CS 260D –

Large Scale ML

Lecture #3: Distributed SGD

Baharan Mirzasoleiman

UCLA Computer Science

Adapted from Dimitris Papailiopoulos

Advances and Challenges in Distributed Machine Learning

Adapted from Dimitris Papailiopoulos

Mostly Challenges

Overview

- Multicore vs. Distributed
- Algorithms of choice
- Open challenges with Performance Gains/Analysis
- Convergence/Generalization/Speedup/Delays?
- Straggler Nodes
- Adversarial attacks

Stochastic Gradient Descent

Different names and flavors

ML / Optimization / Statistics / EE

Perceptron
Incremental Gradient
Back Propagation (NNs)
Oja's iteration (PCA)
LMS Filter
…

Has been around for a while, for good reasons:

- Robust to noise
- Simple to implement
- Near-optimal learning performance *
- Small computational foot-print

Stochastic Gradient Descent

Months!

SGD can take ~~10+~~ days on large data sets

[DawnBench, 2017]

Goal:

Speed up Machine Learning

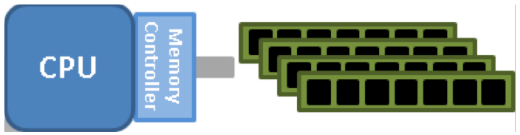
Idea:
Train at scale

Platforms

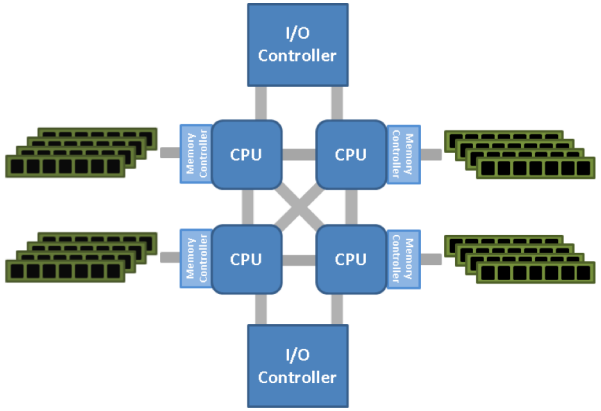
Parallel

Non uniform memory access

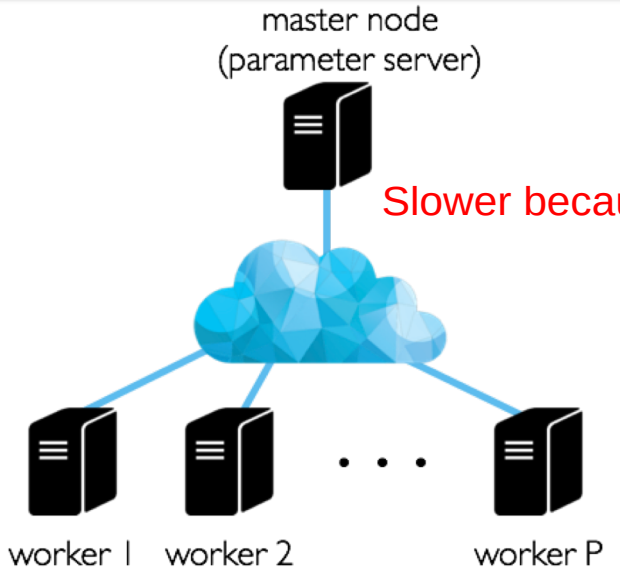
Single machine, multicore



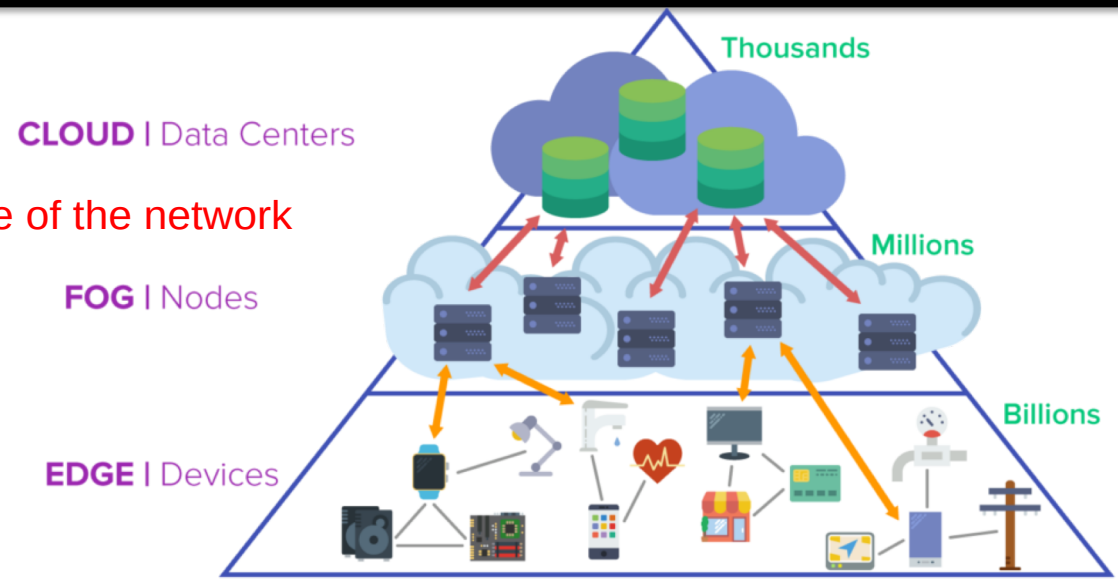
Multi-socket NUMA



Distributed



Slower because of the network



Parallel vs. Distributed

- Parallel (CPU)
  - Single machine, many cores (usually up to 100s)
  - Shared memory (all cores have access to RAM)
  - Comm. to RAM is cheap

Communication

- Distributed
  - Many machines (usually up to 1000s) connected via network
  - Shared-nothing architecture (each node has its own resources)
  - Communication costs non-negligible

Scaling up vs. Scaling out

Scaling up vs. out

What we'd ideally like

- Cores ∞
- RAM ∞
- Comm. 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 vs. out

Scaling up

Pros:

RAM comm. cheap (no network)

Less impl. overheads

Less power/smaller footprint

Cons:

100s cores/machine = expensive

If one fails, the system is dead

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

Choosing your Hardware

Price Comparisons for 4 GPUs racks

- Single machine multi-core (CPU)

  For 4 Tesla ~~V100~~ A100 GPU work station:

  more than $60,000

- Alternative Choice: Rent instances

Google Cloud Platform

Microsoft
Azure

EC2 Price for Scaling-up to 128 cores

- Alternative Choice: Rent Instances

Deciding on HW is a lot of work

Instance Size	GPUs - Tesla V100	GPU Peer to Peer	GPU Memory (GB)	vCPUs	Memory (GB)	Network Bandwidth	EBS Bandwidth	On-Demand Price/hr*	1-yr Reserved Instance Effective Hourly*	3-yr Reserved Instance Effective Hourly*
p3.8xlarge	4	NVLink	64	32	244	10 Gbps	7 Gbps	$12.24	$7.96	$4.93
p3.16xlarge	8	NVLink	128	64	488	25 Gbps	14 Gbps	$24.48	$15.91	$9.87

- Price for 4 GPUs: $8/hr

  10 days nonstop/month, **for a year** ~ $23,000

Let's Think about Running Stuff

How to Parallelize

$$w_{k+1} = w_k - \gamma \nabla \ell_{s_k}(w_k; x_i)$$

- Parallelize computation of **one update**?

  Issue:

  computing $\nabla \ell_i(w; x_i)$ cheap (even for deep nets) [O(d)]

- Parallel Updates?

  Issue: SGD is inherently serial….

所谓串行，指的是一个mini-batch完成后，再进行下一个mini-batch

内在的

Q: Can we parallelize inherently serial algorithms?

Simple idea

- Compute multiple gradients in parallel

K应该是epoch的计数

$$w_{k+1} = w_k - \gamma \nabla \ell_{s_k^1}(w_k; x_i)$$

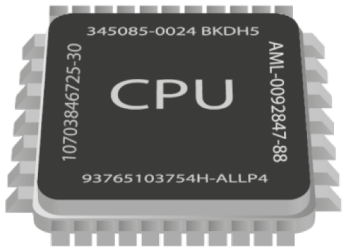
$$w_{k+1} = w_k - \gamma \nabla \ell_{s_k^2}(w_k; x_i)$$

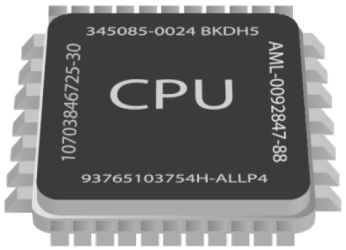
$$w_{k+1} = w_k - \gamma \nabla \ell_{s_k^3}(w_k; x_i)$$

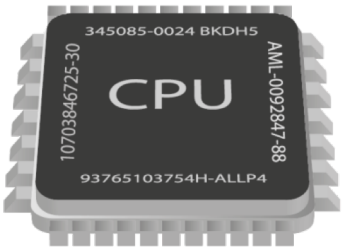
$$w_{k+1} = w_k - \gamma \nabla \ell_{s_k^4}(w_k; x_i)$$

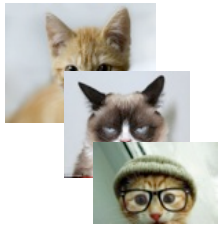
Q: Does it perform the same as SGD?

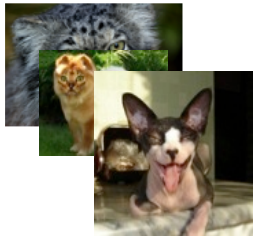
Distribute the effort!











每个机器上都有一个小模型

Several issues

Theory

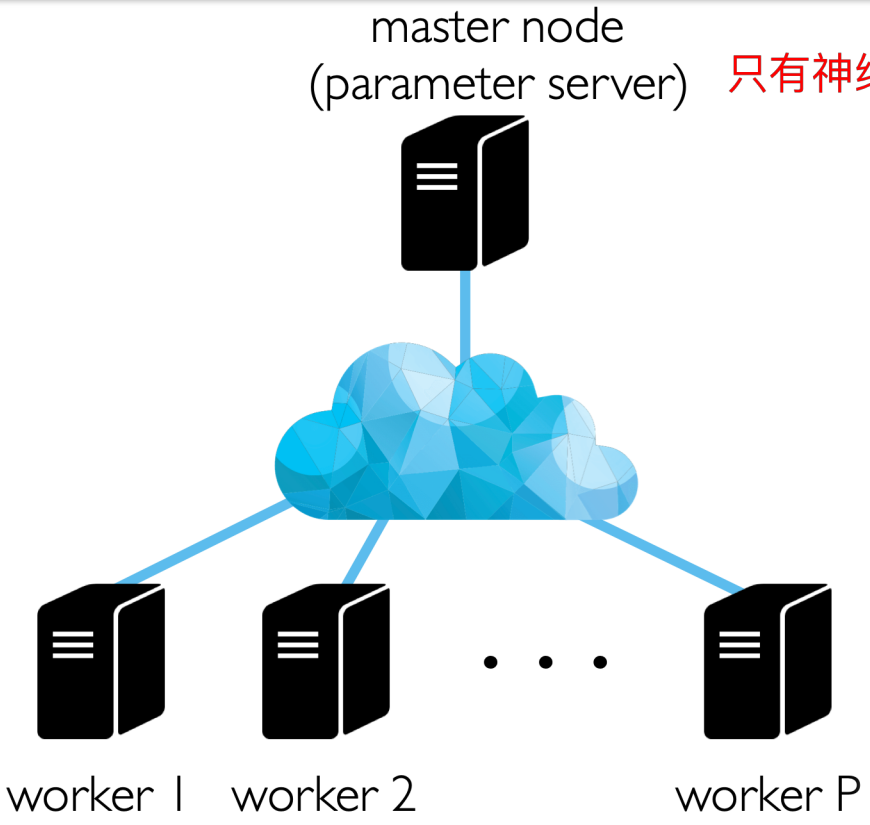
- Many models weaker than one
- Delays and Slow Nodes
- Communication Costs

