TECHNICAL UNIVERSITY OF CRETE
ELECTRICAL AND COMPUTER ENGINEERING DEPARTMENT

Distributed Algorithms for Convex Optimization

by

George Lykoudis

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DIPLOMA DEGREE OF

ELECTRICAL AND COMPUTER ENGINEERING

July 2017

THESIS COMMITTEE

Professor Athanasios P. Liavas, *Thesis Supervisor*
Associate Professor Georgios N. Karystinos
Assistant Professor Vasilis Samoladas

Abstract

We consider convex optimization problems whose cost function can be expressed as the sum of P convex functions. With each function, we associate a convex set and assume that the optimal vector lies in the intersection of these sets. We associate each problem with a network, where each node has its private cost function and private constraint set. We present the Distributed Alternating Direction Method of Multipliers (D-ADMM) for the solution of this problem, and demonstrate it by solving problems for the areas of signal processing and control. We use the Message Passing Interface (MPI) for the development of parallel implementations of the D-ADMM; we describe in detail three variations. We test the efficiency of the implementations in extensive numerical experiments.

Acknowledgements

I would like to thank my friends and my family for supporting me during my studies. Also, I would like to thank my supervisor, Professor Athanasios Liavas, for his guidance throughout this work.

Table of Contents

List of Figures

List of Abbreviations

KKT	Karush Kuhn Tucker
MM	Method of Multipliers
ADMM	Alternating Direction Method of Multipliers
QP	Quadratic Programming
LP	Linear Programming
D-ADMM	Distributed Alternating Direction Method of Multipliers
MPI	Message Passing Interface
MIMD	Multiple Instruction-Multiple Data

Chapter 1

Introduction

1.1 Motivation

The expansion of size and complexity of nowadays datasets, created the need to solve problems with large number of features in relatively small amount of time. In order to solve these problems, distributed algorithms were developed, which work in a decentralized way for various reasons, such as privacy or memory issues. The work of Joãh F. C. Mota, Joãh M. F.Xavier, Pedro M. Q. Aguiar and Markus Püschel in [1] and [2] was the main motivation for this thesis and a great help for understanding these methods. We study one of these algorithms called Distributed Alternating Direction Method of Multipliers (D-ADMM). It is an algorithm built for solving a class of optimization problems, known as separable optimization problems, in networks of interconnected nodes. In separable optimization problems there is a private cost function and a private constraint set at each node.

Our goal is to present this algorithm so that the reader will be able to understand and apply it to a problem he is facing. This is achieved by presenting the framework and different examples in network problem formulations.

The results concerning the speedup attained by the Message Passing Interface (MPI) implementations on a multi-core system.

1.2 Notation

Capital (small) bold face letters will denote matrices (column vectors); small letters will be scalars; $[\cdot]_{ij}$ ($[\cdot]_i$) for the (i,j)-th (i-th) entry of a matrix(vector); $(\cdot)^T$ denotes transposition; $(\cdot)^{-1}$ inverse of a matrix; $\mathbf{I}_N$ denotes $N \times N$ identity matrix; $\mathbf{1}_N$ denotes a $N \times 1$ vector of all ones; $\mathbf{0}_N$ denotes a $N \times 1$ zero vector; $\|\cdot\|_2$ the Euclidean norm; $\|\cdot\|_1$ the l_1 norm; $|\cdot|$ the cardinality of a set; $\nabla_{\mathbf{x}} f(\cdot)$ the gradient of function f with respect to vector $\mathbf{x}$.

1.3 Thesis Outline

The thesis is organized as follows:

- In chapter 2, is an introduction to basic concepts of convex optimization.
- In chapter 3, presents the ADMM and D-ADMM algorithms.
- In chapter 4, presents Message Passing Interface
- In chapter 5, we implement the algorithms presented in chapter 3, developed applications based on chapter 4 and present the results from their combination.
- Finally, in chapter 6 we end our thesis with the conclusion.

Chapter 2

Convex Optimization

Convex optimization refers to the minimization of a convex objective function subject to convex constraints. Convex optimization techniques are important in engineering applications, like parameter estimation and signal processing, communications and networks, electronic circuit design, statistics, finance, etc. The most basic advantage of casting an optimization problem into a convex optimization problem is that, due to the convex nature of the feasible set of the problem, any local optimum is also global optimum. Therefore, algorithms developed to solve convex optimization problems that exploit such properties are reliable enough, much more efficient and also fast.

2.1 Basic Optimization Concepts

In this section, our main goal is to help the reader to develop a working knowledge of convex optimization, before dealing with mathematical problems for engineering applications. The review of the optimization concept, is based on the work of Stephen Boyd and Lieven Vandenberghe [3] and course notes of thesis advisor A. Liavas [4].

2.1.1 Convex Sets

A set $\mathbb{C} \subseteq \mathbb{R}^n$ is called convex if, for any set of points $\mathbf{x}, \mathbf{y} \in \mathbb{C}$, the line segment that joins them also lies in $\mathbb{C}$. So, for $\mathbf{x}, \mathbf{y} \in \mathbb{C}$ and $\forall \theta \in [0, 1]$,

$$\theta \mathbf{x} + (1 - \theta)\mathbf{y} \in \mathbb{C}.$$

Well-known convex sets are cones, ellipsoids, polyhedral sets, and so on. In general, a convex set must be compact, meaning that it should have a solid body, containing no holes. Roughly speaking, a set is convex if every point in the set can be seen by every other point, while both points belong in the set.

Operations between convex sets that preserve convexity and will be of use in the following chapters are the intersection of any number (finite or infinite) sets, and the projection of a convex set onto some of its coordinates.

2.1.2 Convex Functions

A function $f : \mathbf{dom}f \subseteq \mathbb{R}^n \to \mathbb{R}$, is convex if $\mathbf{dom}f$ is convex and if for any points $\mathbf{x}, \mathbf{y} \in \mathbf{dom}f$ and $\forall\theta \in [0, 1]$ applies,

$$f(\theta\mathbf{x} + (1-\theta)\mathbf{y}) \leq \theta f(\mathbf{x}) + (1-\theta)f(\mathbf{y}). \tag{2.1}$$

Geometrically, this means that the line segment joining $(\mathbf{x}, f(\mathbf{x}))$ and $(\mathbf{y}, f(\mathbf{y}))$ is always above the graph of function f, as seen in the following figure. A function f is strictly convex, if strict inequality holds in (2.1) whenever $\mathbf{x} \neq \mathbf{y}$ and $0 < \theta < 1$. We say f is concave if $-f$ is convex, and strictly concave if $-f$ is strictly convex.

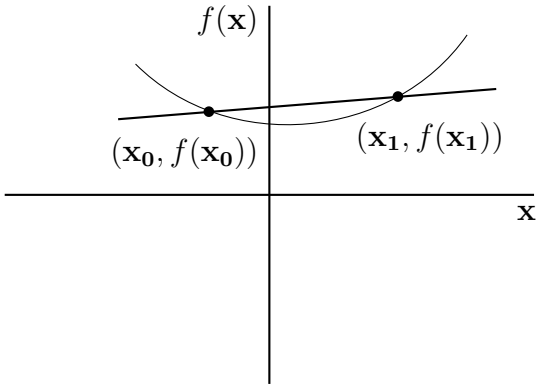


Figure 2.1: Graph of a convex function.

There are many examples of convex functions, including functions on $\mathbb{R}$ with variable x as, $|x|, e^x, x^2$, as well as functions with variable $\mathbf{x} \in \mathbb{R}^n$, like, $\mathbf{a}^T\mathbf{x}+\mathbf{b}$, $\|\mathbf{A}\mathbf{x}\|^2$, where $\mathbf{A}$, $\mathbf{a}$, and $\mathbf{b}$ are given matrix and vectors respectively.

2.1.3 Gradient and Subgradient

If $f : \mathbb{R}^n \to \mathbb{R}$ is differentiable, then the function $\nabla f : \mathbb{R}^n \to \mathbb{R}^n$ defined as

$$\nabla f(\mathbf{x}) = \begin{bmatrix} \frac{\partial f}{\partial x_1}(\mathbf{x}) \\ \vdots \\ \frac{\partial f}{\partial x_n}(\mathbf{x}) \end{bmatrix},$$

is called the gradient of f at $\mathbf{x}$.

An important property of a convex and differentiable function $f : \mathbf{dom}f \to \mathbb{R}$, with $\mathbf{dom}f \subseteq \mathbb{R}^n$, is that, for every $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$, we have

$$f(\mathbf{y}) \geq f(\mathbf{x}) + \nabla f(\mathbf{x})^T(\mathbf{y} - \mathbf{x}).$$

That means that first-order Taylor approximation serves as a global underestimator of f. Furthermore, if f is twice differentiable, then f is convex if and only, if $\mathbf{dom}f$ is convex and its Hessian is positive semidefinite $\forall \mathbf{x} \in \mathbf{dom}f$

$$\nabla^2 f(\mathbf{x}) \geq \mathbf{0}.$$

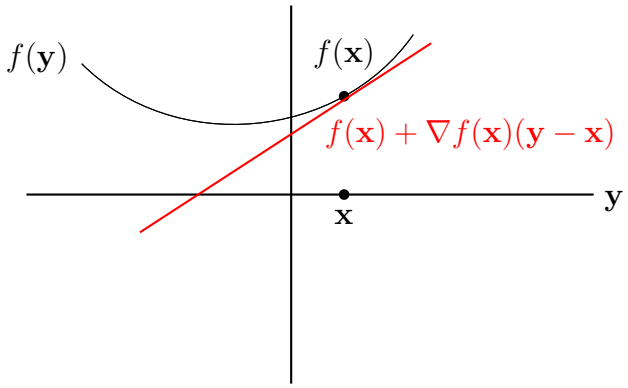


Figure 2.2: Important property of gradient.

When the function of interest is not differentiable everywhere, its gradient can not be computed at the non-smooth points. At these points, we use the subgradient of the function. A subgradient of a function f, at $\mathbf{x}$, is any vector $\mathbf{g}$ such that

$$f(\mathbf{y}) \geq f(\mathbf{x}) + \mathbf{g}^T(\mathbf{y} - \mathbf{x}), \quad \forall \mathbf{y} \in \mathbf{dom}f.$$

The set of all subgradients of f, at point $\mathbf{x}$, is called the subdifferential of f at $\mathbf{x}$ and is denoted as

$$\partial f(\mathbf{x}) = \{\mathbf{g} \mid f(\mathbf{y}) \geq f(\mathbf{x}) + \mathbf{g}^T(\mathbf{y} - \mathbf{x}), \quad \forall \mathbf{y} \in \mathbf{dom}f\}.$$

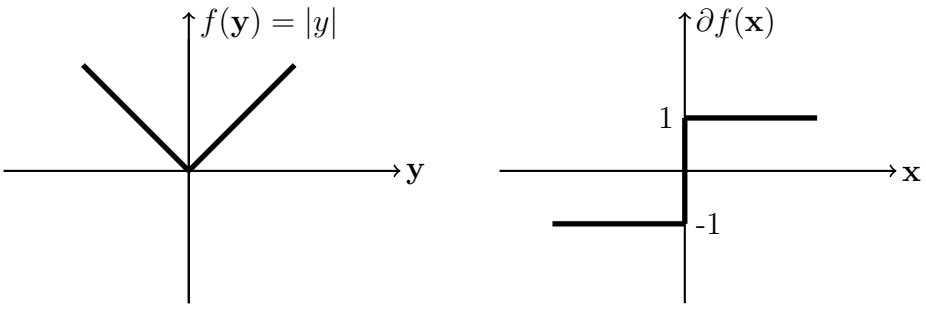


Figure 2.3: The absolute value function (left) and its subdifferential $\partial f(\mathbf{x})$ as a multivalued function of $\mathbf{x}$ (right).

2.1.4 Convex Optimization Problem

A generic convex optimization problem is defined as

$$\begin{aligned} \underset{\mathbf{x}}{\text{minimize}} \quad & f_0(\mathbf{x}) \\ \text{subject to} \quad & f_i(\mathbf{x}) \leq 0,\ i = 1, \ldots, m, \\ & h_i(\mathbf{x}) = 0,\ i = 1, \ldots, p, \end{aligned} \tag{2.2}$$

where vector $\mathbf{x} \in \mathbb{R}^n$ is called optimization variable of the problem and function f_0 the objective (or cost) function. The inequalities $f_i(\mathbf{x}) \leq 0$ are called the inequality constraints, and the corresponding functions $f_i : \mathbb{R}^n \to \mathbb{R}$ are the inequality constraint functions. All functions f_i, for $i = 0, \ldots, m$, are convex. The equations $h_i(\mathbf{x}) = 0$ are called the equality constraints and the functions $h_i : \mathbb{R}^n \to \mathbb{R}$ are the equality constraint functions, which are affine functions, that is $h_i(\mathbf{x}) = \mathbf{a}_i^T\mathbf{x} - b_i$. In case of non-existence of contraints, the problem is called unconstrained.

The set of points for which the objective and all constraint functions are defined is called the domain $\mathcal{D}$ of the convex optimization problem (2.2)

$$\mathcal{D} := \bigcap_{i=0}^{m} \mathbf{dom} f_i \cap \bigcap_{i=1}^{p} \mathbf{dom} h_i. \tag{2.3}$$

A point $\mathbf{x}$ is said to be feasible if $\mathbf{x} \in \mathcal{D}$ and satisfies all inequality and equality constraints. Problem (2.2) is said to be feasible if there exists at least one feasible point, and infeasible otherwise. The set of all feasible points is called the feasible set and is defined as

$$\mathbb{X} \ := \ \{\mathbf{x} \in \mathcal{D} \mid f_i(\mathbf{x}) \leq 0,\ i = 1, \ldots, m,\ h_i(\mathbf{x}) = 0,\ i = 1, \ldots, p\}. \tag{2.4}$$

The feasible set of a convex optimization problem is convex, since it is the intersection of the domain $\cap_{i=0}^{m} \mathbf{dom} f_i$, which is a convex set, with m sublevel sets and p hyperplanes. Thus, in a convex optimization problem, we minimize a convex objective function over a convex set.

The optimal point of **(2.2)** is defined as

$$p^* = \inf\{f_0(\mathbf{x}) \mid \mathbf{x} \in \mathbb{X}\}.$$

A feasible solution $\mathbf{x}^*$ is said to be globally optimal if $f_0(\mathbf{x}^*) \leq f_0(\mathbf{x})$, $\forall \mathbf{x}^* \in \mathbb{X}$.

2.1.5 Lagrangian Duality

While trying to solve an optimization problem, we should keep in mind that, there is always another problem that is closely connected to the first. The original problem is called primal and the second one, the Lagrange dual. Under some conditions (i.e convexity), those two problems have the same optimal point. So, dual problem can be of use in some cases.

In order to define the Langrangian function of a convex optimization problem, we take the constraints in (2.2) into account by augmenting the objective function with a weighted sum of the constraint functions. We form the Lagrangian function $L : \mathcal{D} \times \mathbb{R}^m \times \mathbb{R}^p \to \mathbb{R}$

$$L(\mathbf{x}, \boldsymbol{\lambda}, \mathbf{v}) = f_0(\mathbf{x}) + \sum_{i=1}^{m} \lambda_i f_i(\mathbf{x}) + \sum_{i=1}^{p} v_i h_i(\mathbf{x}). \tag{2.5}$$

We refer to λ_i as the Lagrange multiplier associated with the i-th inequality constraint $f_i(\mathbf{x}) \leq 0$. Similarly, we refer to v_i as the Lagrange multiplier associated with the i-th equality constraint $h_i(\mathbf{x}) = 0$. Vectors $\boldsymbol{\lambda} \in \mathbb{R}^m$ and $\mathbf{v} \in \mathbb{R}^p$ are called dual variables or Lagrange multiplier vectors, associated with problem (2.2).

2.1.6 Lagrange Dual Function

The Lagrange dual function (or just dual function) $g : \mathbb{R}^m \times \mathbb{R}^p \to \mathbb{R}$ is defined as the minimum value of the Lagrangian over $\mathbf{x}$: for $\boldsymbol{\lambda} \in \mathbb{R}^m$, $\mathbf{v} \in \mathbb{R}^p$,

$$g(\boldsymbol{\lambda}, \mathbf{v}) = \inf_{\mathbf{x} \in \mathcal{D}} L(\mathbf{x}, \boldsymbol{\lambda}, \mathbf{v}). \tag{2.6}$$

When the Lagrangian is unbounded from below at $\mathbf{x}$, the dual function takes the value $-\infty$. Since the dual function is the pointwise infimum of a family of affine functions of $(\boldsymbol{\lambda}, \mathbf{v})$, it is concave, even when problem (2.2) is not convex. An important property of the dual function is that it yields lower bounds on the optimal value p_* of the problem (2.2): For any $\boldsymbol{\lambda} \geq 0$ and any $\mathbf{v}$, we have

$$g(\boldsymbol{\lambda}, \mathbf{v}) \leq p_*.$$

2.1.7 Dual Problem

The dual function provides a lower bound that depends on parameters $\boldsymbol{\lambda}, \mathbf{v}$. The best lower bound that can be obtained from the dual function is computed by the optimization problem

$$\begin{aligned} \underset{\boldsymbol{\lambda},\mathbf{v}}{\text{maximize}} \quad & g(\boldsymbol{\lambda}, \mathbf{v}), \\ \text{subject to} \quad & \boldsymbol{\lambda} \geq \mathbf{0}. \end{aligned} \tag{2.7}$$

is called Lagrange dual problem associated with the problem (2.2) In this context, the original problem (2.2) is sometimes called the primal problem. Also, the pairs $(\boldsymbol{\lambda}, \mathbf{v})$ are known as dual feasible, if $\boldsymbol{\lambda} \geq 0$ and dual function is finite. We refer to $(\boldsymbol{\lambda}_*, \mathbf{v}_*)$ as dual optimal or optimal Lagrange multipliers if they are optimal for the problem (2.7).

