TECHNICAL UNIVERSITY OF CRETE
ELECTRICAL AND COMPUTER ENGINEERING DEPARTMENT

Study and implementation of distributed asynchronous algorithms for convex optimization

by

Thalia - Anastasia Stavrianoudaki

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

ELECTRICAL AND COMPUTER ENGINEERING

presentation date

THESIS COMMITTEE

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

Abstract

We consider a convex optimization problem with a quadratic cost function. We partition the data of the problem into a set of processors. We use the Message Passing Interface (MPI) and develop parallel implementations of two iterative methods for the solution of the optimization problem, the Gradient Descent and the Block Coordinate Descent (BCD). We test the convergence properties of the algorithms under various circumstances, by calculating the speed of convergence and the total communication cost.

Acknowledgements

I would like to thank my family, my boyfriend and my friends, for their unconditional love and support and for being always there for me.

I would also like to thank my supervisor, Professor Athanasios Liavas, for his guidance and advice throughout this work.

Contents

List of Figures **7**

List of Tables **8**

1 Introduction **9**

1.1 Motivation 9
1.2 Notations 10
1.3 Thesis Outline 10

2 Convex Optimization **11**

2.1 Convex Sets 11
2.2 Convex Functions 12
2.3 Gradient and Hessian of function f 13
2.4 First and Second order Taylor approximations 13
2.4.1 First-order condition of convexity 14
2.4.2 Second-order condition of convexity 15
2.5 Convex Optimization Problems 15
2.5.1 Constrained optimization problems 15
2.5.2 Unconstrained optimization problems 16
2.6 Solving unconstrained optimization problems 16
2.6.1 Descent methods 17
2.6.2 Gradient Direction 17
2.7 Convergence for strongly convex functions 18

3 Message Passing Interface **19**
3.1 Multiple Instruction - Multiple Data Systems . . . 19
3.1.1 Distributed-Memory Systems . . . 19
3.2 Distributed-Memory APIs . . . 20
3.3 SPMD programs . . . 21
3.4 Basic MPI functions . . . 21
3.4.1 MPI_Init and MPI_Finalize . . . 21
3.4.2 Communicators . . . 21
3.4.3 MPI_Send and MPI_Recv . . . 22
3.5 I/O in MPI . . . 24
3.5.1 Output . . . 24
3.5.2 Input . . . 24
3.6 Collective Communication . . . 24
3.6.1 MPI_Bcast . . . 25
3.6.2 MPI_Scatter . . . 25
3.6.3 MPI_Allgather . . . 26
3.7 Taking timings . . . 26

4 Optimization Algorithms **28**
4.1 The quadratic problem . . . 28
4.2 Setup . . . 29
4.3 Controling the condition number of f . . . 30
4.4 Gradient Descent method . . . 31
4.4.1 Step size λ . . . 33
4.4.2 Terminating condition . . . 33
4.5 Block Coordinate Descent method . . . 33
4.5.1 Terminating condition . . . 35

5 Distributed Asynchronous Iterative Algorithms **36**
5.1 Parallelizing the data . . . 36
5.2 Gradient method in MPI . . . 38
5.3 Block Coordinate Descent in MPI . . . 40
5.3.1 Dividing the data among the processes . . . 40

5.3.2 Conditions of convergence 41

6 Experimental Results **44**

6.1 Set up 44

6.2 Convergence of MPI algorithms 44

6.2.1 Convergence of Gradient Descent 45

6.2.2 Convergence of Block Coordinate Descent 47

6.3 Effect of condition number K 49

6.4 Introducing probability p 51

6.5 Effect of probability p 52

6.6 Comparison of the methods 55

6.7 Communication Cost 55

7 Conclusion **58**

List of Figures

2.1 Convex function 13
2.2 Convexity proven from first - order Taylor approximation 14

3.1 A distributed-memory system 20

4.1 Gradient Descent Method on a series of level sets 32

5.1 Row partition of matrix $\mathbf{P}$ and vector $\mathbf{q}$ 37

6.1 Gradient Descent method with $n = 500$ and $K = 10$ 45
6.2 Gradient Descent method with $n = 1000$ and $K = 10$ 45
6.3 Gradient Descent method with $n = 1600$ and $K = 10$ 46
6.4 Block Coordinate Descent method with $n = 500$ and $K = 10$ 47
6.5 Block Coordinate Descent method with $n = 1000$ and $K = 10$ 48
6.6 Block Coordinate Descent method with $n = 1600$ and $K = 10$ 48
6.7 Distance of f from p_* when $n = 500$ and $K = 100$ 50
6.8 Distance of f from p_* when $n = 500$ and $K = 1000$ 50
6.9 Distance of f from p_* with $p = \{0.1, 0.3, 0.5, 0.8, 1\}$, when $n = 500$, $np = 2$ and $K = 10$ 53
6.10 Distance of f from p_* with $p = \{0.1, 0.3, 0.5, 0.8, 1\}$, when $n = 1000$, $np = 2$ and $K = 10$ 54
6.11 Average number of iterations and communication cost for Gradient Descent 56
6.12 Average number of iterations and communication cost for Block Coordinate Descent 57

List of Tables

Parallel Gradient Descent Algorithm 39
Parallel Block Coordinate Descent algorithm 43

Parallel algorithm with probability p 52

Chapter 1

Introduction

1.1 Motivation

It is a common observation that, recently, the size of problems we need to solve has significantly increased. Instead of building more complex processors, which will be able to solve such problems, we may put multiple simple processors in one chip. Such a change has created the need for building parallel programs, in order to make use of the multiple processors and achieve faster solutions.

In this thesis, we implement two algorithms for solving convex (in fact, quadratic) optimization problems in parallel, the Gradient Descent and the Block Coordinate Descent (BCD). In both algorithms, the data are divided evenly among the processors, so that each processor has to cope with a smaller problem.

In order to implement the parallel algorithms, we use the Message Passing Interface (MPI), a library of functions for programming parallel systems.

We test the algorithms with various inputs and under various conditions, and we present some experimental results.

1.2 Notations

Capital bold letters denote matrices; small bold letters denote vectors; small letters denote scalars; $(\cdot)^T$ determines transposition; $(\cdot)^{-1}$ will be the inverse of a matrix; $\mathbf{I}_n$ denotes the $(n \times n)$ identity matrix; $|| \cdot ||_2$ denotes the Euclidean norm;

1.3 Thesis Outline

This thesis is organized as follows:

• Chapter 2 introduces the basic concepts of Convex Optimization.

• Chapter 3 introduces Message Passing Interface.

• Chapter 4 presents the basic concepts behind the *Gradient Descent* and the *Block Coordinate Descent* methods.

• Chapter 5 considers the MPI implementation of both methods.

• Chapter 6 presents experimental results derived from the implementation of the algorithms presented in Chapter 5.

Chapter 2

Convex Optimization

Convex Optimization is the minimization of a convex function over a convex set. Convexity implies that any local minimum is also a global minimum, which guarantees that local search algorithms lead to optimal solutions. Convex Optimization arises in scientific and engineering applications, such as automatic control systems, signal processing, communications and networks, biomedical engineering, electronic circuit design, finance, statistics, etc.

2.1 Convex Sets

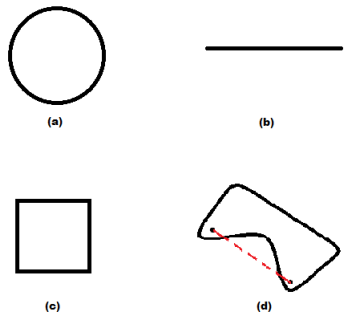
An **affine set** $\mathbb{C} \subseteq \mathbb{R}^n$ is a set for which the straight line joining every the pair of points of the set lies in the set. So, for $\mathbf{x}, \mathbf{y} \in \mathbb{C}$ and $\theta \in \mathbb{R}$,

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

A set $\mathbb{C} \subseteq \mathbb{R}^n$ is called convex if, for every pair of points within the set, every point on the straight line segment joining the pair of points is also within the set. So, for $\mathbf{x}, \mathbf{y} \in \mathbb{C}$ and $0 \leqslant \theta \leqslant 1$,

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

Some examples of convex and non-convex sets are shown in the figure below:



Convex sets (a), (b), (c) and non-convex set (d)

If a set if affine, then it is convex, while the converse is not necessarily true. For example, straight lines are affine, and therefore, convex sets, whereas straight line segments are convex but not affine sets.

Known convex sets are cones, hyperplanes, Euclidean balls, ellipsoids, polyhedra, etc.

2.2 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 every pair of points $\mathbf{x}$,$\mathbf{y} \in \mathbf{dom}f$ and for $0 \leqslant \theta \leqslant 1$, it holds true that

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

Geometrically, this means that the straight line segment joining $(\mathbf{x},f(\mathbf{x}))$ and $(\mathbf{y},f(\mathbf{y}))$ is never below that graph of function f, as seen in the following figure.

A function $f : \mathbf{dom}f \subseteq \mathbb{R}^n \to \mathbb{R}$ is **strictly convex** if $\mathbf{dom}f$ is convex and if, for every pair of points $\mathbf{x}$,$\mathbf{y} \in \mathbf{dom}f$ where $\mathbf{x} \neq \mathbf{y}$ and $0 < \theta < 1$, it holds true that

$$f(\theta\mathbf{x} + (1-\theta)\mathbf{y}) < \theta f(\mathbf{x}) + (1-\theta)f(\mathbf{y}).$$

Function f is **concave** if $-f$ is convex, and **strictly concave** if $-f$ is strictly convex.

An affine function is convex and concave simultaneously.

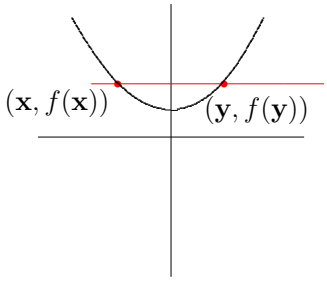


Figure 2.1: Convex function

2.3 Gradient and Hessian of function f

If $f : \mathbb{R}^n \to \mathbb{R}$ is differentiable, then function $\nabla f : \mathbb{R}^n \to \mathbb{R}$, 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 point $\mathbf{x}$. If ∇f is differentiable, then f is twice differentiable. The second derivative of f is the derivative of ∇f, is called **Hessian** of f and is defined as:

$$\nabla^2 f(\mathbf{x}) = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_2 \partial x_1} & \cdots & \frac{\partial^2 f}{\partial x_n \partial x_1} \\ \frac{\partial^2 f}{\partial x_1 \partial x_2} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_n \partial x_2} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_1 \partial x_n} & \frac{\partial^2 f}{\partial x_2 \partial x_n} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix}.$$

2.4 First and Second order Taylor approximations

Let $f : \mathbb{R}^n \to \mathbb{R}$ be twice differentiable. Then, the first and second order Taylor approximations are defined as:

$$\begin{aligned} f(\mathbf{y}) &= f(\mathbf{x}) + \nabla f(\mathbf{x})^T(\mathbf{y} - \mathbf{x}) + O(\|\mathbf{y} - \mathbf{x}\|^2) \\ &= f(\mathbf{x}) + \nabla f(\mathbf{x})^T(\mathbf{y} - \mathbf{x}) + \frac{1}{2}(\mathbf{y} - \mathbf{x})^T \nabla^2 f(\mathbf{x})(\mathbf{y} - \mathbf{x}) + O(\|\mathbf{y} - \mathbf{x}\|^3). \end{aligned}$$

It can be proved that

$$\begin{aligned} f(\mathbf{y}) &= f(\mathbf{x}) + \nabla f(\mathbf{z})^T(\mathbf{y}-\mathbf{x}) \\ &= f(\mathbf{x}) + \nabla f(\mathbf{x})^T(\mathbf{y}-\mathbf{x}) + \frac{1}{2}(\mathbf{y}-\mathbf{x})^T\nabla^2 f(\mathbf{w})(\mathbf{y}-\mathbf{x}). \end{aligned}$$

for $\mathbf{z}$ and $\mathbf{w}$ on the straight line segment which joins points $\mathbf{x}$ and $\mathbf{y}$.

