ASYNC: A Cloud Engine with Asynchrony and History for Distributed Machine Learning

Saeed Soori*, Bugra Can†, Mert Gurbuzbalaban‡ and Maryam Mehri Dehnavi§

* § Department of Computer Science, University of Toronto, Toronto, Canada

† ‡ Department of MBS, Rutgers Univeristy, New Jersey, USA

Email: *sasoori@cs.toronto.edu, †bugra.can@rutgers.edu, ‡mert.gurbuzbalaban@rutgers.edu, §mmehride@cs.toronto.edu

Abstract—**ASYNC is a framework that supports the implementation of asynchrony and history for optimization methods on distributed computing platforms. The popularity of asynchronous optimization methods has increased in distributed machine learning. However, their applicability and practical experimentation on distributed systems are limited because current bulk-processing cloud engines do not provide a robust support for asynchrony and history. With introducing three main modules and bookkeeping system-specific and application parameters, ASYNC provides practitioners with a framework to implement asynchronous machine learning methods. To demonstrate ease-of-implementation in ASYNC, the synchronous and asynchronous variants of two well-known optimization methods, stochastic gradient descent and SAGA, are demonstrated in ASYNC.**

Index Terms—**Machine learning, cloud computing**

I. INTRODUCTION

Distributed optimization has gained significant traction in recent years and is frequently used to solve modern large-scale machine learning problems [1]. The challenges of dealing with huge datasets, has lead to the development of optimizations methods with *asynchrony* and *history*. Asynchronous optimization methods reduce worker idle times and mitigate communication costs. Operations on a history of gradients augments the noise (stochasticity) to improve convergence [2]. Distributed optimization methods operate on batches of data and thus have to be implemented in cluster-computing engines with a bulk (coarse-grained) computation model.

There exists several general coarse-grained distributed data processing systems. Hadoop [3] and Spark [4] are based on the iterative map-reduce model but use a synchronous iterative communication pattern. Thus, because of not supporting asynchrony, their execution is vulnerable to the diverse performance profile caused by slow workers, i.e. stragglers, and network latency in a distributed platform. Also history can not be efficiently maintained in these engines as it requires storing *bulky worker-results*, and introduces overheads to their lineage-based [4] or checkpointing fault tolerance implementations.

Recently, a number of coarse-grained machine learning engines such as Petuum [5] and Litz [6], have adopted the parameter server [7] architecture to implement asynchronous communication between nodes with push-pull operations. Asynchrony in distributed optimization methods is implemented with consistency models, i.e. barriers, expressed via a dependency graph that maintain a trade-off between system efficiency and algorithm convergence. Parameter server paradigms implement a specific class of consistency models, i.e. stale synchronous parallel (SSP) paradigms using a fixed dependency graph, which use a static staleness threshold to control worker wait times. However, recent advancements in distributed optimization [8], [9] demand for wider range of customized consistency models (CCMs), often defined by the user such as throttled-release [9], that control worker wait times using parameters such as worker-task-completion time [8] and require to adaptively adjust the parameters at runtime. CCMs can not be implemented in available parameter server frameworks as it needs the underlying dependency graph to adaptively be reconfigured at runtime. Also, Petuum does not support history and Litz preserves the history by periodic checkpointing with significant overheads. Other distributed parameter-server frameworks such as DistBelief [10] and TensorFlow [11] are specialized for deep learning applications and thus do not naturally support consistency models and history.

Amongst the fine-grained distributed data processing systems, primarily used for streaming applications, RAY [12] and Flink [13] support asynchronous function invocations with dynamic data flow graphs [14]. However, these frameworks do not support CCMs and are primarily designed for fine-grained tasks, and thus can not naturally extend to a bulk-processing engine. Also, while streaming engines (because of processing fine tasks) can store local results and intermediate data on workers to support history with low-overhead, bulk processing engines can not efficiently store the worker-results because of processing coarse tasks.

In principle, with massive system engineering efforts, machine learning practitioners can implement one-off asynchronous optimization methods with re-engineering systems and interfaces. However, this comes at the cost of pushing system challenges such as scheduling, bookkeeping, and fault tolerance to the application developer. For example, Spark can support history if previous worker-results are stored to disk and checkpointed; this will create large storage overheads. Supporting CCMs is more challenging as the entire Spark engine has to change to support asynchronous execution. An expert MPI programmer can use asynchronous primitives to implement SSP [15]. However, this leads to increased program complexity and the complexity will increase if customized consistency models were to be implemented. Noteworthy, MPI does not have a robust support for fault tolerance and thus is

typically not used for cloud computing.

This work presents ASYNC, a bulk processing cloud-computing framework, built on top of Spark, that supports the implementation of distributed optimization methods with asynchrony and history. ASYNC implements an asynchronous execution to Spark's engine and enables the workers and/or the master to *bookkeep* (log) system-specific and application parameters. The asynchronous execution paradigm and the bookkeeping structures work together to construct a dynamic dependence graph for the implementation of custom consistency models and to recover history with a partial broadcast of model parameters. Major contributions of this paper are:

- A novel framework for machine learning practitioners to implement and dispatch asynchronous machine learning applications with custom consistency models on cloud and distributed platforms. ASYNC introduces three modules to cloud engines, *ASYNCcoodinator*, *ASYNCbroadcaster*, and *ASYNCscheduler* to enable the asynchronous gather, broadcast, and schedule of tasks and results.
- A efficient history recovery strategy implemented with the *ASYNCbroadcaster* and bookkeeping *attributes*, to facilitate the implementation of variance reduced optimization methods that operate on historical gradients.
- A robust programming model with extensions to the Spark API that enables the implementation of asynchrony and history while preserving the in-memory and fault tolerant features of Spark.
- A demonstration of ease-of-implementation in ASYNC with the implementation and performance analysis of the stochastic gradient descent (SGD) [1] algorithm and its asynchronous variant using a CCM. Also, the implementation of the history-based optimization method SAGA [16] and its asynchronous variant in ASYNC. Our results demonstrate that asynchronous SAGA (ASAGA) [17] and asynchronous SGD (ASGD) outperform their synchronous variants up to 4 times on a distributed system with stragglers.