The dual problem (2.7) is a convex optimization problem, since the objective to be maximized is concave, and the constraint is convex, regardless of the convexity of the primal problem (2.2).

The optimal value of the Lagrange dual problem, which we denote d_*, is, by definition, the best lower bound on p_* that can be obtained from the Lagrange dual function. In particular, we have the simple but important inequality $d_* \leq p_*$, which holds even if the original problem is not convex. This property is called weak duality.

We refer to the difference $p_* - d_*$ as the optimal duality gap of the original problem, since it gives the gap between the optimal value of the primal problem and the best lower bound on it that can be obtained from the Lagrange dual function. The optimal duality gap is always nonnegative. If the equality $d_* = p_*$ holds, i.e., the optimal duality gap is zero, then we say that strong duality holds. This means that the best bound that can be obtained from the Lagrange dual function is tight. Strong duality does not, in general, hold. But, if the primal problem (2.2) is convex, we usually (but not always) have strong duality.

2.1.8 Optimality Conditions

When a convex optimization problem is unconstrained and the objective function f_0 is differentiable for all $\mathbf{x} \in \mathbf{dom} f_0$, then $\mathbf{x}_0$ is optimal if and only if

$$\nabla f_0(\mathbf{x}_0) = \mathbf{0}.$$

For any optimization problem with differentiable objective and constraint functions for which strong duality obtains, any pair of primal and dual optimal points, $\mathbf{x}_*$ and $(\boldsymbol{\lambda}_*, \mathbf{v}_*)$, must satisfy the Karush Kuhn Tucker (KKT) conditions

$$f_i(\mathbf{x}^*) \leq 0, \text{ for } i = 1, \ldots, m, \tag{2.8}$$

$$h_i(\mathbf{x}^*) = 0, \text{ for } i = 1, \ldots, p, \tag{2.9}$$

$$\lambda_i^* \geq 0, \text{ for } i = 1, \ldots, m, \tag{2.10}$$

$$\lambda_i^* f_i(\mathbf{x}^*) = 0, \text{ for } i = 1, \ldots, m, \tag{2.11}$$

and

$$\nabla f_0(\mathbf{x}) + \sum_{i=1}^{m} \lambda_i^* \nabla f_i(\mathbf{x}) + \sum_{i=1}^{p} v_i^* \nabla h_i(\mathbf{x}) = \mathbf{0}. \tag{2.12}$$

Notice that the first two conditions guarantee primal feasibility of $\mathbf{x}^*$, condition (2.10) guarantees dual feasibility, condition (2.11) signifies the complementary slackness for the primal and dual inequality constraint pairs, $f_i(\mathbf{x}) \leq 0$ and $\lambda_i \geq 0$, while the final condition is equivalent to $\nabla L(\mathbf{x}^*, \boldsymbol{\lambda}, \mathbf{v}^*) = \mathbf{0}$.

In general, the KKT conditions are necessary but not sufficient for optimality. However, for convex optimization problems like (2.2), the KKT conditions are also sufficient. In other words, if f_i are convex and h_i are affine, and for any $\mathbf{x}^*, (\boldsymbol{\lambda}, \mathbf{v})$ points that satisfy the KKT conditions then $\mathbf{x}^*, (\boldsymbol{\lambda}, \mathbf{v})$ are primal and dual optimal, with zero duality gap.

Chapter 3

ADMM and D-ADMM

The enormous growth in size and complexity of modern datasets required for algorithms that could solve problems with huge numbers of features or training examples. Alternating Direction Method of Multipliers (ADMM) is a well suited algorithm for distributed convex optimization and in particular to problems arising in applied statistics and machine learning [5]. Furthermore, D-ADMM is an algorithm based on the extended ADMM [6], for solving separable optimization problems from signal processing and control, in networks of interconnected nodes, as average consensus and compressed sensing, [2].

3.1 Dual Ascent

In the first two sections, we will do a brief review to dual ascent algorithm and method of multipliers, which are precursors of ADMM.

Consider the equality-constrained convex optimization problem,

$$\begin{aligned} &\underset{\mathbf{x}}{\text{minimize}} \quad f(\mathbf{x}), \\ &\text{subject to} \quad \mathbf{A}\mathbf{x} = \mathbf{b}, \end{aligned} \tag{3.1}$$

with $\mathbf{x} \in \mathbb{R}^n$, $\mathbf{A} \in \mathbb{R}^{m \times n}$ and $f : \mathbb{R}^n \to \mathbb{R}$ is convex. As seen in (2.5), the Lagrangian for problem (3.1) is

$$L(\mathbf{x}, \mathbf{y}) = f(\mathbf{x}) + \mathbf{y}^T(\mathbf{A}\mathbf{x} - \mathbf{b}),$$

and the dual function is

$$g(\mathbf{y}) = \inf_{\mathbf{x}} L(\mathbf{x}, \mathbf{y}) = -f^*(-\mathbf{A}^T\mathbf{y}) - \mathbf{b}^T\mathbf{y},$$

where $\mathbf{y}$ is the dual variable or Lagrange multiplier and f^* is the convex conjugate of f. The dual problem is

$$\underset{\mathbf{y}}{\text{maximize}} \quad g(\mathbf{y})$$

with variable $\mathbf{y} \in \mathbb{R}^n$. Optimal values of the primal and dual problems are equal, under the assumption that strong duality holds. Primal optimal point $\mathbf{x}^*$ can be recovered from a dual optimal point $\mathbf{y}^*$ as

$$\mathbf{x}^* = \underset{\mathbf{x}}{\operatorname{argmin}}\ L(\mathbf{x}, \mathbf{y}^*),$$

in the case that only one minimizer of $L(\mathbf{x}, \mathbf{y}^*)$ exists.

The Dual Ascent method is an iterative technique for the solution of (3.1), that can be described as follows. Assuming that dual function g is differentiable and that, at the k-th iteration, we have computed dual variable $\mathbf{y}^k$, the updates of variables $\mathbf{x}, \mathbf{y}$ are given by

$$\mathbf{x}^{k+1} := \underset{\mathbf{x}}{\operatorname{argmin}}\ L(\mathbf{x}, \mathbf{y}^k), \tag{3.2}$$

$$\mathbf{y}^{k+1} := \mathbf{y}^k + \alpha^k(\mathbf{A}\mathbf{x}^{k+1} - \mathbf{b}), \tag{3.3}$$

where $\alpha^k > 0$ is a step size. Equation (3.2) is the $\mathbf{x}$-minimization step, while equation (3.3) is a dual variable update. In our case, $g(\mathbf{y}^k) = f(\mathbf{x}^{k+1}) + (\mathbf{y}^k)^T(\mathbf{A}\mathbf{x}^{k+1} - \mathbf{b})$ and $\nabla g(\mathbf{y}^k) = \mathbf{A}\mathbf{x}^{k+1} - \mathbf{b}$.

The dual variable $\mathbf{y}$ can be described as a vector of prices. Then the $\mathbf{y}$-update is called a price update or price adjustment step. This algorithm is called dual ascent because, with proper choice of α^k , the dual function increases in each step, i.e., $g(\mathbf{y}^{k+1}) > g(\mathbf{y}^k)$.

The dual ascent method can be used even in cases where g is not differentiable. In these cases, the residual $\mathbf{A}\mathbf{x}^{k+1} - \mathbf{b}$ is the negative of a subgradient of $g(\mathbf{y}^k)$. These cases require a different choice of α^k than when g is differentiable, and convergence is not monotone; it is often the case that $g(\mathbf{y}^{k+1}) \not> g(\mathbf{y}^k)$. In these cases, the algorithm is called the dual subgradient method. If α^k is chosen appropriately and several other assumptions hold, then $\mathbf{x}^k$ converges to an optimal point and $\mathbf{y}^k$ converges to an optimal dual point. However, these assumptions do not hold in many applications, so dual ascent often cannot be used.

The advantage of using dual ascent method is that there are cases in which can lead to a decentralized algorithm. For example, we have a separable objective function f (with respect to a partition, or splitting of the variable into subvectors),

$$f(\mathbf{x}) = \sum_{i=1}^{N} f_i(\mathbf{x}_i)$$

where $\mathbf{x} = (\mathbf{x}_1, \ldots, \mathbf{x}_N)$ and the variables $\mathbf{x}_i \in \mathbb{R}^{n_i}$ are subvectors of $\mathbf{x}$.

Partitioning, also the matrix $\mathbf{A}$ as

$$\mathbf{A} = [\mathbf{A}_1 \dots \mathbf{A}_N],$$

so that $\mathbf{A}\mathbf{x} = \sum_{i=1}^{N} \mathbf{A}_i \mathbf{x}_i$, the Lagrangian can be written as

$$L(\mathbf{x}, \mathbf{y}) = \sum_{i=1}^{N} L_i(\mathbf{x}_i, \mathbf{y}) = \sum_{i=1}^{N} (f_i(\mathbf{x}_i) + \mathbf{y}^T \mathbf{A}_i \mathbf{x}_i - \frac{1}{N} \mathbf{y}^T \mathbf{b}),$$

which is also separable in $\mathbf{x}$. This means that the $\mathbf{x}$-minimization step (3.2) splits into N separable problems that can be solved in parallel.
Explicitly, the algorithm is

$$\mathbf{x}_i^{k+1} := \underset{\mathbf{x}_i}{\operatorname{argmin}}\ L_i(\mathbf{x}_i, \mathbf{y}^k), \tag{3.4}$$

$$\mathbf{y}^{k+1} := \mathbf{y}^k + \alpha^k (\mathbf{A}\mathbf{x}^{k+1} - \mathbf{b}), \tag{3.5}$$

From (3.4) we can see that, for each $i = 1, \dots, N$, the $\mathbf{x}$-minimization step can be carried independently. To this form of dual ascent method, we refer as dual decomposition.

3.2 Augmented Lagrangians and Method of Multipliers

3.2.1 Augmented Lagrangian

Augmented Lagrangian methods were developed for two reasons. First, to bring robustness to the dual ascent method and second to yield convergence without making any assumptions for f, like strict convexity or finiteness. The augmented Lagrangian for problem (3.1) is

$$L_\rho(\mathbf{x}, \mathbf{y}) = f(\mathbf{x}) + \mathbf{y}^T(\mathbf{A}\mathbf{x} - \mathbf{b}) + \frac{\rho}{2} \|\mathbf{A}\mathbf{x} - \mathbf{b}\|_2^2, \tag{3.6}$$

where $\rho > 0$ is called the penalty parameter. In case $\rho = 0$, we get the standard Lagrangian for the same problem. The augmented Lagrangian is associated with the problem

$$\begin{aligned} &\underset{\mathbf{x}}{\text{minimize}} \quad f(\mathbf{x}) + \frac{\rho}{2} \|\mathbf{A}\mathbf{x} - \mathbf{b}\|_2^2, \\ &\text{subject to} \quad \mathbf{A}\mathbf{x} = \mathbf{b}. \end{aligned} \tag{3.7}$$

It is clear that problem (3.7) is equivalent to (3.1), since for any feasible $\mathbf{x}$, the term added to the objective function is zero. Also, the dual function for the problem above is $g_\rho(\mathbf{y}) = \inf_{\mathbf{x}} L_\rho(\mathbf{x}, \mathbf{y})$. We should mention that the use of the penalty term, under mild conditions on the original problem, has as result the dual function to be differentiable.

3.2.2 Method of Multipliers

If we apply dual ascent to problem (3.7), the algorithm that occurs is

$$\mathbf{x}^{k+1} := \underset{\mathbf{x}}{\text{minimize}}\ L_\rho(\mathbf{x}, \mathbf{y}^k), \tag{3.8}$$

$$\mathbf{y}^{k+1} := \mathbf{y}^k + \rho(\mathbf{A}\mathbf{x}^{k+1} - \mathbf{b}), \tag{3.9}$$

which is known as the method of multipliers(MM) for solving (3.1). The difference between this algorithm and the dual ascent lies on the $\mathbf{x}$-minimization step. The method of multipliers uses the augmented Lagrangian and the penalty parameter ρ instead of α^k. Also, MM method converges under far more general conditions, including cases when f is infinite or not strictly convex.

Let us assume that f is differentiable (although that is not necessary for the algorithm to work). The optimality conditions for (3.1) are primal and dual feasibility, *i.e.*,

$$\mathbf{A}\mathbf{x}^* - \mathbf{b} = \mathbf{0}, \quad \nabla f(\mathbf{x}^*) + \mathbf{A}^T\mathbf{y} = \mathbf{0},$$

respectively. By definition, $\mathbf{x}^{k+1}$ minimizes $L_\rho(\mathbf{x}, \mathbf{y}^k)$, so

$$\begin{aligned}
&\nabla_{\mathbf{x}} L_\rho(\mathbf{x}^{k+1}, \mathbf{y}^k) = \mathbf{0} \Rightarrow \\
&\nabla_{\mathbf{x}} f(\mathbf{x}^{k+1}) + \mathbf{A}^T\big(\mathbf{y}^k + \rho(\mathbf{A}\mathbf{x}^{k+1} - \mathbf{b})\big) = \mathbf{0} \Rightarrow \\
&\nabla_{\mathbf{x}} f(\mathbf{x}^{k+1}) + \mathbf{A}^T\mathbf{y}^{k+1} = \mathbf{0}
\end{aligned}$$

We see that by using ρ as the step size in the dual update, the iterate $(\mathbf{x}^{k+1}, \mathbf{y}^{k+1})$ is dual feasible. As the MM proceeds, the primal residual, $\mathbf{A}\mathbf{x}^{k+1} - \mathbf{b}$, converges to zero.

Although MM has improved convergence properties compared to dual ascent's, it has a drawback. When f is separable, the augmented Lagrangian L_ρ is not separable, meaning that the $\mathbf{x}$-minimization step (3.8) cannot be carried out separately, in parallel, for each $\mathbf{x}_i$.

3.3 Alternating Direction Method of Multipliers

3.3.1 Algorithm

ADMM is an algorithm that solves problems in the form

$$\begin{aligned} \underset{\mathbf{x},\mathbf{z}}{\text{minimize}} \quad & f(\mathbf{x}) + g(\mathbf{z}), \\ \text{subject to} \quad & \mathbf{A}\mathbf{x} + \mathbf{B}\mathbf{z} = \mathbf{c}, \end{aligned} \tag{3.10}$$

with minimization variables $\mathbf{x} \in \mathbb{R}^n$ and $\mathbf{z} \in \mathbb{R}^m$. Also $\mathbf{A} \in \mathbb{R}^{p\times n}$, $\mathbf{B} \in \mathbb{R}^{p\times m}$ and $\mathbf{c} \in \mathbb{R}^p$. We will make the assumption that both f and g are convex. The only difference between (3.10) and (3.1) is that variables $\mathbf{x}, \mathbf{z}$ in the first, are actually variable $\mathbf{x}$ in the second, divided into two parts, with the objective function separable across the splitting. The optimal value of (3.10) is

$$p^* = \inf\{f(\mathbf{x}) + g(\mathbf{z}) \mid \mathbf{A}\mathbf{x} + \mathbf{B}\mathbf{z} = \mathbf{c}\}.$$

Working as we did with the MM, we form the augmented Lagrangian

$$L_\rho(\mathbf{x}, \mathbf{z}, \mathbf{y}) = f(\mathbf{x}) + g(\mathbf{z}) + \mathbf{y}^T(\mathbf{A}\mathbf{x} + \mathbf{B}\mathbf{z} - \mathbf{c}) + (\rho/2)\left\|\mathbf{A}\mathbf{x} + \mathbf{B}\mathbf{z} - \mathbf{c}\right\|_2^2.$$

ADMM consists of the iterations (unscaled form)

$$\mathbf{x}^{k+1} := \underset{\mathbf{x}}{\text{argmin}}\ L_\rho(\mathbf{x}, \mathbf{z}^k, \mathbf{y}^k), \tag{3.11}$$

$$\mathbf{z}^{k+1} := \underset{\mathbf{z}}{\text{argmin}}\ L_\rho(\mathbf{x}^{k+1}, \mathbf{z}, \mathbf{y}^k), \tag{3.12}$$

$$\mathbf{y}^{k+1} := \mathbf{y}^k + \rho(\mathbf{A}\mathbf{x}^{k+1} + \mathbf{B}\mathbf{z}^{k+1} - \mathbf{c}), \tag{3.13}$$

where $\rho > 0$. Apparently, ADMM algorithm is very close to both methods we discussed above. It is composed of an $\mathbf{x}$-minimization, a $\mathbf{z}$-minimization step, and a dual variable update ((3.11) − (3.13), respectively). Also, the dual variable update is being computed, with a step size equal to the augmented Lagrangian parameter ρ. Notice that variables $\mathbf{x}$ and $\mathbf{z}$ are updated in an alternating or sequential fashion, which accounts for the term alternating direction. By separating these minimizations in two steps, we are able to talk about decomposition when at least one of the functions in the objective is separable.

The algorithm state in ADMM consists of $\mathbf{z}^k$ and $\mathbf{y}^k$, which means that $(\mathbf{z}^{k+1}, \mathbf{y}^{k+1})$ is a function of $(\mathbf{z}^k, \mathbf{y}^k)$. That does not apply for $\mathbf{x}^k$ that is an intermediate result computed from the previous state $(\mathbf{z}^{k-1}, \mathbf{y}^{k-1})$.
Finally, the roles of $\mathbf{x}$ and $\mathbf{z}$ are almost symmetric, but not quite, since the dual update is done after the $\mathbf{z}$-update but before the $\mathbf{x}$-update.

