Distributing Frank-Wolfe via Map-Reduce

A Thesis Presented
by

Armin Moharrer

to

The Department of Electrical and Computer Engineering

in partial fulfillment of the requirements
for the degree of

Master of Science

in

Electrical and Computer Engineering

Northeastern University
Boston, Massachusetts

May 2018

Contents

List of Figures iii

List of Tables iv

List of Acronyms v

Acknowledgments vi

Abstract of the Thesis vii

1 Introduction 1
- 1.1 Motivation . . . 1
- 1.2 Distributing Frank-Wolfe . . . 2
- 1.3 Our Contributions . . . 4

2 Technical Preliminary and Related Work 6
- 2.1 Convex Optimization . . . 6
- 2.2 Distributed Methods for Convex Optimization . . . 8
  - 2.2.1 SGD . . . 8
  - 2.2.2 ADMM . . . 9
  - 2.2.3 DSDCA . . . 11
- 2.3 Frank-Wolfe . . . 12
  - 2.3.1 Frank-Wolfe Algorithm . . . 13
  - 2.3.2 FW Variants . . . 15
  - 2.3.3 Distributed implementations . . . 18
  - 2.3.4 Frank-Wolfe Over the Simplex . . . 19
- 2.4 Distributed Frameworks . . . 19
  - 2.4.1 Map-Reduce Framework . . . 19
  - 2.4.2 Message Passing Interface (MPI) . . . 20

3 Frank-Wolfe via Map-Reduce 22
- 3.1 Gradient Computation through Common Information . . . 22
- 3.2 A Serial Algorithm . . . 24
- 3.3 Parallelization Through Map-Reduce . . . 25

3.4 Selecting the step size. 26

4 Examples and Extensions **28**

4.1 Examples 28

4.2 Extensions 32

5 Experimental Study **34**

5.1 Implementation 34

5.2 Experiments 34

5.2.1 Experiment Setup 34

5.2.2 Serial Execution 36

5.2.3 Effect of Parallelism 37

5.2.4 Subsampling the Gradient 39

5.2.5 LASSO Experiment 41

6 Conclusion **44**

Bibliography **46**

List of Figures

5.1 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. 37

5.2 The t_ϵ as a function of the level of parallelism, measured in terms of cores P. Fig. 5.2a shows results on the 20M variable dataset (Table 5.2) while Fig. 5.2b shows results on the dataset with $d = 1000$ (Table 5.3). 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. 5.2a and 487s, 486s, 62s, and 483s, respectively, in Fig. 5.2b). We see that increasing the level of parallelism speeds up convergence. . 39

5.3 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 5.3a and 5.3b with Figures 5.3c and 5.3d, we see that FW for D-Optimal Design and A-Optimal Design converges slower. 40

5.4 The 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. 5.4a, and 9888s, 7060s, and 1302s for D-optimal Design, A-optimal Design, and Convex Approximation, respectively, in Fig 5.4b. 40

5.5 The 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. 5.2. By comparing Figures 5.5a and 5.5c with Fig. 5.2b, 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. 42

5.6 The comparison between ADMM and our distributed Frank-Wolfe algorithm. Each algorithm uses 400 cores. 43

List of Tables

4.1 Examples of problems satisfying Prop. 1–3. 28

5.1 Dataset A 38
5.2 Dataset B 38
5.3 Dataset C 38
5.4 Real Datasets 38
5.5 A 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. 38

List of Acronyms

FW the Frank-Wolfe algorithm. The optimization algorithm explained in Section 2.3.

ADMM Alternating Directions Method of Multipliers. An optimization algorithm for convex optimization problems (see Section 2.2.2).

SGD Stochastic Gradient Descent. A stochastic optimization algorithm for convex optimization problems, well-suited for problems with separable objectives (see Section 2.2.1).

DSDCA Distributed Stochastic Dual Coordinate Ascent. A distributed stochastic optimization algorithm for convex optimization problems, which solves problems in their dual domain (see Section 2.2.3).

MPI Message Passing Interface. A distributed framework for expressing parallel algorithms.

Acknowledgments

Here I wish to thank my advisor, Dr. Ioannidis, for generously helping me patiently in both developing and presenting the current work. Also, I wish to thank Dr. David Kaeli and Dr. Jennifer Dy for serving as the committee members. Finally, I would like to express my gratitude towards my parents, who have always supported me unconditionally in my work.

Abstract of the Thesis

Distributing Frank-Wolfe via Map-Reduce

by

Armin Moharrer

Master of Science in Electrical and Computer Engineering

Northeastern University, May 2018

Dr. Stratis Ioannidis, Advisor

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 165 hours when executed serially.

Chapter 1

Introduction

1.1 Motivation

Many interesting problems in machine learning, such as logistic regression, linear and non-linear regression, training Support Vector Machines (SVM), and experimental design can be formulated as convex optimization problems. Moreover, as a result of drastic growth in the size of data, these optimization problems in practice are large-scale: often, the problem input size is in the order of terabytes and the number of variables is in the order of millions or billions. This has motivated the study of distributed optimization algorithms, implemented in distributed frameworks that leverage the massively parallel computational power of computer clusters.

Map-reduce [1, 2] is such a distributed framework used to massively parallelize computationally intensive tasks. In comparison to other distributed frameworks, e.g., MPI, it enjoys a wider 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 [3, 4, 5, 6, 7]. Moreover, programming in map-reduce is less cumbersome than in the MPI framework, which requires explicit specifications of communication types as well as of the processors that need to communicate. 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 [1, 2, 8].

There are several optimization techniques, e.g., Stochastic Gradient Descent (SGD) and The Alternating Directions Method of Multipliers (ADMM), that can be parallelized in map-reduce frameworks. SGD [9, 10, 11, 12, 7] 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 [9, 10, 11, 12]. SGD computes the contribution of different terms to the gradient in parallel, and adapts the present solution in a centralized fashion, often asynchronously. Distributed Stochastic Dual Coordinate Ascent (DSDCA) [13] also solves problems with separable objectives by parallelizing their dual.

ADMM [14] applies to both separable and non-separable objectives, including LASSO (c.f. Sec. 4.2). In general, the above methods do not readily generalize to the 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 [9, 11, 14]. We do not assume sum objectives or any sparsity conditions here.

Despite these advantages, there exist important optimization problems that cannot be easily parallelized with these methods. In this thesis, we focus on the so-called Frank-Wolfe algorithm (FW), also known as the conditional gradient algorithm [15], and on its parallelization. The Frank-Wolfe algoithm has attracted interest recently due to its numerous computational advantages [16, 17, 18, 19, 20, 21]. 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 [20, 22, 21]. Therefore, distributing the Frank-Wolfe algorithm is very useful as it can solve this wide set of problems. Most importantly, as described in Section 2.3, the Frank-Wolfe algorithm can be used to solve more general problems than the ones described above. For example, it does not require any assumption on separability or sparsity of the objective.

1.2 Distributing Frank-Wolfe

FW [15] is a convex optimization algorithm that solves the convex optimization problems of the form:

$$\begin{aligned} \text{Minimize} \quad & F(\theta) \\ \text{subj. to:} \quad & \theta \in \mathcal{D}, \end{aligned}$$

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:

$$\begin{aligned} s^k &= \arg\min_{s \in \mathcal{D}} \; s^\top \cdot \nabla F(\theta^k), \\ \theta^{k+1} &= (1-\gamma^k)\theta^k + \gamma^k s^k, \end{aligned}$$

for $k \in \mathbb{N}$, where $\gamma^k \in [0,1]$ is the step size. Basically, at each iteration it finds a feasible point $s^k \in \mathcal{D}$ that minimizes the first-order Taylor approximation of the function F around the current solution θ^k. This is an advantage of FW as it reduces the optimization of a general form convex function F to the optimization of a linear function, subject to the same constraint set. Then, it adapts the solution by finding a convex combination of the points $\theta^k, s^k \in \mathcal{D}$. As a result, it maintains the feasibility of the solution θ^k at all iterations $k \in \mathbb{N}$.

An effort to distribute FW is made by Bellet et al. in [22]. They 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 $d \ll N$. This function form is more general than the separable form assumption of the previous algorithms. Several examples fall in this class, including two we study here (convex approximation and Adaboost); intuitively, $A\theta$ serves as we call the *common information* in our framework (c.f. Prop. 1 in Sec. 3). 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. 4.1), 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.

More specifically, we focus on solving, via map-reduce, optimization problems of the form:

$$\min_{\theta \in \mathcal{D}_0} F(\theta), \tag{1.1}$$

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

$$\mathcal{D}_0 \equiv \left\{ \theta \in \mathbb{R}_+^N : \textstyle\sum_{i=1}^N \theta_i = 1 \right\} \tag{1.2}$$

is the N-dimensional simplex. Several important problems, including experimental design, training SVMs, Adaboost, and projection to a convex hull indeed take this form [20, 22, 23]. 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 detail in Sec. 2.2, this is precisely the regime in which (1.1) is hard to parallelize via, e.g., stochastic gradient descent.

It is well known that (1.1) admits an efficient implementation through the Frank-Wolfe (FW) algorithm [15]. Indeed, as we discuss in Sec. 2.3.4, FW assumes a very simple, elegant form under this simplex constraint, and has important computational advantages [20, 21, 22]. Our main contribution is to identify and formalize a set of conditions under which solving Problem (1.1) through FW *admits a massively parallel implementation via map-reduce*.

1.3 Our Contributions

We make the following contributions:

- 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 extend our results beyond the simplex. In particular, we show that our distributed algorithm can be applied to problems with atomic norm constraints, including, e.g., the popular ℓ_1-norm constraint se, also known as "lasso".
- We implement our distributed FW algorithm on Spark [8], 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 165 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 thesis is organized as follows. We briefly review the formal definition of convex optimization problems and related work, introduce FW and its variants, and introduce distributed frameworks in Chapter 2. In Chapter 3, 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 and possible extensions of applications of our algorithm on constraint sets beyond the simplex are given in Chapter 4. Then, in Chapter 5 we describe our implementation and the results of our experiments over a Spark cluster. Finally, we conclude this thesis in Chapter 6.

