Parallel Optimization in Machine Learning

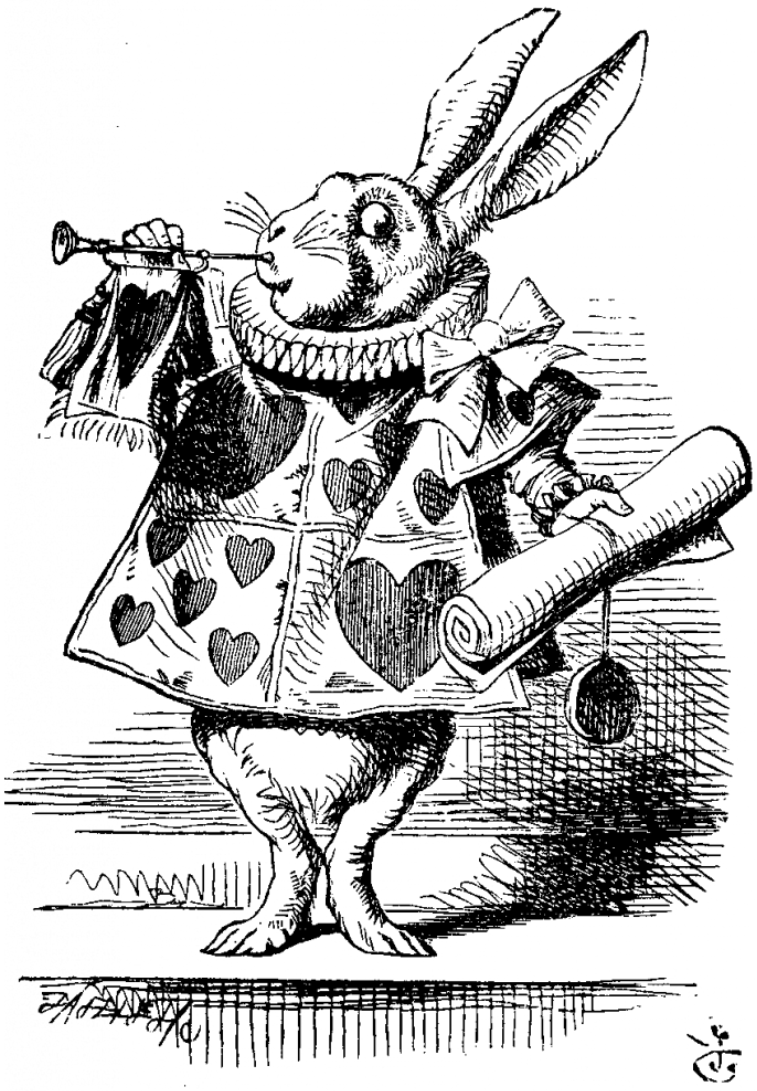
Fabian Pedregosa

December 19, 2017 Huawei Paris Research Center

Berkeley

ETH zürich

About me



- Engineer (2010-2012), Inria Saclay (scikit-learn kickstart).
- PhD (2012-2015), Inria Saclay.
- Postdoc (2015-2016), Dauphine–ENS–Inria Paris.
- Postdoc (2017-present), UC Berkeley - ETH Zurich (Marie-Curie fellowship, European Commission)

Hacker at heart … trapped in a researcher's body.

Motivation

Computer add in 1993

Computer add in 2006

What has changed?

Motivation

Computer add in 1993

Computer add in 2006

What has changed?

2006 = no longer mentions to speed of processors.

Motivation

Computer add in 1993

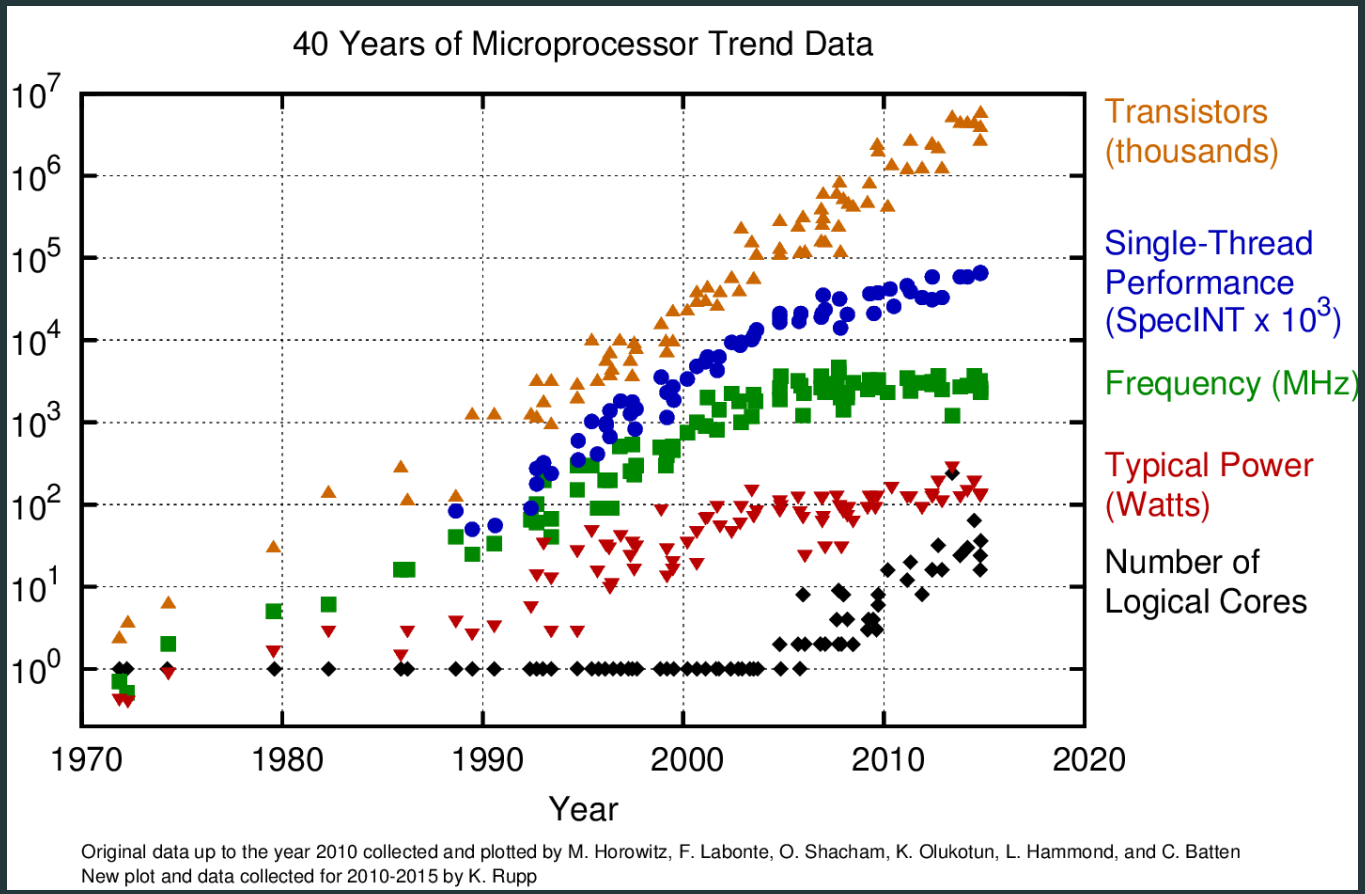
Computer add in 2006

What has changed?

2006 = no longer mentions to speed of processors.

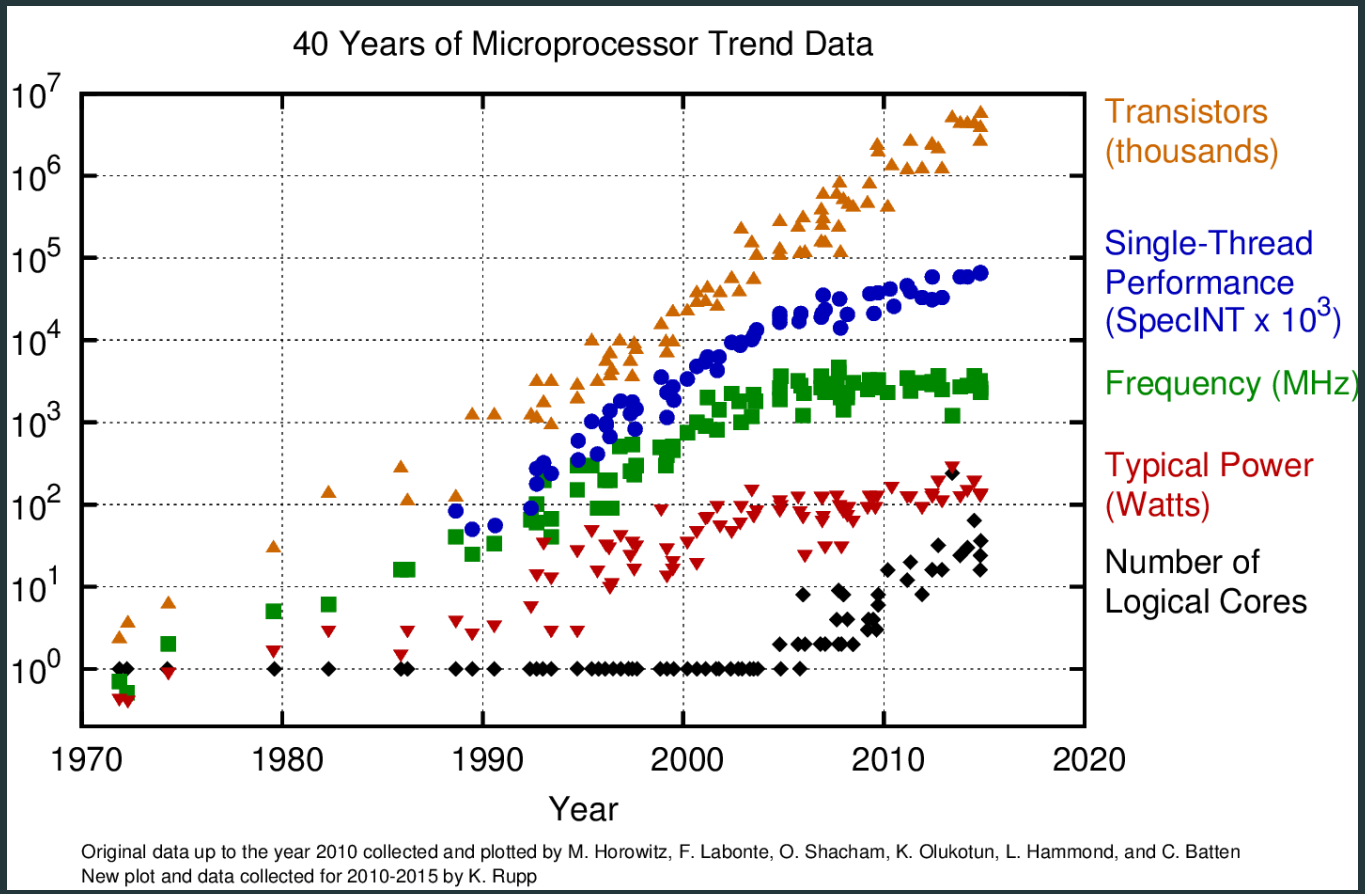
Primary feature: number of cores.

