Distributed linear programming with Apache Spark

by

Ehsan Mohyedin Kermani

B.Sc. Sharif University of Technology, 2011

M.Sc. University of British Columbia, 2013

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

Master of Science

in

THE FACULTY OF GRADUATE AND POSTDOCTORAL
STUDIES

(Computer Science)

The University of British Columbia
(Vancouver)

December 2016

Abstract

For this thesis project, we have implemented Mehrotra's predictor-corrector interior point algorithm on top of Apache Spark for solving large-scale linear programming problems. Our large-scale solver (Spark-LP) is unique because it is open-source, fault-tolerant and can be used on commodity cluster of machines. As a result, Spark-LP provides an opportunity to solve large-scale problems at the lowest possible cost. We have assessed the performance and convergent results of our solver on self-generated, sparse and dense large-scale problems over small to medium-sized clusters, composed of 16 to 64 Amazon's Elastic Computing Cloud r3.xlarge instances. In conclusions, we have made important suggestions for breaking the current structural limitations so that our solver can be used on heterogeneous clusters containing CPUs and GPUs on JVM environment without the usual numerical limitations and overheads.

Preface

All the research ideas and methods explained in this thesis are the results of fruitful discussions between Professor Uri Ascher, Professor Chen Greif and Ehsan Mohyedin Kermani. The software design and obtaining the results were executed by Ehsan Mohyedin Kermani. The manuscript preparations were conducted by Ehsan Mohyedin Kermani with invaluable guidance from Uri Ascher and Chen Greif throughout this process.

Table of contents

List of tables

List of figures

Acknowledgments

First and foremost, I would like to thank my supervisors Professor Uri Ascher and Professor Chen Greif for all their help, trust and financial support throughout my study at the University of British Columbia (UBC). I am indebted to them for the invaluable opportunity they gave me to join the scientific computing lab of the computer science department at UBC. I am grateful to Professor Mark Greenstreet for his time in evaluating and second reading my thesis.

I appreciate Apache Spark community friendly software development environment and would like to thank Databricks company for granting me their academic cluster resources for finalizing the results.

I would also like to express wholeheartedly, my sincere appreciation to my very supportive family, my mother “Mehrzad”, my father “Hamid” and my sister “Afsoon” for all their love and endless encouragements throughout my life.

As last but not least, my deepest appreciation goes to my beloved wife “Shadi” for her continuous love and immense support.

Chapter 1

Introduction

The *volume* of sheer data that is being generated world wide, is increasing at a greater *velocity* than ever before. From Physics particle discovery data to social media data, stock market data and so on. Data comes in different *varieties*, shapes and formats. The three mentioned characteristics have coined the word *Big Data* in industrial side. Nonetheless, the emergence of Big Data in computational science, in particular high performance computing, dates back to a few decades before the industrial recognition for large-scale numerical applications.

In 2004, Google introduced the MapReduce framework [23] to analyze Big Data in order to extract knowledge and insight using large-scale (commodity) clusters of machines in a distributed way. The main features of MapReduce are handling machine failure systematically (*fault-tolerance*), and its minimal programming framework where users need to express the computations in terms of two functions: `map` and `reduce`. This simplistic model allows users to concentrate on their applications by hiding the harder parts through various abstractions. However, MapReduce is insufficient when it comes to fast iterative applications because of the following two main reasons:

- MapReduce does not leverage data persistence well enough.
- There is a significant I/O bottleneck in each iteration.

On the other hand, in high performance computing where objectives are merely high accuracy and fast execution of computations, the current solution for analyzing Big Data is the computational and programming framework known as message passing interface (MPI). MPI supports a variety of functionalities with intricate details that users must explicitly control, resulting into high accuracy and fast execution of computations [15]. However, MPI is not a feasible option from the industrial point of view dealing with Big Data, as the needed infrastructure is very expensive and hard to maintain [26].

In 2010, Apache Spark generalized MapReduce by introducing its distributed memory abstraction, called Resilient Distributed Datasets (RDD) [28]. Apache Spark lies between MapReduce and MPI and has closed the gap between industrial and high performance computing perspectives. In fact, Spark solved the aforementioned issues with MapReduce by

- Leveraging different levels of data persistence in memory so that iterative applications can run much faster.
- Removing the storage requirement for intermediate results.