Chapter 2

Technical Preliminary and Related Work

The main focus of this thesis is distributing the Frank-Wolfe algorithm, which is a convex optimization method. We give the formal definitions of convex optimization and related concepts in Section 2.1. We then introduce several existing distributed convex optimization methods in Section 2.2, discussing their relative advantages and limitations. Next, we describe the Frank-Wolfe algorithm in detail in Section 2.3. We conclude this chapter in Section 2.4 by introducing map-reduce and MPI, two popular distributed frameworks.

2.1 Convex Optimization

Convex set. A set $\mathcal{D} \in \mathbb{R}^N$ is called *convex* if the line segment defined by any two points in the set lies within the set. Formally, for every $\theta_1, \theta_2 \in \mathcal{D}$ and $\alpha \in [0, 1]$ we have:

$$(1 - \alpha)\theta_1 + \alpha\theta_2 \in \mathcal{D}.$$

Given m points $\theta_1, \ldots, \theta_m \in \mathbb{R}^N$, a point $\theta \in \mathbb{R}^N$ is called a *convex combination* of $\theta_1, \ldots, \theta_m$ if there exist $\alpha_1, \ldots \alpha_m \in \mathbb{R}_{\geq 0}$, s.t., $\sum_{i=1}^m \alpha_i = 1$ and

$$\theta = \sum_{i=1}^{m} \alpha_i \theta_i.$$

The set of all convex combinations of a set $\mathcal{D}$ is called the *convex hull* of $\mathcal{D}$, and is denoted by $\textbf{conv}(\mathcal{D})$. It is straightforward to show that the convex hull of every set is a convex set. Moreover, the convex hull of a set is also the smallest convex set that contains it.

Convex function. A function $F : \mathbb{R}^N \to \mathbb{R}$ is *convex* if (a) its domain $\textbf{dom}F$ is a convex set and (b) for all $\theta_1, \theta_2 \in \textbf{dom}F$ and $\alpha \in [0, 1]$ it holds that

$$F((1-\alpha)\theta_1 + \alpha\theta_2) \leq (1-\alpha)F(\theta_1) + \alpha F(\theta_2).$$

In other words, the line segment between the points $(\theta_1, F(\theta_1)), (\theta_2, F(\theta_2))$ lies above the graph of the function F. A function F is called strongly convex if the above inequality holds with strong inequality for $\alpha \neq 0, 1$.

A convex function is continuous but not necessarily differentiable. When F is differentiable, convexity is equivalent to the following first-order condition [23]:

$$F(\theta_2) \geq F(\theta_1) + \nabla F(\theta_1)^\top (\theta_2 - \theta_1) \;\; \forall \theta_1, \theta_2 \in \textbf{dom}F. \tag{2.1}$$

Moreover, if F is differentiable and strongly convex there exists a constant $\beta \in \mathbb{R}_+$, s.t.,

$$F(\theta_2) \geq F(\theta_1) + \nabla F(\theta_1)^\top (\theta_2 - \theta_1) + \beta \|\theta_2 - \theta_1\|_2^2 \quad \forall \theta_1, \theta_2 \in \textbf{dom}F. \tag{2.2}$$

In other words, there is a quadratic lower-bound for F. In this case, we call F *β-strongly convex.*

Lipschitz continuity. For a subset $\mathcal{S} \subset \mathbb{R}^N$ a function F is *Lipschitz continuous* on $\mathcal{S}$ w.r.t. the Euclidean norm $\|\cdot\|$ if there exists a constant L such that

$$\|F(\theta_1) - F(\theta_2)\| \leq L\|\theta_1 - \theta_2\|, \;\; \forall \theta_1, \theta_2 \in \mathcal{S}.$$

Lipchitz continuity measures smoothness of the function F. It can also be extended to derivatives of F: the k-th derivative $F^{(k)}$ is Lipchitz continuous on $\mathcal{S}$ if

$$\|F^{(k)}(\theta_1) - F^{(k)}(\theta_2)\| \leq L\|\theta_1 - \theta_2\|, \;\; \forall \theta_1, \theta_2 \in \mathcal{S}.$$

Moreover, a differentiable function F is called *α-smooth* if there exists a $\alpha \in \mathbb{R}_+$, s.t.,

$$F(\theta_2) \leq F(\theta_1) + \nabla F(\theta_1)^\top (\theta_2 - \theta_1) + \alpha \|\theta_2 - \theta_1\|_2^2 \quad \forall \theta_1, \theta_2 \in \textbf{dom}F.$$

Curvature. A concept closely related to Lipchitz continuity is the *curvature*, which is a measure of nonlinearity of the convex function F over the convex domain $\mathcal{D}$ [21]. Formally, the *curvature* C_f of a convex and differentiable function F is defined as:

$$C_F \equiv \sup_{\theta_1, s \in \mathcal{D}, \alpha \in [0,1], \theta_2 = (1-\alpha)\theta_1 + \alpha s} 2/\alpha^2 (F(\theta_2) - F(\theta_1) - (\theta_2 - \theta_1)^\top \nabla F(\theta_1)).$$

From (2.1) we know that if F is convex and differentiable, then $F(\theta_2)$ lies above its first-order Taylor approximation $F(\theta_1) + (\theta_2 - \theta_1)^\top \nabla F(\theta_1)$. Hence, C_f measures the maximum deviation of F at θ_2 from its linearization at θ_1, scaled with the inverse of the step-size α. For example, if F is a linear function, its curvature C_F is zero.

The curvature of F is related to the Lipchitz continuity constant of the gradient of F: if ∇F is L-Lipchitz continuous on $\mathcal{D}$ then $C_F \leq (\text{diam}(\mathcal{D}))^2 L$, where $\text{diam}(\mathcal{D}) \equiv \sup_{\theta_1,\theta_2 \in \mathcal{D}} \|\theta_1 - \theta_2\|$ denotes the diameter of the set $\mathcal{D}$ [21].

Convex optimization. A general *convex optimization* problem has the form:

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

where $F : \mathbb{R}^N \to \mathbb{R}$ is a convex function and $\mathcal{D}$ is a convex set. A property that makes convex optimization appealing is that every local minimum is a global minimum. Furthermore, for convex optimization the necessary and sufficient optimality condition is given by the following theorem.

Theorem 2.1.1 (Bertsekas, [24]) *For a convex and differentiable function F, the point θ^* is a minimum of F over a convex set $\mathcal{D}$ if and only if*

$$\nabla F(\theta^*)^\top (\theta - \theta^*) \geq 0, \;\; \forall \theta \in \mathcal{D}.$$

2.2 Distributed Methods for Convex Optimization

2.2.1 SGD

Stochastic Gradient Descent (SGD) [25] is a popular algorithm suited for unconstrained optimization problems of the form

$$\text{Minimize} \quad F(\theta) = \sum_{i=1}^{N} \frac{1}{N} F_i(\theta), \tag{2.3}$$

where $F_i(\theta) = L(x_i, \theta) + r(\theta)$. The functions $L : \mathbb{R}^m \times \mathbb{R}^d \to \mathbb{R}, r : \mathbb{R}^d \to \mathbb{R}$ are convex, differentiable and Lipchitz continuous, and each x_i is, e.g., a feature vector of the i-th datapoint in some dataset. Usually, L is a loss function (e.g., for logistic regression it would be logistic loss) and r is a regularization term, e.g., $r(\theta) = \lambda \|\theta\|_2^2$.

SGD solves (2.3) by iteratively performing a sampled form of gradient descent. In particular, the gradient is approximated via the contribution of only one randomly chosen term F_j to ∇F: formally, each iteration $k+1$ of the algorithm is:

$$\theta^{k+1} = \theta^k - \alpha \nabla F_j(\theta^k),$$

where $j \in \{1, \ldots, N\}$ is drawn uniformly at random (u.a.r.) and independently of previous iterations.

ParallelSGD, proposed by [10], parallelizes SGD in the following way: assume that there are P workers, e.g., processors or threads, available. Each worker p independently runs SGD for T iterations and returns a solution θ_p. The final solution returned by the algorithm is the average of the solutions found by the workers:

$$\hat{\theta} = 1/P \sum_{p=1}^{P} \theta_p. \tag{2.4}$$

The most direct implementation requires the storage of the whole dataset on each worker; however, as it is discussed in [10] this can be avoided in practice, as each worker only touches T datapoints, selected u.a.r.; these can be pre-sampled to avoid transferring all N datapoints to a worker.

Despite the simplicity of ParallelSGD, the following strong convergence bound holds (here $g(\theta) = \lambda \|\theta\|_2^2$):

$$\mathop{\mathbb{E}}_{\hat{\theta} \in \mathcal{Q}} [F(\hat{\theta})] - \min_{\theta} F(\theta) \leq \frac{8\alpha G^2}{\sqrt{P\lambda}} \sqrt{L_{\nabla F}} + \frac{8\alpha G^2 L_{\nabla F}}{P\lambda} + 2\alpha G^2,$$

where $\mathcal{Q}$ is the distribution of $\hat{\theta}$, under P workers and $T = \frac{\log P - (\log \alpha - \log \lambda)}{2\alpha\lambda}$ iterations, and $G, L_{\nabla F}$ are upper-bounds for the Lipchitz continuity constants of $L, \nabla F$, respectively.

This and different implementations of distributed SGD [26, 9, 12, 11] suffer from several drawbacks: first of all, the algorithm heavily depends on a separable objective function, while many problems of interest do not have this form (see, e.g., D-OPTIMALDESIGN and A-OPTIMALDESIGN in Section 4.1). Second, it requires the communication and per-worker storage of the whole vector θ, which can be large; this can be avoided when each function F_i depends on a few coordinates of θ [9, 26, 11], but not when the functions $F_1, \ldots, F_N$ have dense support. Moreover, SGD requires a differentiable objective, which further limits the problems that it can solve. For example, it cannot be applied to problems with the widely-used ℓ_1 regularization term.