2.4.1 First-order condition of convexity

Let open set $\mathbf{dom}f \subseteq \mathbb{R}^n$ and $f : \mathbf{dom}f \to \mathbb{R}$ differentiable function. Then, f is convex if and only if $\mathbf{dom}f$ is a convex set and

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

This inequality proves that the first-order Taylor approximation at any point of a convex function f is a global underestimator of f. Thus, from local information (value of function f and its gradient at some point), we get global information (a global underestimator of the function).

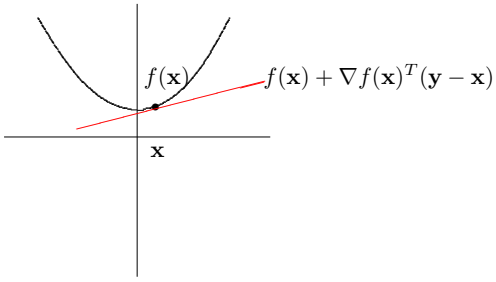


Figure 2.2: Convexity proven from first - order Taylor approximation

2.4.2 Second-order condition of convexity

Let open set $\mathbf{dom}f \subseteq \mathbb{R}^n$ and $f : \mathbf{dom}f \to \mathbb{R}$ twice differentiable. Then, f is convex if $\mathbf{dom}f$ is a convex set and

$$\nabla^2 f(\mathbf{x}) \succeq 0, \quad \forall \mathbf{x} \in \mathbf{dom}f.$$

Equivalently, f is strictly convex if $\mathbf{dom}f$ is a convex set and

$$\nabla^2 f(\mathbf{x}) \succ 0, \quad \forall \mathbf{x} \in \mathbf{dom}f.$$

Known convex functions are e^x, x^2, $|x|$, $\mathbf{a}^T\mathbf{x}+\mathbf{b}$, $||\mathbf{A}\mathbf{x}||^2$, $\frac{1}{2}\mathbf{x}^T\mathbf{P}\mathbf{x} + \mathbf{q}^T\mathbf{x}$, where $\mathbf{A}, \mathbf{a}, \mathbf{P}$ and $\mathbf{b}, \mathbf{q}$ are given matrices and vectors, respectively.

2.5 Convex Optimization Problems

2.5.1 Constrained optimization problems

A generic optimization problem is defined as

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

where vector $\mathbf{x} \in \mathbb{R}^n$ is called the **optimization variable**, function $f_0 : \mathbf{dom}f_0 \subseteq \mathbb{R}^n \to \mathbb{R}$ **cost function**, the inequalities $f_i(\mathbf{x}) \leq 0$ where $f_i : \mathbf{dom}f_i \subseteq \mathbb{R}^n \to \mathbb{R}$, for $i = 1, \ldots, m$, are called **inequality constraints**, and the equalities $h_i(\mathbf{x}) = 0$, where $h_i : \mathbf{dom}h_i \subseteq \mathbb{R}^n \to \mathbb{R}$, for $i = 1, \ldots, p$, are called **equality constraints**. These problems are called **constrained optimization problems**.

A point $\mathbf{x}$ is called **feasible** if it satisfies all constraints. A convex optimization problem is called feasible if at least one feasible point exists, otherwise it is called infeasible. The set of all feasible points is called a feasible set and is defined as

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

2.5.2 Unconstrained optimization problems

A problem defined as

$$\underset{\mathbf{x}}{\text{minimize}} \quad f(\mathbf{x}), \tag{2.1}$$

where $f : \mathbb{R}^n \to \mathbb{R}$ is a convex function, is called **unconstrained convex optimization problem**.

A point $\mathbf{x}_* \in \mathbb{R}^n$ is the solution to the problem if and only if

$$\nabla f(\mathbf{x}_*) = 0. \tag{2.2}$$

Point $\mathbf{x}_*$ is called **optimal point**. The minimum value of f is equal to $p_* := f(\mathbf{x}_*)$. p_* may take values $\pm\infty$. If the problem is infeasible, then $p_* = \infty$. If feasible points x_k exist, with $f(x_k) \to -\infty$ when $k \to \infty$, then $p_* = -\infty$ and the optimization problem is called unbounded from below.

2.6 Solving unconstrained optimization problems

Relation (2.2) is usually a system of non-linear equations, which rarely has a closed-form solution, and it is usually solved through direct or indirect iterative processes. An iterative process produces a sequence of points $\mathbf{x}_k \in \mathbb{R}^n$ such that $\mathbf{x}_k \to \mathbf{x}_*$, when $k \to \infty$.

A direct iterative process is trying to calculate $\mathbf{x}_*$ by solving system (2.2), whereas an indirect iterative process is using properties of function f to calculate $\mathbf{x}_*$ indirectly.

The methods we will use start from an initial point $\mathbf{x}_0 \in \mathbf{dom}f$, for which the set

$$\mathbb{S} = \{\mathbf{x} | f(\mathbf{x}) \leq f(\mathbf{x}_0)\}$$

is closed, i.e. it contains all its limit points. This is satisfied if $\mathbf{x}_0 \in \mathbf{dom}f$ and f is a closed function, that is, if for each $\alpha \in \mathbb{R}$, the sublevel set $\{\mathbf{x} \in \mathbf{dom}f | f(\mathbf{x}) \leq \alpha\}$ is a closed set.

Important cases of closed functions are continuous functions with closed $\mathbf{dom}f$, such that

$$f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T\mathbf{P}\mathbf{x} + \mathbf{q}^T\mathbf{x}$$

where $\mathbf{P} = \mathbf{P}^T \succ \mathbf{0}$, as well as continuous functions with open $\mathbf{dom}f$ if and only if $f(\mathbf{x})$ tends to ∞ when $\mathbf{x}$ tends to a boundary point of $\mathbf{dom}f$.

2.6.1 Descent methods

Descent methods are indirect iterative methods for solving (2.1) or (2.2). Assume that at the k-th iteration, our estimate for the solution of (2.2) is point $\mathbf{x}_k$. Then, the next point is given by

$$\mathbf{x}_{k+1} = \mathbf{x}_k + t_k \Delta \mathbf{x}_k.$$

We choose $t_k > 0$ and $\Delta \mathbf{x}_k$ such that

$$f(\mathbf{x}_{k+1}) \leq f(\mathbf{x}_k),$$

with equality if and only if $\mathbf{x}_k = \mathbf{x}_*$.

Since f is a convex function, we know that, if $\nabla f(\mathbf{x}_k)^T(\mathbf{y} - \mathbf{x}_k) \geq 0$, then $f(\mathbf{y}) \geq f(\mathbf{x}_k)$. So, in order to develop a descent method, $\Delta \mathbf{x}_k$ needs to be chosen so that

$$\nabla f(\mathbf{x}_k)^T \Delta \mathbf{x}_k < 0$$

and t_k so that

$$t_k = \underset{t>0}{\operatorname{argmin}} \quad f(\mathbf{x}_k + t\Delta \mathbf{x}_k).$$

2.6.2 Gradient Direction

The most popular choice for $\Delta \mathbf{x}_k$ is

$$\Delta \mathbf{x}_k = -\nabla f(\mathbf{x}_k),$$

which is called **negative gradient direction**. Geometrically, we are moving along the direction where the reduction rate of f on point $\mathbf{x}_k$ is maximum.

2.7 Convergence for strongly convex functions

Convergence analysis of optimization problem solving methods is crucial, since it provides a performance measure of the methods, the **convergence rate**, the speed at which a sequence approaches its limit. In this section, we will study the convergence of the gradient direction method, assuming the objective function is strongly convex.

Let $f : \mathbb{R}^n \to \mathbb{R}$ strongly convex twice differentiable function. We assume there exist $0 < m \leq M$, such that:

$$m\mathbf{I}_n \preceq \nabla^2 f(\mathbf{x}) \preceq M\mathbf{I}_n, \quad \forall \mathbf{x} \in \mathbb{S}, \tag{2.3}$$

where $\mathbb{S} := \{\mathbf{x} \in \mathbb{R}^n | f(\mathbf{x}) \leq f(\mathbf{x}_0)\}$.

Then, it can be shown that

$$f(\mathbf{x}) - p_* \leq \frac{1}{2m}||\nabla f(\mathbf{x})||_2^2 \tag{2.4}$$

for any $\mathbf{x} \in \mathbb{R}^n$. That is, if $||\nabla f(\mathbf{x})||_2^2$ is "small", then $f(\mathbf{x})$ is "close" to the optimal value p_*.

It can also be shown that, for any $\mathbf{x} \in \mathbb{R}^n$,

$$||\mathbf{x} - \mathbf{x}_*||_2 \leq \frac{2}{m}||\nabla f(\mathbf{x})||_2. \tag{2.5}$$

That is, if $||\nabla f(\mathbf{x})||_2$ is "small", then $\mathbf{x}$ is "close" to the optimal point $\mathbf{x}_*$. Thus, a terminating condition of the gradient direction algorithm may be the following

$$||\nabla f(\mathbf{x})||_2 < \epsilon$$

for "small" $\epsilon > 0$.

Chapter 3

Message Passing Interface

Message Passing Interface (MPI) is a communication protocol for programming parallel computers. It is a library of functions that can be called from C, C++, and Fortran programs, and is widely used in science applications.

3.1 Multiple Instruction - Multiple Data Systems

Multiple instruction - Multiple data (MIMD) systems consist of a collection of fully independent processing units or processors, each of which has its own control unit and its own ALU, and support multiple simultaneous instructions on multiple data. Such systems usually function **asynchronously**, which means that there is no global clock so, at any time, different processors may be executing different instructions on different pieces of data.

There are two types of MIMD systems: **shared-memory** and **distributed-memory** systems, but our focus will lie on the latter.

3.1.1 Distributed-Memory Systems

A distributed-memory system consists of processor-memory pairs connected by an interconnection network, where each processor can directly access only its own private memory, as shown in figure 3.2. Thus, the processors communicate with each other by sending and receiving messages, or by using special functions that provide access

to the memory of another processor.

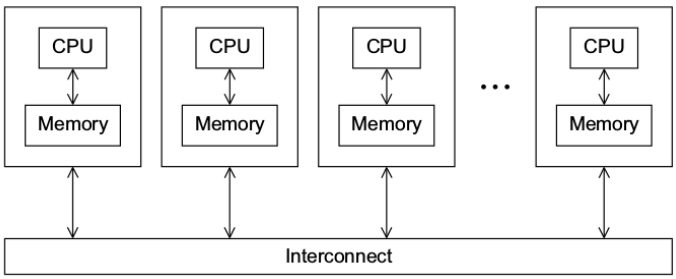


Figure 3.1: A distributed-memory system
Adapted from "Parallel Programming in MPI", by P. S. Pacheco, Morgan Kaufmann, 1997

3.2 Distributed-Memory APIs

Message passing is the most widely used application programming interface (API) for developing distributed-memory programs.

