Distributing Frank-Wolfe via Map-Reduce

Armin Moharrer
Electrical and Computer Engineering
Northeastern University
Boston, MA
moharrer.a@husky.neu.edu

Stratis Ioannidis
Electrical and Computer Engineering
Northeastern University
Boston, MA
ioannidis@ece.neu.edu

***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 10 million variables using 350 cores in 44 minutes; the same operation takes 133 hours when executed serially.**

***Keywords*—Frank-Wolfe; Distributed Algorithms; Convex Optimization; Spark;**

I. Introduction

Map-reduce [**? ?**] 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 [**? ? ? ? ?**]. 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 [**? ? ?**].

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

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

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

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

is the N-dimensional simplex. Several important problems, including experimental design, training SVMs, Adaboost, and projection to a convex hull indeed take this form [**? ? ?**]. 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. II, 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 [**?**]. Indeed, as we discuss in Sec. III-B, FW assumes a very simple, elegant form under simplex constraints, and has important computational advantages [**? ? ?**]. 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 [**?**], 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 10 million variables in 44 minutes, an operation that would take 133 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. II, and introduce FW and the map-reduce framework in Sec. III. In Sec. IV, 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. V. We extend possible applications of our algorithm on constraint sets beyond the simplex in Sec. VI. Finally, in Sec. VII and VIII we describe our implementation and the results of our experiments over a Spark cluster.

II. Related Work

Frank-Wolfe (FW) [**?**] has attracted interest recently due to its numerous computational advantages [**? ? ? ? ? ?**]. 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 [**? ? ?**]. Frank and Wolfe [**?**] showed a convergence rate of $O(\frac{1}{\varepsilon})$ for smooth objectives, and Guélat and Marcotte [**?**] proved that a variant using so-called 'away-points' converges linearly when the objective is strongly convex, constraints form a polytope, and the solution lies in its interior. Several recently proposed FW variants attain linear convergence under weaker conditions [**? ? ? ?**]. 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.

Stochastic Gradient Descent (SGD) [**? ? ? ? ?**] 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 [**? ? ? ?**]. 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) [**?**] also solves problems with separable objectives by parallelizing their dual. The Alternating Directions Method of Multipliers (ADMM) [**?**] applies to both separable and non-separable objectives, including LASSO (c.f. Sec. VI). 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 [**? ? ?**]. We do not assume sum objectives or any sparsity conditions here.

More recently, and more relevant to our work, Bellet et al. [**?**] propose a distributed version of FW for objectives of the form $F(\theta) = g(A\theta)$, for some $A \in \mathbb{R}^{d\times N}$, 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. IV). The authors characterize the message and parallel complexity when A is partitioned across multiple processors under broadcast operations. We (a) consider a broader class of problems, that do not abide by the structure presumed by Bellet et al. (e.g., the two experimental design problems presented in Sec. V), 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 [**? ? ? ?**], using unbiased estimates of the gradient at each step. Hazan and Luo [**?**] improve upon earlier convergence rates [**? ?**] when the objective is smooth, strongly convex, or Lipschitz. Reddi et al. [**?**] 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. VIII.

III. Technical Preliminary

A. *Frank-Wolfe Algorithm*

The FW algorithm [**?**], summarized in Alg. 1, solves problems of the form:

$$\text{Minimize} \quad F(\theta) \tag{3a}$$
$$\text{subj. to:} \quad \theta \in \mathscr{D}, \tag{3b}$$

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

$$s^k = \arg\min_{s\in\mathscr{D}} \; s^\top \cdot \nabla F(\theta^k), \tag{4a}$$
$$\theta^{k+1} = (1-\gamma^k)\theta^k + \gamma^k s^k, \tag{4b}$$

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 $\theta^{k+1} \in \mathscr{D}$, as a convex combination of $\theta^k, s^k \in \mathscr{D}$; 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* [**?**]. The duality gap at feasible point $\theta^k \in \mathscr{D}$ in iteration $k \in \mathbb{N}$ is:

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

The convexity of F implies that $F(\theta^k) - F(\theta^*) \le g(\theta^k)$ for any optimal solution $\theta^* \in \arg\min_{\theta\in\mathscr{D}} F(\theta)$ [**?**]. 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 $\varepsilon > 0$.