3.3.2 Scaled Form

In order to be more convenient, ADMM can be reformed, by combining the linear and quadratic terms in the augmented Lagrangian and scaling the dual variable. Defining the residual $\mathbf{r} = \mathbf{A}\mathbf{x} + \mathbf{B}\mathbf{z} - \mathbf{c}$, we get

$$\mathbf{y}^T\mathbf{r} + \frac{\rho}{2}\|\mathbf{r}\|_2^2 = \frac{\rho}{2}\big(\|\mathbf{r} + \mathbf{u}\|_2^2 - \|\mathbf{u}\|_2^2\big),$$

where $\mathbf{u} = (1/\rho)\mathbf{y}$ is the scaled dual variable. Using this, ADMM can be expressed as

$$\mathbf{x}^{k+1} := \underset{\mathbf{x}}{\operatorname{argmin}}\Big(f(\mathbf{x}) + (\rho/2)\left\|\mathbf{A}\mathbf{x} + \mathbf{B}\mathbf{z}^k - \mathbf{c} + \mathbf{u}^k\right\|_2^2\Big), \tag{3.14}$$

$$\mathbf{z}^{k+1} := \underset{\mathbf{z}}{\operatorname{argmin}}\Big(g(\mathbf{z}) + (\rho/2)\left\|\mathbf{A}\mathbf{x}^{k+1} + \mathbf{B}\mathbf{z} - \mathbf{c} + \mathbf{u}^k\right\|_2^2\Big), \tag{3.15}$$

$$\mathbf{u}^{k+1} := \mathbf{u}^k + \mathbf{A}\mathbf{x}^{k+1} + \mathbf{B}\mathbf{z}^{k+1} - \mathbf{c}. \tag{3.16}$$

We should also define the k-th iteration residual as $\mathbf{r}^k = \mathbf{A}\mathbf{x}^k + \mathbf{B}\mathbf{z}^k - \mathbf{c}$ and the running sum of the residuals on the k-th iteration as, $\mathbf{u}^k = \mathbf{u}^0 + \sum_{j=1}^{k}\mathbf{r}_j$.

3.3.3 Proximity Operator

Equation (3.14) can be expressed as,

$$\mathbf{x}^+ = \underset{\mathbf{x}}{\operatorname{argmin}}\Big(f(\mathbf{x}) + (\rho/2)\|\mathbf{A}\mathbf{x} - \mathbf{v}\|_2^2\Big),$$

with $\mathbf{v} = -\mathbf{B}\mathbf{z} + \mathbf{c} - \mathbf{u}$. If we consider the simple case that $\mathbf{A} = \mathbf{I}$, then the right-hand side is denoted as $\mathbf{prox}_{f,\rho}(\mathbf{v})$, known as the proximity operator of f with penalty ρ. The $\mathbf{x}$-minimization step in the proximity operator is referred as proximal minimization.

3.4 Convergence

We will start by making two assumptions.

Assumption 1:
The functions, $f : \mathbb{R}^n \to \mathbb{R} \cup \{+\infty\}$ and $g : \mathbb{R}^m \to \mathbb{R} \cup \{+\infty\}$ are closed, proper and convex.

This implies that the subproblems arising in the **x**-update and **z**-update are solvable, *i.e.*, there exist $\mathbf{x}$ and $\mathbf{z}$, not necessarily unique, that minimize the Lagrangian. Notice, that we make no presumption for differentiability.

Assumption 2:
The unaugmented Lagrangian L_0 has a saddle point.
There exist $(\mathbf{x}^*, \mathbf{z}^*, \mathbf{y}^*)$, not necessarily unique, for which

$$L_0(\mathbf{x}^*, \mathbf{z}^*, \mathbf{y}) \leq L_0(\mathbf{x}^*, \mathbf{z}^*, \mathbf{y}^*) \leq L_0(\mathbf{x}, \mathbf{z}, \mathbf{y}^*)$$

holds for all $\mathbf{x}, \mathbf{z}, \mathbf{y}$.

From Assumption 1, we have that $L_0(\mathbf{x}^*, \mathbf{z}^*, \mathbf{y}^*) < \infty$ for any saddle point $(\mathbf{x}^*, \mathbf{z}^*, \mathbf{y}^*)$. That means, $(\mathbf{x}^*, \mathbf{z}^*)$ is a solution to (3.10), so the constraint hold and both functions in the objective are finite. Moreover, $\mathbf{y}^*$ is dual optimal, and the optimal values of primal and dual problems are equal(strong duality holds). There is no need for neither matrix in the constraint to be full rank.

Under both assumptions, ADMM iterates satisfy the following:

- Residual convergence, $\mathbf{r}^k \to \mathbf{0}$, as $k \to \infty$.
- Objective convergence, $f(\mathbf{x}^k) + g(\mathbf{z}^k) \to p^*$, as $k \to \infty$.
- Dual variable convergence, $\mathbf{y}^k \to \mathbf{y}^*$, as $k \to \infty$, where $\mathbf{y}^*$ is the dual optimal point.

3.5 Optimality Conditions

The necessary and sufficient optimality conditions for problem (3.10), are primal and dual feasibility,

$$\mathbf{A}\mathbf{x}^* + \mathbf{B}\mathbf{z}^* - \mathbf{c} = \mathbf{0}, \tag{3.17}$$

$$\mathbf{0} \in \partial f(\mathbf{x}^*) + \mathbf{A}^T\mathbf{y}^*, \tag{3.18}$$

$$\mathbf{0} \in \partial g(\mathbf{z}^*) + \mathbf{B}^T\mathbf{y}^*. \tag{3.19}$$

In case that f, g are differentiable then the subgradients of f, g can be replaced by gradient and $\in$ by $=$.

As seen in chapter 3.3 of [5], $\mathbf{z}^{k+1}$ and $\mathbf{y}^{k+1}$ always satisfy (3.17). Then, in order to reach optimality (3.18) and (3.19) should hold. For that case, we get the following quantities,

$$\mathbf{r}^{k+1} = \mathbf{A}\mathbf{x}^{k+1} + \mathbf{B}\mathbf{z}^{k+1} - \mathbf{c} \tag{3.20}$$

$$\mathbf{s}^{k+1} = \rho\mathbf{A}^T\mathbf{B}(\mathbf{z}^{k+1} - \mathbf{z}^k) \tag{3.21}$$

Equation (3.20) can be viewed as a residual for the primal feasibility condition (3.17) and (3.21) as the dual residual for (3.18). Both equations (3.20) and (3.21) as ADMM proceeds, converge to zero.

3.6 Stopping Creteria

Residuals (3.20) and (3.21) can be described as a bound on the objective suboptimality of the current point,*i.e.*, $f(\mathbf{x}) + g(\mathbf{z}) - p^*$. This means, that as the residuals become smaller, so will the objective suboptimality. So the primal and dual residuals should be,

$$\left\|\mathbf{r}^k\right\|_2 \leq \epsilon^{\text{pri}} \quad \text{and} \quad \left\|\mathbf{s}^k\right\|_2 \leq \epsilon^{\text{dual}},$$

where $\epsilon^{\text{pri}} > 0$ and $\epsilon^{\text{pri}} > 0$ are feasibility tolerances for the primal and dual feasibility conditions. These tolerances can be computed by an absolute and relative criterion, such as,

$$\begin{aligned}\epsilon^{\text{pri}} &= \sqrt{p}\epsilon^{\text{abs}} + \epsilon^{\text{rel}} \max\{\left\|\mathbf{A}\mathbf{x}^k\right\|_2, \left\|\mathbf{B}\mathbf{z}^k\right\|_2, \|\mathbf{c}\|_2\},\\ \epsilon^{\text{dual}} &= \sqrt{n}\epsilon^{\text{abs}} + \epsilon^{\text{rel}} \left\|\mathbf{A}^T\mathbf{y}^k\right\|_2,\end{aligned}$$

with $\epsilon^{\text{abs}}, \epsilon^{\text{abs}}$ positive numbers.

3.7 Over-relaxation

In $\mathbf{z}, \mathbf{y}$-updates, we can replace $\mathbf{A}\mathbf{x}^{k+1}$ with

$$\alpha^k \mathbf{A}\mathbf{x}^{k+1} - (1 - \alpha^k)(\mathbf{B}\mathbf{z}^k - \mathbf{c})$$

where $\alpha^k \in (0, 2)$ is a relaxation parameter. In case $\alpha^k > 1$, this is called over-relaxation, and when $\alpha^k < 1$, it is called under-relaxation. This can be used to improve convergence.

3.8 Soft Thresholding

Consider $f(\mathbf{x}) = \lambda \|\mathbf{x}\|_1$, with $\lambda > 0$ and $\mathbf{A} = \mathbf{I}$. Then the scalar x_i-update is,

$$x_i^{k+1} := \underset{x_i}{\operatorname{argmin}} \big(\lambda |x_i| + (\rho/2)(x_i - v_i)^2\big)$$

Although the first term is not differentiable, we can compute a closed-form solution for the problem and it will be denoted as

$$x_i^{k+1} := S_{\lambda/\rho}(v_i).$$

S is called the soft thresholding operator, and is defined as,

$$S_\kappa(\alpha) = \begin{cases} \alpha - \kappa, & \text{if } \alpha > \kappa, \\ 0, & \text{if } |\alpha| \leq \kappa, \\ \alpha + \kappa, & \text{if } \alpha < -\kappa, \end{cases}$$

We will refer to updates that reduce to this form as element-wise soft thresholding.

3.9 Constrained Convex Optimization

The generic constrained convex optimization problem is

$$\begin{aligned} \underset{\mathbf{x}}{\text{minimize}} \quad & f(\mathbf{x}), \\ \text{subject to} \quad & \mathbf{x} \in \mathbb{C}, \end{aligned} \tag{3.22}$$

with $\mathbf{x} \in \mathbb{R}^n$, f convex function and $\mathbb{C}$ convex set.

In ADMM form, the problem can be reformed as

$$\begin{array}{ll} \underset{\mathbf{x}}{\text{minimize}} & f(\mathbf{x}) + g(\mathbf{z}), \\ \text{subject to} & \mathbf{x} - \mathbf{z} = \mathbf{0}, \end{array} \tag{3.23}$$

where function g is the indicator for set $\mathbb{C}$.
The augmented Lagrangian (scaled form) is

$$L_\rho(\mathbf{x}, \mathbf{z}, \mathbf{u}) = f(\mathbf{x}) + g(\mathbf{z}) + (\rho/2) \left\|\mathbf{x} - \mathbf{z} + \mathbf{u}\right\|_2^2,$$

so ADMM for this problem is

$$\begin{aligned} \mathbf{x}^{k+1} &:= \underset{\mathbf{x}}{\operatorname{argmin}} \Big(f(\mathbf{x}) + (\rho/2) \left\|\mathbf{x} - \mathbf{z}^k + \mathbf{u}^k\right\|_2^2 \Big), \\ \mathbf{z}^{k+1} &:= \Pi_{\mathbb{C}}(\mathbf{x}^{k+1} + \mathbf{u}^k), \\ \mathbf{u}^{k+1} &:= \ \mathbf{u}^k + \mathbf{x}^{k+1} - \mathbf{z}^{k+1}, \end{aligned}$$

where $\Pi_{\mathbb{C}}$ denotes projection (in the Euclidean norm) onto $\mathbb{C}$. There is no need for the objective to be smooth; we can simply add more constraints by defining f to be infinite where the constraints are violated. Then the proximity minimization of $\mathbf{x}$ becomes a constrained minimization problem over $\mathbf{dom} f = \{\mathbf{x} \mid f(\mathbf{x}) < \infty\}$.

In all cases where the constraint is $\mathbf{x} - \mathbf{z} = \mathbf{0}$, the primal and dual residuals take the simple form

$$\begin{aligned} \mathbf{r}^k &= \mathbf{x}^k - \mathbf{z}^k, \\ \mathbf{s}^k &= -\rho(\mathbf{z}^k - \mathbf{z}^{k+1}). \end{aligned}$$

3.10 Linear and Quadratic Programming

A generic form for quadratic programming(QP) is,

$$\begin{array}{ll} \underset{\mathbf{x}}{\text{minimize}} & \frac{1}{2}\mathbf{x}^T\mathbf{P}\mathbf{x} + \mathbf{q}^T\mathbf{x}, \\ \text{subject to} & \mathbf{A}\mathbf{x} = \mathbf{b}, \quad \mathbf{x} \geq \mathbf{0}, \end{array} \tag{3.24}$$

with variable $\mathbf{x} \in \mathbb{R}^n$; we assume that $\mathbf{P} \in \mathbb{S}_+^n$. In case that, $\mathbf{P}$ is zero matrix, then (3.24) reduces to standard form linear program (LP). We express, now, QP in ADMM form like (3.23), with

$$f(\mathbf{x}) = (1/2)\mathbf{x}^T\mathbf{P}\mathbf{x} + \mathbf{q}^T\mathbf{x}, \qquad \mathbf{dom}f = \{\mathbf{x} \mid \mathbf{Ax}=\mathbf{b}\},$$

and function g as the indicator function of the nonegative orthant $\mathbb{R}^n_+$.

$$\begin{aligned}
\mathbf{x}^{k+1} &:= \underset{\mathbf{x}}{\operatorname{argmin}}\Big(f(\mathbf{x}) + (\rho/2)\left\|\mathbf{x} - \mathbf{z}^k + \mathbf{u}^k\right\|_2^2\Big)\\
\mathbf{z}^{k+1} &:= (\mathbf{x}^{k+1} + \mathbf{u}^k)_+\\
\mathbf{u}^{k+1} &:= \ \mathbf{u}^k + \mathbf{x}^{k+1} - \mathbf{z}^{k+1}
\end{aligned}$$

where $\mathbf{x}$-minimization step is an equality-constrained least squares problem with optimality conditions

$$\begin{gathered}
(\mathbf{P} + \rho\mathbf{I})\mathbf{x}^{k+1} + \mathbf{A}^T\mathbf{v} + (\mathbf{q} - \rho(\mathbf{z}^k - \mathbf{u}^k)) = \mathbf{0},\\
\mathbf{A}\mathbf{x}^{k+1} - \mathbf{b} = \mathbf{0}.
\end{gathered}$$

3.11 l_1 Norm Problems

There are a variety of problems involving l_1 norms, but in this chapter we will focus on Lasso problem. With the use of ADMM, we gain the advantage of separating the nonsmooth l_1 term from the smooth loss term.

3.11.1 Lasso

The problem l_1 regularized linear regression, or commonly known lasso, is

$$\underset{\mathbf{x}}{\text{minimize}} \quad (1/2)\left\|\mathbf{A}\mathbf{x} - \mathbf{b}\right\|_2^2 + \lambda\left\|\mathbf{x}\right\|_1, \tag{3.25}$$

where $\lambda > 0$, is a scalar regularization parameter. Lasso is mostly beneficial in cases where only a small part of a huge number of possible factors can be of use.
Lasso expressed in ADMM form, with

$$f(\mathbf{x}) = (1/2)\left\|\mathbf{A}\mathbf{x} - \mathbf{b}\right\|_2^2 \ \text{ and } \ g(\mathbf{z}) = \lambda\left\|\mathbf{x}\right\|_1,$$

consists of the iterations

$$\begin{aligned}
\mathbf{x}^{k+1} &:= (\mathbf{A}^T\mathbf{A} + \rho\mathbf{I})^{-1}(\mathbf{A}^T\mathbf{b} + \rho(\mathbf{z}^k - \mathbf{u}^k))\\
\mathbf{z}^{k+1} &:= S_{\lambda/\rho}(\mathbf{x}^{k+1} + \mathbf{u}^k)\\
\mathbf{u}^{k+1} &:= \ \mathbf{u}^k + \mathbf{x}^{k+1} - \mathbf{z}^{k+1}.
\end{aligned}$$

Note that in $\mathbf{x}$-update, $\mathbf{A}^T\mathbf{A} + \rho\mathbf{I}$ is always invertible since $\rho > 0$ and also $\mathbf{x}$-update is a ridge regression (*i.e.*, quadratically regularized least squares) computation.

3.12 Consensus

In this section, we will deal with a generic optimization problem, called consensus and describe ADMM-based methods that use distributed optimization in order to solve it.

3.12.1 Global Variable Consensus Optimization

Consider a single global variable, but the objective and constraints can split into N parts

$$\underset{\mathbf{x}}{\text{minimize}} \quad f(\mathbf{x}) = \sum_{i=1}^{N} f_i(\mathbf{x})$$

where $\mathbf{x} \in \mathbb{R}^n$ and $f_i : \mathbb{R}^n \to \cup\{+\infty\}$ are convex. Each term can also be a constraint, where when it is violated we assign, $f_i(\mathbf{x}) = +\infty$. In order to find variable $\mathbf{x}$, data are 'working together' to develop a global model.
If we rewrite the problem with local variables $\mathbf{x}_i \in \mathbb{R}^n$ and a global variable $\mathbf{z}$ we get,

$$\begin{aligned} \underset{\mathbf{x}_1,\ldots,\mathbf{x}_N,\mathbf{z}}{\text{minimize}} \quad & \sum_{i=1}^{N} f_i(\mathbf{x_i}) \\ \text{subject to} \quad & \mathbf{x}_i - \mathbf{z} = \mathbf{0}, \quad i = 1, \ldots, N \end{aligned} \tag{3.26}$$

This is called global consensus problem. That is because of the constraint that obligates each local variable to agree with the global variable. The augmented Lagrangian (unscaled form) for the problem is,

$$L_\rho(\mathbf{x}_1, \ldots, \mathbf{x}_N, \mathbf{z}, \mathbf{y}) = \sum_{i=1}^{N} \left(f_i(\mathbf{x}_i) + \mathbf{y}_i^T(\mathbf{x}_i - \mathbf{z}) + (\rho/2) \left\| \mathbf{x}_i - \mathbf{z} \right\|_2^2 \right).$$

The resulting ADMM algorithm is

$$\begin{aligned} \mathbf{x}_i^{k+1} &:= \underset{\mathbf{x}_i}{\operatorname{argmin}} \Big(f_i(\mathbf{x}_i) + \mathbf{y}_i^{kT}(\mathbf{x}_i - \mathbf{z}^k) + (\rho/2) \left\| \mathbf{x}_i - \mathbf{z}^k \right\|_2^2 \Big) \\ \mathbf{z}^{k+1} &:= \frac{1}{N} \sum_{i=1}^{N} \left(\mathbf{x}^{k+1} + (1/\rho)\mathbf{y}_i^k \right) \\ \mathbf{y}_i^{k+1} &:= \mathbf{y}_i^k + \rho(\mathbf{x}^{k+1} - \mathbf{z}^{k+1}). \end{aligned}$$

The first and third step are carried out independently for each $i = 1, \ldots, N$ in order to drive variables into consensus. The quadratic regularization helps attract all variables toward their average value, while attempting to minimize each local f_i. Also, the processing element that handles $\mathbf{x}$ variable, is called central controller or fusion center.

