Perturbed Iterate Analysis for Asynchronous Stochastic Optimization

Horia Mania[α,ϵ], **Xinghao Pan**[α,ϵ], **Dimitris Papailiopoulos**[α,ϵ]
Benjamin Recht[α,ϵ,σ], **Kannan Ramchandran**[ϵ], **and Michael I. Jordan**[α,ϵ,σ]
[α]AMPLab, [ϵ]EECS at UC Berkeley, [σ]Statistics at UC Berkeley

Abstract

We introduce and analyze stochastic optimization methods where the input to each gradient update is perturbed by noise. We show that this framework forms the basis of a unified approach to analyze asynchronous stochastic optimization algorithms. The main intuition is that asynchronous algorithms can be thought of as serial methods operating on noisy inputs. Using our framework, we provide a new analysis for HOGWILD! that is simpler than earlier analyses, removes many assumptions of previous models, and can yield improved performance bounds. We further apply our framework to develop and analyze KROMAGNON: a novel, parallel, sparse stochastic variance-reduced gradient (SVRG) algorithm. We demonstrate on a 16-core experiments that KROMAGNON can be up to four orders of magnitude faster than the standard SVRG algorithm.

1 Introduction

Asynchronous parallel stochastic optimization algorithms have recently gained significant traction in algorithmic machine learning. A large body of recent work has demonstrated that near-linear speedups are achievable, in theory and practice, on many common machine learning tasks [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 14, 15, 13, 16].

Although asynchronous stochastic algorithms are simple to implement and enjoy excellent performance in practice, they are challenging to analyze theoretically. The current analyses require lengthy derivations and several assumptions that sometimes may not reflect realistic system behaviors. Moreover, due to the difficult proof machinery, the algorithms analyzed are often simplified versions of the ones actually run in practice. To overcome these difficulties, we propose a general framework for obtaining convergence rates for parallel, lock-free, asynchronous first-order stochastic algorithms. We interpret the algorithmic effects of asynchrony as perturbing the stochastic iterates with bounded noise. This intuition allows us to show how a variety of asynchronous first-order algorithms can be analyzed as their serial counterparts operating on noisy inputs. The advantage of our framework is that it is compact and elementary, can remove or relax simplifying assumptions adopted in prior art, and can yield improved bounds than earlier work.

We demonstrate the general applicability of our framework by providing new convergence analyses for HOGWILD!, and introduce and analyze KROMAGNON: a new asynchronous sparse version of the stochastic variance-reduced gradient (SVRG) method [17]. KROMAGNON is a modified version of SVRG that allows for sparse updates, we show that this method can be parallelized in the asynchronous model. Experimentally, KROMAGNON achieves nearly-linear speedups on a machine with 16 cores and can be up to four orders of magnitude faster than the standard (dense) SVRG.

2 Asynchronous Optimization through a Perturbed Iterate Lens

We study parallel asynchronous iterative algorithms that minimize convex functions $f(\mathbf{x})$ with $\mathbf{x} \in \mathbb{R}^d$. The computational model is the same as that of Niu et al.[1]: a number of cores have access to the same shared memory, and each of them can read and update components of $\mathbf{x}$ in the shared memory. The algorithms that we consider are asynchronous and lock-free: cores do not coordinate

their reads or writes, and while a core is reading/writing other cores can update the shared variables in $\mathbf{x}$. We focus our analysis on functions f that are L-smooth and m-strongly convex.

A popular way to minimize convex functions is via *first-order stochastic* algorithms. These algorithms can be described using the following iteration

$$\mathbf{x}_{j+1} = \mathbf{x}_j - \gamma \mathbf{g}(\mathbf{x}_j, \xi_j) \tag{1}$$

where ξ_j is a random variable independent of $\mathbf{x}_j$ and $\mathbb{E}_{\xi_j} \mathbf{g}(\mathbf{x}_j, \xi_j) = \nabla f(\mathbf{x}_j)$. A major advantage of the above iterative formula is that it can be used to track the algorithmic progress and establish convergence rates to the optimal solution. Unfortunately, the progress of asynchronous parallel algorithms cannot be precisely analyzed using the above iterative framework. Processors do not read from memory actual iterates $\mathbf{x}_j$, as there is no global clock that synchronizes reads or writes among cores, resulting in *inconsistent reads* of the shared variable.