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 [**? ?**]. In this case, both of the above step sizes imply that the k-th iteration of the Frank-Wolfe algorithm satisfies $F(\theta^k) - F(\theta^*) \le O(\frac{1}{k})$ [**?**]. For arbitrary convex objectives with unbounded curvature, FW still converges if the step size is set by the line minimization rule [**?**].

B. *Frank-Wolfe Over the Simplex*

We focus on FW for the special case where the feasible set $\mathscr{D}$ is the simplex D_0, given by (2). As described in Section V, this set of constraints arises in many problems, including training SVMs, convex approximation, Adaboost, and experimental design (see also [**?**]). Under this set of constraints, the linear optimization problem in (4a) has a simple solution: it reduces to finding the minimum element

Algorithm 1 FRANK-WOLFE

```
1: Pick $\theta^0 \in \mathscr{D}$
2: $k := 0$
3: repeat
4:     $s^k := \arg\min_{s \in \mathscr{D}} s^\top \cdot \nabla F(\theta^k)$
5:     gap $:= (\theta^k - s^k)^\top \nabla F(\theta^k)$
6:     $\theta^{k+1} := (1-\gamma^k)\theta^k + \gamma^k s^k$
7:     $k := k+1$
8: until gap $< \varepsilon$
```

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.

C. Map-Reduce Framework

Consider a data structure $D \in \mathscr{X}^N$ comprising N elements $d_i \in \mathscr{X}$, $i \in [N]$, for some domain $\mathscr{X}$. A `map` operation over D applies a function to every element of the data structure. That is, given $f : \mathscr{X} \to \mathscr{X}'$, the operation $D' = D$.`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 $\oplus : \mathscr{X} \times \mathscr{X} \to \mathscr{X}$ 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.`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 [?]. There exist several computational frameworks, including Hadoop [?] and Spark [?], 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.

IV. FRANK-WOLFE VIA MAP-REDUCE

A. Gradient Computation through Common Information

In this section, we identify two properties of function F under which FW over the simplex $\mathscr{D}_0$ 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 $\mathscr{D} = \mathscr{D}_0$. 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

Algorithm 2 Serial FW under Properties 1 and 2

```
1: Pick $\theta^0 \in \mathscr{D}$
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 $< \varepsilon$
```

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 V. 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:

$$\begin{aligned} &\text{ConvexApproximation} \\ &\text{Minimize} \quad F(\theta) = \|X^T\theta - p\|_2^2 \quad (10a) \\ &\text{subj. to:} \quad \theta \in \mathscr{D}_0, \quad (10b) \end{aligned}$$

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

$$h(X;\theta) = X^T\theta - p, \quad (11)$$

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 $N \gg 1$).

B. 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, 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 VIII, this leads to a significant speedup, allowing Alg. 2 to outperform interior-point methods even when executed serially.

C. 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_i^k), 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 implementation 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 $O(d)$ [**?**]. 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 $O(m+d)$.

