Splash: User-friendly Programming Interface for Parallelizing Stochastic Algorithms

Yuchen Zhang **Michael I. Jordan**
Department of Electrical Engineering and Computer Science
University of California, Berkeley CA 94720
{yuczhang,jordan}@eecs.berkeley.edu

Abstract

Stochastic algorithms are efficient approaches to solving machine learning and optimization problems. In this extended abstract, we propose a general framework called *Splash* for parallelizing stochastic algorithms on multi-node distributed systems. Splash consists of a programming interface and an execution engine. Using the programming interface, the user develops sequential stochastic algorithms without concerning any detail about distributed computing. The algorithm is then automatically parallelized by a communication-efficient execution engine. Splash is built on top of Apache Spark. The real-data experiments on logistic regression, collaborative filtering and topic modeling verify that Splash yields order-of-magnitude speedup over single-thread stochastic algorithms and over state-of-the-art implementations on Spark.

1 Introduction

Stochastic optimization algorithms process a large-scale dataset by sequentially processing random subsamples. This processing scheme makes the per-iteration cost of the algorithm much cheaper than that of batch processing algorithms while still yielding effective descent. Indeed, for convex optimization, the efficiency of stochastic gradient descent (SGD) and its variants has been established both in theory and in practice [30, 4, 27, 6, 25, 11]. For non-convex optimization, stochastic methods achieve state-of-the-art performance on a broad class of problems, including matrix factorization [12], neural networks [13] and representation learning [26]. Stochastic algorithms are also widely used in the Bayesian setting for finding approximations to posterior distributions; examples include Markov chain Monte Carlo, expectation propagation [20] and stochastic variational inference [10].

Although classical stochastic approximation procedures are sequential, it is clear that they also present opportunities for parallel and distributed implementations that may yield significant additional speedups. One active line of research studies asynchronous parallel updating schemes in the setting of a lock-free shared memory [24, 7, 17, 33, 9]. When the time delay of concurrent updates are bounded, it is known that such updates preserve statistical correctness [1, 17]. Such asynchronous algorithms yield significant speedups on multi-core machines. On distributed systems connected by commodity networks, however, the communication requirements of such algorithms can be overly expensive. If messages are frequently exchanged across the network, the communication cost will easily dominate the computation cost.

There has also been a flurry of research studying the implementation of stochastic algorithms in the fully distributed setting [34, 32, 22, 8, 16]. Although promising results have been reported, the implementations proposed to date have their limitations—they have been designed for specific algorithms, or they require careful partitioning of the data to avoid inconsistency.

In this extended abstract, we propose a general framework for parallelizing stochastic algorithms on multi-node distributed systems. Our framework is called Splash (**S**ystem for **P**arallelizing **L**earning

Algorithms with **S**tochastic Met**h**ods). Splash consists of a programming interface and an execution engine. Using the programming interface, the user develops sequential stochastic algorithms without thinking about issues of distributed computing. The algorithm is then automatically parallelized by the execution engine. The parallelization is communication efficient, meaning that its separate threads don't communicate with each other until all of them have processed a large bulk of data. Thus, the inter-node communication need not be a performance bottleneck.

The programming interface is designed around a key paradigm: implementing incremental updates that processes weighted data. Unlike existing distributed machine learning systems [5, 28, 14, 21] which requires the user to explicitly specify a distributed algorithm, Splash asks the user to implement a processing function that takes an individual data element as input to incrementally update the corresponding variables. When this function is iteratively called on a sequence of samples, it defines a sequential stochastic algorithm. It can also be called in a distributed manner for constructing parallel algorithms, which is the job of the execution engine. This programming paradigm allows one algorithmic module working on different computing environments, no matter if it is a single-core processor or a large-scale cluster. As a consequence, the challenge of parallelizing these algorithms has been transferred from the developer side to the system side.