An important property of such APIs is that they can be used with shared-memory hardware. Hence, shared-memory is partitioned into private address spaces for the various processes, and a library or compiler implements the communication needed.

In message-passing programs, a program running at one processor-memory pair is called a **process**, and two processes can communicate by calling a *send* and a *receive* function. There also exist functions for various collective communications, such as a broadcast function, in which a single process transmits the same data to all the other processes.

The processes typically identify each other by ranks in the range $0, 1, ..., p-1$, where p is the number of processes.

The implementation of message-passing we will use is **Message Passing Interface (MPI)**.

3.3 SPMD programs

Despite the fact that each process almost always does something fundamentally different from the other processes, we compile a single program, not a different program for each process. In other words, a single program is written and different processes may perform different actions. Such programs are called **Single Program, Multiple Data** or **SPMD** programs.

Additionally, each MPI program should be able to run with any number of processes, because the resources available are not and will not always be the same.

3.4 Basic MPI functions

3.4.1 MPI_Init and MPI_Finalize

MPI_Init does all the necessary setup of the MPI system. It allocates storage for message buffers and defines the rank of each process. Before the call of MPI_Init, no other MPI function should be called. Its syntax is

```
int MPI_Init(
    int*        argc_p     /* in/out */
    char***     argv_p     /* in/out */);
```

The arguments *argc_p* and *argv_p* are pointers to the arguments of main (*argc* and *argv*). Hence, both arguments can be NULL.

MPI_Finalize frees any resources allocated for MPI. After the call of MPI_Finalize, no other MPI function should be called. Its syntax is

```
int MPI_Finalize(void);
```

3.4.2 Communicators

A **communicator** is a collection of processes that are allowed to send messages to each other. The default communicator in MPI is MPI_COMM_WORLD, is defined by MPI_Init and it includes every process the user declared at the beginning of the execution of the program.

The syntax of two of the most common functions on communicators is:

```
int MPI_Comm_size(
    MPI_Comm        comm      /* in */
    int*            comm_sz   /* out */);

int MPI_Comm_rank(
    MPI_Comm        comm      /* in */
    int*            my_rank   /* out */);
```

The first argument of both functions is the communicator, which in our program is the default communicator, MPI_COMM_WORLD. *comm_sz* indicates the number of processes of the communicator, and *my_rank* the rank of the calling process in the communicator.

3.4.3 MPI_Send and MPI_Recv

MPI_Send

Each send is carried out by a call to MPI_Send, whose syntax is:

```
int MPI_Send(
    void*           msg_buf_p     /* in */
    int             msg_size      /* in */)
    MPI_Datatype    msg_type      /* in */)
    int             dest          /* in */)
    int             tag           /* in */)
    MPI_Comm        comm          /* in */);
```

The first three arguments determine the contents of the message: *msg_buf_p* is a pointer to the block of memory containing the message to be sent, *msg_size* holds the amount of data in the message, and *msg_type* determines the type of the message to be sent. The size of *msg_buf_p* must be less than or equal to *msg_size*. Some MPI_Datatypes are: *MPI_CHAR, MPI_SHORT, MPI_INT, MPI_FLOAT*, etc.

The last three arguments determine the destination of the message: *dest* holds the rank of the process that will receive the message, *tag* is the message ID, and *communicator* specifies that the message will be received by a process belonging to the same communicator with the process sending the message. *tag* is used so that, if the receiving process requests messages with a certain tag, messsages with different tags will be buffered by the network until the process requests them.

MPI_Recv

A call to MPI_Recv has syntax:

```
int MPI_Recv(
    void*          msg_buf_p      /* out */
    int            buf_size       /* in */)
    MPI_Datatype   buf_type       /* in */)
    int            source         /* in */)
    int            tag            /* in */)
    MPI_Comm       communicator   /* in */)
    MPI_Status*    status_p       /* out */);
```

The first six arguments correspond to the arguments of MPI_Send, and specify the memory available for receiving the message: *msg_bug_p* containing the message to be received, which is of size *buf_size* and of type *buf_type*, *source* specifies the rank of the process sending the message, *tag* holds the message ID and *communicator* must match the communicator used by the sending process.

The final argument, *status_p*, contains a pointer to an MPI_Status structure where information about the received message is stored. In many cases it will not be used by the calling function, so MPI_STATUS_IGNORED can be passed.

3.5 I/O in MPI

3.5.1 Output

MPI allows all the processes in the default communicator MPI_COMM_WORLD full access to *stdout*, so most MPI implementations allow all the processes to print messages.

However, most MPI implementations do not provide any automatic scheduling of access to output devices. Thus, if multiple processes are trying to write to *stdout*, the order in which the output of the processes appears on the screen will be unpredictable. It can happen that the output of one process will be interrupted by the output of another process.

The reason this happens is that the processes are "competing" for access to the shared output device and it is not possible to predict the order in which the processes' output will be queued up. Thus, the output can vary across runs.

3.5.2 Input

Unlike output, most MPI implementations allow only process 0 in the default communicator MPI_COMM_WORLD to access *stdin*. This happens because if multiple processes have access to *stdin*, the program would not know which process would get which part of the input data, for example, if the data should be divided by lines or characters.

Thus, in order to write MPI programs that can use *scanf*, we will have process 0 to read the data and then send the data to the processes that will need it.

3.6 Collective Communication

Collective communication is a communication of data that involves more than two processes. A collective operation is executed by having all processes in the same communicator call the communication routine with matching arguments. There are many collective operations with different functionalities, but our focus will lie on the ones used in our applications.

3.6.1 MPI_Bcast

MPI_Bcast is a collective operation in which data belonging to a single process is sent to all the processes in the communicator. Its syntax is:

```
int MPI_Bcast(
    void*          data_p        /* in/out */
    int            count         /* in */)
    MPI_Datatype   datatype      /* in */)
    int            source_proc   /* in */)
    MPI_Comm       comm          /* in */);
```

The first argument, *data_p*, is a pointer to the block of memory containing the message to be sent by process *source_proc* to all processes in the communicator *comm*. The arguments *count* and *datatype* indicate the number of characters and the type of the message respectively.

3.6.2 MPI_Scatter

MPI_Scatter reads a vector that is on a single process and sends only the needed components to each of the other processes in the communicator. Its syntax is:

```
int MPI_Scatter(
    void*          send_buf_p    /* in */
    int            send_count    /* in */)
    MPI_Datatype   send_type     /* in */)
    void*          recv_buf_p    /* out */)
    int            recv_count    /* in */)
    MPI_Datatype   recv_type     /* in */)
    int            src_proc      /* in */)
    MPI_Comm       comm          /* in */);
```

Suppose that communicator *comm* contains comm_sz processes. Then, MPI_Scatter divides the data referenced by *send_buf_p* into comm_sz pieces and sends each piece to the corresponding process. *send_count* holds the amount of data going to each process and *recv_count* the amount of data received by each process, so these two arguments have the same value. *recv_buf_p* stores the data that each process will receive, while *src_proc* indicates the process sending the data, that is process 0.

3.6.3 MPI_Allgather

MPI_Allgather gathers data from all the processes in the communicator, and broadcasts the combined data to all the processes. Its syntax is:

```
int MPI_Allgather(
    void*          send_buf_p    /* in */
    int            send_count    /* in */)
    MPI_Datatype   send_type     /* in */)
    void*          recv_buf_p    /* out */)
    int            recv_count    /* in */)
    MPI_Datatype   recv_type     /* in */)
    MPI_Comm       comm          /* in */);
```

The function concatenates the contents of each process' *send_buf_p* and the concatenated data are stored in each process' *recv_buf_p*. *send_count* holds the amount of data to be sent from each process, and *recv_count* holds the amount of data received from each process, so usually these two arguments are the same.

3.7 Taking timings

In order to test the performance of an MPI program, MPI provides a function which returns the number of seconds that have elapsed since some time in the past. Its syntax is:

```
double MPI_Wtime(void);
```

Usually, we are not interested in the time taken from the beginning to the end of the program execution, but only in the time it takes to execute a specific part of the program. Thus, we can time a block of code as follows:

```
double start_time, end_time;
...
start_time = MPI_Wtime();
...   /* Code to time */
end_time = MPI_Wtime();
printf("Process %d, elapsed time = %e seconds", my_rank, end_time -
start_time);
```

Chapter 4

Optimization Algorithms

Quadratic programming is the problem of optimizing a quadratic objective function, and is widely used in finance, statistics, signal and image processing, computer systems and chemical production. In this chapter, we will describe two methods we implemented to solve a quadratic problem without constraints, the **Gradient Descent** method and the **Block Coordinate Descent** method.

4.1 The quadratic problem

An unconstrained quadratic optimization problem with n variables is formulated as follows. Given:

- an $(n \times n)$ symmetric matrix $\mathbf{P}$ and
- an n-dimensional vector $\mathbf{q}$

the objective of quadratic programming is to find an n-dimensional vector $\mathbf{x}$ which will be the solution to the problem:

$$\underset{\mathbf{x}\in\mathbb{R}^n}{\text{minimize}} \quad f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T\mathbf{P}\mathbf{x} + \mathbf{q}^T\mathbf{x}$$

where $f : \mathbb{R}^n \to \mathbb{R}$ is the objective function.

Any constants contained in the objective function are left out of the general formulation of the quadratic problem. The fraction $\frac{1}{2}$ in front of the quadratic term is added to remove the coefficient of 2 that occurs when taking the derivative of a

second order polynomial.

For matrix $\mathbf{P}$ applies:

$$\mathbf{P} = \mathbf{P}^T \succ 0.$$

The gradient and the Hessian of the quadratic function discussed above are defined as follows:

$$\nabla f(\mathbf{x}) = \mathbf{P}\mathbf{x} + \mathbf{q}, \tag{4.1}$$

$$\nabla^2 f(\mathbf{x}) = \mathbf{P}. \tag{4.2}$$

From the second-order condition of convexity discussed in paragraph 2.4.2, to analyze a function's convexity or strict convexity, one can compute its Hessian and verify that it is non-negative definite or positive definite, respectively. In the quadratic problem, the Hessian is equal to matrix $\mathbf{P}$, and since $\mathbf{P} \succ 0$, our function is **strictly convex**.

We note that the quadratic optimization problem of interest has a closed-form solution. In the sequel, we use iterative algorithms for its solution in order to study their behaviour in multi-processing environments.

4.2 Setup

To begin with, all matrix/vector data were created in *Matlab* and stored in .txt files, whereas the programs implementing both Gradient Descend method and Block Coordinate Descent method were written in C++ and use the MPI library. For matrix/vector operations we used routines of **Eigen** library, a C++ template library for linear algebra.

4.3 Controling the condition number of f

The **condition number** of a function with respect to an argument measures how sensitive the output of the function is with respect to changes of the input.

Recall that for $f : \mathbb{R}^n \to \mathbb{R}$ strongly convex twice differentiable function, we have assumed there exist $0 < m \leq M$ such that

$$m\mathbf{I}_n \preceq \nabla^2 f(\mathbf{x}) \preceq M\mathbf{I}_n, \quad \forall \mathbf{x} \in \mathbb{S}$$

where $\mathbb{S} := \{\mathbf{x} \in \mathbb{R}^n | f(\mathbf{x}) \leq f(\mathbf{x}_0)\}$.

