Distributed Frank-Wolfe under Pipelined Stale Synchronous Parallelism

Nam-Luc Tran, Thomas Peel, Sabri Skhiri
EURA NOVA
Email: {namluc.tran, thomas.peel, sabri.skhiri}
@euranova.eu

***Abstract*—Iterative-convergent algorithms represent an important family of applications in big data analytics. These are typically run on distributed processing frameworks deployed on a cluster of machines. On the other hand, we are witnessing the move towards data center operating systems (OS), where resources are unified by a resource manager and processing frameworks coexist with each other. In this context, different processing framework job tasks can be scheduled on the same machine and slow down a worker (straggler problem). Existing work has shown that an iteration model with relaxed consistency such as the Stale Synchronous Parallel (SSP) model, while still guaranteeing convergence, is able to cope with stragglers. In this paper we propose a model for the integration of the SSP model on a pipelined distributed processing framework. We then apply SSP on a distributed version of the Frank-Wolfe algorithm. We theoretically show its sparsity bounds and convergence under SSP. Finally, we experimentally show that the Frank-Wolfe algorithm applied on LASSO regression under SSP is able to converge faster than its BSP counterpart, especially under load conditions similar to those encountered in a data center OS.**

***Keywords*-stale synchronous parallel; distributed convex optimization; Frank-Wolfe; LASSO regression; parameter server; big data**

I. Introduction

Big data analytics have reached a certain level of maturity in industry and science [1]. Machine learning algorithms are one of the most important applications of big data distributed processing frameworks. These algorithms are different from traditional workloads by their iterative-convergent nature: they iterate until they reach a threshold value, such as in minimizing a prediction error. Distributed processing frameworks such as *Twister* [2], *Haloop* [3] or *ScalOps* [4] implement the Bulk Synchronous Parallel (BSP) paradigm [5] and are optimized for this type of iterative workload.

New approaches have tried to push forward those machine learning processing optimizations by implementing a pipelined architecture in which tuples of data are streamed through a set of operators. This is the case for *Dryad* and *Naiad* [6], but also for *Spark* [7] that pipelines mini-batches of resilient distributed datasets (RDD), or even more recently *Flink* [8]–[10]. Those pipelined architectures present interesting advantages such as increased performance and the ability to implement within the same framework both the batch and speed layer of the lambda architecture [11]. All those new approaches are however still based on the BSP paradigm.

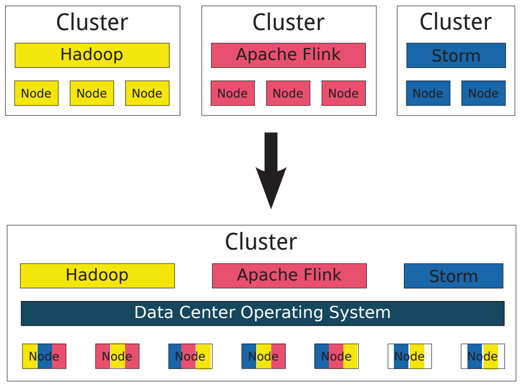

Figure 1. Within a data center operating system, all resources are unified by a resource manager and different frameworks compete with each other for the allocation of resources with regards to the tasks they execute.

Following the recent evolutions in data centers towards the unification of computing resources, BSP-based frameworks could lead to negative impact in performance for such algorithms. Indeed, the rise of resource managers such as *Mesos* [12], *Yarn* [13], *Omega* [14] and *Corona* [15] has defined the concept of *data center operating system* as illustrated in Figure 1. The comparison is often used to describe a modern data center made from a cluster of machines, seen individually as resources, on which task workloads are submitted in a transparent way. The resource manager automatically provisions and executes the tasks, and dynamically shrinks (deallocation) or increases (allocation) the resources dedicated to the tasks of a framework. The whole system is optimized in terms of CPU, memory, disk I/O and data locality. This brings the following advantages: higher rate of data center usage, optimized automation and abstraction of the resources.

Coming back to iterative-convergent tasks, this means that a BSP worker running on a machine is not isolated anymore from other frameworks in the cluster. As a result, another set of tasks can be scheduled on the same machine and slow down the worker. This kind of worker is identified as a *straggler* in [16] where it is shown that BSP programming dramatically suffers in the presence of stragglers. Instead, the authors of [16] propose the *Stale Synchronous Parallel*

(SSP) model of computation.

In the SSP paradigm, the synchronization barrier is relaxed in each superstep and faster workers continue to iterate on stale versions of the model, as shown in Figure 2. Convergence and correctness of the solution are still guaranteed for most of the iterative-convergent algorithms. There is however currently no model for the implementation of SSP on pipeline or stream architectures.

We focus on the distributed Frank-Wolfe algorithm as we think that it is an excellent candidate for asynchronous processing and especially within bounded staleness as is the case for SSP. The Frank-Wolfe algorithm aims at solving convex optimization problems over a convex set and has regained a lot of interest in the machine learning community especially in large-scale applications. In its basic version [17] the algorithm consists of a first phase where parallel computations are performed with regard to a selection criteria, followed by a synchronization phase where the best candidate from the previous phase is selected, in a pure BSP fashion. This work follows the distributed Frank-Wolfe algorithm investigated in a fully synchronous manner by [18] and already discussed in an asynchronous setting in [19].

Our contribution: In this paper, we propose (1) an SSP model based on the Apache Flink[1] pipeline architecture, (2) an asynchronous distributed Frank-Wolfe optimization algorithm adapted to our proposed SSP model and (3) an empirical evaluation of an implementation of the well-known LASSO regression algorithm [20] using our optimization scheme in the core of the algorithm. Flink's pipelining engine and streaming API make it a first choice for pipelined SSP. Our model however is generic and can be applied as such on pure stream processing architecture such as *Storm* or *SAMOA* [21].