2.2.2 ADMM

The Alternating Directions Method of Multipliers (ADMM), proposed by Boyd et al. [14], is another distributed optimization algorithm. ADMM solves problems of the form

$$\text{Minimize} \quad F(\theta_1) + g(\theta_2) \tag{2.5a}$$

$$\text{subj. to} \quad A\theta_1 + B\theta_2 = C, \tag{2.5b}$$

where the functions $F : \mathbb{R}^N \to \mathbb{R}, g : \mathbb{R}^m \to \mathbb{R}$ are convex, $A \in \mathbb{R}^{p\times N}, B \in \mathbb{R}^{p\times m}$, and $C \in \mathbb{R}^p$. ADMM forms the augmented Lagrangian:

$$L(\theta_1, \theta_2, y) = F(\theta_1) + g(\theta_2) + y^\top(A\theta_1 + B\theta_2 - C) + \rho/2\|A\theta_1 + B\theta_2 - C\|_2^2,$$

where $y \in \mathbb{R}^p$ is the dual variable associated with (2.5b). ADMM minimizes the augmented Lagrangian w.r.t. the primal variables θ_1, θ_2 alternatively, i.e., keeping one fixed and optimizing w.r.t. other one. Then, it updates the dual variable via gradient ascent. Formally, the algorithm proceeds as follows:

$$\begin{aligned}
\theta_1^{k+1} &:= \arg\min_{\theta_1} L(\theta_1, \theta_2^k, y^k) \\
\theta_2^{k+1} &:= \arg\min_{\theta_2} L(\theta_1^{k+1}, \theta_2, y^k) \qquad\qquad \text{for } k \in \mathbb{N} \\
y^{k+1} &:= y^k + \alpha(A\theta_1^{k+1} + B\theta_2^{k+1} - C).
\end{aligned}$$

Note that this is similar to Dual Ascent, but allows us to divide the primal variables to two sets and optimize them alternatively.

Consensus ADMM can be used to parallelize optimization problems with a separable objective function, i.e., problems of the form:

$$\text{Minimize} \quad \sum_{i=1}^N F_i(\theta) + g(\theta).$$

Usually, each term $F_i : \mathbb{R}^d \to \mathbb{R}$ represents a loss function associated with the i-th datapoint, $g : \mathbb{R}^d \to \mathbb{R}$ is a regularization term, e.g., ℓ_1 penalty term, and $\theta \in \mathbb{R}^d$. This problem can be reformulated as:

$$\text{Minimize} \quad \sum_{i=1}^N F_i(\theta_i) + g(z) \tag{2.6a}$$

$$\text{subj. to} \quad \theta_i = z \;\; i = 1, \dots, N, \tag{2.6b}$$

which is known as the *consensus* problem. The ADMM steps for (2.6) take the form:

$$\theta_i^{k+1} := \arg\min_{\theta_i} F_i(\theta_i) + (y_i^k)^\top(\theta_i - z^k) + \rho/2\|\theta_i - z^k\|_2^2 \tag{2.7}$$

$$z^{k+1} := \arg\min_z g(z) + \sum_{i=1}^N \left(-(y_i^k)^\top z + \rho/2\|\theta_i^{k+1} - z\|_2^2\right) \tag{2.8}$$

$$y_i^{k+1} := y_i^k + \rho(\theta_i^{k+1} - z^{k+1}). \tag{2.9}$$

Note that here the optimization w.r.t. each θ_i and adaptation of y_i can be done independently by a separate worker, i.e., a core or a thread. Also, the second step (2.8) can be done by a central unit in case d is small. This step can be further parallelized again when $F_1, \ldots, F_N$ depend on few variables [14].

ADMM offers several advantages. First, it can solve convex optimization problems with non-differentiable terms in the objective. In particular, it reduces the optimization of any convex function plus a ℓ_1 regularization term to the optimization of a quadratic term plus the ℓ_1 regularization term, which has a closed-form solution. Moreover, it can parallelize other generic optimization problems (see Sections 7 and 8 of [14]). However, it again assumes a separable objective function, which restricts the class of problems that it can solve. Finally, Consensus ADMM may not scale where the variable θ is high-dimensional and the functions $F_1, \ldots, F_N$ have dense support.

2.2.3 DSDCA

Distributed Stochastic Dual Coordinate Ascent (DSDCA) is another parallelizable optimization algorithm, which solves problems in their dual domain [13]. DSDCA solves problems of the form:

$$\text{Minimize} \quad \sum_{i=1}^{N} F_i(\theta^\top x_i) + g(\theta), \tag{2.10}$$

where functions $F_i : \mathbb{R} \to \mathbb{R}, i = 1, \ldots, N$, are convex and Lipchitz continuous, $g : \mathbb{R}^d \to \mathbb{R}$ is a β-strongly convex function, and $x_1, \ldots, x_N \in \mathbb{R}^d$ are feature vectors. In (2.10), similar to (2.3), each $F_i, i = 1, \ldots, N$, denotes a loss function and g represents a regularization term; here, each loss term F_i is explicitly a function of the linear term $\theta^\top x_i$.

Let us denote the convex conjugate of F_i and g by F_i^* and g^*, respectively. The dual problem of (2.10) is given by

$$\max_{\alpha} \sum_{i=1}^{N} -F_i^*(-\alpha_i) - g^* \left(\sum_{i=1}^{N} \alpha_i x_i \right).$$

In order to solve this problem, DSDCA at each iteration updates a randomly selected subset of the dual variables $\alpha_1, \ldots, \alpha_N$ via gradient ascent.

The separable form of the dual problem allows its solution to be parallelized: assume that the dataset $x_i, i = 1, \ldots, N$, is distributed between P workers. At iteration k of DSDCA each worker p iteratively updates m randomly chosen dual variables corresponding to the datapoints that it holds:

$$\alpha_i^{k+1} = \alpha_i^k + \Delta_i.$$

The step-size Δ_i is given by maximizing the following lower-bound of the dual function, which is a result of strong convexity of g (see (2.2)):

$$\Delta_i = \max_{\Delta\alpha} -F_i^*(-(\alpha_i^k + \Delta\alpha)) - \Delta\alpha x_i^\top \theta^k - \beta(\Delta\alpha)^2 \|x_i\|_2^2,$$

where $\theta^k = \nabla g^*(\sum_{i=1}^N \alpha_i^k x_i)$. A central unit computes θ^k by evaluating the gradient function $\nabla F^* : \mathbb{R}^d \to \mathbb{R}^d$.

Interestingly, it can be verified that for the optimal primal and dual solutions, i.e., θ^* and α^*, the following holds [13]:

$$\theta^* = \nabla g^*(\sum_{i=1}^N \alpha_i^* x_i).$$

Therefore, as α^k converges to α^*, the primal solution θ^k also converges to its optimal value θ^*.

The main advantage of DSDCA over SGD and ADMM is its convergence rate: studies have shown that coordinate-ascent in the dual domain outperforms SGD [27, 28, 29, 13]. Therefore, when the objective has the particular form (2.10), DSDCA is a more efficient alternative. However, similar to SGD and ADMM, its applicability is curbed because of the separable objective assumption. Furthermore, just like SGD and ADMM, if the vector θ is high-dimensional, DSDCA is both computation and communication intensive.

2.3 Frank-Wolfe

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

$$\min_{\theta \in \mathcal{D}_0} F(\theta), \tag{2.11}$$

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

$$\mathcal{D}_0 \equiv \left\{ \theta \in \mathbb{R}^N_{\geq 0} : \sum_{i=1}^N \theta_i = 1 \right\} \tag{2.12}$$

is the N-dimensional simplex. Several important problems, including experimental design, training SVMs, Adaboost, and projection to a convex hull indeed take this form [20, 22, 23] (see Section 4.1). 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

Section 2.2, this is precisely the regime in which (2.11) is hard to parallelize via existing methods such as SGD, ADMM, and DSDCA.

It is well known that (2.11) admits an efficient implementation through the Frank-Wolfe (FW) algorithm [15], also known as the conditional gradient algorithm [20, 21]. We describe the Frank-Wolfe algorithm and its benefits, in detail, in Section 2.3.1. We then discuss several FW variants and FW distributed implementations in Sections 2.3.2 and 2.3.3, respectively. In Section 2.3.4, we illustrate how FW assumes a very simple and elegant form under the simplex constraint $\mathcal{D}_0$. This has important computational advantages, which we use later on for parallelizing FW.

2.3.1 Frank-Wolfe Algorithm

The FW algorithm [15], summarized in Alg. 1, solves problems of the form:

$$\text{Minimize} \quad F(\theta) \tag{2.13a}$$

$$\text{subj. to:} \quad \theta \in \mathcal{D}, \tag{2.13b}$$

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:

$$s^k = \arg\min_{s\in\mathcal{D}} \; s^\top \cdot \nabla F(\theta^k), \tag{2.14a}$$

$$\theta^{k+1} = (1-\gamma^k)\theta^k + \gamma^k s^k, \tag{2.14b}$$

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 (2.14a),(2.14b) 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* [21]. The duality gap at feasible point $\theta^k \in \mathcal{D}$ in iteration $k \in \mathbb{N}$ is:

$$g(\theta^k) \equiv \max_{s\in\mathcal{D}} (\theta^k - s)^\top \nabla F(\theta^k) \overset{(2.14a)}{=} (\theta^k - s^k)^T \nabla F(\theta^k), \tag{2.15}$$

The convexity of F implies that $F(\theta^k) - F(\theta^*) \le g(\theta^k)$ for any optimal solution $\theta^* \in \arg\min_{\theta\in\mathcal{D}} F(\theta)$ [21]. 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 $\epsilon > 0$.

