Distributing Frank-Wolfe via Map-Reduce*

Armin Moharrer and Stratis Ioannidis
Northeastern University
amoharrer@ece.neu.edu, ioannidis@ece.neu.edu

Abstract

We identify structural properties under which a convex optimization over the simplex can be massively parallelized via map-reduce operations using the Frank-Wolfe (FW) algorithm. A broad class of problems, e.g., Convex Approximation, Experimental Designs, and Adaboost, can be tackled this way. We implement FW over Apache Spark, and solve problems with 20 million variables using 350 cores in 79 minutes; the same operation takes 165 hours when executed serially.

1 Introduction

Map-reduce [Dean and 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. Expressing algorithms in map-reduce also allows fast deployment at a massive scale: these algorithms can be quickly implemented and distributed on a commercial cluster via existing programming frameworks [Dean and Ghemawat, 2008; Bialecki *et al.*, 2005; Zaharia *et al.*, 2010].

In this work, we study optimization problems of the form $\min_{\theta\in\mathscr{D}_0} F(\theta)$, where $F:\mathbb{R}^N\to\mathbb{R}$ is a convex, differentiable function, and

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

is the N-dimensional simplex. Several important problems in AI and machine learning, including experimental design, training Support Vector Machines (SVM), Adaboost, and projection to a convex hull indeed take this form [Clarkson, 2010; Bellet *et al.*, 2015; Boyd and 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 decomposed into a sum of several differential functions.

It is well known [Clarkson, 2010; Jaggi, 2013] that problems with simplex constraints admit an efficient implementation through the Frank-Wolfe (FW) algorithm, also known as the conditional gradient algorithm [Frank and Wolfe, 1956]. Our main contribution is to identify and formalize a set of conditions under which solving such problems through FW *admits a massively parallel implementation via map-reduce*. We show that several important optimization problems, including experimental design, Adaboost, and projection to a convex hull satisfy the aforementioned conditions. 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. Finally, 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 using 350 cores in 79 minutes, an operation that would take 165 hours when executed serially.

Related Work. Frank-Wolfe [Frank and Wolfe, 1956] and its variants have attracted interest recently due to the algorithm's numerous computational advantages [Dudik *et al.*, 2012; Hazan and Kale, 2012; Ying and Li, 2012; Joulin *et al.*, 2014; Clarkson, 2010; Jaggi, 2013]. [Bellet *et al.*, 2015] 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 $d \ll N$. [Tran *et al.*, 2015] extend the distributed algorithm of [Bellet *et al.*, 2015] under an asynchronous computation model. We (a) consider a broader class of problems, that do not abide by the structure presumed by Bellet et al. or Tran et al. , and (b) establish properties under which FW can be explicitly parallelized through map-reduce rather than the message passing environment studied by Bellet et al. This allows us to leverage commercial map-reduce frameworks to readily implement and deploy parallel FW on a cluster.

2 Background

Frank-Wolfe Algorithm. The FW algorithm [Frank and Wolfe, 1956] solves problems of the form $\min_{\theta\in\mathscr{D}} F(\theta)$, where $F:\mathbb{R}^N\to\mathbb{R}$ is a convex function and $\mathscr{D}$ is a convex compact subset of $\mathbb{R}^N$. It starts from a feasible $\theta^0\in\mathscr{D}$ and proceeds as follows:

$$s^k = \arg\min_{s\in\mathscr{D}} s^\top \cdot \nabla F(\theta^k), \tag{2a}$$

$$\theta^{k+1} = (1-\gamma^k)\theta^k + \gamma^k s^k, \tag{2b}$$

*This extended abstract is an abridged version of an ICDM 2017 paper by the same authors.