Practice

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

Choosing an Algorithm

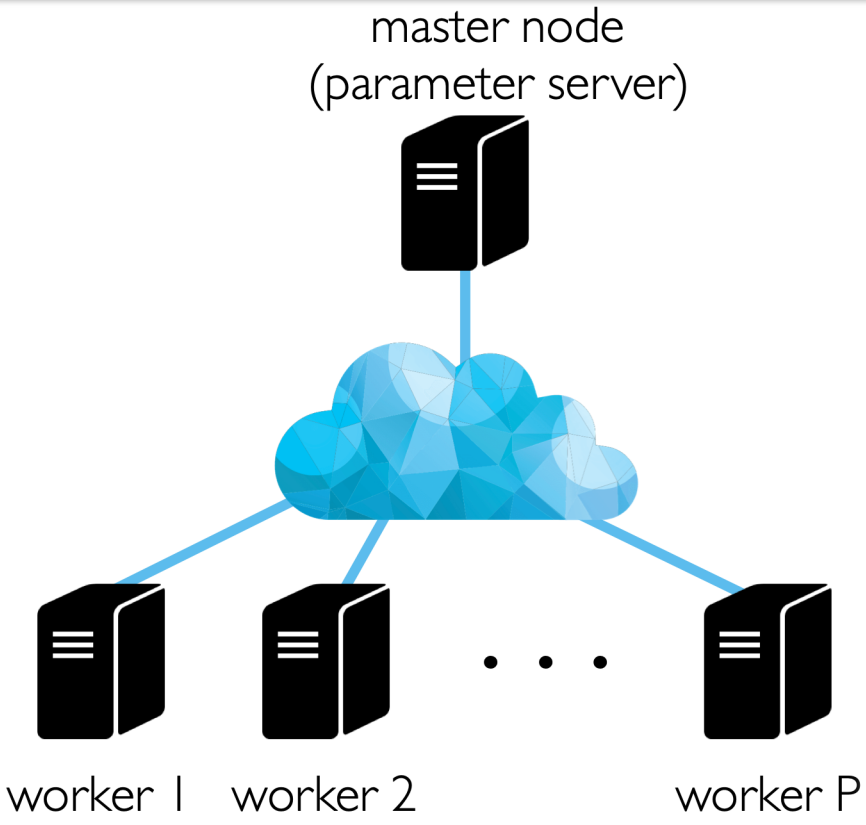
The Master-worker Setting



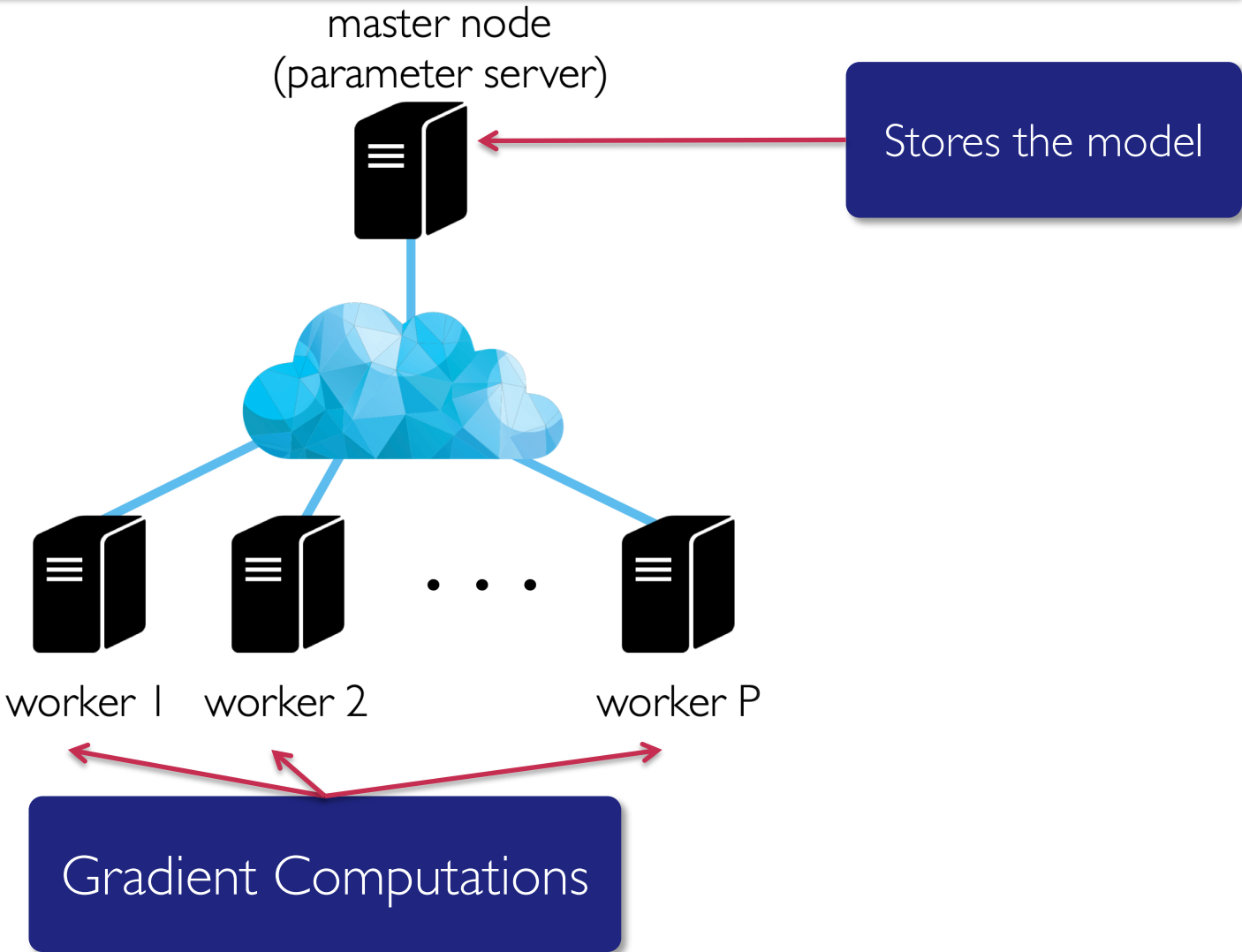
只有神经网络

各自都有数据，在自己的数据中做mini batch，然后把权重发回master

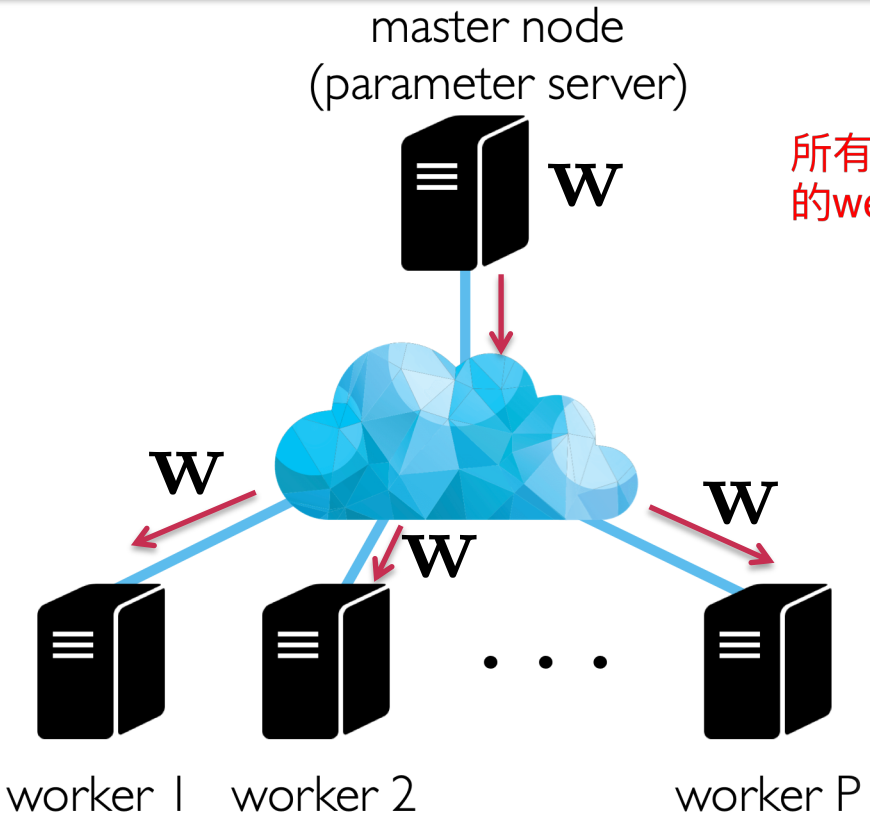
Algorithm of choice: minibatch SGD



Algorithm of choice: minibatch SGD

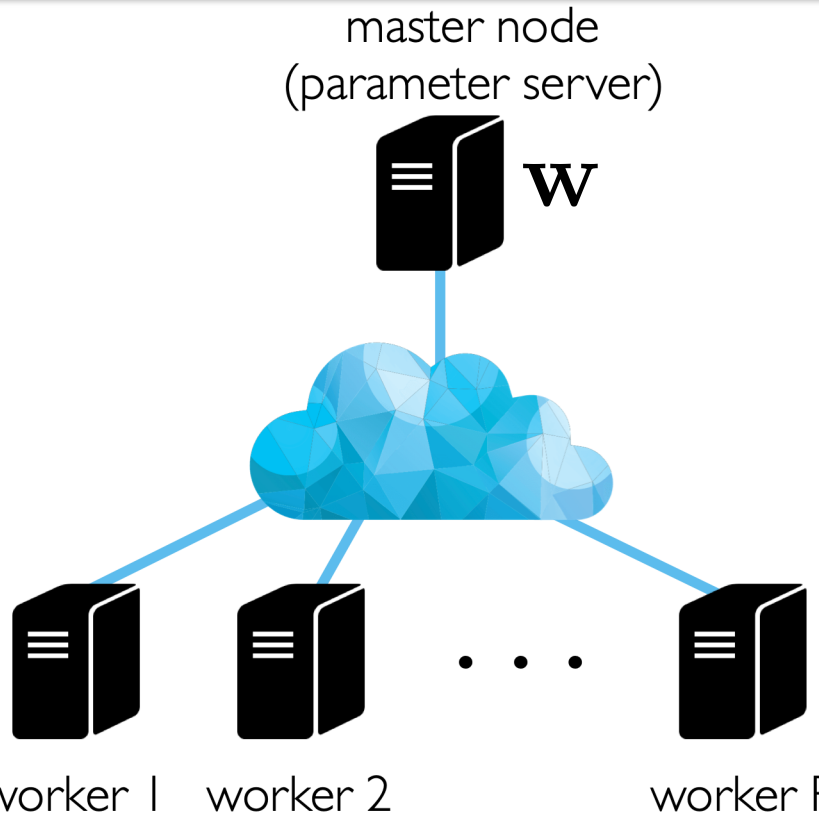


Algorithm of choice: minibatch SGD



所有worker使用的是一开始发来的weight

Algorithm of choice: minibatch SGD

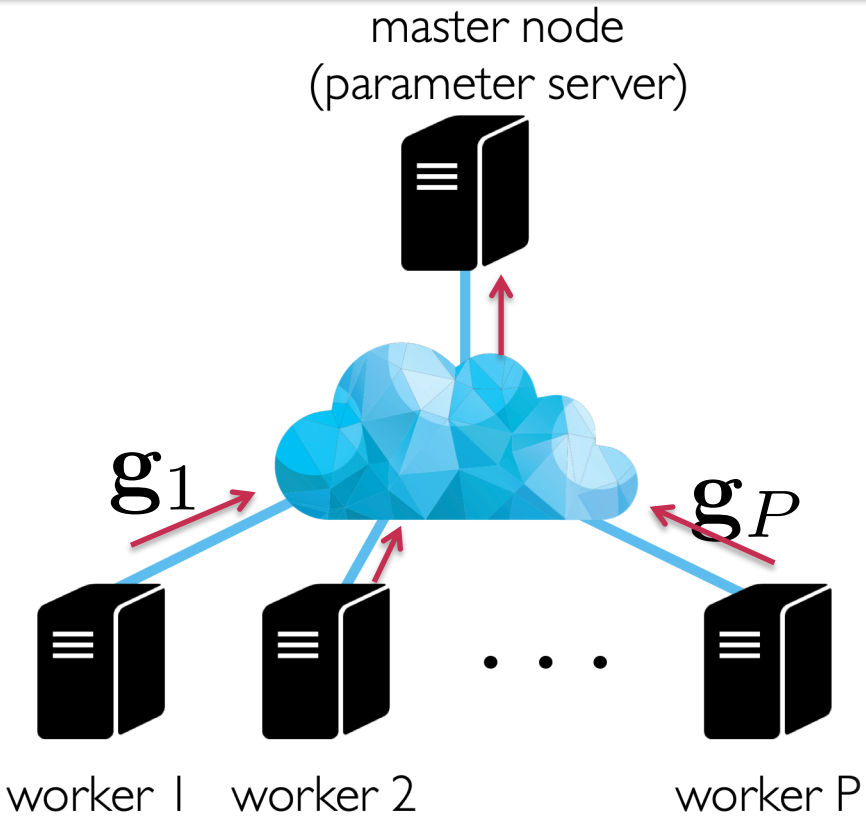


$$\mathbf{g}_1 = \sum_{i \in \mathcal{S}_1} \nabla \ell((\mathbf{x}_i, y_i); \mathbf{w})$$

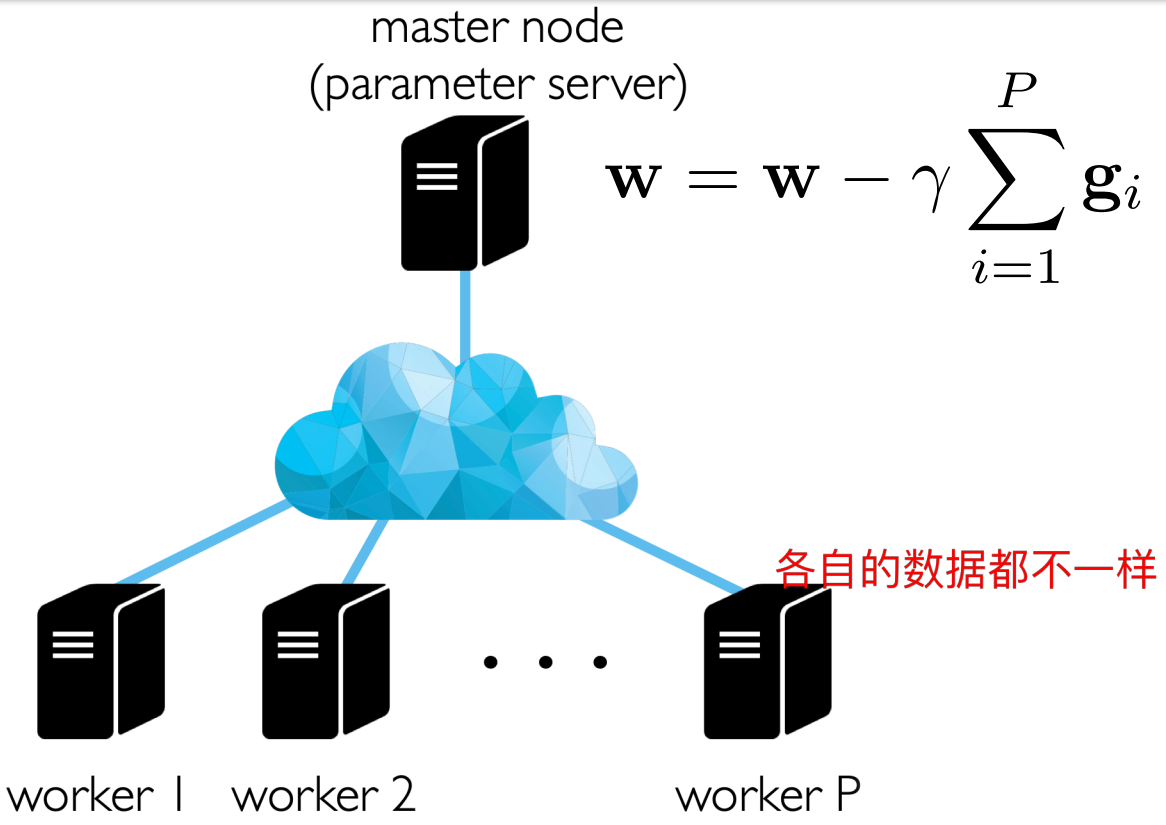
$$\mathbf{g}_P = \sum_{i \in \mathcal{S}_P} \nabla \ell((\mathbf{x}_i, y_i); \mathbf{w})$$

Si是一个worker中所有的数据

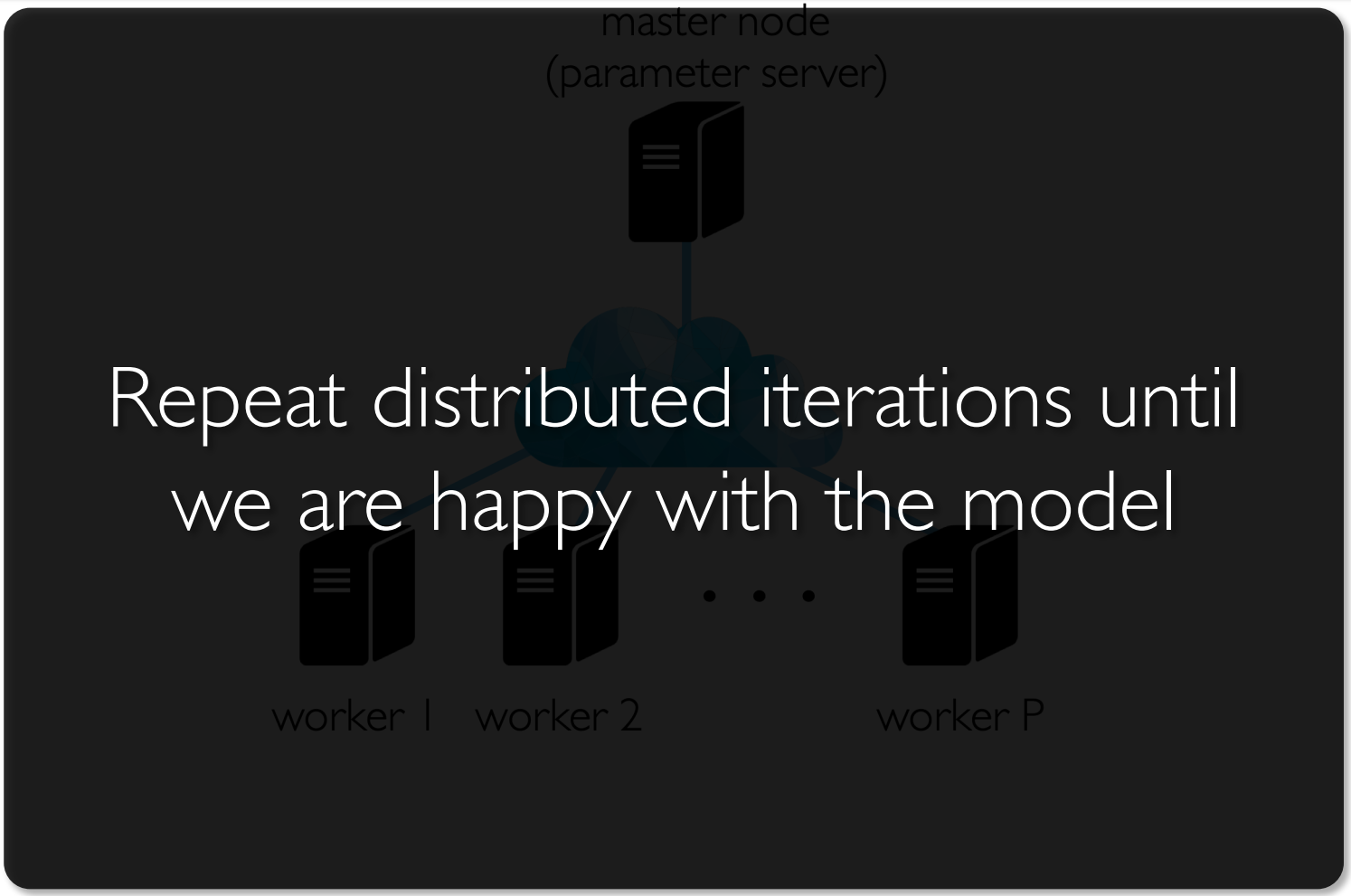
Algorithm of choice: minibatch SGD



Algorithm of choice: minibatch SGD



Algorithm of choice: minibatch SGD

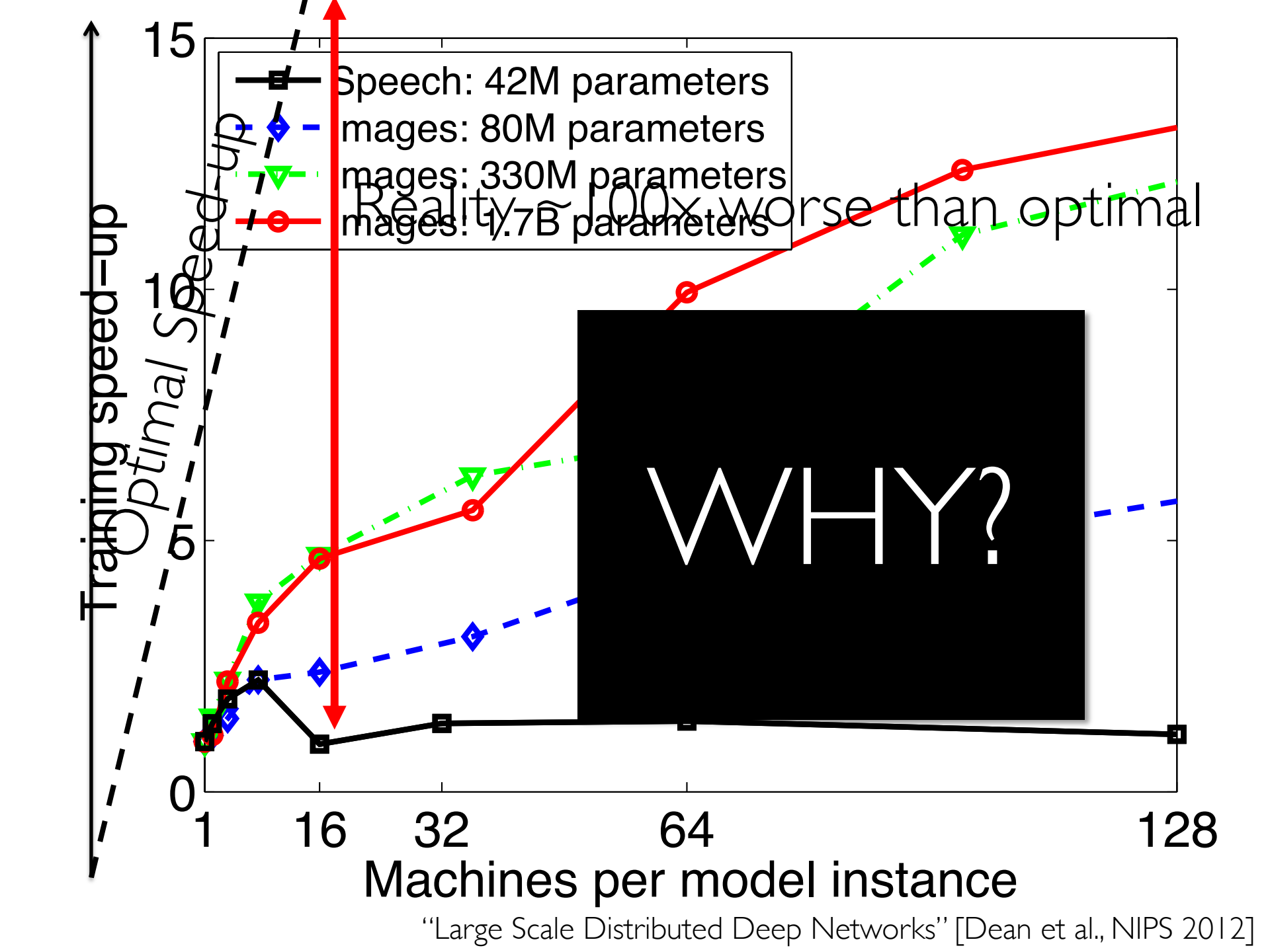


Evaluating the performance of Distributed ML

Many Questions….

- How fast does minibatch-SGD converge?
- How can we measure speedups?
- Comm. is expensive, how often do we average?
- How do we choose the right model?
- What happens with delayed nodes?
- Does fault tolerance matter?

Average指的是在master node上进行合并权重



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

Why so slow?

Two factors control run-time

Time to accuracy ε =

[time per data pass] X [#passes to accuracy ε]

Time cost of each epoch

Epoch numbers

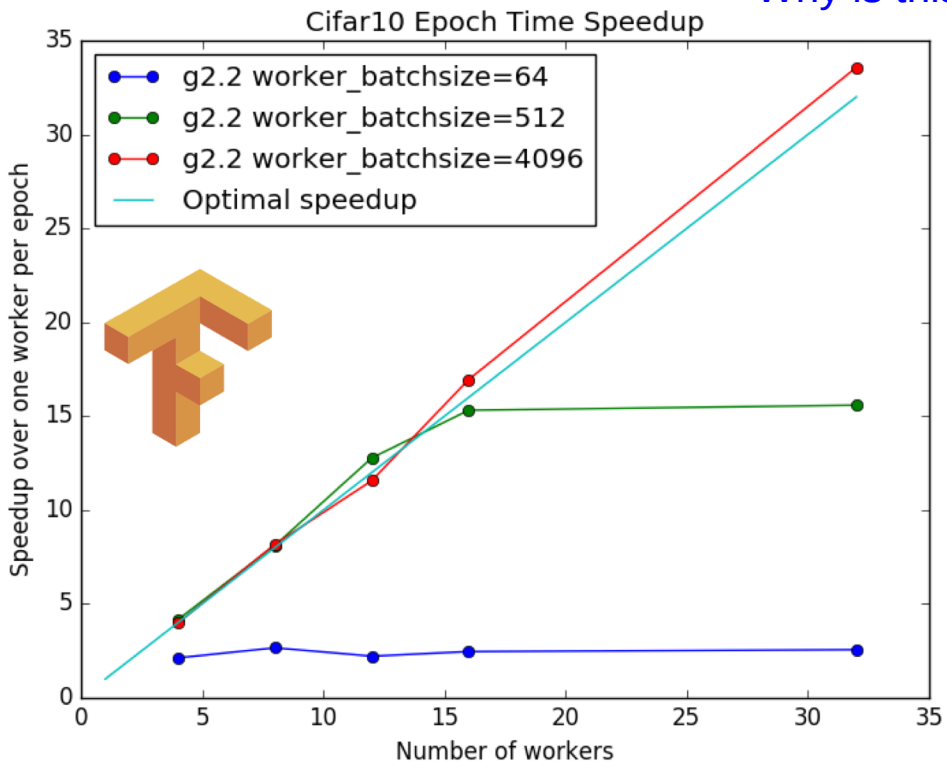
Why so slow?