II. PRELIMINARIES

Distributed machine learning often results in solving an optimization problem in which an objective function is optimized by iteratively updating the *model parameters* until convergence. Distributed implementation of optimization methods includes workers that are assigned tasks to process parts of the training data, and one or more servers, i.e. masters, that store and update the model parameters. Distributed machine learning models often result in the following structure:

$$\min_{w \in \mathbb{R}^d} F(w) = \frac{1}{m} \sum_{i=1}^{m} f^{(i)}(w) \tag{1}$$

where w is the model parameter to be learned, m is the number of workers, and $f^{(i)}(w)$ is the local loss function computed by worker i based on its assigned training data. Each worker has access to n_i data points, where the local cost has the form

$$f^{(i)}(w) := \sum_{j=1}^{n_i} \bar{f}_j^{(i)}(w) \tag{2}$$

for some loss functions $\bar{f}_j^{(i)} : \mathbb{R}^d \to \mathbb{R}$. For example, in supervised learning, given an input-output pair (x_{ij}, y_{ij}), the loss function can be $\bar{f}_j^i(w) = \ell(\langle w, \phi(x_{ij})\rangle, y_{ij})$ where ϕ is a fixed function of choice and $\ell(\cdot, \cdot)$ is a convex loss function that measures the loss if y_{ij} is predicted from x_{ij} based on the model parameter w. This setting covers empirical risk minimization problems in machine learning that include linear and non-linear regression, and other classification problems such as logistic regression [2]. In particular, if $\phi(x) = x$ and the $\ell(\cdot, \cdot)$ function is the square of the Euclidean distance function, we obtain the familiar least squares problem

$$\bar{f}_j^i(w) = \|x_{ij}^T w - y_{ij}\|^2 \tag{3}$$

where

$$f^{(i)}(w) := \sum_{j=1}^{n_i} \bar{f}_j^{(i)}(w) = \|A_i w - b_i\|^2 \tag{4}$$

with $b_i = \{y_{ij}\}_{j=1}^{n_i}$ is a column vector of length n_i and $A_i \in \mathbb{R}^{n_i \times d}$ is called the *data matrix* as its j-th row is given by the input x_{ij}^T.

In the following we use the gradient descent (GD) algorithm as an example to introduce stochastic optimization and other terminology used throughout the paper such as *mini-batch* size. The introduced terms are used in all optimization problems and are widely used in the machine learning literature. GD iteratively computes the gradient of the loss function $\nabla F(w_k) = \frac{1}{m} \sum_{i=1}^{m} \nabla f^{(i)}(w_k)$ to update the model parameters at iteration k. To implement gradient descent on a distributed system, each worker i computes its *local gradient* $\nabla f^{(i)}(w^k)$; the local gradients are aggregated by the master when ready. The full pass over the data at every iteration of the algorithm with synchronous updates leads to large overheads. Distributed stochastic gradient descent (SGD) methods and their variants [18] are on the other hand scalable and popular methods for solving (1). Distributed SGD replaces the local gradient $\nabla f^{(i)}(w_k)$ with an unbiased stochastic estimate $\tilde{\nabla} f^{(i)}(w_k)$ of it, computed from a subset of local data points:

$$\nabla \tilde{f}^{(i)}(w_k) := \frac{1}{b_i} \sum_{s \in S_{i,k}} \nabla \bar{f}_s^{(i)}(w_k), \tag{5}$$

where $S_{i,k} \subset \{1, \ldots, n_i\}$ is a random subset that is sampled with or without replacement at iteration k, and $b_i := |S_{i,k}|$ is the number of elements in $S_{i,k}$ [1], also called the mini-batch size. To obtain desirable accuracy and performance, implementations of stochastic optimization methods require tuning algorithm parameters. For example, the step size and the mini-batch sizes are parameters to tune in SGD [1].

III. MOTIVATION FOR ASYNCHRONY AND HISTORY

Asynchrony is implemented to improve the converge rate and time-to-solution of optimization methods on cluster-computer platforms with slow machines (stragglers). In distributed optimization, workers compute local gradients of

Fig. 1: An overview of the ASYNC framework.

the objective function and then communicate the computed gradients to the server. To proceed to the next iteration of the algorithm, the server updates the shared model parameters with the received gradients, broadcasts the most recent model parameter, and schedules new tasks. In asynchronous optimization, the server can proceed with the update and broadcast of the model parameters without having to wait for all worker tasks to complete. This asynchrony allows the algorithm to make progress in the presence of stragglers which is known as an increase in hardware efficiency [19]. However, this progress in computation comes at a cost, the asynchrony inevitably adds *staleness* to the system wherein some of the workers compute gradients using model parameters that may be several gradient steps behind the most updated set of model parameters which can lead to poor convergence. This is also referred to as a worsening in statistical efficiency [18].

Asynchronous optimization methods are formulated and implemented with properties that balance statistical efficiency and hardware efficiency to maximize the performance of the optimization methods on distributed systems. Consistency models, i.e. barrier control strategies, are used to design asynchronous optimization methods that enable this balance. Barriers in asynchronous algorithms determine if a worker should proceed to the next iteration or if it should wait until a specific number of workers have communicated their results to the server. The most well-known barrier control strategy is the Stale Synchronous Parallel (SSP) in which workers synchronize when staleness (determined by the number of stragglers) exceeds a threshold. ASYNC supports SSP and also facilitates the implementation of custom consistency models that apply barriers based on parameters such as worker-task-completion time and scheduling delays.

