Parallel Optimization in Machine Learning

Fabian Pedregosa

December 19, 2017 Huawei Paris Research Center
About me

• Engineer (2010-2012), Inria Saclay (scikit-learn kickstart).
• PhD (2012-2015), Inria Saclay.
• 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 algorithms can be divided into two large categories: **synchronous** and **asynchronous**.

**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.

Image credits: (Peng et al. 2016)
Outline

Synchronous methods

• Synchronous (stochastic) gradient descent.

Asynchronous methods

• Asynchronous stochastic gradient descent (Hogwild) (Niu et al. 2011)
• 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.
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

\[
\minimize_x f(x) \overset{\text{def}}{=} \frac{1}{n} \sum_{i=1}^{n} f_i(x)
\]

**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 \( n_1, \ldots n_k \) that sum to \( n \).
2. Distribute computation of \( \nabla f(x) \) among \( k \) nodes
   \[
   \nabla f(x) = \frac{1}{n} \sum_{i=1}^{n} \nabla f_i(x)
   = \frac{1}{k} \left( \frac{1}{n_1} \sum_{i=1}^{n_1} \nabla f_i(x) + \ldots + \frac{1}{n_k} \sum_{i=n_{k-1}+1}^{n_k} \nabla f_i(x) \right)
   \]
   done by worker 1
   done by worker \( k \)
3. Perform the gradient descent update by a master node
   \[
   x^+ = x - \gamma \nabla f(x)
   \]
1. Choose $n_1, \ldots, n_k$ that sum to $n$.
2. Distribute computation of $\nabla f(x)$ among $k$ nodes

\[
\nabla f(x) = \frac{1}{n} \sum_{i=1}^{n} \nabla f_i(x)
\]

\[
= \frac{1}{k} \left( \frac{1}{n_1} \sum_{i=1}^{n_1} \nabla f_i(x) + \ldots + \frac{1}{n_k} \sum_{i=n_{k-1}+1}^{n_k} \nabla f_i(x) \right)
\]

\{ done by worker 1 \quad \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 $\nabla f_{i_t}$ 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 $\nabla f_{i_t}$ 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?

YOU CAN’T HAVE A SYNCHRONIZATION BOTTLENECK

IF YOU DON’T SYNCHRONIZE
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 can be very fast. But it's 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!
Simple things become counter-intuitive, e.g., how to **name** the iterates?

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.

\[ \hat{x}_t = (t + 1) \text{-th successful update to shared memory.} \]

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

\[ \therefore \hat{x}_t \text{ and } i_t \text{ are not necessarily independent.} \]
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 $i$

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

uniform sampling $\Rightarrow \sum_{i=1}^{n} \frac{1}{n} \nabla f_i(x) = \nabla f(x)$
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 $\nabla f_1$ is much expensive than $\nabla f_2$. 
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 $\nabla f_1$ is much expensive than $\nabla f_2$.

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

$$\mathbb{E}_i [\nabla f_i] = \frac{1}{4} f_1 + \frac{3}{4} f_2$$

$$\neq \frac{1}{2} f_1 + \frac{1}{2} f_2 !!
The Art of Naming Things
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 $\nabla f_i$.

Asynchronous SAGA
The SAGA algorithm

Setting:

\[
\minimize_{\mathbf{x}} \frac{1}{n} \sum_{i=1}^{n} f_i(\mathbf{x})
\]

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

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

\[
\mathbf{x}^+ = \mathbf{x} - \gamma (\nabla f_i(\mathbf{x}) - \mathbf{\alpha}_i + \overline{\mathbf{\alpha}}) ; \quad \mathbf{\alpha}_i^+ = \nabla f_i(\mathbf{x})
\]

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

Setting:

\[
\min_x \frac{1}{n} \sum_{i=1}^{n} f_i(x)
\]

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

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

\[
x^+ = x - \gamma (\nabla f_i(x) - \alpha_i + \bar{\alpha}) \quad ; \quad \alpha_i^+ = \nabla f_i(x)
\]