This algorithm, is being used to solve problems that the objective functions and constraints are distributed across multiple processors, with each one handles its private objective and constraint, while at every iteration the quadratic term is updated.

For consensus ADMM, primal and dual residuals are,

$$\left\|\mathbf{r}^k\right\|_2^2 = \sum_{i=1}^{N} \left\|\mathbf{x}_i^k - \overline{\mathbf{x}}_i^k\right\|_2^2,$$
$$\left\|\mathbf{s}^k\right\|_2^2 = N\rho^2 \left\|\overline{\mathbf{x}}_i^k - \overline{\mathbf{x}}_i^{k-1}\right\|_2^2,$$

where $\overline{\mathbf{x}}^k = (1/N)\sum_{i=1}^{N} \mathbf{x}^k$.

3.12.2 Global Variable Consensus with Regularization

In this section, we are going to include an objective term g, that could represent a constraint or even a regularization. Function g is handled by the central collector. The problems is

$$\begin{aligned} \underset{\mathbf{x}_1,\ldots,\mathbf{x}_N,\mathbf{z}}{\text{minimize}} \quad & \sum_{i=1}^{N} f_i(\mathbf{x_i}) + g(\mathbf{z}), \\ \text{subject to} \quad & \mathbf{x}_i - \mathbf{z} = \mathbf{0}, \quad i = 1, \ldots, N. \end{aligned} \tag{3.27}$$

The resulting algorithm for problem (3.27) is

$$\mathbf{x}_i^{k+1} := \underset{\mathbf{x}_i}{\operatorname{argmin}}\Big(f_i(\mathbf{x}_i) + \mathbf{y}_i^{kT}(\mathbf{x}_i - \mathbf{z}^k) + (\rho/2)\left\|\mathbf{x}_i - \mathbf{z}^k\right\|_2^2 \Big), \tag{3.28}$$

$$\mathbf{z}^{k+1} := \underset{\mathbf{z}}{\operatorname{argmin}}\Big(g(\mathbf{z}) + (N\rho/2)\left\|\mathbf{z} - \overline{\mathbf{x}}^{k+1} - (1/\rho)\overline{\mathbf{y}}^k\right\|_2^2 \Big), \tag{3.29}$$

$$\mathbf{y}_i^{k+1} := \mathbf{y}_i^k + \rho(\mathbf{x}^{k+1} - \mathbf{z}^{k+1}), \tag{3.30}$$

with $\overline{\mathbf{x}}, \overline{\mathbf{y}}$ the average sum over all $\mathbf{x}_i, \mathbf{y}_i$.

3.13 Distributed Alternating Direction Method of Multipliers

This section will focus on the main algorithm of our thesis. This algorithm, can be used for solving separable optimization problems in networks of interconnected nodes or agents. As we mentioned in the beginning of this chapter, our work is based on [2].

3.13.1 Introduction to the Algorithm

Consider the following separable optimization problem

$$\begin{aligned} \underset{\mathbf{x}}{\text{minimize}} \quad & f_1(\mathbf{x}) + f_2(\mathbf{x}) + \ldots + f_P(\mathbf{x}), \\ \text{subject to} \quad & \mathbf{x} \in \mathbb{X}_1 \cap \mathbb{X}_2 \cap \ldots \cap \mathbb{X}_P, \end{aligned} \tag{3.31}$$

where $\mathbf{x} \in \mathbb{R}^n$ is the optimization variable, and x^* will denote any solution of the problem. Figure 3.1 shows the association between problem (3.31) and a network of P nodes. As illustrated, cost function f_p and set $\mathbb{X}_p$ are private and accessible only from node p. Nodes i, j can exchange messages with each other, only when they are neighbors. Even in the case we do not have an all to all communication, all nodes try to solve (3.31) in a cooperative way. The solution to the problem will occur, with the absence of a central node or aggregating data in the network. Methods that work this way, are called distributed algorithms.

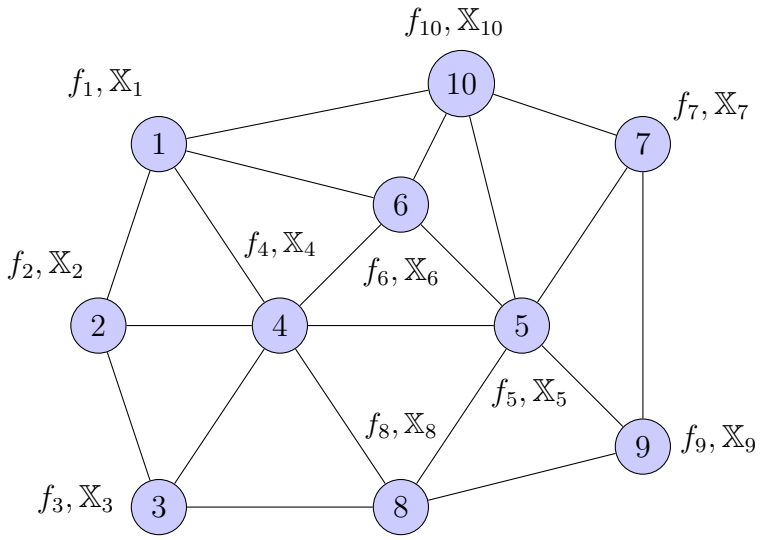


Figure 3.1: Network of $P = 10$ nodes.

Furthermore, we make the following assumptions,

- Each $f_p : \mathbb{R}^n \to \mathbb{R}$ is a convex function over $\mathbb{R}^n$, and each set $\mathbb{X}_p$ is a closed and convex.
- There is at least one solution $\mathbf{x}^*$, for problem (3.31).
- The network is connected and does not vary with time.
- A coloring scheme of the network is available.

More specifically, the third assumption implicates that, when a network is connected then for any two nodes there is a path connecting them. Finally, in the last assumption, a coloring scheme is an assignment of numbers to the nodes of the network, such that no neighboring nodes have the same number. Those numbers, from now on, will be called colors.

The D-ADMM algorithm is based on ADMM, which as we have already seen in section 3.3 is an augmented Lagrangian based algorithm that consists of only one loop (in order to update primal and dual variables). D-ADMM is a distributed algorithm and can be applied to any connected network topology.

3.13.2 Notation

To begin with, the network is represented as an undirected graph $G = (\mathbb{V}, \mathbb{E})$ where $\mathbb{V} = \{1, 2, \ldots, P\}$, is the set of nodes, and $\mathbb{E} \subseteq \mathbb{V} \times \mathbb{V}$ is the sets of edges. The cardinality of these sets are P and E, respectively. An edge of connecting nodes is represented as (i, j), with $i < j$, meaning that these nodes can send and receive messages to each other. The set of neighbors of node p is written as $\mathbb{N}_p$, and its degree is $D_p = |\mathbb{N}_p|$.

According to the fourth assumption we made, a proper coloring with $\mathbb{C} = \{1, \ldots, C\}$ of the graph is available. This means that each node is labeled with a number $c \in \mathbb{C}$, which we call color, so that no adjacent nodes have the same color. The set of nodes that have color c will be denoted with $\mathbb{C}_c$, for $c = 1, \ldots, C$, and its cardinality with $C_c = |\mathbb{C}_c|$.

Assume the nodes are ordered such that the first C_1 nodes have color 1, the next C_2 nodes have color 2 and so on.

3.13.3 Problem Reformulation

ADMM is not directly applicable to problem (3.31), so we need to reformulate it. At first, we create copies $\mathbf{x}_p$ of the global variable $\mathbf{x}$ and attach them to each node p. We also constrain them to be equal since the network is connected. Then, we get

$$\begin{aligned} \underset{\overline{\mathbf{x}}=(\mathbf{x}_1,\ldots,\mathbf{x}_P)}{\text{minimize}} \quad & f_1(\mathbf{x}_1) + \ldots + f_P(\mathbf{x}_P), \\ \text{subject to} \quad & \mathbf{x}_1 \in \mathbb{X}_1, \ldots, \mathbf{x}_P \in \mathbb{X}_P, \\ & \mathbf{x}_i = \mathbf{x}_j, \ (i,j) \in \mathbb{E}, \end{aligned} \tag{3.32}$$

where $\overline{\mathbf{x}} = (\mathbf{x}_1, \ldots, \mathbf{x}_P) \in (\mathbb{R}^n)^P$ is the optimization variable. Problem (3.32) depends now only on the equations $\mathbf{x}_i = \mathbf{x}_j$, for all pairs $(i,j) \in \mathbb{E}$. These constraints can be written as $(\mathbf{B}^T \otimes \mathbf{I}_n)\overline{\mathbf{x}} = \mathbf{0}$, where $\mathbf{B} \in \mathbb{R}^{P \times E}$ is the node arc-incidence matrix of the graph, and $\otimes$ is the Kronecker product. Each column of $\mathbf{B}$ is connected with $(i,j) \in \mathbb{E}$ and has 1 and -1 at the i-th and j-th entry, respectively, while every other entry is zero. The existence of colors imports a partition of $\mathbf{B}$ as $[\mathbf{B}_1^T, \ldots, \mathbf{B}_C^T]^T$, where the columns of $\mathbf{B}_c^T$ are associated to the nodes with color c. In the same notion, we partition $\overline{\mathbf{x}}$ as $\overline{\mathbf{x}}_1, \ldots, \overline{\mathbf{x}}_C$, where $\overline{\mathbf{x}}_c \in (\mathbb{R}^n)^{C_c}$ collects the copies of all nodes with color c. With these changes, we rewrite (3.32) as,

$$\begin{aligned} \underset{\overline{\mathbf{x}}_1,\ldots,\overline{\mathbf{x}}_C}{\text{minimize}} \quad & \sum_{c=1}^{C} \sum_{p \in C_c} f_p(\mathbf{x}_p) \\ \text{subject to} \quad & \overline{\mathbf{x}}_1 \in \overline{\mathbb{X}}_1, \ldots, \overline{\mathbf{x}}_C \in \overline{\mathbb{X}}_C, \\ & \sum_{c=1}^{C} (\mathbf{B}_c^T \otimes \mathbf{I}_n)\overline{\mathbf{x}}_c = \mathbf{0}, \end{aligned} \tag{3.33}$$

where $\overline{\mathbb{X}}_c = \Pi_{p \in C_c} \mathbb{X}_p$.

3.13.4 Extended ADMM

The extended ADMM is a natural generalization of ADMM [6]. In case we have C functions g_c, C sets $\mathbb{X}_c$, and C matrices $\mathbf{A}_c$ (with the same number of rows), the extended ADMM solves

$$\begin{aligned} \underset{\mathbf{x}_1,\ldots,\mathbf{x}_C}{\text{minimize}} \quad & \sum_{c=1}^{C} g_c(\mathbf{x}_c), \\ \text{subject to} \quad & \mathbf{x}_c \in \mathbb{X}_c,\ c = 1,\ldots,C, \\ & \sum_{c=1}^{C} \mathbf{A}_c\mathbf{x}_c = \mathbf{0}, \end{aligned} \tag{3.34}$$

where $\mathbf{x} := (\mathbf{x}_1, \ldots, \mathbf{x}_C)$ is the optimization variable. The augmented Lagrangian for problem (3.34) is

$$L_\rho(\mathbf{x};\boldsymbol{\lambda}) = \sum_{c=1}^{C}(g_c(\mathbf{x}_c) + \boldsymbol{\lambda}^T\mathbf{A}_c\mathbf{x}_c) + \frac{\rho}{2}\left\|\sum_{c=1}^{C}\mathbf{A}_c\mathbf{x}_c\right\|_2^2$$

Assuming that at time k we have completed $\mathbf{x}_1^k, \mathbf{x}_2^k, \ldots, \mathbf{x}_p^k, \boldsymbol{\lambda}^k$, then, the updates of the primal and dual variables of the extended ADMM are computed as follows,

$$\mathbf{x}_1^{k+1} = \underset{\mathbf{x}_1\in\mathbb{X}_1}{\text{argmin}} \quad L_\rho(\mathbf{x}_1, \mathbf{x}_2^k, \ldots, \mathbf{x}_C^k; \boldsymbol{\lambda}^k) \tag{3.35}$$

$$\mathbf{x}_2^{k+1} = \underset{\mathbf{x}_2\in\mathbb{X}_2}{\text{argmin}} \quad L_\rho(\mathbf{x}_1^{k+1}, \mathbf{x}_2, \mathbf{x}_3^k, \ldots, \mathbf{x}_C^k; \boldsymbol{\lambda}^k) \tag{3.36}$$

$$\vdots$$

$$\mathbf{x}_C^{k+1} = \underset{\mathbf{x}_C\in\mathbb{X}_C}{\text{argmin}} \quad L_\rho(\mathbf{x}_1^{k+1}, \mathbf{x}_2^{k+1}, \ldots, \mathbf{x}_{C-1}^{k+1}, \mathbf{x}_C; \boldsymbol{\lambda}^k) \tag{3.37}$$

$$\boldsymbol{\lambda}^{k+1} = \boldsymbol{\lambda}^k + \rho\sum_{c=1}^{C}\mathbf{A}_c\mathbf{x}_c^{k+1}, \tag{3.38}$$

where $\boldsymbol{\lambda}$ is the dual variable and $\rho > 0$. If we have only two colors in our network, then expressions (3.35)−(3.38) becomes the ordinary ADMM, which converges under mild assumptions. However, if we have more colors, then there is only a known proof of convergence when all the functions g_c are strongly convex [6].

Theorem 1: Let $g_c : \mathbb{R}^{n_c} \to \mathbb{R}$ be a convex function over $\mathbb{R}^{n_c}, \mathbb{X}_c \subseteq \mathbb{R}^{n_c}$, which is a closed convex set, and $\mathbf{A}_c \in \mathbb{R}^{m \times n_c}$ a matrix for $c = 1, \ldots, C$. If we assume that (3.34) is solvable and one of the two following assumptions holds

- $C = 2$ and each $\mathbf{A}_c$ has full column rank,
- $C \geq 2$ and each g_c is strongly convex.

then, the sequence $\{(\mathbf{x}_1^k, \ldots, \mathbf{x}_C^k, \boldsymbol{\lambda}^k)\}$ generated by (3.35) − (3.38) converges to $\{(\mathbf{x}_1^*, \ldots, \mathbf{x}_C^*, \boldsymbol{\lambda}^*)\}$, where $(\mathbf{x}_1^*, \ldots, \mathbf{x}_C^*)$ is the optimal point for (3.34) and $\boldsymbol{\lambda}^*$ is the dual optimal for

$$\max_{\boldsymbol{\lambda}} \quad \sum_{c=1}^{C} G_c(\boldsymbol{\lambda}),$$

with $G_c(\boldsymbol{\lambda}) = \inf_{\mathbf{x}_c \in \mathbb{X}_c}(g_c(\mathbf{x}_c) + \boldsymbol{\lambda}^T \mathbf{A}_c \mathbf{x}_c)$, for $c = 1, \ldots, C$.

3.13.5 Laplacian Graph

Before applying the extended ADMM to problem (3.33), we will introduce the Laplacian graph term.

Given a simple graph $G = (\mathbb{V}, E)$, with $\mathbb{V}, \mathbb{E}$ sets as already denoted in section 3.13.2, its Laplacian matrix $\mathbf{L}$ is a $n \times n$ matrix defined in [12] as $\mathbf{L} = \mathbf{D} - \mathbf{A}$. Matrix $\mathbf{D}$ is a diagonal matrix with the degrees for each vertex, and $\mathbf{A}$ is the adjacency matrix of graph G. Matrix $\mathbf{A}$ is also a $n \times n$ matrix where

$$a_{i,j} = \begin{cases} 1, & \text{if } \{i, j\} \in \mathbb{E}, \\ 0, & \text{otherwise.} \end{cases}$$

So, if d_i denotes the degree of the vertex i, then

$$\mathbf{L}_{i,j} = \begin{cases} d_i, & \text{if } i = j, \\ -1, & \text{if } i \neq j \text{ and } i, j \text{ are adjacent,} \\ 0, & \text{otherwise.} \end{cases}$$

This will help us rewrite some equations in the following.

