Under consideration for publication in Knowledge and Information Systems

Distributing Frank-Wolfe via Map-Reduce

Armin Moharrer and Stratis Ioannidis

Electrical and Computer Engineering Department, Northeastern University, Boston MA, USA

Abstract. Large-scale optimization problems abound in data mining and machine learning applications, and the computational challenges they pose are often addressed through parallelization. We identify structural properties under which a convex optimization problem can be massively parallelized via map-reduce operations using the Frank-Wolfe (FW) algorithm. The class of problems that can be tackled this way is quite broad and includes experimental design, AdaBoost, and projection to a convex hull. Implementing FW via map-reduce eases parallelization and deployment via commercial distributed computing frameworks. We demonstrate this by implementing FW over Spark, an engine for parallel data processing, and establish that parallelization through map-reduce yields significant performance improvements: we solve problems with 20 million variables using 350 cores in 79 minutes; the same operation takes 48 hours when executed serially.

Keywords: Frank-Wolfe, Distributed Algorithms, Convex Optimization and Spark

1. Introduction

Map-reduce (Dean & Ghemawat 2008, Bialecki et al. 2005) is a distributed framework used to massively parallelize computationally intensive tasks. It enjoys wide deployment in commercial cloud services such as Amazon Web Services, Microsoft Azure, and Google Cloud, and is extensively used to parallelize a broad array of data-intensive algorithms (Yang et al. 2007, Kumar et al. 2015, Bahmani et al. 2012, Suri & Vassilvitskii 2011, Chu et al. 2006). Expressing algorithms in map-reduce also allows fast deployment at a massive scale: any algorithm expressed in map-reduce operations can be quickly implemented and distributed on a commercial cluster via existing programming frameworks (Dean & Ghemawat 2008, Bialecki et al. 2005, Zaharia et al. 2010).

Received 20 Dec 2017
Revised 28 Sep 2018
Accepted 29 Oct 2018

In this paper, we focus on solving, via map-reduce, optimization problems of the form:

$$\min_{\theta \in \mathcal{D}_0} F(\theta), \tag{1}$$

where $F : \mathbb{R}^N \to \mathbb{R}$ is a convex, differentiable function, and

$$\mathcal{D}_0 \equiv \left\{ \theta \in \mathbb{R}_+^N : \sum_{i=1}^{N} \theta_i = 1 \right\} \tag{2}$$

is the N-dimensional simplex. Several important problems, including experimental design, training SVMs, Adaboost, and projection to a convex hull indeed take this form (Clarkson 2010, Bellet et al. 2015, Boyd & Vandenberghe 2004). We are particularly interested in cases where (a) $N \gg 1$, i.e., the problem is high-dimensional, and, (b) F *cannot* be written as the sum of differentiable convex functions. We note that, as described in Sec. 2, this is precisely the regime in which (1) is hard to parallelize via, e.g., stochastic gradient descent.

It is well known that (1) admits an efficient implementation through the Frank-Wolfe (FW) algorithm, also known as the conditional gradient algorithm (Frank & Wolfe 1956). Indeed, as we discuss in Sec. 3.2, FW assumes a very simple, elegant form under simplex constraints, and has important computational advantages (Clarkson 2010, Jaggi 2013, Bellet et al. 2015). Our main contribution is to identify and formalize a set of conditions under which solving Problem (1) through FW *admits a massively parallel implementation via map-reduce*. In particular:

- We identify two properties of the objective F under which FW can be parallelized through map-reduce operations.
- We show that several important optimization problems, including experimental design, Adaboost, and projection to a convex hull satisfy the aforementioned properties.
- We implement our distributed FW algorithm on Spark (Zaharia et al. 2010), an engine for large-scale distributed data processing. Our implementation is generic: a developer using our code needs to only implement a few problem-specific computational primitives; our code handles execution over a cluster.
- We extensively evaluate our Spark implementation over large synthetic and real-life datasets, illustrating the speedup and scalability properties of our algorithm. For example, using 350 compute cores, we can solve problems of 20 million variables in 79 minutes, an operation that would take 48 hours when executed serially.
- We introduce two stochastic variants of distributed FW, in which we only compute a subsample of the elements of the gradient. We implement these algorithms on Spark and compare their performance with distributed FW.

The remainder of this paper is organized as follows. We briefly review related work in Sec. 2, and introduce FW and the map-reduce framework in Sec. 3. In Sec. 4, we state the properties under which FW admits a parallel implementation via map-reduce, and describe the resulting algorithm. Examples of problems that satisfy these properties are given in Sec. 5. We extend possible applications of our algorithm on constraint sets beyond the simplex in Sec. 6. Finally, in Sec. 7 and 8 we describe our implementation and the results of our experiments over a Spark cluster.

2. Related Work

Frank-Wolfe (FW) (Frank & Wolfe 1956) has attracted interest recently due to its numerous computational advantages (Dudik et al. 2012, Hazan & Kale 2012, Ying & Li 2012, Joulin et al. 2014, Clarkson 2010, Jaggi 2013). It maintains feasibility throughout execution while being projection-free, and minimizes a linear objective in each step; the latter yields sparse solutions for several interesting constraint sets, which often accelerates computation (Clarkson 2010, Bellet et al. 2015, Jaggi 2013).

Frank & Wolfe (1956) showed a convergence rate of $O(\frac{1}{\epsilon})$ for smooth objectives. When the optimal solution lies at the boundary of the constraint set, FW converges slowly, i.e., the $O(\frac{1}{\epsilon})$ convergence rate is tight (Dunn 1979, Canon & Cullum 1968, Frank & Wolfe 1956, Jaggi 2013). This is because the iterations of the classic FW zig-zag between the vertices defining the face that contain the optimal solution. To avoid this zig-zagging phenomenon, Wolfe (1970) proposed a variant using 'away-points'; the basic idea is to move away from a 'bad' direction. Guélat & Marcotte (1986) analyzed this further, and showed a linear convergence rate on polytope constraint sets. Several recently proposed FW variants improve the previous results for Away-steps Frank-Wolfe and attain linear convergence under weaker conditions (Beck & Shtern 2015, Garber & Hazan 2016, Lacoste-Julien & Jaggi 2015, Harchaoui et al. 2012, Garber & Meshi 2016). The problems we consider do not satisfy these conditions, and these FW variants are not readily parallelizable; we thus focus on classic FW in this paper.

A different line of work has tried to improve FW for problems with block-separable constraints. Lacoste-Julien et al. (2013), proposed a random single-block FW algorithm, in which only a single block of variables is updated. At the expense of computing the duality gap, the convergence result was improved (Osokin et al. 2016). Wang et al. (2016) devised the idea of updating multiple blocks of variables in parallel, as randomized multiple-block FW, and Zhang et al. (2017) further elaborated on this. The problems that we consider in this work do not have a block-separable constraint, so we concentrate on classic FW.

Stochastic Gradient Descent (SGD) (Recht et al. 2011, Zinkevich et al. 2010, Li et al. 2013, Abadi et al. 2016, Chu et al. 2006) parallelizes optimization problems in which the objective is the sum of differentiable functions. Many important problems, including regression and classification, fall into this category, and SGD has been tremendously successful at tackling them (Recht et al. 2011, Zinkevich et al. 2010, Li et al. 2013, Abadi et al. 2016). SGD computes the contribution of different terms to the gradient in parallel, and adapts the present solution in a centralized fashion, often asynchronously. Stochastic Dual Coordinate Ascent (SDCA) (Yang 2013) also solves problems with separable objectives by parallelizing their dual.

The Alternating Directions Method of Multipliers (ADMM) (Boyd et al. 2011) applies to both separable and non-separable objectives, including LASSO (c.f. Sec. 6). In general, the above methods do not readily generalize to the remaining optimization problems we study here. Moreover, their message complexity increases with the number of variables; indeed, parallel SGD and ADMM over millions of variables assume that each term depends only on a few coordinates (Recht et al. 2011, Li et al. 2013, Boyd et al. 2011). We do not assume sum objectives or any sparsity conditions here.

More recently, and more relevant to our work, Bellet et al. (2015) propose a distributed version of FW for objectives of the form $F(\theta) = g(A\theta)$, for some $A \in$

θ	N-dimensional variable
F	Convex and differentiable function
∇F	Gradient of F
$\mathcal{D}$	Convex and compact subset of $\mathbb{R}^N$
$\mathcal{D}_0$	The N-dimensional simplex set
γ^k	Step-size at k-th iterations
X	$N \times d$ matrix, which shows the dataset ($N \gg d$).
p	d-dimensional real vector, projected on convex hull of some points in ConvexApproximation
r	d-dimensional binary vector, shows the ground truth labels in AdaBoost
α	Tunable parameter in AdaBoost
t_ϵ	The minimum time for an algorithm to obtain a solution within ϵ-neighborhood of the optimum solution (23).

Table 1. Notation Summary: The table briefly describes the notations used through the paper.

$\mathbb{R}^{d\times N}$, where $d \ll N$. Several examples fall in this class, including two we study here (convex approximation and Adaboost); intuitively, $A\theta$ serves as the common information in our framework (c.f. Sec. 4). The authors characterize the message and parallel complexity when A is partitioned across multiple processors under broadcast operations. Moreover, Tran et al. (2015) elaborated on their algorithm, and proposed an asynchronous version of the distributed Frank-Wolfe algorithm proposed by Bellet et al. (2015). It is based on their *Stale Synchronous Parallel* (SSP) model (Tran et al. 2015). They showed that the SSP based algorithm runs faster than the one based on a *Bulk Synchronous Parallel* (BSP) model, which is commonly used in distributed processing frameworks. We (a) consider a broader class of problems, that do not abide by the structure presumed by Bellet et al or Tran et al.. (e.g., the two experimental design problems presented in Sec. 5), and (b) establish properties under which FW can be explicitly parallelized through map-reduce rather than the message passing environment proposed by Bellet et al. This allows us to leverage commercial map-reduce frameworks to readily implement and deploy parallel FW on a cluster.

Stochastic variants of FW have been proposed recently (Hazan & Kale 2012, Lan & Zhou 2016, Hazan & Luo 2016, Reddi et al. 2016), using unbiased estimates of the gradient at each step. Hazan & Luo (2016) improve upon earlier convergence rates (Hazan & Kale 2012, Lan & Zhou 2016) when the objective is smooth, strongly convex, or Lipschitz. Reddi et al. (2016) extend these results to non-convex functions for which FW converges to a stationary point. We implement two stochastic FW variants based on gradient subsampling, and compare the relative performance of subsampling to increasing parallelism in Sec. 8.