Algorithm 1 FRANK-WOLFE

```
1: Pick $\theta^0 \in \mathcal{D}$
2: $k := 0$
3: **repeat**
4:     $s^k := \arg\min_{s \in \mathcal{D}} s^\top \cdot \nabla F(\theta^k)$
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 $< \epsilon$
```

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

$$\gamma^k = \underset{\gamma \in [0,1]}{\arg\min}\ F\big((1-\gamma)\theta^k + \gamma s^k\big). \tag{2.16}$$

Convergence to an optimal solution is guaranteed in both cases for problems in which the objective has a bounded curvature [15, 21]. In this case, both of the above step sizes imply that the k-th iteration of the Frank-Wolfe algorithm satisfies $F(\theta^k) - F(\theta^*) \leq O(\frac{1}{k})$ [21]. For arbitrary convex objectives with unbounded curvature, FW still converges if the step size is set by the line minimization rule [24].

FW has several advantages. First, it reduces the optimization of a general convex function subject to a convex constraint set to the optimization of a linear objective function (see (2.14a)) subject to the convex constraint set. For several constraint sets of practical interest, e.g., linear constraint sets, these sub-problems are solved efficiently [21, 30]. In particular, when the constraint set is a linear constraint set, FW reduces the original problem to solving a sequences of linear programming problems; these problems can be solved by efficient linear programming techniques. Another example is the simplex constraint set; we show in Section 4.1 that many problems of interest have this constraint set. As we discuss in Section 2.3.4, sub-problem (2.14a) admits a very simple solution under this constraint set.

Another advantage of FW over other methods such as ADMM or barrier methods is that it maintains feasibility by finding convex combinations of feasible points (see (2.14b)). It is important for applications that require feasibility of the solutions through the iterations. Moreover, in contrast to other optimization methods that generate feasible solutions, e.g., projected gradient descent, FW is projection-free. Projected gradient descent, projects the solution on the constraint set at each iteration to obtain a feasible solution. The projection on a constraint set requires minimizing a quadratic function, which measures the distance between the given point and a point on the set, subject to the original constraint set. As discussed in [17] for many constraint sets of interest solving (2.14a) with a linear objective is significantly cheaper than solving the projection problem with a quadratic function. In particular, projection of a matrix $X \in \mathbb{R}^{M \times N}$ on the set of bounded trace norm matrices requires finding the SVD decomposition of X, while solving (2.14a) is done by only finding the top-eigenvectors of X. The former has a time complexity of $O(NM^2)$, while for the latter fast algorithms, linear in the number of non-zero elements of X, exist [17].

Moreover, FW for some constraint sets, e.g., the simplex, generates sparse solutions with provable approximations [20]. Sparse solutions are often desired in practice; for example in [31] they are used to speed up the solutions of SVM problem. The solution to (2.14a) for the simplex is given by (2.19). This solution is extremely sparse, i.e., it has only one non-zero element. This along with the simple adaptation step (2.14b) ensures that the solution at k-iteration θ^k has at most k non-zero elements. Moreover, considering the convergence guarantees of FW, this sparse solution is within the $\frac{1}{k}$-neighborhood of the optimal solution.

2.3.2 FW Variants

There are several variants of FW in the literature. In general, these variants are divided to two groups; first group improves the convergence rate of FW [32, 33, 34, 35, 36, 37]. The other group reduces computation time of each iteration of FW by randomizing the algorithm, while obtaining convergence guarantees [17, 38, 39, 40]. As these algorithms are not readily parallelizable or applicable to the problems we consider here, we focus on parallelizing the classic FW in this thesis. We leave parallelization of these variants for future work. Nonetheless, for completeness, we explain some of these variants in this Section.

Fast-convergent variants. 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 [41, 42, 15, 21]. This is because the iterations of the classic FW *zig-zag* between the vertices defining the face that contains the optimal solution. To avoid this zig-zagging phenomenon, Wolfe in [32] proposed a variant using ‘away-points’; the basic idea is to move away from a ‘bad’ direction. Guélat and Marcotte [43] 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 [33, 34, 35, 36, 37].

Here, we introduce two of these variants, i.e., Away-step FW and Pairwise FW: Lacoste-

Jullien and Jaggi [35] showed that for a strongly-convex objective and a polytope constraint set these variants enjoy a linear convergence rate. In particular, they solve (2.13) for a strongly-convex F and the constraint set

$$\mathcal{D} = \textbf{conv}(\mathcal{A}),$$

where $\mathcal{A}$ is a finite set of M points $a_1, \dots, a_M \in \mathbb{R}^N$, called *atoms*. Note that (2.14a) for this constraint set takes the form:

$$a_{FW}^k = \arg\min_{a \in \mathcal{A}} \; a^\top \cdot \nabla F(\theta^k). \tag{2.17}$$

They refer to a_{FW}^k as *FW atom*. In both of the variants, the solution θ^k at each iteration k is a convex combination of the atoms:

$$\theta^k = \sum_{i=1}^{M} \gamma_i^k a_i.$$

At each iteration, they denote the *active* atoms by the set $\mathcal{A}^k = \{a_i \in \mathcal{A} : \gamma_i^k > 0\}$.

Away-step FW. The basic idea for Away-step FW is to mitigate the zig-zagging phenomenon by moving *away* from a *bad* direction, i.e., the direction d that maximizes the descent potential given by $P \equiv d^\top \cdot \nabla F(\theta)$. At each iteration k, Away-step FW, just like FW, finds the FW atom a_{FW}^k given by (2.17). Then, it finds an *away atom* given by:

$$a_A^k = \arg\max_{a \in \mathcal{A}^k} \; a^\top \cdot \nabla F(\theta^k). \tag{2.18}$$

Note that, here, the search region is over the selected atoms $\mathcal{A}^k \subseteq \mathcal{A}$, which is usually smaller than $\mathcal{A}$. Therefore, this optimization problem is easier than (2.17). Then, Away-step FW evaluates the descent potentials corresponding to the FW and away atoms, i.e., P_{FW}^k and P_{Away}^k, respectively defined as:

$$P_{\text{FW}}^k = (a_{FW}^k - \theta^k)^\top \cdot \nabla F(\theta^k),$$
$$P_{\text{Away}}^k = (\theta^k - a_A^k)^\top \cdot \nabla F(\theta^k).$$

Then, depending on which potential is minimum, Away-step FW adapts the current solution: in case $P_{\text{FW}}^k < P_{\text{Away}}^k$, θ^{k+1} is given by:

$$\theta^{k+1} = \theta^k + \alpha(a_{FW}^k - \theta^k).$$

Note that this is exactly the adaptation step in the classic FW (2.14b). Otherwise, if $P_{\text{FW}}^k > P_{\text{Away}}^k$, the adaption has the form:

$$\theta^{k+1} = \theta^k + \alpha(\theta^k - a_A^k).$$

Note that here for $\alpha \geq 0$, θ^{k+1} is *not* a convex combination of the points θ^k, a_A^k. Hence, the step-size α is selected to ensure $\theta^{k+1} \in \mathcal{D}$.

Pairwise FW. The basic idea in Pairwise FW is to move weights only between two atoms. Formally, at each iteration k, Pairwise FW, similar to Away-step FW, finds the FW and away atoms, i.e., a_{FW}^k and a_A^k, respectively. Then, it adapts the solution θ^k by swapping weights between these two atoms, while keeping all other weights fixed:

$$\theta^{k+1} = \theta^k + \alpha(a_{FW}^k - a_A^k).$$

Note that this is different from classic FW, which shrinks all of the weights, except the FW atom weight, by a factor of $1 - \alpha$. Again, note that θ^{k+1} for $\alpha \geq 0$ is not a convex combination of the points θ^k, a_{FW}^k and a_A^k. Therefore, the step-size is set in a way that guarantees feasibility of θ^{k+1}. The convergence bound for Pairwise FW, proved in [35], is looser than Away-step FW. However, it works well in practice [35].

These two FW variants, as well as others in the literature [44, 45, 46, 35] converge faster than FW. However, as setting the step-size is more challenging and keeping track of the active atoms further complicates the algorithm, in this thesis we focus on parallelizing the classic FW.

Stochastic variants. Stochastic variants of FW have recently been proposed [17, 38, 39, 40], which stochastically approximate the gradient. In general, these variants solve convex optimization problems (2.13), where the objective F has a separable form:

$$F(\theta) = 1/M \sum_{i=1}^{M} F_i(\theta).$$

Moreover, they consider constraint sets $\mathcal{D}$, for which solving the linear sub-problem (2.14a) is easy, while projection over them is expensive. Hazan and Kale [17] list such constraint sets.

Hazan and Kale [17] focused on an online learning setting; however, their result is inferior to a trivial stochastic FW algorithm, called SFW in [39], which similar to SGD estimates the gradient ∇F by ∇F_i for some $i \in \{1, \ldots, M\}$, selected u.a.r. Lan and Zhou [38] introduced the Conditional Gradient Sliding method, which allows the algorithm to skip the computation of the gradient from time to time. They also proposed a stochastic version of this algorithm. Hazan and Lou [39] improved these results by using a *variance-reduced stochastic gradient* [47, 48] to estimate the gradient. Variance-reduced stochastic gradient provides an unbiased estimate of the gradient with a bounded variance, at the expense of computing the exact gradient for a point. Hazan and Lou [39] provide a thorough survey, and show that their algorithms outperform [17, 38] and SFW, in terms of the number of exact and stochastic gradient computations and the number of subproblems (2.14a) needed to obtain a solution within ϵ-neighborhood of the optimal. In particular, they reduce the number of stochastic gradient evaluations from $O(\frac{1}{\epsilon^2})$ to $O(\frac{1}{\epsilon^{1.5}})$ for smooth objective functions and from $O(\frac{1}{\epsilon})$ to $O(\ln\frac{1}{\epsilon})$ for smooth and strongly-convex objective functions. Reddi et al. [40] use the variance-reduced stochastic gradient idea to propose two stochastic variants of FW for non-convex objective functions. They show that both algorithms converge to a stationary point, with convergence rates faster than classic FW.

