ASAGA: Asynchronous Parallel SAGA

Rémi Leblond
INRIA - Sierra team
École normale supérieure, Paris

Fabian Pedregosa
INRIA - Sierra team
ENS, Paris

Simon Lacoste-Julien
Department of CS & OR (DIRO)
Université de Montréal, Montréal

Abstract

We describe ASAGA, a sparse asynchronous parallel version of the incremental gradient algorithm SAGA that enjoys fast linear convergence rates. Through a novel perspective, we revisit and clarify a subtle but important technical issue present in most recent convergence rate proofs for asynchronous parallel optimization, and propose a simplification of the recently introduced "perturbed iterate" framework that resolves it. We prove that ASAGA can obtain a theoretical linear speedup on multi-core systems even without sparsity. We present empirical results on a 40-core machine illustrating the practical speedup as well as the hardware overhead.

1 Introduction

We consider the unconstrained optimization problem of minimizing a *finite sum* of smooth convex functions:

$$\min_{x \in \mathbb{R}^d} f(x), \quad f(x) := \frac{1}{n} \sum_{i=1}^{n} f_i(x), \tag{1}$$

where each f_i is assumed to be convex with L-Lipschitz continuous gradient, f is μ-strongly convex and n is large. We define a condition number for this problem as $\kappa := L/\mu$. A flurry of randomized incremental algorithms have recently been proposed to solve (1) with a fast linear convergence rate, such as SAG [7], SDCA [16], SVRG [6] and SAGA [2]. These algorithms can be interpreted as variance reduced versions of the popular stochastic gradient descent (SGD) algorithm, and they have demonstrated both theoretical and practical improvements over SGD (for the *finite sum* problem (1)).

To take advantage of modern multi-core computers, these algorithms need to be adapted to the asynchronous parallel setting. Much work has been devoted recently in proposing asynchronous parallel variants of algorithms such as SGD [14], SDCA [5] and SVRG [15, 13, 17]. Among the incremental gradient algorithms with fast linear convergence rates that can optimize (1) in its general form, only SVRG has had an asynchronous parallel version proposed. We propose one for SAGA, arguably a more natural candidate as it is not epoch-based and thus has no synchronization barriers (see [8] for a full version of this paper).

Related Work. An asynchronous variant of SGD called HOGWILD was presented by Niu et al. [14]; part of their framework of analysis was re-used by most of the recent literature on asynchronous parallel optimization algorithms with convergence rates [11, 5, 1, 10, 3, 15, 17]. These papers use an unbiased gradient assumption that is not consistent with their proof technique (see Section 3.2). The "perturbed iterate" framework presented in [13] is to the best of our knowledge the only one that does not suffer from this problem. Our convergence analysis builds heavily from their approach, while simplifying it. In particular, the authors assumed that f was both strongly convex and had a bound on the gradient, two *inconsistent* assumptions in the unconstrained setting they analyzed. We overcome these issues through tighter analysis and we obtain linear speedups under weaker conditions. We also propose a more convenient way to label the iterates (see Section 3.2). Reddi et al. [15] presents a hybrid algorithm called HSAG that includes SAGA and SVRG as special cases. Their analysis is epoch-based though, thus does not handle a fully asynchronous version of SAGA as we do. Moreover, they do not propose an efficient sparse implementation for SAGA, in contrast with ASAGA.

29th Conference on Neural Information Processing Systems (NIPS 2016), Barcelona, Spain.

Notation. We denote by $\mathbb{E}$ a full expectation with respect to all the randomness, and by $\mathbf{E}$ the *conditional* expectation of a random i (the index of the factor f_i chosen in SGD and other algorithms), conditioned on all the past, where "past" will be clear from the context.

