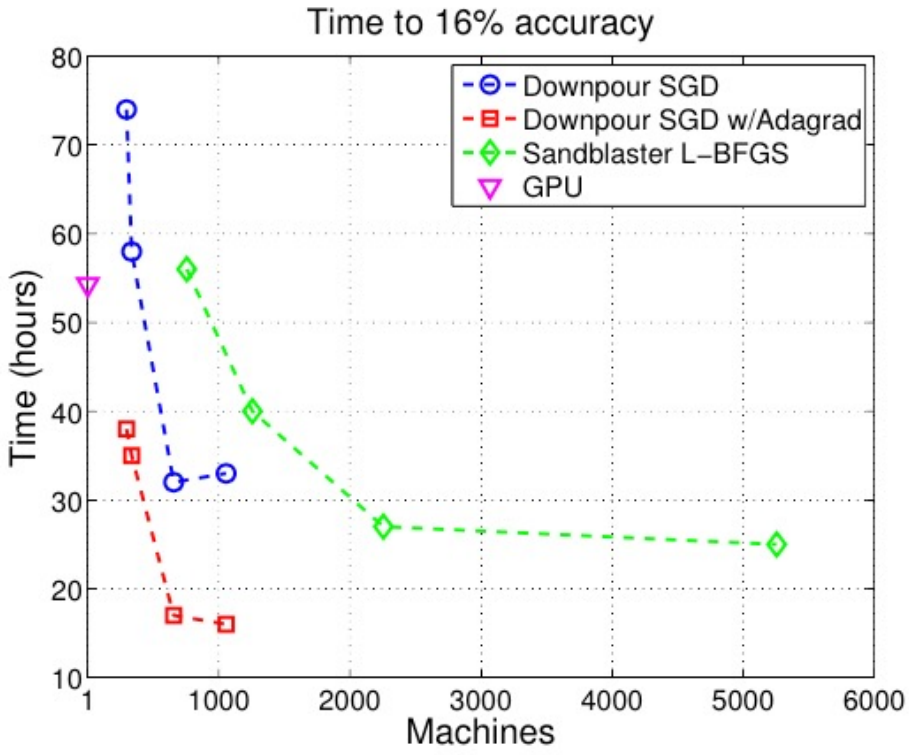
Classification accuracy on the hold out test set as a function of training time.

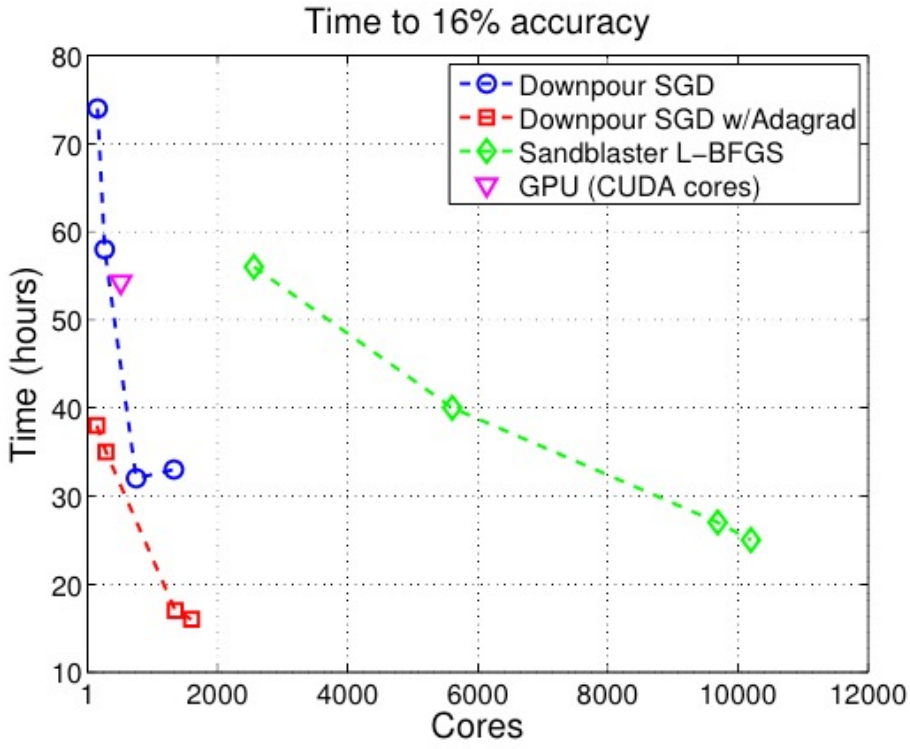
"Large Scale Distributed Deep Networks" [Dean et al., NIPS 2012]

Time to Accuracy Performance

Resource consumption vs. Performance tradeoff

Time each method took to reach 16% accuracy as a function of machines and utilized CPU cores