D. 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

Algorithm 3 FW VIA MAP-REDUCE

```
1:  Pick $\theta^0 \in \mathscr{D}$
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:      $D' = D.\texttt{map}\big((i,x_i,\theta_i) \mapsto (i,x_i,\theta_i,G(h,x_i,\theta_i))\big)$
8:      $(i^*,x_{i^*},\theta_{i^*},z_{i^*}) := D'.\texttt{reduce}\left((i,x_i,\theta_i,z_i),(i',x_{i'},\theta_{i'},z_{i'}) \mapsto \begin{cases}(i,x_i,\theta_i,z_i) & \text{if } z_i < z_{i'}\\ (i',x_{i'},\theta_{i'},z_{i'}) & \text{if } z_i \geq z_{i'}\end{cases}\right)$
9:      $\text{gap} := D'.\texttt{map}\left((i,x_i,\theta_i,z_i) \mapsto \begin{cases}\theta_i \cdot z_i & \text{if } i \neq i^*\\ (\theta_i - 1)\cdot z_i & \text{if } i = i^*\end{cases}\right).\texttt{reduce}(+)$
10:     $D := D.\texttt{map}\left((i,x_i,\theta_i) \mapsto \begin{cases}(i,x_i,(1-\gamma^k)\theta_i) & \text{if } i \neq i^*\\ (i,x_i,(1-\gamma^k)\theta_i + \gamma^k) & \text{if } i = i^*\end{cases}\right)$
11:     $h := H(h, x_{i^*}, \theta_{i^*}, \gamma^k)$.
12:     $k := k+1$
13: until gap $< \varepsilon$
```

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

TABLE I: Examples of problems satisfying Prop. 1–3.

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, satisfy 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:

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

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^*}, $\theta_{i^*}^k$, 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 I, 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.

V. EXAMPLES

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

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 $y_i = \beta^\top x_i + \varepsilon_i$, 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 [?]:

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

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 $\texttt{f} : \mathbb{R}^{d\times d} \to \mathbb{R}$ lead to different optimality criteria; we review two below.

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

D-OPTIMALDESIGN

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

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

D-OPTIMALDESIGN satisfies Prop. 1 as:

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

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 $G(h,x) = -x^\top h x$. Hence, Prop. 1 holds when $d^2 \ll N$. Using the Sherman-Morrison formula [?] we can show that the common information at step $k+1$ is:

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

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:

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

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 $\texttt{f}$ is the trace:

A-OPTIMALDESIGN

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

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

The partial derivative of the F can be written as:

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

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

$$\text{Minimize} \quad F(\theta) = \log\left(\textstyle\sum_{j=1}^d \exp(-\alpha c_j(X,\theta) r_j)\right) \tag{19a}$$
$$\text{subj. to:} \quad \theta \in \mathscr{D}_0, \tag{19b}$$

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 $b_j = \frac{\alpha r_j \exp(-\alpha c_j r_j)}{\sum_{i=1}^d \exp(-\alpha c_j r_j)}$, $j \in [d]$. The common information, $h : \mathbb{R}^{N\times d} \times \mathbb{R}^N \to \mathbb{R}^d$ is $h(X;\theta) = [\exp -\alpha c_j r_j]_{j\in[d]}$, 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 I are convex, and some admit specialized solvers. A-OPTIMALDESIGN can be reduced to a semidefinite program, (see Sec. 7.5 of [**?**]), and solved as an SDP. ADABOOST can be expressed as a geometric program (GP) [**?**], 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. VIII we compare FW to the above specialized solvers, and we see that it outperforms them in all cases.

VI. 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 $\|\theta\|_1 \le K$ appears in many optimization problems as means of enforcing sparsity [**? ?**]. 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 [**?**]:

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

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

$$\min_{\theta:\|\theta\|_{\mathscr{A}}\le K} f(\theta),$$

where $\|x\|_{\mathscr{A}}$ denotes the *atomic norm*: given a set of atoms $\mathscr{A} = \{a_i \in \mathbb{R}^N\}$ the atomic norm is defined as $\|x\|_{\mathscr{A}} = \inf\{t \mid t \ge 0, t \in \mathscr{C}_{\mathscr{A}}\}$, where $\mathscr{C}_{\mathscr{A}}$ 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 [**? ? ? ?**]. Tewari et el. [**?**] propose an FW-like algorithm for this class of problems. In this algorithm, the step 4 of Alg. 1 is replaced by

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

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 $\mathscr{A}$ 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.

VII. IMPLEMENTATION

We implemented Alg. 3 over Spark, an open-source cluster-computing framework [**?**]. 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 [**?**].

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 I and used it in our experiments.