3.13.6 Applying Extended ADMM

We now apply the extended ADMM to problem (3.33), which has the format of (3.34). We will show how the $\overline{\mathbf{x}}_1 = (\mathbf{x}_1, \ldots, \mathbf{x}_{C_1})$ can be updated but, in the same way, we can rewrite the rest minimization steps. So,

$$\overline{\mathbf{x}}_1^{k+1} = \underset{\overline{\mathbf{x}}_1 \in \overline{\mathbb{X}}_1}{\operatorname{argmin}} \quad \sum_{p \in \mathbb{C}_1} f_p(\mathbf{x}_p) + \boldsymbol{\lambda}^{kT} \mathbf{A}_1 \overline{\mathbf{x}}_1 + \frac{\rho}{2} \left\| \mathbf{A}_1 \overline{\mathbf{x}}_1 + \sum_{c=2}^{C} \mathbf{A}_c \overline{\mathbf{x}}_c^k \right\|_2^2, \tag{3.39}$$

with $\mathbf{A}_1 = \mathbf{B}_1^T \otimes \mathbf{I}_n$. The norm term in the equation above can be written as,

$$\frac{\rho}{2} \overline{\mathbf{x}}_1^T \mathbf{A}_1^T \mathbf{A}_1 \overline{\mathbf{x}}_1 + \rho \overline{\mathbf{x}}_1^T \sum_{c=2}^{C} \mathbf{A}_1^T \mathbf{A}_c \overline{\mathbf{x}}_c^k + \frac{\rho}{2} \left\| \sum_{c=2}^{C} \mathbf{A}_c \overline{\mathbf{x}}_c^k \right\|_2^2. \tag{3.40}$$

The first term is essentially $\mathbf{A}_1^T \mathbf{A}_1 = \mathbf{B}_1 \mathbf{B}_1^T \otimes \mathbf{I}_n$, where $\mathbf{B}_1 \mathbf{B}_1^T$ is a diagonal block of the graph Laplacian, which means that the degrees of the respective nodes appear at the diagonal. This way, we can write the first term of (3.40) as, $\overline{\mathbf{x}}_1^T \mathbf{A}_1^T \mathbf{A}_1 \overline{\mathbf{x}}_1 = \sum_{p \in \mathbb{C}_1} D_p \left\| \mathbf{x}_p \right\|_2^2$. Similarly, the second term can be written as $\mathbf{A}_1^T \mathbf{A}_c = \mathbf{B}_1 \mathbf{B}_c^T \otimes \mathbf{I}_n$, where $\mathbf{B}_1 \mathbf{B}_c^T$ is an off-diagonal block of the Laplacian matrix. This means that, for $i \neq j$, if nodes i, j are neighbors, then the ij-th entry of the Laplacian matrix contains -1, and 0 otherwise. Then, we can write the second term of (3.40) as $\overline{\mathbf{x}}_1^T \sum_{c=2}^{C} \mathbf{A}_1^T \mathbf{A}_c \overline{\mathbf{x}}_c^k = -\sum_{p \in \mathbb{C}_1} \sum_{j \in \mathbb{N}_p} \mathbf{x}_p^T \mathbf{x}_j^k$. Finally, we can ignore the last term, since there is no dependency on $\overline{\mathbf{x}}_1$.
Furthermore, we can rewrite the second term in (3.39) as

$$\left((\mathbf{B}_1 \otimes \mathbf{I}_n) \boldsymbol{\lambda}^k \right)^T \overline{\mathbf{x}}_1 = \sum_{p \in \mathbf{C}_1} \sum_{j \in \mathbb{N}_p} \boldsymbol{\lambda}_{pj}^{k\,T} \mathbf{x}_p,$$

where $\boldsymbol{\lambda}_{ij}$ is defined for $i < j$ and is connected with equation $\mathbf{x}_i = \mathbf{x}_j$. We also set $\boldsymbol{\gamma}_p^k = \sum_{j \in \mathbb{N}_p} \boldsymbol{\lambda}_{pj}^k$. Working like this, equation (3.39) simplifies to

$$\overline{\mathbf{x}}_1 = \underset{\overline{\mathbf{x}}_1 \in \overline{\mathbb{X}}_1}{\operatorname{argmin}} \quad \sum_{p \in \mathbb{C}_1} f_p(\mathbf{x}_p) + \left(\boldsymbol{\gamma}_p^k - \rho \sum_{j \in \mathbb{N}_p} \mathbf{x}_j^k \right)^T \mathbf{x}_p + \frac{\rho D_p}{2} \left\| \mathbf{x}_p \right\|_2^2, \tag{3.41}$$

We conclude from (3.41), that problem (3.39) is decomposed into C_1 problems that can be solved in parallel. Working in the same way, we can compute the minimization steps for the other colors. For that, we should define

$$\gamma_p^k = \sum_{j \in \mathbb{N}_p} \operatorname{sign}(j - p) \boldsymbol{\lambda}_{pj}^k,$$

where sign$(a) = 1$, if $a \geq 0$, and sign$(a) = -1$ otherwise.
The resulting algorithm is called Distributed-ADMM.

Initialization: for all $p \in \mathbb{V}$, set $\boldsymbol{\gamma}_p^1 = \mathbf{x}_p^1 = 0$ and $k = 1$

1: **repeat**

2: **for** $c = 1, ..., C$ **do**

3: **for all** $p \in \mathbb{C}_c$ [in parallel] **do**

$$\mathbf{v}_p^k = \boldsymbol{\gamma}_p^k - \rho \sum_{\substack{j \in \mathbb{N}_p \\ j<p}} \mathbf{x}_j^{k+1} - \rho \sum_{\substack{j \in \mathbb{N}_p \\ j>p}} \mathbf{x}_j^k$$

4: and find

$$\mathbf{x}_p^{k+1} = \begin{array}{ll} \text{argmin} & f_p(\mathbf{x}_p) + \mathbf{v}_p^{kT}\mathbf{x}_p + \frac{\rho D_p}{2}||\mathbf{x}_p||^2 \\ \text{subject to} & \mathbf{x}_p \in \mathbb{X}_p \end{array}$$

5: Send $\mathbf{x}_p^{k+1}$ to $\mathbb{N}_p$

6: **end for**

7: **end for**

8: **for all** $p \in \mathbb{V}$ [in parallel] **do**

$$\boldsymbol{\gamma}_p^{k+1} = \boldsymbol{\gamma}_p^k + \rho \sum_{j \in \mathbb{N}_p} (\mathbf{x}_p^{k+1} - \mathbf{x}_p^k)$$

9: **end for**

10: $k = k + 1$

11: **until** some stopping criterion is met

D-ADMM algorithm is an asynchronous algorithm, because all nodes operate in a color-based order. Node p works in parallel with the nodes that belong to the same color, and when it receives the updated $\mathbf{x}^{k+1}$ from all its neighbors in the lower colors, it continues to operate, since steps 4 and 5 from the algorithm can be executed.

Chapter 4

Message Passing Interface

As we discussed in the previous chapter, D-ADMM algorithm is applicable to every network topology, and has steps that can be performed in parallel. For that reason, we will introduce a communication library for both parallel computers and workstation networks, Message Passing Interface (MPI). It is a portable, standard interface for writing parallel applications, in all branches of science and engineering, on systems of all sizes, from laptops to clusters of the largest and most powerful supercomputers in the world. MPI is a library of subprograms that can be called from C, C++ or Fortran 90. This chapter is based on [7] and [8].

4.1 Multiple Instruction-Multiple Data Systems

Multiple instruction-multiple data systems (MIMD) is a technique employed to achieve parallelism. Machines using MIMD have a number of processors that function asynchronously and independently. At any time, different processors may be executing different instructions on different pieces of data. MIMD systems can be of either shared memory or distributed memory systems, but we will focus on distributed memory machines.

4.1.1 Distributed-Memory MIMD

In distributed-memory systems, each processor has its private memory and a generic one is illustrated in 4.1. Also, it can be seen as a graph, where the edges are communication wires. Now, the vertices could be a pair of processor and memory (nodes), which this type of network is called static. On the other hand, vertices could correspond to nodes and the rest to switches, and this is called dynamic network. An example of both type of networks can be seen in Figure 4.2.

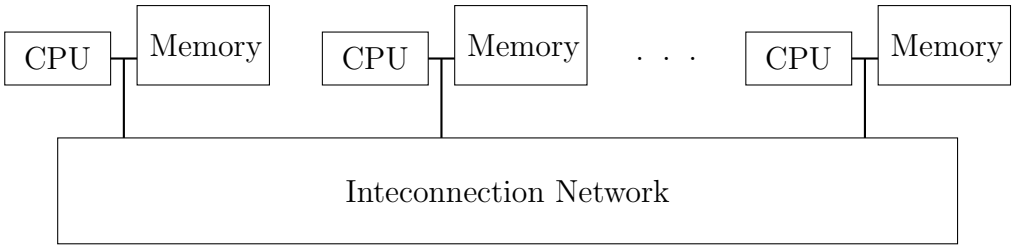


Figure 4.1: Generic distributed-memory system.

The ideal interconnection network is a fully connected network, from a performance and programming perspective, where each node can communicate directly with every other node in the network. With a fully connected network, the communication presents no delay, and every node can exchange messages with other nodes, while other communications are taking place. But, its drawback is that, due to high cost of building such a network, makes it impractical to construct a machine with more than a few nodes.

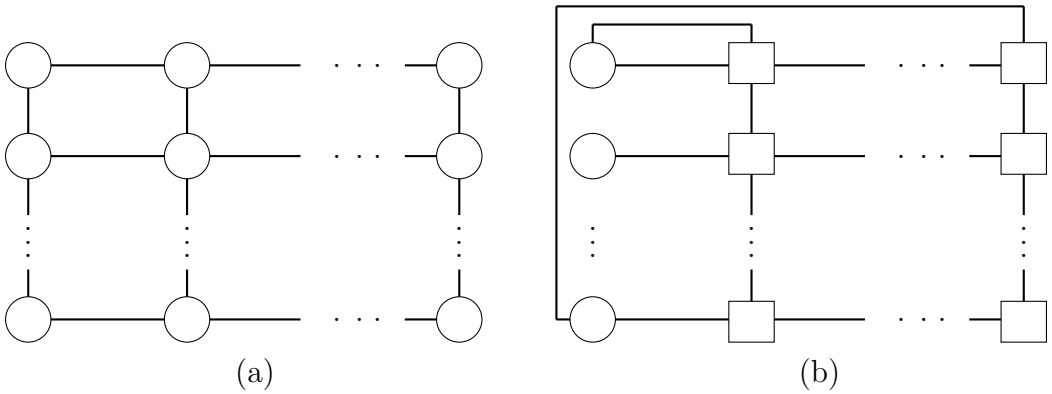


Figure 4.2: Static network (a) and Dynamic Network (b).
Round vertices are nodes and squares are switches.

4.2 Message Passing

Before talking about message passing, we should define the term process. It is an instance of a program or subprogram that is executing essentially autonomously on a physical processor. In a parallel program, there is the possibility of more than one processes coexist.

For programming in distributed-memory MIMD machine, the most common way is by message passing. With message passing, processes coordinate their activities by sending and receiving messages. The standard functions in MPI that are used

for that purpose are:

```
int MPI_Send(void*          buffer        /* in */,
             int            count         /* in */,
             MPI_Datatype   datatype      /* in */,
             int            destination   /* in */,
             int            tag           /* in */,
             MPI_Comm       communicator  /* in */)

int MPI_Recv(void*          buffer        /* out */,
             int            count         /* in */,
             MPI_Datatype   datatype      /* in */,
             int            source        /* in */,
             int            tag           /* in */,
             MPI_Comm       communicator  /* in */,
             MPI_Status*    status        /* out */)
```

At the beginning of the program execution, the processes are set from the user, so that no more processes can be added, which is really useful, because D-ADMM algorithm is suited for static networks. After that, a unique integer, called rank, is assigned to each process, from 0 to $p-1$, with p denoting the number of processes.

We will present an example, for both functions, so that the reader could have a better understanding on them. Consider two processes, with ranks 0 and 1, that try to communicate. First process 0, sends the double variable x to process 1. Then process 1 wants to receive this message, with valid data. In order to do that, process 0 calls the first function and process 1 calls the second one with the same *tag* and *communicator* arguments. Also, the *buffer* should be of the same type and size as the message sent, meaning *datatype* and *count* parameters, respectively. Note that the commands that both processes are using are different, without implying that they should be two separate programs. A possible part of a program that executes the above functionality could be the following:

```
if (proc_rank == 0)
    MPI_Send(&x, 1, MPI_DOUBLE, 1, 0, MPI_COMM_WORLD);
else
    MPI_Recv(&x, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &status);
```

The final argument *status* in MPI_Recv contains information, such as the actual size of the message received.

We should mention also that there are many MPI_Datatype, such as MPI_CHAR, MPI_INT, MPI_LONG, etc, but in our thesis we will use only MPI_DOUBLE and MPI_INT.

The approach to program MIMD systems is called single-program-multiple-data (SPMD). In SPMD programs, the capability of executing different programs is obtained by the conditional branches within the source code.

4.2.1 Buffering

We will continue with the example from the previous section, that we have two processes running on different nodes, A and B respectively. But in this case, process 1 will not call MPI_Recv immediately. By this, process 0 has two "choices". One is to send a "request to send" to the other process and wait until it responds with a "ready to receive", so that they can exchange the message. The other "choice" is the system software to buffer the message. The contents of the message can be copied into a system-controlled block of memory, and process 0 can continue its execution. Now, whenever process 1 is ready to receive, the system software just copies the buffered message into the memory location process 1 has allocated. The first "choice" is called synchronous communication, while the second buffered communication.

The great advantage of the buffered communication is that it does not stall the sending process from executing other work. But on the other hand, it uses up system resources that otherwise it would not be needed and if the receiving process is ready, then it would take more time to finish, because system has to buffer the message and then copy it to the user program memory location.

4.2.2 Blocking Communication

There is a situation that occurs if we reverse the arrival at communication points, that should be mentioned. Consider now that process 1 executes the MPI_Recv, but process 0, does not executes the MPI_Send until some later time. The function being used for receive, MPI_Recv, is blocking, meaning that process 1 remains idle until the message becomes available. In blocking communication, process 0 can continue with the send function, without waiting a "ready to receive" from process 1, unlike synchronous communication.

4.3 Communicators and Distributed Graph Topologies

4.3.1 Communicators

To begin with, we define the concept of a communicator. In general, a communicator can be a subset of processes as a communication universe. The default communicator in MPI is MPI_COMM_WORLD, where it includes every process that the user declared at the beginning of the execution of the program.

Furthermore, in MPI there are two types of communicators: intra-communicators and inter-communicators. Intra-communicators are essentially a set of processes that communicate through collective communication/operations. On the other hand, with inter-communicators the processes exchange messages between disjoint intra-communicators. In our thesis, we focused on intra-communicators because the way inter-communicators work they do not suit our problem. Nevertheless, we stated them for better completeness.

Generally, a communicator is consisted of a *group* and a *context*. A *group* is an ordered collection of processes. Consider a group of p processes, then each process is attached with a unique, positive number from 0 to $p-1$ called rank. A *context* is a system-defined object that uniquely defines a communicator. In case of multiple communicators, each one has a distinct context, even if they have identical underlying groups. Contexts are used to insure that the messages received correctly.

In order to get a better understanding on how to create communicators and work with them, we will make a simple implementation. Assume a network with P=4 processes with ranks = 0,. . . ,3, that we want to divide in four groups. Suppose that in this network, both processes 0 and 1 are connected with processes 2,3. This is an example based on the work we did in our thesis, so that we can adapt D-ADMM on MPI. So in each group we have the root process and their neighbor-processes, with whom they communicate. This implementation is illustrated in Figure 4.3.

At the beginning we set an array, in which the neighbors of each process will be stored, so that we assign it to the new communicator. Then we create a group consisting of these processes. In order to do that, we use two commands. First we associate MPI_COMM_WORLD with a group, world_group, since this is the group from which the processes in the new groups will be taken. Then we create the groups with MPI_Group_Incl. In the end, we create the new communicators neigh_comm[i], with $i = 0, \ldots, 3$, by calling MPI_Comm_create, which associates the context with the new group. After creating the new communicators, the processes that belong in them can perform collective communication operations.

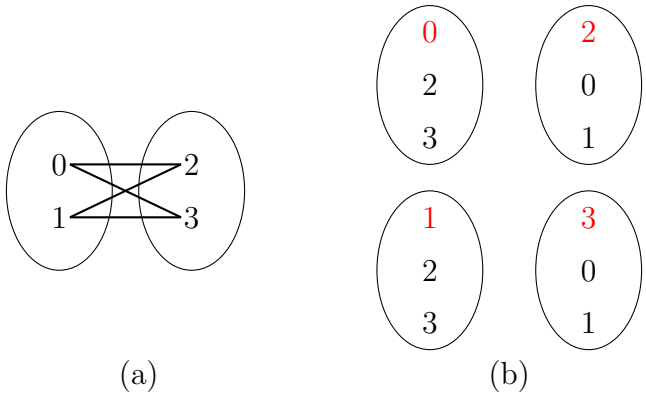


Figure 4.3: Network with one communicator (a) and same network with four communicators (b).

```
int* neighbors;
int size;
MPI_Group group_world;
MPI_Group neigh_group[P];
MPI_Comm neigh_comm[P];

/* Get "neighbors" of a process, including the root process */
find_neighbors(N, P, num_links, &neighbors);

/* Get the size of array neighbors */
find_size(neighbors, &size);

/* Get the group underlying MPI_COMM_WORLD */
MPI_Comm_group(MPI_COMM_WORLD, &group_world);

for(i=0;  i<P;  i++)
{
      /* create the new group */
      MPI_Group_incl(group_world, size, &neighbors, &neigh_group[i]);

      /* create the new communicator */
      MPI_Comm_create(MPI_COMM_WORLD, neigh_group[i], &neigh_comm[i]);
}
```

The syntax of the commands we used are the following. The first command

```
int MPI_Comm_group(MPI_Comm     comm   /* in */,
                   MPI_Group*   group  /* out */)
```

which simply returns the *group* underlying the communicator *comm*. The second command

```
int MPI_Group_incl(MPI_Group   old_group              /* in */,
                   int         new_group_size         /* in */,
                   int         ranks_in_old_group[ ]  /* in */,
                   MPI_Group*  new_group              /* out */)
```

creates a new group from a list of processes in existing group *old_group*. The number of processes in the new group is *new_group_size*, and the processes to be included, are listed in *ranks_in_old_group*. The final command

```
int MPI_Comm_create(MPI_Comm   old_comm   /* in */,
                    MPI_Group  new_group  /* in */,
                    MPI_Comm*  new_comm   /* out */)
```

associates a context with the group *new_group* and creates the communicator *new_comm*. All of the processes in *new_group* belong to the group underlying *old_comm*.