40 years of CPU trends



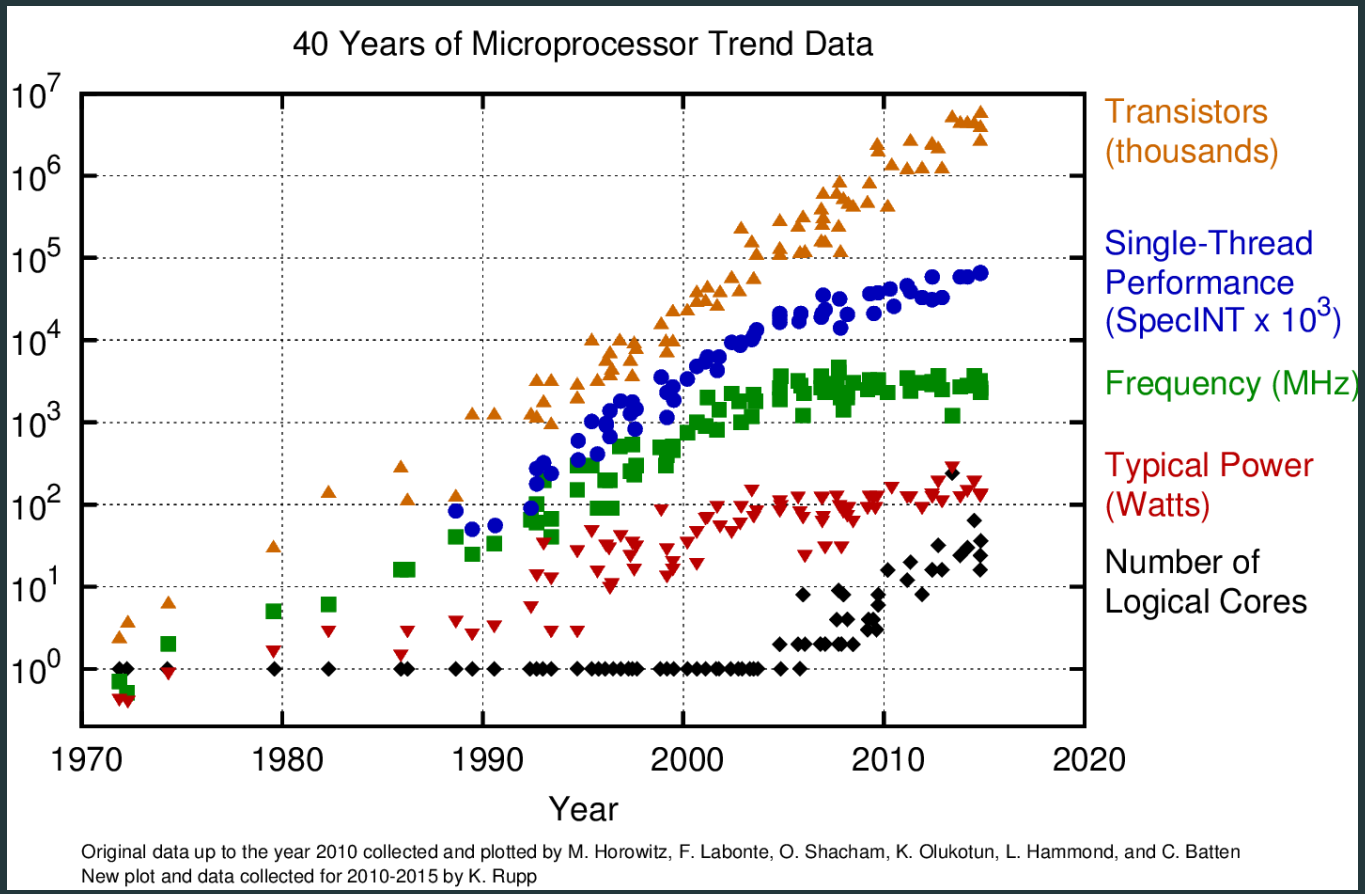
- Speed of CPUs has stagnated since 2005.

40 years of CPU trends



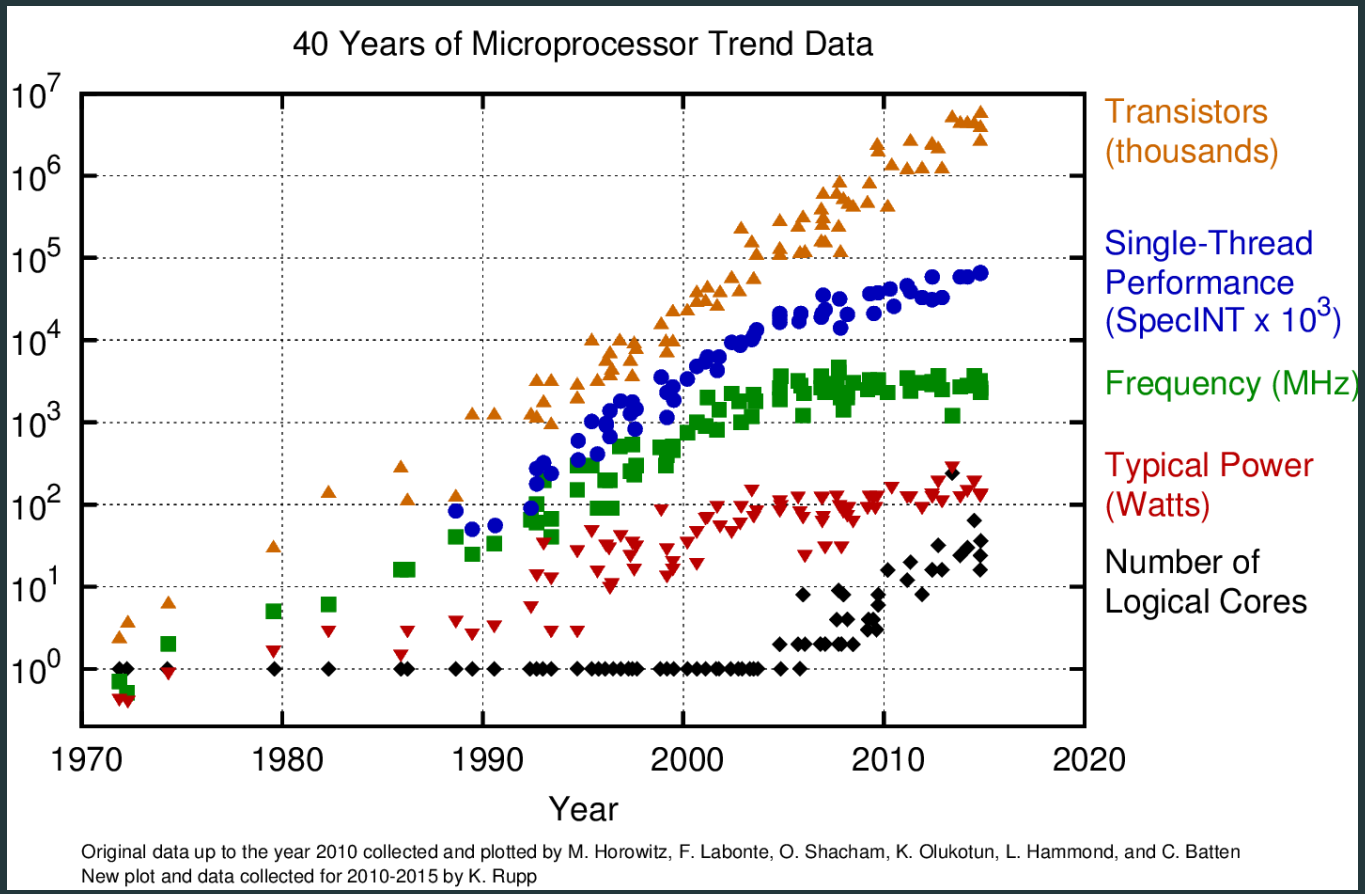
- Speed of CPUs has stagnated since 2005.
- Multi-core architectures are here to stay.

40 years of CPU trends



- Speed of CPUs has stagnated since 2005.
- Multi-core architectures are here to stay.

40 years of CPU trends



- Speed of CPUs has stagnated since 2005.
- Multi-core architectures are here to stay.

Parallel algorithms needed to take advantage of modern CPUs.

Parallel optimization

Parallel algorithms can be divided into two large categories: **synchronous** and **asynchronous**.

Image credits: (Peng et al. 2016)

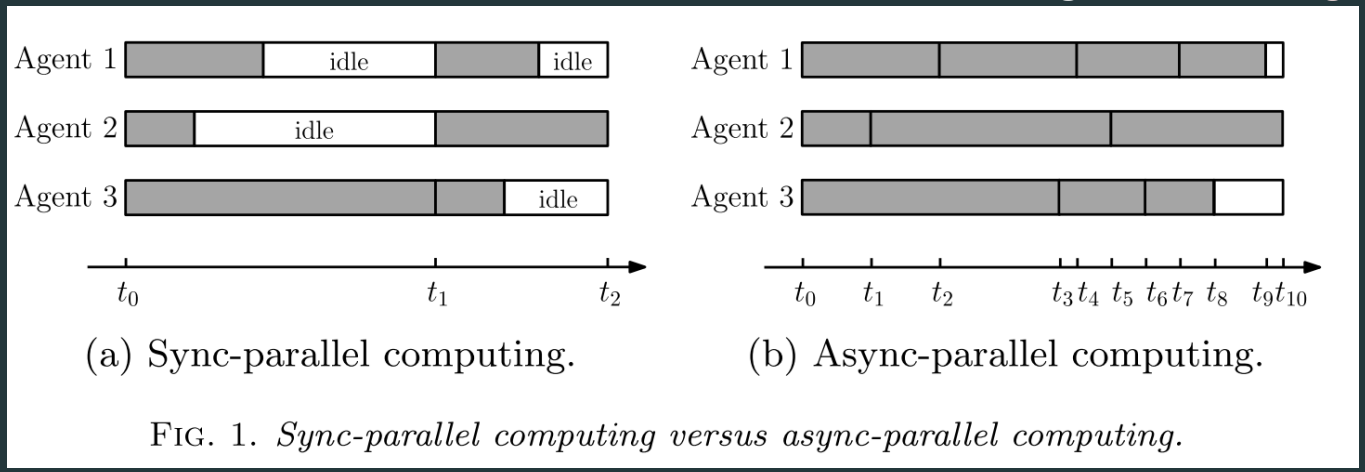


FIG. 1. *Sync-parallel computing versus async-parallel computing.*

Synchronous methods

✔ Easy to implement (i.e., developed software packages).

✔ Well understood.

✖ Limited speedup due to synchronization costs.

Asynchronous methods

✔ Faster, typically larger speedups.

✖ Not well understood, large gap between theory and practice.

✖ No mature software solutions.

Outline

Synchronous methods

- Synchronous (stochastic) gradient descent.

Asynchronous methods

- Asynchronous stochastic gradient descent (Hogwild) (Niu et al. 2011)
- Asynchronous variance-reduced stochastic methods (Leblond, P., and Lacoste-Julien 2017), (Pedregosa, Leblond, and Lacoste-Julien 2017).
- Analysis of asynchronous methods.
- Codes and implementation aspects.