3. Technical Preliminary

3.1. Frank-Wolfe Algorithm

The FW algorithm (Frank & Wolfe 1956), summarized in Alg. 1, solves problems of the form:

$$\text{Minimize} \quad F(\theta) \tag{3a}$$

$$\text{subj. to:} \quad \theta \in \mathcal{D}, \tag{3b}$$

where $F : \mathbb{R}^N \to \mathbb{R}$ is a convex function and $\mathcal{D}$ is a convex compact subset of $\mathbb{R}^N$. The algorithm selects an initial feasible point $\theta^0 \in \mathcal{D}$ and proceeds as follows:

$$s^k = \arg\min_{s \in \mathcal{D}} \; s^\top \cdot \nabla F(\theta^k), \tag{4a}$$

$$\theta^{k+1} = (1 - \gamma^k)\theta^k + \gamma^k s^k, \tag{4b}$$

for $k \in \mathbb{N}$, where $\gamma^k \in [0, 1]$ is the step size. At each iteration $k \in \mathbb{N}$, FW finds a feasible point s^k minimizing the inner product with the current gradient, and interpolates between this point and the present solution. Note that $\theta^{k+1} \in \mathcal{D}$, as a convex combination of $\theta^k, s^k \in \mathcal{D}$; therefore, the algorithm maintains feasibility throughout its execution. Steps (4a),(4b) are repeated until a convergence criterion is met; we describe how to set this criterion and the step size γ^k below.

Convergence criterion. Convergence is typically determined in terms of the *duality gap* (Jaggi 2013). The duality gap at feasible point $\theta^k \in \mathcal{D}$ in iteration $k \in \mathbb{N}$ is:

$$g(\theta^k) \equiv \max_{s \in \mathcal{D}} (\theta^k - s)^\top \nabla F(\theta^k) \overset{(4a)}{=} (\theta^k - s^k)^T \nabla F(\theta^k), \tag{5}$$

The convexity of F implies that $F(\theta^k) - F(\theta^*) \leq g(\theta^k)$ for any optimal solution $\theta^* \in \arg\min_{\theta \in \mathcal{D}} F(\theta)$ (Jaggi 2013). In other words, $g(\theta)$ is an upper bound on the objective value error at step k. The algorithm, therefore, terminates once the duality gap is smaller than some $\epsilon > 0$.

Step Size. The step size can be diminishing, e.g., $\gamma^k = \frac{2}{k+2}$, or set through *line minimization*, i.e.:

$$\gamma^k = \arg\min_{\gamma \in [0,1]} \; F\big((1 - \gamma)\theta^k + \gamma s^k\big). \tag{6}$$

Convergence to an optimal solution is guaranteed in both cases for problems in which the objective has a bounded curvature (Frank & Wolfe 1956, Jaggi 2013). In this case, both of the above step sizes imply that the k-th iteration of the Frank-Wolfe algorithm satisfies $F(\theta^k) - F(\theta^*) \leq O(\frac{1}{k})$ (Jaggi 2013). For arbitrary convex objectives with unbounded curvature, FW still converges if the step size is set by the line minimization rule (Bertsekas 1999).

3.2. Frank-Wolfe Over the Simplex

We focus on FW for the special case where the feasible set $\mathcal{D}$ is the simplex D_0, given by (2). As described in Section 5, this set of constraints arises in many problems, including training SVMs, convex approximation, Adaboost, and experimental design (see also (Clarkson 2010)). Under this set of constraints, the linear optimization problem in (4a) has a simple solution: it reduces to finding the minimum element of the gradient $\nabla F(\theta^k)$. Formally, for $[N] \equiv \{1, 2, \ldots, N\}$, and $\{e_i\}_{i \in [N]}$ the standard basis of $\mathbb{R}^N$, (4a) reduces to:

$$s^k = e_{i^*}, \text{ where } \; i^* \in \arg\min_{i \in [N]} \frac{\partial F(\theta^k)}{\partial \theta_i}. \tag{7}$$

Note that s^k is a vector in the standard basis of $\mathbb{R}^N$, for all $k \in \mathbb{N}$: as such, it is extremely sparse, having only one non-zero element. The sparsity of s^k plays a role in producing our efficient, distributed implementation, as discussed below.

Algorithm 1 FRANK-WOLFE

Pick $\theta^0 \in \mathcal{D}$
$k := 0$
repeat
  $s^k := \arg\min_{s \in \mathcal{D}} s^\top \cdot \nabla F(\theta^k)$
  $\text{gap} := (\theta^k - s^k)^\top \nabla F(\theta^k)$
  $\theta^{k+1} := (1-\gamma^k)\theta^k + \gamma^k s^k$
  $k := k+1$
until $\text{gap} < \epsilon$

3.3. Map-Reduce Framework

Consider a data structure $D \in \mathcal{X}^N$ comprising N elements $d_i \in \mathcal{X}$, $i \in [N]$, for some domain $\mathcal{X}$. A `map` operation over D applies a function to every element of the data structure. That is, given $f : \mathcal{X} \to \mathcal{X}'$, the operation $D' = D.\texttt{map}(f)$ creates a data structure D' in which every element d_i, $i \in [N]$, is replaced with $f(d_i)$. A `reduce` operation performs an aggregation over the data structure, e.g., computing the sum of the data structure's elements. Formally, let $\oplus$ be a binary operator $\oplus : \mathcal{X} \times \mathcal{X} \to \mathcal{X}$ that is *commutative* and *associative*, i.e.,

$$x \oplus y = y \oplus x, \quad \text{and} \quad ((x \oplus y) \oplus z) = (x \oplus (y \oplus z)).$$

Then, $D.\texttt{reduce}(\oplus)$ iteratively applies the binary operator $\oplus$ on D, returning $\bigoplus_{i \in [N]} d_i = d_1 \oplus \ldots \oplus d_N$. Examples of commutative, associative operators $\oplus$ include addition ($+$), the min and max operators, binary AND, OR, and XOR, etc.

Both `map` and `reduce` operations are "embarrassingly parallel". Presuming that the data structure D is distributed over P processors, a `map` can be executed without any communication among processors, other than the one required to broadcast the code that executes f. Such broadcasts require only $\log P$ rounds and the transmission of $P-1$ messages, when the P processors are connected in a hypercube network; the same is true for `reduce` operations (Leighton 2014). There exist several computational frameworks, including Hadoop (Bialecki et al. 2005) and Spark (Zaharia et al. 2010), that readily implement and parallelize map-reduce operations. Hence, expressing an algorithm like FW in terms of `map` and `reduce` operations allows us to (a) parallelize the algorithm in a straightforward manner, and (b) leverage these existing frameworks to quickly implement and deploy FW at scale.

4. Frank-Wolfe via Map-Reduce

4.1. Gradient Computation through Common Information

In this section, we identify two properties of function F under which FW over the simplex $\mathcal{D}_0$ admits a distributed implementation through map-reduce. Intuitively, our approach exploits an additional structure exhibited by several important practical problems: the objective function F often depends on the variables θ as well as a *dataset*, given as input to the problem. We represent this dataset through a matrix $X = [x_i]_{i \in [N]} \in \mathbb{R}^{N \times d}$ whose rows are vectors $x_i \in \mathbb{R}^d, i \in [N]$.

The dataset can be large, as $N \gg 1$; as such, X may be horizontally (i.e., row-wise) partitioned over multiple processors. Note here that the dataset size (N) equals the number of variables in F.

We assume that the dependence of F to the dataset X is governed by two properties. The first property asserts that the partial derivative $\frac{\partial F}{\partial \theta_i}$ for any $i \in [N]$ depends on (a) the variable θ_i, (b) a datapoint x_i in the dataset, as well as (c) some *common information* h. This common information, not depending on i, fully abstracts any additional effect that θ and X may have on partial derivative $\frac{\partial F}{\partial \theta_i}$. Our second property asserts that this common information is *easy to update*: as variables θ^k are adapted according to the FW algorithm (4), the corresponding common information h can be re-computed efficiently, through a computation that does not depend on N. More formally, we assume that the following two properties hold:

Property 1. There exists a matrix $X = [x_i]_{i\in[N]} \in \mathbb{R}^{N\times d}$, whose rows are vectors $x_i \in \mathbb{R}^d$, $i \in [N]$, such that for all $i \in [N]$:

$$\frac{\partial F(\theta)}{\partial \theta_i} = G(h(X;\theta), x_i, \theta_i), \tag{8}$$

for some

$$h : \mathbb{R}^{N\times d} \times \mathbb{R}^N \to \mathbb{R}^m,$$

and

$$G : \mathbb{R}^m \times \mathbb{R}^d \times \mathbb{R} \to \mathbb{R},$$

where $m, d \ll N$.

We refer to h as the *common information* and to G as the *gradient function*. When $X \in \mathbb{R}^{d\times N}$ is partitioned over multiple processors, Prop. 1 implies that a processor having access to θ_i, x_i, and the common information $h(X;\theta)$ can compute the partial derivative $\frac{\partial F}{\partial \theta_i}$. No further information on other variables or datapoints is required other than h. Moreover, computing G is efficient, as its inputs are variables of size $m, d \ll N$.

Recall from (4) and (7) that, when the constraint set is the simplex, adaptations to θ^k take the form:

$$\theta^{k+1} = (1-\gamma^k)\theta^k + \gamma e_{i^*}, \quad \text{where } i^* \in \arg\min_{i\in[N]} \frac{\partial F(\theta^k)}{\partial \theta_i}.$$

Our second property asserts that when θ^k is adapted thusly, the common information h can be easily updated, rather than re-computed from scratch from X and θ^{k+1}:

Property 2. Let $\mathcal{D} = \mathcal{D}_0$. Given $h(X;\theta^k)$, the common information at iteration k of the FW algorithm, the common information $h(X;\theta^{k+1})$ at iteration $k+1$ is:

$$h(X;\theta^{k+1}) = H(h(X;\theta^k), x_{i^*}, \theta^k_{i^*}, \gamma^k), \tag{9}$$

for some

$$H : \mathbb{R}^m \times \mathbb{R}^d \times \mathbb{R} \times \mathbb{R} \to \mathbb{R}^m,$$

where $i^* \in \arg\min_{i\in[N]} \frac{\partial F(\theta^k)}{\partial \theta_i}$.

Prop. 2, therefore, implies that a machine having access to x_{i^*}, $\theta^k_{i^*}$, γ^k, and the common information $h(X;\theta^k)$ in the last iteration can compute the new common information $h(X;\theta^{k+1})$. Again, no additional knowledge of X or θ^k is required. Moreover, similar to the computation of G in Prop. 1, this computation is efficient, as it again only depends on variables of size $m, d \ll N$. As we will see, in establishing that Prop. 2 holds for different problems, we leverage the sparsity of s^k at iteration $k \in \mathbb{N}$, as induced by (7): the fact that θ^k is interpolated with vector e_{i^*}, containing only a single non-zero coordinate, is precisely the reason why the common information can be updated efficiently.