A different stochastic variants of FW solves problems with block-separable constraints [49]. Lacoste-Julien et al. in [49], proposed a random single-block FW algorithm, in which only a single block of variables, selected u.a.r., is updated. At the expense of computing the duality gap, the convergence result was improved in [50].

We implement two stochastic FW variants based on gradient subsampling: the basic idea is to compute the partial derivatives in only a random subset of the directions.We compare the relative performance of subsampling to increasing parallelism in Section 5.2.

2.3.3 Distributed implementations

More recently, and more relevant to our work, Bellet et al. [22] 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. 3). The authors characterize the message and parallel complexity when A is partitioned across multiple processors under broadcast operations. Moreover, Tran et al in [51] elaborated on their algorithm, and proposed an asynchronous version of the distributed Frank-Wolfe algorithm in [22]. It is based on their *Stale Synchronous Parallel* (SSP) model [51]. 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 differ from these implementations in the following two ways:

- We 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. 4.1). Our algorithm can be viewed as a generalization of their algorithm. In particular, our distributed algorithm can be applied to the problems they consider, i.e., $F(\theta) = g(A\theta)$, by defining the common information as $A\theta$ (see Prop. 1 and 2 in Section 3).

- We 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.

2.3.4 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 (2.12). As described in Section 4.1, this set of constraints arises in many problems, including training SVMs, convex approximation, Adaboost, and experimental design (see also [20]). Under this set of constraints, the linear optimization problem in (2.14a) 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 $\mathbb{R}^N$, (2.14a) reduces to:

$$s^k = e_{i^*}, \text{ where } i^* \in \arg\min_{i\in[N]} \frac{\partial F(\theta^k)}{\partial \theta_i}. \tag{2.19}$$

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.

2.4 Distributed Frameworks

2.4.1 Map-Reduce Framework

Map-reduce [1, 2] 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 [3, 4, 5, 6, 7]. 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 [1, 2, 8].

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 [52]. There exist several computational frameworks, including Hadoop [2] and Spark [8], 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.

We opted to implement our distributed algorithm in the map-reduce framework. The first reason is that its implementations such as Hadoop [2] or Spark [8] are readily available on commercially used clusters. The second reason is that map-reduce implementations allow programming in high-level languages, e.g., Python, which are easy to program with. Code in these high-level languages can be written in a compact and concise fashion. In particular, we implement our distributed FW algorithm over Spark [8], which is an open-source implementation of map-reduce well-suited for parallelizing iterative machine learning algorithms: this is because Spark caches the results on RAM, so that they can be used in the next iteration.

2.4.2 Message Passing Interface (MPI)

Message Passing Interface (MPI) is a standard for message passing libraries aimed at running programs on HPC platforms, e.g., clusters of computers. The standard defines syntax and semantics of a set of library routines for different programming languages such as C, C++, or python. MPI is the dominant framework used in HPC applications [53]. MPI provides topology, communication, and synchronization between a set of processes: each process is usually mapped to a processor. MPI uses objects called *communicators* to determine which subset of processes may communicate with each other. MPI provides both *point-to-point* and *collective* communication routines.

Point-to-point routines are used for communication between two specific processes. For example, `MPI_Send` allows a specified process to send a message to another specified process. Correspondingly, `MPI_Recieve` lets a specified process to receive a message from another specified process. Collective routines involve communication between all processes in a communicator. Examples of collective routines are `MPI_Bcast`, `MPI_Scatter`, `MPI_Reduce`, or `MPI_Allreduce`. `MPI_Bcast` sends data from a specified process to all other processes. `MPI_Scatter` distributes data from a stipulated process between all of the processes. `MPI_Reduce`, similar to `reduce` in the map-reduce framework, aggregates over data held by different processes. `MPI_Allreduce` performs a reduction over the data and then broadcasts the result to every processor; it is a `MPI_Reduce` followed by `MPI_Bcast`.

Map-reduce has several advantages over MPI. First, map-reduce is readily available on commercial clusters, e.g., Amazon Web Services, Microsoft Azure, and Google Cloud. Second, programming in the map-reduce framework is much easier: implementation in MPI is done on lower-level languages, moreover, it requires explicit specifications of communication types as well as of the processors that need to communicate. On the other hand, MPI allows for both point-to-point routines for communication between two specific processors and more sophisticated collective operations such as `MPI_Allreduce`.

Chapter 3

Frank-Wolfe via Map-Reduce

3.1 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 (2.14), 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 $i \in [N]$:*

$$\frac{\partial F(\theta)}{\partial \theta_i} = G(h(X;\theta), x_i, \theta_i), \tag{3.1}$$

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 (2.14) and (2.19) 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:*

$$h(X;\theta^{k+1}) = H(h(X;\theta^k), x_{i^*}, \theta^k_{i^*}, \gamma^k), \tag{3.2}$$

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 (2.19): 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 4.1. 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

$$\text{Minimize} \quad F(\theta) = \|X^T\theta - p\|_2^2 \tag{3.3a}$$

$$\text{subj. to:} \quad \theta \in \mathcal{D}_0, \tag{3.3b}$$

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

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

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

$$h(X;\theta) = X^T\theta - p, \tag{3.4}$$

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 (2.14) and (2.19), 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 $N \gg 1$).

3.2 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, 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.

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 $< \epsilon$
```

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 (2.14) 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 5.2, this leads to a significant speedup, allowing Alg. 2 to outperform interior-point methods even when executed serially.

3.3 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_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

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:     $D' = D.\texttt{map}\big((i,x_i,\theta_i) \mapsto (i,x_i,\theta_i,G(h,x_i,\theta_i)\big)$
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 $< \epsilon$
```

$i^* \in \arg\min_{i\in[N]} z_i$. 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 $O(d)$ [52]. 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 $O(m+d)$.

3.4 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 (2.16) is used. Nevertheless, all problems we consider here, including

CONVEXAPPROXIMATION, satisfy an additional property that ensures that (2.16) 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 (2.16) at iteration k is:

$$\gamma^k = \underset{\gamma \in [0,1]}{\arg\min}\ \hat{F}\left(h(X; (1-\gamma)\theta^k + \gamma e_{i^*})\right). \tag{3.5}$$

The argument of $\hat{F}$ is the updated common information h^{k+1} under step size γ. Hence, using Prop. 2, Eq. (3.5) becomes:

$$\gamma^k = \underset{\gamma \in [0,1]}{\arg\min}\ \hat{F}\left(H(h, x_{i^*}, \theta^k_{i^*}, \gamma)\right), \tag{3.6}$$

where h is the present common information. As F is convex in θ^k, it is also convex in γ, so (3.6) is also a convex optimization problem. Crucially, (3.6) 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 (3.6). 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 4.1, 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.

Chapter 4

Examples and Extensions

4.1 Examples

We provide several examples of problems that satisfy Prop. 1, 2, and 3; a summary is given in Table 4.1.

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

Table 4.1: Examples of problems satisfying Prop. 1–3.

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 [23]:

$$\min_{\theta \in \mathcal{D}_0} \mathtt{f}\left(\left(\sum_{i=1}^{N} \theta_i x_i x_i^\top\right)^{-1}\right), \tag{4.1}$$

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 `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 `f` is the log-determinant, and (4.1) becomes:

$$\text{D-OptimalDesign}$$

$$\text{Minimize} \quad F(\theta) = \operatorname{logdet}\left(\sum_{i=1}^{N} \theta_i x_i x_i^\top\right)^{-1} \tag{4.2a}$$

$$\text{subj. to:} \quad \theta \in \mathcal{D}_0, \tag{4.2b}$$

D-OptimalDesign satisfies Prop. 1 as:

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

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

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

Hence, Prop. 1 holds when $d^2 \ll N$. Using the Sherman-Morrison formula [54] we can show that the common information at step $k+1$ is:

$$A^{-1}(\theta^{k+1}) = \frac{A^{-1}(\theta^k)}{1-\gamma} - \frac{\frac{\gamma}{(1-\gamma)^2} A^{-1}(\theta^k) x_{i^*} x_{i^*}^\top A^{-1}(\theta^k)}{1 + \frac{\gamma}{1-\gamma} x_{i^*}^\top A^{-1}(\theta^k) x_{i^*}}. \tag{4.3}$$

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:

$$H(h,x,\gamma) = \frac{h}{1-\gamma} - \frac{\frac{\gamma}{(1-\gamma)^2} h x x^\top h}{1 + \frac{\gamma}{1-\gamma} x^\top h x}. \tag{4.4}$$

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 `f` is the trace:

A-OPTIMALDESIGN

$$\text{Minimize} \quad F(\theta) = \text{Tr}\left(A^{-1}(\theta)\right) \tag{4.5a}$$

$$\text{subj. to:} \quad \theta \in \mathcal{D}_0. \tag{4.5b}$$

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

$$h_1 = A^{-1}(\theta),$$
$$h_2 = A^{-2}(\theta).$$

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 (4.3). 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 (4.4), 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:

$$\textsc{AdaBoost}$$

$$\text{Minimize} \quad F(\theta) = \log\left(\sum_{j=1}^{d} \exp(-\alpha c_j(X,\theta) r_j)\right) \tag{4.6a}$$

$$\text{subj. to:} \quad \theta \in \mathcal{D}_0, \tag{4.6b}$$

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, (4.6) 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

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

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

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

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 (2.14) and (2.19), 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 4.1 are convex, and some admit specialized solvers. A-OPTIMALDESIGN can be reduced to a semidefinite program, (see Sec. 7.5 of [23]), and solved as an SDP. ADABOOST can be expressed as a geometric program (GP) [20], 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. 5.2 we compare FW to the above specialized solvers, and we see that it outperforms them in all cases.