Then, the condition number of function f is defined as:

$$K := \frac{M}{m}.$$

In order to control the condition number of our function, we express matrix $\mathbf{P}$ in ***Matlab*** as

$$\mathbf{P} = \mathbf{U}\mathbf{\Lambda}\mathbf{U}^T.$$

Matrix $\mathbf{U}$ is an $(n \times n)$ orthonormal matrix, which means that its rows and columns are orthogonal unit vectors, i.e. $\mathbf{U}\mathbf{U}^T = \mathbf{U}^T\mathbf{U} = \mathbf{I}_n$. It is constructed by calculating the singular-value decomposition of a random $(n \times n)$ matrix $\mathbf{A}$, using Matlab's *svd* function, as:

$$[\mathbf{U}, \mathbf{S}, \mathbf{V}] = svd(\mathbf{A});$$

and its columns are the eigenvectors of matrix $\mathbf{P}$.

Matrix $\mathbf{\Lambda}$ is an $(n \times n)$ diagonal matrix, which holds in the main diagonal the eigenvalues λ_i of matrix $\mathbf{P}$, for $i = 1, \ldots, n$. It is constructed by selecting the smallest and largest eigenvalues, λ_{min} and λ_{max} respectively, creating $(n-2)$ random uniformly distributed numbers in the interval $[\lambda_{min}, \lambda_{max}]$, and placing these values in the main diagonal of matrix $\mathbf{\Lambda}$.

The condition number of our function is:

$$K := \frac{\lambda_{max}}{\lambda_{min}}.$$

and we have:

$$\lambda_{min}\mathbf{I}_n \preceq \nabla^2 f(\mathbf{x}) \preceq \lambda_{max}\mathbf{I}_n, \quad \forall \mathbf{x} \in \mathbb{S}.$$

Vector $\mathbf{q} \in \mathbb{R}^n$ is a vector of independent normally distributed random numbers. Both $\mathbf{P}$ and $\mathbf{q}$ are stored in .txt files, to be used later in our program.

Note: The singular value decomposition of an $(n \times n)$ *matrix* $\boldsymbol{A}$ *is a factorization of* $\boldsymbol{A}$ *into a product of matrices of the form* $\boldsymbol{USV}^T$*, where:*

- U *is an* $(n \times n)$ *orthonormal matrix whose columns hold the eigenvectors of* $\boldsymbol{AA}^T$*;*
- V *is an* $(n \times n)$ *orthonormal matrix whose columns hold the eigenvectors of* $\boldsymbol{A}^T\boldsymbol{A}$*;*
- S *is an* $(n \times n)$ *diagonal matrix whose diagonal elements hold the square roots of the eigenvalues of both* $\boldsymbol{AA}^T$ *and* $\boldsymbol{A}^T\boldsymbol{A}$.

4.4 Gradient Descent method

Gradient Descent is an iterative optimization algorithm for finding the minimum of a function. It is based on the observation that if the objective function $f(\cdot)$ is defined and differentiable in a neighborhood of a point $\mathbf{x}$, then $f(\cdot)$ descreases fastest if we move along the direction of $-\nabla f(\mathbf{x})$. If we move along the direction of the *positive* of the gradient at the current point, then the local maximum of the function would be found, and the method would be called *gradient ascent*.

Assuming that in the k-th step of the iterative process our estimate for the solution of the problem is $\mathbf{x}_k$, it follows that the next point is given by

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \lambda_k \nabla f(\mathbf{x}_k)$$

for a step-size λ_k for which

$$f(\mathbf{x}_{k+1}) \leq f(\mathbf{x}_k).$$

Based on these observations, the method begins from an initial point $\mathbf{x}_0$ and constructs the sequence $\mathbf{x}_0$, $\mathbf{x}_1$, $\mathbf{x}_2$, ..., such that

$$f(\mathbf{x}_0) \geq f(\mathbf{x}_1) \geq f(\mathbf{x}_2) \geq \dots$$

so, at each step, function f is dcreased until the sequence $\{\mathbf{x}_k\}$ converges to a local minimum. If $f(\mathbf{x})$ is (strongly) convex, any local minimum is also a global minimum. Thus, the gradient descent converges to a global minimum.

The global minimum of $f(\mathbf{x})$ is called optimal point, is defined as $\mathbf{x}_*$, and for it holds that

$$\nabla f(\mathbf{x}_*) = 0.$$

The gradient descent method is illustrated in figure (4.1) below. It is assumed that f is defined on a plane. The blue curves are the level sets of function f. The red arrows show the direction of the negative gradient from one point to the other, while trying to reach the optimal point.

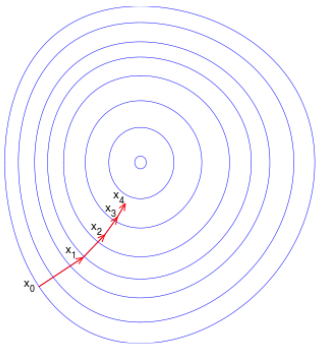


Figure 4.1: Gradient Descent Method on a series of level sets
Adapted from https://en.wikipedia.org/wiki/Gradient_descent

4.4.1 Step size λ

Note that the value of step size λ is allowed to change at every iteration. A careful selection of λ is crucial. We can either choose a fixed step size which will assure convergence or choose a different step size at each iteration. In our program, we chose a fixed step size. It can be shown that

$$\lambda = \frac{1}{\lambda_{\max}}$$

is a good choice for the step size, and this is our choice in our experiments.

4.4.2 Terminating condition

In section 2.7 of Chapter 2, we mentioned that

$$||\mathbf{x} - \mathbf{x}_*||_2 \leq \frac{2}{\lambda_{\min}} ||\nabla f(\mathbf{x})||_2.$$

Thus, if $||\nabla f(\mathbf{x})||_2$ is "small", then this $\mathbf{x}$ is "close" to optimal point $\mathbf{x}_*$. Based on this fact, in our programs we use the terminating condition

$$||\nabla f(\mathbf{x}_k)||_2 < \epsilon$$

where ϵ is "small" (for example, $\epsilon = 10^{-4}$).

4.5 Block Coordinate Descent method

Coordinate Descent is an iterative algorithm which minimizes the objective function along coordinate directions. Thus, at each iteration the function is minimized along one coordinate or coordinate block, while keeping the other coordinates or coordinate blocks fixed.

The method begins from an initial point

$$\mathbf{x}^0 = (\mathbf{x}_0^0, \ldots, \mathbf{x}_{n-1}^0),$$

where n is the problem size, and cyclically moves along one coordinate or coordinate block at a time, minimizes on it and moves to the next. Assuming that, in the k-th iteration, our estimate for the solution of the problem is $\mathbf{x}^k$, we calculate the i-th coordinate of iteration $(k+1)$, $\mathbf{x}_i^{k+1}$, by using the Jacobi or the Gauss-Seidel coordinate descent algorithms.

The **Jacobi** coordinate descent algorithm is defined as:

$$\mathbf{x}_i^{k+1} = \underset{\mathbf{x}_i \in X_i}{\operatorname{argmin}} \quad f(\mathbf{x}_0^k, \ldots, \mathbf{x}_{i-1}^k, \mathbf{x}_i, \mathbf{x}_{i+1}^k, \ldots, \mathbf{x}_{n-1}^k)$$

and the **Gauss - Seidel** coordinate descent algorithm as:

$$\mathbf{x}_i^{k+1} = \underset{\mathbf{x}_i \in X_i}{\operatorname{argmin}} \quad f(\mathbf{x}_0^{k+1}, \ldots, \mathbf{x}_{i-1}^{k+1}, \mathbf{x}_i, \mathbf{x}_{i+1}^k, \ldots, \mathbf{x}_{n-1}^k).$$

The Jacobi algorithm can be parallelized by assigning each block of coordinates $\mathbf{x}_i$ to a different processor, whereas the Gauss-Seidel algorithm can be parallelized if a coloring scheme is applied.

Block Coordinate Descent (BCD) is based on the same logic, with the only difference being that the coordinates are partitioned into blocks. Suppose we break the n coordinates in b blocks, where each block holds $\frac{n}{b}$ coordinates. That way, at each iteration, we compute a block of coordinates.

Each coordinate or coordinate block is minimized exactly, by solving the problem with closed form, or inexactly, by minimizing it along the gradient direction.

In this thesis, we will implement a Jacobi Block Coordinate Descent algorithm to solve the quadratic problem discussed above.

Note that for both methods, we have:

$$f(\mathbf{x}^0) \geq f(\mathbf{x}^1) \geq f(\mathbf{x}^2) \geq \ldots \tag{4.3}$$

4.5.1 Terminating condition

As the terminating condition for our algorithm we used

$$||\mathbf{x}^k - \mathbf{x}^*||_2 < \epsilon,$$

where ϵ is "small", $\epsilon = 10^{-4}$ specifically.

This inequality means that, if the distance between the point calculated from iteration k and the optimal point is "small", then we can say that the optimal point is reached and the iterative method should end.

Chapter 5

Distributed Asynchronous Iterative Algorithms

We will now consider distributed asynchronous implementations of the algorithms discussed in the preceding chapter. An iterative algorithm is parallelized by separating it into several local algorithms operating concurrently at different processors.

5.1 Parallelizing the data

At the beginning of the program execution, the number of processes are set from the user, and no more processes can be added before the program finishes execution. Each process is assigned a unique integer called *rank*, from 0 to $np - 1$, with np denoting the total number of processes in the communicator.

Given a network of np processes, we want to solve

$$\underset{\mathbf{x}\in\mathbb{R}^n}{\text{minimize}} \quad f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T\mathbf{P}\mathbf{x} + \mathbf{q}^T\mathbf{x}$$

We will solve the problem with row partition. We suppose that np evenly divides n, the number of rows of $\mathbf{P}$ and $\mathbf{q}$, so there will be np block-rows, one for each process in the communicator. The row partition is shown in the figure below:

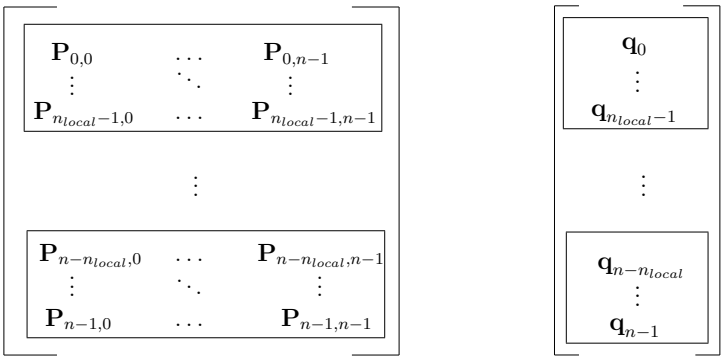


Figure 5.1: Row partition of matrix $\mathbf{P}$ and vector $\mathbf{q}$

So, in each process, a row - block of $\mathbf{P}$ and $\mathbf{q}$ will be stored, which are $\mathbf{P}_{local}$ and $\mathbf{q}_{local}$ respectively. The number of rows of each block is equal to

$$\mathbf{n}_{local} = \frac{n}{np}.$$

So, $\mathbf{P}_{local}$ will be of size $(n_{local} \times n)$, and $\mathbf{q}_{local}$ will be of size n_{local}.