To ensure parallelizability, the user is asked to implement a slightly stronger version of the base sequential algorithm: it needs to be capable of processing *weighted samples*. An m-weighted sample tells the processing function that the sample appears m times consecutively in the sequence. Many stochastic algorithms can be generalized to processing weighted samples without sacrificing computational efficiency. We will demonstrate SGD and collapsed Gibbs sampling as two concrete examples. Since the processing of weighted samples can be carried out within a sequential paradigm, this requirement does not force the user to think about a distributed implementation.

In order to parallelize the algorithm, Splash converts a distributed processing task into a sequential processing task using distributed versions of *averaging* and *reweighting*. During the execution of the algorithm, we let every thread sequentially process its local data. The local updates are iteratively averaged to construct the global update. Critically, however, although averaging reduces the variance of the local updates, it doesn't reduce their bias. In contrast to the sequential case in which a thread processes a full sequence of random samples, in the distributed setting every individual thread touches only a small subset of samples, resulting in a significant bias relative to the full update. Our reweighting scheme addresses this problem by feeding the algorithm with weighted samples, ensuring that the total weight processed by each thread is equal to the number of samples in the full sequence. This helps individual threads to generate nearly-unbiased estimates of the full update. Using this approach, Splash automatically detects the best degree of parallelism for the algorithm.

We conduct extensive experiments on a variety of stochastic algorithms, including algorithms for logistic regression, collaborative filtering and topic modeling. The experiments verify that Splash can yield orders-of-magnitude speedups over single-thread stochastic algorithms and over state-of-the-art batch algorithms.

Besides its performance, Splash is a contribution on the distributed computing systems front, providing a flexible interface for the implementation of stochastic algorithms. We build Splash on top of Apache Spark [29], a popular distributed data-processing framework for batch algorithms. Splash takes the standard Resilient Distributed Dataset (RDD) of Spark as input and generates an RDD as output. The data structure also supports default RDD operators such as Map and Reduce, ensuring convenient interaction with Spark. Because of this integration, Splash works seamlessly with other data analytics tools in the Spark ecosystem, enabling a single system to address the entire analytics pipeline.

2 Strategy for Parallelization

We describe the strategy for combining parallel updates. First we introduce the operators that Splash supports for manipulating shared variables. Then we illustrate how conflicting updates are combined by the reweighting scheme.

Operators The programming interface requires the user to implement a function called `process`$(\texttt{elem}, \texttt{weight}, \texttt{var})$ which processes a weighted data element to update the set of shared variables. The user is allowed to manipulate shared variables inside their algorithm via *operators*.

An operator is a function that maps a real number to another real number. Splash supports three types of operators: *add*, *delayed add* and *multiply*. The system employs different strategies for parallelizing different types of operators.

The *add* operator is the the most commonly used operator. When the operation is performed on variable v, the variable is updated by $v \leftarrow v + \delta$ where δ is a user-specified scalar. The SGD update can be implemented using this operator.

The *delayed add* operator performs the same mapping $v \leftarrow v + \delta$; however, the operation will not be executed until the next time that the same element is processed by the system. Delayed operations are useful in implementing sampling-based stochastic algorithms. In particular, before the new value is sampled, the old value should be removed. This "reverse" operation can be declared as a delayed operator when the old value was sampled, and executed before the new value is sampled.

The *multiply* operator scales the variable by $v \leftarrow \gamma \cdot v$ where γ is a user-specified scalar. The multiply operator is especially efficient for scaling high-dimensional arrays. The array multiplication costs $\mathcal{O}(1)$ computation time, independent of the dimension of the array.

Reweighting Assume that there are m thread running in parallel. Note that all Splash operators are linear transformations. When these operators are applied sequentially, they merge into a single linear transformation. Let S_i be the sequence of samples processed by thread i, which is a fraction $1/m$ of the full sequence $\mathcal{S}$. For an arbitrary shared variable v, we can write thread i's transformation of this variable in the following form:

$$v \leftarrow \Gamma(S_i) \cdot v + \Delta(S_i) + T(S_i), \tag{1}$$

Here, both $\Gamma(S_i)$, $\Delta(S_i)$ and $T(S_i)$ are thread-level operators constructed by the execution engine: $\Gamma(S_i)$ is the aggregated multiply operator, $\Delta(S_i)$ is the term resulting from the add operators, and $T(S_i)$ is the term resulting from the delayed add operators executed in the current iteration. A detailed construction of $\Gamma(S_i)$, $\Delta(S_i)$ and $T(S_i)$ is given in the full version of this extended abstract [31].