In the subsequent sections, we show that the following simple perturbed variant of (1) can capture the algorithmic progress of asynchronous stochastic algorithms:

$$\mathbf{x}_{j+1} = \mathbf{x}_j - \gamma \mathbf{g}(\mathbf{x}_j + \mathbf{n}_j, \xi_j) \tag{2}$$

where $\mathbf{n}_j$ is a stochastic error term. For simplicity let $\hat{\mathbf{x}}_j = \mathbf{x}_j + \mathbf{n}_j$. If we define $a_j = \mathbb{E}\|\mathbf{x}_j - \mathbf{x}^*\|^2$, we obtain the following recursion after elementary manipulations

$$a_{j+1} \leq (1-\gamma m) a_j + \gamma^2 \underbrace{\mathbb{E}\|g(\hat{\mathbf{x}}_j, \xi_j)\|^2}_{R_0^j} + 2\gamma m \underbrace{\mathbb{E}\|\hat{\mathbf{x}}_j - \mathbf{x}_j\|^2}_{R_1^j} + 2\gamma \underbrace{\mathbb{E}\langle \hat{\mathbf{x}}_j - \mathbf{x}_j, \mathbf{g}(\hat{\mathbf{x}}_j, \xi_j)\rangle}_{R_2^j}. \tag{3}$$

The above recursion is key to our analysis. We show that for given R_0^j, R_1^j, and R_2^j, we can establish convergence rates through elementary algebraic manipulations. The key contribution of our work is to show that 1) this iteration can capture the algorithmic progress of asynchronous algorithms, and 2) the error terms can be bounded to obtain the "right" convergence rates for HOGWILD!, and KROMAGNON our novel asynchronous sparse SVRG.

Analyzing HOGWILD! We provide a simple analysis of HOGWILD!, a popular asynchronous implementation of SGM [1]. We assume that f is decomposable in n terms $f(\mathbf{x}) = \frac{1}{n}\sum_{i=1}^n f_{e_i}(\mathbf{x})$ where $\mathbf{x} \in \mathbb{R}^d$, and each term $f_{e_i}(\mathbf{x})$ depends only on the coordinates indexed by the subset e_i of $\{1, 2, \ldots, d\}$. We refer to the sets e_i as *hyperedges* and denote the set of hyperedges by $\mathcal{E}$. The hyperedges imply a conflict graph between the n function terms, where two terms f_{e_i} and f_{e_j} are connected if $e_i \cap e_j \neq \emptyset$. Let us define by Δ_c, the average degree of this conflict graph.

HOGWILD! (Alg. 1) is deployed on multiple cores that have access to shared memory, where the optimization variable $\mathbf{x}$ and the data points that define the f terms are stored. During its execution each core samples uniformly at random a hyperedge s from $\mathcal{E}$. It reads the coordinates $v \in s$ of the shared vector $\mathbf{x}$, evaluates ∇f_s at the point read, and adds $-\gamma \nabla f_s$ to the shared variable. In HOGWILD! cores do not synchronize or follow an order between reads or writes. Moreover, they access/update a set of coordinates in $\mathbf{x}$ without the use of any locking mechanisms that would ensure a conflict-free execution. The reads/writes of distinct cores can intertwine in arbitrary ways, *e.g.*, while a core updates a subset of variables, other cores can access/update the same variables.

Algorithm 1 HOGWILD!

1: **while** iterations $\leq T$ **do in parallel**
2: $\quad \hat{\mathbf{x}}$ = inconsistent read of the shared variable
3: $\quad s$ = random hyperedge
4: $\quad \mathbf{u} = -\gamma \cdot \mathbf{g}(\hat{\mathbf{x}}, s)$
5: $\quad$ **for all** $v \in s$ **do**
6: $\quad\quad [\mathbf{x}]_v = [\mathbf{x}]_v + [\mathbf{u}]_v$ // atomic write

In [1], the authors analyzed a simplified variant of HOGWILD!, and established convergence rates under several assumptions. In their analysis only a single coordinate per sampled hyperedge is updated (*i.e.*, the for loop in HOGWILD! is replaced with a single coordinate update). The authors moreover assume consistent reads, i.e., while a processor is reading, no writes to the shared memory occur. We show how our perturbed gradient framework can be used in an elementary way to analyze the "full updates" version of HOGWILD!, while obtaining improved bounds compared to [1] and removing several assumptions.