Example: For the sake of concreteness, we give an example of an optimization problem over the simplex that satisfies Properties 1 and 2, namely, ConvexApproximation; additional examples are presented in Section 5. Given a point $p \in \mathbb{R}^d$ and N vectors $x_i \in \mathbb{R}^d, i \in [N]$, the goal of ConvexApproximation is to find the projection of p on the convex hull of set $\{x_i \mid i \in [N]\}$. This can be formulated as:

ConvexApproximation

$$\text{Minimize} \quad F(\theta) = \|X^T\theta - p\|_2^2 \tag{10a}$$

$$\text{subj. to:} \quad \theta \in \mathcal{D}_0, \tag{10b}$$

where $X = [x_i]_{i\in[N]} \in \mathbb{R}^{N\times d}$. ConvexApproximation satisfies Prop. 1 as

$$\frac{\partial F(\theta)}{\partial \theta_i} = 2x_i^T(X^T\theta - p) = G(h(X;\theta), x_i),$$

$i \in [N]$, where common information $h : \mathbb{R}^{N\times d} \times \mathbb{R}^N \to \mathbb{R}^d$ is

$$h(X;\theta) = X^T\theta - p, \tag{11}$$

and gradient function $G : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$ is

$$G(h, x) = 2x^T h.$$

Prop. 1 thus holds when $d \ll N$. Prop. 2 also holds because, under (4) and (7), the common information at step $k+1$ is:

$$\begin{aligned} h(X;\theta^{k+1}) &= (1-\gamma^k)h(X;\theta^k) + \gamma^k(x_{i^*} - p) \\ &= H(h(X;\theta^k), x_{i^*}, \gamma^k), \end{aligned}$$

where $H : \mathbb{R}^d \times \mathbb{R}^d \times \mathbb{R} \to \mathbb{R}^d$ is given by

$$H(h, x, \gamma) = (1-\gamma)h + \gamma(x - p).$$

Note that, in this problem, $m = d \ll N$. Moreover, given their arguments, functions G and H can be computed in $O(d)$ time (i.e., their complexity does not depend on $N \gg 1$).

4.2. A Serial Algorithm

Before describing our parallel version of FW, we first discuss how it can be implemented serially when Properties 1 and 2 hold. The main steps are outlined in Alg. 2. Beyond picking an initial feasible point, the algorithm computes the initial value of the common information h. At each iteration of the for loop, the algorithm computes the gradient ∇F using the present common information,

Algorithm 2 SERIAL FW UNDER PROPERTIES 1 AND 2

1: Pick $\theta^0 \in \mathcal{D}$
2: $h := h(X; \theta^0)$
3: $k := 0$
4: **repeat**
5: **for** each $i \in [N]$ **do**
6: $z_i := G(h, x_i, \theta_i)$
7: **end for**
8: Find $i^* := \arg\min_{i \in [N]} z_i$
9: gap $:= (\theta^k - e_{i^*})^\top z$
10: $\theta^{k+1} := (1 - \gamma^k)\theta^k + \gamma^k e_{i^*}$
11: $h := H(h, x_{i^*}, \theta^k_{i^*}, \gamma^k)$.
12: $k := k + 1$
13: **until** gap $< \epsilon$

and updates both θ^k and the common information h to be used in the next step. It is easy to see that all steps in the main loop of Alg. 2 that involve computations depending on N (namely, Lines 5–10) can be parallelized through map-reduce operations, when X and θ are distributed over multiple processors. We describe this in detail in the next section; crucially, the adaptation of the common information h (Line 11) does *not depend on* N, and can, therefore, be performed efficiently in one processor.

We note here that exploiting Properties 1 and 2 has efficiency advantages even in *serial execution*. In general, the complexity of computing the gradient ∇F as a function of $\theta \in \mathbb{R}^N$ may be quadratic in N, or higher, as each partial derivative $\frac{\partial F}{\partial \theta_i}$, $i \in [N]$, is a function of N variables. Instead, Properties 1 and 2 imply that the complexity of computing the gradient ∇F at each iteration of (4) is $O(N)$: this is the complexity when the common information is adapted through H and used to compute new partial derivatives through the gradient function G. For example, in the case of CONVEXAPPROXIMATION, the complexity is $O(Nd)$. As we show in Section 8, this leads to a significant speedup, allowing Alg. 2 to outperform interior-point methods even when executed serially.

4.3. Parallelization Through Map-Reduce

We now outline how to parallelize Alg. 2 through map-reduce operations. The algorithm is summarized in Alg. 3, where we use the notation $x \mapsto f(x)$ and $x, y \mapsto g(x, y)$, to indicate a unitary function f and a binary function g, respectively. The main data structure D contains tuples of the form (i, x_i, θ^k_i), for $i \in [N]$, partitioned and distributed over P processors. A master processor executes the map-reduce code in Alg. 3, keeping track of the common information h and the duality gap at each step. A `reduce` returns the computed value to the master, while a `map` constructs a new data structure distributed over the P processors.

Each step in the main loop of Alg. 2 has a corresponding map-reduce imple-

Algorithm 3 FW via Map-Reduce

1: Pick $\theta^0 \in \mathcal{D}$
2: Compute $h := h(X;\theta^0)$
3: Let $D := \{(i, x_i, \theta_i^0)\}_{i\in[N]}$
4: Distribute D over P processors
5: $k := 0$
6: **repeat**
7: $\quad D' = D.\texttt{map}\big((i, x_i, \theta_i) \mapsto (i, x_i, \theta_i, G(h, x_i, \theta_i)) \big)$
8: $\quad (i^*,x_{i^*},\theta_{i^*},z_{i^*}) := D'.\texttt{reduce}\left((i,x_i,\theta_i,z_i),(i',x_{i'},\theta_{i'},z_{i'}) \mapsto \begin{cases} (i,x_i,\theta_i,z_i) & \text{if } z_i < z_{i'} \\ (i',x_{i'},\theta_{i'},z_{i'}) & \text{if } z_i \geq z_{i'} \end{cases} \right)$
9: $\quad \text{gap} := D'.\texttt{map}\left((i,x_i,\theta_i,z_i) \mapsto \begin{cases} \theta_i \cdot z_i & \text{if } i \neq i^* \\ (\theta_i - 1) \cdot z_i & \text{if } i = i^* \end{cases} \right).\texttt{reduce}(+)$
10: $\quad D := D.\texttt{map}\left((i, x_i, \theta_i) \mapsto \begin{cases} (i,x_i,(1-\gamma^k)\theta_i) & \text{if } i \neq i^* \\ (i,x_i,(1-\gamma^k)\theta_i + \gamma^k) & \text{if } i = i^* \end{cases} \right)$
11: $\quad h := H(h, x_{i^*}, \theta_{i^*}, \gamma^k)$.
12: $\quad k := k + 1$
13: **until** gap $< \epsilon$

mentation in Alg. 3. In the main loop, a simple `map` using function G appends

$$z_i = \frac{\partial F(\ell^k)}{\partial \theta_i}$$

to every tuple in D, yielding D' (Line 7 in Alg. 3). A `reduce` on D' (Line 8) computes a tuple $(i^*, x_{i^*}, \theta_{i^*}, z_{i^*})$, for $i^* \in \arg\min_{i\in[N]} z_i$. Similarly, a `map` and a `reduce` on D' (a summation) yields the duality gap (Line 9), while a `map` adapts the present solution θ in data structure D (Line 10). Finally, the common information h is adapted centrally at the master node (Line 11), as in Alg. 2.

Message and Parallel Complexity. The `reduce` in Line 8 requires $\log P$ parallel rounds, involving $P-1$ messages of size $O(d)$ (Leighton 2014). Computing the gradient in parallel through a `map` in Line 7 requires knowledge of the common information at each processor. Hence, in the beginning of each iteration, h is broadcast to the P processors over which D is distributed: this again requires in $\log P$ rounds and $P-1$ messages. Note that the corresponding message has size $O(m)$, that does not depend on N. Similarly, the reductions in Lines 9 and 10 require broadcasting i^*, which has size $O(1)$. In practice, such variables are typically shipped to the processors by the master along with the code of the function or operator to be executed by the corresponding `map` or `reduce`. The operations in Lines 7–10 thus require $\log P$ parallel rounds and the transmission of $O(P)$ messages of size $O(m + d)$.

4.4. Selecting the step size.

Our exposition so far assumes that the step size γ^k is computed at the master node before updating D and h. This is certainly the case if, e.g.,

$$\gamma^k = \frac{2}{k+2},$$

but it does not readily follow when the line minimization rule (6) is used. Nevertheless, all problems we consider here, including ConvexApproximation, sat-

Problems	$F(\theta)$	m	G compl.	H compl.
Convex Approximation	$\|X\theta - p\|_2^2$	d	$O(d)$	$O(d)$
Adaboost	$\log\left(\sum_{j=1}^{d} \exp(\alpha c_j r_j)\right)$	d	$O(d)$	$O(d)$
D-optimal Design	$-\log\det A(\theta)$	d^2	$O(d^2)$	$O(d^2)$
A-optimal Design	$\operatorname{trace}\left(A^{-1}(\theta)\right)$	$2d^2$	$O(d^2)$	$O(d^2)$

Table 2. Examples of problems satisfying Prop. 1–3. As we see time complexity of computing G and H functions are independent of N.

isfy an additional property that ensures that (6) can also be computed efficiently in a centralized fashion:

Property 3. There exists an $\hat{F} : \mathbb{R}^m \to \mathbb{R}$ such that $F(\theta) = \hat{F}\left(h(X;\theta)\right)$.