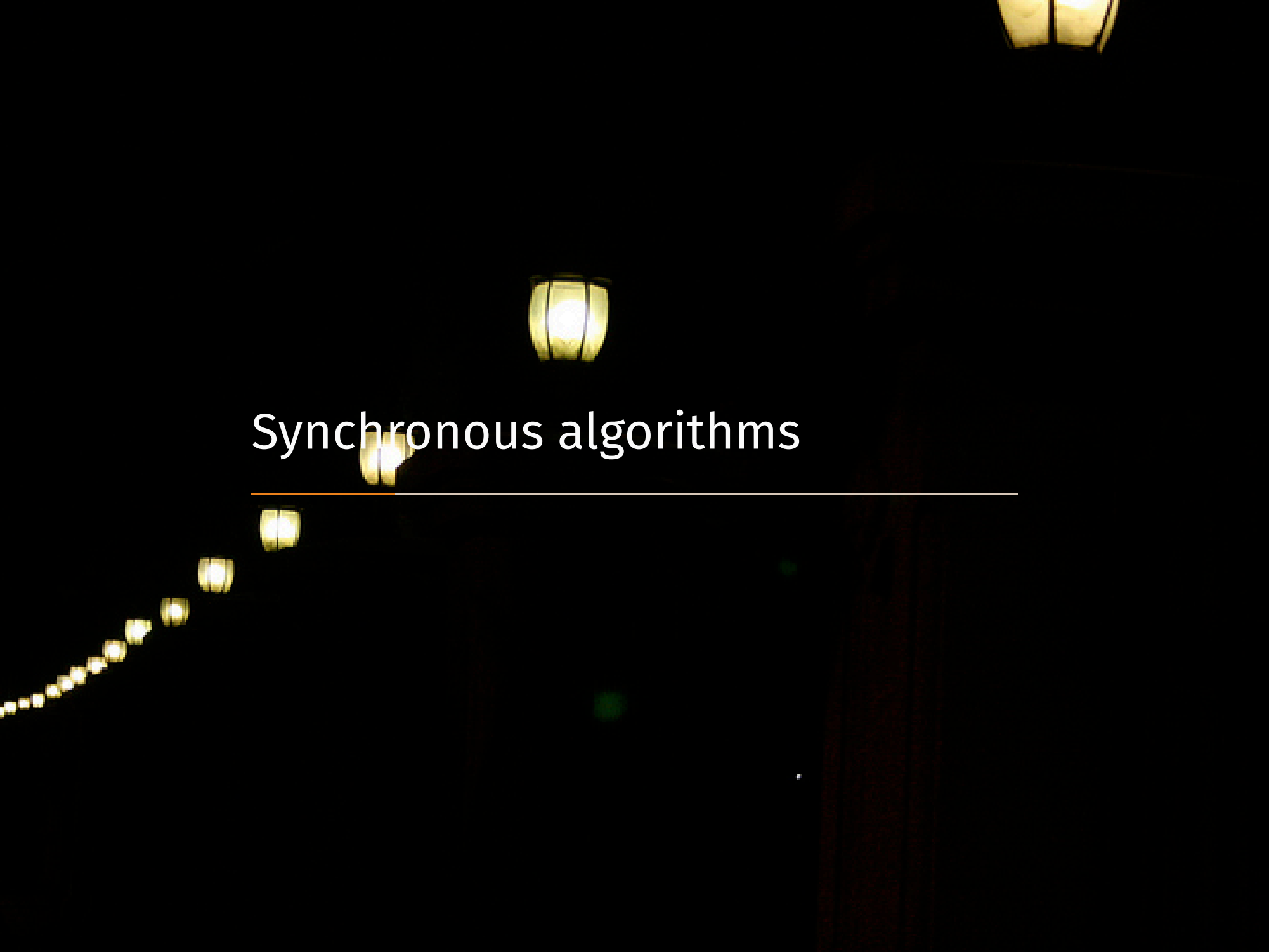
Leaving out many parallel synchronous methods: ADMM (Glowinski and Marroco 1975), CoCoA (Jaggi et al. 2014), DANE (Shamir, Srebro, and Zhang 2014), to name a few.

Outline

Most of the following is joint work with Rémi Leblond and Simon Lacoste-Julien

Rémi Leblond

Simon Lacoste–Julien



Synchronous algorithms

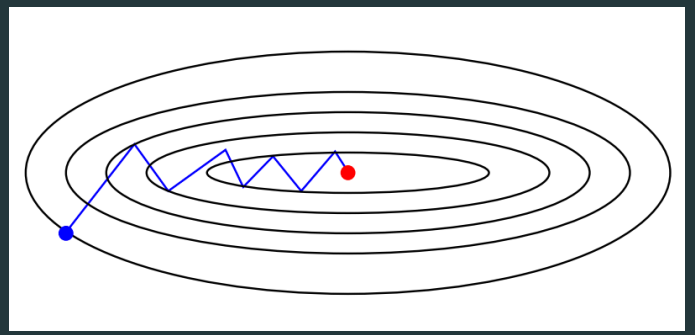
Optimization for machine learning

Large part of problems in machine learning can be framed as optimization problems of the form

$$\underset{x}{\text{minimize}}\, f(x) \stackrel{\text{def}}{=} \frac{1}{n}\sum_{i=1}^{n} f_i(x)$$

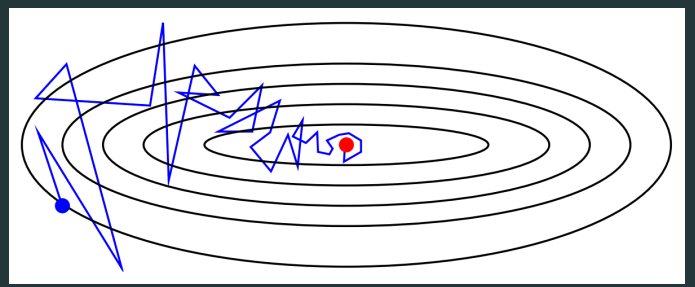
Gradient descent (Cauchy 1847). Descend along steepest direction $(-\nabla f(x))$

$$x^+ = x - \gamma \nabla f(x)$$



Stochastic gradient descent (SGD) (Robbins and Monro 1951). Select a random index i and descent along $-\nabla f_i(x)$:

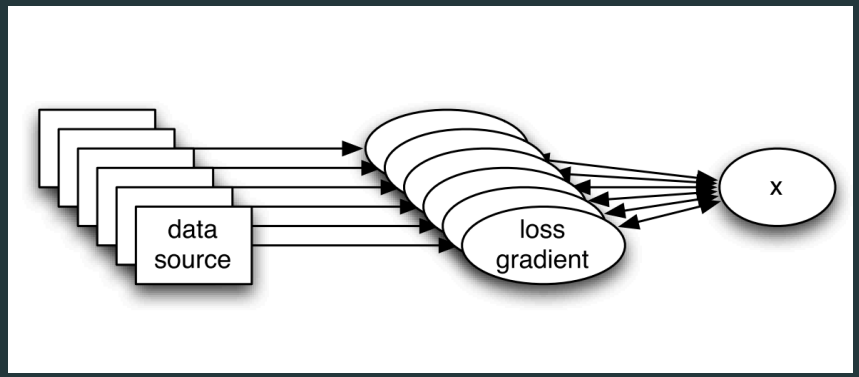
$$x^+ = x - \gamma \nabla f_i(x)$$



images source: Francis Bach

Parallel synchronous gradient descent

Computation of gradient is distributed among k workers



- Workers can be: different computers, CPUs or GPUs
- Popular frameworks: Spark, Tensorflow, PyTorch, neHadoop.

Parallel synchronous gradient descent

1. Choose $n_1, \dots n_k$ that sum to n.
2. Distribute computation of $\nabla f(x)$ among k nodes

$$\nabla f(x) = \frac{1}{n} \sum_{i=1} \nabla f_i(x)$$

$$= \frac{1}{k} (\underbrace{\frac{1}{n_1} \sum_{i=1}^{n_1} \nabla f_i(x)}_{\text{done by worker 1}} + \dots + \underbrace{\frac{1}{n_1} \sum_{i=n_{k-1}}^{n_k} \nabla f_i(x)}_{\text{done by worker } k})$$

3. Perform the gradient descent update by a master node

$$x^+ = x - \gamma \nabla f(x)$$

Parallel synchronous gradient descent

1. Choose $n_1, \dots n_k$ that sum to n.
2. Distribute computation of $\nabla f(x)$ among k nodes

$$\nabla f(x) = \frac{1}{n} \sum_{i=1} \nabla f_i(x)$$

$$= \frac{1}{k} (\underbrace{\frac{1}{n_1} \sum_{i=1}^{n_1} \nabla f_i(x)}_{\text{done by worker 1}} + \dots + \underbrace{\frac{1}{n_1} \sum_{i=n_{k-1}}^{n_k} \nabla f_i(x)}_{\text{done by worker } k})$$

3. Perform the gradient descent update by a master node

$$x^+ = x - \gamma \nabla f(x)$$

✔ Trivial parallelization, same analysis as gradient descent.

✖ Synchronization step every iteration (3.).

Parallel synchronous SGD

Can also be extended to stochastic gradient descent.

1. Select k samples $i_0, \ldots, i_k$ uniformly at random.
2. Compute in parallel ∇f_{i_t} on worker t
3. Perform the (mini-batch) stochastic gradient descent update

$$x^+ = x - \gamma \frac{1}{k} \sum_{t=1}^{k} \nabla f_{i_t}(x)$$

Parallel synchronous SGD

Can also be extended to stochastic gradient descent.

1. Select k samples $i_0, \ldots, i_k$ uniformly at random.
2. Compute in parallel ∇f_{i_t} on worker t
3. Perform the (mini-batch) stochastic gradient descent update

$$x^+ = x - \gamma \frac{1}{k} \sum_{t=1}^{k} \nabla f_{i_t}(x)$$

✔ Trivial parallelization, same analysis as (mini-batch) stochastic gradient descent.

✔ The kind of parallelization that is implemented in deep learning libraries (tensorflow, PyTorch, Thano, etc.).

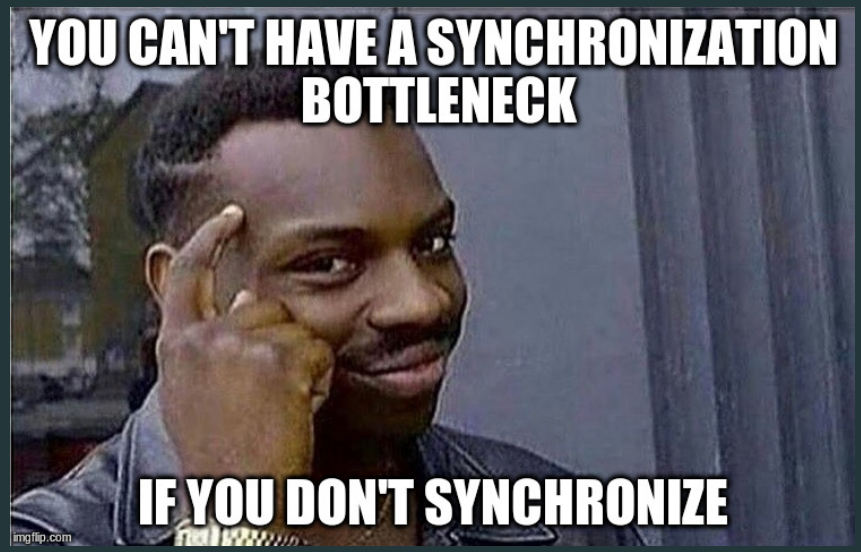
✖ Synchronization step every iteration (3.).

Asynchronous algorithms

Asynchronous SGD

Synchronization is the bottleneck.