A subtle, but important point is that we order the samples based on the order in which they were sampled, *not* the order in which cores complete processing them. In our analysis, s_i denotes the

sample obtained when line 2 in Alg. 1 is executed for the i-th time. This ordering is distinct from the one used in the original work of [1]. In that setting, the samples are ordered according to the completion time of each thread. The problem with the latter ordering is that the distribution of the samples is not always uniform (a theoretical requirement for SGM), something that is disregarded in [1]. Our ordering resolves this issue by guaranteeing uniformity among samples.

Assuming atomic writes (i.e., writes complete successfully at *some* point in time), then they will all appear in the shared memory by the end of the execution, in the form of additive coordinate-wise updates. Due to the commutativity of addition, the following vector is contained in the shared memory after processing a total of T hyperedges:[1]

$$\underbrace{\overbrace{\mathbf{x}_0 - \gamma \mathbf{g}(\hat{\mathbf{x}}_0, s_0)}^{\mathbf{x}_1} - \ldots - \gamma \mathbf{g}(\hat{\mathbf{x}}_{T-1}, s_{T-1})}_{\mathbf{x}_T},$$

where $\mathbf{x}_0$ is the initial guess and $\hat{\mathbf{x}}_i$ is what the processor working on s_i read from the shared memory. We now define the perturbed iterates as $\mathbf{x}_{i+1} = \mathbf{x}_i - \gamma \mathbf{g}(\hat{\mathbf{x}}_i, s_i)$ where s_i is the i-th uniformly sampled hyperedge. Observe that all but the last of these iterates are "fake": there might not be an actual time when they exist in the shared memory during the execution. However, $\mathbf{x}_0$ is what is stored in memory before the execution starts, and $\mathbf{x}_T$ is exactly what is stored in shared memory at the end of the execution. We observe that the above iterates place HOGWILD! in our perturbed gradient framework. We are only left to bound the three error terms R_0^j, R_1^j, R_2^j.

To measure the distance between $\hat{\mathbf{x}}_j$ and $\mathbf{x}_j$, observe that any difference between them is solely caused by hyperedges that overlap in time with s_j (i.e. samples s_i that are processed at the same time with s_j). To see this, let s_i be an "earlier" sample, *i.e.* $i < j$, that does not overlap with s_j in time. This implies that the processing of s_i finishes before s_j starts being processed. Hence, the full contribution of $\gamma \mathbf{g}(\hat{\mathbf{x}}_i, s_i)$ will be recorded in both $\hat{\mathbf{x}}_j$ and $\mathbf{x}_j$ (for the latter this holds by definition). Similarly, if $i > j$ and s_i does not overlap with s_j in time, then neither $\hat{\mathbf{x}}_j$ nor $\mathbf{x}_j$ (for the latter, again by definition) contain *any* of the coordinate updates involved in the gradient update $\gamma \mathbf{g}(\hat{\mathbf{x}}_i, s_i)$. To proceed, we adopt a simple assumption held in prior art.

Assumption 1. *No more than τ hyperedges can overlap in time with a single sampled hyperedge.*

Assumption 1 guarantees that if $i < j - \tau$ or $i > j + \tau$, then the sample s_i *does not* overlap in time with s_j. Hence, there exist diagonal matrices $\mathbf{S}_i^j$ with entries in $\{-1, 0, 1\}$ such that $\hat{\mathbf{x}}_j - \mathbf{x}_j = \sum_{i=j-\tau,\, i\neq j}^{j+\tau} \gamma \mathbf{S}_i^j \mathbf{g}(\hat{\mathbf{x}}_i, s_i)$. These diagonal matrices account for all possible updates occurring while sample s_j is being processed. Using some elementary graph theoretic and probability arguments, we obtain the following error bounds.