History augments the noise from stochastic gradients to improve the convergence rate of the optimization method. Distributed optimization methods used in machine learning applications are typically stochastic [1]. Stochastic optimization methods use a noisy gradient computed from random

data samples instead of the true gradient which can lead to poor convergence. Variance reduction techniques, used in both synchronous and asynchronous optimization, augment the noisy gradient to reduce this variance. A class of variance-reduced asynchronous algorithms that have led to significant improvements over traditional methods memorize the gradients computed in previous iterations, i.e. historical gradients [16]. Historical gradients can not be implemented in cluster-computing engines such as Spark primarily because Spark can only broadcast the entire history of the model parameters which can be very large and can lead to significant overheads.

IV. ASYNC: A CLOUD COMPUTING FRAMEWORK WITH ASYNCHRONY AND HISTORY

ASYNC is a framework, built on top of Spark [4], for the implementation and execution of asynchrony and history in optimization algorithms while retaining the map-reduce model, scalability, and fault tolerance of state-of-the-art cluster computing engines. Figure 1 demonstrates an overview of the ASYNC engine. The three main modules in ASYNC are the *ASYNCcoordinator*, *ASYNCbroadcaster*, *ASYNCscheduler*. ASYNC also collects and stores *bookkeeping structures*. These structures are communicated between the workers and the master and are either system-specific, i.e. *status*, or are related to the application, i.e. *attributes*. This section elaborates how the internal elements of ASYNC work together to facilitate the implementation of asynchrony and history.

Bookkeeping structures in ASYNC. Bookkeeping structures are used by the main modules of ASYNC to enable the implementation of asynchrony and history. These structures are collected by ASYNC at runtime and are stored on the master. With the help of the ASYNCcoordinator, each worker communicates to the master, application-specific attributes such as task results and the mini-batch size. Workers' recent status such as worker staleness, average-task-completion time, and availability[1] are also logged and stored in a table called

---

[1]A worker is available if it is not executing a task and unavailable otherwise

TABLE I: Transformations, actions, and methods in ASYNC. AC is the ASYNCcontext and Seq[T] is a sequence of elements of type T.

Actions	ASYNCreduce(f:(T,T) $\Rightarrow$ T, AC)	Reduces the elements of the RDD using the specified associative binary operator.
	ASYNCaggregate(zeroValue: U) (seqOp: (U, T) $\Rightarrow$ U, combOp: (U, U) $\Rightarrow$ U), AC)	Aggregates the elements of the partition using the combine functions and a neutral "zero" value.
Transformations	ASYNCbarrier(f:T $\Rightarrow$ Bool, Seq[T])	Returns a RDD containing elements that satisfy a predicate f.
Methods	ASYNCcollect()	Returns a task result.
	ASYNCcollectAll()	Returns a task result and its attributes.
	ASYNCbroadcast(T)	Creates a dynamic broadcast variable.
	AC.STAT	Returns the current status of all workers.
	AC.hasNext()	Returns true if a task result exists.

STAT with the help of the ASYNCcoordinator.

Implementing asynchrony with the ASYNCcoordinator, ASYNCscheduler, and the status structures. To implement asynchrony, ASYNC implements a dynamic task graph computation model which uses the consistency model to dynamically determine executing tasks and their assigned workers. The execution of tasks on workers is automatically triggered by the system using a computation graph. Task and data objects are the nodes in this graph and the edges are the dependency amongst nodes/tasks. The computation graph in classic consistency models such as SSP does not change at runtime because the models do not rely on runtime information such as the system state. However, many CCMs take information from the current state of the system as input and couple this information with the barrier control strategy to dynamically build the computation graph. To implement CCMs, the ASYNCcoordinator periodically communicates with the workers to update system-specific parameters in the STAT table. The ASYNCscheduler uses the parameters in STAT and a user-defined barrier control strategy to update the computation graph. The computation graph is then executed to apply the desired constancy model.

Implementing history with the ASYNCbroadcaster and the attributes. In each iteration of an optimization method that uses history, the computed gradients from previous iterations are used together with the current model parameters to update the model parameters. Implementing history in a coarse-grained computation engine via explicitly storing bulky worker-results, i.e. previous gradients, leads to significant storage overheads. A fault tolerant execution will also have overheads in this approach as large gradients have to be periodically checkpointed or recomputed explicitly using a lineage.

ASYNC does not explicitly store, communicate, or compute past gradients. Instead we use the approach from [2] in which the history of past gradients is recovered, when needed, using previous model parameters. Recovering history has low storage and computation overheads in coarse-grained computation models. By recovering history, workers in ASYNC do not need to store any previously computed gradients and only the previous model parameters are stored on the master. The cost of storing the model parameters has an inverse relation to the batch size [1] and thus reduces as the granularity of tasks increase, e.g. larger batch sizes. Also, to recover a past gradient, a worker only needs to subtract its recent

model parameter from the previous model parameter that is broadcasted to it from the master; the approach in [2] is then used to update the master-side model parameters based on the history. The ASYNCbroadcaster in ASYNC is responsible for the asynchronous broadcast of model parameters between the master and individual workers. Attributes such as the minibatch size, required by the master to apply history to its model parameters, are also broadcast using the ASYNCbroadcaster.