Prop. 3 implies that line minimization (6) at iteration k is:

$$\gamma^k = \arg\min_{\gamma\in[0,1]} \ \hat{F}\left(h(X;(1-\gamma)\theta^k + \gamma e_{i^*})\right). \tag{12}$$

The argument of $\hat{F}$ is the updated common information h^{k+1} under step size γ. Hence, using Prop. 2, Eq. (12) becomes:

$$\gamma^k = \arg\min_{\gamma\in[0,1]} \ \hat{F}\left(H(h, x_{i^*}, \theta^k_{i^*}, \gamma)\right), \tag{13}$$

where h is the present common information. As F is convex in θ^k, it is also convex in γ, so (13) is also a convex optimization problem. Crucially, (13) depends on the full dataset X and the full variable θ only through h. Therefore, the master processor (having access to x_{i^*}, $\theta^k_{i^*}$, γ, and h) can find the step size via standard convex optimization techniques solving (13). In fact, for several of the problems we consider here, line minimization has a closed form solution; for example, for ConvexApproximation, the optimal step size is given by:

$$\gamma^k = \frac{h^\top h - (x_{i^*} - p)^\top h}{(x_{i^*} - p)^\top (x_{i^*} - p) + h^\top h - 2(x_{i^*} - p)^\top h}.$$

Though all problems we study, listed in Table 2, satisfy Prop. 1, 2, *as well as* 3, we stress again that Prop. 3 is not strictly necessary to parallelize FW, as a parallel implementation can always resort to a diminishing step size.

5. Examples

We provide several examples of problems that satisfy Prop. 1, 2, and 3; a summary is given in Table 2.

Experimental Design: In experimental design, a learner wishes to regress a linear model $\beta \in \mathbb{R}^d$ from input data $(x_i, y_i) \in \mathbb{R}^d \times \mathbb{R}, i \in [N]$, where

$$y_i = \beta^\top x_i + \epsilon_i,$$

for ϵ_i, $i \in [N]$, i.i.d. noise variables. The learner has access to features $x_i, i \in [N]$, and wishes to determine which labels y_i to collect (i.e., which experiments to conduct) to accurately estimate β. This problem can be posed as (Boyd &

Vandenberghe 2004):

$$\min_{\theta\in\mathcal{D}_0} \mathtt{f}\left(\left(\sum_{i=1}^{N}\theta_i x_i x_i^\top\right)^{-1}\right), \tag{14}$$

where θ_i indicates the portion of experiments conducted by the learner with feature x_i. The quantity

$$A(X;\theta)=\sum_{i=1}^{N}\theta_i x_i x_i^\top$$

is the *design matrix* of the experiment. For brevity, we represent $A(X;\theta)$ as $A(\theta)$ below. Different choices of $\mathtt{f}:\mathbb{R}^{d\times d}\to\mathbb{R}$ lead to different optimality criteria; we review two below.

D-Optimal Design: In D-Optimal design $\mathtt{f}$ is the log-determinant, and (14) becomes:

D-OptimalDesign

$$\text{Minimize}\quad F(\theta)=\operatorname{logdet}\left(\sum_{i=1}^{N}\theta_i x_i x_i^\top\right)^{-1} \tag{15a}$$

$$\text{subj. to:}\quad \theta\in\mathcal{D}_0, \tag{15b}$$

D-OptimalDesign satisfies Prop. 1 as:

$$\tfrac{\partial F}{\partial\theta_i}=-x_i^\top A^{-1}(\theta)x_i=G(h(X,\theta),x_i),\qquad \text{for all } i\in[N],$$

where the common information $h:\mathbb{R}^{N\times d}\times\mathbb{R}^N\to\mathbb{R}^{d\times d}$ is

$$h(X;\theta)=A^{-1}(\theta),$$

and the gradient function $G:\mathbb{R}^{d\times d}\times\mathbb{R}^d\to\mathbb{R}$, is given by

$$G(h,x)=-x^\top hx.$$

Hence, Prop. 1 holds when $d^2\ll N$. Using the Sherman-Morrison formula (Sherman & Morrison 1950) we can show that the common information at step $k+1$ is:

$$A^{-1}(\theta^{k+1})=\tfrac{A^{-1}(\theta^k)}{1-\gamma}-\tfrac{\frac{\gamma}{(1-\gamma)^2}A^{-1}(\theta^k)x_{i^*}x_{i^*}^\top A^{-1}(\theta^k)}{1+\frac{\gamma}{1-\gamma}x_{i^*}^\top A^{-1}(\theta^k)x_{i^*}}. \tag{16}$$

As a result, $h(X;\theta^{k+1})=H(h(X,\theta^k),x_{i^*},\gamma)$, where $H:\mathbb{R}^{d\times d}\times\mathbb{R}^d\times\mathbb{R}\to\mathbb{R}^{d\times d}$ is:

$$H(h,x,\gamma)=\tfrac{h}{1-\gamma}-\tfrac{\frac{\gamma}{(1-\gamma)^2}hxx^\top h}{1+\frac{\gamma}{1-\gamma}x^\top hx}. \tag{17}$$

Therefore, Prop. 2 also holds. Note that, in this problem, $m=d^2\ll N$. Functions G and H include only matrix-to-vector and vector-to-vector multiplications; hence, given their arguments, they can be computed in $O(d^2)$ time.

A-Optimal Design: In A-Optimal design $\mathtt{f}$ is the trace:

A-OptimalDesign

$$\text{Minimize}\quad F(\theta)=\operatorname{Tr}\left(A^{-1}(\theta)\right) \tag{18a}$$

$$\text{subj. to:}\quad \theta\in\mathcal{D}_0. \tag{18b}$$

The partial derivative of the F can be written as:

$$\frac{\partial F}{\partial \theta_i} = -x_i^\top A^{-2}(\theta) x_i = G(h(X;\theta), x_i), \qquad \text{for all } i \in [N].$$

where the common information $h : \mathbb{R}^{N\times d} \times \mathbb{R}^N \to \mathbb{R}^{d\times d} \times \mathbb{R}^{d\times d}$ is $h(X;\theta) = (h_1, h_2)$, where

$$h_1 = A^{-1}(\theta)$$

and

$$h_2 = A^{-2}(\theta).$$

The gradient function $G : \mathbb{R}^{d\times d} \times \mathbb{R}^d \to \mathbb{R}$ is

$$G((h_1, h_2), x) = -x^\top h_2 x.$$

Hence, Property 1 holds when $d^2 \ll N$. The common information at step $k+1$ is

$$\left(A^{-1}(\theta^{k+1}), A^{-2}(\theta^{k+1})\right).$$

The first term can be computed as in (16). The second term is the square of the first term; expanding it gives a formula in terms of $A^{-1}(\theta^k)$ and $A^{-2}(\theta^k)$. More formally, the common information at iteration $k+1$ can be written as:

$$h(X;\theta^{k+1}) = (h_1^{k+1}, h_2^{k+1}) = H(h(X;\theta^k), x_{i^*}, \gamma),$$

where

$$H((h_1, h_2), x, \gamma), = (H_1(h_1, x, \gamma), H_2(h_1, h_2, x, \gamma)),$$

and function H_1 is given by (17), while $H_2 : \mathbb{R}^{d\times d} \times \mathbb{R}^{d\times d} \times \mathbb{R}^d \times \mathbb{R} \to \mathbb{R}^{d\times d}$ is:

$$H_2(h_1,h_2,x,\gamma)=\frac{h_2}{(1-\gamma)^2} - \frac{\frac{\gamma}{(1-\gamma)^3} h_2 x x^\top h_1}{1+\frac{\gamma}{1-\gamma} x^\top h_1 x_i} - \frac{\frac{\gamma}{(1-\gamma)^3} h_1 x x^\top h_2}{1+\frac{\gamma}{1-\gamma} x^\top h_1} + \frac{\frac{\gamma^2}{(1-\gamma)^4} x^\top h_2 x h_1 x x^\top h_2}{(1+\frac{\gamma}{1-\gamma} x^\top h_1 x)^2}.$$

This illustrates why common information includes both $A^{-1}(\theta^k)$ and $A^{-2}(\theta^k)$: adapting the latter requires knowledge of both quantities. Note also that $m = 2d^2 \ll N$. Functions G and H again only require matrix-to-vector and vector-to-vector multiplications and, hence, can be computed in $O(d^2)$ time.

AdaBoost: Assume that N classifiers and ground-truth labels for d data points are given. The classification result is represented by a binary matrix $X \in \{-1, +1\}^{N\times d}$, where x_{ij} is the label generated by the i-th classifier for the j-th data point. The true classification labels are given by a binary vector $r \in \{-1, +1\}^d$. The goal of Adaboost is to find a linear combination of classifiers, defined as:

$$c(X, \theta) = X^\top \theta,$$

such that the mismatch between the new classifiers and ground-truth labels is minimized. The problem can be formulated as:

AdaBoost

$$\text{Minimize} \quad F(\theta) = \log\left(\textstyle\sum_{j=1}^d \exp(-\alpha c_j(X,\theta) r_j)\right) \tag{19a}$$

$$\text{subj. to:} \quad \theta \in \mathcal{D}_0, \tag{19b}$$

where r_j and c_j are, respectively, the j th element of the r and c vectors, and $\alpha \in \mathbb{R}$ is a tunable parameter.

Again, (19) satisfies Prop. 1 as:

$$\frac{\partial F(\theta)}{\partial \theta_i} = -x_i^\top b = G(h(X;\theta), x_i), \qquad \text{for all } i \in [N],$$

where $b \in \mathbb{R}^d$ is a vector, whose elements are

$$b_j = \frac{\alpha r_j \exp\left(-\alpha c_j r_j\right)}{\sum_{i=1}^d \exp(-\alpha c_j r_j)}, \qquad \text{for all } j \in [d].$$

The common information, $h : \mathbb{R}^{N\times d} \times \mathbb{R}^N \to \mathbb{R}^d$ is

$$h(X;\theta) = \left[\exp -\alpha c_j r_j\right]_{j\in[d]},$$

and the gradient function $G : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$ is

$$G(h,x) = x^\top \hat{h},$$

where

$$\hat{h} = \left[\frac{\alpha r_j h_j}{\sum_{i=1}^d h_i}\right]_{j\in[d]}.$$

Hence, Prop. 1 holds when $d \ll N$. Prop. 2 also holds because, under (4) and (7), the common information at step $k+1$ is

$$h(X;\theta^{k+1}) = H(h(X,\theta^k), x_{i^*}, \gamma),$$

where $H : \mathbb{R}^d \times \mathbb{R}^d \times \mathbb{R} \to \mathbb{R}^d$ is given by

$$H(h, x_i, \gamma) = \left[h_j^{(1-\gamma)} \exp(-\gamma\alpha x_{ji} r_j)\right]_{j\in[d]}.$$

In this problem, $m = d \ll N$ and functions G and H can be computed in $O(d)$ time.

Serial Solvers: All four problems in Table 2 are convex, and some admit specialized solvers. A-OptimalDesign can be reduced to a semidefinite program, (see Sec. 7.5 of (Boyd & Vandenberghe 2004)), and solved as an SDP. AdaBoost can be expressed as a geometric program (GP) (Clarkson 2010), and ConvexApproximation is a quadratic program (QP). D-OptimalDesign is a general convex optimization problem, and can be solved by standard techniques such as, e.g., barrier methods. In Sec. 8 we compare FW to the above specialized solvers, and we see that it outperforms them in all cases.