for $k \in \mathbb{N}$, where $\gamma^k \in [0,1]$ is a step size. At each iteration $k \in \mathbb{N}$, FW finds a feasible point s^k minimizing the inner product with the current gradient $\nabla F(\theta^k)$, 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. Convergence is typically determined via the *duality gap* [Jaggi, 2013] at iteration k:

$$g(\theta^k) \equiv \max_{s \in \mathscr{D}} (\theta^k - s)^\top \nabla F(\theta^k) \overset{(2a)}{=} (\theta^k - s^k)^T \nabla F(\theta^k), \quad (3)$$

which upper bounds the objective value error at step k. The step size can be diminishing, e.g., $\gamma^k = \frac{2}{k+2}$, or set through line minimization, i.e.:

$$\gamma^k = \arg\min_{\gamma \in [0,1]} F\big((1-\gamma)\theta^k + \gamma s^k\big). \quad (4)$$

In both cases, convergence to an optimal solution occurs at an $O(\frac{1}{k})$ rate for objectives with bounded curvature [Frank and Wolfe, 1956; Jaggi, 2013].

Frank-Wolfe Over the Simplex. We focus on FW for the special case where the feasible set $\mathscr{D} = \mathscr{D}_0$, given by (1). Then, the linear optimization in (2a) 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 $\mathbb{R}^N$, (2a) reduces to:

$$s^k = e_{i^*}, \text{ where } i^* \in \arg\min_{i \in [N]} \frac{\partial F(\theta^k)}{\partial \theta_i}. \quad (5)$$

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 $D' = D.\mathtt{map}(f)$ 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, $D.\mathtt{reduce}(\oplus)$ 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.

3 Frank-Wolfe via Map-Reduce

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. Our approach exploits an additional structure often exhibited in practice: the objective F depends on the variables θ *as well as a dataset*, given as a problem input. We represent this dataset as 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 is large, i.e., $N \gg 1$, and X may be horizontally (i.e., row-wise) partitioned over multiple processors.

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, not depending on i: the latter abstracts the effect that θ and X have on $\frac{\partial F}{\partial \theta_i}$. Our second property asserts that this common information is *easy to update*: as variables θ^k are adapted according to the FW algorithm (2), 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* $x_i \in \mathbb{R}^d$, $i \in [N]$, *s.t. for all* $i \in [N]$: $\frac{\partial F(\theta)}{\partial \theta_i} = G(h(X;\theta), x_i, \theta_i)$, *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 $X \in \mathbb{R}^{N \times d}$ 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. Computing G is efficient, as its inputs have size $m, d \ll N$.

Recall from (2) and (5) 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:* $h(X;\theta^{k+1}) = H(h(X;\theta^k), x_{i^*}, \theta^k_{i^*}, \gamma^k)$, *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 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. As was the case for G, computing H is again efficient as it only depends on inputs of size $m, d \ll N$.