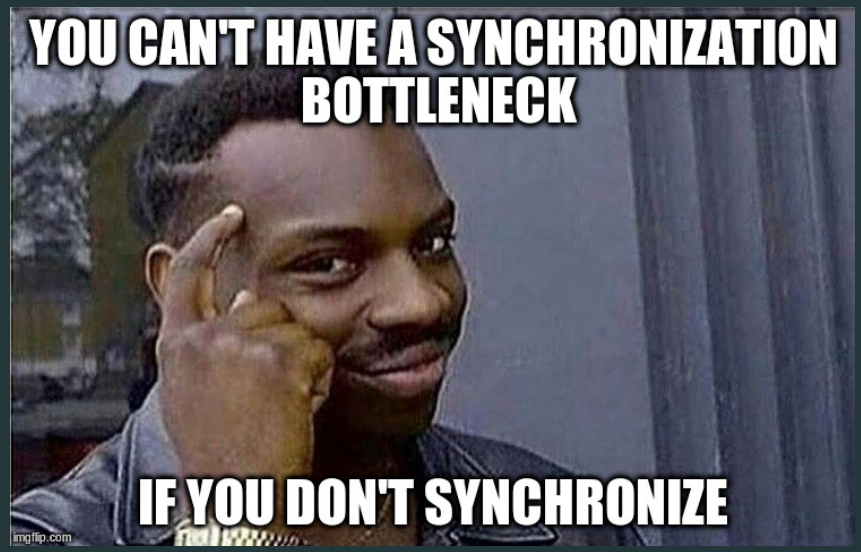
What if we just ignore it?



Asynchronous SGD

Synchronization is the bottleneck.

What if we just ignore it?



Hogwild (Niu et al. 2011): each core runs SGD in parallel, without synchronization, and updates the same vector of coefficients.

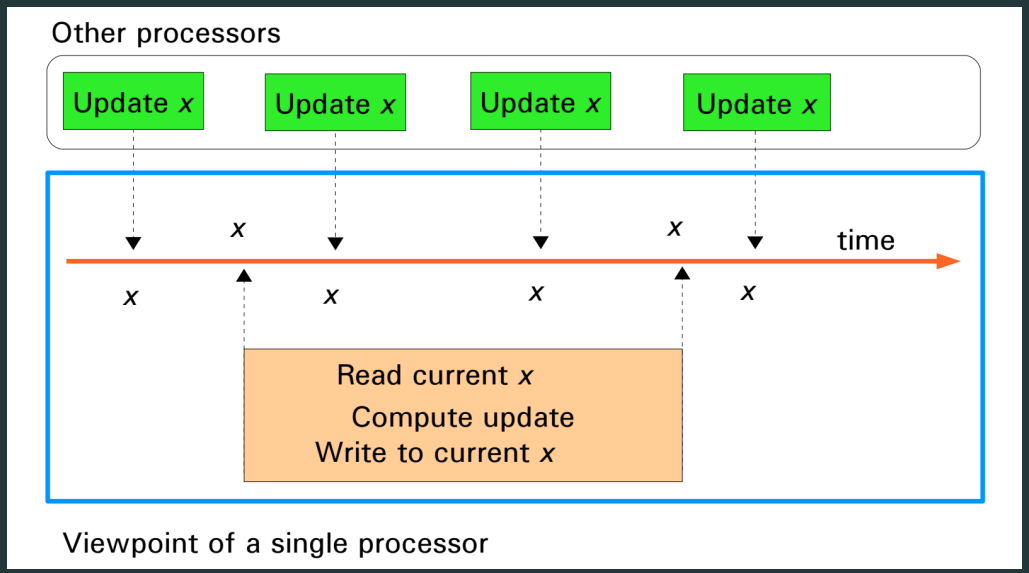
In theory: convergence under very strong assumptions.

In practice: just works.

Hogwild in more detail

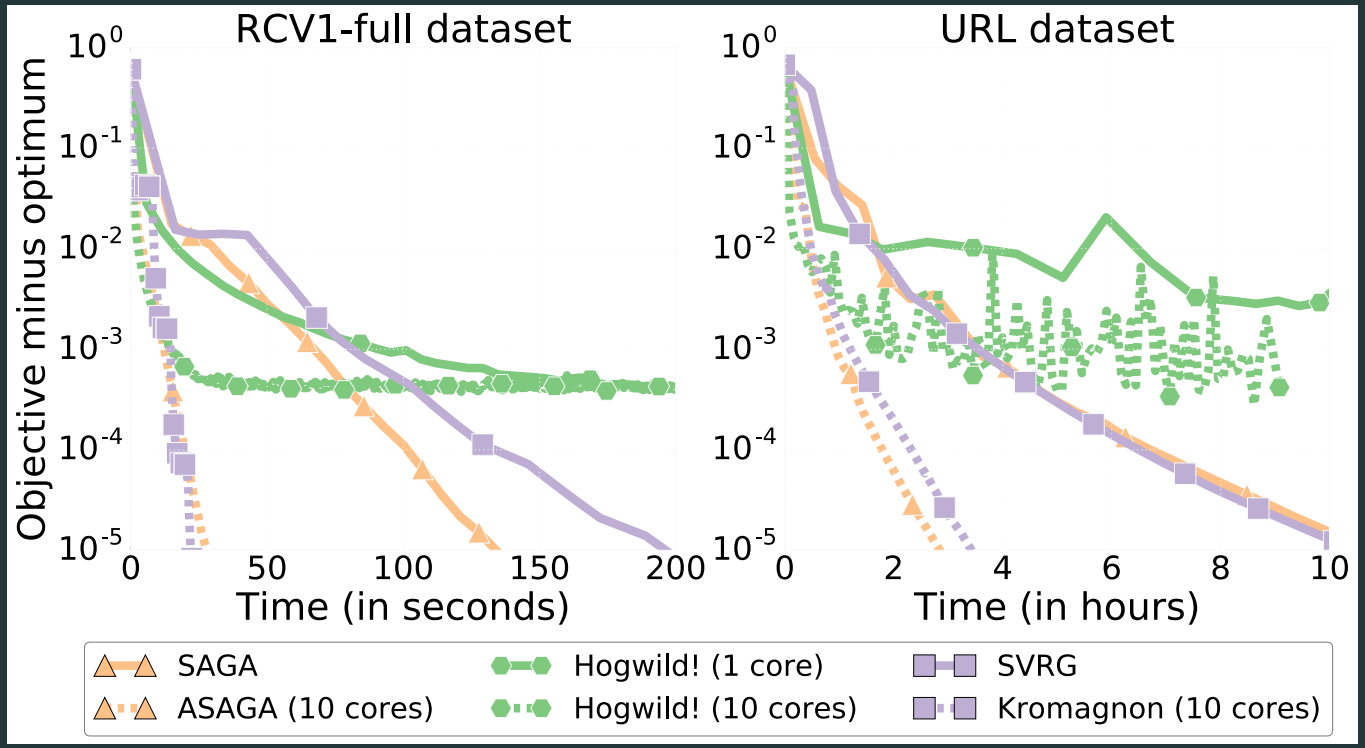
Each core follows the same procedure

1. Read the information from shared memory $\hat{x}$.
2. Sample $i \in \{1, \ldots, n\}$ uniformly at random.
3. Compute partial gradient $\nabla f_i(\hat{x})$.
4. Write the SGD update to shared memory $x = x - \gamma \nabla f_i(\hat{x})$.



Hogwild is fast

Hogwild can be very fast. But its still SGD...



- With constant step size, bounces around the optimum.
- With decreasing step size, slow convergence.
- There are better alternatives (Emilie already mentioned some)


Looking for excitement? ...
analyze asynchronous methods!

Analysis of asynchronous methods

Simple things become counter-intuitive, e.g, how to **name** the iterates?

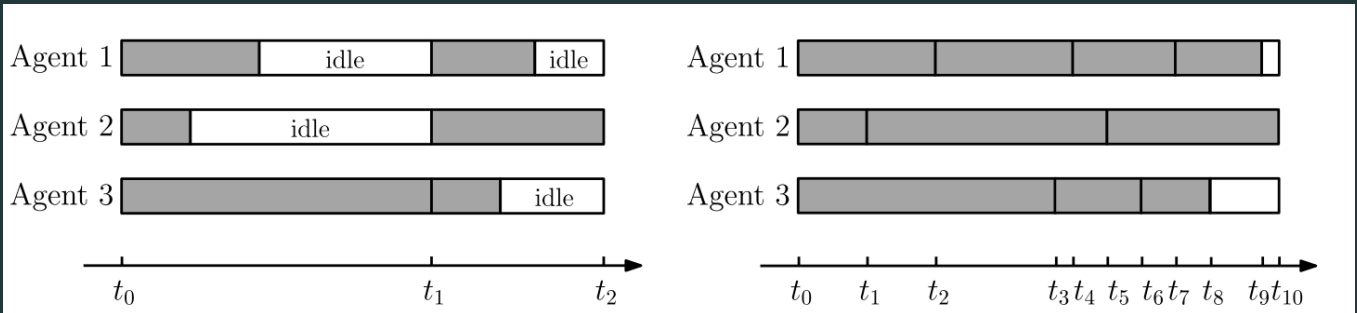


FIG. 1. *Sync-parallel computing versus async-parallel computing.*

⚠ Iterates will change depending on the speed of processors

Naming scheme in Hogwild

Simple, intuitive and wrong

Each time a core has finished writing to shared memory, increment iteration counter.

$\iff \hat{x}_t = (t+1)$-th succesfull update to shared memory.

Value of $\hat{x}_t$ and i_t are not determined until the iteration has finished.

$\implies \hat{x}_t$ and i_t are not necessarily independent.

Unbiased gradient estimate

SGD-like algorithms crucially rely on the unbiased property $\mathbb{E}_i[\nabla f_i(x)] = \nabla f(x)$.

For synchronous algorithms, follows from the uniform sampling of i

$$\mathbb{E}_i[\nabla f_i(x)] = \sum_{i=1}^{n} \text{Proba}(\text{selecting } i) \nabla f_i(x)$$
$$\overset{\text{uniform sampling}}{=} \sum_{i=1}^{n} \frac{1}{n} \nabla f_i(x) = \nabla f(x)$$

A problematic example

This labeling scheme is *incompatible* with unbiasedness assumption used in proofs.

A problematic example

This labeling scheme is *incompatible* with unbiasedness assumption used in proofs.

Illustration: problem with two samples and two cores $f = \frac{1}{2}(f_1 + f_2)$. Computing ∇f_1 is much expensive than ∇f_2.

A problematic example

This labeling scheme is *incompatible* with unbiasedness assumption used in proofs.