Lemma 1. HOGWILD! *satisfies recursion* (3) *with* $R_1^j = \mathbb{E}\|\hat{\mathbf{x}}_j - \mathbf{x}_j\|^2 \leq \mathcal{O}(1) \cdot \gamma^2 M^2 \left(\tau + \tau^2 \frac{\Delta_c}{n}\right)$ *and* $R_2^j = \mathbb{E}\langle \hat{\mathbf{x}}_j - \mathbf{x}_j, \mathbf{g}(\hat{\mathbf{x}}_j, s_j)\rangle \leq \mathcal{O}(1) \cdot \gamma M^2 \cdot \tau \frac{\Delta_c}{n}$. *where* $M \geq \|\mathbf{g}\|$.

Plugging the bounds of Lemma 1 in our recursive formula (3), asserts that HOGWILD! satisfies the recursion $a_{j+1} \leq (1 - \gamma m)\, a_j + \mathcal{O}(1)\gamma^2 M^2 \left(1 + \tau \frac{\Delta_c}{n} + \gamma m \tau + \gamma m \tau^2 \frac{\Delta_c}{n}\right)$. Observe that if τ—a proxy for the number of cores—is $\mathcal{O}(\min\{n/\Delta_c, M^2/(\epsilon m^2)\})$, then—up to constant factors—HOGWILD! satisfies the same recursion as SGM. Our main result follows.

Theorem 2. *If* $\tau = \mathcal{O}(\min\{n/\Delta_c, M^2/\epsilon m^2\})$, *then* HOGWILD! *with step-size* $\gamma = \mathcal{O}(1)\epsilon m/M^2$, *reaches an accuracy of* $\mathbb{E}\|\mathbf{x}_T - \mathbf{x}^*\|^2 \leq \epsilon$ *after* $T \geq \mathcal{O}(1)\frac{M^2 \log(a_0/\epsilon)}{\epsilon m^2}$ *iterations.*

Comparison to the original HOGWILD! analysis of [1] We would like to summarize the key points of improvement compared to the original HOGWILD! analysis: *1)* Our perturbed iterate analysis is elementary and compact, and follows simply by bounding the the asynchrony error terms R_0^j, R_1^j, R_2^j. *2)* We do not assume consistent reads: while a core is reading the shared variable other cores are allowed to read, or write. *3)* We analyze the "full-update" version of HOGWILD!; in the original HOGWILD! analysis only a single random coordinate of a sampled hyperedge is updated. *4)* We use an ordering of the samples that guarantees they have a uniform distribution, a key property for the convergence analysis. *5)* [1] establishes a nearly-linear speedup for HOGWILD! if τ, the proxy for the number of cores, is bounded as $\tau = \mathcal{O}(\sqrt[4]{n/\Delta_c})$. Here, we obtain a linear speedup for up to $\tau = \mathcal{O}(\min\{n/\Delta_c, M^2/\epsilon m^2\})$, which can be orders of magnitude larger.

[1] throughout this section we denote $\mathbf{g}(\mathbf{x}, s_j) = \nabla f_{s_j}(\mathbf{x})$, which we assume to be bounded $\|\mathbf{g}(\mathbf{x}, s)\| \leq M$

3 KROMAGNON: Sparse and Asynchronous SVRG

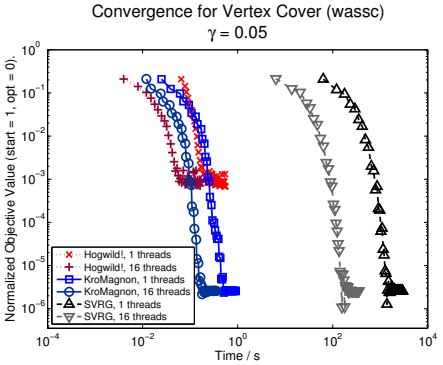
The stochastic variance reduced gradient (SVRG) algorithm is a variant of SGM that uses intermediate full gradient computations to achieve linear rates [17]. Can SVRG be parallelized in the asynchronous setting? In practice one of the bottlenecks on sparse problems is that the SVRG iterates are dense, leading to severe memory conflicts, and redundant access overheads. We overcome that by appropriately projecting each stochastic gradient step on the support of the sampled hyperedge. This sparse-updates SVRG allows us to analyze it under the asynchronous model using our perturbed iterates framework. In practice, the sparsification of the updates leads to a significantly faster algorithm compared to dense SVRG. KROMAGNON is our asynchronous implementation of sparse SVRG, and is given in Algorithm 2. The diagonal matrix $\mathbf{D}_s$ has non-zero entries proportional to the inverse degree of each coordinate of $\mathbf{x}$ on the bipartite graph between function terms and variables in $\mathbf{x}$ that is defined by the hyperedges. Our main result of this section is given below.