4.2 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 [55, 56]. For this constraint, adaptation (2.14b) becomes:

$$s^k = \sigma_{i^*} e_{i^*}, \text{ where } i^* = \arg\max_{i \in [N]} \left|\frac{\partial f}{\partial \theta_i}\right|, \tag{4.7}$$

and $\sigma_{i^*} = -K \text{sign}(\frac{\partial f}{\partial \theta_{i^*}})$. Eq. (4.7) 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 [56]:

$$\min_{\theta : \|\theta\|_1 \le K} \|X^\top \theta - p\|_2^2. \tag{4.8}$$

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 (3.4) 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

$$\|x\|_{\mathcal{A}} = \inf\{t \geq 0 : x \in t\textbf{conv}(\mathcal{A})\},$$

where $\textbf{conv}(\mathcal{A})$ 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 [57, 58, 59, 30]. Tewari et al. [30] propose an FW-like algorithm for this class of problems. In this algorithm, the step 4 of Alg. 1 is replaced by

$$s^k = \arg\min_{a\in\mathcal{A}} a^\top \cdot \nabla F(\theta^k). \tag{4.9}$$

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. (4.9) 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.

Chapter 5

Experimental Study

5.1 Implementation

We implemented Alg. 3 over Spark, an open-source cluster-computing framework [8]. 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 [8].

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 4.1 and used it in our experiments.

5.2 Experiments

5.2.1 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

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

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.

Algorithms. We solve Convex Approximation, Adaboost, D-Optimal Design, and A-Optimal Design summarized in Table 4.1, as well as LASSO (c.f. Sec. 4.2). We implement both serial and parallel solvers. First, we implement Serial FW (Alg. 2) in Python, setting γ using the line minimization rule (2.16). 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 [14]. We also implement our parallel algorithm (Alg. 3) using our Spark generic implementation. We again set the step size using the line minimization rule (2.16). We refer to this algorithm as Parallel FW. We also introduce two stochastic parallel variants that subsample the gradient; we discuss these in Section 5.2.4.

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 5.1–5.3.

Real Data. We also experiment with 4 real datasets, summarized in Table 5.4. The first dataset is Movielens [60]. 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 [61]. The dataset has 10^6 data points and 28 features.

[2] `cvxopt.org`

We represent it as a matrix with $N = 10^6$ and $d = 28$. The third dataset is the MSD dataset [61], 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 [62], 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.

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:

$$t_\epsilon = \min\left\{t : \tfrac{F(\theta(t))}{F(\theta(t))-g(\theta(t))} \leq 1+\epsilon\right\}, \tag{5.1}$$

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.

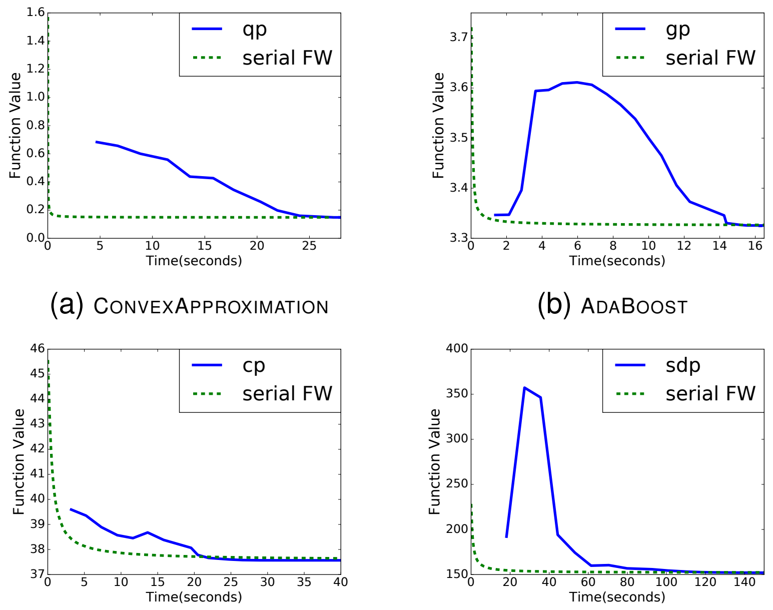
5.2.2 Serial Execution

Our first experiment compares the Serial FW algorithm with the specialized interior point solvers mentioned in Section 4.1 (i.e., `cp`, `qp`, `sdp`, and `gp`) for each of the problems in Table 4.1. We use the small synthetic dataset (Dataset A) in Table 5.1.

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. 5.1 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

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



(a) ConvexApproximation (b) AdaBoost

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

Figure 5.1: 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.

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.

5.2.3 Effect of Parallelism

To study parallelism, we first show results for two large-scale synthetic datasets: Dataset B, a dataset with $N = 20$M and $d = 100$ (Table 5.2), and Dataset C with $N = 100$K and $d = 1$K (Table 5.3). Fig. 5.2 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. Figure 5.3 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 5.5. 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. For the input sizes used in these

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 5.1: Dataset A

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 5.2: 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

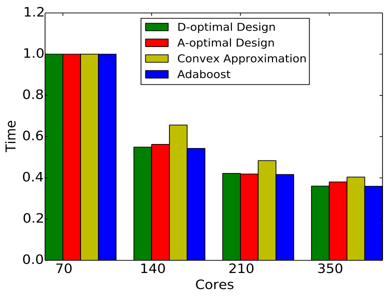
Table 5.3: Dataset C

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

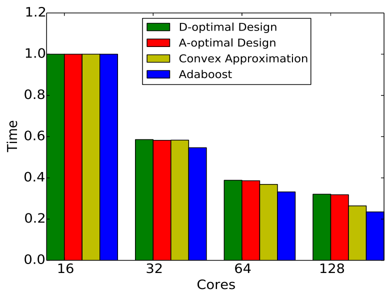
Table 5.4: Real Datasets

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 5.5: A 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) Dataset B



(b) Dataset C

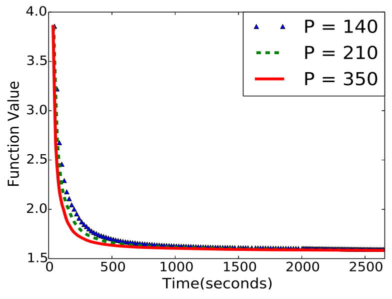
Figure 5.2: The t_ϵ as a function of the level of parallelism, measured in terms of cores P. Fig. 5.2a shows results on the 20M variable dataset (Table 5.2) while Fig. 5.2b shows results on the dataset with $d = 1000$ (Table 5.3). We normalize t_ϵ by its value at the lowest level of parallelism (13134s, 27420s, 6573s, and 4447s, respectively, for each of the four problems in Fig. 5.2a and 487s, 486s, 62s, and 483s, respectively, in Fig. 5.2b). We see that increasing the level of parallelism speeds up convergence.

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 5.3a and 5.3b with Figures 5.3c and Figure 5.3d, 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 3, FW for these problems does not have a $O(\frac{1}{k})$ convergence rate.

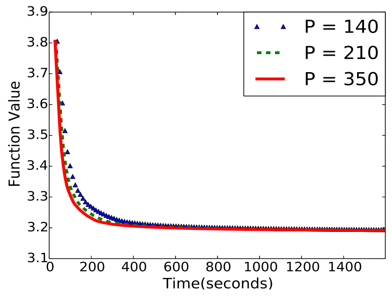
Next, we move on to experiments on the real datasets, summarized in Table 5.4. For brevity, we only report D-Optimal Design for Movielens, HEPMASS, and MSD datasets, and D-optimal design, A-optimal Design, and Convex Approximation for the YAHOO dataset. Fig. 5.4 shows the measured t_ϵ for different levels of parallelism. For each dataset, t_ϵ is normalized by the value of t_ϵ for the lowest level of parallelism. Again, we see that we gain a significant speedup by parallelism.

5.2.4 Subsampling the Gradient

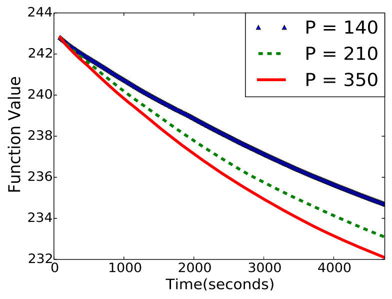
In this section, we study the effect of subsampling the gradient on the performance of FW. We have seen that parallelism reduces the cost of computation of the gradients. An alternative is to compute the gradient stochastically by subsampling only a few partial derivatives and using the minimal in this sub-sampled set. This reduces the amount of computation occurring in each iteration. Moreover, such a stochastic estimation of the gradient still guarantees convergence [40], albeit at a



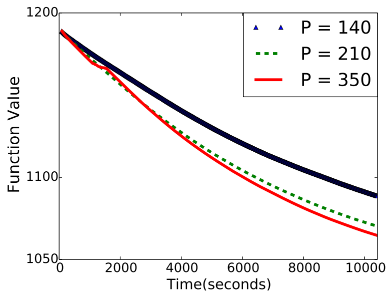
(a) ConvexApproximation



(b) AdaBoost

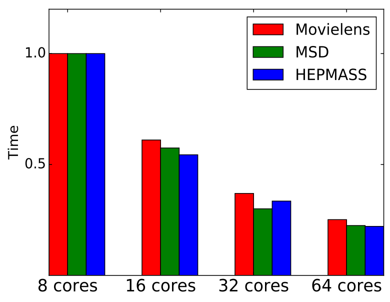


(c) D-OptimalDesign

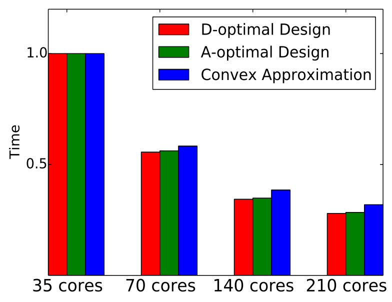


(d) A-OptimalDesign

Figure 5.3: 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 5.3a and 5.3b with Figures 5.3c and 5.3d, we see that FW for D-Optimal Design and A-Optimal Design converges slower.