We show a few theoretical results in a worst case scenario including a convergence analysis of the distributed Frank-Wolfe algorithm under SSP and an upper bound on the sparsity of the final solution. Those results are then evaluated in practice through an application of our algorithm to solve a synthetic LASSO regression problem i.e. a l_1 regularized sparse approximation problem. Our choice of the LASSO is motivated by its popularity in the feature selection domain making it a good candidate to highlight our method.

This work is a first step toward a more detailed theoretical analysis of our algorithm and an extension to the on-line convex learning setting.

Outline: Section two discusses our model for SSP on a pipelined architecture. Section three describes the distributed Franck-Wolfe algorithm in an asynchronous environment leveraging the SSP model. In this section, we also present our proposal for the bound on the sparsity of the solution as well as the theoretical proof of convergence of our algorithm under SSP. The application of this algorithm on LASSO regression is detailed in Section four. Finally, Section five discusses the experiments and the results.

[1]The Apache Flink project: http://flink.apache.org

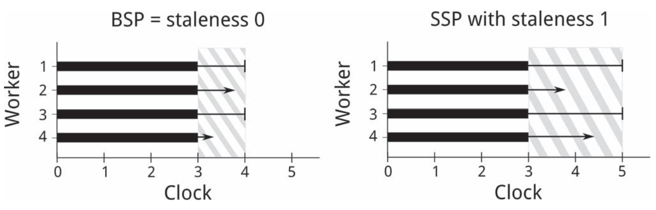

Figure 2. In Bulk Synchronous Parallelism (left), workers synchronize their updates to the model after each clock. Under Stale Synchronous Parallelism (right), workers access the updates of their co-workers in a best-effort mode within the bounds of staleness.

Notations: In the rest of the paper, we denote vectors by bold lower case letters $\boldsymbol{x}$ and matrices by bold upper case letters $\boldsymbol{A}$. $\|\boldsymbol{x}\|_0$ is the number of non-zero entries in the vector $\boldsymbol{x}$. $\nabla f(\boldsymbol{\alpha})$ denotes the gradient of a differentiable function f taken at $\boldsymbol{\alpha}$. We do not distinguish local worker threads and worker threads located on different hosts: we call them all "worker threads".

II. Stale Synchronous Parallel iterations in pipelined processing frameworks

Distributed processing is one solution to the challenge of treating bigger and bigger volumes of data. There are currently numerous frameworks that store and process data in a distributed and scalable way.

In a distributed processing cluster, many situations can lead to threads or processes acting slower than their peers for some time. These processes are called *stragglers* and have various causes, including the algorithm itself. Other reasons include hardware heterogeneity, skewed data distribution, and garbage collection in the case of high-level languages. Within data center operating systems (Figure 1), other causes for stragglers include concurrency between tasks of different frameworks.

The *Bulk Synchronous Parallel* model (BSP), one of the most currently used paradigms in distributed processing frameworks for iterative algorithms, has shown its limits in terms of scalability [22]. The frequent and explicit nature of the synchronization in BSP implies that each iteration proceeds at the pace of the slowest thread. This leads to poorer performance in the presence of stragglers. The authors of [23] have shown that in some cases the percentage of effective time spent on the computation represents 25 percent of total time spent in the iteration, the rest of the time is spent in network communication and synchronization.

A. Generalization of the iteration model

Within an iteration, a *clock* represents the amount of work a worker performs on its data partition. Each worker does not see the adjustments performed from the other workers on their partition during a clock. This leads to the notion of *staleness* which defines the number of clocks during which a worker does not see the adjustments from the other workers.

In *Stale Synchronous Parallel* processing (SSP), the staleness parameter can take arbitrary values and different workers can be at different values of clock, with the staleness parameter defining the maximum number of clocks the fastest worker may be ahead of the slowest. Staleness is a parameter to be tuned depending on the algorithm and the size of the cluster [16]. Under this generalization, BSP is the particular case where the staleness parameter equals zero.

As there are no explicit barriers forcing every worker to synchronize their adjustments, there is a need for a shared data structure that stores the current algorithm state and that the workers are able to update individually on each clock. This component is called the *parameter server*.

In SSP, each worker starts with an internal clock equal to zero. The workers repeat the following sequence: (1) perform computation using the shared model stored in the parameter server, (2) perform additive updates to the shared model in the parameter server, (3) advance its internal clock by 1. This sequence of operations results in a window bounded by the slowest and the fastest worker, and with a maximum length defined by the staleness value (Figure 2). The SSP condition [24] is such that within that window, each worker sees a noisy view of the system state, composed of, on one hand, the updates guaranteed until the lower bound of the window, and on the other side, the best-effort updates made within the window.

B. Convergence guarantees

To the best of our knowledge, the convergence proof for the SSP model has been demonstrated only for the stochastic gradient descent algorithm in [23] and [24]. Although the authors have used SSP for other algorithms such as latent Dirichlet allocation, there is no formal demonstration for variants such as coordinate descent algorithms. The distributed Frank-Wolfe algorithm belongs to the latter category. In this paper, we propose an SSP implementation for the distributed Frank-Wolfe algorithm as well as a theoretical proof of its convergence under SSP.

C. Integration model on pipelined processing frameworks

In a pipelined architecture, the distributed processing framework processes the data tuples one by one by streaming them through the operators that compose the dataflow graph of operations. From an architectural viewpoint, this can be seen as a stream processing environment on which the data tuples are streamed to the workers.