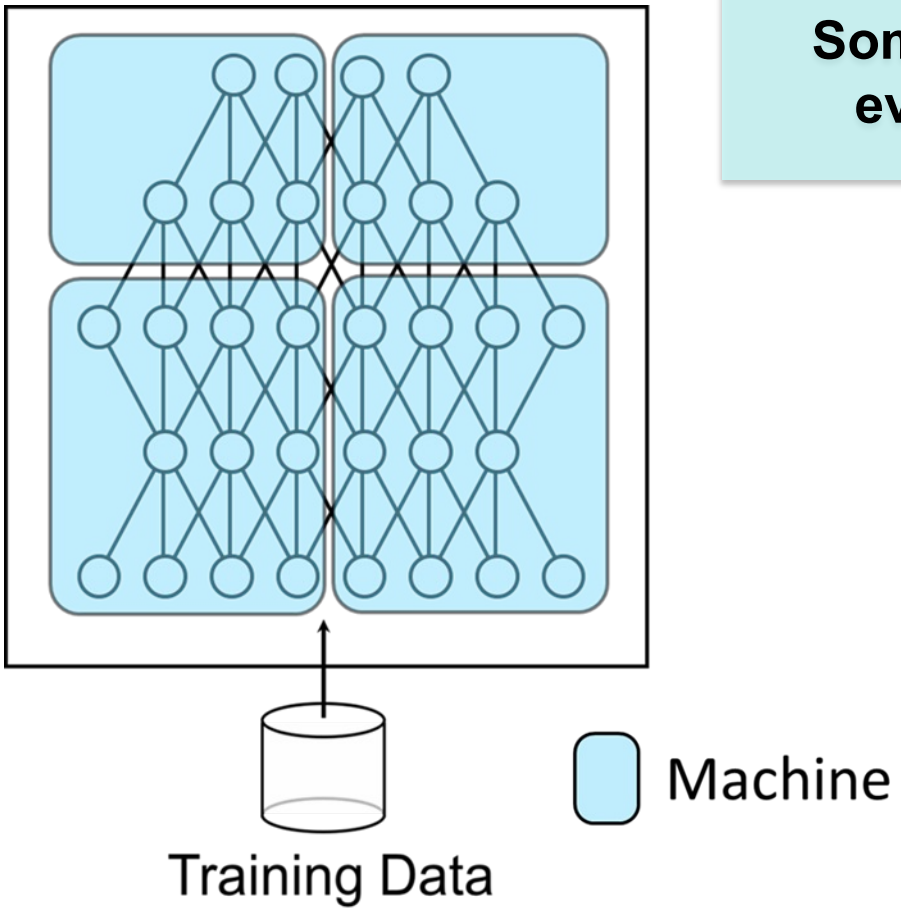




Points closer to the origin are preferable in that they take less time while using fewer resources.
Downpour SGD with Adagrad appears to be the best trade-off

"Large Scale Distributed Deep Networks" [Dean et al., NIPS 2012]