6. Extensions

Our proposed distributed Frank-Wolfe algorithm can be extended to a more general class of problems, with constraints beyond the simplex.

ℓ_1-constraint: The ℓ_1 (or *lasso*) constraint

$$\|\theta\|_1 \leq K$$

appears in many optimization problems as means of enforcing sparsity (Ng 2004, Tibshirani 1996). For this constraint, adaptation (4b) becomes:

$$s^k = \sigma_{i^*} e_{i^*}, \text{ where } i^* = \arg\max_{i\in[N]} \left|\frac{\partial f}{\partial \theta_i}\right|, \tag{20}$$

and

$$\sigma_{i^*} = -K\text{sign}(\frac{\partial f}{\partial \theta_{i^*}}).$$

Eq. (20) can be computed in parallel through a `reduce`. The adaptation step of γ^k is slightly different from the simplex case, as we interpolate between θ^k a scaled basis vector $\sigma_{i^*}e_{i^*}$.

As an example, consider the LASSO problem (Tibshirani 1996):

$$\min_{\theta:\|\theta\|_1 \leq K} \|X^\top \theta - p\|_2^2. \tag{21}$$

Here, $\theta \in \mathbb{R}^N$ is the vector of weights, $X \in \mathbb{R}^{N\times d}$ is the matrix of $N-$dimensional features for d datapoints, and $p \in \mathbb{R}^d$ is the observed outputs. Note that LASSO has exactly the same objective as ConvexApproximation, so the common information from (11) is $h(X;\theta) = X^T\theta - p$. The common information can be updated as

$$h(X;\theta^{k+1}) = (1-\gamma^k)h(X;\theta^k) + \gamma^k(\sigma_{i^*}x_{i^*} - p),$$

i.e., it is a function of $h(X;\theta^k)$ and the usual "local" information at i^*, now including also σ_{i^*}.

Atomic Norms: More generally, consider the problem

$$\min_{\theta:\|\theta\|_{\mathcal{A}} \leq K} f(\theta),$$

where $\|x\|_{\mathcal{A}}$ denotes the *atomic norm*: given a set of atoms $\mathcal{A} = \{a_i \in \mathbb{R}^N\}$ the atomic norm is defined as

$$\|x\|_{\mathcal{A}} = \inf\{t \geq 0 : x \in t\mathcal{C}_{\mathcal{A}}\},$$

where $\mathcal{C}_{\mathcal{A}}$ is the convex hull of the atoms. Atomic norms are used to encourage solutions that have a low-dimensional structure, modelled as a linear combination of only few atoms (Shah et al. 2012, Chen & Banerjee 2015, Chandrasekaran et al. 2012, Tewari et al. 2011). Tewari et al. (Tewari et al. 2011) propose an FW-like algorithm for this class of problems. In this algorithm, the step 4 of Alg. 1 is replaced by

$$s^k = \arg\min_{a\in\mathcal{A}} a^\top \cdot \nabla F(\theta^k). \tag{22}$$

Then, the new solution is convex combination of the current solution and Ks^k, similar to FW Algorithm.

Our approach can be extended to problems of this form, where the set $\mathcal{A}$ comprises atoms $\{\pm\alpha_i e_i\}$, where $\alpha_i > 0$ s are arbitrary scalars. Eq. (22) becomes

$$s^k = -\alpha_{i^*}\text{sign}(\frac{\partial f}{\partial \theta_{i^*}})e_{i^*},$$

where

$$i^* = \arg\max_{i\in[N]} |\alpha_i \frac{\partial f}{\partial \theta_i}|.$$

This can be implemented through a reduce, and adaptation is slightly different from the simplex case as again s^k is a scaled basis vector. An appropriate variant of Prop. 2, should hold w.r.t. this adaptation step.

7. Implementation

We implemented Alg. 3 over Spark, an open-source cluster-computing framework (Zaharia et al. 2010). Spark inherently supports map-reduce operations, and is well-suited for parallelizing iterative algorithms; this is because results of map-reduce operations can be cached in RAM, over multiple machines, and accessed in the next iteration of the algorithm (Zaharia et al. 2010).

Our FW implementation is generic, relying on an abstract class. A developer only needs to implement three methods in this class: (a) the gradient function G, (b) the common information function h, and (c) the common information adaptation function H. Once these functions are implemented, our code takes care of executing Alg. 3 in its entirety, and distributes its execution over a Spark cluster. Our implementation, which is publicly available,[1] can thus be used to solve arbitrary problems that satisfy Prop. 1 and 2, and quickly deploy and parallelize their execution over a Spark cluster. We have also instantiated this class for the problems summarized in Table 2 and used it in our experiments.

8. Experiments

8.1. Experiment Setup

Cluster. Our cluster comprises 8 worker machines. Each worker has 2 Intel(R) Xeon(R) CPUs (E5-2680 v4) with 2.4GHz clock speed and 14 cores, at 28 cores in total. Moreover, each core supports hyper-threading; as a result, each physical core appears as two logical cores to the operating system. Therefore, each worker has

$$2\text{CPU} \times 14\frac{\text{cores}}{\text{CPU}} \times 2\frac{\text{threads}}{\text{core}} = 56 \text{ threads (virtual cores)},$$

and the cluster has $8 \times 56 = 448$ (virtual) cores in total. Thus, the maximum level of parallelism for our cluster is 448. Also, each worker has 529 GB of RAM, 32KB L1 cache for instruction and data, 256KB for L2 cache, and 35.84MB for L3 cache. The cluster has 4TB of RAM in total. All code is implemented in Python (v2.7.5) and Spark (v1.4.1); we also use python's CVXOPT module (v1.1.8).

Algorithms. We solve Convex Approximation, Adaboost, D-Optimal Design, and A-Optimal Design summarized in Table 2, as well as LASSO (c.f. Sec. 6). We implement both serial and parallel solvers. First, we implement Serial FW (Alg. 2) in Python, setting γ using the line minimization rule (6). In addition, we solve Convex Approximation, D-Optimal Design, A-Optimal Design, and Adaboost using CVXOPT solvers, `qp`, `cp`, `sdp`, and `gp`, respectively. CVXOPT is a software package for convex optimization based on the Python programming language.[2] Beyond Serial FW and using CVXOPT solvers, we also implement a third, naïve serial algorithm in which the gradient is computed from scratch at each iteration, not exploiting the common information introduced in Prop. 1 and 2 (as Serial FW does). We call this implementation Oracle FW, as it computes the gradient via a "function oracle". We also implement our parallel algorithm

[1] `https://github.com/neu-spiral/FrankWolfe`
[2] `cvxopt.org`

(Alg. 3) using our Spark generic implementation. We again set the step size using the line minimization rule (6). We refer to this algorithm as Parallel FW. We control the level of parallelism, i.e., the number of cores P, by either setting the number of partitions of Spark resilient distributed datasets (RDDs) to P or by controlling the `total-executor-cores` in Spark's configuration and using a fixed high number of partitions, e.g., 600. We use the former approach when dealing with smaller datasets and the latter for larger datasets, as maintaining a large number of small partitions (executed serially) avoids memory crashes in Spark. We also introduce two stochastic parallel variants that subsample the gradient; we discuss these in Section 8.4. Finally, we also implement distributed ADMM for the LASSO problem, as described in Section 8.3 of (Boyd et al. 2011).

Synthetic Data. For D-optimal Design, A-optimal Design, Convex Approximation, and LASSO, the synthetic data has the form of a matrix $X \in \mathbb{R}^{N\times d}$. The point p in Convex Approximation is a vector $p \in \mathbb{R}^d$. The elements of X and p are sampled independently from a uniform distribution in $[0, 1]$. For Adaboost, input data is given by a binary matrix $X \in \{-1, +1\}^{N\times d}$ and ground-truth labels are represented by a binary vector $r \in \{-1, +1\}^d$. The elements of r are sampled independently from a Bernoulli distribution with parameter 0.5. Then each row of X is generated from r as follows: each element x_{ij} is equal to r_j with probability 0.7, and it is equal to $-r_j$ with probability 0.3. For LASSO, the observed outputs are denoted by a vector $p \in \mathbb{R}^d$, which is generated as follows: a sparse vector $\theta^* \in \mathbb{R}^N$ is sampled from a uniform distribution in [0,1], s.t., only 1 percent of its elements are non-zero. Then the vector p is synthesized as $p = X^\top\theta^* + \epsilon$, where $\epsilon \in \mathbb{R}^d$ is the noise vector, and its elements are sampled from a uniform distribution in $[0, 0.01]$. We create three synthetic datasets with different values of N and d, summarized in Tables 3–5.

Real Data. We also experiment with 4 real datasets, summarized in Table 6. The first dataset is Movielens (Harper & Konstan 2015). This includes 20,000,263 ratings for 27,278 movies generated by 138,493 users. We have kept the top 500 most-rated movies, resulting in 413,304 ratings, rated by 137,768 users. We have represented the data as a matrix $X \in \mathbb{R}^{N\times d}$ with $N = 137768$ and $d = 500$, so that x_{ij} indicates the rating of user i for movie j. Missing entries are set to zero. The second dataset is a high-energy physics dataset, HEPMASS (Lichman 2013). The dataset has 10^6 data points and 28 features. We represent it as a matrix with $N = 10^6$ and $d = 28$. The third dataset is the MSD dataset (Lichman 2013), which comprises 515345 songs with 90 features. We represent it as a matrix with $N = 515345$ and $d = 90$. The fourth dataset is from Yahoo Webscope.[3] It represents a snapshot of the Yahoo! Music community's preferences for various songs. We used the test section of the dataset, which contains 18,231,790 ratings of 136,735 songs by over 1.8M users. We find the 100-dimensional latent vectors via matrix factorization technique (Koren et al. 2009), using the parameters $\mu = 0.001$ and $\lambda = 0.001$. We represent the latent vectors corresponding to users as a matrix $X \in \mathbb{R}^{N\times d}$ with $N = 1,823,179$ (number of users) and $d = 100$. We refer to this dataset as YAHOO dataset. When solving Convex Approximation problem for the YAHOO dataset, the vector $p \in \mathbb{R}^{100}$ is generated as follows. An arbitrary point from the dataset x_i is chosen, then it is corrupted by noise: $p = x_i + \epsilon$, where the elements of $\epsilon \in \mathbb{R}^{100}$ are sampled independently form a uniform distribution in $[0, 0.1]$. Finally, the point x_i is removed from the dataset.