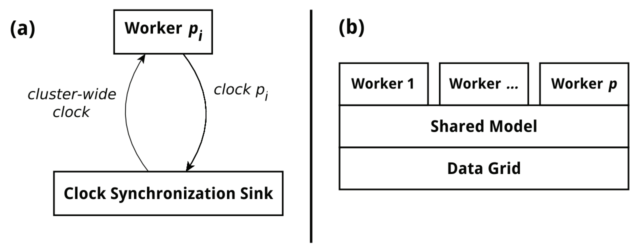

Figure 3. SSP iteration control model for a pipelined distributed processing framework. (a) Workers communicate their internal clock to the sink, which communicates the cluster-wide clock, using an event-driven architecture. (b) Overview of the parameter server, built on top of a data grid. Each worker stores a partition of the grid and automatically benefits from local reads and replication of the grid.

In this context we have designed SSP iterations to work in such pipelined frameworks. Instead of relying on a design centered around the parameter server which handles both the clock synchronization between workers through blocking calls and the storage of the parameter as in [24], we break down the architecture in two components: the first part handles the staleness synchronization among workers and the second provides a shared data structure between workers.

As a result, the most important changes come from (1) the event-driven nature of the parameter server and (2) the adaptation of the iteration control model to be SSP-compliant. In Flink, when a job is submitted, it is first compiled into a dataflow graph of tasks including the execution control structures, and then transformed into a physical execution plan that defines where each task will be executed. The tasks that handle iterations are controlled by special structures that we call the *iteration control model*.

In the current iteration control model, all the tasks spawned by an iterative job are connected to a clock synchronization sink in an event-driven fashion. The clock synchronization sink holds the value of the current cluster-wide clock, defined as the minimum clock value among the workers. Each time a worker finishes a clock within an iterative task, it sends its updated clock to the clock synchronization task. The clock synchronization task collects all the clock updates from the worker, and keeps track of all the internal clocks for each worker. Each time the minimum clock value changes, the clock synchronization sink broadcasts the value to all the workers as the new cluster-wide clock value. This process is formalized on Figure 1.

During the execution, a worker starts working on its next clock only if its own clock value is not greater than the cluster-wide clock plus the value of the staleness. This constraint guarantees that the fastest worker is never ahead of the slowest worker by a number of clocks greater than the value of the staleness, and thus verifies the SSP conditions [24]. Algorithm 3 illustrates the process of a

```
1: $C = (0,0,0\ldots)$ where $|C| = |V|$ and $V$ is the set of all workers
2: $clock = 0$
3: loop
4:     $c_i$ = received clock for worker $v_i$
5:     $C_i = c_i$
6:     if $min(C) > clock$ then
7:         $clock = min(C)$
8:         Broadcast $clock$ to all workers as the new cluster-wide clock
9:     end if
10: end loop
```

Algorithm 1. Process of the clock synchronization task involved in the SSP iteration control model.

worker under SSP according to our iteration control model.

The parameter server is built on top of a data grid distributed among all the workers in the cluster. This has the advantage that writes made by a worker to the grid are propagated and replicated to each of the workers in the background. Values written by a worker are locally stored on its partition of the grid. Writes to the model overwrite the version present on the parameter server and reads always return the latest update written to the parameter server. The latter has the effect of directly "pushing" the latest version of the shared model and allows slower workers to immediately benefit from the advance of faster workers, while the former enables the benefits of local accesses in a pipelined implementation of SSP.

Finally, one can notice that this implementation is fully applicable on pure stream processing systems such as Storm.

Related work: Peetuum is the implementation of a parameter server used for SSP iterations as described in [23]. The SSP iteration control model is implemented as blocking calls to the workers. In a more recent contribution, [24] describes *Eager SSP (ESSP)* in which updates to the model are pushed immediately to the workers. In our SSP integration model, writes to the parameter server overwrite the value already present. This leads to the same effect as ESSP in practice.

The concept of *microstep iteration* describing a setup where data is partitioned among workers and each iteration takes a single element from a working set and updates an element in a solution set has been defined in [25]. When the data flow between the solution set and the working set do not cross partition boundaries, this leads to fully asynchronous iterations. However, in distributed processing frameworks, this makes iteration controlling difficult as well as the checkpointing of intermediate results for recovery. These issues however can be addressed with a parameter server approach.

III. Definition of the distributed Frank-Wolfe algorithm under Stale Synchronous Parallelism

In this section, we remind the reader of the basic Frank-Wolfe algorithm and its distributed counterpart. Then, we present our variant of the distributed version under the SSP paradigm. Finally, we state some nice properties of our algorithm and compare it to related works.

A. The Frank-Wolfe algorithm

The Frank-Wolfe algorithm [17] is a simple yet powerful algorithm targeting the following optimization problem:

$$\min_{x \in \mathcal{D}} f(x), \tag{1}$$

where the f is a continuously differentiable convex function, and the domain $\mathcal{D}$ is a compact convex subset of $\mathbb{R}$. Algorithm 2 shows the basic Frank-Wolfe algorithm.

Despite its simplicity, the Frank-Wolfe algorithm shows nice convergence properties: let $\boldsymbol{\alpha}^*$ be the optimal solution of Equation (1), Theorem 1 states that the Frank-Wolfe algorithm finds an ϵ-approximate solution $\tilde{\boldsymbol{\alpha}}$ after $\mathcal{O}(1/\epsilon)$ iterations at most.