Apache Spark as the distributed computing platform for Big Data applications, has provided distributed linear algebra and convex optimization supports through one simple assumption that divides vector and matrix operations: vectors are *local* and matrices must be distributed across the cluster. That is, since matrices are quadratically taking more storage space than vectors, then vector operations should be kept local while matrix operations should be distributed across the cluster [30]. For example, matrix-matrix operations are done over the cluster and matrix-local vector operations can be efficiently done by *broadcasting* the local vector to workers containing the matrix partitions.

Our main motivation is to assess Spark for large-scale numerical applications which are closer to high performance computing and to analyze its advantages and disadvantages. For this reason, we have chosen linear programming which is one of the most fundamental and well-known optimization techniques in scientific computing area.

A linear programming in standard form is described as

$$\begin{aligned} \min \quad & c^T x \\ \text{subject to} \quad & Ax = b \\ & x \geq 0 \end{aligned}$$

where c, x are in $\mathbb{R}^n$, b is in $\mathbb{R}^m$, A is an $m \times n$ real-valued matrix.

We have implemented *Mehrotra's predictor-corrector interior point algorithm*, described in Section 2.2.3, to solve **large-scale** linear programming problems on top of Apache Spark in [20, Spark-LP]. Briefly, when a user specifies c, b as *distributed vectors* (`RDD[DenseVector]`) and A^T as *distributed matrix* (`RDD[Vector]`) by determining the same number of *partitions* on their underlying data, the Spark diver program is connected to the cluster via a (shared) `SparkContext` instance as described in Section 3.2.3. Then, the resource manager allocates Spark workers (executors), distributes c, b and A among workers, and the scheduler schedules *tasks* to run in executors in parallel. Note that, as Mehrotra's algorithm iteratively updates x, the computations in each iteration happen in *multiple parallel steps* depending on the number of partitions and the total number of available cores in the cluster. Additionally, Spark tries to manage most of the computations locally within each worker and limits the number of communications across workers.

To leverage hardware accelerations and fast local computations, we have used *Basic Linear Algebra Subprograms* (BLAS) and *Linear Algebra Package* (LAPACK) native routines written in *Fortran*, through the *Java Native Interface* (JNI) provided by Netlib-java [17].

Our solver [20] is **open-source**, **fault-tolerant** and can be used on **commodity clusters** of machines for solving large-scale linear programming problems. As a result, it provides an opportunity to solve large-scale problems at the *lowest* possible cost.

Note that there are a number of software packages for solving large-scale linear programming problems but typically they are *not* fault-tolerant, and therefore they require access to expensive infinite bandwidth clusters. As such, they are not straightforward to use in Big Data and large-scale setting on commodity clusters. In addition, the majority of large-scale solvers are *proprietary* and very expensive to use.

For testing/tuning purposes and describing the inevitable overheads that Spark brings to the table, compared to non fault-tolerant solvers with low level controls such as GLPK [10], we have included some of the local results from solving small sized Netlib [18] problems in Section 4.2.1. We have generated large-scale problems in order to assess the performance of our solver on large-scale problems over small to medium clusters. The results are provided and analyzed in Section 4.2.2.

Finally in Chapter 5, we have made suggestions for improving the current structural limitations namely how to use a heterogeneous cluster consisting of CPUs and GPUs by using `RDD[INDArray]` provided in [16, ND4J] library.

Chapter 2

Linear programming and primal dual interior point methods

In this chapter, first we briefly introduce some of the basic properties of linear programming. Then, we end this chapter with detailed description of Mehrohra's predictor-corrector interior point algorithm given in Section 2.2.3.

2.1 Linear programming

Linear programming (LP) is a special case of mathematical optimization where both the objective and constraints are described by linear equations. Formally, its standard form is written as

$$\begin{aligned} \min \quad & c^T x \\ \text{subject to} \quad & Ax = b \\ & x \geq 0 \end{aligned} \tag{2.1}$$

where c, x are in $\mathbb{R}^n$, b is in $\mathbb{R}^m$, A is an $m \times n$ real-valued matrix and $(\cdot)^T$ is the transpose operation. Throughout, we will assume $\text{rank}(A) = m \leq n$.