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

Super easy to use in scikit-learn

```python
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 $\nabla f_i$ are sparse too.
- Asynchronous algorithms work best when updates are sparse.

SAGA update is inefficient for sparse data

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

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

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

$$x^+ = x - \gamma (\nabla f_i(x) - \alpha_i + P_i D \bar{\alpha}); \quad \alpha_{i}^+ = \nabla f_i(x)$$

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

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

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

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

• 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
Previous methods assume objective function is smooth. Cannot be applied to Lasso, Group Lasso, box constraints, etc.

**Objective:** minimize composite objective function:

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

where $f_i$ is smooth (and $\| \cdot \|_1$ is not). For simplicity we consider the nonsmooth term to be $\ell_1$ norm, but this is general to any convex function for which we have access to its proximal operator.
The ProxSAGA update is inefficient

\[ x^+ = \text{prox}_{\gamma h}(x - \gamma (\nabla f_i(x) - \alpha_i + \overline{\alpha})) ; \quad \alpha_i^+ = \nabla f_i(x) \]

\[ \implies \text{a sparse variant is needed as a prerequisite for a practical parallel method.} \]
Sparse Proximal SAGA. (Pedregosa, Leblond, and Lacoste-Julien 2017)
Extension of Sparse SAGA to composite optimization problems

$$v_i = \nabla f_i(x) + D_P; x_{i+1} = \text{prox}_{\phi_i}(x_i + v_i)$$

Where $P, D$ are as in Sparse SAGA and $\phi_i$ def $= \sum d_j (P D)$.

$\phi_i$ has two key properties:
1. $\text{support of } \phi_i = \text{support of } \nabla f_i$ (sparse updates)
2. $E_i[\phi_i] = \|x_i\|_1$ (unbiasedness)

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

\[ v_i = \nabla f_i(x) - \alpha_i + DP_i \bar{\alpha}; \]
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

\[ v_i = \nabla f_i(x) - \alpha_i + DP_i \alpha; \quad x^+ = \text{prox}_{\gamma \phi_i}(x - \gamma v_i); \quad \alpha_i^+ = \nabla f_i(x) \]
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

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

Where $P_i, D$ are as in Sparse SAGA and $\varphi_i \overset{\text{def}}{=} \sum_j^d (P_i D)_{i,j} |x_j|$. $\varphi_i$ has two key properties: i) support of $\varphi_i = \text{support of } \nabla f_i$ (sparse updates) and ii) $\mathbb{E}[\varphi_i] = \|x\|_1$ (unbiasedness)
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

\[ v_i = \nabla f_i(x) - \alpha_i + DP_i \bar{\alpha} \; ; \; x^+ = \text{prox}_{\gamma \varphi_i}(x - \gamma v_i) \; ; \; \alpha_i^+ = \nabla f_i(x) \]

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

\( \varphi_i \) has two key properties: i) support of \( \varphi_i = \) support of \( \nabla f_i \) (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 $\mathbf{x}$, $\mathbf{\alpha}$ and $\mathbf{\overline{\alpha}}$ 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, $\ell_1$-regularized logistic regression