V. PROGRAMMING WITH ASYNC

To use ASYNC, developers are provided an additional set of ASYNC-specific functions, on top of what Spark provides, to access the bookkeeping structures and to implement asynchrony and history. The programming model in ASYNC is close to that of Spark. It operates on resilient distributed datasets (RDD) to preserve the fault tolerant and in-memory execution of Spark. The ASYNC-specific functions also either transform the RDDs, known as *transformations* in Spark, or conduct lazy actions. In this section, ASYNC's programming model and API is first discussed. We then show the implementation of SGD and its asynchronous variant which uses a CCM. A well-known history-based optimization method called SAGA [16] and its asynchronous variant with a CCM is also implemented. Finally, we discuss the implementation of other consistency models in ASYNC.

A. The ASYNC programming model

Asynchronous Context (AC) is the entry point to ASYNC and should be created only once in the beginning of the application. The ASYNCscheduler, the ASYNCbroadcaster, and the ASYNCcoordinator communicate via the AC and with this communication create barrier controls, broadcast variables, and store workers' task results and status. AC maintains the bookkeeping structures and ASYNC-specific functions, including actions and transformations that operate on RDDs. Workers use ASYNC functions to interact with AC and to store their results and attributes in the bookkeeping structures. The server queries AC to update the model parameters or to access workers' status. Table I lists the main functions available in ASYNC. We show the signature of each operation by demonstrating the type parameters in square brackets.

Collective operations in ASYNC. ASYNCreduce is an action that aggregates the elements of the RDD on the worker and returns the result to the server. ASYNCreduce differs from

Spark's *reduce* in two ways. First, Spark aggregates data across each partition and then combines the partial results together to produce a final value. However, ASYNCreduce executes only on the worker and for each partition. Secondly, *reduce* returns only when all partial results are combined on the server, but ASYNCreduce returns immediately. Task results on the server are accessed using the *ASYNCcollect* and *ASYNCcollectAll* methods. ASYNCcollect returns task results in FIFO (first-in-first-out) order and also returns the worker attributes. The workers' status can also be accessed with *ASYNC.STAT*.

Algorithm 1: The SGD Algorithm

Input : points, numIterations, learning rate α_i, sampling rate b
Output: model parameter w
1 **for** $i = 1$ to numIterations **do**
2   w_br = sc.broadcast(w)
3   gradient = points.sample(b).map(p $\Rightarrow \nabla f_p(w\_br.value)$). reduce(_+_)
4   w -= $\alpha_i *$ gradient
5 **end**
6 return w

Algorithm 2: The ASGD Algorithm

Input : points, numIterations, learning rate α_i, sampling rate b
Output: model parameter w
1 AC = new ASYNCcontext
2 **for** $i = 1$ to numIterations **do**
3   w_br = sc.broadcast(w)
4   points.ASYNCbarrier(f, AC.STAT).sample(b).map(p $\Rightarrow \nabla f_p(w\_br.value)$) .ASYNCreduce(_+_, AC)
5   **while** *AC.hasNext()* **do**
6   gradient= AC.ASYNCcollect()
7   w -= $\alpha_i *$ gradient
8   **end**
9 **end**
10 return w

Algorithm 3: The SAGA Algorithm

Input : points, numIterations, learning rate α, sampling rate b, number of points n
Output: model parameter w
1 averageHistory = 0
2 store w in table
3 **for** $i = 1$ to numIterations **do**
4   w_br =sc.broadcast(w)
5   (gradient, history)= points.sample(b).map((index,p) $\Rightarrow \nabla f_p(w\_br.value), \nabla f_p(table[index])$).reduce(_+_)
6   averageHistory += (gradient - history)$* b*n$
7   w -= $\alpha *$ (gradient - history + averageHistory)
8   update table
9 **end**
10 return w

Barrier and broadcast in ASYNC. ASYNCbarrier is a *transformation*, i.e. a deterministic operation which creates a new RDD based on the workers' status. ASYNCbarrier takes the recent status of workers.*STAT* and decides which workers to assign new tasks to, based on a user-defined function. For

---

Algorithm 4: The ASAGA Algorithm

Input : points, numIterations, learning rate α, sampling rate b, #points n, #partitions P
Output: model parameter w
1 AC = new ASYNCcontext
2 averageHistory = 0
3 **for** $i = 1$ to numIterations **do**
4   w_br = AC.ASYNCbroadcast(w)
5   points.ASYNCbarrier(f, AC.STAT) .sample(b).map((index,p) $\Rightarrow \nabla f_p(w\_br.value),$ $\nabla f_p(w\_br.value(index))$). ASYNCreduce(_+_, AC)
6   **while** *AC.hasNext()* **do**
7   (gradient,history)= AC.ASYNCcollect()
8   averageHistory += (gradient - history)$* b*n/P$
9   w -= $\alpha *$ (gradient - history + averageHistory)
10   **end**
11 **end**
12 return w

---

example, for a fully asynchronous barrier model the following function is declared: $f : STAT.foreach(true)$. In Spark, broadcast parameters are "broadcast variable" objects that wrap around the to-be-broadcast value. *ASYNCBroadcast* also uses broadcast variables and similar to Spark the method *value* can be used to access the broadcast value. However, ASYNCbroadcast differs from the broadcast implementation in Spark since it has access to an *index*. The index is used internally by ASYNCbroadcast to get the ID of the previously broadcast variables for the specified index. ASYNCbroadcast eliminates the need to broadcast values when accessing the history of broadcast values.

B. Case studies

The robust programming model in ASYNC provides control of low-level features in both the algorithm and the execution platform to facilitate the implementation of asynchrony and history in optimization methods. The following demonstrates the implementation of well-known asynchronous optimization methods ASGD and ASAGA in ASYNC as examples.