To be more specific, process 0 will have rows $0, \ldots, n_{local} - 1$, process 1 will have rows $n_{local}, \ldots, 2n_{local} - 1$, and so on.

As mentioned in section 3.6.2, MPI function *MPI_Scatter* reads in an entire vector that is on process 0 and partitions its data evenly between the processes. We have:

MPI_Scatter($\mathbf{P}$, $n_{local} \times n$, MPI_DOUBLE, $\mathbf{P}_{local}$,
$n_{local} \times n$, MPI_DOUBLE, 0, MPI_COMM_WORLD);

MPI_Scatter($\mathbf{q}$, n_{local}, MPI_DOUBLE, $\mathbf{q}_{local}$,
n_{local}, MPI_DOUBLE, 0, MPI_COMM_WORLD);

MPI_Scatter divides $\mathbf{P}$ and $\mathbf{q}$ into blocks of size $(n_{local}) \times n$ and n_{local} respectively, and sends each blocks to its corresponding process. In fact, process 0 will get the first block, process 1 will get the second block, and so on.

The second and fifth argument hold the amount of data going to each process and the amount of data received by each process respectively, so they are equal.

The seventh argument denotes the source process so, since $\mathbf{P}$ and $\mathbf{q}$ are read from the .txt files in process 0, then process 0 is the source process.

The final argument denotes the communicator in which the receiving processes belong to, which is equal to the default communicator.

Then, a call to MPI_Bcast is made, to send step t to all the processes in the communicator, as follows:

MPI_Bcast(t, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);

Specifically, process 0, which is the source process since it calculates step t, sends it into all the processes belonging to the default communicator. t is of size 1 and of type MPI_DOUBLE, as observed from the second and third argument respectively.

5.2 Gradient method in MPI

After the data are divided among the processes, we set initial point

$$\mathbf{x}_0 = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

where $\mathbf{x}_0 \in \mathbb{R}^n$ and then we begin the iterative method.

At each step of the iterative process, each process computes a block of the next point as follows. Suppose that in the k-th step of the process, each process' estimate for the solution of the problem is $\mathbf{x}^k_{local}$, then the next local point is given by:

$$\mathbf{x}^{k+1}_{local} = \mathbf{x}^k_{local} + \lambda \Delta \mathbf{x}^k.$$

$\Delta \mathbf{x}^k$ moves along the direction of the negative of the local gradient, i.e. the block of gradient for each process, as:

$$\Delta \mathbf{x}^k = -\nabla_{local} f(\mathbf{x}^k) = -(\mathbf{P}_{local}\mathbf{x} + \mathbf{q}_{local}).$$

Algorithm 5.1 Parallel Gradient Descent Algorithm

Set $x_0 \in \mathbb{R}^n$, $k = 0$
start_time = MPI_Wtime();
while (terminating condition is FALSE) **do**
  $\nabla_{local} f(\mathbf{x}^k) = \mathbf{P}_{local}\mathbf{x}^k + \mathbf{q}_{local}$
  $\Delta\mathbf{x}_k = -\nabla_{local} f(\mathbf{x}^k)$
  $\mathbf{x}_{local}^{k+1} = \mathbf{x}_{local}^k + \lambda\Delta\mathbf{x}_k$
  MPI_Allgather($\nabla_{local} f(\mathbf{x}^k)$)
  MPI_Allgather($\mathbf{x}_{local}^{k+1}$)
  k := k + 1
end while
end_time = MPI_Wtime();

Then, we need to concatenate and broadcast to all processes the blocks of $\mathbf{x}^{k+1}$ and $\nabla_{local} f(\mathbf{x}^k)$ of each process, because the first is needed in the next computation of $\Delta\mathbf{x}_{local}$ and the second is needed for the terminating condition. To do so, MPI_Allgather was used. We have:

MPI_Allgather($gradient_{local}$, local_n, MPI_DOUBLE,
gradient, local_n, MPI_DOUBLE, MPI_COMM_WORLD);

MPI_Allgather($\mathbf{x}_{local}$, local_n, MPI_DOUBLE,
$\mathbf{x}$, local_n, MPI_DOUBLE, MPI_COMM_WORLD);

MPI_Allgather concatenates vectors $gradient_{local}$ and $\mathbf{x}_{local}$ to vectors $gradient$ and $\mathbf{x}$ respectively, and broadcasts them to all the processes belonging to the default communicator MPI_COMM_WORLD.

The second and fifth argument of MPI_Allgather denote the number of elements to be gathered from each process and the number of elements to be received by each process, so they are equal.

A pseudocode of the parallel program we implemented is shown in Table 5.1.

5.3 Block Coordinate Descent in MPI

5.3.1 Dividing the data among the processes

The problem will be solved with closed-form solution. When we refer to closed-form solution, we mean solving the problem

$$\nabla f(\mathbf{x}) = 0 \Rightarrow \mathbf{P}\mathbf{x} + \mathbf{q} = 0 \Rightarrow \mathbf{x} = -\mathbf{P}^{-1}\mathbf{q}.$$

Since each process has access to a block-row of $\mathbf{P}$ and $\mathbf{q}$, each process is called to solve a "smaller" local problem with closed-form solution, i.e. a problem of the form:

$$\nabla_{local} f(\mathbf{x}) = 0 \Rightarrow \mathbf{A}\mathbf{x}_{local} + \mathbf{b} = 0 \Rightarrow \mathbf{x}_{local} = -\mathbf{A}^{-1}\mathbf{b}, \tag{5.1}$$

where $\mathbf{A}$ is an $(n_{local} \times n_{local})$ matrix and $\mathbf{b}$ is an $(n_{local} \times 1)$ vector.

In order to construct $\mathbf{A}$ and $\mathbf{b}$, we partition matrix $\mathbf{P}$ in np block-rows and np block-columns and vector $\mathbf{q}$ in np blocks. For the objective function we have:

$$f(\mathbf{x}) = \frac{1}{2}\begin{bmatrix}\mathbf{x}_1^T & \dots & \mathbf{x}_i^T & \dots & \mathbf{x}_{np}^T\end{bmatrix}\begin{bmatrix}\mathbf{P}_{1,1} & \dots & \mathbf{P}_{1,i} & \dots & \mathbf{P}_{1,np} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ \mathbf{P}_{i,1} & \dots & \mathbf{P}_{i,i} & \dots & \mathbf{P}_{i,np} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ \mathbf{P}_{np,1} & \dots & \mathbf{P}_{np,i} & \dots & \mathbf{P}_{np,np}\end{bmatrix}\begin{bmatrix}\mathbf{x}_1 \\ \vdots \\ \mathbf{x}_i \\ \vdots \\ \mathbf{x}_{np}\end{bmatrix} +$$

$$+\begin{bmatrix}\mathbf{q}_1^T & \dots & \mathbf{q}_i^T & \dots & \mathbf{q}_{np}^T\end{bmatrix}\begin{bmatrix}\mathbf{x}_1 \\ \vdots \\ \mathbf{x}_i \\ \vdots \\ \mathbf{x}_{np}\end{bmatrix},$$

where $\mathbf{x}_i$ is the i-th block of $\mathbf{x}$ and is of size $(1 \times n_{local})$, $\mathbf{x}_i^T$ denotes the transpose of $\mathbf{x}_i$ and is of size $(n_{local} \times 1)$, $\mathbf{P}_{i,i}$ denotes a block of $\mathbf{P}$ of size $(n_{local} \times n_{local})$ and $\mathbf{q}_i^T$ is the i-th block of the transpose of $\mathbf{q}$ and is of size $(n_{local} \times 1)$.

By linear algebra operations in the above equation, we get:

$$f(\mathbf{x}) = \frac{1}{2}\mathbf{x}_1\mathbf{P}_{1,1}\mathbf{x}_1^T + \cdots + \frac{1}{2}\mathbf{x}_1\mathbf{P}_{i,1}\mathbf{x}_i^T + \cdots + \frac{1}{2}\mathbf{x}_1\mathbf{P}_{np,1}\mathbf{x}_{np}^T$$

$$+ \cdots$$

$$+\frac{1}{2}\mathbf{x}_i\mathbf{P}_{1,i}\mathbf{x}_1^T + \cdots + \frac{1}{2}\mathbf{x}_i\mathbf{P}_{i,i}\mathbf{x}_i^T + \cdots + \frac{1}{2}\mathbf{x}_i\mathbf{P}_{np,i}\mathbf{x}_{np}^T$$

$$+ \cdots$$

$$+\frac{1}{2}\mathbf{x}_{np}\mathbf{P}_{1,np}\mathbf{x}_1^T + \cdots + \frac{1}{2}\mathbf{x}_{np}\mathbf{P}_{i,np}\mathbf{x}_i^T + \cdots + \frac{1}{2}\mathbf{x}_{np}\mathbf{P}_{np,np}\mathbf{x}_{np}^T$$

$$+\mathbf{x}_1\mathbf{q}_1^T + \cdots + \mathbf{x}_i\mathbf{q}_i^T + \cdots + \mathbf{x}_{np}\mathbf{q}_{np}^T.$$

For the i-th block, we define the coefficient of the second order term as matrix $\mathbf{A}$ and the coefficient of the first order term as vector $\mathbf{b}$. Thus:

$$\mathbf{A} = \mathbf{P}_{i,i}, \tag{5.2}$$

$$\mathbf{b} = \mathbf{P}_{i,1}\mathbf{x}_1 + \cdots + \mathbf{P}_{i,np}\mathbf{x}_{np} + \mathbf{x}_i\mathbf{q}_i^T. \tag{5.3}$$

5.3.2 Conditions of convergence

Unfortunately, this method does not always converge. A sufficient but not necessary condition of convergence for the method is if matrix $\mathbf{P}$ is diagonally dominant. However, the method can sometimes converge even if this condition is not satisfied.

In order to solve this problem, at iteration $(k+1)$, we compare the value f_{local}^{k+1} with the value f_{local}^{k}, i.e. the values of f of iterations $k+1$ and k, as computed by each process. From theory (equation 4.3), we must have:

$$f_{local}^{k+1} \leq f_{local}^{k} \leq \dots$$

However, if this does not happen at iteration $(k+1)$ for some process, then this process does not update its part of $\mathbf{x}$ at iteration $(k+1)$, but it keeps the one it calculated at iteration k. This way, we guarantee that the condition 4.3 is satisfied, and that our algorithm always converges.

The algorithm begins from initial point

$$\mathbf{x}_0 = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

and at each iteration, each process computes its $\mathbf{A}$ and $\mathbf{b}$ using the equations 5.2 and 5.3, and then solves the problem:

$$\mathbf{x}_{local}^{k+1} = -\mathbf{A}_{local}^{-1}\mathbf{b}_{local}^{k}$$

where $\mathbf{x}_{local}^{k+1}$ denotes the block of $\mathbf{x}$ for iteration $(k+1)$ which each process calculates, $\mathbf{A}_{local}^{-1}$ denotes matrix $\mathbf{A}$ of each process, and $\mathbf{b}_{local}^{k}$ denotes vector $\mathbf{b}$ of each process as calculated using vector $\mathbf{x}$ of iteration k. As one can easily observe, each process' $\mathbf{A}$ remains the same at each iteration, but $\mathbf{b}$ changes since it depends on vector $\mathbf{x}$, which changes at each iteration.