We should mention here, that the first two commands are local operations, while the last one is a collective operation. As for collective operations, we will look deep into them in the next section.

4.3.2 Distributed Graph Topologies

MPI offers the developer the facility to associate further information, beyond the group and the context with the communicator. Such information, like communication relationship between processes, is cached to the communicator and is called process topology. This mechanism simplifies the management of communication relations for the programmer, but also provides information about the application's communication behavior to the MPI implementation.

The interface that provides MPI-2.2 to specify the process communication topology, is the distributed graph topology, which is highly scalable to large communicators. It offers two interface variants the "adjacent" and the "general". The first interface requires each process to specify all its neighbors, while with the other, each

process can specify an arbitary edge in the graph. For our thesis, we used the "adjacent" interface, due to the third assumption we made for D-ADMM in section 3.13, that our network is static and does not vary with time. So when the applications start each process knows its neighbors.

The adjacent interface specifies each edge at two processes, once at the source process as an outgoing edge and once at the target process as an incoming edge. That may seem as a disadvantage, because it requires double memory during topology creation, but it does not require any communication in the creation routine. The syntax of the command, that creates the distributed graph communicator, is

```
int MPI_Dist_graph_create_adjacent(MPI_Comm     comm_old          /* in */,
                                   int          indegree          /* in */,
                                   const int    sources[ ]        /* in */,
                                   const int    sourceweights[ ]  /* in */,
                                   int          outdegree         /* in */,
                                   const int    destinations[ ]   /* in */,
                                   const int    destweights[ ]    /* in */,
                                   MPI_Info     info              /* in */,
                                   int          reorder           /* in */,
                                   MPI_Comm*    comm_dist_graph   /* out */)
```

This method receives *comm_old* and returns *comm_dist_graph*, which is a copy of the first, meaning that it includes the same processes, but with the topology information attached. It also has six parameters that refer to the calling process's local adjacency list. Parameters, *indegree* and *outdegree*, specify the number of incoming and outgoing edges respectively. The arrays *sources* and *destinations* are the lists with source and destination processes. Also, *sourceweights* and *destweights* are arrays defining the edge weights. Parameters *sources* and *sourceweights* are of size *indegree*, while *destweights* and *sourceweights* are of size *outdegree*. We should note here that, since D-ADMM can be seen as an undirected graph, when we create the distributed graph communicator, the parameters' values from sender's perspective are equal to the ones from receiver's perspective. As for the arrays sourceweights and destweights, they are all equal to 1, because every edge has the same weight. The alternative way is to replace these arrays with MPI_UNWEIGHTED. The parameter *reorder* can be used to point out whether the processes may be renumbered in the new communicator. Finally, parameter *info* can be used to provide additional information to the MPI implementation, but in our case there is no need for further information, so we set this parameter as MPI_INFO_NULL.

Consider the topology we showed in section 4.3.1 with the four processes. In this case, the values for MPI_Dist_graph_create_adjacent would be

process	indegree	sources	sources	destinations
0	2	2,3	2	2,3
1	2	2,3	2	2,3
2	2	0,1	2	0,1
3	2	0,1	2	0,1

Figure 4.4: Adjacent distributed graph example, with 4 processes.

Furthermore, we set *comm_old* as MPI_COMM_WORLD, *info* as MPI_INFO_NULL, *reorder* as 1, and the weight parameters as MPI_UNWEIGHTED.

4.4 Collective Communication

The main functions for sending and receiving messages with MPI are MPI_Send and MPI_Recv. But their disadvantage is that, at some point, there might exist idle processes, due to blocking communication. In order to avoid that, we can divide the load of work evenly among the processes and also have functions which are able to send or receive, to or from, multiple processes. So, a communication pattern that involves every process in a communicator is a collective communication. There are many collective operations, with different functionalities, but we will focus on those that had been used in our applications.

4.4.1 Broadcast

One really useful collective operation is MPI_Bcast. It is a function which includes a single process, with rank root, sending a copy of the same message to every other process that "lives" in the same communicator. Its syntax is

```
int MPI_Bcast(void*          message      /* in/out */,
              int            count        /* in */,
              MPI_Datatype   datatype     /* in */,
              int            root         /* in */,
              MPI_Comm       communicator /* in */)
```

As seen in the syntax of MPI_Bcast, the *message* is both an in and out parameter. That is because, in collective communication, the functions have to be called from

all processes in the same communicator. Assume that process 0 calls this function, so that it can send a message in the *communicator*. Then, every other process should also call the same function, in order to receive the message, using *root* = 0 and the same *count* and *datatype*. In order to make it clear, an implementation of MPI_Bcast with 4 processes, where process 0 sends the message, can be seen in the Figure 4.5.

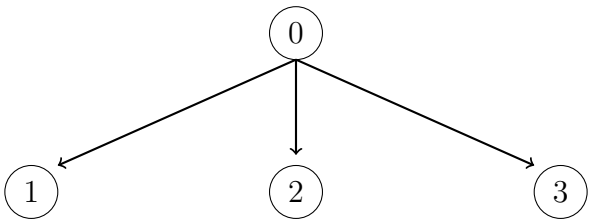


Figure 4.5: MPI_Bcast with 4 processes.

A possible code is the following,

```
// Process 0 sends double variable x to the other processes in MPI_COMM_WORLD
if(world_rank == 0)
    MPI_Bcast(&x, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
// Rest processes receive from 0, double variable x and store it in double variable y
else
    MPI_Bcast(&y, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

4.4.2 Allreduce

Another important collective operation is MPI_Allreduce. In many cases there is the need, every process in a communicator, to collect a result-message that occurred from an operation. Then MPI_Allreduce is being used with syntax

```
int MPI_Allreduce(void*          operand       /* in */,
                  void*          result        /* out */,
                  int            count         /* in */,
                  MPI_Datatype   datatype      /* in */,
                  MPI_Op         operator      /* in */,
                  MPI_Comm       communicator  /* in */)
```

In our case, as we will see in the next chapter with the applications, we used this function in order to compute a termination condition. In general, some of the operations that are supported are MPI_MAX, MPI_MIN, MPI_SUM, MPI_PRODUCT, etc.

In order to explain better, consider again 4 processes (circles) that need to operate (sum) over a variable (red rectangular) and keep the result on the same variable. As illustrated in Figure 4.6, the result of the sum is 18 and it is stored in the same memory allocation, on each process.

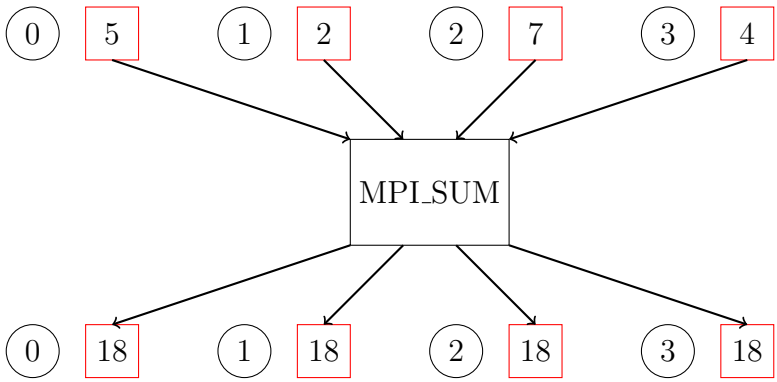


Figure 4.6: MPI_Allreduce with 4 processes.

There are more collective operations, if the reader wishes to look up in [7].

4.4.3 Neighborhood Collectives

The latest version of MPI, MPI-3, provides communication functions suitable for graph topologies, which are called neighborhood collective operations. The two basic methods are MPI_Neighbor_allgather and MPI_Neighbor_alltoall. There are also the extended versions of the above functions, that support different buffer sizes for each outgoing and incoming process's messages. These are MPI_Neighbor_allgatherv and MPI_Neighbor_alltotallv. They are vector neighborhood collectives and they allow users to specify different numbers of elements of the same type. Finally, there is the function MPI_Neighbor_alltoallw, that enables user to specify different datatypes for each incoming and outgoing neighbor and it can be used to enable efficient zero-copy communication in process neighborhoods.

Each neighborhood allgather function collects a message from every incoming neighbor and stores them in a contiguous buffer. These functions work in the same way as the MPI_Bcast in 4.4.1, but in a neighborhood, from a sender's perspective. On the other hand, in a neighborhood alltoall, each process specifies a different message buffer for each outgoing process and it receives into a different buffer from each incoming process.

For our applications we used only the MPI_Neighbor_allgather, but for better understanding on how graph topology works, we represent the syntax of both MPI_Neighbor_allgather and MPI_Neighbor_alltoall.

```
int MPI_Neighbor_allgather(const void*    sendbuf     /* in */,
                           int            sendcount   /* in */,
                           MPI_Datatype   sendtype    /* in */,
                           void*          recvebuf    /* in */,
                           int            recvcount   /* in */,
                           MPI_Datatype   recvtype    /* in */,
                           MPI_Comm       comm        /* out */)