Algorithm 1 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:     Each processor computes the corresponding partial derivatives $z_i = G(h, x_i, \theta_i^k)$ via map.
8:     The processors find the minimum partial derivative $z_{i^*}$ and some corresponding information, i.e., $i^*, x_{i^*}$, and $\theta_{i^*}$ via reduce.
9:     Each processor computes the corresponding products $(\theta_i^k - e_{i^*})^\top z_i$, then the duality gap (3) is computed by summing up the products via reduce.
10:    Each processor updates their corresponding variables $\theta_i$, based on (2b), via map.
11:    $h := H(h, x_{i^*}, \theta_{i^*}, \gamma^k)$.
12:    $k := k+1$
13: until gap $< \varepsilon$
```

Parallelization Through Map-Reduce. We now outline how to parallelize the Frank-Wolfe algorithm through map-reduce operations. The algorithm is summarized in Alg. 1. 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. 1, 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 over the P processors. In the main loop, the master processor broadcasts the common information to the processors. Then, 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. 1). 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). The master processor broadcasts the step-size γ^k and i^* to the processors: then, 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).

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 (4) is used. Nevertheless, all problems we consider (listed in Table 1) satisfy an additional property that ensures that (4) can also be computed efficiently in a centralized fashion:

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

Prop. 3 implies that line minimization (4) at iteration k is:

$$\gamma^k = \arg\min_{\gamma\in[0,1]} \hat{F}\left(h(X;(1-\gamma)\theta^k + \gamma e_{i^*})\right). \qquad (6)$$

The argument of $\hat{F}$ is the updated common information h^{k+1} under step size γ. Hence, using Prop. 2, Eq. (6) becomes:

$$\gamma^k = \arg\min_{\gamma\in[0,1]} \hat{F}\left(H(h, x_{i^*}, \theta_{i^*}^k, \gamma)\right), \qquad (7)$$

where h is the present common information. As F is convex in θ^k, it is also convex in γ, so this is also a convex optimization problem. Crucially, (7) now 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 (7). In fact, for several of the problems we consider here, line minimization via (7) has a closed form solution. 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.

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	$\mathrm{trace}\left(A^{-1}(\theta)\right)$	$2d^2$	$O(d^2)$	$O(d^2)$

Table 1: Examples of problems satisfying Prop. 1–3.

Examples. We list here several machine learning and data mining problems that satisfy Prop. 1, 2, and 3. They are summarized in Table 1. The columns of the table indicate the objective function, the size of the common information m as a function of the dimension d, as well as the complexity of the respective gradient function G and the common information update function H. For further details refer to [Moharrer and Ioannidis, 2017].

Convex Approximation: In Convex Approximation, a point $p \in \mathbb{R}^d$ and N points $x_i \in \mathbb{R}^d, i \in [N]$, are given, and the goal is to project p on the convex hull of x_i s. In other words, Convex Approximation *approximates* p by a convex combination of the points $x_i, i \in [N]$.

Experimental Design: In this problem, 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. The quantity $A(X;\theta) = \sum_{i=1}^N \theta_i x_i x_i^\top$ is referred as the *design matrix*, where $\theta_i \in \mathbb{R}$ is the portion of experiments conducted by the learner with feature x_i. For brevity, we denote the design matrix by $A(\theta)$. Experimental design minimizes a function of the design matrix, i.e., $\mathtt{f}(A(\theta))$ for some $\mathtt{f} : \mathbb{R}^{d\times d} \to \mathbb{R}$. Choosing different functions result in different experimental designs. In D-optimal Design, `f` is the log-determinant and in A-optimal-Design it is the trace. They both satisfy the stated properties.