ASGD with ASYNC. An implementation of mini-batch stochastic gradient descent (SGD) using the map-reduce model in Spark is shown in Algorithm 1. The map phase applies the gradient function on the input data independently on workers. The reduce phase has to wait for all the map tasks to complete. Afterwards, the server aggregates the task results and updates the model parameter w. The asynchronous implementation of SGD in ASYNC is shown in Algorithm 2. With only a few extra lines from the ASYNC API, colored in blue, the synchronous implementation of SGD in Spark is transformed to ASGD. An ASYNCcontext is created in line 1 and is used in line 4 to create a barrier using the user-defined CCM indicated by f and based on the current workers' status, AC.STAT. The partial results from each partition are then obtained and stored in AC in line 4. Finally, these partial results are accessed in line 6 and are used to update the model parameter in line 7.

ASAGA with ASYNC. The SAGA implementation in Spark is shown in Algorithm 3. This implementation is inefficient and

not practical for large datasets as it needs to synchronously broadcast a table of all stored model parameters to each worker, colored in red in Algorithm 3 line 5. The size of this increases after each iteration and thus broadcasting it leads to large communication overheads. As a result of the overhead, machine learning libraries that are build on top of Spark such as Mllib [20] do not provide implementations of optimization methods such as SAGA that requires the history of gradients. ASYNC resolves the overhead with ASYNCbroadcast. The implementation of ASAGA is shown in Algorithm 4. ASYNCbroadcast is used to define a dynamic broadcast in line 4. Then, the broadcast variable is used to compute the historical gradients in line 5. In order to access the last model parameters for sample *index*, the method *value* is called in line 5. As shown in Algorithm 4, there is no need to broadcast a table of parameters which allows for efficient implementation of both SAGA and ASAGA in ASYNC.

CCMs in ASYNC. To enable the implementation of custom consistency models, ASYNC provides the interface to implement user-defined functions that selectively choose from available workers based on their status. Listing 1 demonstrates the implementation of two CCMs in ASYNC, CCM1 and CCM2, as well as the SSP model. CCM1 is the throttled-release [9] barrier strategy which submits tasks to available workers only when the number of available workers is at least k. CCM2 implements a fully asynchronous barrier that allows workers to progress as soon as their current task finishes.

```
f: STAT.foreach(Avaialble_Workers >= k) % CCM1
f: STAT.foreach(true) % CCM2
f: STAT.foreach(MAX_Staleness < s) % The SSP
   barrier control with a staleness
   threshold 's'
points.ASYNCbarrier(f, AC.STAT) % Apply the
   barrier
```

Listing 1: Pseudo-code for implementing CCMs in ASYNC.

VI. RESULTS

We evaluate the performance of ASYNC by implementing two asynchronous optimization methods, namely ASGD and ASAGA, and their synchronous variants to solve least squares problems. We implement the throttled-release CCM for the both asynchronous methods and use history in ASAGA and SAGA. The performance of ASGD and ASAGA are compared to their synchronous implementations in Spark. To the best of our knowledge, no library or implementation of asynchronous optimization methods exists on Spark. However, to demonstrate that the synchronous implementations of the algorithms using ASYNC are well-optimized, we first compare the performance of the synchronous variants of the tested optimization methods in ASYNC with the state-of-the-art machine learning library, Mllib [20]. Mllib is a library that provides implementations of a number of synchronous optimization methods. In subsection VI-C we evaluate the performance of ASGD and ASAGA in ASYNC in the presence of stragglers.

Dataset	Row numbers	Column numbers	Size
rcv1_full.binary	697,641	47,236	851.2MB
mnist8m	8,100,000	784	19GB
epsilon	400,000	2000	12.16GB

TABLE II: Datasets for the experimental study.

Fig. 2: The performance of SGD implemented in ASYNC versus Mllib.

A. Experimental setup

We consider the distributed least squares problem defined in (4). Our experiments use the datasets listed in Table II from the LIBSVM library [21], all of which vary in size and sparsity. The first dataset rcv1_full.binary is about documents in the Reuters Corpus Volume I (RCV1) archive, which are newswire stories. The second dataset mnist8m contains handwritten digits commonly used for training various image processing systems, and the third dataset epsilon is the Pascal Challenge 2008 that predicts the presence/absence of an object in an image. For the experiments, we use ASYNC, Scala 2.11, Mllib [20], and Spark 2.3.2. Breeze 0.13.2 and netlib 1.1.2 are used for the (sparse/dense) BLAS operations in ASYNC. XSEDE Comet CPUs [22] are used to assemble the cluster.

To demonstrate the performance of asynchronous algorithms and their robustness to the heterogeneity in cloud environments, we evaluate the implemented methods in the presence of stragglers. Two different straggler behaviours are used: *(i) Controlled Delay Straggler (CDS)* experiments in which a single worker is delayed with different intensities; *(ii)* the *Production Cluster Stragglers (PCS)* experiments in which straggler patterns from real production clusters are used. The CDS experiments are ran with all three datasets on a cluster composed of a server and 8 workers. The PCS experiments require a larger cluster and thus are conducted on a cluster of 32 workers with one server using the two larger datasets (mnist8m and epsilon). In all configurations a worker runs an executor with 2 cores. The number of data partitions is 32 for all datasets and in the implemented algorithms. The

(a) mnist8m   (b) epsilon   (c) rcv1_full.binary

Fig. 3: The performance of ASGD and SGD in ASYNC with 8 workers for different delay intensities of 0%, 30%, 60% and 100% which are shown with ASYNC/SYNC, ASYNC/SYNC-0.3, ASYNC/SYNC-0.6 and ASYNC/SYNC-1.0 respectively.

experiments are repeated three times; the average reported.