"[…] on more than 8 machines […]
network overhead starts to dominate […]"

- TL;DR: Communication is the Bottleneck
- Why?

Larger batch size faster but more expensive on communication

Why is this better than optimum



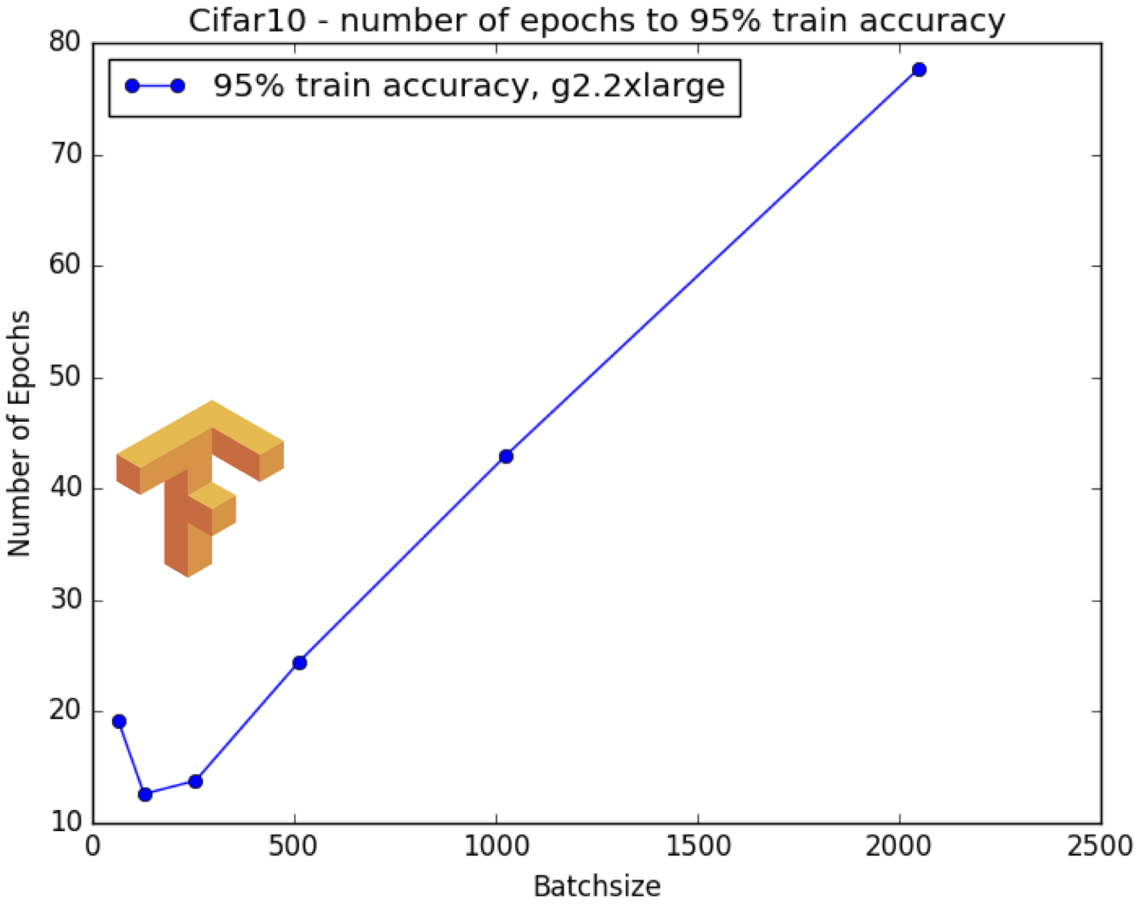
More worker, less batch numbers

Time per pass:
time for dataset_size/batch_size
distributed iterations

Bigger Batch
⇒ Less Communication
(smaller time per epoch)

What's wrong with Large Batches?

- If small batch is bad, then maximize it



Large Batch
⇒ worse train error
(more #passes to accuracy **ε**)

Large Batch
=> worse generalization

[Keskar, Mudigere, Nocedal, Smelyanskiy, Tang, ICLR 2017]

How to analyze parallel algorithms?

- Main measure of performance

$$\text{speedup} = \frac{\text{Time of serial } \mathcal{A} \text{ to accuraccy } \epsilon}{\text{Time of parallel } \mathcal{A} \text{ to accuraccy } \epsilon}$$

Example: Gradient Descent

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

Embarrassingly Parallel O(#cores) speedup

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 take same steps after T iterations
- Speedup is independent of covergence rate

Not true for mini-batch SGD

How to Analyze mini-batch?

- Measure of performance

Epsilon is the accuracy goal

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

Main Question:
How does minibatch SGD compare against serial SGD?

Main questions:
Convergence after T gradient computations

Answer is Complicated: Depends on Problem
Generally if batch B_0 > B(data, loss)
Minibatch SGD offers no speedup.

Diversity related

Generalization?

Do models trained with minibatch SGD generalize

You can't train your neural network on too large batch size

Batch	Top-1 Acc	Top-5 Acc
256	58.42%	81.51%
512	59.19%	81.84%
1024	59.00%	81.94%
2048	58.88%	81.73%
4096	57.97%	81.00%
8192	55.90%	79.40%

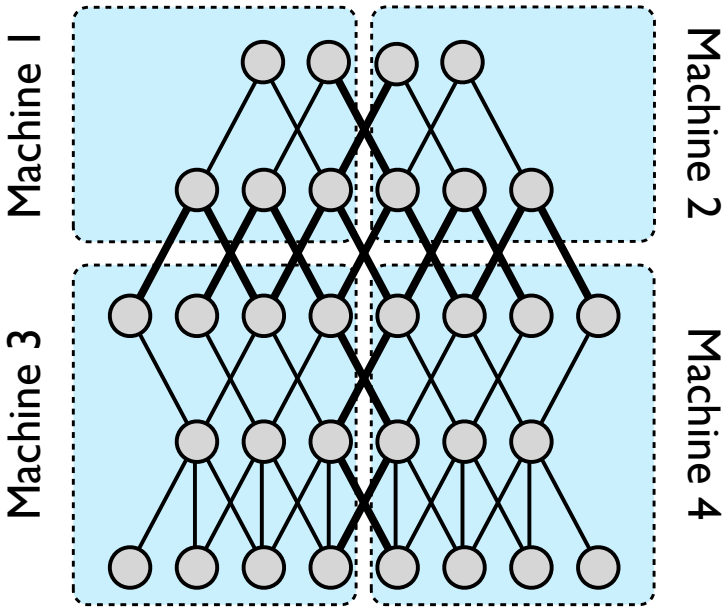
Alexnet on Imagenet

How to choose the right model

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

How to choose the right model

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



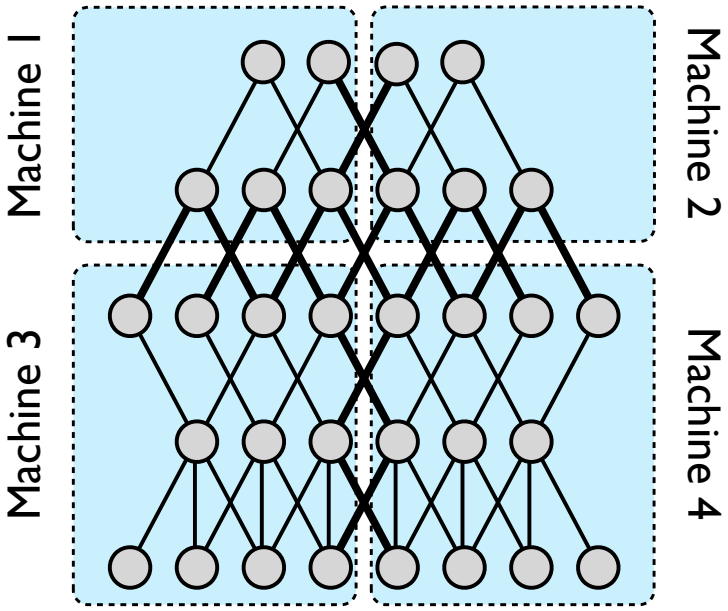
It's fully connected so it's hard to find a place to split

被切断的链接仍然存在，并没有消失（具体实现不清楚）

How to choose the right model

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

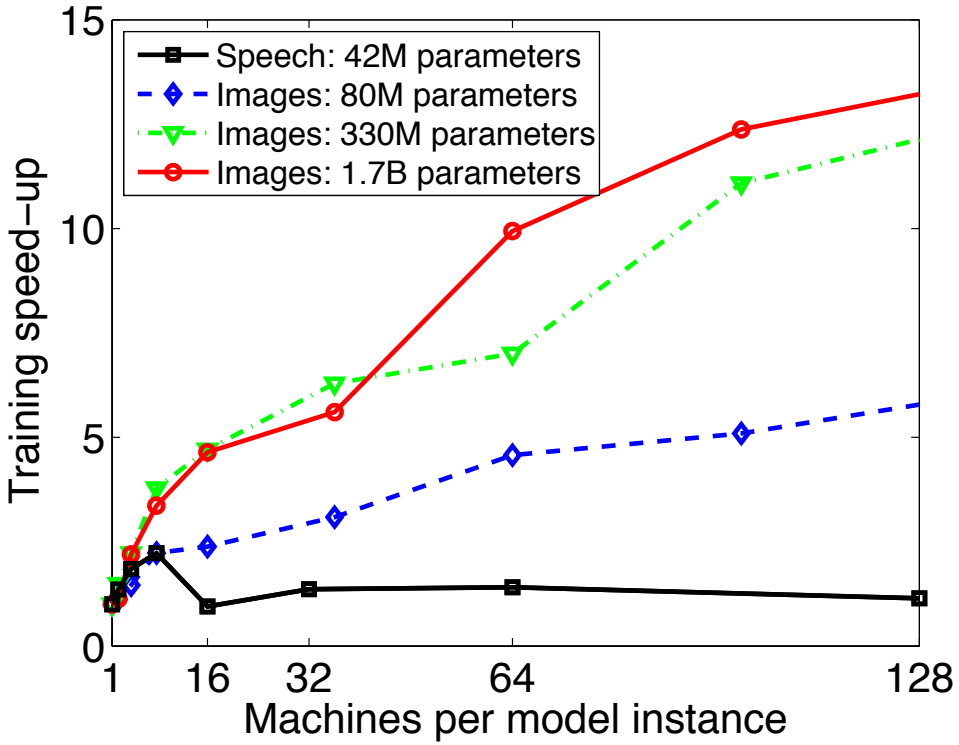
What do we do with the cutted connection



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

How to chose the right model

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



Parallelism Gains?

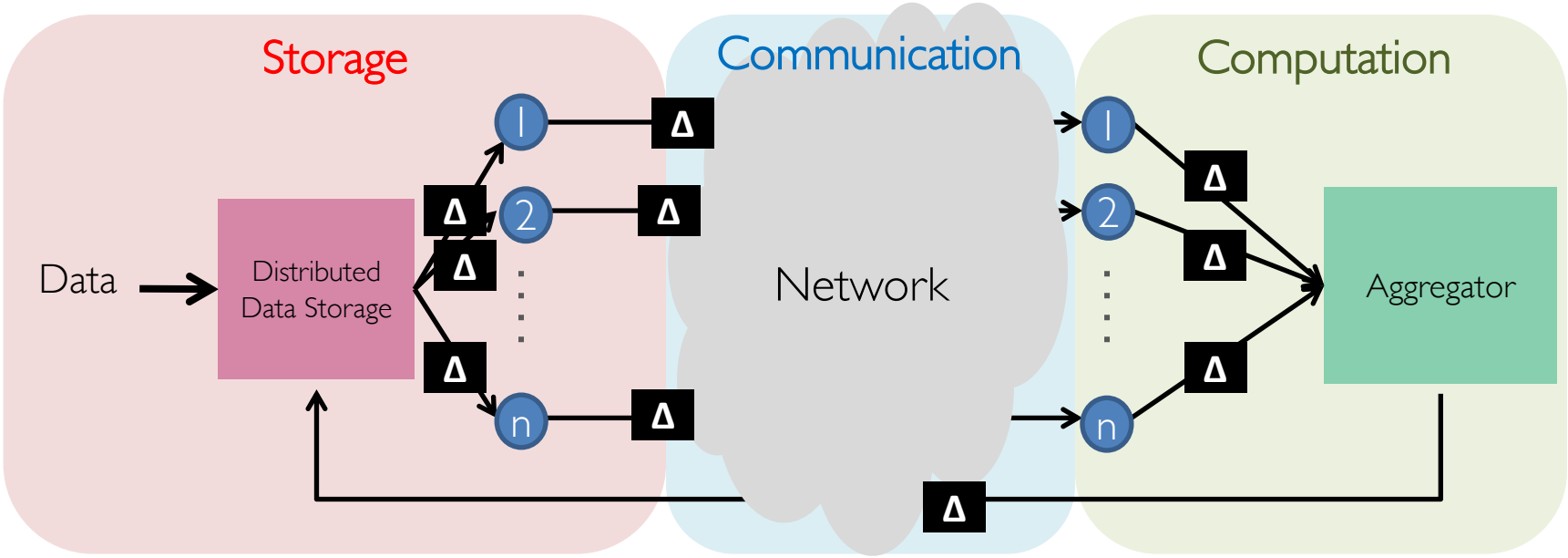
- Weak Scaling: Easy

- Strong Scaling: Nontrivial, and not there yet

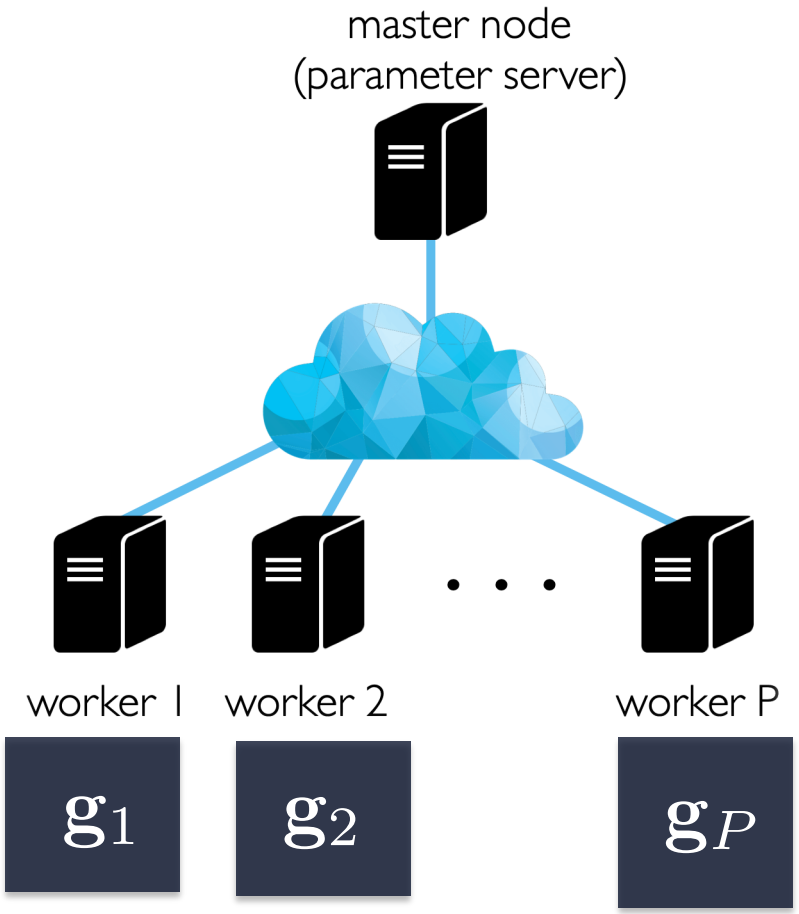
Major Open Problem!!!

Avoiding Communication Bottlenecks

Large-scale Distributed Machine Learning Systems



A better algorithm?



Communication / worker:

O(size of gradient)*32 bits

Q: Lower precision?

O(size of gradient)*4 bits

A: Yes!