Algorithm 2 KROMAGNON

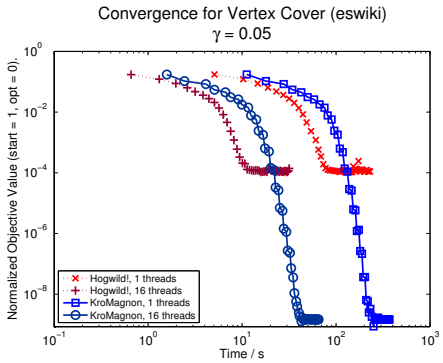
$\mathbf{x} = \mathbf{y} = \mathbf{x}_0$
for all Epoch = $1 : E$ **do**
  $\mathbf{y} = \mathbf{x}$; Compute in parallel $\mathbf{z} = \nabla f(\mathbf{y})$
  while iterations $\leq S$ **do in parallel**
    $\hat{\mathbf{x}}$ = an inconsistent RAM read
    sample a random hyperedge s
    $[\mathbf{u}]_s = -\gamma \cdot (\nabla f_s([\mathbf{x}]_s) - \nabla f_s([\mathbf{y}]_s) - \mathbf{D}_s\mathbf{z})$
    for all $v \in s$ **do**
      $[\mathbf{x}]_v = [\mathbf{x}]_v + [\mathbf{u}]_v$ // atomic write

Theorem 3. *Let* $\tau = \mathcal{O}(\min\{\kappa/\log(M^2/L^2\epsilon), \sqrt[6]{n/\Delta_\epsilon}\})$ *where* $\kappa = L/m$. *Then,* KROMAGNON, *with step-size* $\gamma = \mathcal{O}(1)\frac{1}{L\kappa}$ *and epoch size* $S = \mathcal{O}(1)\kappa^2$, *attains* $\mathbb{E}\|\mathbf{y}_E - \mathbf{x}^*\|^2 \leq \epsilon$ *after* $E = \mathcal{O}(1)\log\left(\frac{a_0}{\epsilon}\right)$ *epochs, where* $\mathbf{y}_E$ *is the final iterate and* $a_0 = \|\mathbf{x}_0 - \mathbf{x}^*\|^2$.

Empirical Evaluation We implemented HOGWILD!, dense SVRG, and KROMAGNON in Scala. Following [6], we optimize a quadratic penalty relaxation for vertex cover eswiki-2013 where $n \approx 970K$, and $d \approx 23M$ and wordassociation-2011 with $n \approx 10K$ and $d \approx 72K$ [18, 19, 20]. Each algorithm was run for 50 epochs and up to 16 threads. For the SVRG algorithms, we recompute $\mathbf{y}$ and the full gradient $\nabla f(\mathbf{y})$ every 2 epochs. We normalize the objective values such that the objective at the initial starting point has a value of 1, and the minimum attained across all algorithms and epochs has a value of 0. Experiments were conducted on a Linux machine with 2 Intel Xeon Processor E5-2670 (2.60GHz, 8 cores each) with 250Gb memory. We were unable to run dense SVRG on the eswiki-2013 dataset due to the large number of features. We observe that KROMAGNON is at least one and up to four orders of magnitude faster than dense SVRG. Observe that both dense SVRG and KROMAGNON attain similar optima.



(a) Vertex cover, wordassociation



(b) Vertex cover, eswiki-2013

References

[1] Feng Niu, Benjamin Recht, Christopher Re, and Stephen Wright. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in Neural Information Processing Systems*, pages 693–701, 2011.