VIII. Experiments

A. Experiment Setup

Cluster. Our cluster comprises 8 worker machines, each with 56 Intel Xeon 2.6GHz CPU cores and 512GB of RAM, at a total capacity of 448 cores and 4TB of RAM. We deploy Spark over this cluster in standalone mode.

Algorithms. We solve Convex Approximation, Adaboost, D-Optimal Design, and A-Optimal Design summarized in Table I, as well as LASSO (c.f. Sec. VI). 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] We implement the distributed ADMM for LASSO problem, as described in Section 8.3 of [**?**]. We also implement our parallel algorithm (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 also introduce two stochastic parallel variants that subsample the gradient; we discuss these in Section VIII-D.

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 $p = X^\top\theta^* + \varepsilon$, where $\varepsilon \in \mathbb{R}^d$ 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 II–IV.

Real Data. We also experiment with 3 real datasets, summarized in Table V. The first dataset is Movielens [**?**]. 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 [**?**]. 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 [**?**], which comprises 515345 songs with 90 features. We represent it as a matrix with $N = 515345$ and $d = 90$.

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 $F(\theta) - g(\theta) \le F(\theta^*)$ instead. More formally:

$$t_\varepsilon = \min\left\{t : \tfrac{F(\theta(t))}{F(\theta(t))-g(\theta(t))} \le 1+\varepsilon\right\}, \tag{23}$$

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

B. Serial Execution

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

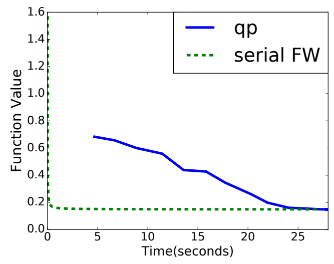
In each execution, we keep track of the objective function F as a function of time elapsed. Unlike FW, the interior-point methods do not generate feasible solutions at each iteration. Therefore, we project the solutions at each iteration on the feasible set, and compute the objective F on the projected 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 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.

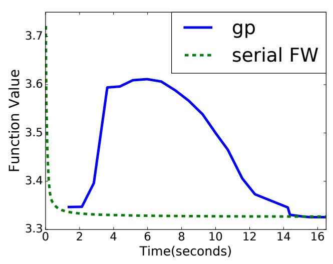
C. Effect of Parallelism

To study parallelism, we first show results for two large-scale synthetic datasets: Dataset B, a dataset with $N = 10^7$ and $d = 100$ (Table III), and Dataset C with N ranging between

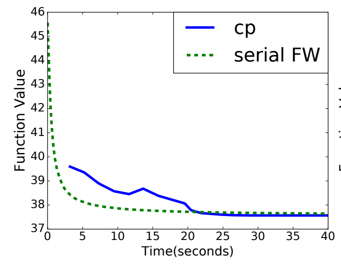
[1]https://github.com/neu-spiral/FrankWolfe
[2]cvxopt.org



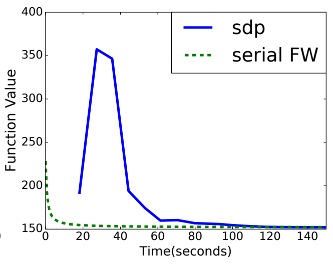
(a) ConvexApproximation



(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.

Problems	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 II: Dataset A

Problem	N	d	ε
Conv. Approx.	10M	100	0.12
Adaboost	10M	100	0.07
D-opt. Design	10M	100	0.12
A-opt. Design	10M	100	0.25

TABLE III: Dataset B

Problems	N	d	ε
Conv. Approx.	220000	500	0.01
Adaboost	190000	500	0.001
D-opt. Design	110000	500	0.05
A-opt. Design	110000	500	0.14