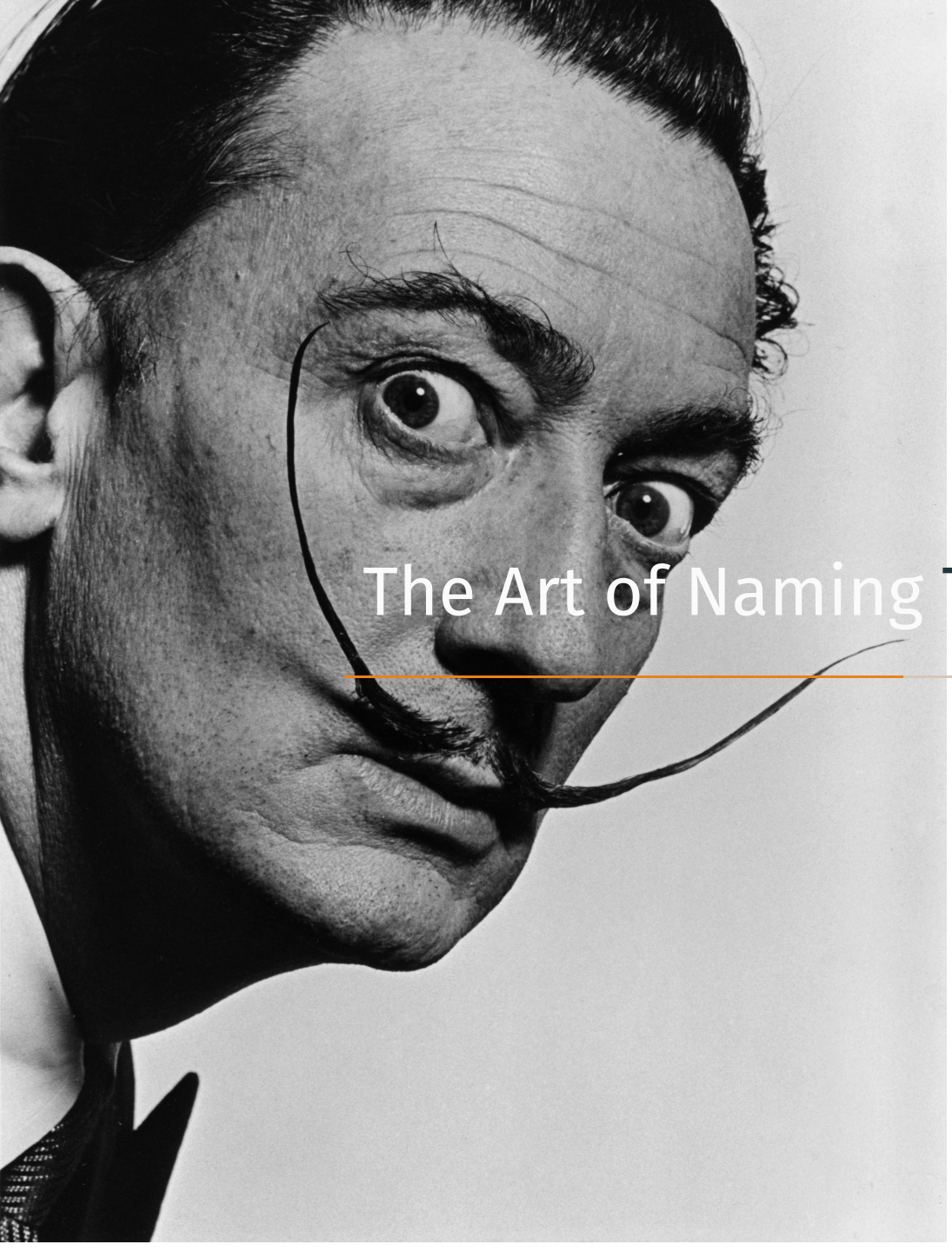
Illustration: problem with two samples and two cores $f = \frac{1}{2}(f_1 + f_2)$. Computing ∇f_1 is much expensive than ∇f_2.

Start at x_0. Because of the random sampling there are 4 possible scenarios:

1. Core 1 selects f_1, Core 2 selects $f_1 \implies x_1 = x_0 - \gamma \nabla f_1(x)$
2. Core 1 selects f_1, Core 2 selects $f_2 \implies x_1 = x_0 - \gamma \nabla f_2(x)$
3. Core 1 selects f_2, Core 2 selects $f_1 \implies x_1 = x_0 - \gamma \nabla f_2(x)$
4. Core 1 selects f_2, Core 2 selects $f_2 \implies x_1 = x_0 - \gamma \nabla f_2(x)$

So we have

$$\begin{aligned} \mathbb{E}_i \left[\nabla f_i\right] &= \frac{1}{4} f_1 + \frac{3}{4} f_2 \\ &\neq \frac{1}{2} f_1 + \frac{1}{2} f_2 \text{ !!} \end{aligned}$$



The Art of Naming Things

A new labeling scheme

New way to name iterates.

"After read" labeling (Leblond, P., and Lacoste-Julien 2017). Increment counter each time we *read* the vector of coefficients from shared memory.

A new labeling scheme

New way to name iterates.

"After read" labeling (Leblond, P., and Lacoste-Julien 2017). Increment counter each time we *read* the vector of coefficients from shared memory.

✔ No dependency between i_t and the cost of computing ∇f_{i_t}.

✔ Full analysis of Hogwild and other asynchronous methods in *"Improved parallel stochastic optimization analysis for incremental methods", Leblond, P., and Lacoste-Julien* (submitted).

Asynchronous SAGA

The SAGA algorithm

Setting:

$$\underset{x}{\text{minimize}} \frac{1}{n} \sum_{i=1}^{n} f_i(x)$$

The **SAGA** algorithm (Defazio, Bach, and Lacoste-Julien 2014).

Select $i \in \{1, \ldots, n\}$ and compute (x^+, α^+) as

$$x^+ = x - \gamma(\nabla f_i(x) - \alpha_i + \overline{\alpha}) \,;\; \alpha_i^+ = \nabla f_i(x)$$

- Like SGD, update is unbiased, i.e., $\mathbb{E}_i[\nabla f_i(x) - \alpha_i + \overline{\alpha})] = \nabla f(x)$.
- Unlike SGD, because of memory terms α, variance $\to 0$.
- Unlike SGD, converges with fixed step size $(1/3L)$

The SAGA algorithm

Setting:

$$\underset{x}{\text{minimize}} \frac{1}{n} \sum_{i=1}^{n} f_i(x)$$

The **SAGA** algorithm (Defazio, Bach, and Lacoste-Julien 2014).

Select $i \in \{1, \ldots, n\}$ and compute (x^+, α^+) as

$$x^+ = x - \gamma(\nabla f_i(x) - \alpha_i + \overline{\alpha}) \,;\; \alpha_i^+ = \nabla f_i(x)$$

- Like SGD, update is unbiased, i.e., $\mathbb{E}_i[\nabla f_i(x) - \alpha_i + \overline{\alpha})] = \nabla f(x)$.
- Unlike SGD, because of memory terms α, variance $\to 0$.
- Unlike SGD, converges with fixed step size $(1/3L)$

Super easy to use in scikit-learn

```
from sklearn.linear_model import LogisticRegression
clf = LogisticRegression(solver='saga')
clf.fit(X, y)
```

Sparse SAGA

Need for a sparse variant of SAGA

- A large part of large scale datasets are sparse.
- For sparse datasets and generalized linear models (e.g., least squares, logistic regression, etc.), partial gradients ∇f_i are sparse too.
- Asynchronous algorithms work best when updates are sparse.

SAGA update is inefficient for sparse data

$$x^+ = x - \gamma(\underbrace{\nabla f_i(x)}_{\text{sparse}} - \underbrace{\alpha_i}_{\text{sparse}} + \underbrace{\overline{\alpha}}_{\text{dense!}}) \,;\; \alpha_i^+ = \nabla f_i(x)$$

[scikit-learn uses many tricks to make it efficient that we cannot use in asynchronous version]

Sparse SAGA

Sparse variant of SAGA. Relies on

- Diagonal matrix $P_i =$ projection onto the support of ∇f_i
- Diagonal matrix D defined as
  $D_{j,j} = n/$number of times $\nabla_j f_i$ is nonzero.

Sparse SAGA algorithm (Leblond, P., and Lacoste-Julien 2017)

$$x^+ = x - \gamma(\nabla f_i(x) - \alpha_i + P_i D\overline{\alpha})\,;\ \alpha_i^+ = \nabla f_i(x)$$

- All operations are sparse, cost per iteration is $\mathcal{O}(\text{nonzeros in } \nabla f_i)$.
- Same convergence properties than SAGA, but with cheaper iterations in presence of sparsity.
- Crucial property: $\mathbb{E}_i[P_i D] = I$.

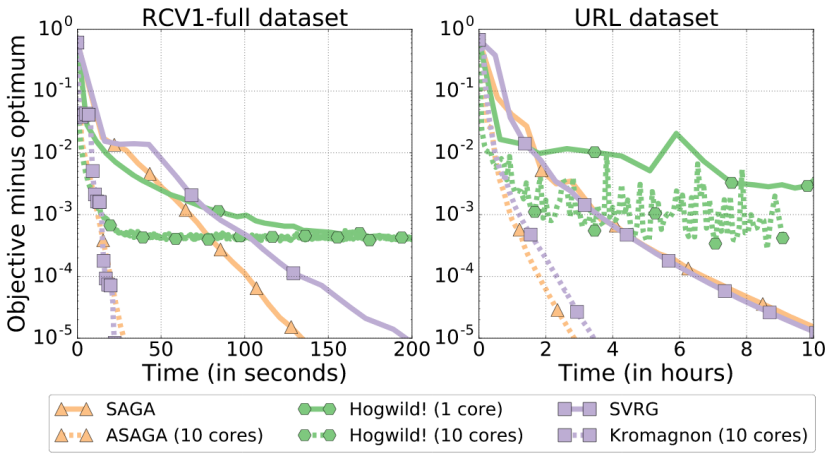
Asynchronous SAGA (ASAGA)

- Each core runs an instance of Sparse SAGA.
- Updates the same vector of coefficients $\boldsymbol{\alpha}, \overline{\boldsymbol{\alpha}}$.

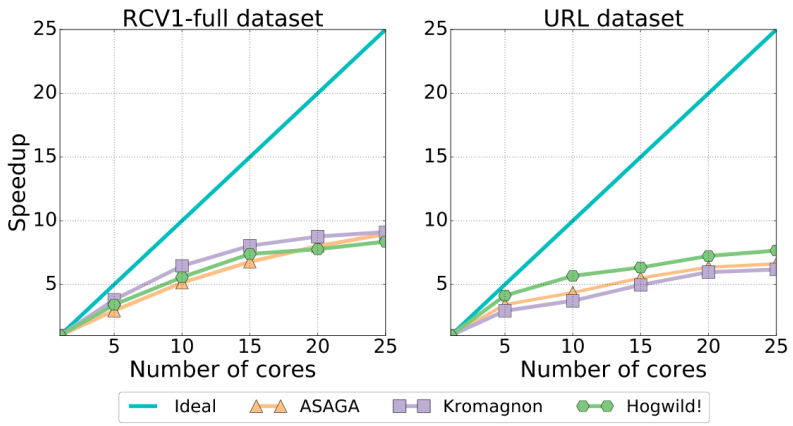
Theory: Under standard assumptions (bounded dalays), same convergence rate than sequential version.

$\Longrightarrow$ theoretical linear speedup with respect to number of cores.