2 Sparse SAGA

Original SAGA Algorithm. We borrow our notation from Hofmann et al. [4]. The standard SAGA algorithm [2] maintains two moving quantities to optimize (1): the current iterate x and a table (memory) of historical gradients $(\alpha_i)_{i=1}^n$.[1] At every iteration, the SAGA algorithm samples uniformly at random an index $i \in \{1, \dots, n\}$, and then executes the following update on x and α:

$$x^+ = x - \gamma\big(f_i'(x) - \alpha_i + \bar{\alpha}\big); \qquad \alpha_i^+ = f_i'(x), \tag{2}$$

where γ is the step size and $\bar{\alpha} := {}^1\!/_n \sum_{i=1}^n \alpha_i$ can be updated efficiently in an online fashion. Crucially, $\mathbf{E}\alpha_i = \bar{\alpha}$ and thus the update direction is unbiased ($\mathbf{E}x^+ = x - \gamma f'(x)$). Furthermore, it can be proven (see [2]) that under a reasonable condition on γ, the update has vanishing variance, which enables the algorithm to converge linearly with a constant step size.

Sparse SAGA Algorithm. In its current form, every SAGA update is dense – even if the individual gradients are sparse – due to the historical gradient ($\bar{\alpha}$) term. Unfortunately, the usual tricks to resolve this issue are not portable to the parallel setting. Thus, we introduce Sparse SAGA, a novel variant which explicitly takes sparsity into account and is easily parallelizable. As in the Sparse SVRG algorithm proposed in [13], we obtain Sparse SAGA by a simple modification of the parameter update rule in (2) where $\bar{\alpha}$ is replaced by a sparse version equivalent in expectation:

$$x^+ = x - \gamma(f_i'(x) - \alpha_i + D_i\bar{\alpha}), \tag{3}$$

where D_i is a diagonal matrix that makes a weighted projection on the support of f_i'. More precisely, let S_i be the support of the gradient function f_i' (i.e., the set of coordinates where f_i' can be nonzero). Let D be a $d \times d$ diagonal reweighting matrix, with coefficients ${}^1\!/_{p_v}$ on the diagonal, where p_v is the probability that dimension v belongs to S_i when i is sampled uniformly at random in $\{1, ..., n\}$. We then define $D_i := P_{S_i} D$, where P_{S_i} is the projection onto S_i. The normalization from D ensures that $\mathbf{E}D_i\bar{\alpha} = \bar{\alpha}$, and thus that the update is still unbiased despite the projection.

Convergence Result. We model our convergence result after Hofmann et al. [4, Corollary 3], which provides the simplest form (note that the rate for Sparse SAGA is the same as SAGA).

Theorem 1. *Let $\gamma = \frac{a}{5L}$ for any $a \leq 1$. Then* Sparse SAGA *converges geometrically in expectation with a rate factor of at least $\rho(a) = \frac{1}{5}\min\left\{\frac{1}{n}, a\frac{1}{\kappa}\right\}$, i.e., for x_t obtained after t updates, we have $\mathbb{E}f(x_t) - f(x^*) \leq (1-\rho)^t C_0$, where $C_0 := \|x_0 - x^*\|^2 + \frac{n}{5L^2}\mathbf{E}\|\alpha_i^0 - f_i'(x^*)\|^2$.*

3 Asynchronous Parallel Sparse SAGA

We use a similar hardware model to Niu et al. [14], with multiple cores which read and update a shared central parameter vector in asynchronous and lock-free fashion. However we *do not* assume consistent vector reads: multiple cores can read and write different coordinates of the shared vector concurrently. Thus a full vector read by a core may not correspond to any consistent state in memory.

3.1 Perturbed Iterate Framework

In the sequential setting we can use a simple update rule to characterize SGD and its variants: $x_{t+1} = x_t - \gamma g(x_t, i_t)$, where i_t is a random variable independent from x_t and we have $\mathbf{E}g(x_t, i_t) = f'(x_t)$. But in the parallel setting we manipulate stale, inconsistent reads of shared parameters and thus do not have such a simple relationship. This was noted by Mania et al. [13] who proposed to separate $\hat{x}_t$ – the actual value read by a core during execution – with x_t, a "virtual iterate" that is *defined* by the update equation: $x_{t+1} := x_t - \gamma g(\hat{x}_t, i_t)$. We can thus interpret $\hat{x}_t$ as a noisy (perturbed) version of x_t due to the effect of asynchrony. In the specific case of Sparse SAGA, we get the following update:

$$x_{t+1} := x_t - \gamma g(\hat{x}_t, \hat{\alpha}^t, i_t); \qquad g(\hat{x}_t, \hat{\alpha}^t, i_t) := f_{i_t}'(\hat{x}_t) - \hat{\alpha}_{i_t}^t + D_{i_t}\left({}^1\!/_n \sum_{i=1}^n \hat{\alpha}_i^t\right). \tag{4}$$

We note that all the papers mentioned in the related work section (that analyzed asynchronous parallel randomized algorithms) assumed the following unbiasedness condition (and relied heavily on it):

$$[\text{unbiasedness condition}] \quad \mathbf{E}[g(\hat{x}_t, i_t)|\hat{x}_t] = f'(\hat{x}_t). \tag{5}$$

Mania et al. [13] correctly pointed out that most of the literature thus made the often implicit assumption that i_t is independent of $\hat{x}_t$. As we explain below, this assumption is incompatible with a non-uniform asynchronous model in the analysis approach used in most of the recent literature.

[1] For linear predictor models, the memory α_i^0 can be stored as a scalar.

3.2 On the Difficulty of Labeling the Iterates

Formalizing the meaning of x_t and $\hat{x}_t$ highlights a subtle but important difficulty arising when analyzing *randomized* parallel algorithms: what is the meaning of t? This is the problem of *labeling* the iterates for the analysis, and this labeling can have randomness itself that needs to be taken in consideration when interpreting an expression like $\mathbb{E}[x_t]$. In this section, we contrast three different approaches in a unified framework. We clarify the dependency issues mentioned in Mania et al. [13] and propose a new, simpler labeling which allows for much simpler proof techniques.

The "After Write" Approach. This is the standard labeling scheme used in Niu et al. [14] and most papers in the related work section ([13] and [3] excepted). t is a (virtual) global counter recording the number of *successful writes* to the shared memory x. (4) then means that $\hat{x}_t$ represents the (delayed) local copy value of the core that made the $(t+1)^{\text{th}}$ successful update; i_t is the associated factor sampled. Notice that if some values of i_t yield faster updates than others, it will influence the label assignment defining $\hat{x}_t$. We thus see that $\hat{x}_t$ and i_t share dependence through the t label assignment. In order to preserve the unbiasedness condition (5), we have to add the implicit assumption that the computation time for computing an update is independent of the sample i chosen. This assumption seems overly strong and is thus a fundamental flaw for analyzing the algorithms.

The "Before Read" Approach. Mania et al. [13] address this issue by proposing instead to increment the global t counter just *before* a new core starts to *read* the shared memory. $\hat{x}_t$ represents the read that was made by this core in this computational block, and i_t is the picked sample. The update rule (4) represents a *definition* of the meaning of x_t, which is now a "virtual iterate" (and is only ever used in the analysis) as it does not correspond to the content of the shared memory at any point.

A New Global Ordering: the "After Read" Approach. The "before read" approach gives rise to the following complication in the analysis: $\hat{x}_t$ can depend on i_r for $r > t$, since we have no guarantee on how long it takes a core to read. This means that we need to consider both the "future" and the "past" when analyzing x_t. To crucially simplify the analysis, we propose a third labeling: $\hat{x}_t$ represents the $(t+1)^{\text{th}}$ fully completed read. As the "before read" labeling, this approach ensures that there is no dependency between i_t and x_t injected through the labeling. But unlike in the "before read" approach, t does represent a global ordering on the $\hat{x}_t$ iterates – and thus we have that i_r is independent of $\hat{x}_t$ for $r > t$. Again using (4) as the definition of the virtual iterate x_t, we then have a very simple form for the value of $\hat{x}_t$ and x_t which we can use for the convergence analysis:

$$x_t = x_0 - \gamma \sum_{u=0}^{t-1} g(\hat{x}_u, \hat{\alpha}^u, i_u)\,; \qquad [\hat{x}_t]_v = [x_0]_v - \gamma \sum_{\substack{u=0 \\ \text{u s.t. coordinate } v \text{ was written} \\ \text{for } u \text{ before } t}}^{t-1} [g(\hat{x}_u, \hat{\alpha}^u, i_u)]_v\,. \tag{6}$$