TABLE IV: Dataset C

Dataset	N	d	ε
Movielens	137768	500	0.18
HEPMASS	1M	38	0.04
MSD	515345	90	0.01

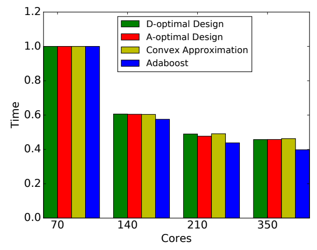
TABLE V: Real Datasets

Problem	Dataset	Speedup	# of cores
Conv. Approx.	Dataset C	35	64
Conv. Approx.	Dataset B	97	210
Adaboost	Dataset C	31	64
Adaboost	Dataset B	138	210
D-opt. Design	Dataset C	30	64
D-opt. Design	Dataset B	110	210
D-opt. Design	HEPMASS	35	64
D-opt. Design	Movielens	33	64
D-opt. Design	MSD	35	64
A-opt. Design	Dataset C	30	64
A-opt. Design	Dataset B	117	210

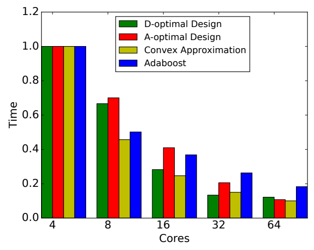
TABLE VI: A summary of speedups (over serial implementation) obtained by parallel FW for each problem and dataset, along with level of parallelism. Beyond this number of cores, no significant speedup improvement is observed.

100K and 220K and $d = 500$ (Table IV). 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 $P = 4$, respectively. Figure 3 shows objective F, as a function of time for different levels of parallelism.

The speedup of Parallel FW execution time over Serial FW is shown in Table VI. Both figures and the table show that increasing parallelism leads to significant speedups. For example, using 350 compute cores, we can solve the 10M-variable instance of D-optimal Design in 44 minutes, an operation that would take 133 hours when executed serially. For the input sizes used in these experiments, the benefit of parallelism saturates beyond 210 cores and 64 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. 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 IV, FW for these problems does not have a $O(\frac{1}{k})$ convergence rate.



(a) Dataset B



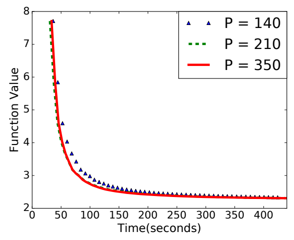
(b) Dataset C

Fig. 2: The t_ε as a function of the level of parallelism, measured in terms of cores P. Fig. 2a shows results on the 10M variable dataset (Table III) while Fig. 2b shows results on the dataset with $d = 500$ (Table IV). We normalize t_ε by its value at the lowest level of parallelism (7381s, 17502s, 571s, and 1199s, respectively, for each of the four problems in Fig. 2a and 3488s, 2433s, 2821s, and 1333s, respectively, in Fig. 2b). We see that increasing the level of parallelism speeds up convergence.

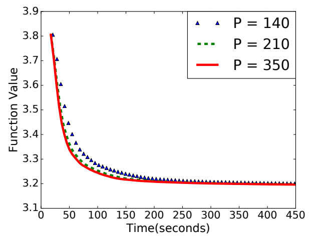
Next, we move on to experiments on the real datasets, summarized in Table V. For brevity, we only report D-Optimal Design for these datasets. Fig. 4 shows the measured t_ε for different levels of parallelism. For each dataset, t_ε is normalized by the value of t_ε for 8 cores. Again, we see that we gain a significant speedup by parallelism.

D. Subsampling the Gradient

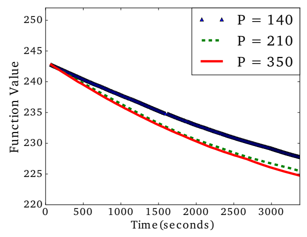
In this section, we study the effect of subsampling the gradient on the performance of FW. We have seen that parallelism reduces the cost of computation of the gradients. An alternative is to compute the gradient stochastically by subsampling only a few partial derivatives and using the minimal in this sub-sampled set. This reduces the amount of computation occurring in each iteration. Moreover, such a stochastic estimation of the gradient still guarantees convergence [**?**], 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.