[2] Ben Recht, Christopher Re, Joel Tropp, and Victor Bittorf. Factoring nonnegative matrices with linear programs. In *Advances in Neural Information Processing Systems*, pages 1214–1222, 2012.

[3] Yong Zhuang, Wei-Sheng Chin, Yu-Chin Juan, and Chih-Jen Lin. A fast parallel sgd for matrix factorization in shared memory systems. In *Proceedings of the 7th ACM conference on Recommender systems*, pages 249–256. ACM, 2013.

[4] Hyokun Yun, Hsiang-Fu Yu, Cho-Jui Hsieh, SVN Vishwanathan, and Inderjit Dhillon. Nomad: Non-locking, stochastic multi-machine algorithm for asynchronous and decentralized matrix completion. *arXiv preprint arXiv:1312.0193*, 2013.

[5] John Duchi, Michael I Jordan, and Brendan McMahan. Estimation, optimization, and parallelism when data is sparse. In *Advances in Neural Information Processing Systems*, pages 2832–2840, 2013.

[6] Ji Liu, Steve Wright, Christopher Re, Victor Bittorf, and Srikrishna Sridhar. An asynchronous parallel stochastic coordinate descent algorithm. In *Proceedings of the 31st International Conference on Machine Learning (ICML-14)*, pages 469–477, 2014.

[7] Yu-Xiang Wang, Veeranjaneyulu Sadhanala, Wei Dai, Willie Neiswanger, Suvrit Sra, and Eric P Xing. Asynchronous parallel block-coordinate frank-wolfe. *arXiv preprint arXiv:1409.6086*, 2014.

[8] Mingyi Hong. A distributed, asynchronous and incremental algorithm for nonconvex optimization: An admm based approach. *arXiv preprint arXiv:1412.6058*, 2014.

[9] Cho-Jui Hsieh, Hsiang-Fu Yu, and Inderjit S Dhillon. Passcode: Parallel asynchronous stochastic dual co-ordinate descent. *arXiv preprint arXiv:1504.01365*, 2015.

[10] Hamid Reza Feyzmahdavian, Arda Aytekin, and Mikael Johansson. An asynchronous mini-batch algorithm for regularized stochastic optimization. *arXiv preprint arXiv:1505.04824*, 2015.

[11] Ji Liu, Stephen J Wright, and Srikrishna Sridhar. An asynchronous parallel randomized kaczmarz algorithm. *arXiv preprint arXiv:1401.4780*, 2014.

[12] Haim Avron, Alex Druinsky, and Anshul Gupta. Revisiting asynchronous linear solvers: Provable convergence rate through randomization. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 198–207. IEEE, 2014.

[13] Sashank J Reddi, Ahmed Hefny, Suvrit Sra, Barnabás Póczos, and Alex Smola. On variance reduction in stochastic gradient descent and its asynchronous variants. *arXiv preprint arXiv:1506.06840*, 2015.

[14] Christopher De Sa, Ce Zhang, Kunle Olukotun, and Christopher Ré. Taming the wild: A unified analysis of hogwild!-style algorithms. *arXiv preprint arXiv:1506.06438*, 2015.

[15] Xiangru Lian, Yijun Huang, Yuncheng Li, and Ji Liu. Asynchronous parallel stochastic gradient for nonconvex optimization. *arXiv preprint arXiv:1506.08272*, 2015.

[16] Zhimin Peng, Yangyang Xu, Ming Yan, and Wotao Yin. ARock: an Algorithmic Framework for Asynchronous Parallel Coordinate Updates. *arXiv preprint arXiv:1506.02396*, 2015.

[17] Rie Johnson and Tong Zhang. Accelerating stochastic gradient descent using predictive variance reduction. In *Advances in Neural Information Processing Systems*, pages 315–323, 2013.

[18] Paolo Boldi and Sebastiano Vigna. The WebGraph framework I: Compression techniques. In *"WWW"*, 2004.

[19] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In *WWW*. "ACM Press", 2011.

[20] Paolo Boldi, Bruno Codenotti, Massimo Santini, and Sebastiano Vigna. Ubicrawler: A scalable fully distributed web crawler. *Software: Practice & Experience*, 34(8):"711–726", 2004.