降低精度，减少内存的 SGD

A lot of recent work on quantized SGD

QSGD: Communication-Efficient SGD via Gradient Quantization and Encoding

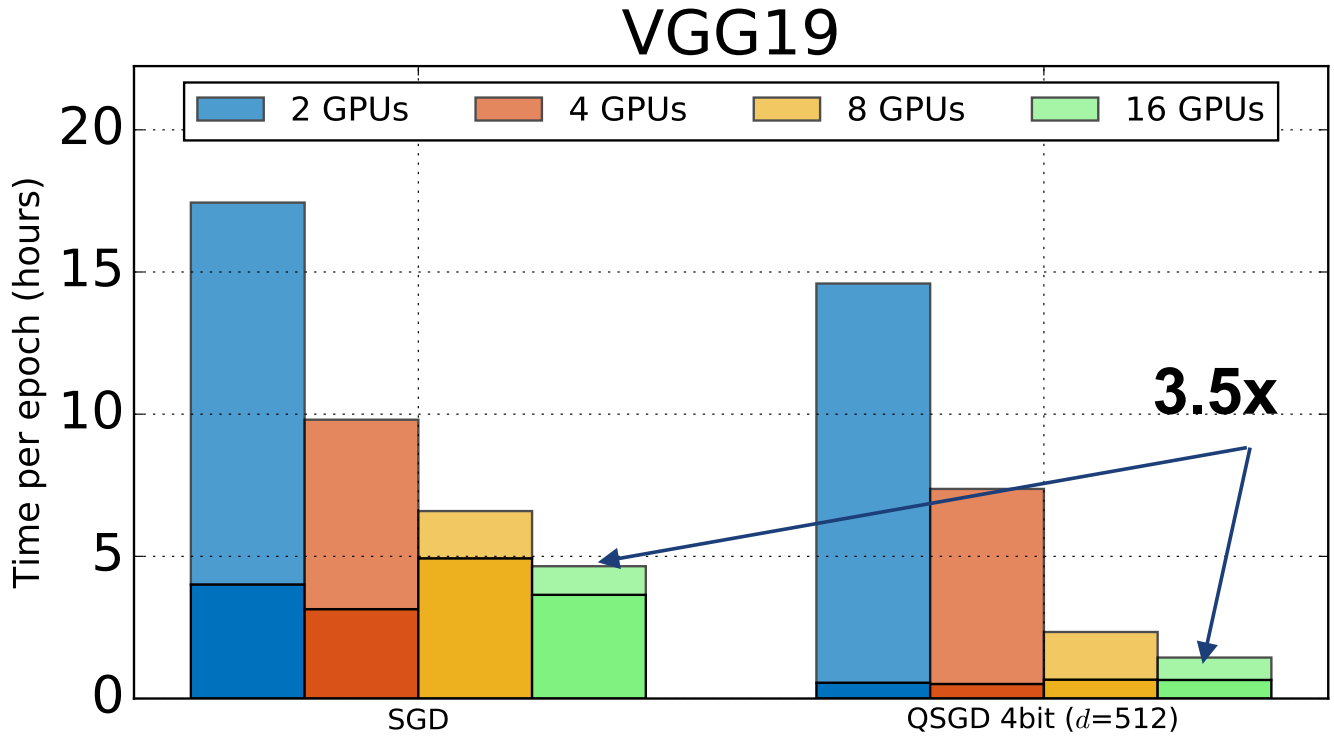
Dan Alistarh
IST Austria & ETH Zurich
dan.alistarh@ist.ac.at

Demjan Grubic
ETH Zurich & Google
demjangrubic@gmail.com

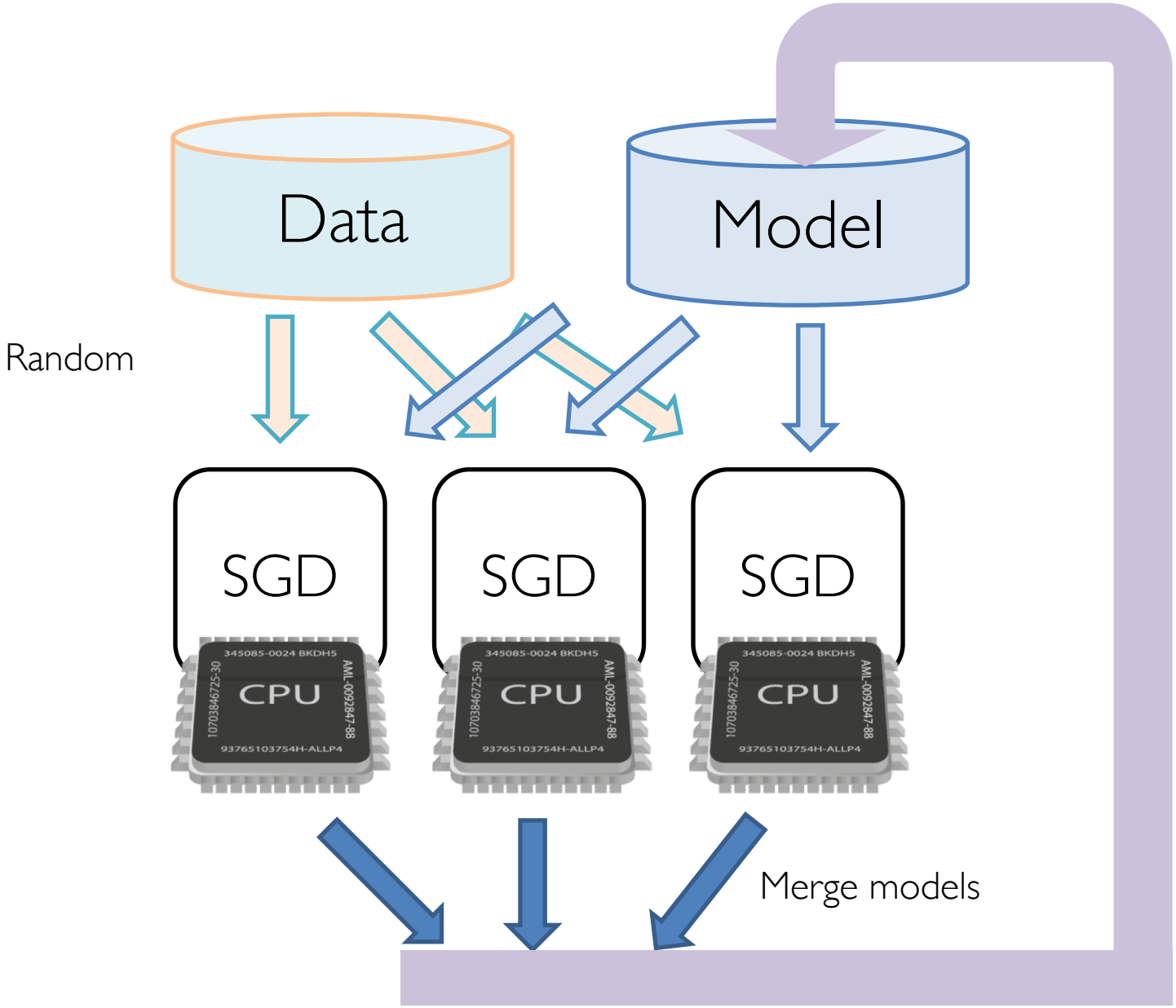
Jerry Z. Li
MIT
jerryzli@mit.edu

Ryota Tomioka
Microsoft Research
ryoto@microsoft.com

Milan Vojnovic
London School of Economics
M.Vojnovic@lse.ac.uk



Data Shuffling

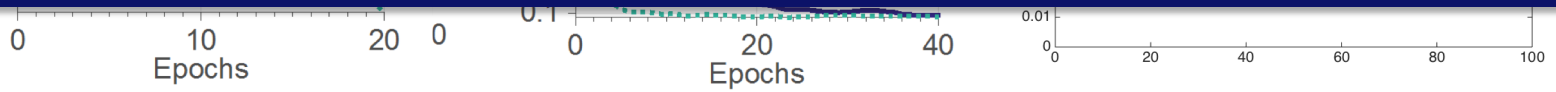


Parallel Stochastic Gradient Descent (PSGD)

Shuffle data in different machines cost a lot

Data shuffling

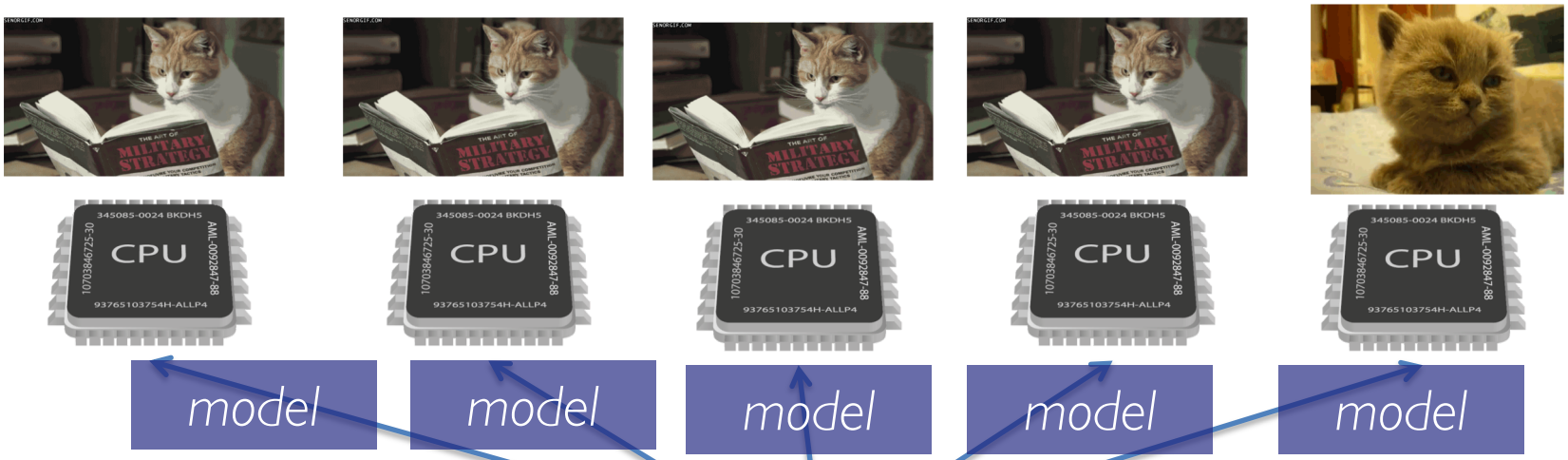
Very high communication cost: does shuffling still makes sense?



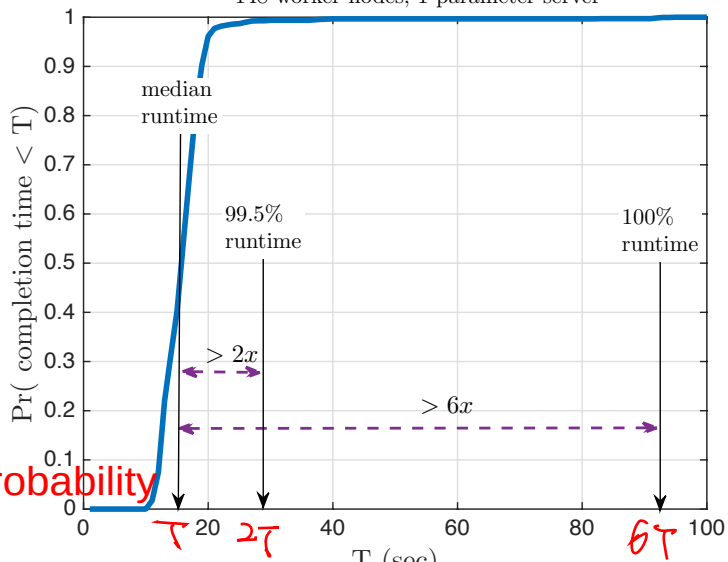
PSGD with shuffling converges faster

* [Recht and Re, 2013], [Bottou, 2012], [Zhang and Re, 2014], [Gurbuzbalaban et al., 2015], [Ioffe and Szegedy, 2015], [Zhang et al. 2015]

Bottleneck 2: Straggling Learners



Iteration Completion Time per worker
Data set = CIFAR-10
t2.small EC2 instances
148 worker nodes, 1 parameter server

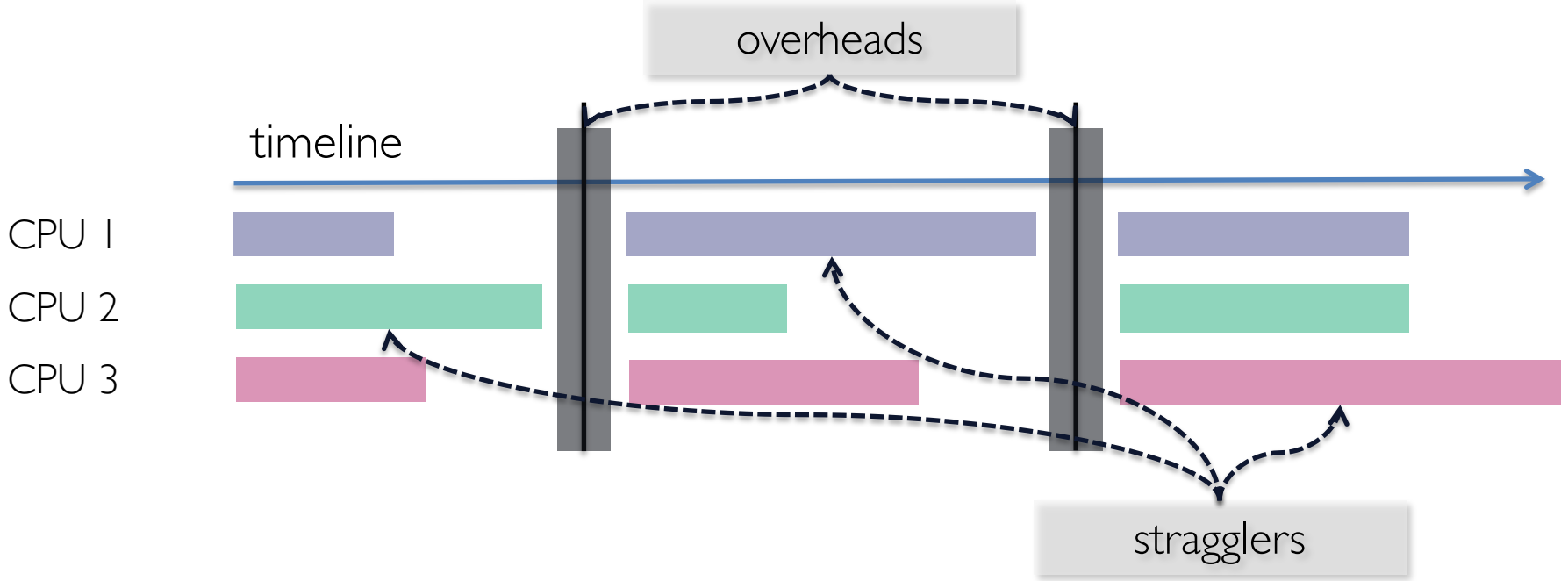


If you wait for all to finish, it takes a long time

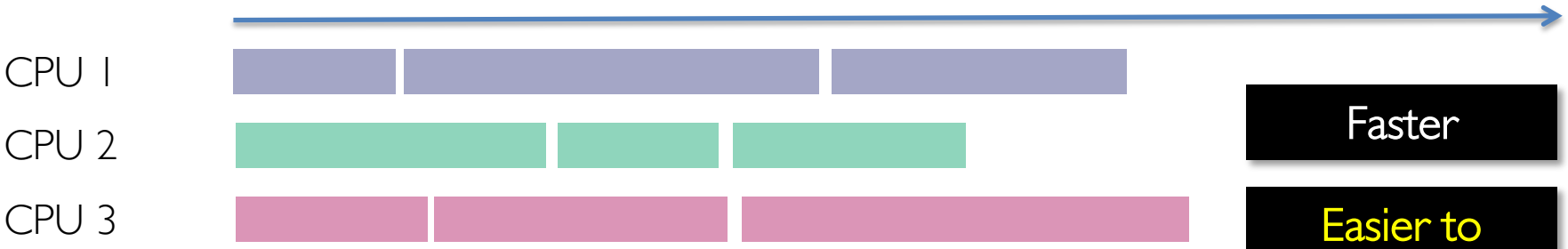
Can we "robustify" distributed ML against stragglers?

Measured on Amazon AWS

A case against Synchronization



Asynchronous World



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 = constant + Exp(λ)

Lemma:

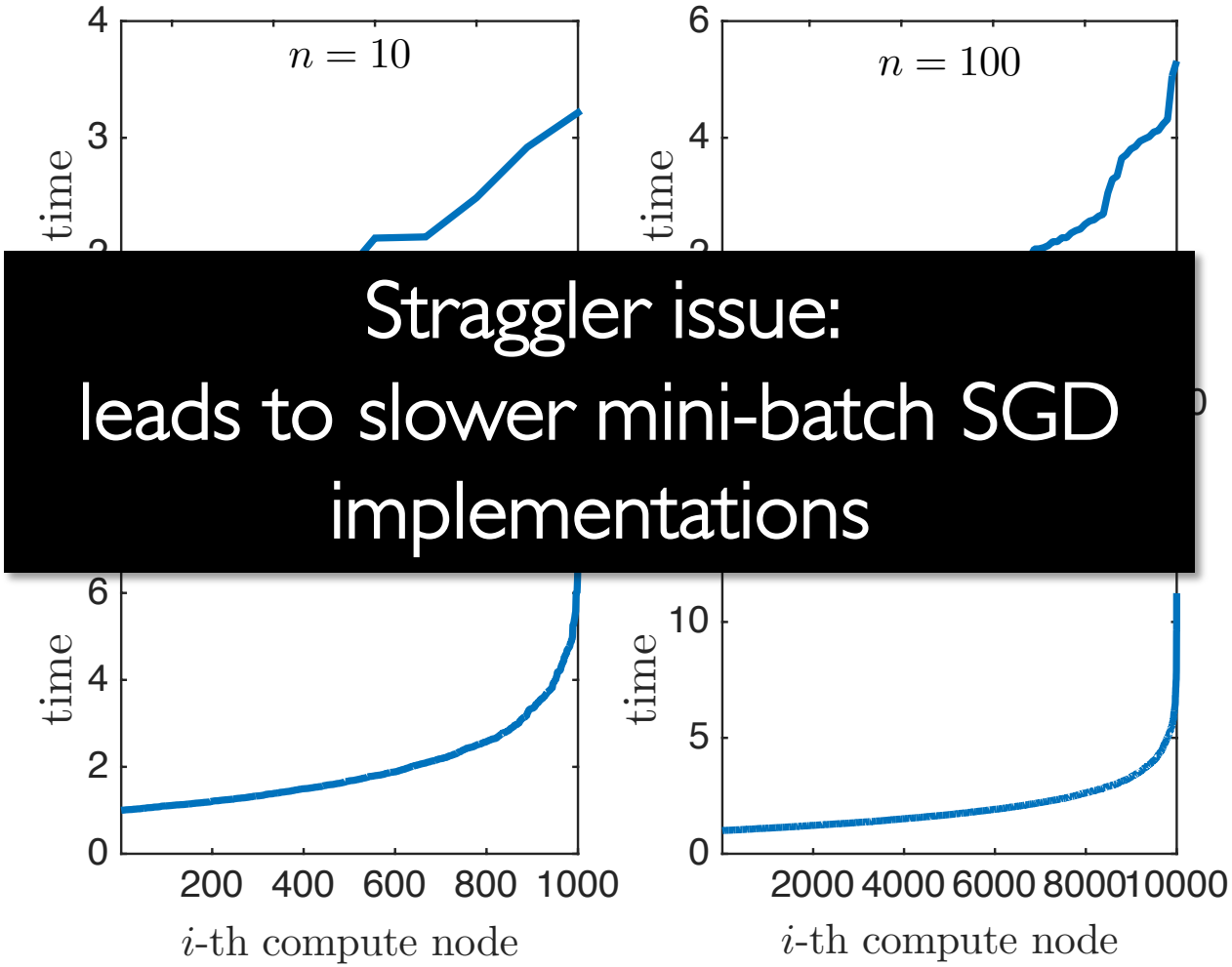
$$\mathbb{E}\{X_{(i)}\} = 1 + \frac{1}{\lambda} \sum_{n-i+1}^{n} \frac{1}{i}$$

Remark:

Slowest node is log(n) times slower than fastest

Simulation

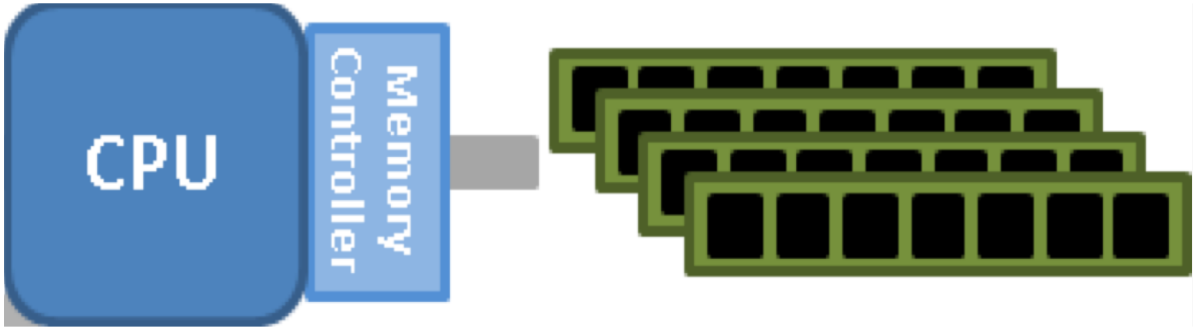
- X(t) = 1+ Exp(0.5), n = 10, 100, 1000, 1000



Straggler issue:
leads to slower mini-batch SGD implementations

Parallelizing Sparse SGD on shared memory architectures

Single Machine, Multi-core



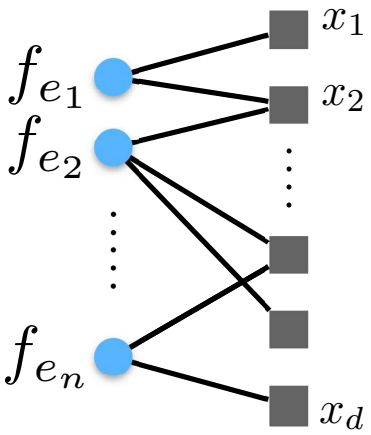
Asynchronous SGD on Sparse Functions

SGD on sparse functions

$$f(x) = \sum_{e \in \mathcal{E}} f_e(x_e)$$

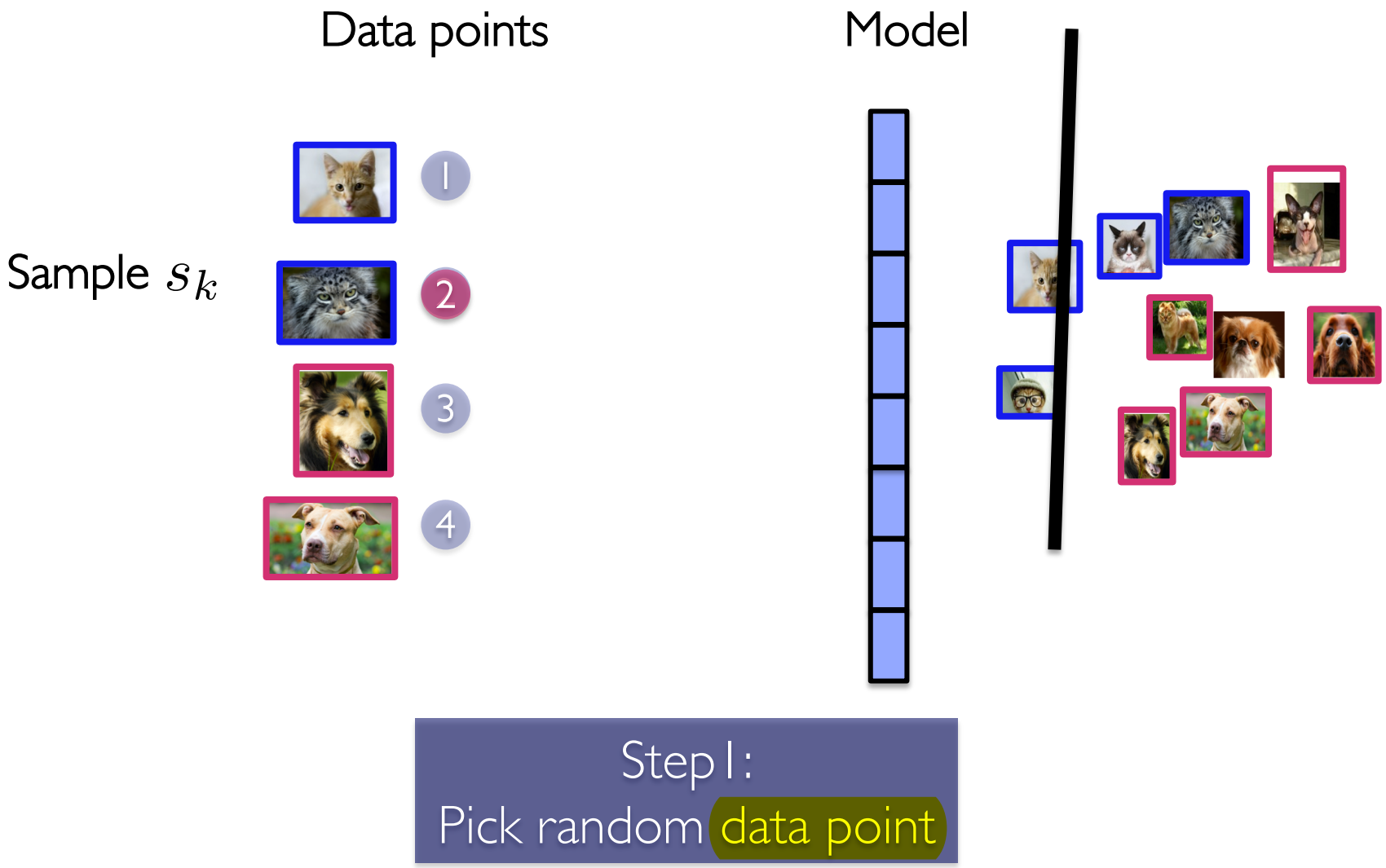
- Def:
  Hyperedge e = the subset of variables that f_e depends on

- The function-variable graph

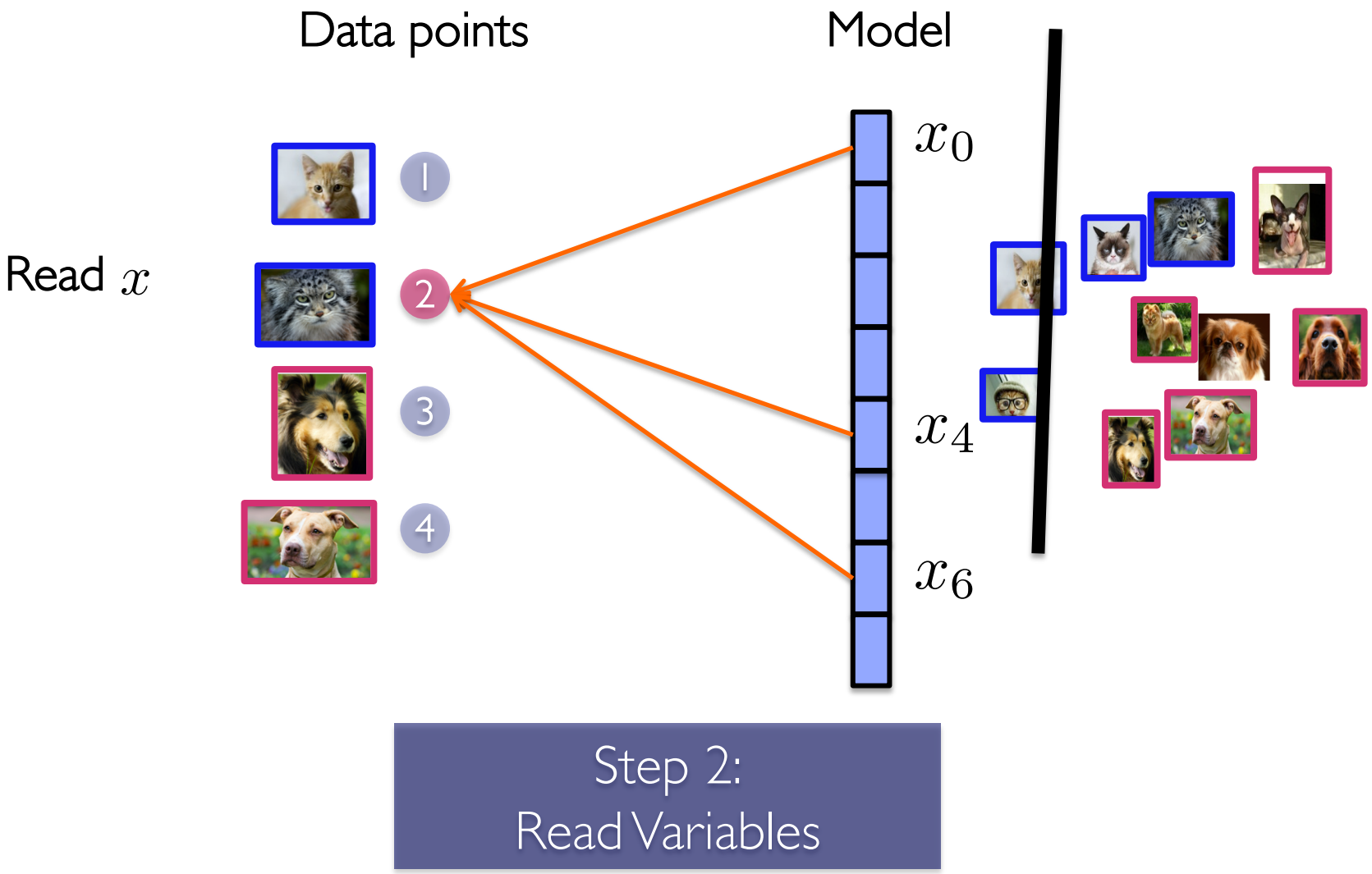


Matrix Fact./Comp.
Graph cuts
Graph/text Classification
Topic Modeling
Dropout
…

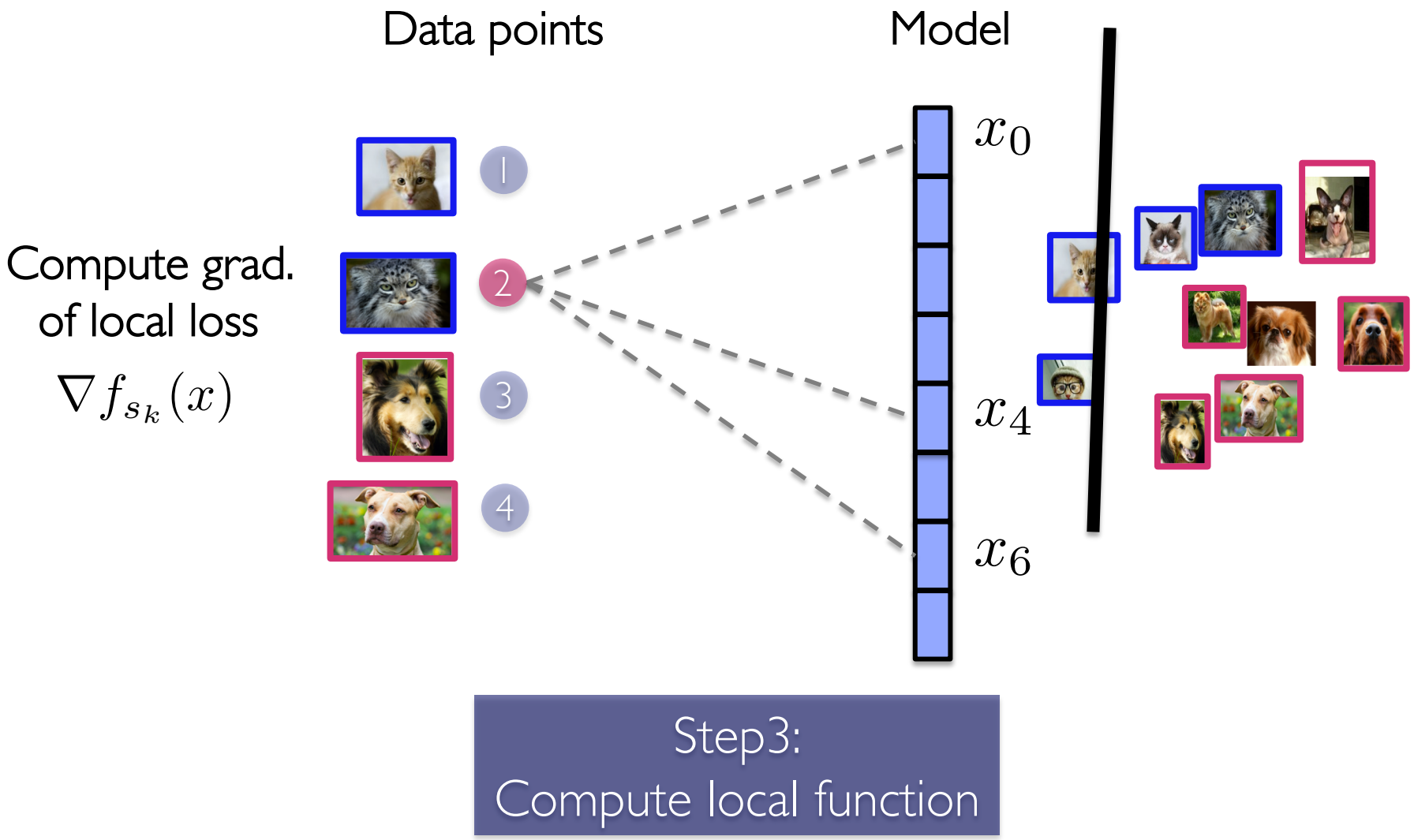
SGD on sparse functions



SGD on sparse functions



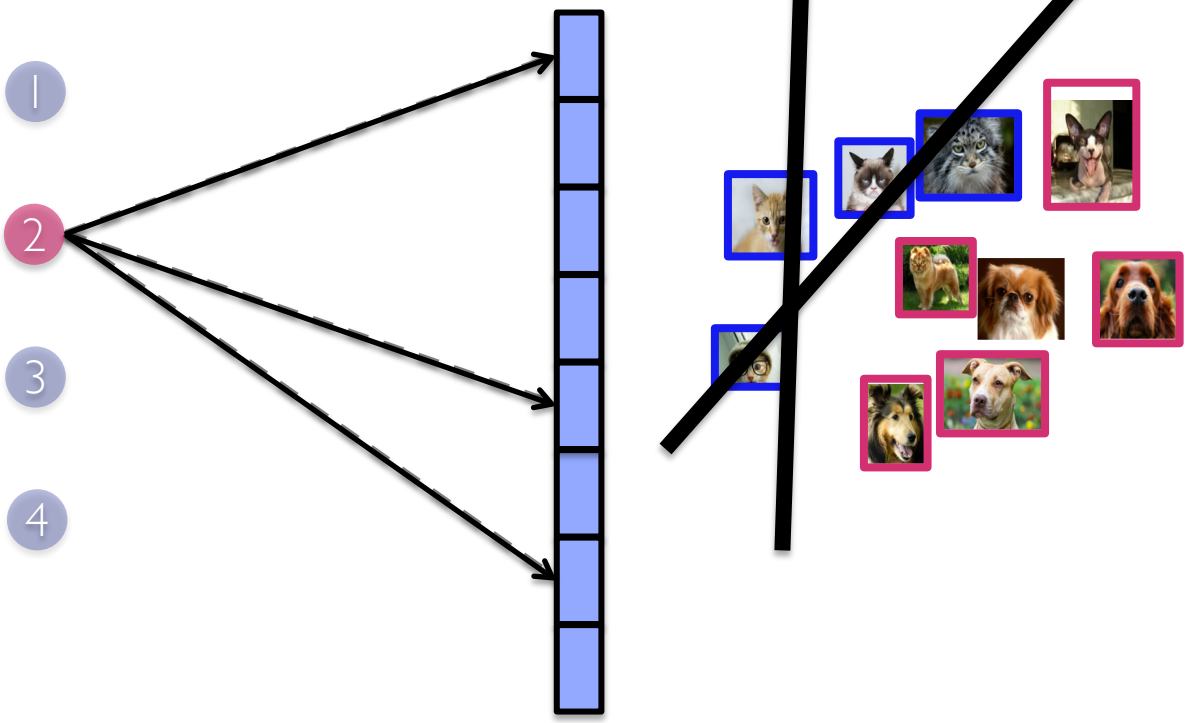
SGD on sparse functions



SGD on sparse functions

Data points

Model



Update Model

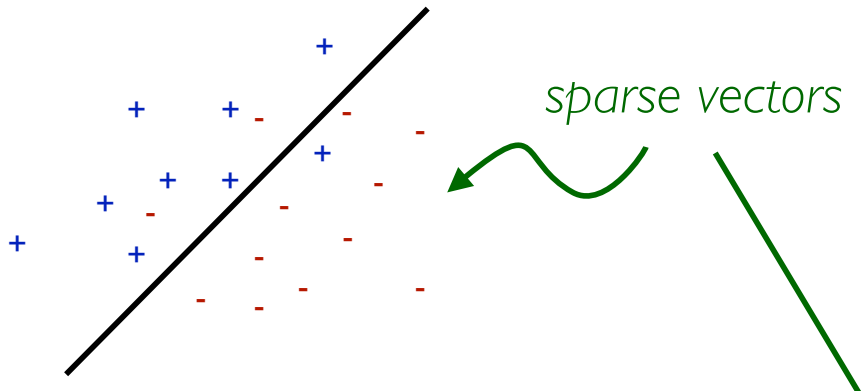
$$x \leftarrow x - \gamma_k \cdot \nabla f_{s_k}(x)$$

Only calculate the gradient of the variable in function

Step4:
Update Model

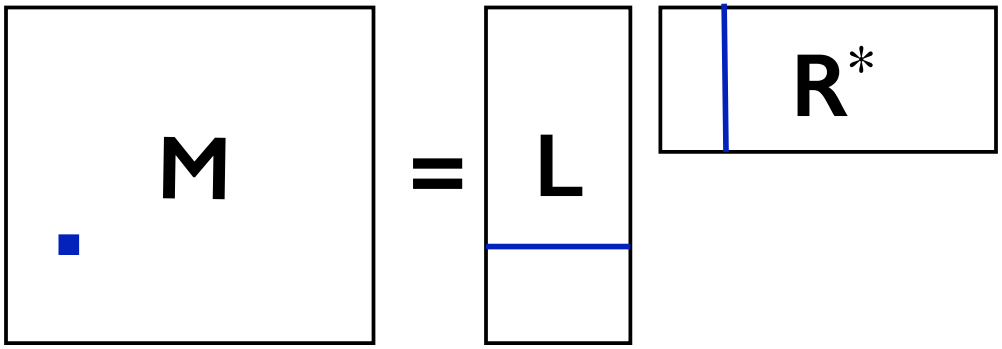
Examples of Sparse Problems

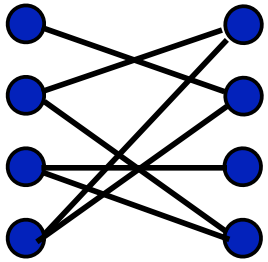
Sparse Support Vector Machines



$$\text{minimize}_x \sum_{\alpha \in E} \max(1 - y_\alpha x^T z_\alpha, 0) + \lambda \|x\|_2^2$$