Directly combining these transformations leads to divergence or slow convergence (or both) [31]. The reweighting scheme addresses this dilemma by assigning weights to the samples. Since the update (1) is constructed on a fraction $1/m$ of the full sequence $\mathcal{S}$, we reweight every element by m in the local sequence. After reweighting, the data distribution of S_i will approximate the data distribution of $\mathcal{S}$. If the update (1) is a (randomized) function of the data distribution of S_i, then it will approximate the full sequential update after the reweighting, thus generating a nearly unbiased update.

More concretely, the algorithm manipulates the variable by taking sample weights into account. An m-weighted sample tells the algorithm that it appears m times consecutively in the sequence. We rename the transformations in (1) by $\Gamma(mS_i)$, $\Delta(mS_i)$ and $T(mS_i)$, emphasizing that they are constructed by processing m-weighted samples. Then we redefine the transformation of thread i by

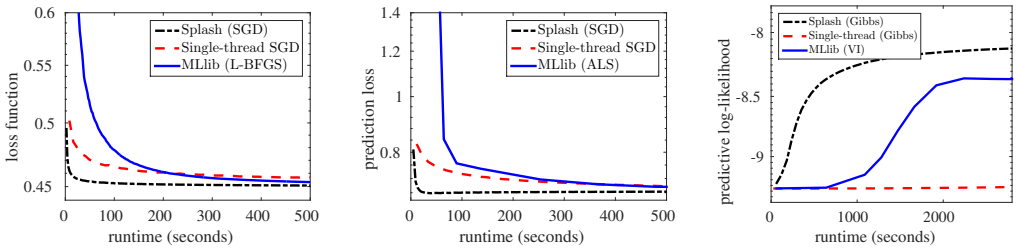
$$v \leftarrow \Gamma(mS_i) \cdot v + \Delta(mS_i) + T(mS_i) \tag{2}$$

and define the global update by

$$v_{\text{new}} = \frac{1}{m} \sum_{i=1}^{m} \Big(\Gamma(mG_i) \cdot v_{\text{old}} + \Delta(mS_i) \Big) + \sum_{i=1}^{m} T(mS_i). \tag{3}$$

Equation (3) combines the transformations of all threads. The terms $\Gamma(mS_i)$ and $\Delta(mS_i)$ are scaled by a factor $1/m$ because they were constructed on m times the amount of data. The term $T(mS_i)$ is not scaled, because the delayed operators were declared in earlier iterations, independent of the reweighting. Finally, the scaling factor $1/m$ should be multiplied to all delayed operators declared in the current iteration, because these delayed operators were also constructed on m times the amount of data.

Determining the degree of parallelism To determine the thread number m, the execution engine partitions the available cores into different-sized groups. Suppose that group i contains m_i cores. These cores will execute the algorithm tentatively on m_i parallel threads. The best thread number is then determined by cross-validation and is dynamically updated. The cross-validation requires the user to implement a loss function, which takes the variable set and an individual data element



(a) LR on MNIST 8M (b) CF on Netflix (c) LDA on NYTimes

Figure 1: Comparing Splash with baseline methods on logistic regression (LR), collaborative filtering (CF) and Topic Modelling (LDA).

as input to return the loss value. See the full version of this extended abstract [31] for a detailed description. To find the best degree of parallelism, the base algorithm needs to be robust in terms of processing a wide range of sample weights.

3 Experiments

In this section, we report the empirical performance of Splash. Our implementation of Splash runs on an Amazon EC2 cluster with eight nodes. Each node is powered by an eight-core Intel Xeon E5-2665 with 30GB of memory and was connected to a commodity 1GB network, so that the cluster contains 64 cores. For all experiments, we compare Splash with MLlib v1.3 [19] — the official distributed machine learning library for Spark. We also compare Splash against single-thread stochastic algorithms.