3.3 Analysis setup

We describe ASAGA, an asynchronous parallel extension of Sparse SAGA, in Algorithm 1. Before stating its convergence, we highlight some properties of Algorithm 1 and make one central assumption. **First**, thanks to the "after read" global ordering, i_r is independent of $\hat{x}_t$ $\forall r \geq t$. We enforce the independence for $r = t$ by having the core read all the shared parameters before their iterations.

Second, the update, $g_t := g(\hat{x}_t, \hat{\alpha}^t, i_t)$, is an unbiased estimator of the true gradient at $\hat{x}_t$ (i.e. (4) yields (5) in conditional expectation). This property comes from the independence of i_t with $\hat{x}_t$. **Third**, the shared parameter coordinate update of $[x]_v$ on line 11 is atomic. As our updates are additions, this means there are no overwrites, even if several cores compete for the same resources. **Finally**, we assume that there exists a uniform bound, τ, on the maximum number of iterations that can overlap. This means that every coordinate update from iteration t is successfully written to memory before iteration $t + \tau + 1$ starts. τ is usually seen as a linear proxy for the number of cores, but it actually depends on several other factors and can be much bigger in real-life experiments. Our result will give us conditions on τ subject to which we have linear speedups.

Algorithm 1 ASAGA

1: Initialize shared variables x and $(\alpha_i)_{i=1}^n$
2: **keep doing in parallel**
3: $\quad\hat{x} =$ inconsistent read of x
4: $\quad\forall j, \hat{\alpha}_j =$ inconsistent read of α_j
5: $\quad$Sample i uniformly at random in $\{1, ..., n\}$
6: $\quad$Let S_i be f_i's support
7: $\quad[\bar{\alpha}]_{S_i} := {}^1\!/_n \sum_{k=1}^n [\hat{\alpha}_k]_{S_i}$
8: $\quad[\delta x]_{S_i} := -\gamma(f_i'(\hat{x}) - \hat{\alpha}_i + D_i[\bar{\alpha}]_{S_i})$
9:
10: $\quad$**for** v **in** S_i **do**
11: $\quad\quad[x]_v \leftarrow [x]_v + [\delta x]_v$ // atomic
12: $\quad\quad[\alpha_i]_v \leftarrow [f_i'(\hat{x})]_v$
13:
14: $\quad$**end for**
15: **end parallel loop**

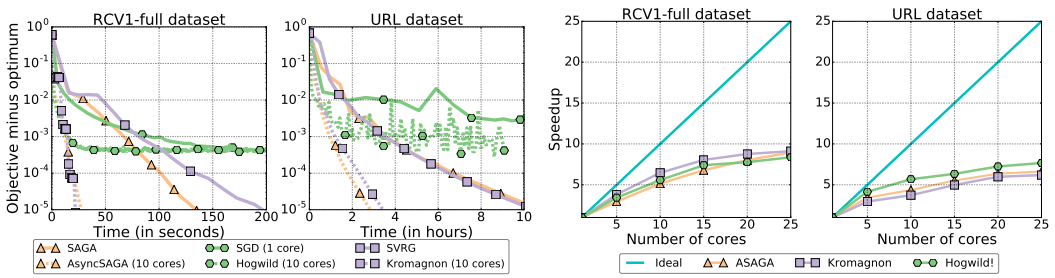

Figure 1: **Convergence and speedup for asynchronous stochastic gradient descent methods**.