Then, in order to ensure that our method will always converge, we apply the check we introduced above, i.e.:

$$\text{if } (f(\mathbf{x}_{local}^{k+1}) > f(\mathbf{x}_{local}^{k}))$$
$$\mathbf{x}_{local}^{k+1} = \mathbf{x}_{local}^{k}$$

This way, we ensure that the condition $f_{local}^{k+1} \leq f_{local}^{k}$ is satisfied.

Algorithm 5.2 Parallel Block Coordinate Descent algorithm

Set $x_0 \in \mathbb{R}^n$, $k = 0$
start_time = MPI_Wtime();
Calculate **A**
while (terminating condition is FALSE) **do**
  Calculate **b**
  $\mathbf{x}_{local}^{k+1} = -\mathbf{A}^{-1} * \mathbf{b}$
  if $(f(\mathbf{x}_{local}^{k+1}) > f(\mathbf{x}_{local}^{k}))$
    $\mathbf{x}_{local}^{k+1} = \mathbf{x}_{local}^{k}$
  MPI_Allgather($\mathbf{x}_{local}^{k+1}$)
  k := k + 1
end while
end_time = MPI_Wtime();

After each process calculates the new point $\mathbf{x}_{local}^{k+1}$, MPI function MPI_Allgather is used, as shown below:

MPI_Allgather($\mathbf{x}_{local}$, local_n, MPI_DOUBLE,
$\mathbf{x}$, local_n, MPI_DOUBLE, MPI_COMM_WORLD);

to concatenate and broadcast the new point to all the processes in the communicator.

A pseudocode of the parallel program we implemented for Block Coordinate Descent is shown in Table 5.2

Chapter 6

Experimental Results

In this chapter, we present results obtained from the MPI programs of Gradient Method and Block Coordinate Descent method.

6.1 Set up

We note that all figures below were created in ***Matlab***, by importing data from files created by the MPI algorithms. For all the figures below, we used *Matlab*'s function **semilogy**, which plots data in the logarithmic scale for the y-axis.

6.2 Convergence of MPI algorithms

In order to confirm the convergence of our algorithms, we examined our implementations for multiple problem sizes n and number of processes np. More specifically, the set for n is $\{500, 1000, 1600\}$ and np is taken from the set $\{2, 4, 10\}$.

We note that the condition number K for the problems presented in this section is equal to 10. More about the condition number on section 6.3.

6.2.1 Convergence of Gradient Descent

The left figures below present the distance of f at each iteration k from p_*, the minimum value of f. The right figures present the distance of $\mathbf{x}$ at each iteration k from $\mathbf{x}_*$, the optimal point reached.

We observe that f is minimizing and the optimal point $\mathbf{x}_*$ is reached since all figures tend to 0, so our method converges irrespective of the problem size and the number of processes.

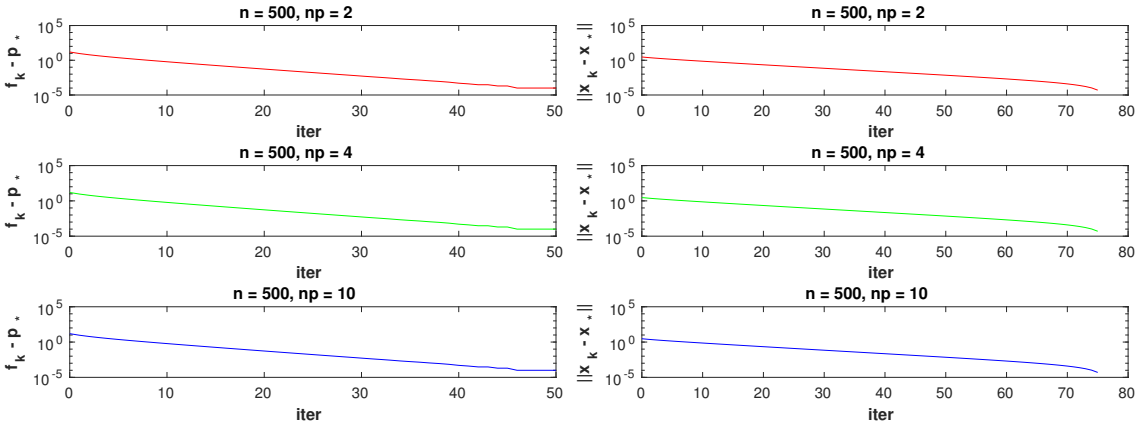


Figure 6.1: Gradient Descent method with $n = 500$ and $K = 10$

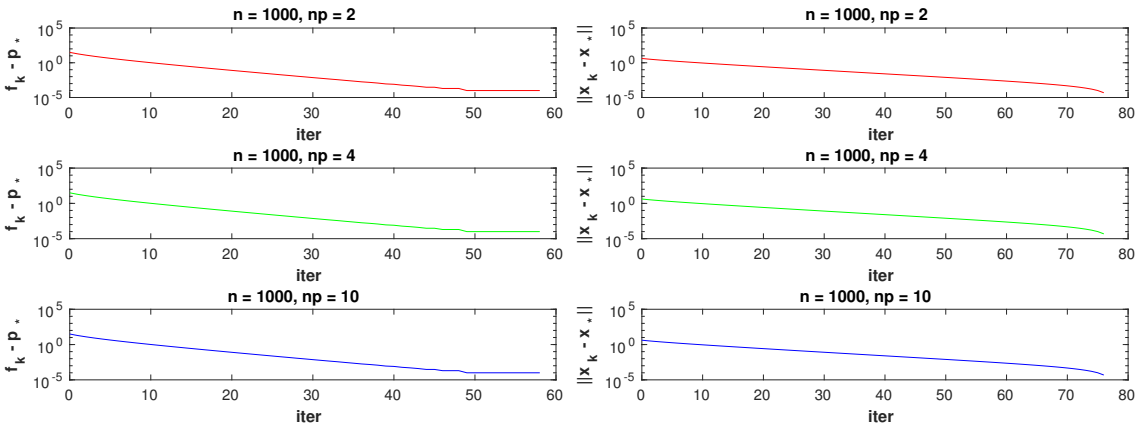


Figure 6.2: Gradient Descent method with $n = 1000$ and $K = 10$

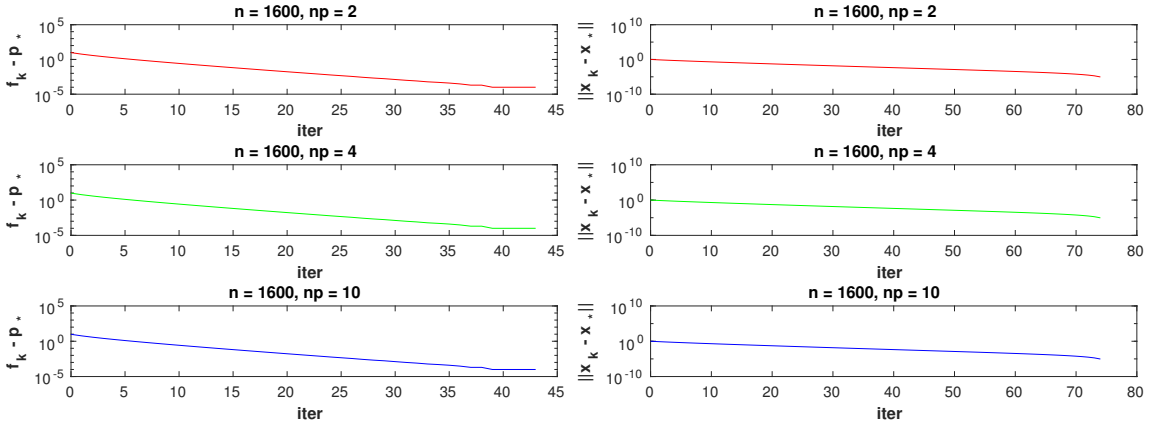


Figure 6.3: Gradient Descent method with $n = 1600$ and $K = 10$

The number of iterations and the run time for the above problems are shown in the table below:

Number of iterations			
Problem size n	$np = 2$	$np = 4$	$n = 10$
500	77	77	77
1000	78	78	78
1600	76	76	76

Run Time			
Problem size n	$np = 2$	$np = 4$	$n = 10$
500	0.33	0.46	5.58
1000	1.3	2.44	9.05
1600	3.45	5.18	14.88

From the multiple times we run the algorithm, but also from the results presented above, we observe that the problem size does not interfere much on the number of iterations, but only on the run time of the algorithm.

We can also observe that, as the number of processes increase, the number of iterations until the optimal point is reached remains the same. However, the time it takes for the algorithm to converge increases.

6.2.2 Convergence of Block Coordinate Descent

In this section we present the results taken when applying the method of Block Coordinate Descent at the same data as in section 6.2.1. As in the previous section, the left figures present the distance of f at each iteration k from p_*, whereas the right figures present the distance of $\mathbf{x}$ at each iteration k from $\mathbf{x}_*$.

We observe that f is minimizing and the optimal point $\mathbf{x}_*$ is reached since all figures tend to 0, so our method converges irrespective of the problem size and the number of processes.