int MPI_Neighbor_alltoall(const void*     sendbuf     /* in */,
                          int             sendcount   /* in */,
                          MPI_Datatype    sendtype    /* in */,
                          void*           recvebuf    /* in */,
                          int             recvcount   /* in */,
                          MPI_Datatype    recvtype    /* in */,
                          MPI_Comm        comm        /* out */)
```

Their functionality is illustrated in 4.7, based on the same scheme with the four processes.

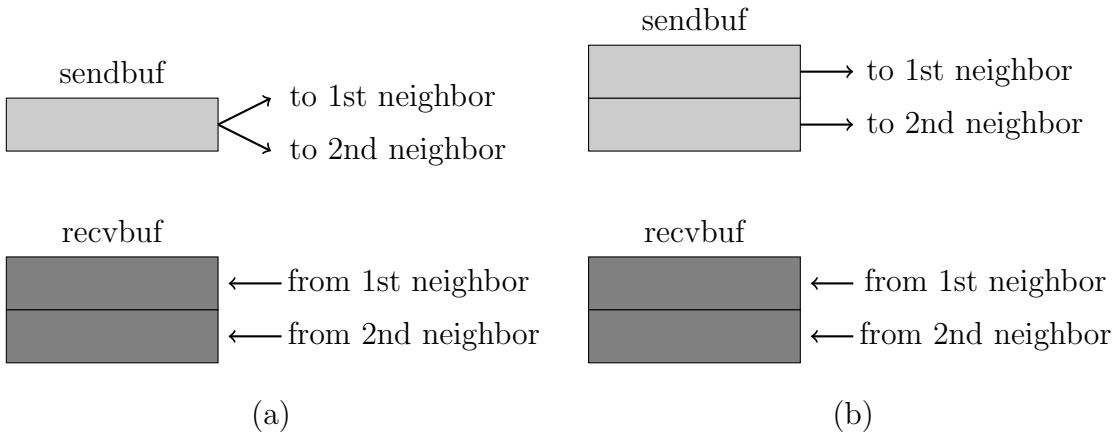


Figure 4.7: MPI_Neighbor_allgather (a) and MPI_Neighbor_alltoall (b).

Chapter 5

Experimental Results

In this chapter we present results, that were obtained from the MPI programs, implementing important optimization problems that have been recast as (3.31). We note that, in all the problems that will be examined, except consensus, none of the functions f_p is strongly convex. Therefore, D-ADMM is only guaranteed to converge under the first condition of Theorem 1 in section 3.13.4.

5.1 Performance Measure

As performance measure, we take the communication step, which occurs when all nodes have transmitted a vector of size n to their neighbors. The communication step increases until an arbitrary node converges. By converge, we mean that the relative error reduces to a small amount, $\left\|\mathbf{x}^{k+1}-\mathbf{x}^{k}\right\| / \left\|\mathbf{x}^{k}\right\| \leq \epsilon$, or a maximum number of iterations have been reached.

5.2 Setup

The programs are executed on Aris, a Greek supercomputer, deployed and operated by Greek Research and Technology Network(GRNET) [10]. Aris provides 44 nodes, where each node has 4 Intel(R) Xeon(R) CPU E5-4650v2 processors at 2.4 GHz and also each processor has 10 cores. Every node has memory of 512Gb.

For many matrix/vector operations we used routines of the Eigen library, which is a C++ template library for linear algebra [9].

Furthermore, for all optimization problems we used all three communicators that we discussed in chapter 4. Meaning that, in the first implementation, all processes communicate in the MPI_COMM_WORLD by using the basic Message Passing functions, MPI_Send and MPI_Recv. For the next implementation we created as many intra-communicators, as the number of processes we want to communicate. The exchange of messages in the intra-communicators was achieved through collective operations. As for the last implementation, we created a distributed graph topology, using the adjacent interface, where the communication was managed with the neighborhood collective operation MPI_Neighbor_allgather.

In our work all of the networks that have been used, have a coloring scheme of

$C = 2$ colors. But, we generated two types of networks. One is a fully connected network and the other a partially connected network, with "random" neighbors. In addition, we examined our implementations for different number of P nodes at each network type. The set for P is $\{2, 10, 20, 50, 100\}$.

Parameter ρ affects strongly ADMM-based algorithms. Hence, to make a fair comparison, we tested our implementations with several values of ρ, taken from the set $\{10^{-3}, 10^{-2}, 10^{-1}, 1, 10, 10^{2}\}$. In the end, we picked the one that yields the best result.

We should note that, all data were created in Matlab, but the way that were created will be discussed in the upcoming sections. Also, in the following figures $\mathbf{x}^*$ denotes the solution of the problems we examined, obtained by CVX [11].

5.3 Consensus

Given a network of P nodes we want to solve:

$$\underset{\mathbf{x}\in\mathbb{R}^n}{\text{minimize}} \quad f(\mathbf{x}) = \frac{1}{2}\left\|\mathbf{A}\mathbf{x} - \mathbf{b}\right\|^2, \tag{5.1}$$

which is clearly an unconstrained version of 3.31. $\mathbf{A} \in \mathbb{R}^{m\times n}$ full column-rank and $\mathbf{b} \in \mathbb{R}^m$, with elements drawn from the standard normal distribution. We will solve the problem with row partition visualized in Figure 5.1,

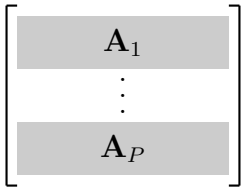


Figure 5.1: Row partition of $\mathbf{A}$ of P blocks, where a block is a set of rows.

So in each node p a row-block of $\mathbf{A}$ and $\mathbf{b}$ will be stored, $\mathbf{A}_p$ and $\mathbf{b}_p$ respectively. The number of rows each node will have is m/P. To be more specific, node 1 will have rows from 1 to m/P, node 2 will have rows $m/P+1$ to $2m/P$ and so on. Also, $f_p(\mathbf{x}) = (1/2)\left\|\mathbf{A}_p\mathbf{x} - \mathbf{b}_p\right\|^2$. Thus, it can be solved with D-ADMM. The problem of

step 4 of D-ADMM algorithm, in this case, is the following

$$\begin{aligned}
h(\mathbf{x}_p) &= f_p(\mathbf{x}_p) + \mathbf{v}_p^{k^T}\mathbf{x}_p + \frac{\rho D_p}{2}\|\mathbf{x}_p\|^2 \\
&= \frac{1}{2}\|\mathbf{A}_p\mathbf{x}_p - \mathbf{b}_p\|^2 + \mathbf{v}_p^{k^T}\mathbf{x}_p + \frac{\rho D_p}{2}\|\mathbf{x}_p\|^2 \\
&= \frac{1}{2}(\mathbf{A}_p\mathbf{x}_p - \mathbf{b}_p)^T(\mathbf{A}_p\mathbf{x}_p - \mathbf{b}_p) + \mathbf{v}_p^{k^T}\mathbf{x}_p + \frac{\rho D_p}{2}\mathbf{x}_p^T\mathbf{x}_p \\
&= \frac{1}{2}\mathbf{x}_p^T\mathbf{A}_p^T\mathbf{A}_p\mathbf{x}_p - \mathbf{b}_p^T\mathbf{A}_p\mathbf{x}_p + \frac{1}{2}\mathbf{b}_p^T\mathbf{b}_p + \mathbf{v}_p^{k^T}\mathbf{x}_p + \frac{\rho D_p}{2}\mathbf{x}_p^T\mathbf{x}_p \\
&= \frac{1}{2}\mathbf{x}_p^T(\mathbf{A}_p^T\mathbf{A}_p + \rho D_p\mathbf{I})\mathbf{x}_p + (\mathbf{v}_p^k - \mathbf{A}_p^T\mathbf{b}_p)^T\mathbf{x}_p + \frac{1}{2}\mathbf{b}_p^T\mathbf{b}_p \\
&= \frac{1}{2}\mathbf{x}_p^T\mathbf{Q}_p\mathbf{x}_p + \mathbf{q}_p^{k^T}\mathbf{x}_p + \frac{1}{2}\mathbf{b}_p^T\mathbf{b}_p
\end{aligned}$$

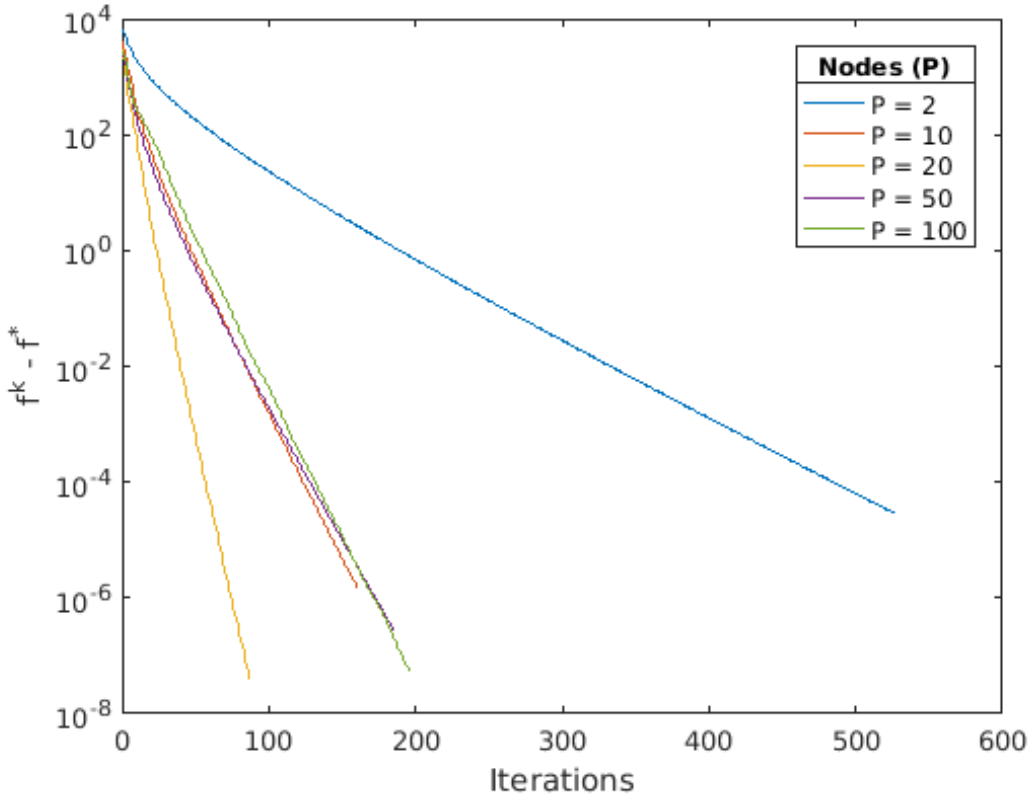
where $\mathbf{Q}_p = \mathbf{A}_p^T\mathbf{A}_p + \rho D_p\mathbf{I}$, and $\mathbf{q}_p^k = \mathbf{v}_p^k - \mathbf{A}_p^T\mathbf{b}_p$. When each node learns its private variables, they cache matrix $\mathbf{Q}_p$ and vector $\mathbf{A}_p^T\mathbf{x}_p$, in order to speed up the algorithm. Furthermore, the minimization step has a closed-form solution, which is computed with KKT conditions with no constraints.

$$\begin{aligned}
\nabla_{\mathbf{x}_p} h(\mathbf{x}_p) = \mathbf{0} \Rightarrow \\
\mathbf{Q}_p\mathbf{x}_p + \mathbf{q}_p = \mathbf{0}
\end{aligned}$$

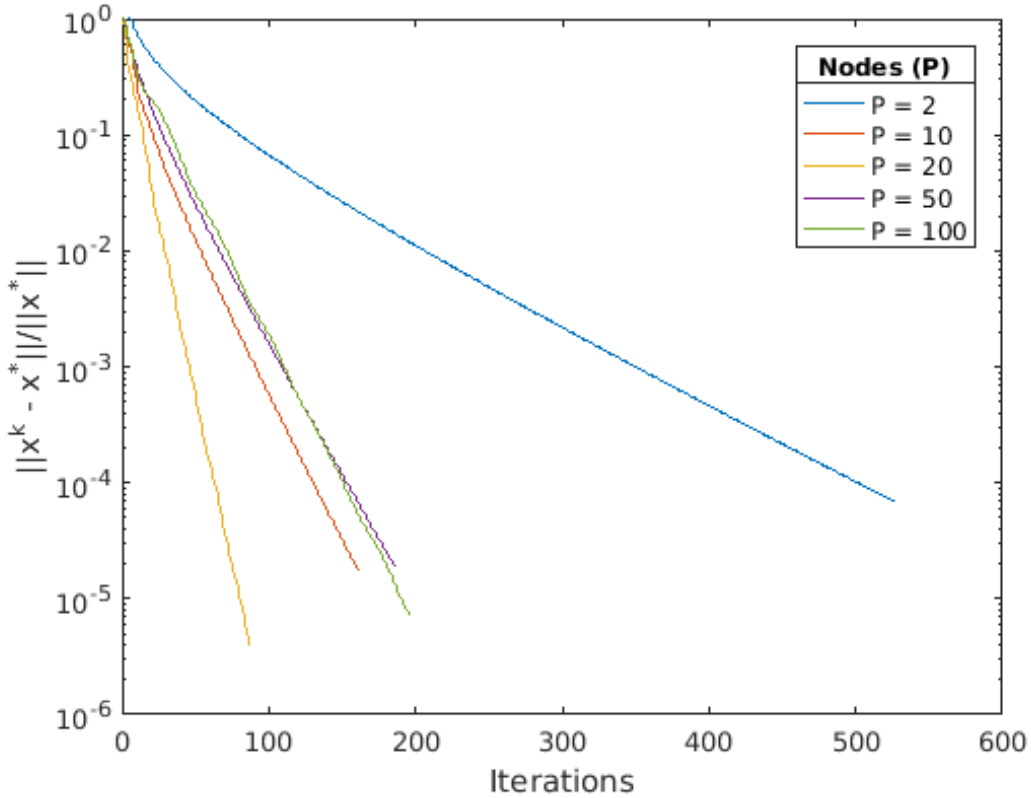
$\mathbf{Q}_p$ is invertible, since $\mathbf{A}_p$ is a full-column rank. So the $\mathbf{x}$-update at $(k+1)$-iteration is

$$\mathbf{x}_p^{k+1} = (\mathbf{A}_p^T\mathbf{A}_p + \rho D_p\mathbf{I})^{-1}(\mathbf{A}_p^T\mathbf{b}_p - \mathbf{v}_p^k).$$

In Figure 5.2 and Figure 5.3, we illustrate the convergence we have reached for both types of networks. Matrix $\mathbf{A}$ and vector $\mathbf{b}$, have dimensions $m = 15000$, $n = 5000$, relative error $\epsilon = 10^{-6}$ and maximum iterations $= 3000$ and all variations for variables P and ρ.

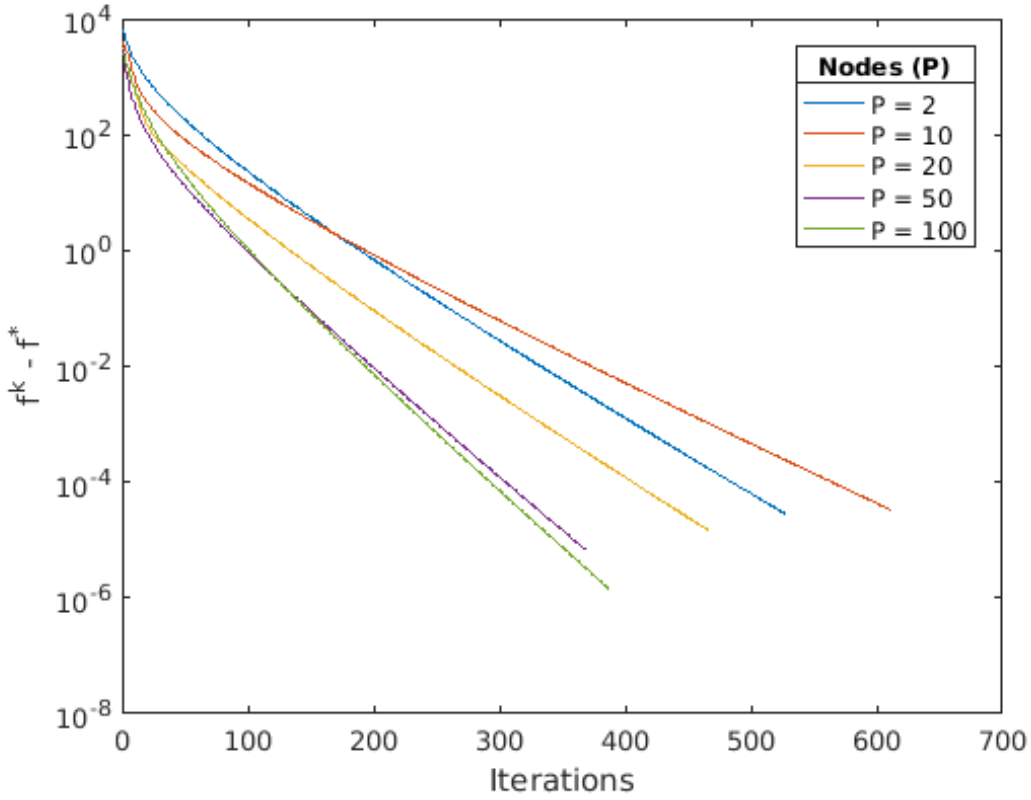


(a) Distance of f from f^*.

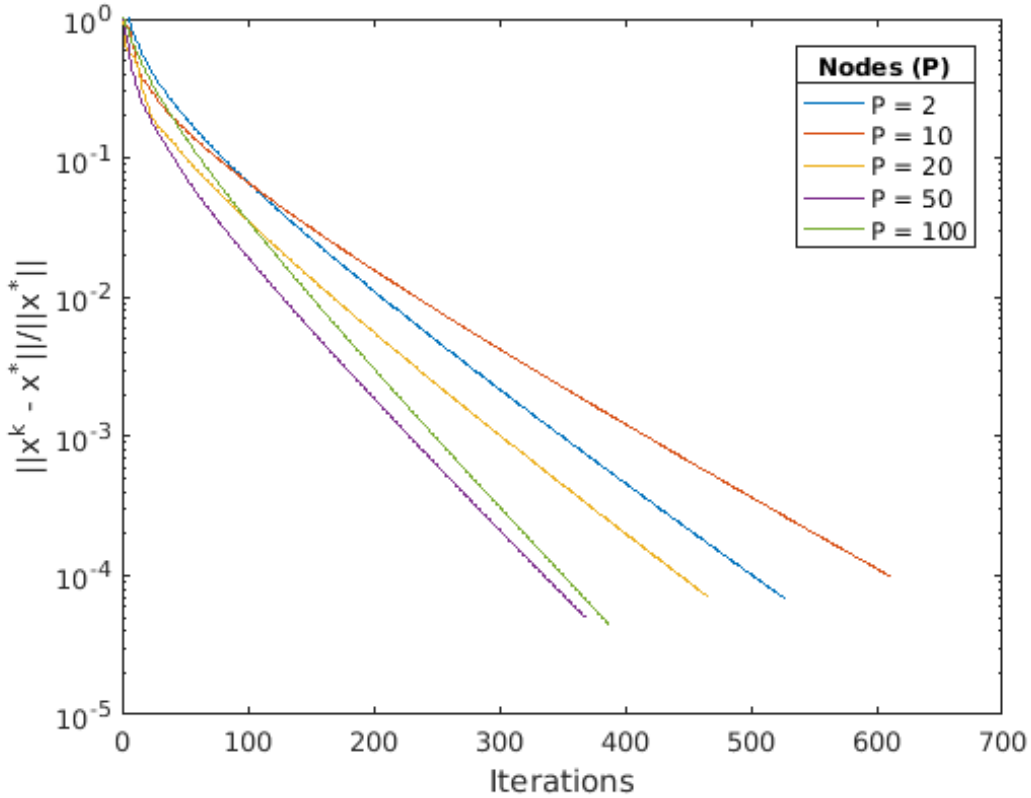


(b) Relative error from $\mathbf{x}^*$.

Figure 5.2: Fully connected networks.



(a) Distance of f from f^*.



(b) Relative error from $\mathbf{x}^*$.

Figure 5.3: Partially connected networks.

5.4 Lasso

Finding sparse solutions of linear systems is important in many areas, including statistics, compressed sensing etc. A common approach to tackle this problem is by solving Lasso, as discussed in section 3.11.1.

$$\underset{\mathbf{x}\in\mathbb{R}^n}{\text{minimize}} \quad (1/2)\left\|\mathbf{A}\mathbf{x}-\mathbf{b}\right\|_2^2,$$

where $\mathbf{A}\in\mathbb{R}^{m\times n}$ full column-rank, $\mathbf{b}\in\mathbb{R}^m$, and parameter $\lambda>0$ is known to every node. We will solve the problem again with row partition. In substance, our problem can be rewritten as

$$\underset{\mathbf{x}\in\mathbb{R}^n}{\text{minimize}} \quad \frac{1}{2}\sum_{p=1}^{P}\Big(\left\|\mathbf{A}_p\mathbf{x}-\mathbf{b}_p\right\|_2^2+\frac{\lambda}{P}\left\|\mathbf{x}\right\|_1\Big),$$