Adaboost: In Adaboost, 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.

Extensions. Our proposed distributed Frank-Wolfe algorithm can be extended to a more general class of problems, with constraints beyond the simplex. In particular, it can be applied to problems with ℓ_1-norm constraint set and some atomic-norm constraint sets. For a detailed discussion refer to Section 6 in [Moharrer and Ioannidis, 2017].

Problem	N	d	ε
Conv. Approx.	20M	100	0.02
Adaboost	20M	100	0.002
D-opt. Design	20M	100	0.09
A-opt. Design	20M	100	0.19

Table 2: Dataset B

Problem	N	d	ε
Conv. Approx.	100000	1000	0.12
Adaboost	100000	1000	0.004
D-opt. Design	100000	1000	0.03
A-opt. Design	100000	1000	0.09

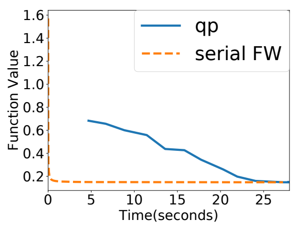
Table 3: Dataset C

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

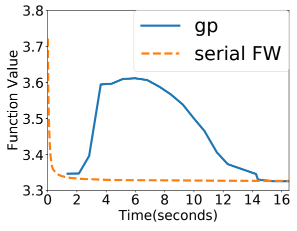
Table 4: Real Datasets

Problem	Dataset	Speedup	# of cores
Conv. Approx.	Dataset C	42	128
Conv. Approx.	Dataset B	98	350
Conv. Approx.	YAHOO	78	210
Adaboost	Dataset C	45	128
Adaboost	Dataset B	133	350
D-opt. Design	Dataset C	48	128
D-opt. Design	Dataset B	126	350
D-opt. Design	HEPMASS	35	64
D-opt. Design	Movielens	33	64
D-opt. Design	MSD	35	64
D-opt. Design	YAHOO	93	210
A-opt. Design	Dataset C	49	128
A-opt. Design	Dataset B	102	350
A-opt. Design	YAHOO	90	210

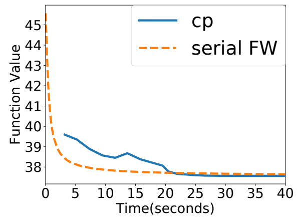
Table 5: Summary of speedups (over serial implementation) obtained by parallel FW for each problem and dataset, along with the level of parallelism. Beyond this number of cores, no significant speedup improvement is observed.



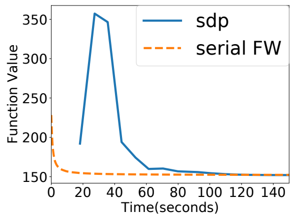
(a) CONVEXAPPROXIMATION



(b) ADABOOST



(c) D-OPTIMALDESIGN



(d) A-OPTIMALDESIGN

Figure 1: Values of the objective function generated by the algorithms as a function of time over a synthetic dataset with $N = 5000$ points. We see that Serial FW converges faster than interior point methods.

4 Experiments

Implementation. We implemented Alg. 1 over Spark, an open-source cluster-computing framework [Zaharia *et al.*, 2010]. Our FW implementation, which is publicly available,[1] is generic and relies 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. 1 in its entirety, and distributes its execution over a Spark cluster.

Algorithms. We solve Convex Approximation, Adaboost, D-Optimal Design, and A-Optimal Design summarized in Table 1. We implement both serial and parallel solvers. First, we implement FW on a single processor (Serial FW) in Python, setting γ using the line minimization rule (4). In addition, we solve Convex Approximation, D-Optimal Design, A-Optimal Design, and Adaboost using CVXOPT[2] solvers, `qp`, `cp`, `sdp`, and `gp`, respectively. We also implement our parallel algorithm (Alg. 1) using our Spark implementation. We set the step size using the line minimization rule (4). We refer to this algorithm as Parallel FW.

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. In our experiments the level of parallelism is measured in terms of the number of worker cores P, ranging from 1 to 350.

Serial Execution. We compare the Serial FW algorithm with the specialized interior point solvers (i.e., `cp`, `qp`, `sdp`, and `gp`) for each of the problems in Table 1. 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. 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.

Effect of Parallelism. We use two large synthetic datasets to construct X: Dataset B, a dataset with $N = 20\text{M}$ and $d = 100$ (Table 2), and Dataset C with $N = 100\text{K}$ and $d = 1\text{K}$ (Table 3). We also use the real datasets (Table 4). Fig. 2 and Fig. 3 show the convergence time t_ε as a function of the level of parallelism, measured in terms of the number of cores P. We normalize t_ε by its value at the lowest level of parallelism. The speedup of Parallel FW execution time over Serial FW is shown in Table 5. Both figures and the table show that 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, an operation that would take 165 hours when executed serially.