Experiments

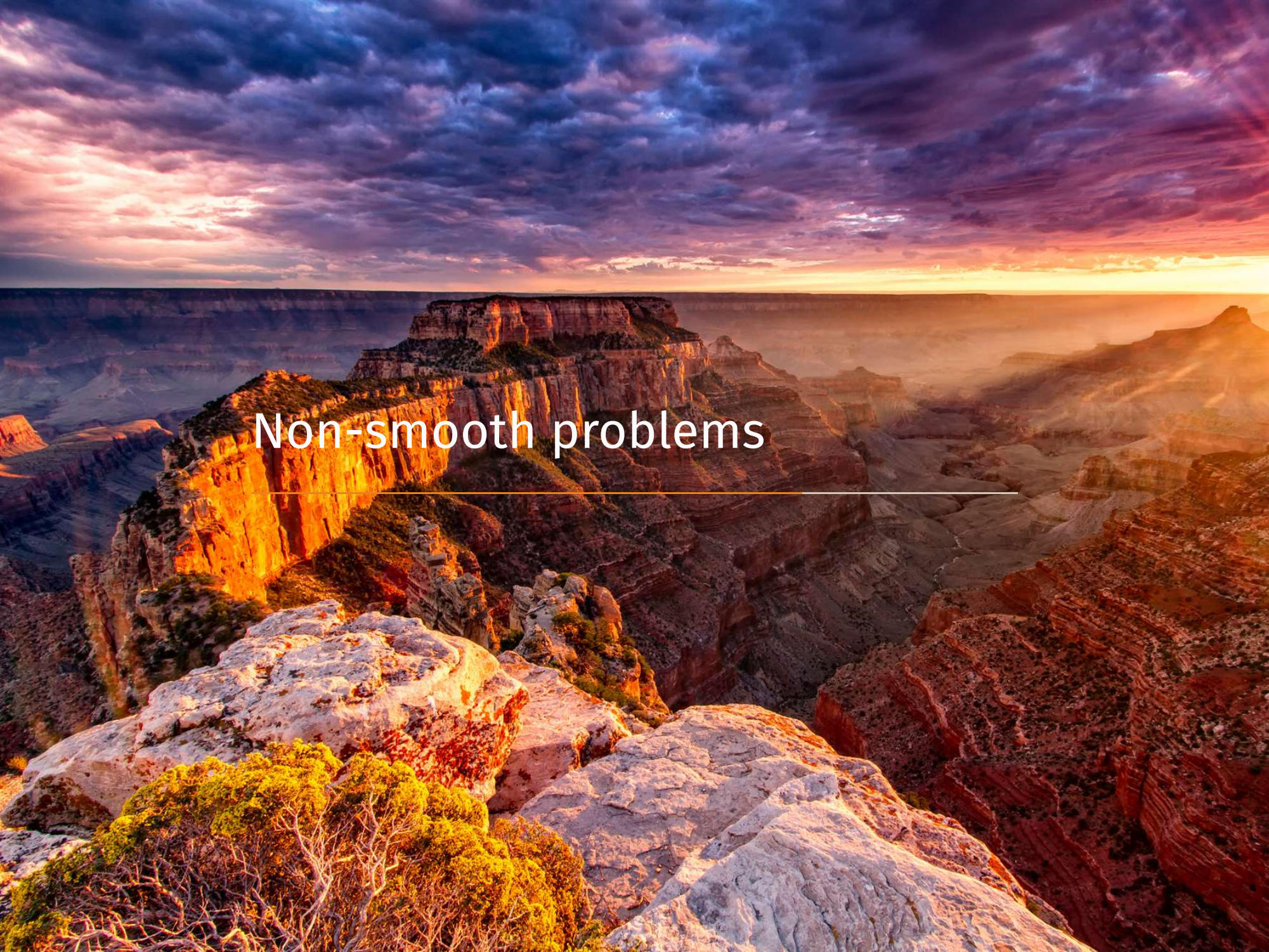


(a) Suboptimality as a function of time.



(b) Speedup as a function of the number of cores

- Improved convergence of variance-reduced methods wrt SGD.
- Significant improvement between 1 and 10 cores.
- Speedup is significant, but far from ideal.


Non-smooth problems

Composite objective

Previous methods assume objective function is smooth.

Cannot be applied to Lasso, Group Lasso, box constraints, etc.

Objective: minimize composite objective function:

$$\underset{x}{\text{minimize}} \ \frac{1}{n}\sum_{i=1}^{n} f_i(x) + \|x\|_1$$

where f_i is smooth (and $\|\cdot\|_1$ is not). For simplicity we consider the nonsmooth term to be ℓ_1 norm, but this is general to any convex function for which we have access to its proximal operator.

(Prox)SAGA

The ProxSAGA update is inefficient

$$x^+ = \underbrace{\mathrm{prox}_{\gamma h}}_{\text{dense!}}(x - \gamma(\underbrace{\nabla f_i(x)}_{\text{sparse}} - \underbrace{\alpha_i}_{\text{sparse}} + \underbrace{\overline{\alpha}}_{\text{dense!}})); \; \alpha_i^+ = \nabla f_i(x)$$

$\Longrightarrow$ a sparse variant is needed as a prerequisite for a practical parallel method.

Sparse Proximal SAGA

Sparse Proximal SAGA. (Pedregosa, Leblond, and Lacoste-Julien 2017)
Extension of Sparse SAGA to composite optimization problems

Sparse Proximal SAGA

Sparse Proximal SAGA. (Pedregosa, Leblond, and Lacoste-Julien 2017)
Extension of Sparse SAGA to composite optimization problems

Like SAGA, it relies on unbiased gradient estimate

$$v_i = \nabla f_i(x) - \alpha_i + DP_i\overline{\alpha}\,;$$

Sparse Proximal SAGA

Sparse Proximal SAGA. (Pedregosa, Leblond, and Lacoste-Julien 2017)
Extension of Sparse SAGA to composite optimization problems

Like SAGA, it relies on unbiased gradient estimate and proximal step

$$v_i = \nabla f_i(x) - \alpha_i + DP_i \overline{\alpha}\,;\; x^+ = \mathrm{prox}_{\gamma\varphi_i}(x - \gamma v_i)\,;\; \alpha_i^+ = \nabla f_i(x)$$

Sparse Proximal SAGA

Sparse Proximal SAGA. (Pedregosa, Leblond, and Lacoste-Julien 2017) Extension of Sparse SAGA to composite optimization problems

Like SAGA, it relies on unbiased gradient estimate and proximal step

$$v_i = \nabla f_i(x) - \alpha_i + DP_i\overline{\alpha}\,;\; x^+ = \mathbf{prox}_{\gamma\varphi_i}(x - \gamma v_i)\,;\; \alpha_i^+ = \nabla f_i(x)$$

Where P_i, D are as in Sparse SAGA and $\varphi_i \overset{\text{def}}{=} \sum_j^d (P_i D)_{i,i} |x_j|$.

φ_i has two key properties: *i*) support of φ_i = support of ∇f_i (sparse updates) and *ii*) $\mathbb{E}_i[\varphi_i] = \|x\|_1$ (unbiasedness)

Sparse Proximal SAGA

Sparse Proximal SAGA. (Pedregosa, Leblond, and Lacoste-Julien 2017) Extension of Sparse SAGA to composite optimization problems

Like SAGA, it relies on unbiased gradient estimate and proximal step

$$v_i = \nabla f_i(x) - \alpha_i + DP_i\overline{\alpha}\,;\; x^+ = \mathbf{prox}_{\gamma\varphi_i}(x - \gamma v_i)\,;\; \alpha_i^+ = \nabla f_i(x)$$

Where P_i, D are as in Sparse SAGA and $\varphi_i \overset{\text{def}}{=} \sum_j^d (P_i D)_{i,i} |x_j|$.

φ_i has two key properties: *i*) support of φ_i = support of ∇f_i (sparse updates) and *ii*) $\mathbb{E}_i[\varphi_i] = \|x\|_1$ (unbiasedness)

Convergence: same linear convergence rate as SAGA, with cheaper updates in presence of sparsity.

Proximal Asynchronous SAGA (ProxASAGA)

Each core runs Sparse Proximal SAGA asynchronously without locks and updates x, α and $\overline{\alpha}$ in shared memory.

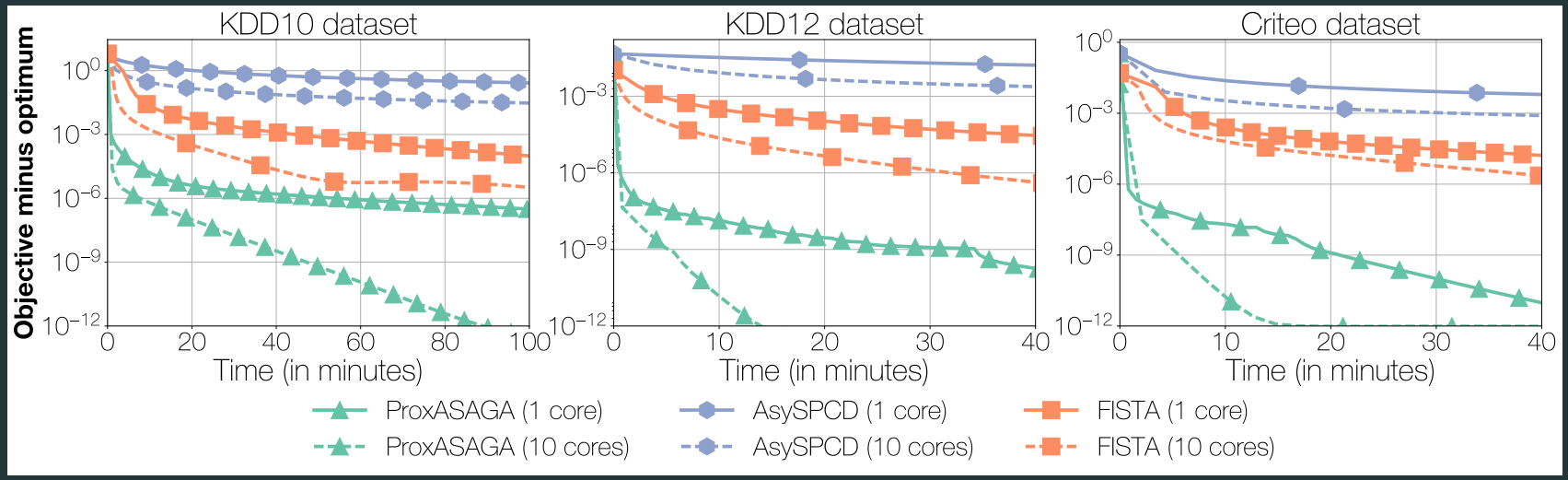
🔀 All read/write operations to shared memory are *inconsistent*, i.e., no performance destroying vector-level locks while reading/writing.

Convergence: under sparsity assumptions, ProxASAGA converges with the same rate as the sequential algorithm $\implies$ theoretical linear speedup with respect to the number of cores.