Parameter tuning: A sampling rate of *b = 10%* is selected for the mini-batching SGD for *mnist8m* and *epsilon* and *b = 5%* is used for *rcv1_full.binary*. SAGA and ASAGA use *b = 10%* for *epsilon*, *b = 2%* for *rcv1_full.binary*, and use *b =1%* for *mnist8m*. For the PCS experiment, we use *b = 1%* for *mnist8m* and *epsilon*. We use the same step size as Mllib and tune it for SGD to converge faster. A fixed step size is used in SAGA which is also tuned for faster convergence. The step size is not tuned for the asynchronous algorithms. Instead, we use the following heuristic, the step size of ASGD and ASAGA is computed by dividing the initial step size of their synchronous variants by the number of workers [23]. We run the SGD algorithm in Mllib for 15000 iterations with sampling rate of 10% and use its final objective value as the *baseline* for the least squares problem.

B. Comparison with Mllib

We use ASYNC for implementations of both the synchronous and the asynchronous variants of the algorithms because *(i)* ASYNC's performance for synchronous methods is similar to that of Mllib's; *(ii)* asynchronous methods are not supported in Mllib; *(iii)* synchronous methods that require history of gradients can not be implemented in Mllib because of discussed overheads. To demonstrate that our implementations in ASYNC are optimized, we compare the performance of SGD in ASYNC and Mllib for solving the least squares problem [24]. Both implementations use the same initial step size. The *error* is defined as *objective function value* minus the *baseline*. Figure 2 shows the error for three different datasets. The figure demonstrates that SGD in ASYNC has a similar performance to that of Mllib's on 8 workers, the same pattern is observed on 32 workers. Therefore, for the rest of the experiments, we compare the asynchronous and synchronous implementations in ASYNC.

C. Robustness to stragglers

Controlled Delay Straggler: We demonstrate the effect of different delay intensities in a single worker on SGD, ASGD,

Fig. 4: Average wait time per iteration with 8 workers for ASGD and SGD in ASYNC for different delay intensities.

SAGA, and ASAGA by simulating a straggler with controlled delay [19], [25]. From the 8 workers in the cluster, a delay between 0% to 100% of the time of an iteration is added to one of the workers. The delay intensity, which we show with *delay-value %*, is the percentage by which a worker is delayed, e.g. a 100% delay means the worker is executing jobs at half speed. The controlled delay is implemented with the sleep command. The first 100 iterations of both the synchronous and asynchronous algorithms are used to measure the average iteration execution time.

	SAGA	ASAGA	SGD	ASGD
mnist8m	42.8367 ms	9.8125 ms	6.4433 ms	3.5745 ms
epsilon	6.9926 ms	1.1721 ms	5.3112 ms	1.4165 ms

TABLE III: Average wait time per iteration on 32 workers.

The performance of SGD and ASGD for different delay intensities are shown in Figure 3 where for the same delay intensity the asynchronous implementation always converges

(a) mnist8m   (b) epsilon   (c) rcv1_full.binary

Fig. 5: The performance of ASAGA and SAGA in ASYNC for different delay intensities of 0%, 30%, 60% and 100% which are shown with ASYNC/SYNC, ASYNC/SYNC-0.3, ASYNC/SYNC-0.6 and ASYNC/SYNC-1.0 respectively.

faster to the optimal solution compared to the synchronous variant of the algorithm. As the delay intensity increases, the straggler has a more negative effect on the runtime of SGD. However, ASGD converges to the optimal point with almost the same rate for different delay intensities. This is because the ASYNCscheduler continues to assign tasks to workers without having to wait for the straggler. When the task result from the straggling worker is ready, it independently updates the model parameter. Thus, while ASGD in ASYNC requires more iterations to converge, its overall runtime is considerably faster than the synchronous method. With a delay intensity of %100, a speedup of up to $2\times$ is achieved with ASGD vs. SGD.

Figure 4 shows the average wait time for each worker over all iterations for SGD and ASGD. The wait time is defined as the time from when a worker submits its task result to the server until it receives a new task. In the asynchronous algorithm, workers proceed without waiting for stragglers. Thus the average wait time does not change with changes in delay intensity. However, in the synchronous implementation worker wait times increase with a slower straggler. For example, for the *mnist8m* dataset in Figure 4, the average wait time for SGD increases significantly when the straggler is two times slower (delay = 100%). Comparing Figure 3 with Figure 4 shows that the overall runtime of ASGD and SGD is directly related to their average wait time where an increase in the wait time negatively affects the algorithms convergence rate.

The slow worker pattern used for the ASGD experiments is also used for ASAGA. Figure 5 shows experiment results for SAGA and ASAGA. The communication pattern in ASAGA is different from ASGD because of the broadcast required to compute historical gradients. In ASAGA, the straggler and its delay intensity only affects the computation time of a worker and does not change the communication cost. Therefore, the delay intensity does not have a linear effect on the overall runtime. However, Figure 5 shows that increasing the delay intensity negatively affects the convergence rate of SAGA while the ASAGA algorithm maintains the same convergence rate for different delay intensities.

Fig. 6: Average wait time per iteration with 8 workers for ASAGA and SAGA in ASYNC for different delay intensities.

The workers' average wait time for ASAGA is shown in Figure 6. With an increase in delay intensity, workers in SAGA wait more for new tasks. The difference between the average wait time of SAGA and ASAGA is more noticeable when the delay increases to 100%. In this case, the computation time is significant enough to affect the performance of the synchronous algorithm, however, ASAGA has the same wait time for all delay intensities.

Production Cluster Stragglers: Our PCS experiments are conducted on 32 workers with straggler patterns in real production clusters [26], [27]; these clusters are used frequently by machine learning practitioners. We use the straggler behaviors reported in previous research [28], [29] all of which are based on empirical analysis of production clusters from Microsoft Bing [27] and Google [26]. Empirical analysis from production clusters concluded that approximately 25% of machines in cloud clusters are stragglers. From those, 80% have a uniform probability of being delayed between 150% to

Fig. 7: The performance of ASGD and SGD in ASYNC on 32 workers shown with ASYNC and SYNC respectively.