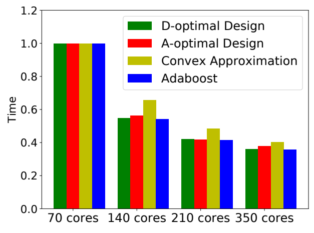
5 Conclusions

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

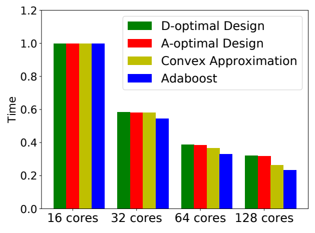
Acknowledgments

Research presented here was generously supported by NSF grants CCF-1750539 and IIS-1741197.

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

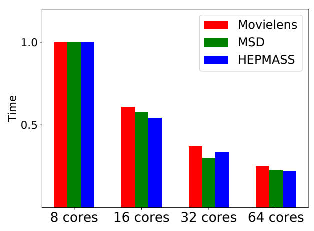


(a) Dataset B

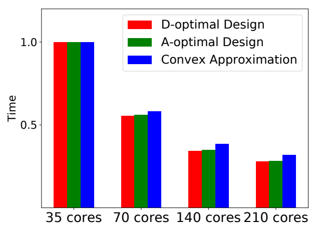


(b) Dataset C

Figure 2: Convergence time t_ε as a function of the level of parallelism, measured in terms of cores P. We normalize t_ε by its value at the lowest level of parallelism (13134s, 27420s, 6573s, and 4447s, respectively, for each of the four problems in Fig. 2a and 487s, 486s, 62s, and 483s, respectively, in Fig. 2b).



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



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

Figure 3: Convergence time on 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. 3a, and 9888s, 7060s, and 1302s for D-optimal Design, A-optimal Design, and Convex Approximation, respectively, in Fig 3b.

References

[Bellet *et al.*, 2015] Aurélien Bellet, Yingyu Liang, Alireza Bagheri Garakani, Maria-Florina Balcan, and Fei Sha. A Distributed Frank-Wolfe Algorithm for Communication-Efficient Sparse Learning. In *SDM*, 2015.

[Bialecki *et al.*, 2005] Andrzej Bialecki, Michael Cafarella, Doug Cutting, and Owen O?malley. Hadoop: a framework for running applications on large clusters built of commodity hardware, 2005.

[Bian *et al.*, 2017] Yatao Bian, Baharan Mirzasoleiman, Joachim M. Buhmann, and Andreas Krause. Guaranteed non-convex optimization: Submodular maximization over continuous domains. In *AISTATS*, 2017.

[Boyd and Vandenberghe, 2004] Stephen Boyd and Lieven Vandenberghe. *Convex Optimization*. Cambridge University Press, New York, NY, USA, 2004.

[Calinescu *et al.*, 2011] Gruia Calinescu, Chandra Chekuri, Martin Pál, and Jan Vondrák. Maximizing a monotone submodular function subject to a matroid constraint. *SIAM Journal on Computing*, 40(6):1740–1766, 2011.

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

[Dean and Ghemawat, 2008] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[Dudik *et al.*, 2012] Miro Dudik, Zaid Harchaoui, and Jérôme Malick. Lifted coordinate descent for learning with trace-norm regularization. In *AISTATS*, 2012.

[Frank and Wolfe, 1956] Marguerite Frank and Philip Wolfe. An Algorithm for Quadratic Programming. *Naval Research Logistics Quarterly*, 3(1-2):95–110, 1956.

[Hazan and Kale, 2012] Elad Hazan and Satyen Kale. Projection-free online learning. In *ICML*, 2012.

[Jaggi, 2013] Martin Jaggi. Revisiting Frank-Wolfe: Projection-free sparse convex optimization. In *ICML*, 2013.

[Joulin *et al.*, 2014] Armand Joulin, Kevin Tang, and Li Fei-Fei. Efficient image and video co-localization with Frank-Wolfe algorithm. In *ECVV*, 2014.

[Leighton, 2014] F Thomson Leighton. *Introduction to parallel algorithms and architectures: Trees Hypercubes*. Elsevier, 2014.

[Moharrer and Ioannidis, 2017] Armin Moharrer and Stratis Ioannidis. Distributing frank-wolfe via map-reduce. In *ICDM*, 2017.

[Reddi *et al.*, 2016] Sashank J Reddi, Suvrit Sra, Barnabás Póczós, and Alex Smola. Stochastic Frank-Wolfe methods for non-convex optimization. In *Allerton*, 2016.

[Tran *et al.*, 2015] N. L. Tran, T. Peel, and S. Skhiri. Distributed frank-wolfe under pipelined stale synchronous parallelism. In *2015 IEEE International Conference on Big Data (Big Data)*, 2015.

[Ying and Li, 2012] Yiming Ying and Peng Li. Distance metric learning with eigenvalue optimization. *Journal of Machine Learning Research*, 13(Jan):1–26, 2012.

[Zaharia *et al.*, 2010] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. In *HotCloud*, 2010.