Matrix Completion



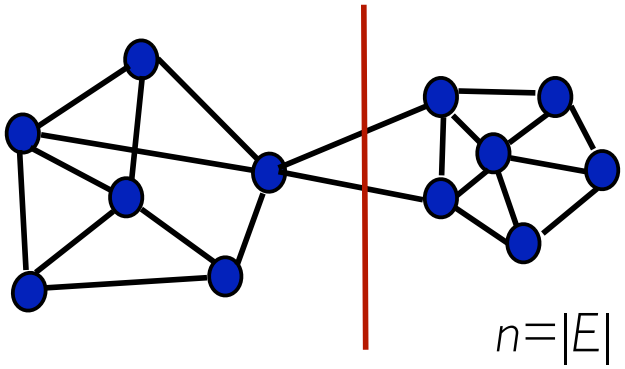


$d_1 \times d_2$ $(d_1 \leq d_2)$ $\quad d_2 \times r \quad r \times d_2$

Entries Specified on set E (with $|E|=n$)

$$\text{minimize}_{(\mathbf{L},\mathbf{R})} \sum_{(u,v)\in E} \left\{ (\mathbf{L}_u \mathbf{R}_v^T - M_{uv})^2 + \mu_u \|\mathbf{L}_u\|_F^2 + \mu_v \|\mathbf{R}_v\|_F^2 \right\}$$

Graph Cuts

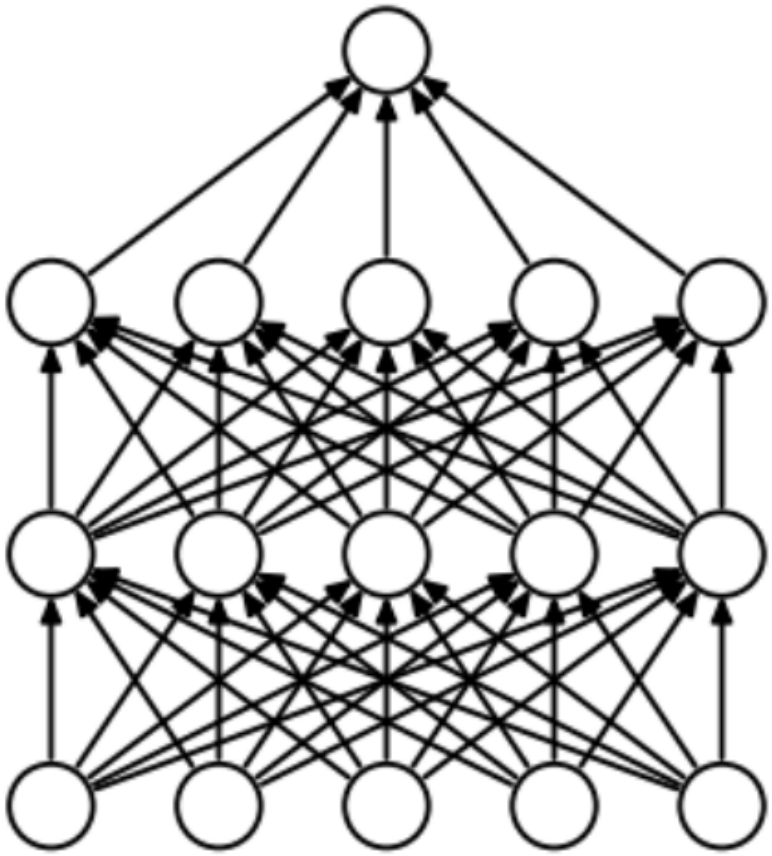


- Image Segmentation
- Entity Resolution
- Topic Modeling

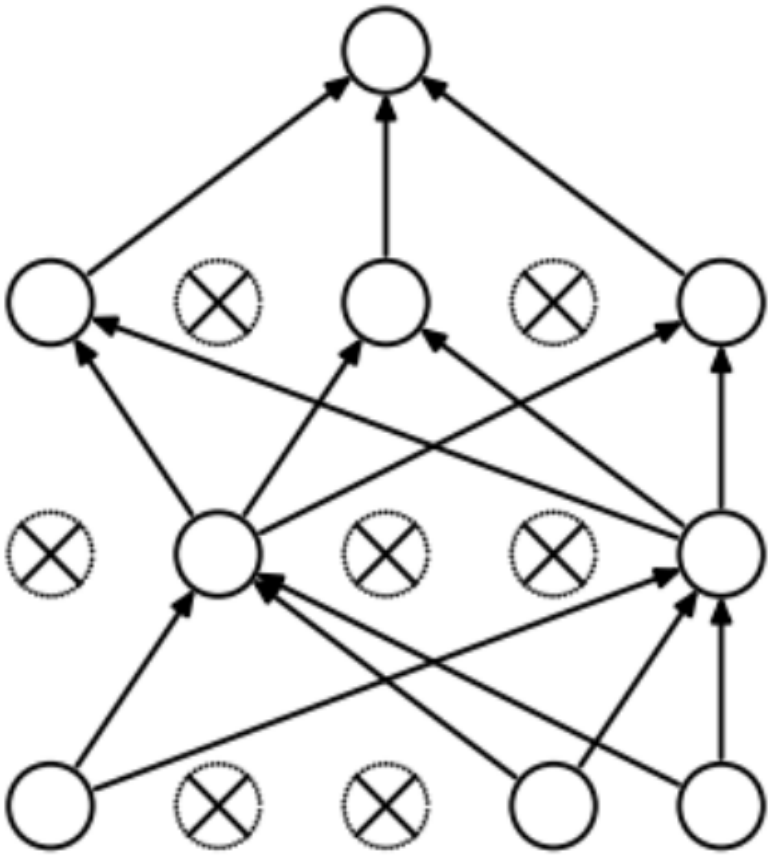
$$\begin{aligned} &\text{minimize}_x \quad \textstyle\sum_{(u,v)\in E} w_{uv}\|x_u - x_v\|_1 \\ &\text{subject to} \quad \mathbf{1}_K^T x_v = 1\,,\ x_v \geq 0\,, \quad \text{for } v = 1,\ldots,D \end{aligned}$$

Sparsified BackProp

Dropout: drop some nodes



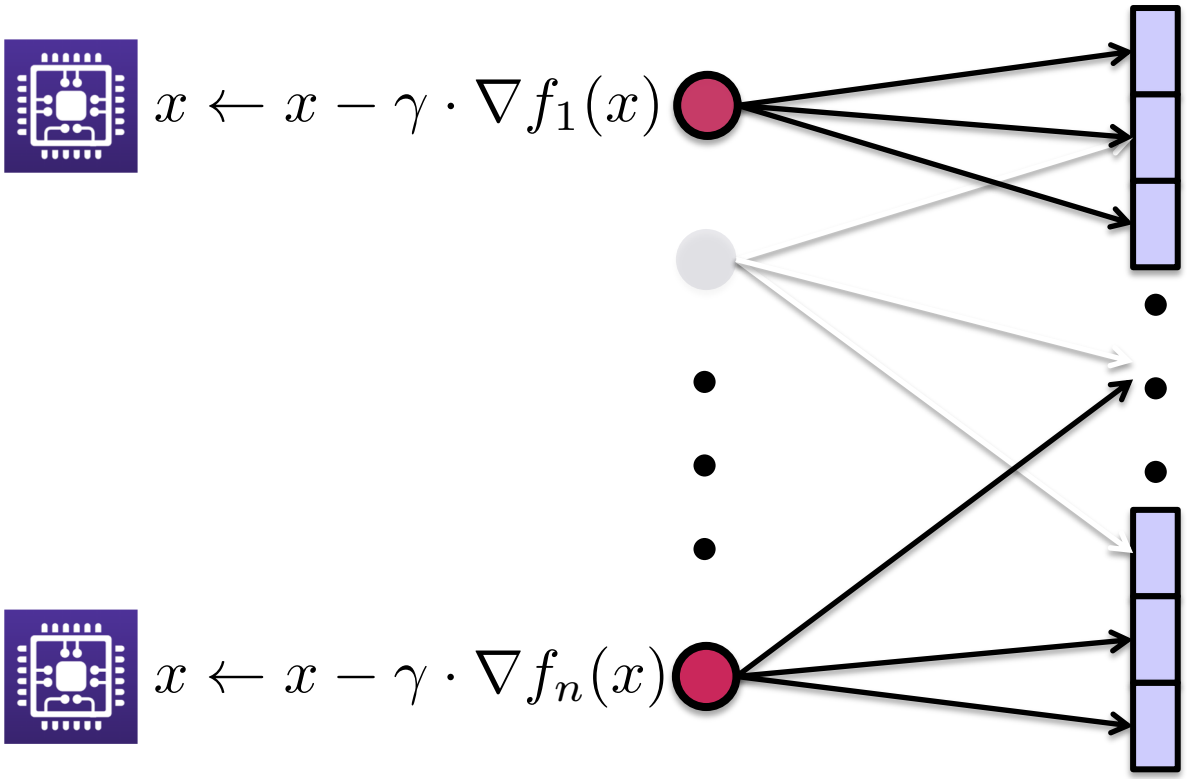
(a) Standard Neural Net



(b) After applying dropout.

Challenges in Parallel SGD

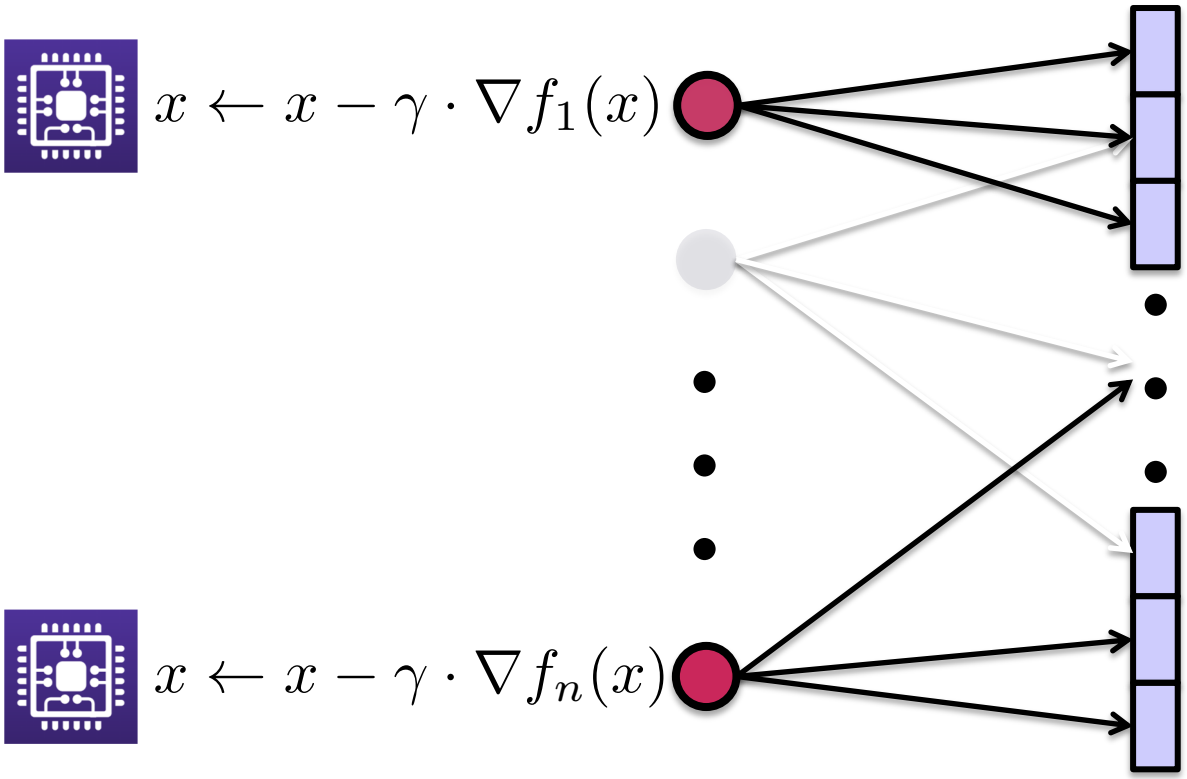
data points | shared variables



No conflict =>
2 parallel iterations = 2 serial iterations

Challenges in Parallel SGD

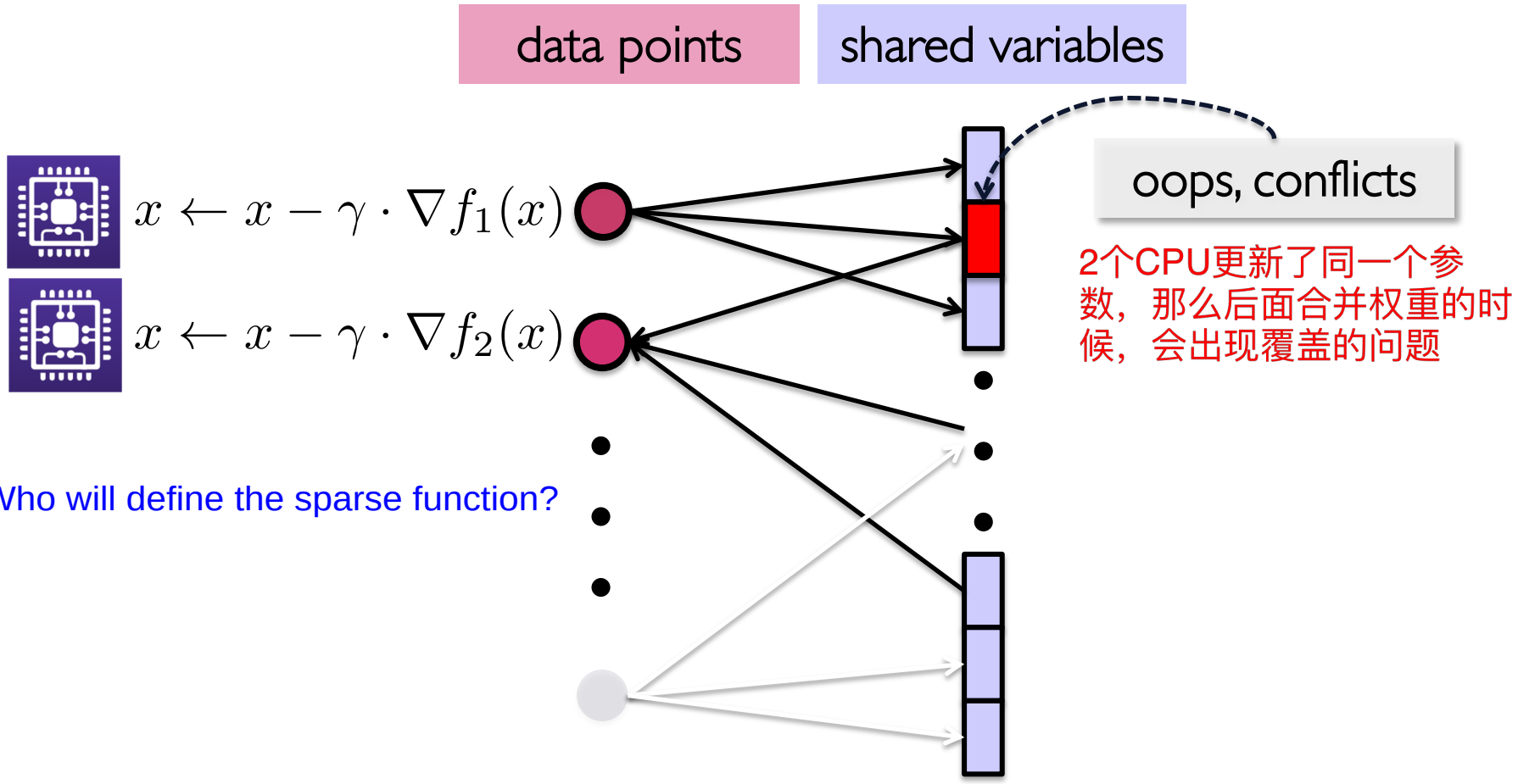
data points | shared variables



No conflict => Speedup

Challenges in Parallel SGD

Shared RAM



What should we do for conflicts?

Approach 1: *Coordinate or Lock*

Approach 2: *Don't Care (Lock-free Async.)*

Prior to 2011 Work

Long line of theoretical work since the 60s
[Chazan, Miranker, 1969]

Foundational work on Asynchronous Optimization
Master/Worker model [Tsitsiklis, Bertsekas, 1986, 1989]

Recent hardware/software advances renewed the interest
Round-robin approach [Zinkevich, Langford, Smola, 2009]
Average Runs [Zinkevich et al., 2009],
Average Gradients [Duchi et al, Dekel et al. 2010]

Many based on "Coordinate" or "Lock" approach

Why Coordinate or Lock?