Associated to the above LP problem in *primal* form, is the *dual* problem as follows

$$\begin{aligned} \max \quad & b^T\lambda \\ \text{subject to} \quad & A^T\lambda + s = c \\ & s \geq 0 \end{aligned} \tag{2.2}$$

where $\lambda \in \mathbb{R}^m$ and $s \in \mathbb{R}^n$.

Historically [33], Dantzig introduced the first algorithm, called thes *simplex method* to efficiently tackle most LP cases in 1940s. Later in 1979, Khachiyan showed that LP problems are solvable in polynomial time. In 1984, Karmaker introduced a new class of methods known as *interior-point methods* which led to the discovery of *primal-dual algorithms*. Among such algorithms, *Mehrohra's predictor-corrector* interior point method has had significant impact on computational side and software aspect of LP solvers since 1990. This method is the main subject of this chapter.

2.1.1 Duality theorems and Farkas lemma

First, we introduce the following definitions

Definition 2.1.1. *Let $\mathscr{F}_P = \{x \mid Ax = b, x \geq 0\}$ be the set of* primal feasible solutions *of* (2.1). *Then,* (2.1)

a) is infeasible, *if $\mathscr{F}_P = \emptyset$*

b) is unbounded, *if there is a sequence $(x^k)_{n\geq 1}$ of elements of $\mathscr{F}_P$ such that $\lim_{k\to\infty} c^T x^k = -\infty$*

Definition 2.1.2. *A* primal optimal solution $x^\star$ *of* (2.1) *is a primal feasible solution such that $c^T x^\star \leq c^T x$ for all $x \in \mathscr{F}_P$.*

Similarly, we have the following definitions for the dual problem (2.2)

Definition 2.1.3. *Let $\mathscr{F}_D = \{(\lambda, s) \mid A^T\lambda + s = b, s \geq 0\}$ be the set of* dual feasible solutions *of* (2.2). *Then,* (2.2)

a) is infeasible, *if $\mathscr{F}_D = \emptyset$*

b) *is* unbounded, *if there is a sequence* $(\lambda^k, s^k)_{n\geq 1}$ *of elements of* $\mathscr{F}_D$ *such that* $\lim_{k\to\infty} b^T\lambda^k = +\infty$

Definition 2.1.4. *A* dual optimal solution $(\lambda^\star, s^\star)$ *of* (2.2) *is a dual feasible solution such that* $b^T\lambda^\star \geq b^T\lambda$ *for all* $(\lambda, s) \in \mathscr{F}_D$.

Therefore, the following results bridge the gap between the solutions of primal and dual LP problems

Theorem 2.1.5 (Weak Duality [32])**.** *Let* $x \in \mathscr{F}_P$ *and* $(\lambda, s) \in \mathscr{F}_D$. *Then,* $c^Tx \geq b^T\lambda$, *where equality holds if and only if* $x^T(c - A^T\lambda) = 0$, *or equivalently,*

$$x_i(c - A^T\lambda)_i = 0 \text{ for all } 1 \leq i \leq n$$

Proof. We have $Ax = b$, then $b^T\lambda = (Ax)^T\lambda = x^TA^T\lambda$. Therefore,

$$c^Tx - b^T\lambda = c^Tx - x^TA^T\lambda = x^Tc - x^TA^T\lambda = x^T(c - A^T\lambda) = x^Ts \geq 0.$$

□

The condition $x_i(c - A^T\lambda)_i = x_is_i = 0$ for all $1 \leq i \leq n$, is known as the *complementarity* condition and $\mu = c^Tx - b^T\lambda$ is called the *duality gap* for the solution $x \in \mathscr{F}_P$ of (2.1) and $\lambda \in \mathscr{F}_D$ of (2.2).

The following consequences of the Weak Duality Theorem are immediate

Corollary 2.1.6. *a) If* (2.1) *is unbounded then* (2.2) *is infeasible.*

b) If (2.2) *is unbounded then* (2.1) *is infeasible.*

c) If there is a $x^\star \in \mathscr{F}_P$ *and an* $\lambda^\star \in \mathscr{F}_D$ *such that* $c^Tx^\star = b^T\lambda^\star$, *then* $x^\star$ *is an optimal solution to* (2.1) *and* $\lambda^\star$ *is an optimal solution to* (2.2).