[3] `https://webscope.sandbox.yahoo.com`

Problem	N	d	algs
Conv. Approx.	5000	20	`qp`
Adaboost	5000	100	`gp`
D-opt. Design	5000	20	`cp`
A-opt. Design	5000	20	`sdp`

Table 3. Dataset A: Dataset used for serial experiments (see Section 8.2).

Problem	N	d	ϵ
Conv. Approx.	100M	100	0.15
Adaboost	100M	100	0.004
D-opt. Design	20M	100	0.09
A-opt. Design	20M	100	0.19

Table 4. Dataset B: Dataset with $N = O(10M)$ and $d = 100$, used for parallel algorithms (see Sections 8.3 and 8.4).

Metrics. We use two metrics. The first is the objective F of each problem, whose evolution we track as different algorithms progress. Our second metric is t_ϵ, the minimum time for the algorithm to obtain a solution θ within an ϵ-neighborhood of the optimal solution $F(\theta^*)$. As we do not know $F(\theta^*)$, we use $F(\theta) - g(\theta) \leq F(\theta^*)$ instead. More formally:

$$t_\epsilon = \min \left\{ t : \frac{F(\theta(t))}{F(\theta(t)) - g(\theta(t))} \leq 1 + \epsilon \right\}, \tag{23}$$

where $\theta(t)$ denotes the obtained solution at time t. As $F(\theta) - g(\theta) \leq F(\theta^*)$, t_ϵ overestimates the time to convergence.

8.2. Serial Execution

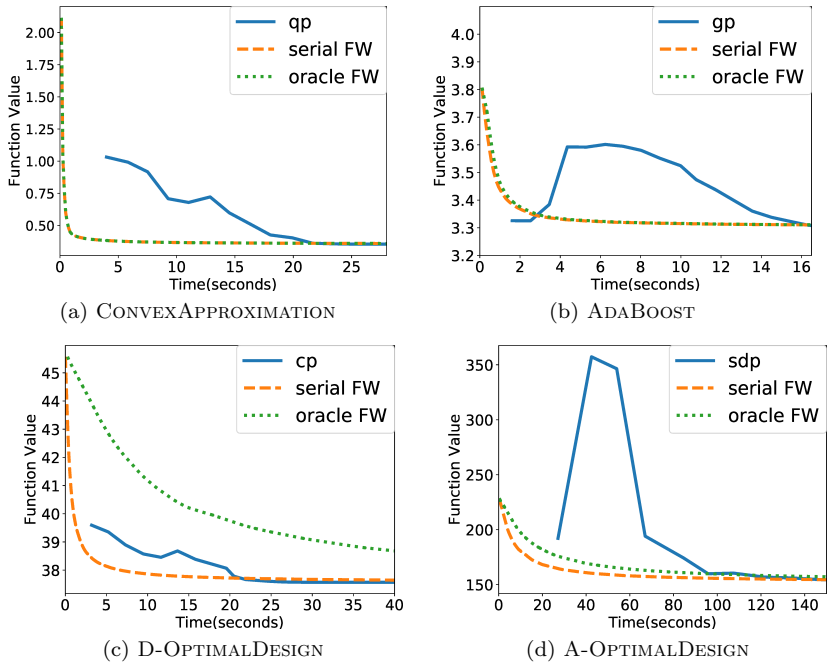
Our first experiment compares the Serial FW algorithm with (a) the specialized interior point solvers mentioned in Section 5 (i.e., `cp`, `qp`, `sdp`, and `gp`) and (b) with Oracle FW, for each of the problems in Table 2. We use the small synthetic dataset (Dataset A) in Table 3.

Problem	N	d	ϵ
Conv. Approx.	500000	5000	0.13
Adaboost	500000	5000	0.003
D-opt. Design	100000	1000	0.03
A-opt. Design	100000	1000	0.09

Table 5. Dataset C: Dataset with $N = O(100K)$ and $d = 1K$, used for parallel algorithms (see Sections 8.3 and 8.4).

Problem	Dataset	N	d	ϵ
D-opt. Design	Movielens	137768	500	0.18
D-opt. Design	HEPMASS	1M	38	0.04
D-opt. Design	MSD	515345	90	0.01
D-opt. Design	YAHOO	1,823,179	100	0.09
A-opt. Design	YAHOO	1,823,179	100	0.17
Conv. Approx.	YAHOO	1,823,178	100	0.03

Table 6. Real Datasets: Real-wold datasets used for parallel experiments in Section 8.3.



(a) ConvexApproximation (b) AdaBoost

(c) D-OptimalDesign (d) A-OptimalDesign

Fig. 1. Values of the objective function generated by the algorithms as a function of time over Dataset A. We see that Serial FW converges faster than interior point methods. Comparing it to Oracle FW, the benefits of exploiting the common information in serial computation are more pronounced for experimental design objectives, where partial derivative computation is quadratic.

In each execution, we keep track of the objective function F as a function of time elapsed. Unlike FW, the interior-point methods do not generate feasible solutions at each iteration. Therefore, we project the solutions at each iteration on the feasible set, and compute the objective F on the projected solution. The time taken for the projection is not considered in time measurements; as such, our plots underestimate the time taken by the interior-point algorithms.

Fig. 1 shows function values generated by the algorithms as a function of time. Serial FW outperforms the interior-point methods, even when not accounting for projections. The reason is that, in contrast to interior-point methods, the time

Problem	Dataset	Speedup vs. Parallel FW with P=1	Speedup vs. Serial FW	Speedup vs. Oracle FW	# of cores
Conv. Approx.	Dataset C	42	1.12	1.45	128
Conv. Approx.	Dataset B	98	28.57	29.82	350
Conv. Approx.	YAHOO	78	18.6	21.83	210
Adaboost	Dataset C	45	1.24	2.07	128
Adaboost	Dataset B	133	35.4	35.47	350
D-opt. Design	Dataset C	48	12.7	112.3	128
D-opt. Design	Dataset B	126	36.7	271.6	350
D-opt. Design	HEPMASS	35	7.5	23.28	64
D-opt. Design	Movielens	33	3.77	115.5	64
D-opt. Design	MSD	35	6.52	37.24	64
D-opt. Design	YAHOO	93	19	159.1	210
A-opt. Design	Dataset C	49	10.5	125.07	128
A-opt. Design	Dataset B	102	32.5	273.12	350
A-opt. Design	YAHOO	90	20.3	164.95	210

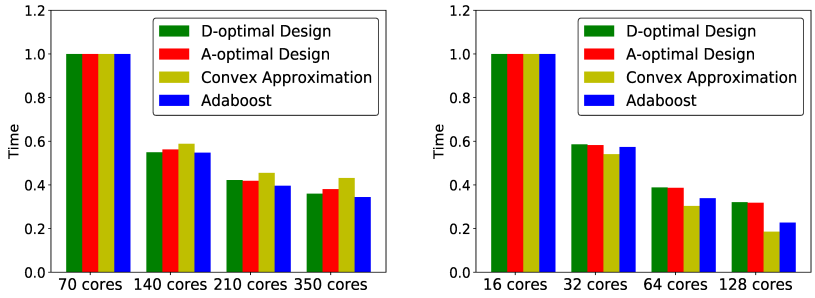
Table 7. A summary of speedups (over three serial implementations) obtained by parallel FW for each problem and dataset, along with the level of parallelism. Beyond this number of cores, no significant speedup improvement is observed.

complexity of computations at each iteration of Serial FW is linearly dependent on N. As a result, when $d \ll N$, Serial FW is considerably faster, even though it requires more iterations to converge. Note that the objective values generated by interior-point methods are non-monotone, as these methods alternate between improving feasibility and optimality. Comparing Serial FW to Oracle FW, the benefits of exploiting the common information in serial computation are more pronounced for experimental design objectives, where partial derivative computation is quadratic.

8.3. Effect of Parallelism

We compare the speedup of Parallel FW over three serial implementations. The first is Parallel FW with $P = 1$, i.e., our parallel Spark code using only one processor. The second is Serial FW, as described in Alg. 2; the third is Oracle FW, which computes the gradient naïvely from scratch, not exploiting the common information introduced in Prop. 1 and 2. We do not report results of serial execution via CVXOPT, as the latter crashes with out-of-memory errors on all these inputs. We execute 10 iterations of these serial implementations and then estimate the total running time based on the average per-iteration running time; all values reported correspond to the same number of iterations.

The measured speedups of Parallel FW over these serial implementations are shown in Table 7. Increasing parallelism leads to significant speedups. For example, using 350 compute cores, we can solve the 20M-variable instance of D-optimal Design in 79 minutes, when Serial FW would take and 48.3 hours for the same problem and input. Note that, even in serial implementation, exploiting Properties 1-3 leads to accelarated execution: this is evident from the fact that



(a) Dataset B (b) Dataset C

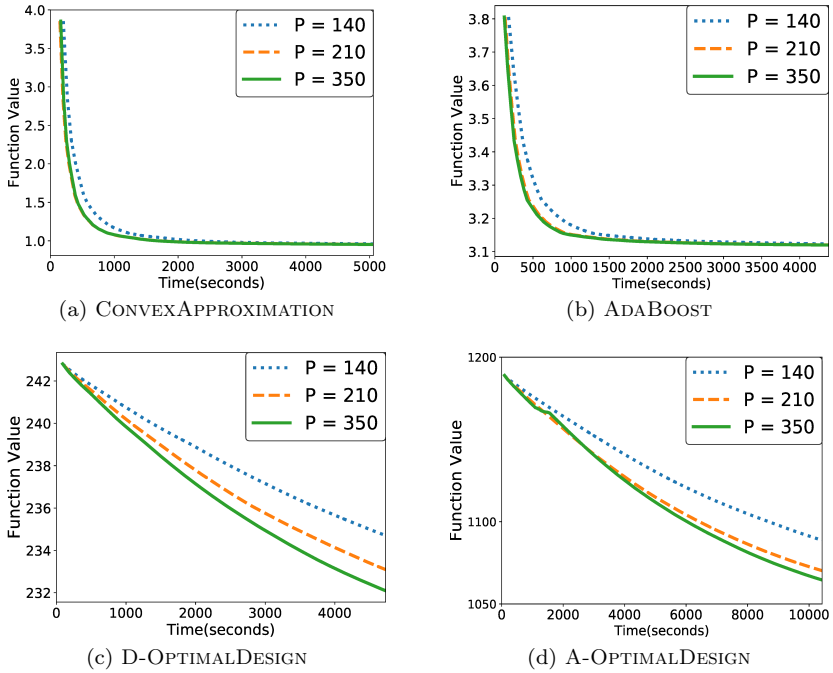
Fig. 2. t_ϵ as a function of the level of parallelism, measured in terms of cores P. Fig. 2a shows results on the $O(20\text{M})$ variable dataset (Table 4) while Fig. 2b shows results on the dataset with $d = O(1000)$ (Table 5). We normalize t_ϵ by its value at the lowest level of parallelism (13134s, 27420s, 11727s, and 11375s, respectively, for each of the four problems in Fig. 2a and 487s, 486s, 2096s, and 4783s, respectively, in Fig. 2b). We see that increasing the level of parallelism speeds up convergence.

the speedup over Oracle FW is considerably higher than over Serial FW. This is more prominent in the two experimental design objectives: this is expected, as the complexity of computing the gradient is $O(Nd^3)$, while by using the common information we can compute the gradient in $O(Nd^2)$. For example, the same 20M-variable D-optimal instance would take more than 14 days for Oracle FW.