Logistic Regression We solve a digit recognition problem on the MNIST 8M dataset [18] using multi-class logistic regression. The dataset contains 8 million hand-written digits. Each digit is represented by a feature vector of dimension $d = 784$. Splash solves the optimization problem by SGD. We compare Splash against the single-thread SGD (with AdaGrad) and the MLlib implementation of L-BFGS [23]. Figure 1(a) shows the convergence plots of the three methods. Splash converges in a few seconds to a good solution. The single-thread AdaGrad and the L-BFGS algorithm converges to the same accuracy in much longer time. Splash is 15x - 30x faster than MLlib.

Collaborative Filtering For personalized movie recommendation, we use the Netflix prize dataset [2], which contains 100 million movie ratings made by 480k users on 17k movies. The goal is to predict the ratings in the test set given ratings in the training set. The number of parameters to be learned is 65 million. We compare Splash against the single-thread SGD method and the MLlib implementation of alternating least square (ALS) method. According to Figure 1(b), Splash converges much faster than the single-thread SGD and the ALS. This is because that SGD can learn accurate movie vectors by processing a fraction of the the data. For example, to achieve a prediction loss lower than 0.70, it takes Splash only 13 seconds, processing 60% of the training set. To achieve the same prediction loss, it takes the ALS 480 seconds, taking 40 passes over the full training set. In other words, Splash features a 36x speedup over the MLlib.

Topic Modeling We use the NYTimes article dataset from the UCI machine learning repository [15]. The dataset contains 300k documents and 100 million word tokens. The vocabulary size is 100k. The goal is to learn $K = 500$ topics from these documents. The number of parameters to be learned is 200 million. We employ the LDA model [3]. We compare Splash with the single-thread collapsed Gibbs sampling algorithm and the MLlib implementation of the variational inference (VI) method [3]. Figure 1(c) plots the predictive log-likelihoods. Among the three methods, the single-thread collapsed Gibbs sampling algorithm exhibits little progress in the first 3,000 seconds. But when the algorithm is parallelized by Splash, it converges faster and better than the MLlib implementation of variational inference (VI). In particular, Splash converges to a predictive log-likelihoods of -8.12, while MLlib converges to -8.36. When measured at fixed target scores, Splash is 3x - 6x faster than MLlib.

References

[1] A. Agarwal and J. C. Duchi. Distributed delayed stochastic optimization. In *NIPS*, pages 873–881, 2011.

[2] J. Bennett and S. Lanning. The netflix prize. In *Proceedings of KDD cup and workshop*, volume 2007, page 35, 2007.

[3] D. M. Blei, A. Y. Ng, and M. I. Jordan. Latent Dirichlet allocation. *the Journal of machine Learning research*, 3:993–1022, 2003.

[4] L. Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*, pages 177–186. Springer, 2010.

[5] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, A. Senior, P. Tucker, K. Yang, Q. V. Le, et al. Large scale distributed deep networks. In *Advances in Neural Information Processing Systems*, pages 1223–1231, 2012.

[6] J. Duchi, E. Hazan, and Y. Singer. Adaptive subgradient methods for online learning and stochastic optimization. *The Journal of Machine Learning Research*, 12:2121–2159, 2011.

[7] J. C. Duchi, A. Agarwal, and M. J. Wainwright. Dual averaging for distributed optimization: convergence analysis and network scaling. *Automatic Control, IEEE Transactions on*, 57(3):592–606, 2012.

[8] R. Gemulla, E. Nijkamp, P. J. Haas, and Y. Sismanis. Large-scale matrix factorization with distributed stochastic gradient descent. In *SIGKDD*, pages 69–77. ACM, 2011.

[9] Q. Ho, J. Cipar, H. Cui, S. Lee, J. K. Kim, P. B. Gibbons, G. A. Gibson, G. Ganger, and E. P. Xing. More effective distributed ML via a stale synchronous parallel parameter server. In *NIPS*, pages 1223–1231, 2013.