Theorem 1 (Jaggi, 2013 [26])**.** *Let C_f be the curvature of f. Algorithm 2 outputs a feasible point $\tilde{\boldsymbol{\alpha}}$ satisfying $f(\tilde{\boldsymbol{\alpha}}) - f(\boldsymbol{\alpha}^*) \leq \epsilon$ after at most $(1+\delta) \times 6.75\, C_f/\epsilon$ iterations where $\delta \geq 0$ is the accuracy to which the linear sub-problems are solved.*

The $\mathcal{O}(1/\epsilon)$ standard convergence rate of the Frank-Wolfe algorithm compares badly to optimal first order methods.[2] However, the Frank-Wolfe method iterates have good properties. For example, when the convex domain $\mathcal{D} = conv(\mathcal{S})$ is a convex hull of another set $\mathcal{S}$ then the iterates are expressed as a linear combination of elements from $\mathcal{S}$. In such a setting, if the elements (referred to as *atoms* thereafter) expose a structure (sparsity or low-rank) then the iterates can inherit this structure. Moreover, only one atom per iteration

```
1: Let $\boldsymbol{\alpha}^{(0)} \in \mathcal{D}$
2: for $k = 0, 1, 2, \ldots$ do
3:     $\boldsymbol{s}^{(k)} = \arg\min_{\boldsymbol{s} \in \mathcal{D}} \langle \boldsymbol{s}, \nabla f(\boldsymbol{\alpha}^{(k)}) \rangle$
4:     $\boldsymbol{\alpha}^{(k+1)} = (1-\gamma)\boldsymbol{\alpha}^{(k)} + \gamma \boldsymbol{s}^{(k)}$, where $\gamma = \frac{2}{k+2}$ or obtained via line-search
5: end for
6: stopping criterion: $\langle \boldsymbol{\alpha}^{(k)} - \boldsymbol{s}^{(k)}, \nabla f(\boldsymbol{\alpha}^{(k)}) \rangle \leq \epsilon$
```

Algorithm 2. The Frank-Wolfe algorithm.

[2]This can be improved with additional assumptions (see [27] and references therein).

can be added to the current solution. Thus, the iterates of the algorithm have a compact representation that can be leveraged to reduce the memory usage of the algorithm. We now recall the distributed version of the Frank-Wolfe algorithm.

B. Distributed Frank-Wolfe algorithm

In [28], the authors propose a distributed version of this algorithm to solve a slightly different problem, namely :

$$\min_{\boldsymbol{\alpha} \in \mathbb{R}^n} f(\boldsymbol{\alpha})\, s.t.\, \|\boldsymbol{\alpha}\|_1 \leq \beta, \tag{2}$$

where $f(\boldsymbol{\alpha}) = g(\boldsymbol{A}\boldsymbol{\alpha})$ for a matrix $\boldsymbol{A} = [\boldsymbol{a}_1, \ldots, \boldsymbol{a}_n] \in \mathbb{R}^{d \times n}$ with $d << n$. We name *atom* a column of the matrix $\boldsymbol{A}$. We assume with no loss of generality that the atoms are unit norm vectors: $\|\boldsymbol{a}_i\|_2 = 1$. We consider the column-wise partitioning of $\boldsymbol{A}$ across a set of N worker $V = \{v_i\}_{i=1}^N$. A node v_i is given a set of columns denoted by $\mathcal{A}_i$ such that $\bigcup_i \mathcal{A}_i = \boldsymbol{A}$ and $\mathcal{A}_i \bigcap \mathcal{A}_j = \emptyset\ \forall i \neq j$. In this setting, one wishes to find a weight vector $\boldsymbol{\alpha} \in \mathbb{R}^n$ under a sparsity constraint. This formulation is tightly related to optimizing over an atomic set as mentioned in [26] and fits well to the distributed setting: on the one hand the linear sub-problems can be computed in parallel and on the other hand the iterates being a sparse linear combination of atoms allow for an efficient communication scheme.

Each iteration of the algorithm proposed in [28] takes place in three steps:

Step 1 : each worker computes the best atom $\boldsymbol{s}_i$ locally and broadcasts it to all other workers,
Step 2 : each worker elects the best atom from all the atoms it has received,
Step 3 : each worker updates its local version of parameter $\boldsymbol{\alpha}$ using the atom selected during the previous step.

This algorithm is part of the BSP iterative algorithms paradigm and the authors demonstrated the convergence and correctness of their approach. In [28] the authors focus on decreasing the amount of communication and the waiting costs of their procedure and propose an extension to cases where workers are heterogeneous. Basically, they propose to distribute less atoms to slow workers and more to the fastest ones by running a clustering algorithm to group atoms around centroids that they propose to use as proxies on which to run the Frank-Wolfe algorithm.

However, they noticed that running their algorithm in a simulated asynchronous environment not only converges but also has a high acceleration potential. When stragglers randomly appear among workers, a common scenario in a data center OS, an unbalanced partitioning like the one proposed in [28] is not appropriate. Moreover, in such situation, obtaining a dynamic load-balancing scheduling policy can be costly. This scenario led us to explore an asynchronous variant we describe in the following subsection.

C. Distributed Frank-Wolfe under SSP

We propose an asynchronous version of the distributed Frank-Wolfe algorithm based on the SSP paradigm. More precisely, each worker can use a locally optimal atom in order to update its current possibly out-of-date, but with bounded staleness, version of the parameter vector $\boldsymbol{\alpha}$. Our claim is that avoiding the synchronized update step can help the algorithm be tolerant to the straggler problem without sacrificing the convergence rate. Algorithm 3 formalizes the use of the SSP paradigm to reach our goal. At each iteration, each worker requests the current value of the parameter stored in a parameter server. Then, it processes the sub-problem optimization step with respect to its local atom set. Finally, it updates the parameter and writes the new parameter value on the parameter server.

Implementation details: The atoms being in the support of the current solution are stored only once in a shared replicated cache and the coefficients are partitioned across workers in another cache. This allows for communication-efficient update propagation. The solution is updated in a greedy fashion. This can lead to losing improvements made by workers upstream when a late worker updates the parameter vector $\boldsymbol{\alpha}$. To prevent this kind of scenario, the function *updateParameter* checks the improvement between the current stored solution and the solution to be inserted. Hence, our algorithm is guaranteed to make improvements at each iteration. This leads to the following theorem that states the convergence of our algorithm.