Issue: Synchronization and comm. overheads

HOGWILD! 2011

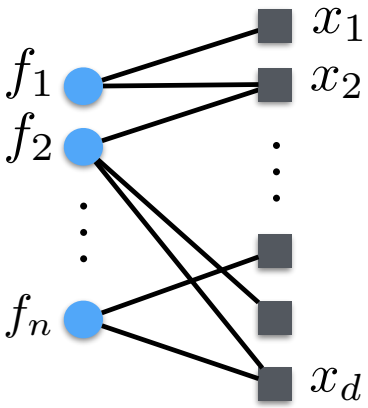
"Run parallel lock-free SGD without synchronization"


Niu


Recht


Ré


Wright



Each processor in parallel

sample function f_i

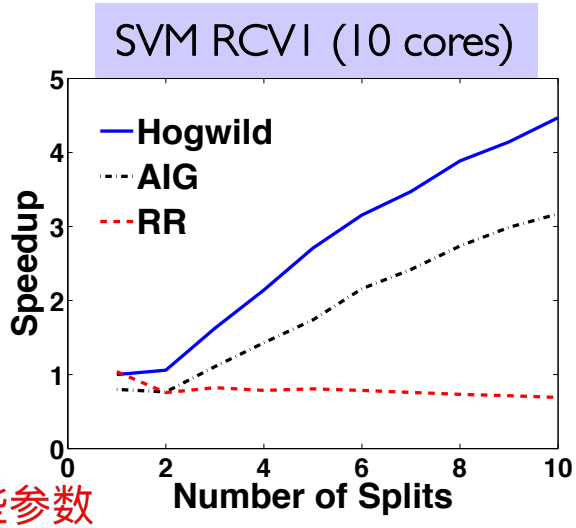
$x =$ read shared memory

$g = -\gamma \cdot \nabla f_i(x)$

for v in the support of f **do**

$x_v \leftarrow x_v + g_v$

只更新f映射的那些参数

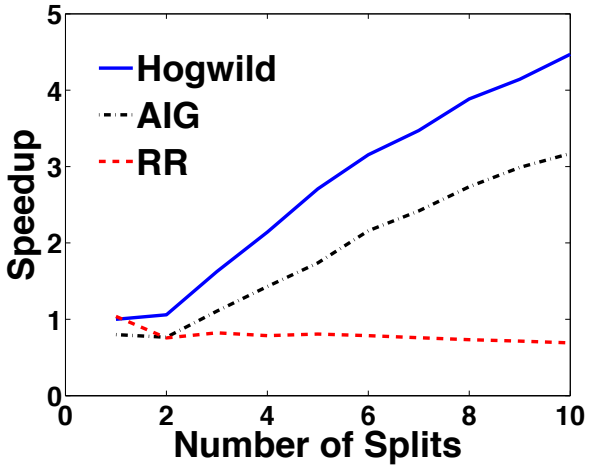


Impact

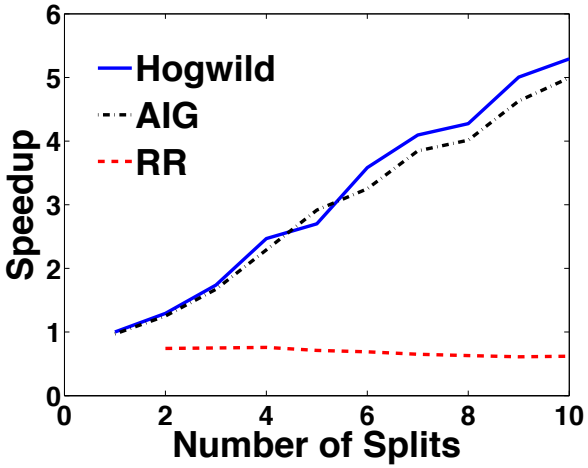
Google Downpour SGD, Microsoft Project Adam *use HOGWILD!*

Renewed interest on async. optimization

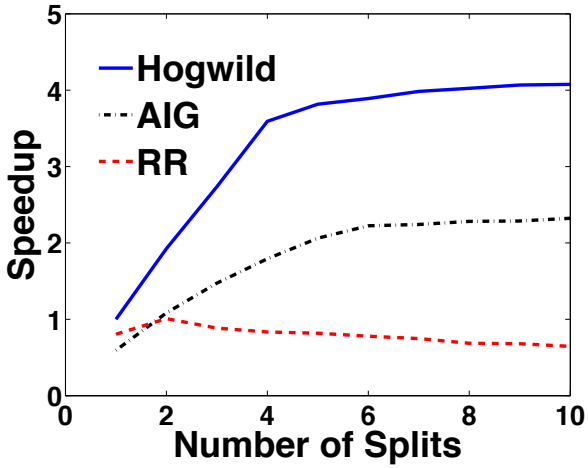
Speedups



SVM
RCV1



MC
Netflix



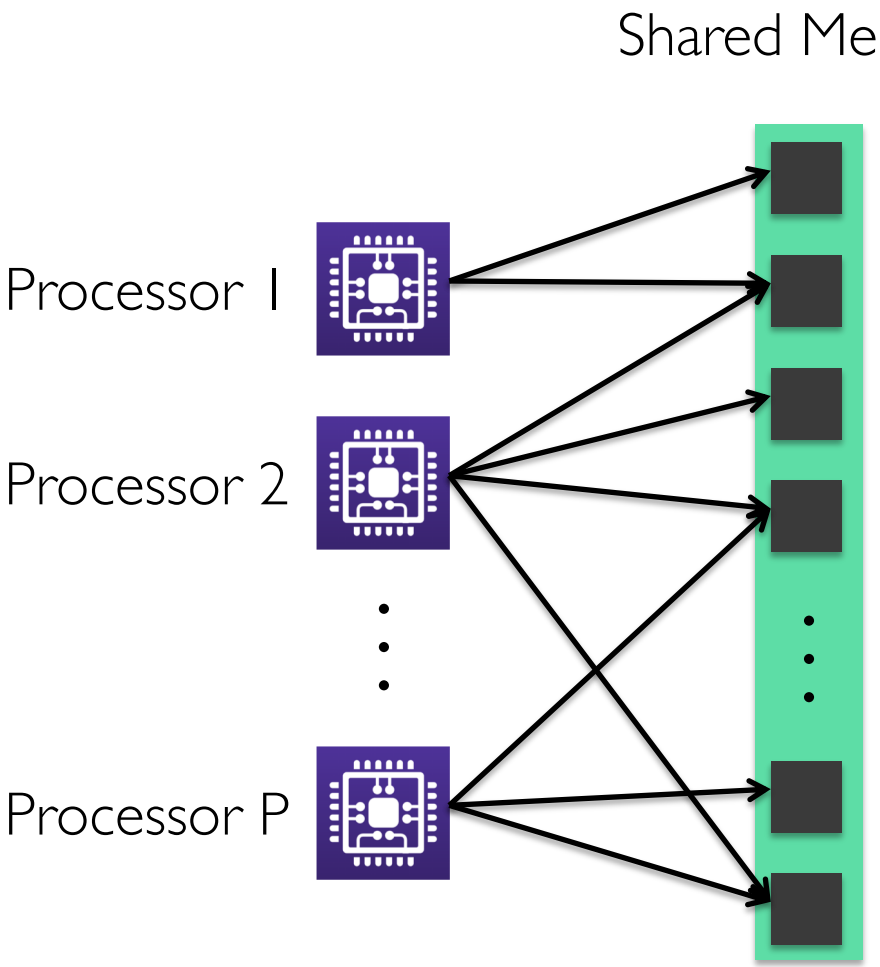
CUTS
Abdomen

Experiments run on 12 core machine

10 cores for gradients, 1 core for data shuffling

Challenges in Analysis

Challenges in Hogwild!



Issues:

1. updates can be old
2. results can overwritten

Incompatible with classic SGD analysis

Analyzing Asynchronous Schemes

A Noisy Lens for Asynchronous Algorithms

Noice的视角

Main Idea

Noisy viewpoint:

Asynchronous(*Algo.*(INPUT)) $\equiv$ Serial(*Algo.*(INPUT + Noise)

扰乱的

Perturbed Iterate Analysis for Asynchronous Stochastic Optimization

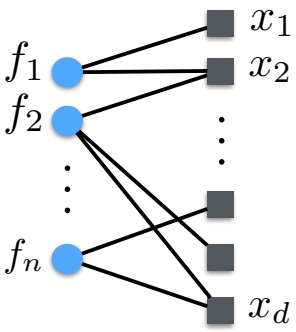
[Mania, Pan, **P**, Recht, Ramchandran, Jordan, 2015]

Joint work with





HOGWILD! as noisy SGD



Each processor in parallel

sample function f_i

$x =$ read shared memory

$g = -\gamma \cdot \nabla f_i(x)$

for v in the support of f **do**

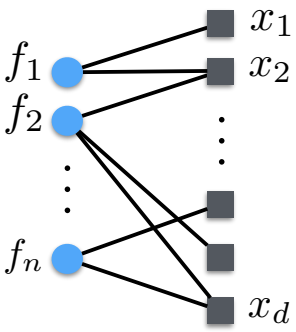
$x_v \leftarrow x_v + g_v$

- Def: s_k is the k-th sampled data point
- Fact: Cores don't read "actual" iterates x_k but "noisy iterates" $\hat{x}_k$

- After *T* processed samples, the contents of RAM are:
  (atomic writes + commutativity)

Ex.

HOGWILD! as noisy SGD



Each processor in parallel

sample function f_i

$x =$ read shared memory

$g = -\gamma \cdot \nabla f_i(x)$

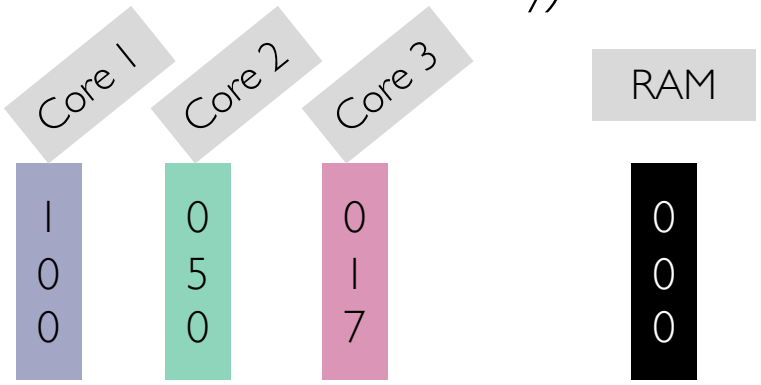
for v in the support of f **do**

$x_v \leftarrow x_v + g_v$

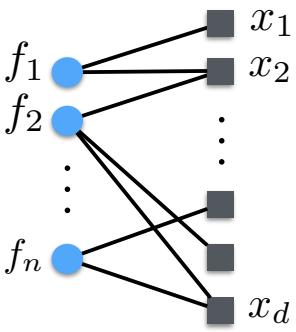
- Def: s_k is the k-th sampled data point
- Fact: Cores don't read "actual" iterates x_k but "noisy iterates" $\hat{x}_k$

- After *T* processed samples, the contents of RAM are:
  (atomic writes + commutativity)

Ex.



HOGWILD! as noisy SGD



Each processor in parallel

sample function f_i

$x =$ read shared memory

$g = -\gamma \cdot \nabla f_i(x)$

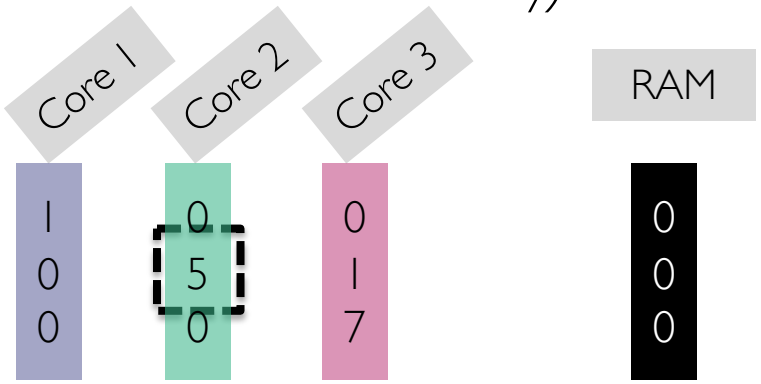
for v in the support of f **do**

$x_v \leftarrow x_v + g_v$

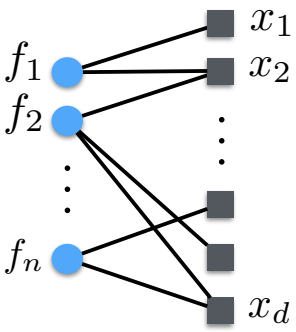
- Def: s_k is the k-th sampled data point
- Fact: Cores don't read "actual" iterates x_k but "noisy iterates" $\hat{x}_k$

- After *T* processed samples, the contents of RAM are:
  (atomic writes + commutativity)

Ex.



HOGWILD! as noisy SGD



Each processor in parallel

sample function f_i

$x =$ read shared memory

$g = -\gamma \cdot \nabla f_i(x)$

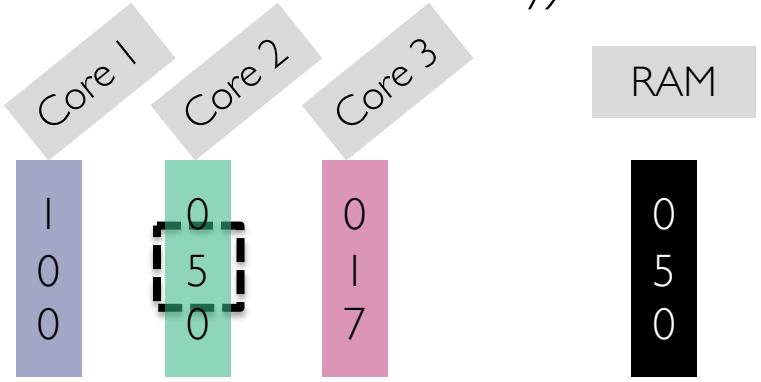
for v in the support of f **do**

$x_v \leftarrow x_v + g_v$

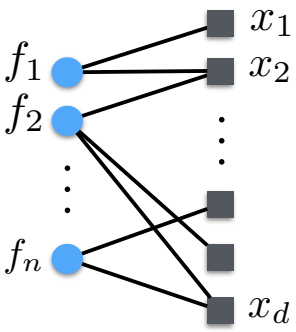
- Def: s_k is the k-th sampled data point
- Fact: Cores don't read "actual" iterates x_k but "noisy iterates" $\hat{x}_k$

- After *T* processed samples, the contents of RAM are:
  (atomic writes + commutativity)

Ex.



HOGWILD! as noisy SGD



Each processor in parallel

sample function f_i

$x =$ read shared memory

$g = -\gamma \cdot \nabla f_i(x)$

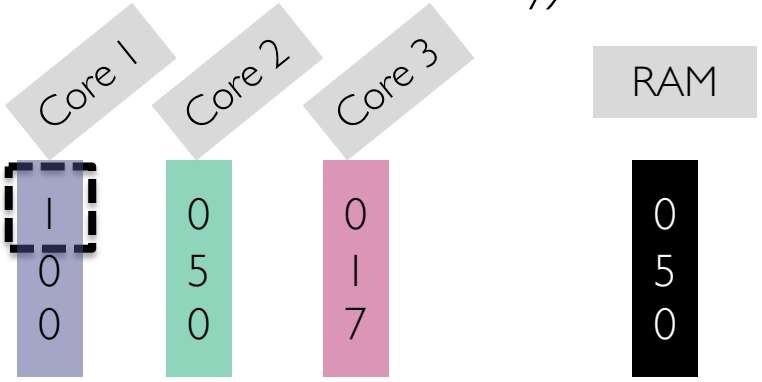
for v in the support of f **do**

$x_v \leftarrow x_v + g_v$

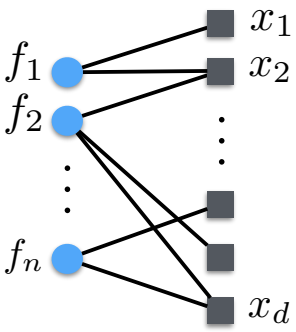
- Def: s_k is the k-th sampled data point
- Fact: Cores don't read "actual" iterates x_k but "noisy iterates" $\hat{x}_k$

- After *T* processed samples, the contents of RAM are:
  (atomic writes + commutativity)

Ex.



HOGWILD! as noisy SGD



Each processor in parallel

sample function f_i

$x =$ read shared memory

$g = -\gamma \cdot \nabla f_i(x)$

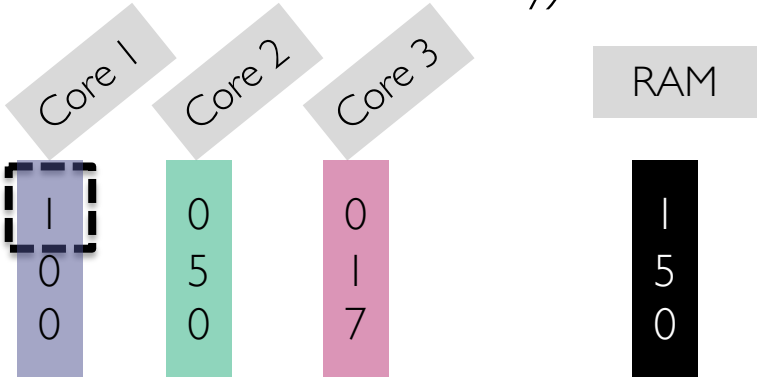
for v in the support of f **do**

$x_v \leftarrow x_v + g_v$

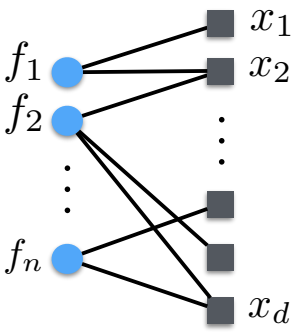
- Def: s_k is the k-th sampled data point
- Fact: Cores don't read "actual" iterates x_k but "noisy iterates" $\hat{x}_k$

- After *T* processed samples, the contents of RAM are:
  (atomic writes + commutativity)

Ex.



HOGWILD! as noisy SGD



Each processor in parallel

sample function f_i

$x =$ read shared memory

$g = -\gamma \cdot \nabla f_i(x)$

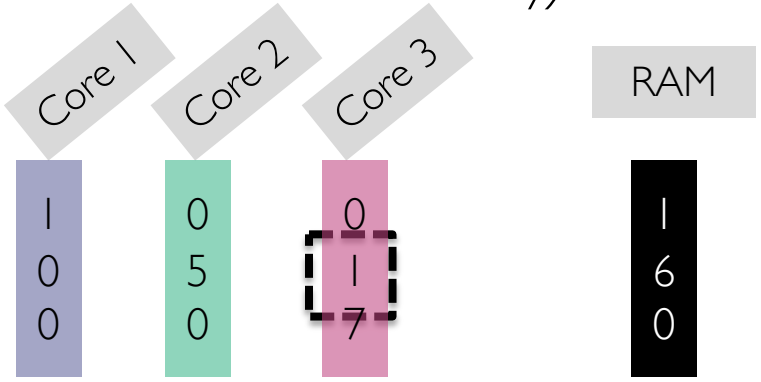
for v in the support of f **do**

$x_v \leftarrow x_v + g_v$

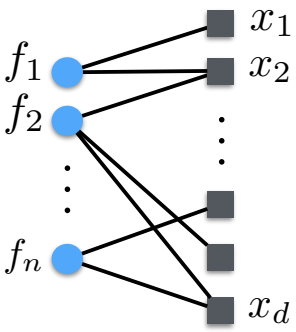
- Def: s_k is the k-th sampled data point
- Fact: Cores don't read "actual" iterates x_k but "noisy iterates" $\hat{x}_k$

- After *T* processed samples, the contents of RAM are:
  (atomic writes + commutativity)

Ex.



HOGWILD! as noisy SGD



Each processor in parallel

sample function f_i

$x =$ read shared memory

$g = -\gamma \cdot \nabla f_i(x)$

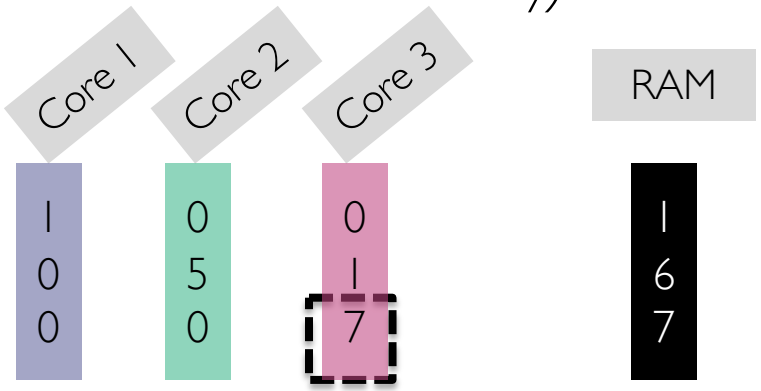
for v in the support of f **do**

$x_v \leftarrow x_v + g_v$

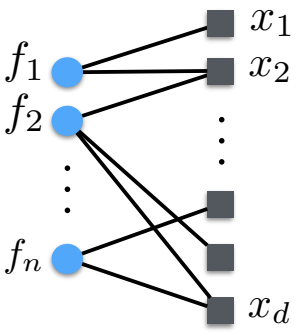
- Def: s_k is the k-th sampled data point
- Fact: Cores don't read "actual" iterates x_k but "noisy iterates" $\hat{x}_k$

- After *T* processed samples, the contents of RAM are:
  (atomic writes + commutativity)

Ex.