which is also an unconstrained version of 3.31. Also, $f_p(\mathbf{x})=(1/2)\left\|\mathbf{A}_p\mathbf{x}-\mathbf{b}_p\right\|_2^2+(\lambda/P)\left\|\mathbf{x}\right\|_1$. Now, the minimizations problem of step 4 of D-ADMM algorithm, is

$$\begin{aligned}
h(\mathbf{x}_p) &= f_p(\mathbf{x}_p)+\mathbf{v}_p^{kT}\mathbf{x}_p+\frac{\rho D_p}{2}\left\|\mathbf{x}_p\right\|^2\\
&= \frac{1}{2}\left\|\mathbf{A}_p\mathbf{x}_p-\mathbf{b}_p\right\|^2+\frac{\lambda}{P}\left\|\mathbf{x}_p\right\|_1+\mathbf{v}_p^{kT}\mathbf{x}_p+\frac{\rho D_p}{2}\left\|\mathbf{x}_p\right\|^2\\
&= \frac{1}{2}(\mathbf{A}_p\mathbf{x}_p-\mathbf{b}_p)^T(\mathbf{A}_p\mathbf{x}_p-\mathbf{b}_p)+\frac{\lambda}{P}\left\|\mathbf{x}_p\right\|_1+\mathbf{v}_p^{kT}\mathbf{x}_p+\frac{\rho D_p}{2}\mathbf{x}_p^T\mathbf{x}_p\\
&= \frac{1}{2}\mathbf{x}_p^T\mathbf{A}_p^T\mathbf{A}_p\mathbf{x}_p-\mathbf{b}_p^T\mathbf{A}_p\mathbf{x}_p+\frac{1}{2}\mathbf{b}_p^T\mathbf{b}_p+\frac{\lambda}{P}\left\|\mathbf{x}_p\right\|_1+\mathbf{v}_p^{kT}\mathbf{x}_p+\frac{\rho D_p}{2}\mathbf{x}_p^T\mathbf{x}_p\\
&= \frac{1}{2}\mathbf{x}_p^T(\mathbf{A}_p^T\mathbf{A}_p+\rho D_p\mathbf{I})\mathbf{x}_p+(\mathbf{v}_p^k-\mathbf{A}_p^T\mathbf{b}_p)^T\mathbf{x}_p+\frac{\lambda}{P}\left\|\mathbf{x}_p\right\|_1+\frac{1}{2}\mathbf{b}_p^T\mathbf{b}_p\\
&= \frac{1}{2}\mathbf{x}_p^T\mathbf{Q}_p\mathbf{x}_p+\mathbf{q}_p^{kT}\mathbf{x}_p+r+\frac{\lambda}{P}\left\|\mathbf{x}_p\right\|_1
\end{aligned}$$

with $\mathbf{Q}_p=\mathbf{A}_p^T\mathbf{A}_p+\rho D_p\mathbf{I}$, $\mathbf{q}_p^k=\mathbf{v}_p^k-\mathbf{A}_p^T\mathbf{b}_p$, and $r=(1/2)\mathbf{b}_p^T\mathbf{b}_p$. So it is still a Lasso problem, with a quadratic and an l_1 term. The minimization of this problem has no closed-form solution, so we will find $\mathbf{x}_p^*$, using ADMM algorithm.

So the minimization problem in ADMM form, applied to each node, can be written as follows

$$\begin{aligned}
&\underset{\mathbf{x}_p,\mathbf{z}_p}{\text{minimize}} \quad \phi(\mathbf{x}_p)+g(\mathbf{z}_p)\\
&\text{subject to} \quad \mathbf{x}_p-\mathbf{z}_p=\mathbf{0},
\end{aligned}$$

where $\phi(\mathbf{x}_p)=(1/2)\mathbf{x}_p^T\mathbf{Q}_p\mathbf{x}_p+\mathbf{q}_p^{kT}\mathbf{x}_p+r$ and $g(\mathbf{z}_p)=(\lambda/P)\left\|\mathbf{z}_p\right\|_1$. As we saw in section 3.3, we will form the augmented Lagrangian in scaled form for our problem. We refer to ρ parameter of ADMM algorithm as ρ', to avoid conflict with the ρ

parameter of D-ADMM.

$$\begin{aligned} L_{\rho'}(\mathbf{x}_p, \mathbf{z}_p, \mathbf{u}_p) = &\ \phi(\mathbf{x}_p) + g(\mathbf{z}_p) + (\rho'/2)\left\|\mathbf{x}_p - \mathbf{z}_p + \mathbf{u}_p\right\|_2^2 - (\rho'/2)\left\|\mathbf{u}_p\right\|_2^2 \\ = &\ (1/2)\mathbf{x}_p^T\mathbf{Q}_p\mathbf{x}_p + \mathbf{q}_p^{k^T}\mathbf{x}_p + r + (\lambda/P)\left\|\mathbf{z}_p\right\|_1 + \\ &\ (\rho'/2)(\mathbf{x}_p - \mathbf{z}_p + \mathbf{u}_p)^T(\mathbf{x}_p - \mathbf{z}_p + \mathbf{u}_p) - (\rho'/2)\mathbf{u}_p^T\mathbf{u}_p \\ = &\ (1/2)\mathbf{x}_p^T(\mathbf{Q}_p + \rho'\mathbf{I})\mathbf{x}_p + \left(\mathbf{q}_p^k + \rho'(\mathbf{u}_p - \mathbf{z}_p)\right)^T\mathbf{x}_p + r + (\lambda/P)\left\|\mathbf{z}_p\right\|_1 + \\ &\ (\rho'/2)\mathbf{z}_p^T\mathbf{z}_p - \rho'\mathbf{u}_p^T\mathbf{z}_p \end{aligned}$$

The $\mathbf{x}_p-$update,

$$\begin{aligned} \nabla_{\mathbf{x}_p} L_{\rho'}(\mathbf{x}_p, \mathbf{z}_p^k, \mathbf{u}_p^k) = \mathbf{0} \Rightarrow \\ (\mathbf{Q}_p + \rho'\mathbf{I})\mathbf{x}_p + \mathbf{q}_p^k + \rho'(\mathbf{u}_p - \mathbf{z}_p) = \mathbf{0} \end{aligned}$$

and because matrix $\mathbf{Q}_p + \rho'\mathbf{I}$ is invertible, we have,

$$\mathbf{x}_p^{k+1} = (\mathbf{Q}_p + \rho'\mathbf{I})^{-1}\left(\rho'(\mathbf{z}_p - \mathbf{u}_p) - \mathbf{q}_p^k\right).$$

For the $\mathbf{z}_p-$update, because l_1 norm is not differentiable, we will use subgradient and the soft thresholding operator.

$$\begin{aligned} \nabla_{\mathbf{z}_p} L_{\rho'}(\mathbf{x}_p^{k+1}, \mathbf{z}_p, \mathbf{u}_p^k) = \mathbf{0} \Rightarrow \\ \partial\left((\lambda/P)\left\|\mathbf{z}_p\right\|_1\right) + \rho'\mathbf{z}_p - \rho'(\mathbf{x}_p + \mathbf{u}_p) = \mathbf{0} \Rightarrow \\ \partial\left((\lambda/P\rho')\left\|\mathbf{z}_p\right\|_1\right) + \mathbf{z}_p - \mathbf{x}_p + \mathbf{u}_p = \mathbf{0} \end{aligned}$$

The scalar z_p^i-update

$$(\lambda/P\rho')\partial(|z_p^i|) + z_p^i - x_p^i + u_p^i = 0$$

$$z_p^i = \begin{cases} u_p^i + x_p^i - \lambda/P\rho', & \text{if } z_p^i > 0 \\ u_p^i + x_p^i + \lambda/P\rho', & \text{if } z_p^i < 0 \\ |u_p^i + x_p^i| \leq \lambda/P\rho', & \text{if } z_p^i = 0 \end{cases}$$

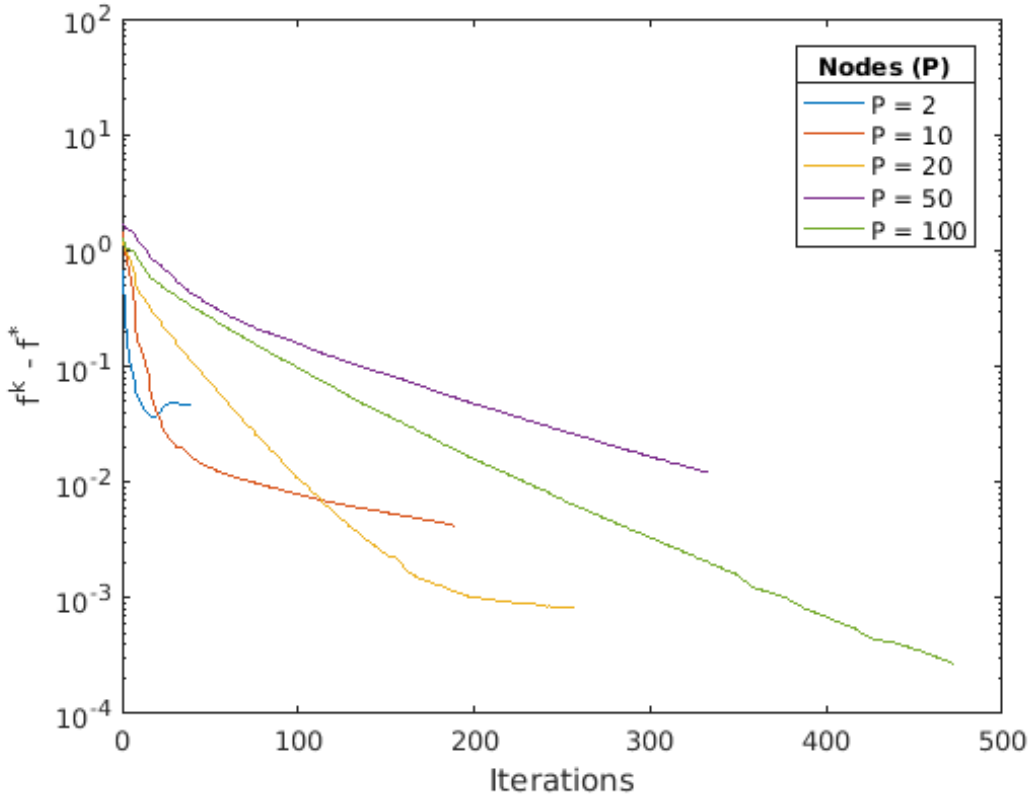
So, we can use the soft thresholding operator for the $\mathbf{z}_p-$update as,

$$\mathbf{z}_p^{k+1} = S_{\lambda/\rho' P}(\mathbf{x}_p^{k+1} - \mathbf{u}_p^k)$$

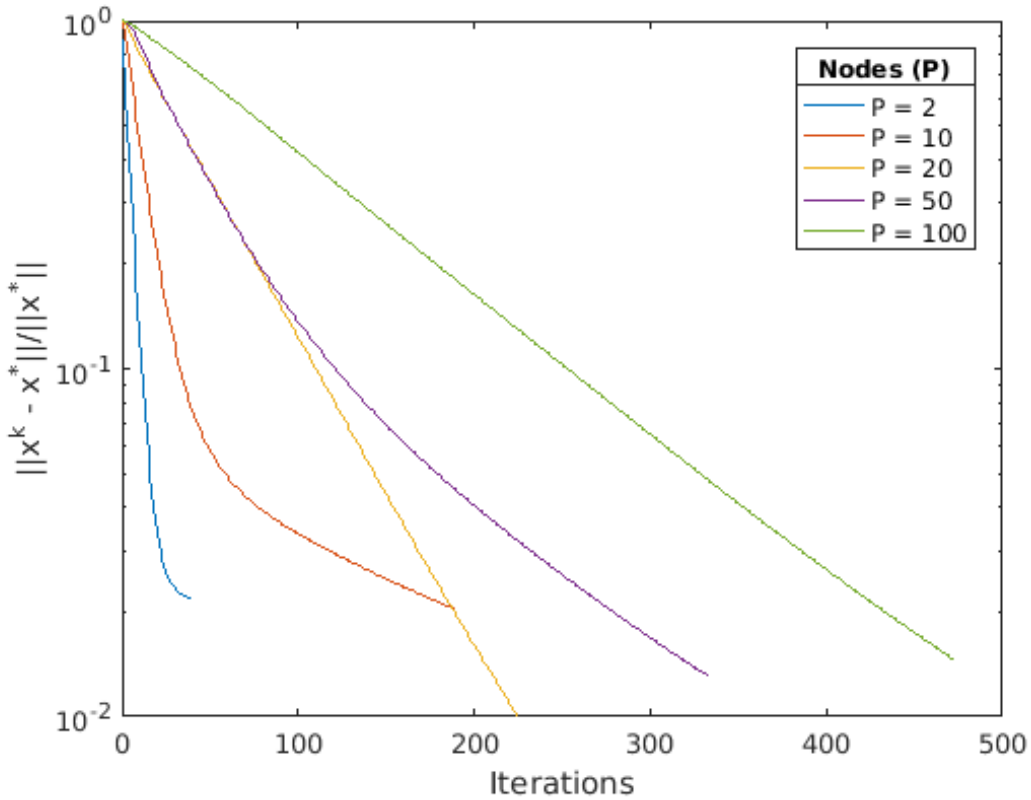
Finally, the $\mathbf{u}_p-$update,

$$\mathbf{u}_p^{k+1} = \mathbf{u}_p^k + \mathbf{x}_p^{k+1} - \mathbf{z}_p^{k+1}$$

In Figure 5.4 and Figure 5.5, we plot the convergence we have reached for both types of networks, with matrix $\mathbf{A}$ and vector $\mathbf{b}$ having dimensions, $m = 500$, $n = 2000$. We generate the data as follows. We first choose A_{ij} from $\mathcal{N}(0, 1)$ and then normalize the columns to have unit l_2 norm. A 'true' value $\mathbf{x}^{\text{true}} \in \mathbb{R}^n$ is generated with 3% nonzero entries, each sampled from an $\mathcal{N}(0, 1)$. The labels $\mathbf{b}$ are then computed as $\mathbf{b} = \mathbf{A}\mathbf{x}^{\text{true}} + \mathbf{w}$, where $\mathbf{w}$ corresponds to a signal-to-noise ratio, drawn from $\mathcal{N}(0, 10^{-3}\mathbf{I})$. Furthermore, relative error $\epsilon = 10^{-4}$, maximum iterations = 1000, $\lambda = 0.3$ and $\rho' = 1$. Take into consideration that we used the same data for all variations of P and ρ.

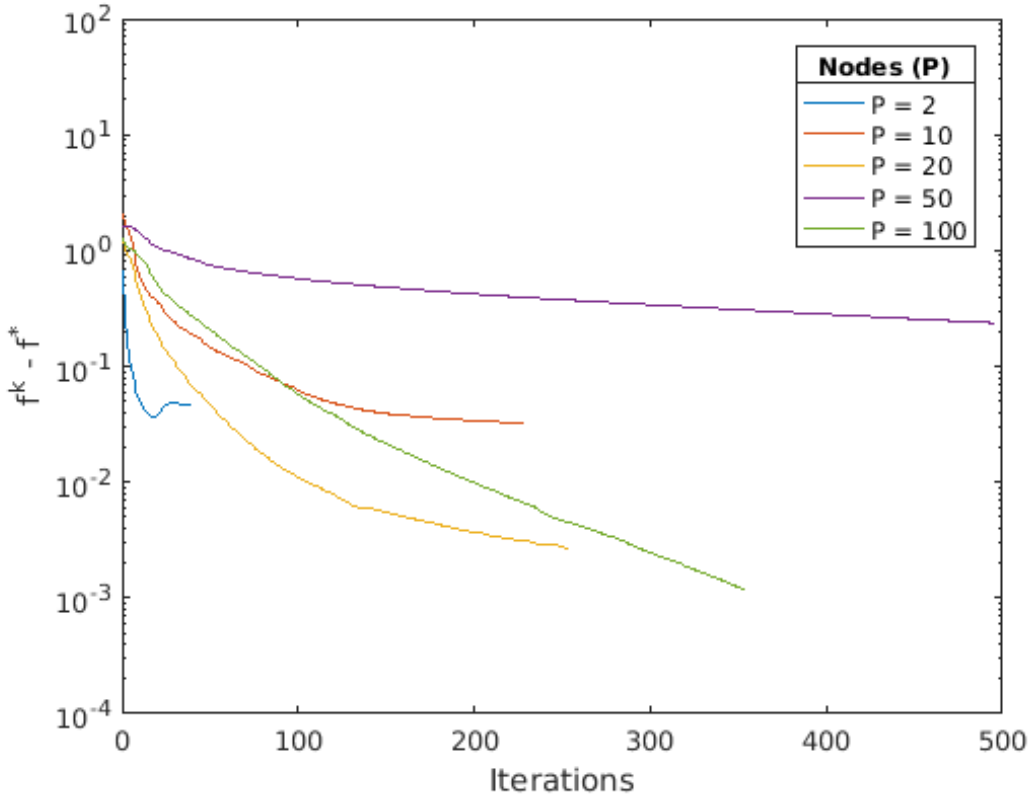


(a) Distance f from f^*.

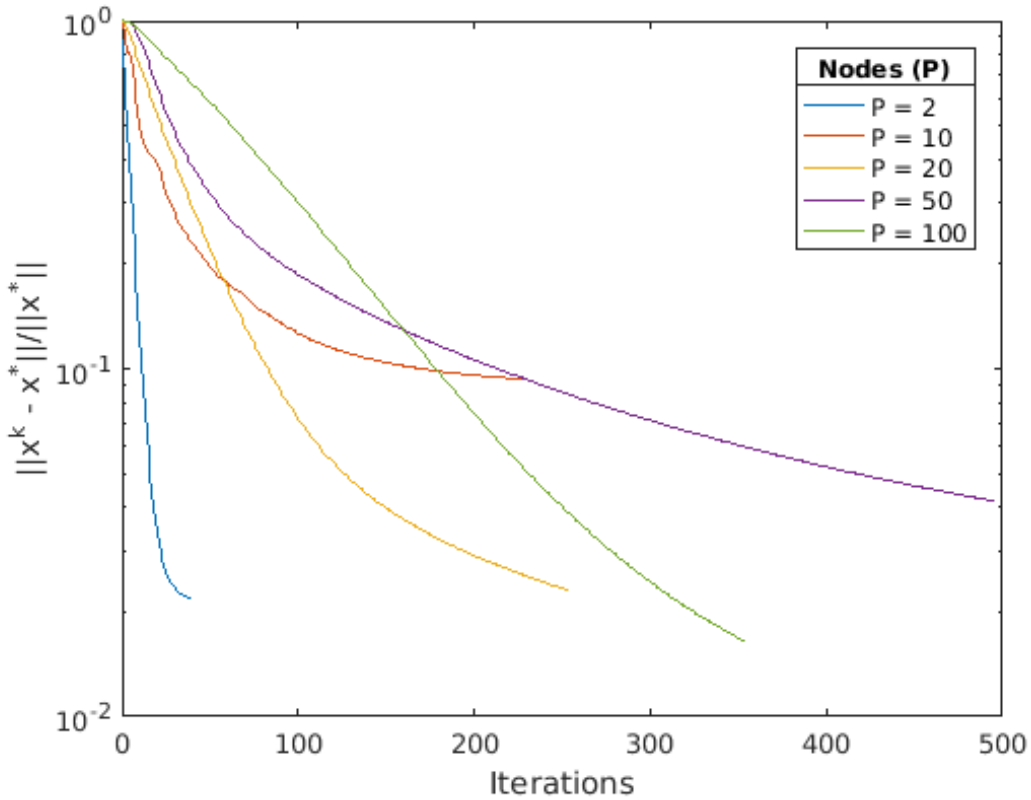


(b) Relative error from $\mathbf{x}^*$.

Figure 5.4: Fully connected networks.



(a) Distance f from f^*.



(b) Relative error from $\mathbf{x}^*$.

Figure 5.5: Partially connected networks.

Chapter 6

Conclusion

In this thesis, we consider convex optimization problems that can be expressed as the sum of P convex functions, where each function is associated with a private convex set and the optimal vector lies in the intersection of these sets. We examined the D-ADMM algorithm, which solves this type of convex optimization problems, and developed three implementations for parallel applications with MPI. It was shown that D-ADMM converges for all types of network we tested, in a relatively small number of iterations.

Bibliography

[1] J. F. Mota, J. M. Xavier, P. M. Aguiar, and M. Puschel, *"Distributed basis pursuit"*. IEEE Transactions on Signal Processing, 60(4), 1942-1956. 2012.

[2] J. F. Mota, J. M. Xavier, P. M. Aguiar, and M. Puschel, *"D-ADMM: A communication-efficient distributed algorithm for separable optimization"*. IEEE Transactions on Signal Processing, 61(10), 2718-2723, 2013.

[3] S. Boyd, L. Vandenberghe, *"Convex optimization"*. Cambridge university press, 2004.

[4] A. P. Liavas , *"Convex Optimization Lecture Notes,"* 2015.

[5] 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, 3(1), 1-122, 2011.

[6] D. Han, and X. Yuan, *"A note on the alternating direction method of multipliers"*. Journal of Optimization Theory and Applications, 155(1), 227-238, 2012.

[7] P. S. Pacheco, *"Parallel programming with MPI"*. Morgan Kaufmann, 1997.

[8] W. Gropp, T. Hoefler, R. Thakur, and E. Lusk, *"Using advanced MPI: Modern features of the message-passing interface"*. MIT Press, 2014.

[9] *"Eigen Library"*, `http://eigen.tuxfamily.org`.

[10] *"Greek Research and Technology Network"*, `https://grnet.gr/`.

[11] M. Grant and S. Boyd, *"Matlab Software for Disciplined Convex Programming, version 2.1"*, `http://cvxr.com/cvx`, 2014.

[12] F. Morbidi, *"The deformed consensus protocol"*. Automatica, 49(10), 3049-3055, 2013.