Theorem 2 (Convergence of Algorithm 3)**.** *Let C_f be the curvature of f. Algorithm 3 outputs a feasible point $\tilde{\boldsymbol{\alpha}}$ satisfying $f(\tilde{\boldsymbol{\alpha}}) - f(\boldsymbol{\alpha}^*) \leq \epsilon$ after at most $2s \times (1+\delta) \times 6.75\, C_f/\epsilon$ iterations where $\delta \geq 0$ is the accuracy to which the linear sub-problems are solved.*

Sketch of proof: The intuition behind the proof is the following. Each worker can only improve the current solution. Moreover, because of the SSP paradigm, we are assured to see each atom at least once every $2s$ iterations. Thus, we make an update that is as good as Frank-Wolfe updates every $2s$ iterations at least. ■

This convergence rate is a worst case result that badly compares to the original distributed Frank-Wolfe algorithm. However, in practice, we show in Section V that our algorithm converges more quickly to the optimum. A more detailed analysis of this property is left for future work.

We shall now make two propositions related to properties of the distributed Frank-Wolfe algorithm under SSP. The first one is that our asynchronous setting does not increase too much the number of non-zero entries of each of the iterates that remain upper bounded.

Proposition 1 (Sparsity of the iterates)**.** *At iteration k, the*

```
1:  Let $\boldsymbol{\alpha}_i^{(0)} = \mathbf{0}$, $c_i = 0$ for all worker $v_i \in V$, $clock = 0$
    and $s$ the staleness parameter.
2:  $clock$ is updated onwards from the clock synchronization sink
3:  for all worker $v_i \in V$ in parallel do
4:    repeat
5:      if $c_i \leq clock + s$ then
6:        $\boldsymbol{\alpha}^{(c_i)} = getParameter()$
7:        $\boldsymbol{s}^{(c_i)} = \arg\min_{\boldsymbol{s}\in\mathcal{D}_i} \langle \boldsymbol{s}, \nabla f(\boldsymbol{\alpha}^{(c_i)})_i \rangle$
8:        $\boldsymbol{\alpha}^{(c_i+1)} = (1-\gamma)\boldsymbol{\alpha}^{(c_i)} + \gamma \boldsymbol{s}^{(c_i)}$, where $\gamma$ is
          obtained via line-search
9:        $updateParameter\left(i, c_i, \boldsymbol{\alpha}^{(c_i+1)}\right)$
10:       $c_i = c_i + 1$
11:       send $c_i$ to the clock synchronization sink
12:     else
13:       wait until $c_i \leq clock + s$
14:     end if
15:   until $\langle \boldsymbol{\alpha}^{(c_i)} - \boldsymbol{s}^{(c_i)}, \nabla f(\boldsymbol{\alpha}^{(c_i)}) \rangle \leq \epsilon$ for all nodes $v_i$
16: end for
```

Algorithm 3. Stale Synchronous Parallel distributed Frank-Wolfe Algorithm. The main contributions lie in the staleness bound (line 5) in which at each clock iteration the latest parameter is obtained (line 6). The local optimal value is updated (lines 7-8) and an update to the parameter server is sent (line 9). Finally, the clock is incremented and propagated (lines 10-11).

number of non-zero elements in the weight vector $\boldsymbol{\alpha}^k$ is at most :

$$\|\boldsymbol{\alpha}^k\|_0 \leq k \times N \tag{3}$$

Proof: It is easy to prove the proposition by induction on k. At the first iteration, $k = 1$ and since each worker can only add one atom per iteration to the current solution, hence adding only one non-zero coefficient, we have that $\|\boldsymbol{\alpha}^1\|_0 \leq N$. In the worst case scenario, workers sequentially update the parameter within each iteration and since $\mathcal{A}_i \bigcap \mathcal{A}_j = \emptyset \; \forall i \neq j$ there is at most N new non-zero entries in $\boldsymbol{\alpha}^{k+1}$ compared to $\boldsymbol{\alpha}^k$. Suppose that the assumption holds at iteration k and show that it holds for $k+1$:

$$\begin{aligned} \|\boldsymbol{\alpha}^{k+1}\|_0 &\leq \|\boldsymbol{\alpha}^k\|_0 + N \\ &\leq k \times N + N \\ &\leq (k+1) \times N, \end{aligned}$$

where the first line comes from the preceding observation and the second one is given by our induction hypothesis. This concludes the proof. ■

The second proposition deals with an interesting metric which is the communication cost incurred by our algorithm. It tells us that the communication cost is not worse than the one of the distributed Frank-Wolfe algorithm [28] in a star network setting. Actually, in practice, the cost is slightly lower in our implementation because we do not need to send over the network an atom that has already been chosen in a previous iteration. Indeed, we leverage the cache feature of the parameter server. Note that the algorithm of [28] can also benefit from this feature with a small change.

Proposition 2 (Communication cost)**.** *We assume that sending a number over the network has a constant cost. Then the total communication cost of our algorithm to obtain an ϵ approximate solution is at most*

$$\mathcal{O}\left(\delta \times (N^2 + Nd) \times 6.75\, C_f/\epsilon\right).$$

Sketch of proof: This result is a straightforward application of Theorem 1 jointly with Proposition (1). ■

Again, this is a worst case result assuming that none of the atoms is selected twice. In practice, we will see that atoms are selected multiple times and hence we can leverage the atom grid that caches the already selected atoms at a worker level avoiding unnecessary atom broadcasting once it is present in the cache.