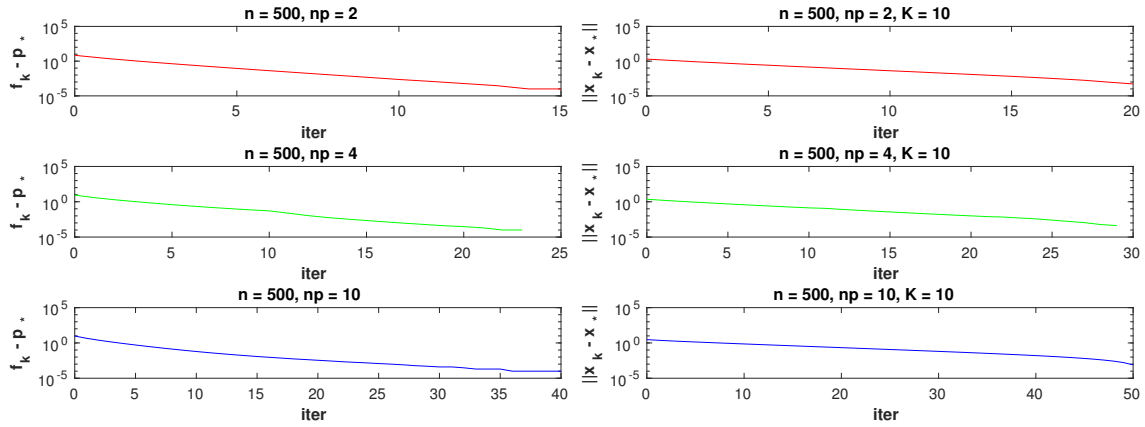


Figure 6.4: Block Coordinate Descent method with $n = 500$ and $K = 10$

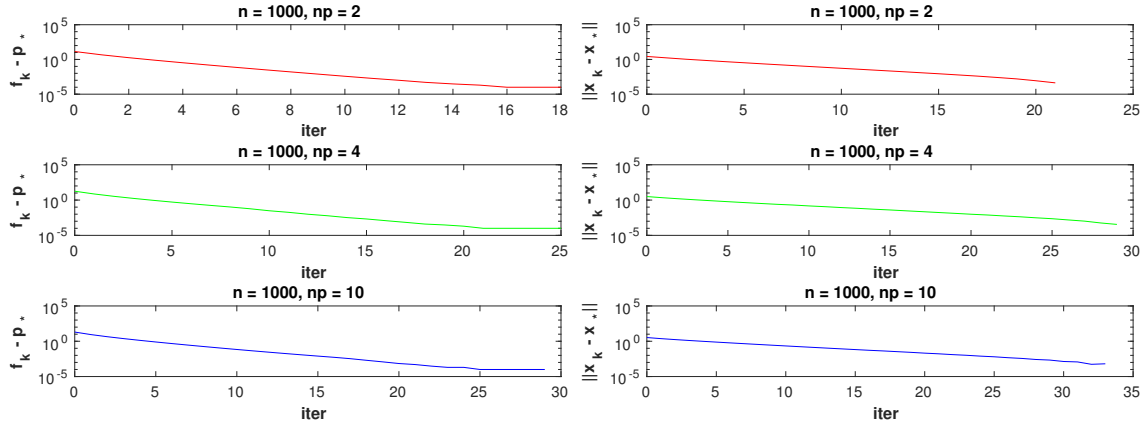


Figure 6.5: Block Coordinate Descent method with $n = 1000$ and $K = 10$

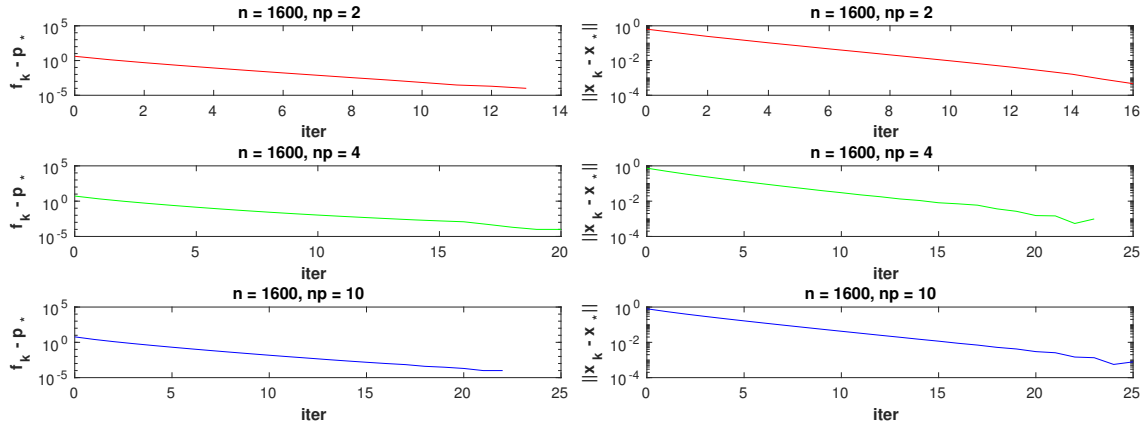


Figure 6.6: Block Coordinate Descent method with $n = 1600$ and $K = 10$

The number of iterations and the run time for the above problems are shown in the table below:

Number of iterations			
Problem size n	$np = 2$	$np = 4$	$n = 10$
500	22	31	52
1000	23	31	35
1600	18	25	27

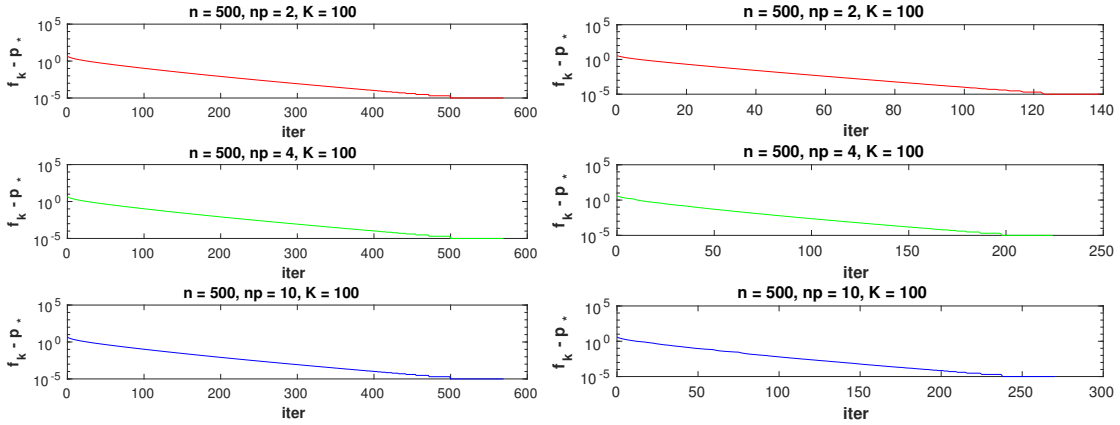
Run Time			
Problem size n	$np = 2$	$np = 4$	$n = 10$
500	4.36	2.24	2.08
1000	47.45	14.73	7.73
1600	147.73	44.39	11.03

From the multiple times we run the algorithm, but also from the results presented above, we observed that the problem size does not interfere much on the number of iterations, but only on the run time of the algorithm, just like the Gradient Descent method. However, the run time increases more when the problem size increases, in compare to the Gradient Descent method.

An essential notice is that the number of iterations the algorithm needs until the optimal point is reached increases as the number of processes increase, whereas the run time is significantly reduced. This is very important, since it might take more iterations to converge, but it needs less time, so the algorithm is more efficient.

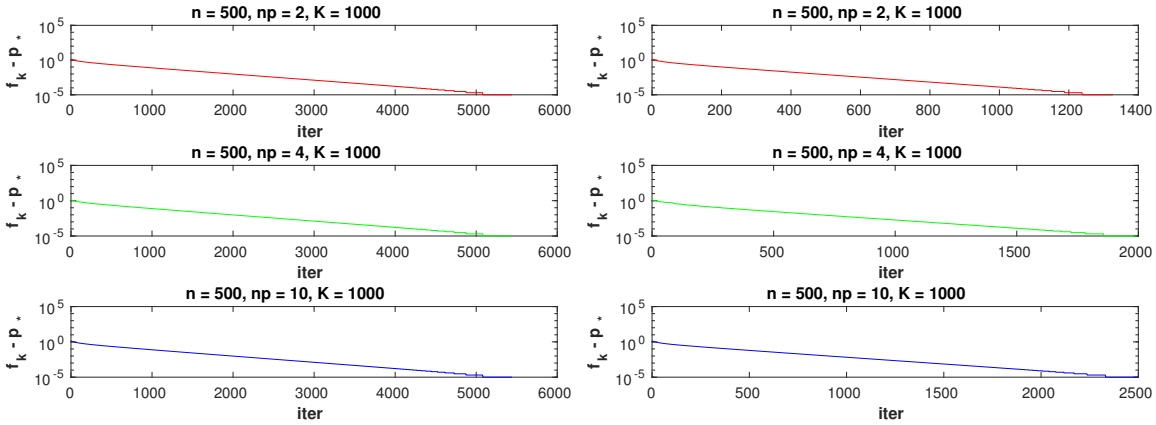
6.3 Effect of condition number K

So far, we have seen the effect that the size of the problem and the number of processes have on both our algorithms. In this section we will study the effect of the condition number K. We will focus on problems with $n = 500$, np will be on the set $\{2, 4, 10\}$ and we will present results with $K = \{100, 1000\}$. Note that we have already presented results with $K = 10$ in the previous section.



(a) Gradient Descent (b) Block Coordinate Descent

Figure 6.7: Distance of f from p_* when $n = 500$ and $K = 100$



(a) Gradient Descent (b) Block Coordinate Descent

Figure 6.8: Distance of f from p_* when $n = 500$ and $K = 1000$

The number of iterations and the run time for the above problems are:

Number of iterations for Gradient Descent			
Condition number K	$np = 2$	$np = 4$	$n = 10$
100	698	698	698
1000	7255	7255	7255

Run Time for Gradient Descent			
Condition number K	$np = 2$	$np = 4$	$n = 10$
100	4.5	9.88	51.47
1000	49.48	90.18	646.2

Number of iterations for Block Coordinate Descent			
Condition number K	$np = 2$	$np = 4$	$n = 10$
100	153	246	297
1000	1607	2415	3012

Run Time for Block Coordinate Descent			
Condition number K	$np = 2$	$np = 4$	$n = 10$
100	45.76	20.13	15.24
1000	468.54	170.18	150.55

From the figures above, we observe that when the condition number increases, the number of iterations needed for both algorithms to reach the optimal point increases proportionally. More specifically, we notice that when K increases by the power of 10, the iterations also increase by the power of 10.

We should also notice that, when K increases, the run time of both algorithms also increases proportionally.

6.4 Introducing probability p

In this phase of the thesis, we introduce probability p to our programs.

In fact, before the iterative method begins, we set variable p equal to a value taken from the set $\{0.1, 0.2, \ldots, 1\}$. Then, at each iteration, each process separately, choses randomly a value in the interval $[0, 1]$, and stores it in variable tmp_p. If the randomly chosen tmp_p is less than probability p, then the process moves on to calculating new $\mathbf{x}_{local}$, otherwise

$$\mathbf{x}_{local}^{k+1} = \mathbf{x}_{local}^{k}.$$

Actually, after each iteration, some blocks of $\mathbf{x}$ are updated, and some others not.

Algorithm 6.1 Parallel algorithm with probability p

```
Set x_0 ∈ R^n, k = 0, p
start_time = MPI_Wtime();
while (terminating condition is FALSE) do
  tmp_p = rand()
  if (tmp_p < p)
      update x_local
  MPI_Allgather(x_local^{k+1})
  k := k + 1
end while
end_time = MPI_Wtime();
```

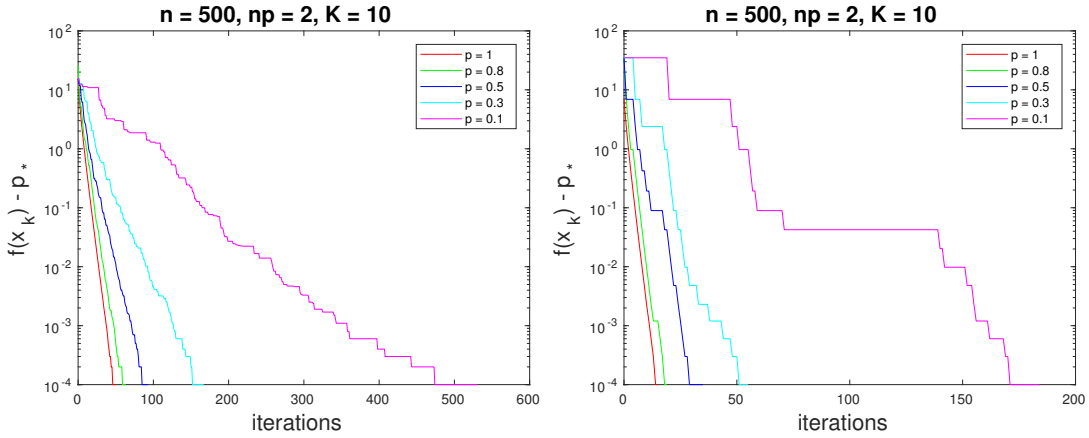
Probability p was introduced in order to observe the changes in the number of iterations and the run time of the algorithms, as well as if the optimal point will be affected by the fact that at each iteration not every process updates its $\mathbf{x}_{local}$.