Explicit effect of asynchrony. By using the overlap assumption in the expression (6) for the iterates, we obtain the following explicit effect of asynchrony that is crucially used in our proof:

$$\hat{x}_t - x_t = \gamma \sum_{u=(t-\tau)_+}^{t-1} S_u^t g(\hat{x}_u, \hat{\alpha}^u, i_u), \tag{7}$$

where S_u^t are $d \times d$ diagonal matrices with terms in $\{+1, 0\}$. Though every update in $\hat{x}_t$ is already in x_t – this is the 0 case – some updates might be late – this is the $+1$ case. Crucially, while $\hat{x}_t$ may be lacking some "past" updates, given our global ordering definition, it cannot contain "future" updates.

3.4 Convergence and speedup results

Definition 1 (Sparsity)**.** *Following Niu et al. [14], we introduce* $\Delta_r := \max_{v=1..d} |\{(i : v \in S_i\}|$. Δ_r *is the maximum number of data points with a specific feature. For succinctness, we also define* $\Delta := \Delta_r / n$. *We have* $1 \leq \Delta_r \leq n$, *and hence* $1/n \leq \Delta \leq 1$.

Theorem 2 (Convergence guarantee and rate of ASAGA)**.** *Suppose* $\tau < n/10$.[2] *Let*

$$a^*(\tau) := \frac{1}{32\left(1 + \tau\sqrt{\Delta}\right)\xi(\kappa, \Delta, \tau)} \qquad \text{where } \xi(\kappa, \Delta, \tau) := \sqrt{1 + \frac{1}{8\kappa}\min\{\frac{1}{\sqrt{\Delta}}, \tau\}} \tag{8}$$

(note that $\xi(\kappa, \Delta, \tau) \approx 1$ *unless* $\kappa < 1/\sqrt{\Delta}$ $(\leq \sqrt{n})$*).*

For any step size $\gamma = \frac{a}{L}$ *with* $a \leq a^*(\tau)$, *the inconsistent read iterates of Algorithm 1 converge in expectation at a geometric rate of at least:* $\rho(a) = \frac{1}{5}\min\left\{\frac{1}{n}, a\frac{1}{\kappa}\right\}$, *i.e.,* $\mathbb{E}f(\hat{x}_t) - f(x^*) \leq (1-\rho)^t \tilde{C}_0$, *where* $\tilde{C}_0$ *is a constant independent of* t $(\approx \frac{n}{\gamma} C_0$ *with* C_0 *as defined in Theorem 1).*

Within constants, this result is very close to SAGA's original convergence theorem, but with the maximum step size divided by an extra $1 + \tau\sqrt{\Delta}$ factor. Referring to Hofmann et al. [4] and our own Theorem 1, the rate factor for SAGA is $\min\{1/n, 1/\kappa\}$ up to a constant factor. Comparing this rate with Theorem 2 and inferring the conditions on the maximum step size $a^*(\tau)$, we get the following conditions on the overlap τ for ASAGA to have the same rate as SAGA (comparing upper bounds).

Corollary 3 (Speedup condition)**.** *Suppose* $\tau \leq \mathcal{O}(n)$ *and* $\tau \leq \mathcal{O}(\frac{1}{\sqrt{\Delta}}\max\{1, \frac{n}{\kappa}\})$. *Then using the step size* $\gamma = a^*(\tau)/L$ *from (8),* ASAGA *converges geometrically with the rate factor* $\Omega(\min\{\frac{1}{n}, \frac{1}{\kappa}\})$ *(similar to* SAGA*), and is thus linearly faster than its sequential counterpart up to a constant factor. Moreover, if* $\tau \leq \mathcal{O}(\frac{1}{\sqrt{\Delta}})$, *then a universal step size of* $\Theta(\frac{1}{L})$ *can be used for* ASAGA *to be adaptive to local strong convexity with a similar rate to* SAGA *(i.e., knowledge of* κ *is not required).*

Interestingly, in the well-conditioned regime ($n > \kappa$), ASAGA can get the same rate as SAGA even without sparsity ($\Delta = 1$) for $\tau < \mathcal{O}(n/\kappa)$ – in contrast to previous work where asynchronous methods required some kind of sparsity to get a theoretical linear speedup [14, 13].