HOGWILD! as noisy SGD



Each processor in parallel

sample function f_i

$x =$ read shared memory

$g = -\gamma \cdot \nabla f_i(x)$

for v in the support of f **do**

$x_v \leftarrow x_v + g_v$

- Def: s_k is the k-th sampled data point
- Fact: Cores don't read "actual" iterates x_k but "noisy iterates" $\hat{x}_k$

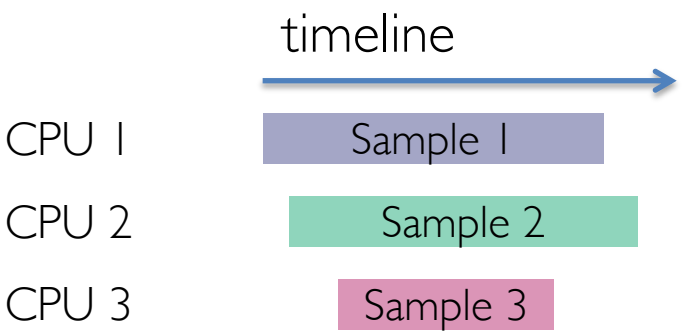
- After *T* processed samples, the contents of RAM are:
  (atomic writes + commutativity)

$$x_0 - \gamma \cdot \nabla f_{s_0}(\hat{x}_0) - \ldots - \gamma \cdot \nabla f_{s_{T-1}}(\hat{x}_{T-1})$$

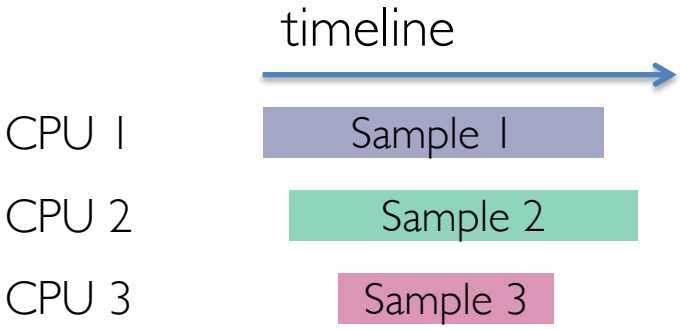
Main Questions:

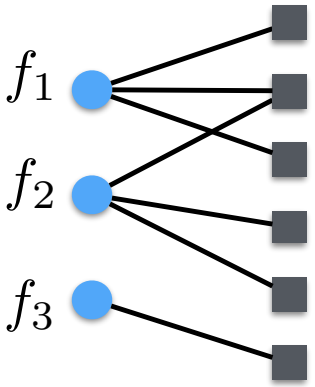
1) Where does noise come from?
2) How strong is it?

Understanding Asynchrony Noise

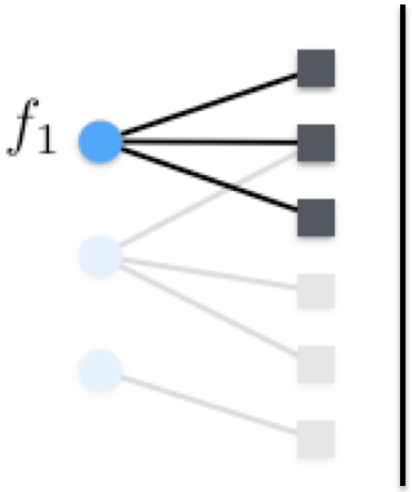


Understanding Asynchrony Noise

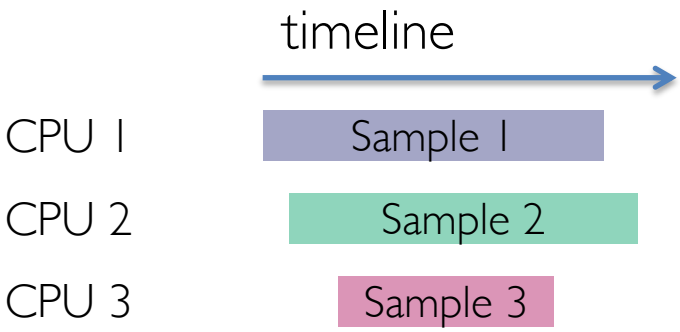


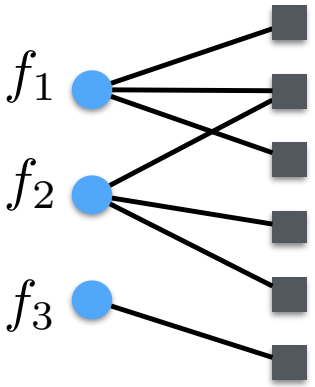


"Serialized" Processing Timeline

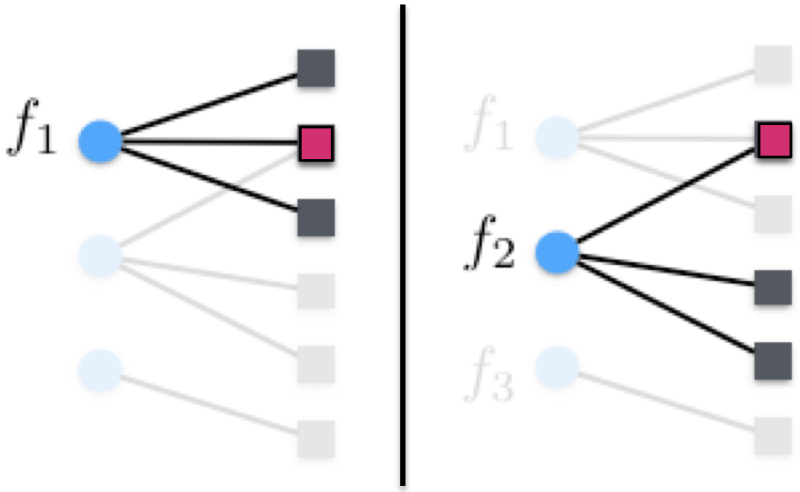


Understanding Asynchrony Noise

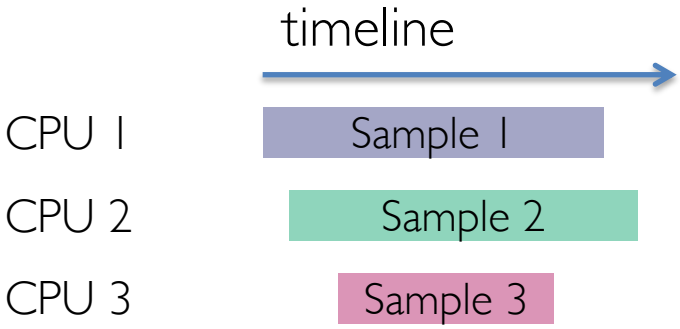


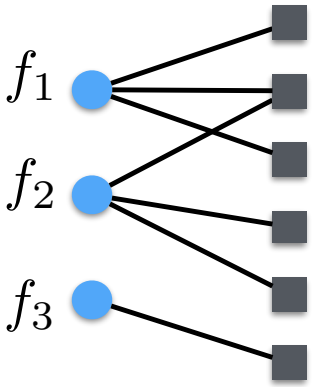


"Serialized" Processing Timeline

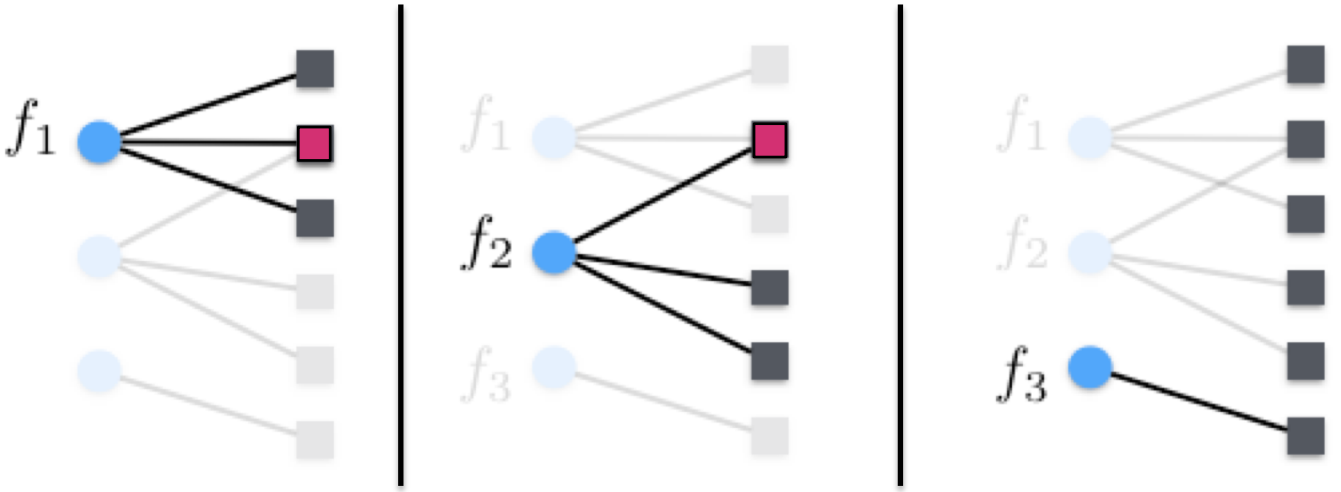


Understanding Asynchrony Noise

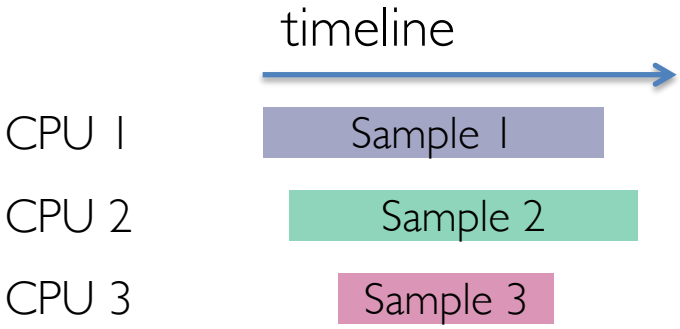


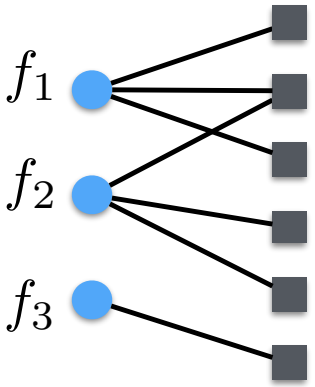


"Serialized" Processing Timeline

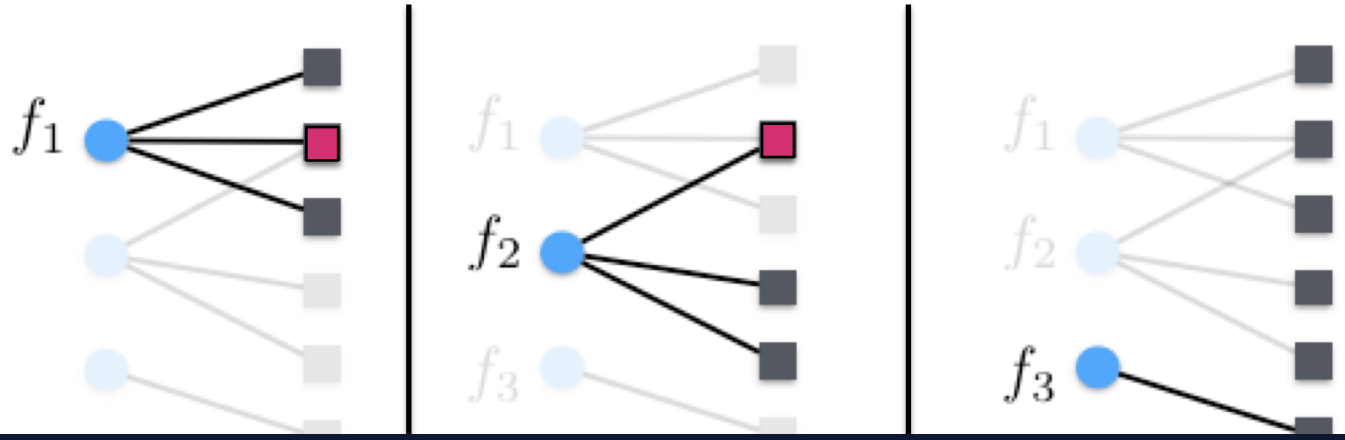


Understanding Asynchrony Noise





"Serialized" Processing Timeline



Asynchrony noise is **combinatorial**
coordinates in conflict can be as noisy as possible.
(no generative model assumptions)

Hogwild Rates: Proof Recap

Hogwild is equivalent to a noisy serial SGD

asynchrony noise affects rates, but if bounded, not by much

When core delay is less than $\tau \leq \frac{n}{2\Delta_{\text{av}}}$, noise does not affect convergence

Hogwild! Achieves linear speedups

*=in terms of worst case convergence

Convergence of Hogwild

THEOREM 3.4. *If the number of samples that overlap in time with a single sample during the execution of* HOGWILD! *is bounded as*

$$\tau = \mathcal{O}\left(\min\left\{\frac{n}{\overline{\Delta}_C}, \frac{M^2}{\epsilon m^2}\right\}\right),$$

HOGWILD!, *with step size* $\gamma = \frac{\epsilon m}{2M^2}$, *reaches an accuracy of* $\mathbb{E}\|\mathbf{x}_k - \mathbf{x}^*\|^2 \leq \epsilon$ *after*

$$T \geq \mathcal{O}(1)\frac{M^2 \log\left(\frac{a_0}{\epsilon}\right)}{\epsilon m^2}$$

iterations.

Open Problems

Open Problems: Part I

Assumptions: Sparsity + convexity => linear speedups

O.P. :
Hogwild! On Dense Problems

Maybe we should featurize dense ML Problems,
so that updates are **sparse**

O.P. :
Fundamental Trade-off
Sparsity vs Learning Quality?

Open Problems: Part 2

- What we proved:

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

- What we really care about:

$$\text{speedup} = \frac{\text{Time of serial } \mathcal{A} \text{ to accuraccy } \epsilon}{\text{Time of parallel } \mathcal{A} \text{ to accuraccy } \epsilon}$$

O.P. :
True Speedup Proofs for Hogwild

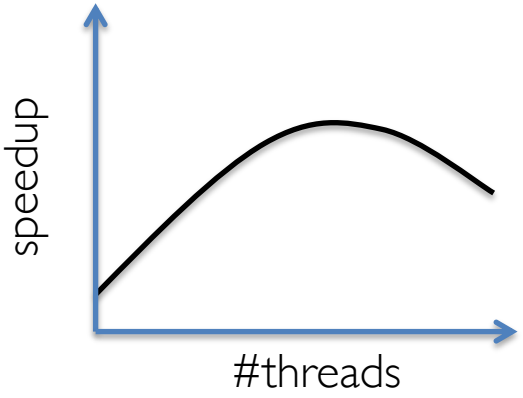
Holy Grail

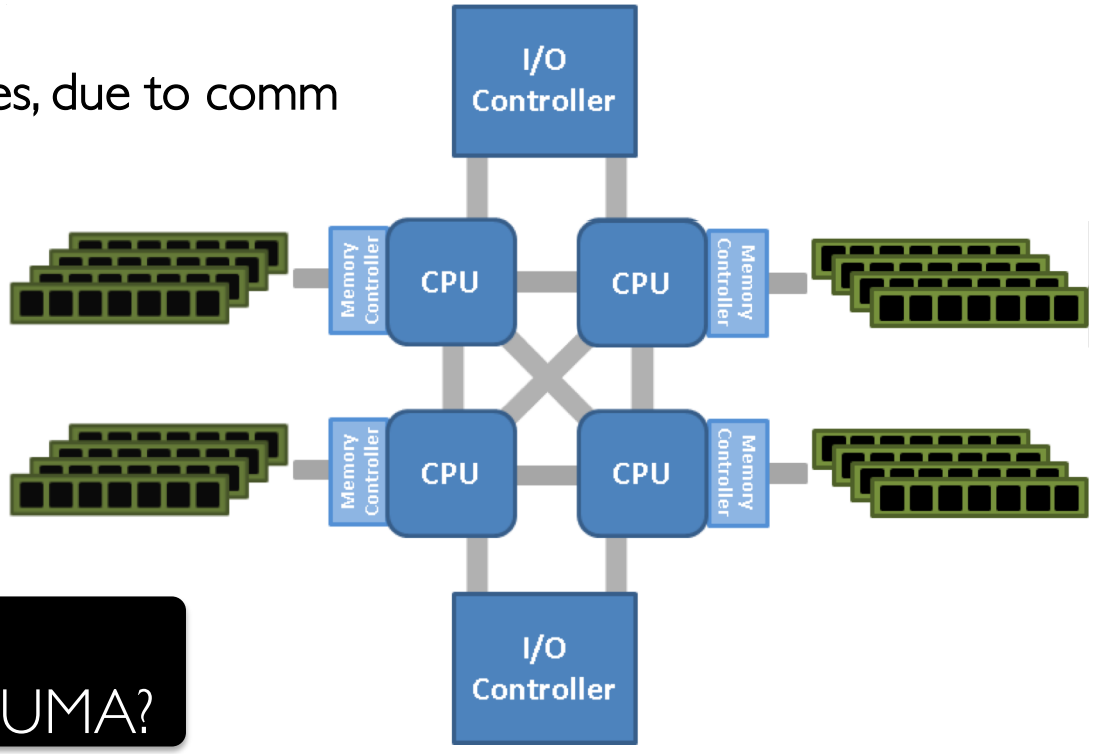
O.P. :
Guarantees for Nonconvex Problems?

Open Problems: Part 3

Hogwild! Algorithms great for Shared Memory Systems

- Issues when scaling across nodes, due to comm





O.P. : 可查验地
How to provably scale on NUMA?

- Similar Issues for Distributed:

O.P. :
Sync vs Async is still open

Reproducible Models

Reproducibility

因为结束时间随机，覆盖的顺序不一样

- HOGWILD! Models are not reproducible

- Each training session has inherent "system" randomness

- Does not allow to recreate models if needed

可解释性

- Barrier for accountability and reproducibility

- How can we resolve it?

Presentation Suggestion

- Suggestion: Minibatch, Hogwils Analysis? Hadoop?

Next Week's Presentation

- Polyak, Boris T. "Some methods of speeding up the convergence of iteration methods." *Ussr computational mathematics and mathematical physics* 4.5 (1964): 1-17.

- Sutskever, Ilya, et al. "On the importance of initialization and momentum in deep learning." *International conference on machine learning*. PMLR, 2013.

- Why Momentum Really Works, Distill, [Goh, 2017]. http://doi.org/10.23915/distill.00006

- Martens, James, and Roger Grosse. "Optimizing neural networks with kronecker-factored approximate curvature." *International conference on machine learning*. PMLR, 2015.