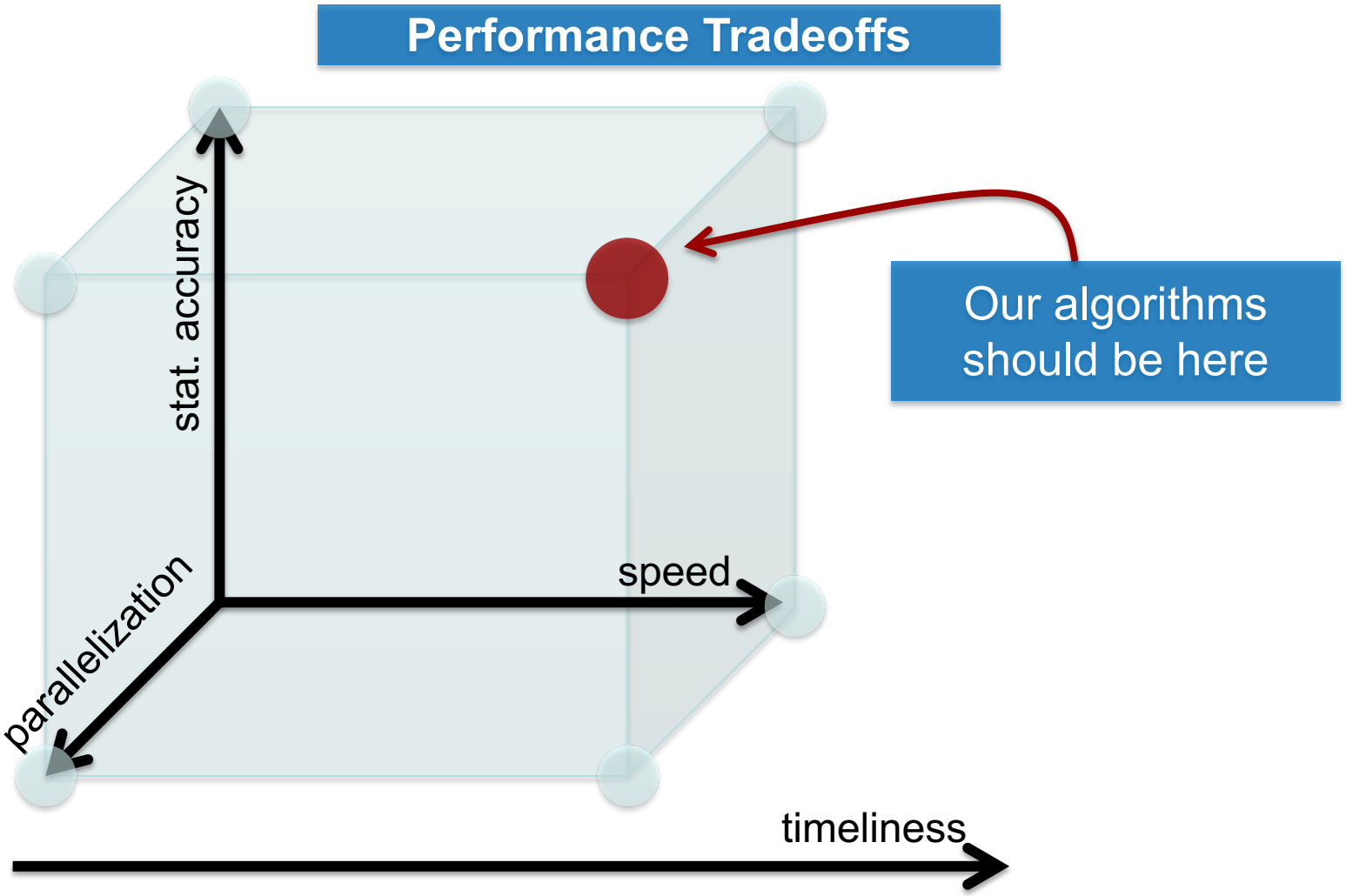
Choosing the Right Model



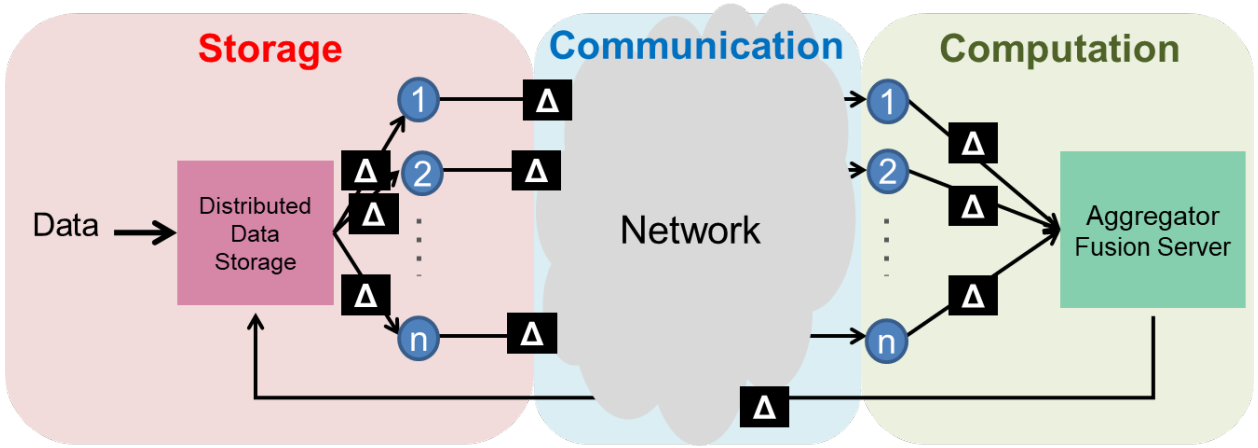
Some models are better than others, even from a systems perspective

- Does it fit in a single machine?
- Is model architecture amenable to low communication?
- Some models easier to partition
- Can we increase sparsity (less communication) without losing with accuracy?

The 3+1 Dimensions of ML Performance



Distributed Learning System Model



- Massively large distributed systems with individually small and unreliable computational nodes
- "System noise" – anomalous system behavior and bottlenecks
- System noise = latency variability (hard to predict)

Bottlenecks

Computation time

- Batch optimization
- Coded: preprocessing, gradient coding
- Uncoded: partial computations

Communication load

- Quantization/compression
- Sparsification
- Local parameter updates
- Model compression /Pruning

Robust and Fast Edge Learning over Wireless Networks