4 Empirical results

We run logistic regression on RCV1 [9] and URL [12]. We compare three different algorithms: ASAGA, KROMAGNON (the asynchronous sparse SVRG method described in [13]) and HOGWILD [14]. For each method we consider its asynchronous version with both one (hence sequential) and ten processors (Figure 1, left) and we examine the speedup relative to the increase in the number of cores (right). We observe that although the asynchronous version offers a significant runtime speedup, it is not linear as predicted by our theory (and confirmed by our iterations speedup). This phenomenon can be explained by the fact that there is no such thing as *shared memory*. In reality, as we add more cores, we start using slower types of memory (RAM vs cache) for information passing.

[2] ASAGA can actually converge for any τ, but the bound on the maximum step size gets much worse.

References

[1] C. De Sa, C. Zhang, K. Olukotun, and C. Ré. Taming the wild: a unified analysis of Hogwild!-style algorithms. In *NIPS*, 2015.

[2] A. Defazio, F. Bach, and S. Lacoste-Julien. SAGA: A fast incremental gradient method with support for non-strongly convex composite objectives. In *NIPS*, 2014.

[3] J. C. Duchi, S. Chaturapruek, and C. Ré. Asynchronous stochastic convex optimization. In *NIPS*, 2015.

[4] T. Hofmann, A. Lucchi, S. Lacoste-Julien, and B. McWilliams. Variance reduced stochastic gradient descent with neighbors. In *NIPS*, 2015.

[5] C.-J. Hsieh, H.-F. Yu, and I. Dhillon. PASSCoDe: Parallel ASynchronous Stochastic dual Co-ordinate Descent. In *ICML*, 2015.

[6] R. Johnson and T. Zhang. Accelerating stochastic gradient descent using predictive variance reduction. In *NIPS*, 2013.

[7] N. Le Roux, M. Schmidt, and F. Bach. A stochastic gradient method with an exponential convergence rate for finite training sets. In *NIPS*, 2012.

[8] R. Leblond, F. Pedregosa, and S. Lacoste-Julien. Asaga: Asynchronous parallel Saga. *arXiv:1606.04809*, 2016.

[9] D. D. Lewis, Y. Yang, T. G. Rose, and F. Li. RCV1: A new benchmark collection for text categorization research. *JMLR*, 5:361–397, 2004.

[10] X. Lian, Y. Huang, Y. Li, and J. Liu. Asynchronous parallel stochastic gradient for nonconvex optimization. In *NIPS*, 2015.

[11] J. Liu, S. J. Wright, C. Ré, V. Bittorf, and S. Sridhar. An asynchronous parallel stochastic coordinate descent algorithm. *JMLR*, 16:285–322, 2015.

[12] J. Ma, L. K. Saul, S. Savage, and G. M. Voelker. Identifying suspicious URLs: an application of large-scale online learning. In *ICML*, 2009.

[13] H. Mania, X. Pan, D. Papailiopoulos, B. Recht, K. Ramchandran, and M. I. Jordan. Perturbed iterate analysis for asynchronous stochastic optimization. *arXiv:1507.06970*, 2015.

[14] F. Niu, B. Recht, C. Re, and S. Wright. Hogwild: a lock-free approach to parallelizing stochastic gradient descent. In *NIPS*, 2011.

[15] S. J. Reddi, A. Hefny, S. Sra, B. Póczos, and A. Smola. On variance reduction in stochastic gradient descent and its asynchronous variants. In *NIPS*, 2015.

[16] S. Shalev-Shwartz and T. Zhang. Stochastic dual coordinate ascent methods for regularized loss. *JMLR*, 14:567–599, 2013.

[17] S.-Y. Zhao and W.-J. Li. Fast asynchronous parallel stochastic gradient descent. In *AAAI*, 2016.