The corollary Theorem 2.1.6 establishes sufficient conditions for the infeasibility of (2.1) and (2.2) as well as the optimality of a pair of primal and dual solutions. However, infeasibility of one of the problems does not imply unboundedness of the other and the true reverse statement can be obtained by the following theorems, known as Farkas Lemma

Theorem 2.1.7 (Primal Farkas Lemma [24])**.** *Exactly one of the two systems has a feasible solution*

a) $Ax = b,\ x \geq 0$

b) $A^T\lambda \leq 0,\ b^T\lambda > 0$

Theorem 2.1.8 (Dual Farkas Lemma)**.** *Exactly one of the two systems has a feasible solution*

a) $c^Tx < 0,\ Ax = 0,\ x \geq 0$

b) $A^T\lambda \leq c$

We can use the Farkas Lemma to prove the following

Corollary 2.1.9. *a) If* (2.1) *is infeasible, then* (2.2) *is either infeasible or unbounded.*

b) If (2.2) *is infeasible, then* (2.1) *is either infeasible or unbounded.*

Proof. We prove $a)$ and the proof of $b)$ will be similar. If (2.1) is infeasible, then there is no $x \in \mathbb{R}^n$ such that

$$Ax = b,\ x \geq 0.$$

Then, by Primal Farkas Lemma Theorem 2.1.7, there exists a $\lambda \in \mathbb{R}^m$, such that

$$A^T\lambda \leq 0,\ b^T\lambda > 0.$$

If (2.2) is infeasible, then we are done. But if (2.2) is feasible, then there exists $\bar{\lambda} \in \mathbb{R}^m$ such that $A^T\bar{\lambda} \leq c$ and for any $\alpha > 0$, we have

$$A^T(\bar{\lambda} + \alpha\lambda) \leq c$$

resulting into

$$\lim_{\alpha \to +\infty} b^T(\alpha\lambda + \bar{\lambda}) = +\infty.$$

Hence, (2.2) is unbounded.

□

The above theorem establishes the reverse statements of corollary Theorem 2.1.6 parts *a*) and *b*). The reverse of part *c*) is highly non-trivial. We know that if both (2.1), (2.2) have feasible solutions, by Weak Duality, the optimal values are bounded. However, it is not clear if both primal and dual problems have optimal solutions, whether the optimal objective values of primal and dual are equal or not. That is, in fact, reflected in the following theorem

Theorem 2.1.10 (Strong Duality [22])**.** *If one of the primal or dual problems has a finite optimal value, then so does the other and the optimal objective values are equal.*

That is, $c^T x^\star = b^T \lambda^\star$, *where* $x^\star$ *is a primal optimal solution and* $\lambda^\star$ *is a dual optimal solution.*

2.2 Primal-dual interior point methods

The solution of primal (2.1) and dual (2.2) problems (x, λ, s) are characterized by the Karush-Kuhn-Tucker conditions

$$\begin{aligned} A^T\lambda + s &= c \\ Ax &= b \\ x &\geq 0 \\ s &\geq 0 \\ x_i s_i &= 0, \;\; 1 \leq i \leq n \end{aligned}$$

Primal-dual methods find solutions $(x^\star, \lambda^\star, s^\star)$ of the above system, by applying Newton's method and modifying search directions and step lengths so that we have the strict inequality $x^T s > 0$ in every iteration.

To obtain primal-dual interior point methods, we can write the above equations in a single mapping as follows

$$F(x,\lambda,s) = \begin{pmatrix} A^T\lambda + s - c \\ Ax - b \\ XSe \end{pmatrix} = 0,\ x^T s \geq 0$$

where $X = \text{diag}(x_1, \cdots, x_n)$, $S = \text{diag}(s_1, \cdots, s_n)$ and e is the vector of all 1s.

To compute the Newton search directions, we need to solve

$$J(x,\lambda,s) \begin{pmatrix} \Delta x \\ \Delta\lambda \\ \Delta s \end{pmatrix} = -F(x,\lambda,s)$$

where

$$J(x,\lambda,s) = \begin{pmatrix} 0 & A^T & I \\ A & 0 & 0 \\ S & 0 & X \end{pmatrix}$$

is the Jacobian of F.

In order to preserve the condition $x^T s > 0$, we can do line search along the Newton direction and define the updates as

$$(x,\lambda,s) + \alpha(\Delta x, \Delta\lambda, \Delta s)$$

for some $\alpha \in (0,1]$.