Empirical results

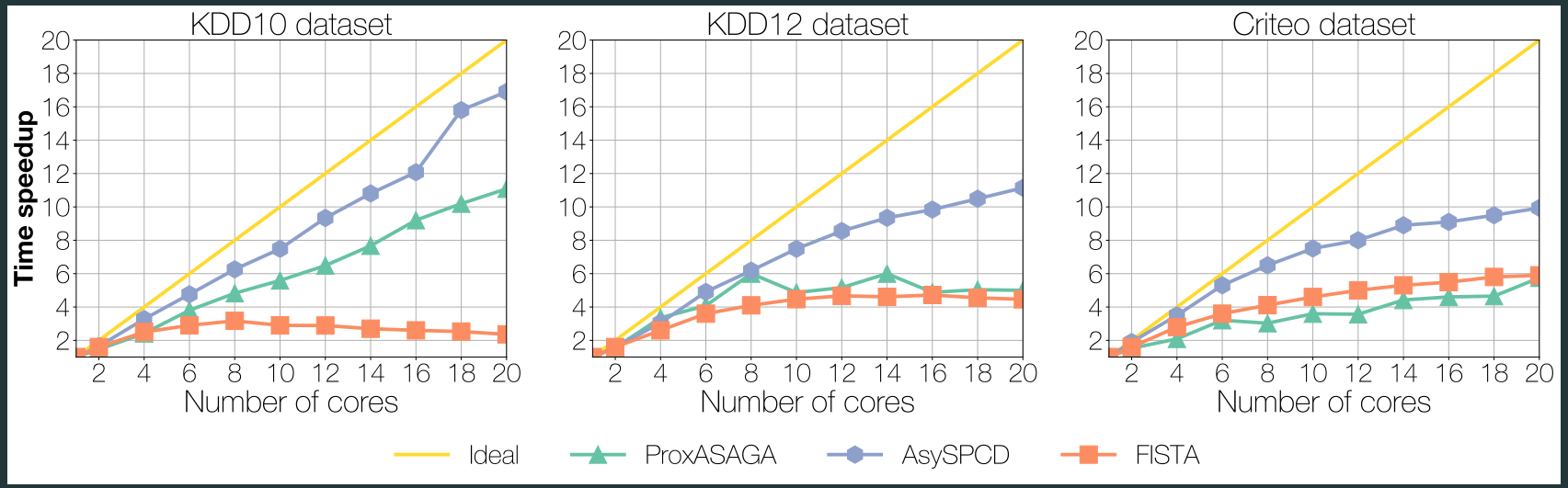
ProxASAGA vs competing methods on 3 large-scale datasets, ℓ_1-regularized logistic regression

Dataset	n	p	density	L	Δ
KDD 2010	19,264,097	1,163,024	10^{-6}	28.12	0.15
KDD 2012	149,639,105	54,686,452	2×10^{-7}	1.25	0.85
Criteo	45,840,617	1,000,000	4×10^{-5}	1.25	0.89



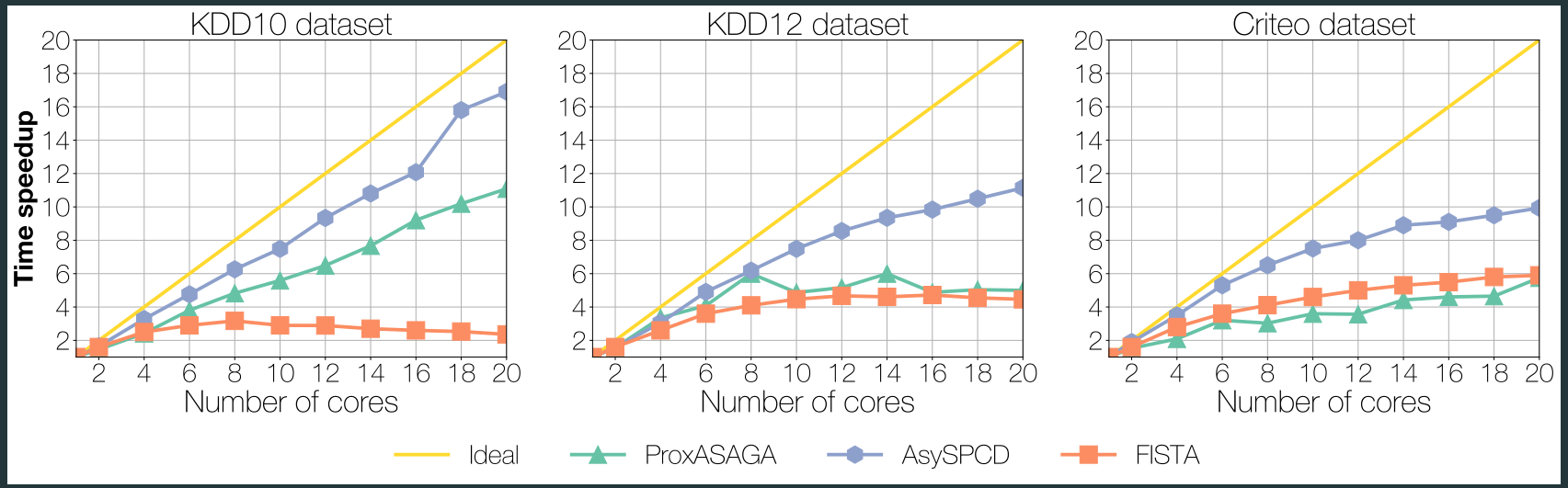
Empirical results - Speedup

$$\text{Speedup} = \frac{\text{Time to } 10^{-10} \text{ suboptimality on one core}}{\text{Time to same suboptimality on } k \text{ cores}}$$



Empirical results - Speedup

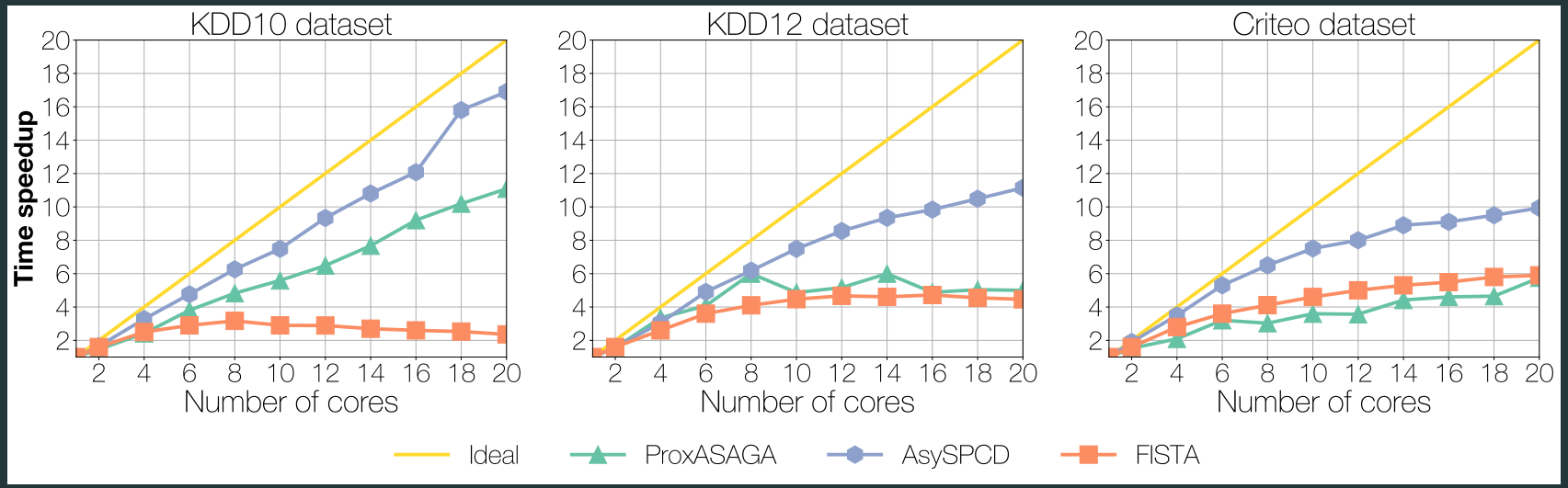
$$\text{Speedup} = \frac{\text{Time to } 10^{-10} \text{ suboptimality on one core}}{\text{Time to same suboptimality on } k \text{ cores}}$$



- ProxASAGA achieves speedups between 6x and 12x on a 20 cores architecture.

Empirical results - Speedup

$$\text{Speedup} = \frac{\text{Time to } 10^{-10} \text{ suboptimality on one core}}{\text{Time to same suboptimality on } k \text{ cores}}$$



- ProxASAGA achieves speedups between 6x and 12x on a 20 cores architecture.
- As predicted by theory, there is a high correlation between degree of sparsity and speedup.

Perspectives

- Scale above 20 cores.
- Asynchronous optimization on the GPU.
- Acceleration.
- Software development.

Codes

Code is in github: `https://github.com/fabianp/ProxASAGA`.

Computational code is C++ (use of atomic type) but wrapped in Python.

A very efficient implementation of SAGA can be found in the scikit-learn and lightning (`https://github.com/scikit-learn-contrib/lightning`) libraries.

References

- Cauchy, Augustin (1847). “Méthode générale pour la résolution des systemes d’équations simultanées”. In: *Comp. Rend. Sci. Paris*.
- Defazio, Aaron, Francis Bach, and Simon Lacoste-Julien (2014). “SAGA: A fast incremental gradient method with support for non-strongly convex composite objectives”. In: *Advances in Neural Information Processing Systems*.
- Glowinski, Roland and A Marroco (1975). “Sur l’approximation, par éléments finis d’ordre un, et la résolution, par pénalisation-dualité d’une classe de problèmes de Dirichlet non linéaires”. In: *Revue française d’automatique, informatique, recherche opérationnelle. Analyse numérique*.
- Jaggi, Martin et al. (2014). “Communication-Efficient Distributed Dual Coordinate Ascent”. In: *Advances in Neural Information Processing Systems 27*.
- Leblond, Rémi, Fabian P., and Simon Lacoste-Julien (2017). “ASAGA: asynchronous parallel SAGA”. In: *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics (AISTATS 2017)*.
- Niu, Feng et al. (2011). “Hogwild: A lock-free approach to parallelizing stochastic gradient descent”. In: *Advances in Neural Information Processing Systems*.


Pedregosa, Fabian, Rémi Leblond, and Simon Lacoste-Julien (2017). “Breaking the Nonsmooth Barrier: A Scalable Parallel Method for Composite Optimization”. In: *Advances in Neural Information Processing Systems 30*.


Peng, Zhimin et al. (2016). “ARock: an algorithmic framework for asynchronous parallel coordinate updates”. In: *SIAM Journal on Scientific Computing*.


Robbins, Herbert and Sutton Monro (1951). “A Stochastic Approximation Method”. In: *Ann. Math. Statist.*


Shamir, Ohad, Nati Srebro, and Tong Zhang (2014). “Communication-efficient distributed optimization using an approximate newton-type method”. In: *International conference on machine learning*.