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



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

Figure 5.4: The 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. 5.4a, and 9888s, 7060s, and 1302s for D-optimal Design, A-optimal Design, and Convex Approximation, respectively, in Fig 5.4b.

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 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 5.3) in this experiment: we solve the corresponding problems using Sampled FW and Smoothened FW on 16 cores. The results are shown in Fig. 5.5. Values t_ϵ are normalized by t_ϵ for $p = 1$. This makes experiments in Figures 5.5 and 5.2b comparable: each core computes the same number of partial derivatives in expectation.

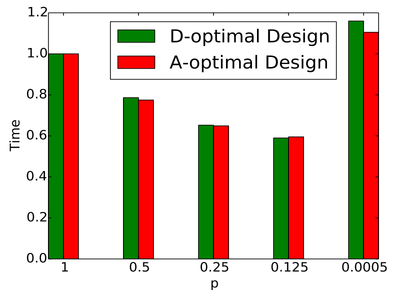
By comparing Figures 5.5 and 5.2b, we see that subsampling matches the benefits of parallelism, at least for large p, for D-optimal and A-optimal design. In contrast, the benefits of subsampling for Convex Approximation and AdaBoost are almost negligible. This is because Parallel FW guarantees a $O(\frac{1}{k})$ 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 5.5d and 5.5c, where t_ϵ for Convex Approximation and AdaBoost is considerably smaller under Smoothened FW.

5.2.5 LASSO Experiment

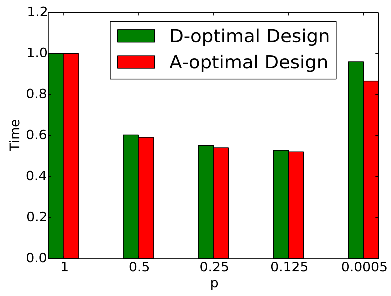
To show the performance of our algorithm on the cases beyond simplex constrained problems, we solve the LASSO problem (4.8). We compare our distributed FW with distributed ADMM.

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

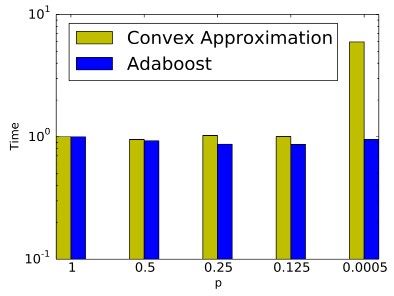
$$\min_{\theta} \frac{1}{2}\|X^\top \theta - p\|_2^2 + \|\theta\|_1,$$



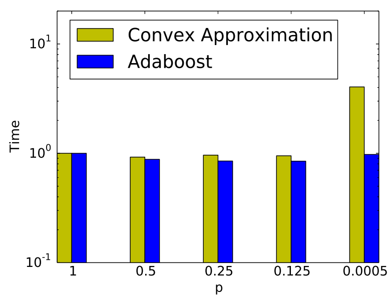
(a) Sampled FW for D-OPTIMALDESIGN and A-OPTIMALDESIGN



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



(c) Sampled FW for ADABOOST and CONVEXAPPROXIMATION



(d) Smoothened FW for ADABOOST and CONVEXAPPROXIMATION

Figure 5.5: The 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. 5.2. By comparing Figures 5.5a and 5.5c with Fig. 5.2b, 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.

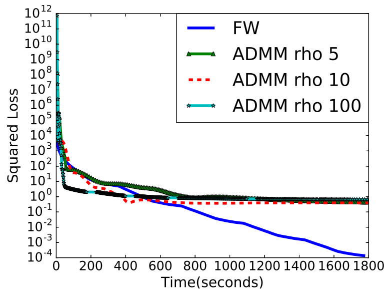


Figure 5.6: The comparison between ADMM and our distributed Frank-Wolfe algorithm. Each algorithm uses 400 cores.

with distributed ADMM using 400 cores and for different values of ρ, which is a parameter controlling convergence (see Section 8.3 of [14]). 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. 5.6 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.

Chapter 6

Conclusion

We establish structural properties under which FW admits a highly scalable parallel implementation via map-reduce. We show that problems distributed by our algorithm achieve significant speedups. In particular, by using 350 cores we are able to solve a problem with 20 million variables in 79 minutes, while the serial implementation takes 165 hours. Moreover, we show that our results extend beyond the simplex constraint. For instance, our distributed FW algorithm can be applied to problems with the popular and widely-used ℓ_1-norm constraint.

The Frank-Wolfe algorithm is related to approximate algorithms for so-called *submodular maximization problems*. *Submodularity* is a structural property associated with *set functions*. Submodularity captures the notion of *diminishing returns* or *decreasing marginal utilities* [63]; this makes it a suitable objective in computer science or economics to represent subset evaluations of, e.g, a set of utilities [63]. In particular, *maximizing submodular functions subject to matroid constraints* has numerous applications in the combinatorial optimization domain, such as variable selection [64], dictionary learning [65, 66], document summarization [67, 68], etc. Maximizing submodular functions subject to matroid constraints are known as the convex optimization counterpart in the combinatorial optimization domain [63]: though NP-hard, these problems can be solved with approximate guarantees [69, 70, 71]. A *greedy* algorithm [70] produces a solution that is guaranteed to be within $1/2$ ratio of the optimal solution. The so-called *continuous greedy algorithm* improved this ratio to $(1 - 1/e)$ [72, 73]; moreover, this ratio cannot be improved further for polynomial-time algorithms [74]. The continuous greedy algorithm is in fact a variant of FW. It maximizes a *multilinear relaxation* [69] of the original submodular maximization problem. Through this connection, FW has important applications in solving these combinatorial optimization problems. Parallelizing this variant of FW, which solves generic submodular optimization problems with guarantees is an important direction as a future work.

Though submodularity is conventionally defined for set functions, its definition has also been extended for continuous functions [75, 76]. More recently, Bian et al. [71] defined *DR-submodular* functions, which are a subset of the continuous submodular functions. The scope of the DR-submodular functions comprises a subset of convex functions, a subset of concave functions, and a subset of functions that are neither convex nor concave. Moreover, they show that many interesting computer science problems such as, maximzing linear extensions, e.g., the Lovasz extension [77], of submodular set functions, non-convex/non-concave quadratic programming, optimal budget allocation, etc, have DR-submodular objectives. They also prove that a FW variant, which is similar to the continuous greedy algorithm, maximizes the monotone DR-submodular functions again with the $(1 - 1/e)$ ratio within the optimal solution. In other words, this FW variant, with guarantees, solves a class of generic problems with diverse applications, which interestingly includes non-convex optimization problems. Therefore, another useful area as future work is parallelizing this FW variant: this allows to solve large-scale monotone DR-submodular maximization problems in a moderate time.

Bibliography

[1] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[2] A. Bialecki, M. Cafarella, D. Cutting, and O. O?malley, "Hadoop: a framework for running applications on large clusters built of commodity hardware," 2005.

[3] H.-c. Yang, A. Dasdan, R.-L. Hsiao, and D. S. Parker, "Map-reduce-merge: simplified relational data processing on large clusters," in *SIGMOD*, 2007.

[4] R. Kumar, B. Moseley, S. Vassilvitskii, and A. Vattani, "Fast greedy algorithms in mapreduce and streaming," *ACM Transactions on Parallel Computing*, vol. 2, no. 3, p. 14, 2015.

[5] B. Bahmani, B. Moseley, A. Vattani, R. Kumar, and S. Vassilvitskii, "Scalable k-means++," *VLDB*, 2012.

[6] S. Suri and S. Vassilvitskii, "Counting triangles and the curse of the last reducer," in *WWW*, 2011.

[7] C.-T. Chu, S. K. Kim, Y.-A. Lin, Y. Yu, G. Bradski, A. Y. Ng, and K. Olukotun, "Map-reduce for machine learning on multicore," in *NIPS*, 2006.

[8] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: cluster computing with working sets." in *HotCloud*, 2010.

[9] B. Recht, C. Re, S. Wright, and F. Niu, "Hogwild: A lock-free approach to parallelizing stochastic gradient descent," in *NIPS*, 2011.

[10] M. Zinkevich, M. Weimer, L. Li, and A. J. Smola, "Parallelized stochastic gradient descent," in *NIPS*, 2010.

[11] M. Li, L. Zhou, Z. Yang, A. Li, F. Xia, D. G. Andersen, and A. Smola, "Parameter server for distributed machine learning," in *NIPS Workshop*, 2013.

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

[13] T. Yang, "Trading computation for communication: Distributed stochastic dual coordinate ascent," in *NIPS*, 2013.

[14] S. Boyd, N. Parikh, E. Chu, B. Peleato, and J. Eckstein, "Distributed optimization and statistical learning via the alternating direction method of multipliers," *Foundations and Trends in Machine Learning*, vol. 3, no. 1, pp. 1–122, 2011.

[15] M. Frank and P. Wolfe, "An Algorithm for Quadratic Programming," *Naval Research Logistics Quarterly*, vol. 3, no. 1-2, pp. 95–110, 1956.

[16] M. Dudik, Z. Harchaoui, and J. Malick, "Lifted coordinate descent for learning with trace-norm regularization," in *AISTATS*, 2012.

[17] E. Hazan and S. Kale, "Projection-free online learning," in *ICML*, 2012.

[18] Y. Ying and P. Li, "Distance metric learning with eigenvalue optimization," *Journal of Machine Learning Research*, vol. 13, no. Jan, pp. 1–26, 2012.

[19] A. Joulin, K. Tang, and L. Fei-Fei, "Efficient image and video co-localization with Frank-Wolfe algorithm," in *ECVV*, 2014.

[20] K. L. Clarkson, "Coresets, sparse greedy approximation, and the Frank-Wolfe algorithm," *ACM Trans. Algorithms*, vol. 6, no. 4, pp. 63:1–63:30, Sep. 2010.