In most primal-dual methods, the goal is milder than directly solving the above equations. It is about taking a Newton step towards a point which $x_i s_i = \sigma\mu$, where

$$\mu = \sum_{i=1}^{n} \frac{x_i s_i}{n} = \frac{x^T s}{n}$$

is the current *duality measure* and $\sigma \in [0,1]$ is the reduction factor in the duality measure (also known as the *centering parameter*).

Thus, the simplified step direction is computed by solving

$$\begin{pmatrix} 0 & A^T & I \\ A & 0 & 0 \\ S & 0 & X \end{pmatrix} \begin{pmatrix} \Delta x \\ \Delta \lambda \\ \Delta s \end{pmatrix} = \begin{pmatrix} -r_c \\ -r_b \\ -XSe + \sigma \mu e \end{pmatrix} \tag{2.3}$$

where $r_b = Ax - b$ and $r_c = A^T \lambda + s - c$. Therefore, here is the pseudo-code for the primal-dual interior point methods algorithm discussed above

```
procedure PRIMAL-DUAL INTERIOR POINT METHODS($c$,$A$,$b$) [27, page 396]
    Given $(x^0, \lambda^0, s^0)$ where $(x^0)^T s^0 > 0$

    for $k = 0, 1, 2, \cdots$ do
        Choose $\sigma_k \in [0,1]$ and solve
            $\begin{pmatrix} 0 & A^T & I \\ A & 0 & 0 \\ S^k & 0 & X^k \end{pmatrix} \begin{pmatrix} \Delta x^k \\ \Delta \lambda^k \\ \Delta s^k \end{pmatrix} = \begin{pmatrix} -r_c^k \\ -r_b^k \\ -X^k S^k e + \sigma_k \mu_k e \end{pmatrix}$
        where $\mu_k = \frac{(x^k)^T s^k}{n}$.
        Set $(x^{k+1}, \lambda^{k+1}, s^{k+1}) = (x^k, \lambda^k, s^k) + \alpha_k (\Delta x^k, \Delta \lambda^k, \Delta s^k)$
        for some $\alpha_k$ such that $(x^{k+1})^T s^{k+1} > 0$.
```

2.2.1 A practical algorithm

Newton's method forms a linear model (*linearization*) around the current point and solves Equation 2.3 to obtain a search direction, this is known as *affine scaling direction*. This linearization has an error in modeling the equation $x_i s_i = 0$ for $1 \leq i \leq n$. A key point in practical algorithms is their use of corrector step [27].

That is, if we consider the affine scaling direction obtained from Equation 2.3

$$\begin{pmatrix} 0 & A^T & I \\ A & 0 & 0 \\ S & 0 & X \end{pmatrix} \begin{pmatrix} \Delta x^{\text{aff}} \\ \Delta \lambda^{\text{aff}} \\ \Delta s^{\text{aff}} \end{pmatrix} = \begin{pmatrix} -r_c \\ -r_b \\ -XSe + \sigma \mu e \end{pmatrix} \tag{2.4}$$

then, taking a full step in this direction will lead us to

$$(x_i+\Delta x_i^{\text{aff}})(s_i+\Delta s_i^{\text{aff}}) = x_i s_i + x_i \Delta s_i^{\text{aff}} + s_i \Delta x_i^{\text{aff}} + \Delta x_i^{\text{aff}} \Delta s_i^{\text{aff}} = \Delta x_i^{\text{aff}} \Delta s_i^{\text{aff}}$$

However, that shows that the updated $x_i s_i$ is $\Delta x_i^{\text{aff}} \Delta s_i^{\text{aff}}$ not the desired value of 0. To correct that, we can solve the following system

$$\begin{pmatrix} 0 & A^T & I \\ A & 0 & 0 \\ S & 0 & X \end{pmatrix} \begin{pmatrix} \Delta x^{\text{cor}} \\ \Delta \lambda^{\text{cor}} \\ \Delta s^{\text{cor}} \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ -X^{\text{aff}} S^{\text{aff}} e \end{pmatrix} \tag{2.5}$$

Therefore, we need to combine the two solutions from Equation 2.4 and Equation 2.5 to obtain

$$(\Delta x^{\text{aff}}, \Delta \lambda^{\text{aff}}, \Delta s^{\text{aff}}) + (\Delta x^{\text{cor}}, \Delta \lambda^{\text{cor}}, \Delta s^{\text{cor}}).$$

In most practical cases, the above update reduces the duality gap more than pure affine scaling update.