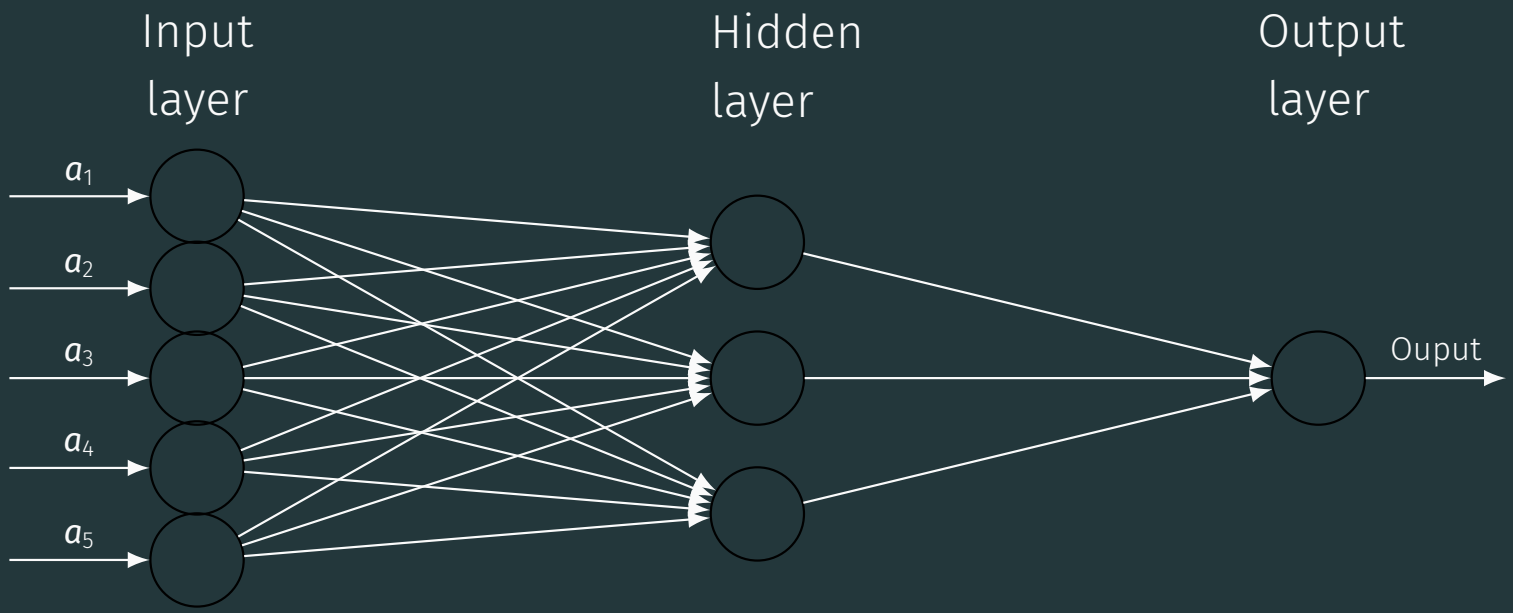
Supervised Machine Learning

Data: n observations $(a_i, b_i) \in \mathbb{R}^p \times \mathbb{R}$

Prediction function: $h(a, x) \in \mathbb{R}$

Motivating examples:

- Linear prediction: $h(a, x) = x^T a$
- Neural networks: $h(a, x) = x_m^T \sigma(x_{m-1} \sigma(\cdots x_2^T \sigma(x_1^T a))$



Supervised Machine Learning

Data: n observations $(a_i, b_i) \in \mathbb{R}^p \times \mathbb{R}$

Prediction function: $h(a, x) \in \mathbb{R}$

Motivating examples:

- Linear prediction: $h(a, x) = x^T a$
- Neural networks: $h(a, x) = x_m^T \sigma(x_{m-1} \sigma(\cdots x_2^T \sigma(x_1^T a))$

Minimize some distance (e.g., quadratic) between the prediction

$$\underset{x}{\text{minimize}} \frac{1}{n} \sum_{i=1}^{n} \ell(b_i, h(a_i, x)) \overset{\text{notation}}{=} \frac{1}{n} \sum_{i=1}^{n} f_i(x)$$

where popular examples of ℓ are

- Squared loss, $\ell(b_i, h(a_i, x)) \overset{\text{def}}{=} (b_i - h(a_i, x))^2$
- Logistic (softmax), $\ell(b_i, h(a_i, x)) \overset{\text{def}}{=} \log(1 + \exp(-b_i h(a_i, x)))$

Sparse Proximal SAGA

For step size $\gamma = \frac{1}{5L}$ and f μ-strongly convex ($\mu > 0$), Sparse Proximal SAGA converges geometrically in expectation. At iteration t we have

$$\mathbb{E}\|x_t - x^*\|^2 \leq (1 - \tfrac{1}{5}\min\{\tfrac{1}{n}, \tfrac{1}{\kappa}\})^t C_0 \ ,$$

with $C_0 = \|x_0 - x^*\|^2 + \frac{1}{5L^2}\sum_{i=1}^n \|\alpha_i^0 - \nabla f_i(x^*)\|^2$ and $\kappa = \frac{L}{\mu}$ (condition number).

Implications

- Same convergence rate than SAGA with cheaper updates.
  - In the "big data regime" ($n \geq \kappa$): rate in $\mathcal{O}(1/n)$.
  - In the "ill-conditioned regime" ($n \leq \kappa$): rate in $\mathcal{O}(1/\kappa)$.
- Adaptivity to strong convexity, i.e., no need to know strong convexity parameter to obtain linear convergence.

Convergence ProxASAGA

Suppose $\tau \leq \frac{1}{10\sqrt{\Delta}}$. Then:

- If $\kappa \geq n$, then with step size $\gamma = \frac{1}{36L}$, ProxASAGA converges geometrically with rate factor $\Omega(\frac{1}{\kappa})$.
- If $\kappa < n$, then using the step size $\gamma = \frac{1}{36n\mu}$, ProxASAGA converges geometrically with rate factor $\Omega(\frac{1}{n})$.

In both cases, the convergence rate is the same as Sparse Proximal SAGA $\implies$ ProxASAGA is **linearly faster** up to constant factor. In both cases the **step size does not depend on** τ.

If $\tau \leq 6\kappa$, a universal step size of $\Theta(\frac{1}{L})$ achieves a similar rate than Sparse Proximal SAGA, making it adaptive to local strong convexity (knowledge of κ not required).

ASAGA algorithm

Algorithm 1 ASAGA (analyzed algorithm)

Initialize shared variables x and $(\alpha_i)_{i=1}^n$
keep doing in parallel
  $\hat{x} =$ inconsistent read of x
  $\forall j$, $\hat{\alpha}_j =$ inconsistent read of α_j
  Sample i uniformly at random in $\{1, ..., n\}$
  Let S_i be f_i's support
  $[\bar{\alpha}]_{S_i} = {}^1/_n \sum_{k=1}^n [\hat{\alpha}_k]_{S_i}$
  $[\delta x]_{S_i} = -\gamma(f_i'(\hat{x}) - \hat{\alpha}_i + D_i[\bar{\alpha}]_{S_i})$

  for v **in** S_i **do**
    $[x]_v \leftarrow [x]_v + [\delta x]_v$ // atomic
    $[\alpha_i]_v \leftarrow [f_i'(\hat{x})]_v$
    // ('$\leftarrow$' denotes a shared memory update.)
  end for
end parallel loop

Algorithm 2 ASAGA (implementation)

Initialize shared variables x, $(\alpha_i)_{i=1}^n$ and $\bar{\alpha}$
keep doing in parallel
  Sample i uniformly at random in $\{1, ..., n\}$
  Let S_i be f_i's support
  $[\hat{x}]_{S_i} =$ inconsistent read of x on S_i
  $\hat{\alpha}_i =$ inconsistent read of α_i
  $[\bar{\alpha}]_{S_i} =$ inconsistent read of $\bar{\alpha}$ on S_i
  $[\delta\alpha]_{S_i} = f_i'([\hat{x}]_{S_i}) - \hat{\alpha}_i$
  $[\delta x]_{S_i} = -\gamma([\delta\alpha]_{S_i} + D_i[\bar{\alpha}]_{S_i})$

  for v **in** S_i **do**
    $[x]_v \leftarrow [x]_v + [\delta x]_v$ // atomic
    $[\alpha_i]_v \leftarrow [\alpha_i]_v + [\delta\alpha]_v$ // atomic
    $[\bar{\alpha}]_v \leftarrow [\bar{\alpha}]_v + {}^1/_n[\delta\alpha]_v$ // atomic
  end for
end parallel loop

ProxASAGA algorithm

Algorithm 1 PROXASAGA (analyzed)

Initialize shared variables $\boldsymbol{x}$ and $(\boldsymbol{\alpha}_i)_{i=1}^n$
keep doing in parallel
  $\hat{\boldsymbol{x}} =$ inconsistent read of $\boldsymbol{x}$
  $\hat{\boldsymbol{\alpha}} =$ inconsistent read of $\boldsymbol{\alpha}$
  Sample i uniformly in $\{1, ..., n\}$
  $S_i :=$ support of ∇f_i
  $T_i :=$ extended support of ∇f_i in $\mathcal{B}$
  $[\overline{\boldsymbol{\alpha}}]_{T_i} = {}^1\!/_n \sum_{j=1}^n [\hat{\boldsymbol{\alpha}}_j]_{T_i}$
  $[\delta\boldsymbol{\alpha}]_{S_i} = [\nabla f_i(\hat{\boldsymbol{x}})]_{S_i} - [\hat{\boldsymbol{\alpha}}_i]_{S_i}$
  $[\hat{\boldsymbol{v}}]_{T_i} = [\delta\boldsymbol{\alpha}]_{T_i} + [\boldsymbol{D}_i\overline{\boldsymbol{\alpha}}]_{T_i}$
  $[\delta\boldsymbol{x}]_{T_i} = [\mathbf{prox}_{\gamma\varphi_i}(\hat{\boldsymbol{x}} - \gamma\hat{\boldsymbol{v}})]_{T_i} - [\hat{\boldsymbol{x}}]_{T_i}$
  for B **in** T_i **do**
    for $b \in B$ **do**
      $[\boldsymbol{x}]_b \leftarrow [\boldsymbol{x}]_b + [\delta\boldsymbol{x}]_b$ ▷ atomic
      if $b \in S_i$ **then**
        $[\boldsymbol{\alpha}_i]_b \leftarrow [\nabla f_i(\hat{\boldsymbol{x}})]_b$
      end if
    end for
  end for
  // ('$\leftarrow$' denotes shared memory update.)
end parallel loop

Algorithm 2 PROXASAGA (implemented)

Initialize shared variables $\boldsymbol{x}$, $(\boldsymbol{\alpha}_i)_{i=1}^n$, $\overline{\boldsymbol{\alpha}}$
keep doing in parallel
  Sample i uniformly in $\{1, ..., n\}$
  $S_i :=$ support of ∇f_i
  $T_i :=$ extended support of ∇f_i in $\mathcal{B}$
  $[\hat{\boldsymbol{x}}]_{T_i} =$ inconsistent read of $\boldsymbol{x}$ on T_i
  $\hat{\boldsymbol{\alpha}}_i =$ inconsistent read of $\boldsymbol{\alpha}_i$
  $[\overline{\boldsymbol{\alpha}}]_{T_i} =$ inconsistent read of $\overline{\boldsymbol{\alpha}}$ on T_i
  $[\delta\boldsymbol{\alpha}]_{S_i} = [\nabla f_i(\hat{\boldsymbol{x}})]_{S_i} - [\hat{\boldsymbol{\alpha}}_i]_{S_i}$
  $[\hat{\boldsymbol{v}}]_{T_i} = [\delta\boldsymbol{\alpha}]_{T_i} + [\boldsymbol{D}_i\overline{\boldsymbol{\alpha}}]_{T_i}$
  $[\delta\boldsymbol{x}]_{T_i} = [\mathbf{prox}_{\gamma\varphi_i}(\hat{\boldsymbol{x}} - \gamma\hat{\boldsymbol{v}})]_{T_i} - [\hat{\boldsymbol{x}}]_{T_i}$
  for B **in** T_i **do**
    for b **in** B **do**
      $[\boldsymbol{x}]_b \leftarrow [\boldsymbol{x}]_b + [\delta\boldsymbol{x}]_b$ ▷ atomic
      if $b \in S_i$ **then**
        $[\overline{\boldsymbol{\alpha}}]_b \leftarrow [\overline{\boldsymbol{\alpha}}]_b + {}^1\!/_n[\delta\boldsymbol{\alpha}]_b$ ▷ atomic
      end if
    end for
  end for
  $\boldsymbol{\alpha}_i \leftarrow \nabla f_i(\hat{\boldsymbol{x}})$ (scalar update) ▷ atomic
end parallel loop

Atomic vs non-atomic