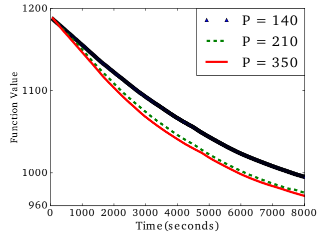
(a) CONVEXAPPROXIMATION



(b) ADABOOST



(c) D-OPTIMALDESIGN



(d) A-OPTIMALDESIGN

Fig. 3: The 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.

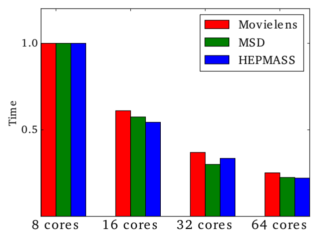


Fig. 4: The summary of parallelism experiments on the real datasets. For each dataset the t_ε is divided by the t_ε for 8 cores, which are 15247(s), 3899(s), and 4766(s) for Movielens, MSD, and HEPMASS, respectively.

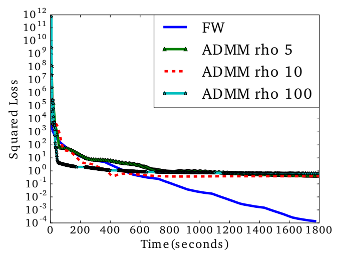
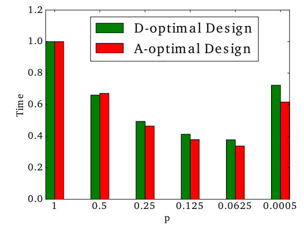


Fig. 5: The comparison between ADMM and our distributed Frank-Wolfe algorithm. Each algorithm uses 400 cores.

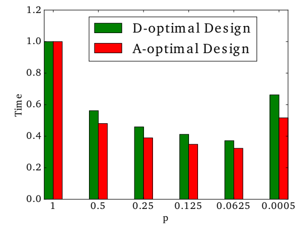
We consider two variants of subsampling. In Sampled FW, we compute each partial derivative $\frac{\partial F}{\partial \theta_i}$ with probability p. Then, we find the minimum among the computed partial derivatives. Note that this speeds derivative computations: at most $p \cdot N$ partial derivatives are computed, in expectation. In Smoothened 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 IV) in this experiment: we solve the corresponding problems using Sampled FW and Smoothened FW on 4 cores. The results are shown in Fig. 6. Values t_ε are normalized by t_ε for $p = 1$. This makes experiments in Figures 6 and 2b comparable: each core computes the same number of partial derivatives in expectation.

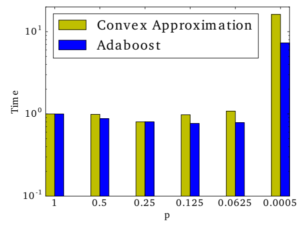
By comparing Figures 6 and 2b, we see that subsampling



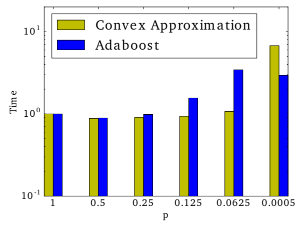
(a) Sampled FW for D-OPTIMALDESIGN and A-OPTIMALDESIGN



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



(c) Sampled FW for ADABOOST and CONVEXAPPROXIMATION



(d) Smoothened FW for ADABOOST and CONVEXAPPROXIMATION

Fig. 6: The measured t_ε under Sampled and Smoothened FW, over Dataset C. We normalize t_ε by the measured t_ε for 4 cores, which is reported in Fig. 2. By comparing Figures 6a and 6c 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.

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 6d and 6c, where t_ε for Convex Approximation and AdaBoost is considerably smaller under Smoothened FW.

E. 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 [**?**]). 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. 5 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.

IX. Conclusion

We establishe structural conditions under which FW admits a highly scalable parallel implementation via map-reduce. FW has found recent applications in non-convex optimization [**?**], and a variant has been applied to combinatorial optimization [**? ?**]; exploring the applicability of our approach in these areas is an important open problem.