We note that, for the input sizes used in these experiments, the benefit of parallelism saturates beyond 350 cores and 128 cores, for Datasets B and C, respectively. The reason is that for this input size, after increasing the level of parallelism beyond these values, the cost of computing the gradient at each core becomes negligible.

We further illustrate the effect of increasing parallelism on two large-scale synthetic datasets: Dataset B, a dataset with $N = O(20\text{M})$ and $d = 100$ (Table 4), and Dataset C with $N = O(100\text{K})$ and $d = O(1\text{K})$ (Table 5). Fig. 2 shows t_ϵ as a function of the level of parallelism, measured in terms of the number of cores P, for each of the two datasets. We normalize t_ϵ by its value at $P = 70$ and $P = 16$, respectively. This lowest level of parallelism ($P = 70$ and $P = 16$) is chosen so that the slowest execution time is moderate, i.e., approximately 10 hours. Figure 3 shows objective F, as a function of time for different levels of parallelism. The highest level of parallelism (e.g., 350 for dataset B) is the saturation point, beyond which no significant speedup is observed. By comparing Figures 3a and 3b with Figures 3c and Figure 3d, we see that Parallel FW converges much faster for Convex Approximation and Adaboost. The reason is that the objective function in D-Optimal Design and A-optimal Design does not have a bounded curvature; therefore, as mentioned in Section 4, FW for these problems does not have a $O(\frac{1}{k})$ convergence rate.

We also illustrate how parallelism affects performance on real datasets, summarized in Table 6. For brevity, we only report D-Optimal Design for Movielens, HEPMASS, and MSD datasets, and D-optimal design, A-optimal Design, and Convex Approximation for the YAHOO dataset. Fig. 4 shows the measured t_ϵ for different levels of parallelism. For each dataset, t_ϵ is normalized by the value



(a) ConvexApproximation (b) AdaBoost

(c) D-OptimalDesign (d) A-OptimalDesign

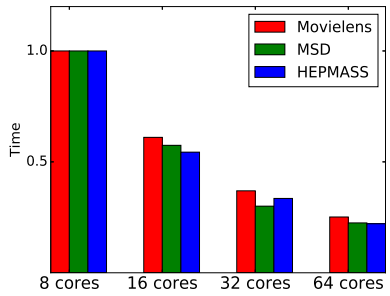
Fig. 3. Objective F as a function of time over Dataset B. We see that increasing the level of parallelism makes convergence faster. By comparing Figures 3a and 3b with Figures 3c and 3d, we see that FW for D-Optimal Design and A-Optimal Design converges slower.

of t_ϵ for the lowest level of parallelism. Again, we see that we gain a significant speedup by parallelism.

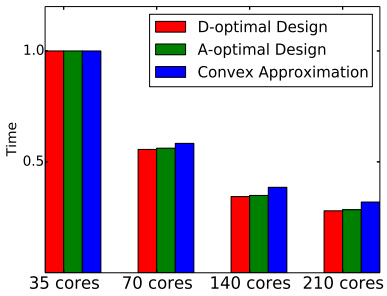
8.4. Subsampling the Gradient

In this section, we study the effect of subsampling the gradient on the performance of FW. We have seen that parallelism reduces the cost of computation of the gradients. An alternative is to compute the gradient stochastically by subsampling only a few partial derivatives and using the minimal in this sub-sampled set. This reduces the amount of computation occurring in each iteration. Moreover, such a stochastic estimation of the gradient still guarantees convergence (Reddi et al. 2016), albeit at a slower rate. Therefore, subsampling decreases the computation time for each iteration; this has a similar effect to increasing parallelism, without incurring additional communication overhead. In contrast to increasing parallelism, however, subsampling may also increase the number of iterations till convergence.

We consider two variants of subsampling. In Sampled FW, we compute each partial derivative $\frac{\partial F}{\partial \theta_i}$ with probability p. Then, we find the minimum among the computed partial derivatives. Note that this speeds derivative computations: at most $p \cdot N$ partial derivatives are computed, in expectation. In Smoothened

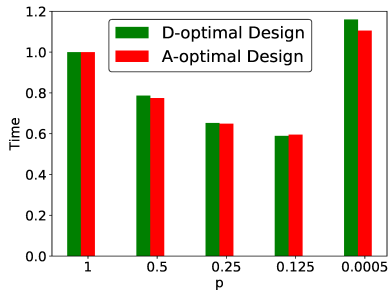


(a) D-optimal Design for Movielens, MSD, and HEPMASS

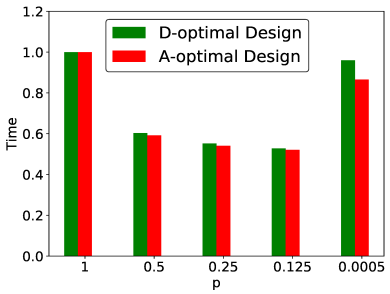


(b) D-optimal Design, A-optimal Design, and Convex Approximation for the YAHOO dataset

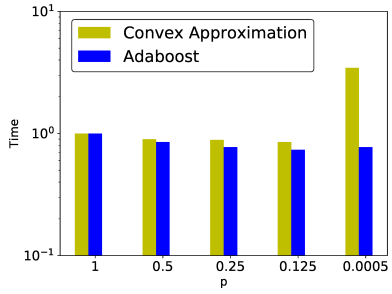
Fig. 4. Summary of parallelism experiments on the real datasets. We normalize t_ϵ by its value at the lowest level of parallelism (15247s, 3899s, and 4766s for Movielens, MSD, and HEPMASS, respectively, in Fig. 4a, and 9888s, 7060s, and 1302s for D-optimal Design, A-optimal Design, and Convex Approximation, respectively, in Fig 4b.



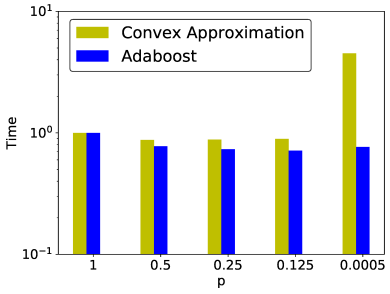
(a) Sampled FW for D-OptimalDesign and A-OptimalDesign



(b) Smoothened FW for D-OptimalDesign and A-OptimalDesign



(c) Sampled FW for AdaBoost and ConvexApproximation



(d) Smoothened FW for AdaBoost and ConvexApproximation

Fig. 5. Measured t_ϵ under Sampled and Smoothened FW, over Dataset C. We normalize t_ϵ by the measured t_ϵ for 16 cores, which is reported in Fig. 2. By comparing Figures 5a and 5c with Fig. 2b, subsampling does not match the benefits of parallelism. In an ultra-low regime, e.g., $p = 0.0005$ convergence is very slow. Smoothened FW can enhance the performance in this case.

FW, we compute each partial derivative with probability p, but maintain an exponentially-weighted moving average (EWMA) between the computed value and past values: this estimate is used instead to compute the current minimum partial derivative.

We use Dataset C (Table 5) in this experiment: we solve the corresponding problems using Sampled FW and Smoothened FW on 16 cores. The results are shown in Fig. 5. Values t_ϵ are normalized by t_ϵ for $p = 1$. This makes experiments in Figures 5 and 2b comparable: each core computes the same number of partial derivatives in expectation.

By comparing Figures 5 and 2b, we see that subsampling matches the benefits of parallelism, at least for large p, for D-optimal and A-optimal design. In contrast, the benefits of subsampling for Convex Approximation and AdaBoost are almost negligible. This is because Parallel FW guarantees a $O(\frac{1}{k})$ convergence rate for these problems. As a result, though subsampling reduces the cost of computation per iteration, the increase in number of iterations negates this advantage. In fact, when p is in an ultra-low regime, e.g., $p = 0.0005$, Sampled FW converges extremely slowly for *all problems*. Interestingly, Smoothened FW performs better in this case, ameliorating the performance deterioration. This is most evident in Figures 5d and 5c, where t_ϵ for Convex Approximation and AdaBoost is considerably smaller under Smoothened FW.

8.5. LASSO Experiment

To show the performance of our algorithm on the cases beyond simplex constrained problems, we solve the LASSO problem (21). We compare our distributed FW with distributed ADMM.

The input data is synthetic and with $N = 100,000$ and $d = 1000$. First, we solve the following problem:

$$\min_{\theta} \ \frac{1}{2}\|X^\top \theta - p\|_2^2 + \|\theta\|_1,$$

with distributed ADMM using 400 cores and for different values of ρ, which is a parameter controlling convergence (see Section 8.3 of (Boyd et al. 2011)). We then solve the LASSO with our Distributed FW algorithm, setting K equal to the ℓ_1 norm of the solution obtained by ADMM. For a fair comparison, we use 400 cores. Fig. 6 shows the value of the squared loss $\frac{1}{2}\|X\theta - p\|_2^2$ as a function of time for FW and ADMM. As we see, FW outperforms ADMM.

9. Conclusion

We establish structural conditions under which FW admits a highly scalable parallel implementation via map-reduce. FW has found recent applications in non-convex optimization (Reddi et al. 2016), and a variant has been applied to combinatorial optimization (Calinescu et al. 2011, Bian et al. 2017); exploring the applicability of our approach in these areas is an important open problem.

Acknowledgements. We kindly thank our reviewers for their very useful comments and suggestions. The work was supported by National Science Foundation (NSF) CAREER grant CCF-1750539.

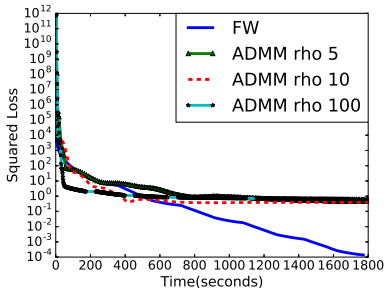


Fig. 6. Comparison between ADMM and our distributed Frank-Wolfe algorithm. Each algorithm uses 400 cores.