Fig. 8: The performance of ASAGA and SAGA in ASYNC on 32 workers shown with ASYNC and SYNC respectively.

250% of average-task-completion time. The remaining 20% of the stragglers have abnormal delays and are known as Long Tail workers. Long tail workers have a random delay between 250% to 10×. From the 32 workers in our experiment, 6 are assigned a random delay between 150%-250% and two are long tail workers with a random delay over 250% up to 10×. The randomized delay seed is fixed across three executions of the same experiment.

The performance of SGD and ASGD on 32 workers with PCS is shown in Figure 7. As shown, ASGD converges to the solution considerably faster that SGD and leads to a speedup of 3× for *mnist8m* and 4× for *epsilon*. From Figure 8, ASAGA compared to SAGA obtains a speedup of 3.5× and 4× for *mnist8m* and *epsilon* respectively. The average wait time for both algorithms on 32 workers is shown in Table III. The wait time increases considerably for all synchronous implementations which results in slower convergence of the synchronous methods.

VII. RELATED WORK

To mitigate the negative effects of stale gradients on convergence, numerous optimization methods support asynchrony. The most widely used optimization algorithms with asynchrony are stochastic gradient methods [10], [23] and coordinate descent algorithms [30]. Other work implement asynchrony by altering the execution bound staleness [19], [31], by theoretically adapting the method to the stale gradients [32], and by using barrier control strategies [8], [33]. Variance reduction approaches use the history of gradients to reduce the variance incurred by stochastic gradients and to improve convergence [2], [24], [34]. Numerous algorithms implement variance reduction techniques in asynchronous methods, some of which include ASAGA and DisSVRG [34] which supports asynchrony in convex and non-convex problems.

The demand for large-scale machine learning has led to the development of numerous cloud and distributed computing frameworks. Commodity distributed dataflow systems such as Hadoop [3] and Spark [4], as well as libraries implemented on top of them such as Mllib [20], are optimized for coarse-grained, often bulk synchronous, parallel data transformations and thus do not provide asynchrony in their execution models [3], [4], [35], [36]. Recent work has modified frameworks such as Spark to support asynchronous optimization methods. ASIP [37] introduces a communication layer to Spark to support asynchrony, however, it only implements the asynchronous parallel consistency model [5] and does not support history. Glint [7] integrates the parameter server model on Spark. However, it is designed for topic models with a specialized consistency model.

Parameter server architectures such as [5], [6] are widely used in distributed machine leaning since they support asynchrony in their execution models using a static dependency graph. Petuum [5] implements the SSP execution model. Other parameter server frameworks include MLNET [38] and Litz [6]. MLNET deploys a communication layer that uses tree-based overlays to implement distributed aggregation to only communicate the aggregated updates without the support for individual communication of worker-results. These implementations do not support custom consistency models required by asynchronous optimization methods nor the history of gradients. Finally, numerous distributed computing frameworks have been developed to support specific applications. For example DistBelief [10] and TensorFlow [11] support deep learning applications while fine-grained data processing systems such as RAY [12] and Flink [13] are designed for streaming problems. The frameworks can not be naturally extended to support mini-batch optimization methods that require coarse-grained computation models.

VIII. Conclusion

This work introduces the ASYNC framework that facilities the implementation of asynchrony and history in machine learning methods on cloud and distributed platforms. Along with bookkeeping structures, the modules in ASYNC facilitate the implementation of numerous consistency models and history. ASYNC is built on top of Spark to benefit from Spark's in-memory computation model and fault tolerant execution. We present the programming model and interface that comes with ASYNC and implement the synchronous and asynchronous variants of two well-known optimization methods as examples. These examples only scratch the surface of the types of algorithms that can be implemented in ASYNC. We hope that ASYNC helps machine learning practitioners with the implementation and investigation to the promise of asynchronous optimization methods.

References

[1] L. Bottou, "Stochastic gradient descent tricks," in *Neural networks: Tricks of the trade*. Springer, 2012, pp. 421–436.

[2] S. J. Reddi, A. Hefny, S. Sra, B. Poczos, and A. J. Smola, "On variance reduction in stochastic gradient descent and its asynchronous variants," in *Advances in Neural Information Processing Systems*, 2015, pp. 2647–2655.

[3] A. Hadoop, "Apache hadoop," *URL http://hadoop. apache. org*, 2011.

[4] 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 *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 2012, pp. 2–2.

[5] 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," *IEEE Transactions on Big Data*, vol. 1, no. 2, pp. 49–67, 2015.

[6] A. Qiao, A. Aghayev, W. Yu, H. Chen, Q. Ho, G. A. Gibson, and E. P. Xing, "Litz: Elastic framework for high-performance distributed machine learning," in *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, 2018, pp. 631–644.

[7] R. Jagerman and C. Eickhoff, "Web-scale topic models in spark: An asynchronous parameter server," *arXiv preprint arXiv:1605.07422*, 2016.

[8] J. Zhang, H. Tu, Y. Ren, J. Wan, L. Zhou, M. Li, and J. Wang, "An adaptive synchronous parallel strategy for distributed machine learning," *IEEE Access*, vol. 6, pp. 19 222–19 230, 2018.

[9] J. R. Albrecht, C. Tuttle, A. C. Snoeren, and A. Vahdat, "Loose synchronization for large-scale networked systems." in *USENIX Annual Technical Conference, General Track*, 2006, pp. 301–314.

[10] 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*, 2012, pp. 1223–1231.

[11] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, 2016, pp. 265–283.