Related work: In [28], the authors have already pointed out that their algorithm is robust to asynchronous updates. Under random communication drops to simulated asynchronism, they show that their algorithm empirically converges. In [19], the authors study an asynchronous version of the Frank-Wolfe algorithm to solve a convex optimization problem subject to block-separable constraints and they discuss a distributed variant of the algorithm. In a nutshell, they sample groups of atoms from the entire set and each of these is processed asynchronously by a worker lying in a worker pool. When the sub-samples set is empty, they update the parameter and repeat the procedure until a convergence criterion is met. They propose an extension in order to leverage a distributed environment. Each worker asynchronously samples one block of coordinates at a time and gives back its result to a parameter server. Every τ results received, the parameter server updates the parameter and broadcasts the new value the workers. This work is related to what we present in this paper but instead of letting workers randomly choose blocks of variables we leverage the data locality provided by our pipelined setting such that each block is assigned to a worker at the beginning of the process. In such a setting we are sure not to process the same block more than once at each iteration, thus reducing the number of redundant computations that can appear in [19]. Moreover, as our updates take place at the worker-side there is no computation overhead at the task manager. Finally, we propose a more detailed view on how atoms are stored and sent to workers.

IV. Application: LASSO regression

We choose the LASSO algorithm [20] to empirically validate the effectiveness of our approach. LASSO is a linear regression method for solving the following sparse approximation problem :

$$\min_{\boldsymbol{\alpha}\in\mathbb{R}^n} \tfrac{1}{2}\left\|\boldsymbol{y}-\boldsymbol{A}\boldsymbol{\alpha}\right\|_2^2 \; s.t. \; \|\boldsymbol{\alpha}\|_1 \leq \beta, \tag{4}$$

where we seek to approximate the target value y_i for the training point i by a sparse linear combination of its features a_{ij}, using the same small number of features for all data points.

A. Duality gap

The Frank-Wolfe algorithm exposes a nice certificate for the current iterate quality, the so-called duality-gap :

$$h(\boldsymbol{\alpha}) = \max_{\boldsymbol{\alpha}\in\mathcal{D}} \langle \boldsymbol{\alpha}-\boldsymbol{s}, \nabla f(\boldsymbol{\alpha})\rangle \geq f(\boldsymbol{\alpha}) - f(\boldsymbol{\alpha}^*).$$

Given that $\boldsymbol{s}$ is a minimizer of the linearized problem at point $\boldsymbol{\alpha}$, the duality gap is given by the by-product $\langle \boldsymbol{\alpha}-\boldsymbol{s}, \nabla f(\boldsymbol{\alpha})\rangle$. Moreover in the special case of LASSO, the duality-gap is fast to compute. This quantity, as shown below, only depends on information that is available at each worker level :

$$\begin{aligned}\langle \boldsymbol{\alpha}-\boldsymbol{s}, \nabla f(\boldsymbol{\alpha})\rangle &= -\left(\boldsymbol{\alpha}-\boldsymbol{s}\right)^{\mathsf{T}}\boldsymbol{A}^{\mathsf{T}}\left(\boldsymbol{y}-\boldsymbol{A}\boldsymbol{\alpha}\right)\\ &= \left(\boldsymbol{A}\left(\boldsymbol{s}-\boldsymbol{\alpha}\right)\right)^{\mathsf{T}}\left(\boldsymbol{y}-\boldsymbol{A}\boldsymbol{\alpha}\right)\\ &= \left(\boldsymbol{A}\boldsymbol{s}-\boldsymbol{A}\boldsymbol{\alpha}\right)^{\mathsf{T}}\boldsymbol{r}\end{aligned}$$

The residual is easily computable by each worker (or it can be broadcast along with the current sparse approximation) and $\boldsymbol{s}$ belongs to the set of available columns of this worker. Moreover, the matrix-vector products only involve sparse vectors : $\boldsymbol{A}\boldsymbol{s}$ is only the multiplication of a column of $\boldsymbol{A}$ by a scalar.

B. Line search

For LASSO, the line search problem can be obtained *analytically* with nearly no additional computation cost. Indeed, the optimal step-size is obtained by solving the following problem :

$$\gamma^* = \operatorname*{arg\,min}_{\gamma\in[0,1]} f\left(\boldsymbol{\alpha}^{(k)} + \gamma(\boldsymbol{s}^{(k)} - \boldsymbol{\alpha}^{(k)})\right), \tag{5}$$

which is equivalent to finding the minimum of a quadratic function. Setting the derivative with respect to γ to zero gives :

$$\begin{aligned}\frac{\partial f}{\partial \gamma} = 0 &\Leftrightarrow -\left(\boldsymbol{A}(\boldsymbol{s}-\boldsymbol{\alpha})\right)^{\mathsf{T}}\left(\boldsymbol{y}-\boldsymbol{A}(\boldsymbol{\alpha}+\gamma^*(\boldsymbol{s}-\boldsymbol{\alpha})\right) = 0\\ &\Leftrightarrow \gamma^* = \frac{\left(\boldsymbol{s}-\boldsymbol{\alpha}\right)^{\mathsf{T}}\boldsymbol{A}^{\mathsf{T}}\left(\boldsymbol{y}-\boldsymbol{A}\boldsymbol{\alpha}\right)}{\|\boldsymbol{A}(\boldsymbol{s}-\boldsymbol{\alpha})\|_2^2}\\ &\Leftrightarrow \gamma^* = \frac{\langle \boldsymbol{\alpha}-\boldsymbol{s}, \nabla f(\boldsymbol{\alpha})\rangle}{\|\boldsymbol{A}(\boldsymbol{s}-\boldsymbol{\alpha})\|_2^2}\end{aligned}$$

Thus

$$\gamma^* = \max\left(0, \min\left(1, \frac{\langle \boldsymbol{\alpha}-\boldsymbol{s}, \nabla f(\boldsymbol{\alpha})\rangle}{\|\boldsymbol{A}(\boldsymbol{s}-\boldsymbol{\alpha})\|_2^2}\right)\right) \tag{6}$$

The optimal step-size is obtained through the duality-gap and $\|\boldsymbol{A}(\boldsymbol{s}-\boldsymbol{\alpha})\|_2^2$ can be evaluated efficiently (because $\boldsymbol{A}(\boldsymbol{s}-\boldsymbol{\alpha})$ is available as it is involved in the duality-gap computation).