References

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M. et al. (2016), Tensorflow: A system for large-scale machine learning, *in* 'OSDI'.

Bahmani, B., Moseley, B., Vattani, A., Kumar, R. & Vassilvitskii, S. (2012), 'Scalable k-means++', *VLDB* .

Beck, A. & Shtern, S. (2015), 'Linearly convergent away-step conditional gradient for non-strongly convex functions', *Mathematical Programming* pp. 1–27.

Bellet, A., Liang, Y., Garakani, A. B., Balcan, M.-F. & Sha, F. (2015), A Distributed Frank-Wolfe Algorithm for Communication-Efficient Sparse Learning, *in* 'SDM'.

Bertsekas, D. P. (1999), *Nonlinear programming*, Athena scientific, Belmont, MA, USA.

Bialecki, A., Cafarella, M., Cutting, D. & O?malley, O. (2005), 'Hadoop: a framework for running applications on large clusters built of commodity hardware'.

Bian, Y., Mirzasoleiman, B., Buhmann, J. M. & Krause, A. (2017), Guaranteed non-convex optimization: Submodular maximization over continuous domains, *in* 'AISTATS'.

Boyd, S., Parikh, N., Chu, E., Peleato, B. & Eckstein, J. (2011), 'Distributed optimization and statistical learning via the alternating direction method of multipliers', *Foundations and Trends in Machine Learning* **3**(1), 1–122.

Boyd, S. & Vandenberghe, L. (2004), *Convex Optimization*, Cambridge University Press, New York, NY, USA.

Calinescu, G., Chekuri, C., Pál, M. & Vondrák, J. (2011), 'Maximizing a monotone submodular function subject to a matroid constraint', *SIAM Journal on Computing* **40**(6), 1740–1766.

Canon, M. & Cullum, C. (1968), 'A tight upper bound on the rate of convergence of frank-wolfe algorithm', *SIAM Journal on Control* **6**(4), 509–516.

Chandrasekaran, V., Recht, B., Parrilo, P. A. & Willsky, A. S. (2012), 'The convex geometry of linear inverse problems', *Foundations of Computational Mathematics* **12**(6), 805–849.

Chen, S. & Banerjee, A. (2015), Structured estimation with atomic norms: General bounds and applications, *in* 'NIPS'.

Chu, C.-T., Kim, S. K., Lin, Y.-A., Yu, Y., Bradski, G., Ng, A. Y. & Olukotun, K. (2006), Map-reduce for machine learning on multicore, *in* 'NIPS'.

Clarkson, K. L. (2010), 'Coresets, sparse greedy approximation, and the Frank-Wolfe algorithm', *ACM Trans. Algorithms* **6**(4), 63:1–63:30.

Dean, J. & Ghemawat, S. (2008), 'Mapreduce: simplified data processing on large clusters', *Communications of the ACM* **51**(1), 107–113.

Dudik, M., Harchaoui, Z. & Malick, J. (2012), Lifted coordinate descent for learning with trace-norm regularization, *in* 'AISTATS'.

Dunn, J. C. (1979), 'Rates of convergence for conditional gradient algorithms near singular and nonsingular extremals', *SIAM Journal on Control and Optimization* **17**(2), 187–211.

Frank, M. & Wolfe, P. (1956), 'An Algorithm for Quadratic Programming', *Naval Research Logistics Quarterly* **3**(1-2), 95–110.

Garber, D. & Hazan, E. (2016), 'A linearly convergent variant of the conditional gradient algorithm under strong convexity, with applications to online and stochastic optimization', *SIAM Journal on Optimization* **26**(3), 1493–1528.

Garber, D. & Meshi, O. (2016), Linear-memory and decomposition-invariant linearly convergent conditional gradient algorithm for structured polytopes, *in* 'Advances in Neural Information Processing Systems', pp. 1001–1009.

Guélat, J. & Marcotte, P. (1986), 'Some comments on Wolfe's away step', *Mathematical Programming* **35**(1), 110–119.

Harchaoui, Z., Douze, M., Paulin, M., Dudik, M. & Malick, J. (2012), Large-scale image classification with trace-norm regularization, *in* 'CVPR'.

Harper, F. M. & Konstan, J. A. (2015), 'The movielens datasets: History and context', *ACM Trans. Interact. Intell. Syst.* **5**(4), 19:1–19:19.

Hazan, E. & Kale, S. (2012), Projection-free online learning, *in* 'ICML'.

Hazan, E. & Luo, H. (2016), Variance-reduced and projection-free stochastic optimization, *in* 'ICML'.

Jaggi, M. (2013), Revisiting Frank-Wolfe: Projection-free sparse convex optimization., *in* 'ICML'.

Joulin, A., Tang, K. & Fei-Fei, L. (2014), Efficient image and video co-localization with Frank-Wolfe algorithm, *in* 'ECVV'.

Koren, Y., Bell, R. & Volinsky, C. (2009), 'Matrix factorization techniques for recommender systems', *Computer* **42**(8).

Kumar, R., Moseley, B., Vassilvitskii, S. & Vattani, A. (2015), 'Fast greedy algorithms in mapreduce and streaming', *ACM Transactions on Parallel Computing* **2**(3), 14.

Lacoste-Julien, S. & Jaggi, M. (2015), On the global linear convergence of Frank-Wolfe optimization variants, *in* 'NIPS'.

Lacoste-Julien, S., Jaggi, M., Schmidt, M. & Pletscher, P. (2013), Block-coordinate frank-wolfe optimization for structural svms, *in* 'Proceedings of ICML'.

Lan, G. & Zhou, Y. (2016), 'Conditional gradient sliding for convex optimization', *SIAM Journal on Optimization* **26**(2), 1379–1409.

Leighton, F. T. (2014), *Introduction to parallel algorithms and architectures: Trees Hypercubes*, Elsevier.

Li, M., Zhou, L., Yang, Z., Li, A., Xia, F., Andersen, D. G. & Smola, A. (2013), Parameter server for distributed machine learning, *in* 'NIPS Workshop'.

Lichman, M. (2013), 'UCI machine learning repository'.

Ng, A. Y. (2004), Feature selection, l 1 vs. l 2 regularization, and rotational invariance, *in* 'ICML'.

Osokin, A., Alayrac, J.-B., Lukasewitz, I., Dokania, P. K. & Lacoste-Julien, S. (2016), Minding the Gaps for Block Frank-Wolfe Optimization of Structured SVMs, *in* 'ICML'.

Recht, B., Re, C., Wright, S. & Niu, F. (2011), Hogwild: A lock-free approach to parallelizing stochastic gradient descent, *in* 'NIPS'.

Reddi, S. J., Sra, S., Póczós, B. & Smola, A. (2016), Stochastic Frank-Wolfe methods for non-convex optimization, *in* 'Allerton'.

Shah, P., Bhaskar, B. N., Tang, G. & Recht, B. (2012), Linear system identification via atomic norm regularization, *in* 'CDC'.

Sherman, J. & Morrison, W. J. (1950), 'Adjustment of an inverse matrix corresponding to a change in one element of a given matrix', *The Annals of Mathematical Statistics* **21**(1), 124–127.

Suri, S. & Vassilvitskii, S. (2011), Counting triangles and the curse of the last reducer, *in* 'WWW'.

Tewari, A., Ravikumar, P. K. & Dhillon, I. S. (2011), Greedy algorithms for structurally constrained high dimensional problems, *in* 'NIPS'.

Tibshirani, R. (1996), 'Regression shrinkage and selection via the lasso', *Journal of the Royal Statistical Society. Series B (Methodological)* pp. 267–288.

Tran, N. L., Peel, T. & Skhiri, S. (2015), Distributed frank-wolfe under pipelined stale synchronous parallelism, *in* '2015 IEEE International Conference on Big Data (Big Data)'.

Wang, Y.-X., Sadhanala, V., Dai, W., Neiswanger, W., Sra, S. & Xing, E. (2016), Parallel and distributed block-coordinate frank-wolfe algorithms, *in* 'Proceedings of ICML'.

Wolfe, P. (1970), 'Convergence theory in nonlinear programming', *Integer and nonlinear programming* pp. 1–36.

Yang, H.-c., Dasdan, A., Hsiao, R.-L. & Parker, D. S. (2007), Map-reduce-merge: simplified relational data processing on large clusters, *in* 'SIGMOD'.

Yang, T. (2013), Trading computation for communication: Distributed stochastic dual coordinate ascent, *in* 'NIPS'.

Ying, Y. & Li, P. (2012), 'Distance metric learning with eigenvalue optimization', *Journal of Machine Learning Research* **13**(Jan), 1–26.

Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S. & Stoica, I. (2010), Spark: cluster computing with working sets., *in* 'HotCloud'.

Zhang, L., Wang, G., Romero, D. & Giannakis, G. B. (2017), 'Randomized block frank-wolfe for convergent large-scale learning', *IEEE Transactions on Signal Processing* **65**(4), 6448–6461.

Zinkevich, M., Weimer, M., Li, L. & Smola, A. J. (2010), Parallelized stochastic gradient descent, *in* 'NIPS'.

Author Biographies

Armin Moharrer received a B.Sc. degree in Electrical Engineering from Amirkabir University of Technology (Tehran Polytechnic), Tehran, Iran, in 2015 and his M.Sc. degree in Electrical and Computer Engineering from Northeastern University, Boston, MA, in 2018. Since Jan 2016 he has been a Ph.D. student in the Electrical and Computer Engineering at Northeastern University under supervision of Prof. Stratis Ioannidis. His research focuses on distributed algorithms and data mining.

Stratis Ioannidis is an Assistant Professor in the Electrical and Computer Engineering department at Northeastern University, in Boston, MA, where he also holds a courtesy appointment with the College of Computer & Information Sciences. He received his B.Sc. (2002) in Electrical and Computer Engineering from the National Technical University of Athens, Greece, and his M.Sc. (2004) and Ph.D. (2009) in Computer Science from the University of Toronto, Canada. Prior to joining Northeastern, he was a research scientist at the Technicolor research centers in Paris, France, and Palo Alto, CA, as well as at Yahoo Labs in Sunnyvale, CA. He is the recipient of an NSF CAREER award, a Google Faculty Research Award, and a Best Paper Award at the 2017 ACM Conference on Information-centric Networking (ICN).

Correspondence and offprint requests to: Armin Moharrer, Department of Electrical & Computer Engineering, Northeastern University, Boston, MA 02155 USA. Email: amoharrer@ece.neu.edu