[12] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan *et al.*, "Ray: A distributed framework for emerging {AI} applications," in *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, 2018, pp. 561–577.

[13] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink: Stream and batch processing in a single engine," *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, vol. 36, no. 4, 2015.

[14] S. Wang, J. Liagouris, R. Nishihara, P. Moritz, U. Misra, A. Tumanov, and I. Stoica, "Lineage stash: Fault tolerance off the critical path," in *Proceedings of Symposium on Operating Systems Principles, SOSP*, vol. 19, 2019.

[15] R. L. Graham, T. S. Woodall, and J. M. Squyres, "Open mpi: A flexible high performance mpi," in *International Conference on Parallel Processing and Applied Mathematics*. Springer, 2005, pp. 228–239.

[16] A. Defazio, F. Bach, and S. Lacoste-Julien, "Saga: A fast incremental gradient method with support for non-strongly convex composite objectives," in *Advances in Neural Information Processing Systems*, 2014, pp. 1646–1654.

[17] R. Leblond, F. Pedregosa, and S. Lacoste-Julien, "Asaga: asynchronous parallel saga," *arXiv preprint arXiv:1606.04809*, 2016.

[18] J. Chen, X. Pan, R. Monga, S. Bengio, and R. Jozefowicz, "Revisiting distributed synchronous sgd," *arXiv preprint arXiv:1604.00981*, 2016.

[19] J. Cipar, Q. Ho, J. K. Kim, S. Lee, G. R. Ganger, G. Gibson, K. Keeton, and E. Xing, "Solving the straggler problem with bounded staleness," in *Presented as part of the 14th Workshop on Hot Topics in Operating Systems*, 2013.

[20] 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," *The Journal of Machine Learning Research*, vol. 17, no. 1, pp. 1235–1241, 2016.

[21] C.-C. Chang and C.-J. Lin, "Libsvm: a library for support vector machines," *ACM transactions on intelligent systems and technology (TIST)*, vol. 2, no. 3, p. 27, 2011.

[22] J. Towns, T. Cockerill, M. Dahan, I. Foster, K. Gaither, A. Grimshaw, V. Hazlewood, S. Lathrop, D. Lifka, G. D. Peterson *et al.*, "Xsede: accelerating scientific discovery," *Computing in Science & Engineering*, vol. 16, no. 5, pp. 62–74, 2014.

[23] B. Recht, C. Re, S. Wright, and F. Niu, "Hogwild: A lock-free approach to parallelizing stochastic gradient descent," in *Advances in Neural Information Processing Systems*, 2011, pp. 693–701.

[24] R. Johnson and T. Zhang, "Accelerating stochastic gradient descent using predictive variance reduction," in *Advances in Neural Information Processing Systems*, 2013, pp. 315–323.

[25] C. Karakus, Y. Sun, S. Diggavi, and W. Yin, "Straggler mitigation in distributed optimization through data encoding," in *Advances in Neural Information Processing Systems*, 2017, pp. 5434–5442.

[26] I. S. Moreno, P. Garraghan, P. Townend, and J. Xu, "Analysis, modeling and simulation of workload patterns in a large-scale utility cloud," *IEEE Transactions on Cloud Computing*, vol. 2, no. 2, pp. 208–221, 2014.

[27] G. Ananthanarayanan, S. Kandula, A. G. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris, "Reining in the outliers in map-reduce clusters using mantri." in *Osdi*, vol. 10, no. 1, 2010, p. 24.

[28] P. Garraghan, X. Ouyang, R. Yang, D. McKee, and J. Xu, "Straggler root-cause and impact analysis for massive-scale virtualized cloud datacenters," *IEEE Transactions on Services Computing*, 2016.

[29] X. Ouyang, P. Garraghan, D. McKee, P. Townend, and J. Xu, "Straggler detection in parallel computing systems through dynamic threshold calculation," in *2016 IEEE 30th International Conference on Advanced Information Networking and Applications (AINA)*. IEEE, 2016, pp. 414–421.

[30] X. Lian, Y. Huang, Y. Li, and J. Liu, "Asynchronous parallel stochastic gradient for nonconvex optimization," in *Advances in Neural Information Processing Systems*, 2015, pp. 2737–2745.

[31] A. Agarwal and J. C. Duchi, "Distributed delayed stochastic optimization," in *Advances in Neural Information Processing Systems*, 2011, pp. 873–881.

[32] W. Zhang, S. Gupta, X. Lian, and J. Liu, "Staleness-aware async-sgd for distributed deep learning," *arXiv preprint arXiv:1511.05950*, 2015.

[33] L. Wang, B. Catterall, and R. Mortier, "Probabilistic synchronous parallel," *arXiv preprint arXiv:1709.07772*, 2017.

[34] Y. Ming, Y. Zhao, C. Wu, K. Li, and J. Yin, "Distributed and asynchronous stochastic gradient descent with variance reduction," *Neurocomputing*, vol. 281, pp. 27–36, 2018.

[35] A. Mahout, "Scalable machine-learning and data-mining library," *available at mahout. apache. org*, 2008.

[36] A. G. B. Saadon and H. M. Mokhtar, "iihadoop: an asynchronous distributed framework for incremental iterative computations," *Journal of Big Data*, vol. 4, no. 1, p. 24, 2017.

[37] J. E. Gonzalez, P. Bailis, M. I. Jordan, M. J. Franklin, J. M. Hellerstein, A. Ghodsi, and I. Stoica, "Asynchronous complex analytics in a distributed dataflow architecture," *arXiv preprint arXiv:1510.07092*, 2015.

[38] L. Mai, C. Hong, and P. Costa, "Optimizing network performance in distributed machine learning," in *7th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 15)*, 2015.