<table>
<thead>
<tr>
<th>Dataset</th>
<th>n</th>
<th>p</th>
<th>density</th>
<th>$L$</th>
<th>$\Delta$</th>
</tr>
</thead>
<tbody>
<tr>
<td>KDD 2010</td>
<td>19,264,097</td>
<td>1,163,024</td>
<td>$10^{-6}$</td>
<td>28.12</td>
<td>0.15</td>
</tr>
<tr>
<td>KDD 2012</td>
<td>149,639,105</td>
<td>54,686,452</td>
<td>$2 \times 10^{-7}$</td>
<td>1.25</td>
<td>0.85</td>
</tr>
<tr>
<td>Criteo</td>
<td>45,840,617</td>
<td>1,000,000</td>
<td>$4 \times 10^{-5}$</td>
<td>1.25</td>
<td>0.89</td>
</tr>
</tbody>
</table>
Empirical results - Speedup

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.
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

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.
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


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: \( h(a, x) = x_m^T \sigma(x_{m-1}^T \sigma(\cdots x_2^T \sigma(x_1^T a))) \)
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: $h(a, x) = x^Ta$
- Neural networks: $h(a, x) = x^T_m \sigma(x_{m-1} \sigma(\cdots x^T_2 \sigma(x^T_1 a)))$

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

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

where popular examples of $\ell$ 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 $f$ $\mu$-strongly convex ($\mu > 0$), Sparse Proximal SAGA converges geometrically in expectation. At iteration $t$ we have

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

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

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 $\tau$.

If $\tau \leq 6\kappa$, a universal step size of $\Theta\left(\frac{1}{L}\right)$ achieves a similar rate than Sparse Proximal SAGA, making it adaptive to local strong convexity (knowledge of $\kappa$ not required).
Algorithm 1 ASAGA (analyzed algorithm)
1: Initialize shared variables $x$ and $(\alpha_i)_{i=1}^n$
2: keep doing in parallel
3: $\hat{x} =$ inconsistent read of $x$
4: $\forall j, \hat{\alpha}_j =$ inconsistent read of $\alpha_j$
5: Sample $i$ uniformly at random in $\{1, \ldots, n\}$
6: Let $S_i$ be $f_i$’s support
7: $[\hat{\alpha}]_{S_i} = \frac{1}{n} \sum_{k=1}^n [\hat{\alpha}_k]_{S_i}$
8: $[\delta x]_{S_i} = -\gamma (f'_i(\hat{x}) - \hat{\alpha}_i + D_i[\hat{\alpha}]_{S_i})$
9:  
10: for $v \in S_i$ do
11: $[x]_v \leftarrow [x]_v + [\delta x]_v$ \hspace{1em} // atomic
12: $[\alpha_i]_v \leftarrow [f'_i(\hat{x})]_v$
13: // (‘$\leftarrow$’ denotes a shared memory update.)
14: end for
15: end parallel loop

Algorithm 2 ASAGA (implementation)
1: Initialize shared variables $x$, $(\alpha_i)_{i=1}^n$ and $\bar{\alpha}$
2: keep doing in parallel
3: Sample $i$ uniformly at random in $\{1, \ldots, n\}$
4: Let $S_i$ be $f_i$’s support
5: $[\hat{x}]_{S_i} =$ inconsistent read of $x$ on $S_i$
6: $\hat{\alpha}_i =$ inconsistent read of $\alpha_i$
7: $[\bar{\alpha}]_{S_i} =$ inconsistent read of $\bar{\alpha}$ on $S_i$
8: $[\delta \alpha]_{S_i} = f'_i([\hat{x}]_{S_i}) - \hat{\alpha}_i$