[21] M. Jaggi, "Revisiting Frank-Wolfe: Projection-free sparse convex optimization." in *ICML*, 2013.

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

[23] S. Boyd and L. Vandenberghe, *Convex Optimization*. New York, NY, USA: Cambridge University Press, 2004.

[24] D. P. Bertsekas, *Nonlinear programming*. Athena scientific Belmont, 1999.

[25] J. Kiefer and J. Wolfowitz, “Stochastic estimation of the maximum of a regression function,” *The Annals of Mathematical Statistics*, pp. 462–466, 1952.

[26] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su, “Scaling distributed machine learning with the parameter server.” in *OSDI*, vol. 14, 2014, pp. 583–598.

[27] K.-W. Chang, C. Hsieh, C. Lin, S. Keerthi, and S. Sundararajan, “A dual coordinate descent method for large-scale linear svm,” in *Proc. Intl. Conf. on Machine Learning (ICML)*, 2008.

[28] S. Shalev-Shwartz and T. Zhang, “Stochastic dual coordinate ascent methods for regularized loss minimization,” *Journal of Machine Learning Research*, vol. 14, no. Feb, pp. 567–599, 2013.

[29] Z.-Q. Luo and P. Tseng, “On the convergence of the coordinate descent method for convex differentiable minimization,” *Journal of Optimization Theory and Applications*, vol. 72, no. 1, pp. 7–35, 1992.

[30] A. Tewari, P. K. Ravikumar, and I. S. Dhillon, “Greedy algorithms for structurally constrained high dimensional problems,” in *NIPS*, 2011.

[31] I. W. Tsang, J. T. Kwok, and P.-M. Cheung, “Core vector machines: Fast svm training on very large data sets,” *Journal of Machine Learning Research*, vol. 6, no. Apr, pp. 363–392, 2005.

[32] P. Wolfe, “Convergence theory in nonlinear programming,” *Integer and nonlinear programming*, pp. 1–36, 1970.

[33] A. Beck and S. Shtern, “Linearly convergent away-step conditional gradient for non-strongly convex functions,” *Mathematical Programming*, pp. 1–27, 2015.

[34] D. Garber and E. Hazan, “A linearly convergent variant of the conditional gradient algorithm under strong convexity, with applications to online and stochastic optimization,” *SIAM Journal on Optimization*, vol. 26, no. 3, pp. 1493–1528, 2016.

[35] S. Lacoste-Julien and M. Jaggi, “On the global linear convergence of Frank-Wolfe optimization variants,” in *NIPS*, 2015.

[36] Z. Harchaoui, M. Douze, M. Paulin, M. Dudik, and J. Malick, “Large-scale image classification with trace-norm regularization,” in *CVPR*, 2012.

[37] D. Garber and O. Meshi, “Linear-memory and decomposition-invariant linearly convergent conditional gradient algorithm for structured polytopes,” in *Advances in Neural Information Processing Systems*, 2016, pp. 1001–1009.

[38] G. Lan and Y. Zhou, “Conditional gradient sliding for convex optimization,” *SIAM Journal on Optimization*, vol. 26, no. 2, pp. 1379–1409, 2016.

[39] E. Hazan and H. Luo, “Variance-reduced and projection-free stochastic optimization,” in *ICML*, 2016.

[40] S. J. Reddi, S. Sra, B. Póczós, and A. Smola, “Stochastic Frank-Wolfe methods for non-convex optimization,” in *Allerton*, 2016.

[41] J. C. Dunn, “Rates of convergence for conditional gradient algorithms near singular and nonsingular extremals,” *SIAM Journal on Control and Optimization*, vol. 17, no. 2, pp. 187–211, 1979.

[42] M. Canon and C. Cullum, “A tight upper bound on the rate of convergence of frank-wolfe algorithm,” *SIAM Journal on Control*, vol. 6, no. 4, pp. 509–516, 1968.

[43] J. Guélat and P. Marcotte, “Some comments on Wolfe’s away step,” *Mathematical Programming*, vol. 35, no. 1, pp. 110–119, 1986.

[44] P. Kumar and E. A. Yıldırım, “A linearly convergent linear-time first-order algorithm for support vector classification with a core set result,” *INFORMS Journal on Computing*, vol. 23, no. 3, pp. 377–391, 2011.

[45] R. Ñanculef, E. Frandi, C. Sartori, and H. Allende, “A novel frank–wolfe algorithm. analysis and applications to large-scale svm training,” *Information Sciences*, vol. 285, pp. 66–99, 2014.

[46] S. Damla Ahipasaoglu, P. Sun, and M. J. Todd, “Linear convergence of a modified frank–wolfe algorithm for computing minimum-volume enclosing ellipsoids,” *Optimisation Methods and Software*, vol. 23, no. 1, pp. 5–19, 2008.

[47] R. Johnson and T. Zhang, “Accelerating stochastic gradient descent using predictive variance reduction,” in *Advances in neural information processing systems*, 2013, pp. 315–323.

[48] M. Mahdavi, L. Zhang, and R. Jin, “Mixed optimization for smooth functions,” in *Advances in Neural Information Processing Systems*, 2013, pp. 674–682.

[49] S. Lacoste-Julien, M. Jaggi, M. Schmidt, and P. Pletscher, “Block-coordinate frank-wolfe optimization for structural svms,” in *Proceedings of ICML*, 2013.

[50] A. Osokin, J.-B. Alayrac, I. Lukasewitz, P. K. Dokania, and S. Lacoste-Julien, “Minding the Gaps for Block Frank-Wolfe Optimization of Structured SVMs,” in *ICML*, 2016.

[51] 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.

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

[53] S. Sur, M. J. Koop, and D. K. Panda, “High-performance and scalable mpi over infiniband with reduced memory usage: an in-depth performance analysis,” in *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*. ACM, 2006, p. 105.

[54] J. Sherman and W. J. Morrison, “Adjustment of an inverse matrix corresponding to a change in one element of a given matrix,” *The Annals of Mathematical Statistics*, vol. 21, no. 1, pp. 124–127, 1950.

[55] A. Y. Ng, “Feature selection, l 1 vs. l 2 regularization, and rotational invariance,” in *ICML*, 2004.

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

[57] P. Shah, B. N. Bhaskar, G. Tang, and B. Recht, “Linear system identification via atomic norm regularization,” in *CDC*, 2012.

[58] S. Chen and A. Banerjee, “Structured estimation with atomic norms: General bounds and applications,” in *NIPS*, 2015.

[59] V. Chandrasekaran, B. Recht, P. A. Parrilo, and A. S. Willsky, “The convex geometry of linear inverse problems,” *Foundations of Computational Mathematics*, vol. 12, no. 6, pp. 805–849, 2012.

[60] F. M. Harper and J. A. Konstan, “The movielens datasets: History and context,” *ACM Trans. Interact. Intell. Syst.*, vol. 5, no. 4, pp. 19:1–19:19, Dec. 2015.

[61] M. Lichman, "UCI machine learning repository," 2013.

[62] Y. Koren, R. Bell, and C. Volinsky, "Matrix factorization techniques for recommender systems," *Computer*, vol. 42, no. 8, 2009.

[63] P. R. Goundan and A. S. Schulz, "Revisiting the greedy approach to submodular set function maximization," *Optimization online*, pp. 1–25, 2007.

[64] A. Krause and C. E. Guestrin, "Near-optimal nonmyopic value of information in graphical models," *arXiv preprint arXiv:1207.1394*, 2012.

[65] A. Krause and V. Cevher, "Submodular dictionary selection for sparse representation," in *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, 2010, pp. 567–574.

[66] A. Das and D. Kempe, "Submodular meets spectral: Greedy algorithms for subset selection, sparse approximation and dictionary selection," *arXiv preprint arXiv:1102.3975*, 2011.

[67] H. Lin and J. Bilmes, "A class of submodular functions for document summarization," in *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies-Volume 1*. Association for Computational Linguistics, 2011, pp. 510–520.

[68] B. Mirzasoleiman, A. Karbasi, R. Sarkar, and A. Krause, "Distributed submodular maximization: Identifying representative elements in massive data," in *Advances in Neural Information Processing Systems*, 2013, pp. 2049–2057.

[69] G. Calinescu, C. Chekuri, M. Pál, and J. Vondrák, "Maximizing a submodular set function subject to a matroid constraint," in *International Conference on Integer Programming and Combinatorial Optimization*. Springer, 2007, pp. 182–196.

[70] G. L. Nemhauser, L. A. Wolsey, and M. L. Fisher, "An analysis of approximations for maximizing submodular set functions?i," *Mathematical Programming*, vol. 14, no. 1, pp. 265–294, 1978.

[71] Y. Bian, B. Mirzasoleiman, J. M. Buhmann, and A. Krause, "Guaranteed non-convex optimization: Submodular maximization over continuous domains," in *AISTATS*, 2017.

[72] J. Vondrák, "Optimal approximation for the submodular welfare problem in the value oracle model," in *Proceedings of the fortieth annual ACM symposium on Theory of computing*. ACM, 2008, pp. 67–74.

[73] G. Calinescu, C. Chekuri, M. Pál, and J. Vondrák, "Maximizing a monotone submodular function subject to a matroid constraint," *SIAM Journal on Computing*, vol. 40, no. 6, pp. 1740–1766, 2011.

[74] G. L. Nemhauser and L. A. Wolsey, "Best algorithms for approximating the maximum of a submodular set function," *Mathematics of operations research*, vol. 3, no. 3, pp. 177–188, 1978.

[75] F. Bach, "Submodular functions: from discrete to continous domains," *arXiv preprint arXiv:1511.00394*, 2015.

[76] D. M. Topkis, "Minimizing a submodular function on a lattice," *Operations research*, vol. 26, no. 2, pp. 305–321, 1978.

[77] L. Lovász, "Submodular functions and convexity," in *Mathematical Programming The State of the Art*. Springer, 1983, pp. 235–257.