Thus, in order to measure the time taken for both methods to minimize the function and calculate the optimal point $\mathbf{x}_*$, we use MPI function MPI_Wtime, as described in section 3.7.

A pseudocode of the parallel program with the probability p is shown in Table 6.1.

6.5 Effect of probability p

In this section, we present results after running both algorithms to observe the effect probability p has on their convergence. Since we have analyzed the effect of n, np and K in the previous sections, for convenience we will present results when $n = 500$, $np = 2$ and $K = 10$. Probability p is chosen to be in the set $\{0.1, 0.3, 0.5, 0.8, 1\}$.



(a) Gradient Descent (b) Block Coordinate Descent

Figure 6.9: Distance of f from p_* with $p = \{0.1, 0.3, 0.5, 0.8, 1\}$, when $n = 500$, $np = 2$ and $K = 10$

The number of iterations and the run time of the algorithms are showen in the tables below:

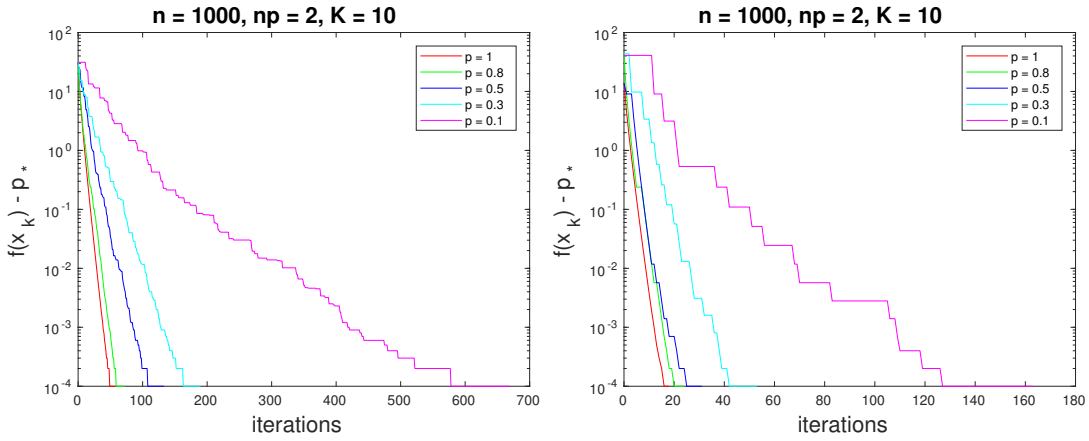
Number of iterations					
Method	$p = 1$	$p = 0.8$	$p = 0.5$	$p = 0.3$	$p = 0.1$
Gradient Descent	77	94	156	252	829
Block Coordinate Descent	22	27	49	72	248

Number of iterations					
Method	$p = 1$	$p = 0.8$	$p = 0.5$	$p = 0.3$	$p = 0.1$
Gradient Descent	0.33	1.08	1.93	2.37	6.12
Block Coordinate Descent	6.36	6.58	7.5	7.61	8.02

From the figures above, we observe that, when p decreases, the number of iterations and the run time of the algorithms increase. This makes sence, since, as p decreases, the probability of tmp_p to be less than p decreases, which means not every process updates its $\mathbf{x}_{local}$ at every iteration, so it takes more iterations and more time to reach the optimal point $\mathbf{x}_*$.

However, despite the increase in the number of iterations, the algorithm always reaches the optimal point $\mathbf{x}_*$, so probability p does not affect the result of the algorithms.

We will now increase the problem size to $n = 1000$ to see if the methods behave any differently. We have:



(a) Gradient Descent (b) Block Coordinate Descent

Figure 6.10: Distance of f from p_* with $p = \{0.1, 0.3, 0.5, 0.8, 1\}$, when $n = 1000$, $np = 2$ and $K = 10$

The number of iterations and the run time of both algorithms are:

Number of iterations					
Method	$p = 1$	$p = 0.8$	$p = 0.5$	$p = 0.3$	$p = 0.1$
Gradient Descent	78	95	173	250	820
Block Coordinate Descent	23	30	41	59	184

Run time					
Method	$p = 1$	$p = 0.8$	$p = 0.5$	$p = 0.3$	$p = 0.1$
Gradient Descent	1.3	1.8	4.34	4.6	12.19
Block Coordinate Descent	47.45	59.72	76.21	79.84	84.17

The results we got when we increase the problem size were expected. More specifically, the number of iterations of both methods remained about the same, whereas the run time of the algorithms increased. However, p affected both algorithms in the same way since, when p decreases, the number of iterations and the run time of the algorithms increases.

We mention though that both algorithms always reach the optimal point, even when p is very small, so probability p does not affect the result of the algorithms, just when it is reached.

6.6 Comparison of the methods

If we compare the results of sections 6.2.1 and 6.2.2, we observe that Block Coordinate Descent needs with less iterations until it converges from Gradient Descent method. However, it is very interesting to observe the run time it takes for both methods to converge.

When the number of processes is 'small', Block Coordinate Descent needs more time to reach the optimal point than Gradient Descent. On the other hand, when the number of processes increases, the run time decreases, and Block Coordinate Descent now converges faster than Gradient Descent.

So, we conclude that Gradient Descent Method is more suitable on systems with less processes, and Block Coordinate Descent for systems with more processes.

6.7 Communication Cost

In this section, we consider the total communication cost for each process for each method. The normalized communication cost is defined as

$$\frac{iters(p) \times p \times E}{iters(1) \times 1 \times E} \tag{6.1}$$

where $iters(p)$ is the number of iterations it takes for the method to converge with probability p, $iters(1)$ is the number of iterations when $p = 1$, and E is the unit communication cost.

Since the unit communication cost is independent of the probability p, the definition can be simplified as:

$$\frac{iters(p) \times p}{iters(1)} \tag{6.2}$$

Our main purpose is to examine the communication cost for each probability p and discover an optimal p which minimizes the total communication cost for each method.

The figures on the left below present the average number of iterations both methods need to converge, calculated after multiple runs of both algorithms, whereas the figures on the right show the communication cost, based on the average number of iterations, as calculated using equation 6.2.

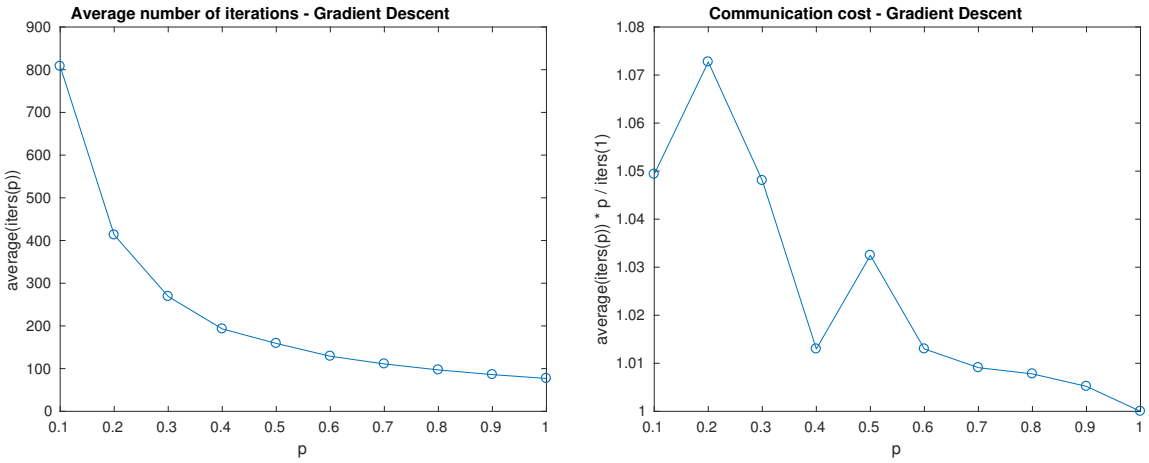


Figure 6.11: Average number of iterations and communication cost for Gradient Descent

From the figures above, we can observe that the optimal communication cost is achieved on average for $p = 1$. We should also recall that the minimum number of iterations and run time of the algorithm were achieved when $p = 1$. Having said that, we conclude that adding the factor of p does not improve the performance of the method.

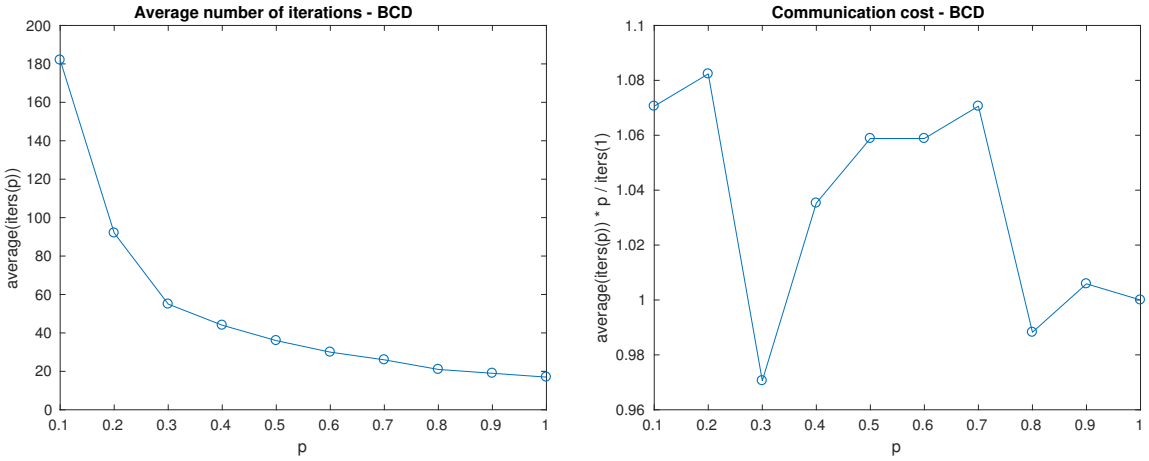


Figure 6.12: Average number of iterations and communication cost for Block Coordinate Descent

From the figures presented above, we can notice that the optimal communication cost is achieved on average for $p = 0.3$. However, as observed in section (6.5), p affects the number of iterations, since, as p descreases, the number of iterations increase, whereas it has some or little influence on the run time of the algorithm, based on the data of the system. Therefore, the probability chosen for the method depends on the system available, as well as on our requirements.

Chapter 7

Conclusion

In this thesis, we considered a convex optimization problem with a quadratic cost function, which was partitioned into a set of processors with their own local cost function. We examined the Gradient Descent method and the Block Coordinate Descent method to solve the above problem, and used Message Passing Interface (MPI) to parallelize the methods. We tested the convergence properties and the computational cost of both algorithms under various circumstances, and we reached the conclusion that the Gradient Descent is more suitable for systems with a few processors, whereas the Block Coordinate Descent performs better with more processors.

Bibliography

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

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

[3] P. S. Pacheco, *"Parallel Programming in MPI"*, Morgan Kaufmann, 1997.

[4] D. P. Bertsekas, *"Convex Optimization Algorithms"*, Athena Scientific, 2015.

[5] D. P. Bertsekas, N. J. Tsitsiklis, *"Parallel and Distributed Computation: Numerical Methods"*, Athena Scientific, 1997

[6] "`https://en.wikipedia.org/wiki/Gradient_descent`"

[7] MPICH, "`https://www.mpich.org/static/docs/v3.2/www3/`"