9: $[\delta x]_{S_i} = -\gamma ([\delta \alpha]_{S_i} + D_i[\bar{\alpha}]_{S_i})$
10: for $v \in S_i$ do
11: $[x]_v \leftarrow [x]_v + [\delta x]_v$ \hspace{1em} // atomic
12: $[\alpha_i]_v \leftarrow [\alpha_i]_v + [\delta \alpha]_v$ \hspace{1em} // atomic
13: $[\bar{\alpha}]_v \leftarrow [\bar{\alpha}]_v + 1/n[\delta \alpha]_v$ \hspace{1em} // atomic
14: end for
15: end parallel loop
Algorithm 1 ProxASAGA (analyzed)

1: Initialize shared variables \( \mathbf{x} \) and \( (\mathbf{\alpha}_i)_{i=1}^n \)
2: keep doing in parallel
3: \( \mathbf{\hat{x}} = \) inconsistent read of \( \mathbf{x} \)
4: \( \mathbf{\hat{\alpha}} = \) inconsistent read of \( \mathbf{\alpha} \)
5: Sample \( i \) uniformly in \( \{1, \ldots, n\} \)
6: \( S_i := \) support of \( \nabla f_i \)
7: \( T_i := \) extended support of \( \nabla f_i \) in \( \mathcal{B} \)
8: \[ \alpha_{T_i} = \frac{1}{n} \sum_{j=1}^n \alpha_{j_{T_i}} \]
9: \[ \delta \alpha_{S_i} = [\nabla f_i(\mathbf{\hat{x}})]_{S_i} - [\mathbf{\hat{\alpha}}]_{S_i} \]
10: \[ \mathbf{\hat{\nu}}_{T_i} = \delta \alpha_{T_i} + [D_i \mathbf{\alpha}]_{T_i} \]
11: \[ \delta \mathbf{x}_{T_i} = [\text{prox}_{[\mathbf{\gamma} \mathbf{\nu}_{T_i}]}(\mathbf{\hat{x}} - \gamma \mathbf{\hat{\nu}})]_{T_i} - [\mathbf{x}]_{T_i} \]
12: for \( B \) in \( T_i \) do
13: for \( b \in B \) do
14: \[ [\mathbf{x}]_b \leftarrow [\mathbf{x}]_b + [\delta \mathbf{x}]_b \quad \triangleright \text{atomic} \]
15: if \( b \in S_i \) then
16: \[ [\mathbf{\alpha}_i]_b \leftarrow [\nabla f_i(\mathbf{\hat{x}})]_b \]
17: end if
18: end for
19: end for
20: // (\( \leftarrow \)) denotes shared memory update.
21: end parallel loop

Algorithm 2 ProxASAGA (implemented)

1: Initialize shared variables \( \mathbf{x}, (\mathbf{\alpha}_i)_{i=1}^n, \mathbf{\alpha} \)
2: keep doing in parallel
3: Sample \( i \) uniformly in \( \{1, \ldots, n\} \)
4: \( S_i := \) support of \( \nabla f_i \)
5: \( T_i := \) extended support of \( \nabla f_i \) in \( \mathcal{B} \)
6: \[ [\mathbf{\hat{x}}]_{T_i} = \) inconsistent read of \( \mathbf{x} \) on \( T_i \)
7: \( \mathbf{\hat{\alpha}}_i = \) inconsistent read of \( \mathbf{\alpha}_i \)
8: \[ [\mathbf{\alpha}]_{T_i} = \) inconsistent read of \( \mathbf{\alpha} \) on \( T_i \)
9: \[ [\delta \mathbf{\alpha}]_{S_i} = [\nabla f_i(\mathbf{\hat{x}})]_{S_i} - [\mathbf{\hat{\alpha}}]_{S_i} \]
10: \[ \mathbf{\hat{\nu}}_{T_i} = [\delta \alpha]_{T_i} + [D_i \mathbf{\alpha}]_{T_i} \]
11: \[ [\delta \mathbf{x}]_{T_i} = [\text{prox}_{[\mathbf{\gamma} \mathbf{\nu}_{T_i}]}(\mathbf{\hat{x}} - \gamma \mathbf{\hat{\nu}})]_{T_i} - [\mathbf{x}]_{T_i} \]
12: for \( B \) in \( T_i \) do
13: for \( b \in B \) do
14: \[ [\mathbf{x}]_b \leftarrow [\mathbf{x}]_b + [\delta \mathbf{x}]_b \quad \triangleright \text{atomic} \]
15: if \( b \in S_i \) then
16: \[ [\mathbf{\alpha}]_b \leftarrow [\mathbf{\alpha}]_b + \frac{1}{n} [\delta \mathbf{\alpha}]_b \quad \triangleright \text{atomic} \]
17: end if
18: end for
19: end for
20: \( \mathbf{\alpha}_i \leftarrow \nabla f_i(\mathbf{\hat{x}}) \) (scalar update) \quad \triangleright \text{atomic}
21: end parallel loop
Atomic vs non-atomic