V. Experiments

We have implemented the SSP model on an existing pipelined distributed processing framework. The pipelined model we have described does not rely on any particular framework feature and is suited for most existing frameworks supporting a pipelined setup. We have chosen the Apache Flink project as the basis for our implementation. At its core Flink defines each data processing job as a dataflow graph with pipelined operators. The graph describes the transformations of the data set during the process.

The platform supports the execution of iterative algorithms with a convergence criterion. Like many other frameworks, it supports the bulk synchronous parallel model for iterative algorithms among distributed workers. We have extended the existing structures for the control of the iterations to implement SSP and we have integrated our own parameter server built on top of a data grid.

We have run experiments on sparse random matrices of dimensions 1.000×10.000 with a sparsity ratio of 0.001 and a random vector $\boldsymbol{\alpha}^*$ such that $\|\boldsymbol{\alpha}^*\|_0 = 100$. For each staleness bounds, we have repeated the experiment 5 times. The cluster is composed of five nodes on the same switch, each equipped with a 2Ghz single-core equivalent and 3Gb of memory. We use a plain BSP implementation of the Frank-Wolfe algorithm in Flink with no use of the parameter server as the baseline for our comparisons.

A. Convergence and quality of the solution

Figure 4 shows the evolution of the residual over the iterations. We observe that, despite the staleness introduced by SSP, the model converges well. Under the values for the staleness that we have experimented with, the convergence with regard to the residual is even better than the baseline BSP counterpart with no staleness. This can be explained by the fact that in SSP, a worker at a certain clock can read a model that has already been updated by another worker at a more advanced clock. While being several clocks behind, a worker can already work with a more converged model updated by other workers.

B. Performance under load

In order to test our solution in a situation where different frameworks share the same cluster in an environment similar to Yarn or Mesos, we have written a tool that generates load on the hosts. The generated load is independent from the processing framework and is not predictable from the scheduling point of view of the processing framework.

This behaviour simulates a concurrent task deployed on the cluster by another framework involving computing or garbage collection. This type of temporary load has an impact on the performance of a worker but due to its short duration, computing a dynamic load-balancing policy and relocating the task from a worker to another host is costly and inefficient.

Figure 4 shows the convergence of the residual under generated load. Our tool selectively runs a compute-intensive process on a random worker for 12 seconds continuously, one worker at a time. The results show that even under load our solution is able to perform better than the baseline BSP implementation, with a speedup of up to two times faster.

C. Sparsity of the final model

Figure 5 shows the distribution of the α_i coefficients at convergence after 250 iterations for our SSP implementation and the baseline BSP implementation. The results show that the final model obtained with our SSP implementation is nearly two times less sparse than the model obtained with BSP, despite being asynchronous with a staleness parameter of 10 and having 5 workers in parallel. This is in line with the bounds stated in Proposition 1.

These results however show that the final model has a high proportion of very small α_i coefficients. This opens the way to pruning methods that will clean up the model and render it more sparse. It is possible to prune the model either dynamically, in which case the intermediate models will then be cleaner, either at the end of the job, which allows for a better control of the error. To do so, we think that incorporating the use of away-steps [29] may be a good candidate to remove those small coefficients.

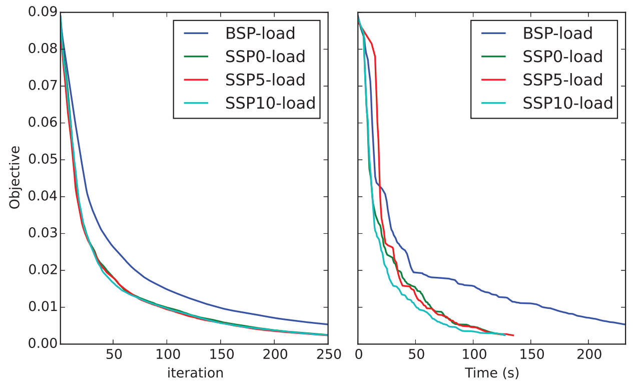

Figure 4. Evolution of the residual of the Frank-Wolfe algorithm under various values of SSP staleness and in BSP, in function of the iterations (left) and over time (right). A load is generated on a random node in the cluster at any moment during twelve seconds for the duration the experiment.

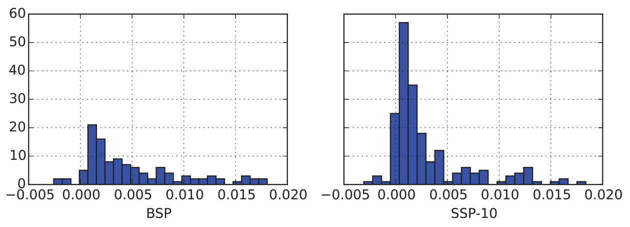

Figure 5. Histogram of the distribution of the weights in the solution obtained after 250 iterations. Left: BSP, right: SSP with staleness 10.

VI. Conclusion and future direction

In this paper we have presented a model for SSP iterations suited for distributed and pipelined processing frameworks. We have proposed an implementation of the Frank-Wolfe algorithm under SSP to assess our SSP framework. We have theoretically shown the upper bounds of the distributed and SSP variant of the algorithm, as well as its convergence. We show with experiments that our solution is able to converge faster than its BSP counterpart, even under a randomly loaded environment. This result is particularly valuable in the light of clusters managed by the recent resource managers, in which several distributed processing frameworks compete for resources over their respective tasks.

Our future line of research includes the study of the more general context of optimizing over atomic sets, the study of the sparsity of the iterates (away steps might be useful) and the comparison of our solution with other asynchronous approaches like [19], [30].

Acknowledgment

The authors would like to thank the Apache Flink team for their support during the implementation.