[10] M. D. Hoffman, D. M. Blei, C. Wang, and J. Paisley. Stochastic variational inference. *The Journal of Machine Learning Research*, 14(1):1303–1347, 2013.

[11] R. Johnson and T. Zhang. Accelerating stochastic gradient descent using predictive variance reduction. In *NIPS*, pages 315–323, 2013.

[12] Y. Koren, R. Bell, and C. Volinsky. Matrix factorization techniques for recommender systems. *Computer*, (8):30–37, 2009.

[13] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *NIPS*, pages 1097–1105, 2012.

[14] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su. Scaling distributed machine learning with the parameter server. In *Proc. OSDI*, pages 583–598, 2014.

[15] M. Lichman. UCI machine learning repository, 2013.

[16] C. Liu, H.-c. Yang, J. Fan, L.-W. He, and Y.-M. Wang. Distributed nonnegative matrix factorization for web-scale dyadic data analysis on mapreduce. In *WWW*, pages 681–690. ACM, 2010.

[17] J. Liu, S. J. Wright, C. Ré, V. Bittorf, and S. Sridhar. An asynchronous parallel stochastic coordinate descent algorithm. *arXiv preprint arXiv:1311.1873*, 2013.

[18] G. Loosli, S. Canu, and L. Bottou. Training invariant support vector machines using selective sampling. *Large scale kernel machines*, pages 301–320, 2007.

[19] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen, et al. Mllib: Machine learning in apache spark. *arXiv preprint arXiv:1505.06807*, 2015.

[20] T. P. Minka. Expectation propagation for approximate Bayesian inference. In *UAI*, pages 362–369. Morgan Kaufmann Publishers Inc., 2001.

[21] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 439–455. ACM, 2013.

[22] D. Newman, P. Smyth, M. Welling, and A. U. Asuncion. Distributed inference for latent Dirichlet allocation. In *NIPS*, pages 1081–1088, 2007.

[23] J. Nocedal and S. J. Wright. *Numerical Optimization*. Springer, New York, 2nd edition, 2006.

[24] B. Recht, C. Re, S. Wright, and F. Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *NIPS*, pages 693–701, 2011.

[25] M. Schmidt, N. L. Roux, and F. Bach. Minimizing finite sums with the stochastic average gradient. *arXiv preprint arXiv:1309.2388*, 2013.

[26] P. Vincent, H. Larochelle, Y. Bengio, and P.-A. Manzagol. Extracting and composing robust features with denoising autoencoders. In *ICML*, pages 1096–1103. ACM, 2008.

[27] L. Xiao. Dual averaging method for regularized stochastic learning and online optimization. In *NIPS*, pages 2116–2124, 2009.

[28] E. P. Xing, Q. Ho, W. Dai, J. K. Kim, J. Wei, S. Lee, X. Zheng, P. Xie, A. Kumar, and Y. Yu. Petuum: A new platform for distributed machine learning on big data. *arXiv preprint arXiv:1312.7651*, 2013.

[29] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*. USENIX Association, 2012.

[30] T. Zhang. Solving large scale linear prediction problems using stochastic gradient descent algorithms. In *ICML*, page 116. ACM, 2004.

[31] Y. Zhang and M. I. Jordan. Splash: User-friendly programming interface for parallelizing stochastic algorithms. *arXiv preprint arXiv:1506.07552*, 2015.

[32] Y. Zhang, M. J. Wainwright, and J. C. Duchi. Communication-efficient algorithms for statistical optimization. In *NIPS*, pages 1502–1510, 2012.

[33] Y. Zhuang, W.-S. Chin, Y.-C. Juan, and C.-J. Lin. A fast parallel SGD for matrix factorization in shared memory systems. In *RecSys*, pages 249–256. ACM, 2013.

[34] M. Zinkevich, M. Weimer, L. Li, and A. J. Smola. Parallelized stochastic gradient descent. In *NIPS*, pages 2595–2603, 2010.