References

[1] Applications powered by hadoop. [Online]. Available: http://wiki.apache.org/hadoop/PoweredBy

[2] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox, "Twister: A runtime for iterative mapreduce," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, ser. HPDC '10. New York, NY, USA: ACM, 2010, pp. 810–818.

[3] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst, "Haloop: Efficient iterative data processing on large clusters," *Proc. VLDB Endow.*, vol. 3, no. 1-2, pp. 285–296, Sep. 2010.

[4] M. Weimer, T. Condie, and R. Ramakrishnan, "Machine learning in scalops, a higher order cloud computing language," in *NIPS 2011 Workshop on parallel and large-scale machine learning (BigLearn)*, December 2011.

[5] L. G. Valiant, "A bridging model for parallel computation," *Commun. ACM*, vol. 33, no. 8, pp. 103–111, Aug. 1990.

[6] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi, "Naiad: A timely dataflow system," in *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*. ACM, November 2013.

[7] 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*, ser. NSDI'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 2–2.

[8] S. Dudoladov, C. Xu, S. Schelter, A. Katsifodimos, S. Ewen, K. Tzoumas, and V. Markl, "Optimistic recovery for iterative dataflows in action," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, 2015, pp. 1439–1443.

[9] S. Ewen, S. Schelter, K. Tzoumas, D. Warneke, and V. Markl, "Iterative parallel data processing with stratosphere: An inside look," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '13. New York, NY, USA: ACM, 2013, pp. 1053–1056.

[10] Flink iteration and delta-iteration. [Online]. Available: ci.apache.org/projects/flink/flink-docs-release-0.6/iterations.html

[11] N. Marz. Big data lambda architecture. [Online]. Available: http://www.databasetube.com/database/big-data-lambda-architecture/

[12] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center," in *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'11. Berkeley, CA, USA: USENIX Association, 2011, pp. 295–308.

[13] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler, "Apache hadoop yarn: Yet another resource negotiator," in *Proceedings of the 4th Annual Symposium on Cloud Computing*, ser. SOCC '13. New York, NY, USA: ACM, 2013, pp. 5:1–5:16.

[14] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes, "Omega: Flexible, scalable schedulers for large compute clusters," in *Proceedings of the 8th ACM European Conference on Computer Systems*, ser. EuroSys '13. New York, NY, USA: ACM, 2013, pp. 351–364.

[15] (2012) Under the hood: Scheduling mapreduce jobs more efficiently with corona. [Online]. Available: http://on.fb.me/TxUsYN

[16] H. Cui, J. Cipar, Q. Ho, J. K. Kim, S. Lee, A. Kumar, J. Wei, W. Dai, G. R. Ganger, P. B. Gibbons, G. A. Gibson, and E. P. Xing, "Exploiting bounded staleness to speed up big data analytics," in *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, ser. USENIX ATC'14. Berkeley, CA, USA: USENIX Association, 2014, pp. 37–48.

[17] M. Frank and P. Wolfe, "An algorithm for quadratic programming," *Naval research logistics quarterly*, vol. 3, no. 1-2, pp. 95–110, 1956.

[18] A. Bellet, Y. Liang, A. B. Garakani, M. Balcan, and F. Sha, "Distributed frank-wolfe algorithm: A unified framework for communication-efficient sparse learning," *CoRR*, vol. abs/1404.2644, 2014.

[19] Y.-x. Wang, V. Sadhanala, W. Dai, W. Neiswanger, S. Sra, and E. P. Xing, "Asynchronous Parallel Block-Coordinate Frank-Wolfe," 2014.

[20] R. Tibshirani, "Regression Shrinkage and Selection via the Lasso," *Journal of the Royal Statistical Society*, vol. 58, no. 1, pp. 267–288, 1996.

[21] G. De Francisci Morales and A. Bifet, "Samoa: Scalable advanced massive online analysis," *J. Mach. Learn. Res.*, vol. 16, no. 1, pp. 149–153, Jan. 2015.

[22] 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 *Proceedings of the 14th USENIX Conference on Hot Topics in Operating Systems*, ser. HotOS'13. Berkeley, CA, USA: USENIX Association, 2013, pp. 22–22.

[23] Q. Ho, J. Cipar, H. Cui, and S. Lee, "More Effective distributed ML via a Stale synchronous parallel parameter server," *Nips*, no. July, pp. 1–9, 2013.

[24] W. Dai, A. Kumar, J. Wei, Q. Ho, G. Gibson, and E. P. Xing, "Analysis of High-Performance Distributed ML at Scale through Parameter Server Consistency Models," 2014.

[25] S. Ewen, K. Tzoumas, M. Kaufmann, and V. Markl, "Spinning fast iterative data flows," *Proc. VLDB Endow.*, vol. 5, no. 11, pp. 1268–1279, Jul. 2012.

[26] M. Jaggi, "Revisiting Frank-Wolfe: Projection-Free Sparse Convex Optimization," *Proceedings of the 30th International Conference on Machine Learning*, vol. 28, pp. 427–435, 2013.

[27] D. Garber and E. Hazan, "Faster rates for the frank-wolfe method over strongly-convex sets," in *The 32nd International Conference on Machine Learning*, 2015.

[28] A. Bellet, Y. Liang, A. B. Garakani, F. Sha, and M.-F. Balcan, "A Distributed Frank-Wolfe Algorithm for Communication-Efficient Sparse Learning," 2015.

[29] J. Guélat and P. Marcotte, "Some comments of wolfe's 'away step'," *Math. Program.*, vol. 35, no. 1, pp. 110–119, May 1986. [Online]. Available: http://dx.doi.org/10.1007/BF01589445

[30] J. Liu, S. Wright, C. Ré, and V. Bittorf, "An asynchronous parallel stochastic coordinate descent algorithm," 2014.