2.2.2 Starting point

The choice of an starting point (x^0, λ^0, s^0) is crucial in making the algorithm robust to failure in convergence. The following is a heuristic that finds a starting point satisfying the equality constraints in primal and dual problems reasonably well, while ensuring the positivity of x and s. We first solve [27, page 410]

$$\min_x \frac{1}{2} x^T x \text{ subject to } Ax = b$$

$$\min_{(\lambda,s)} \frac{1}{2} s^T s \text{ subject to } A^T \lambda + s = c$$

Their solutions are $\tilde{x} = A^T (AA^T)^{-1} b$, $\tilde{\lambda} = (AA^T)^{-1} Ac$, $\tilde{s} = c - A^T \tilde{\lambda}$.

We need to adjust the solutions so that they have positive components. Therefore, let

$$\delta_x = \max\{-1.5 \min_i \tilde{x}_i, 0\}, \quad \delta_s = \max\{-1.5 \min_i \tilde{s}_i, 0\},$$

and we update $\tilde{x}, \tilde{s}$ as follows

$$\hat{x} = \tilde{x} + \delta_x e, \ \hat{s} = \tilde{s} + \delta_s e,$$

where $e = (1, 1, \cdots, 1)^T$.

To ensure the starting points are not too close to zero, we define

$$\tilde{\delta}_x = \frac{1}{2} \frac{\hat{x}^T \hat{s}}{e^T \hat{s}}, \ \ \tilde{\delta}_s = \frac{1}{2} \frac{\hat{x}^T \hat{s}}{e^T \hat{x}}.$$

Thus, the starting points are

$$\begin{aligned} x^0 &= \hat{x} + \hat{\delta_x} e \\ \lambda^0 &= \tilde{\lambda} \\ s^0 &= \hat{s} + \hat{\delta}_s e \end{aligned} \tag{2.6}$$

Simplifying the equations Equation 2.4 and Equation 2.5 lead us to the following Mehrotra's predictor-corrector algorithm.

2.2.3 Mehrotra's predictor-corrector algorithm

The following is the Mehrotra's predictor-corrector algorithm that we will use later

```
procedure SOLVE(c, A, b, tol, maxIter)
    Set $x_0, \lambda_0, s_0 \leftarrow \text{init(c, A, b)}$ from 2.6.
    Set $\text{iter} \leftarrow 1$.

    while not converged and $\text{iter} \leq \text{maxIter}$ do
        Set $r_b \leftarrow Ax - b$, $r_c \leftarrow A^T\lambda + s - c$, and $D \leftarrow S^{-1/2}X^{1/2}$.
        Solve using Cholesky decomposition:
            $$AD^2A^T\Delta\lambda^{\text{aff}} = -r_b + A^T(-D^2r_c + x).$$
        Set $\Delta s^{\text{aff}} \leftarrow -r_c - A^T\Delta\lambda^{\text{aff}}$, and $\Delta x^{\text{aff}} \leftarrow -x - D^2\Delta s^{\text{aff}}$.
        Set
            $$\alpha_{\text{aff}}^{\text{Pri}} \leftarrow \min\left\{1, \min_{i:\Delta x_i^{\text{aff}}<0} -\frac{x_i}{\Delta x_i^{\text{aff}}}\right\},$$
            $$\alpha_{\text{aff}}^{\text{Dual}} \leftarrow \min\left\{1, \min_{i:\Delta s_i^{\text{aff}}<0} -\frac{s_i}{\Delta s_i^{\text{aff}}}\right\}.$$
        Set $\sigma \leftarrow (\mu_{\text{aff}}/\mu)^3$ where $\mu$ is the duality gap $s^Tx/n$.
        Solve using Cholesky decomposition:
            $$AD^2A^T\Delta\lambda = -r_b + AD^2(-r_c + s + X^{-1}\Delta X^{\text{aff}}\Delta S^{\text{aff}}e - \sigma\mu X^{-1}e).$$
        Update $\Delta s \leftarrow -r_c - A^T\Delta\lambda$ and
            $$\Delta x \leftarrow -D^2\Delta s - x - S^{-1}\Delta X^{\text{aff}}\Delta S^{\text{aff}}e + \sigma\mu S^{-1}e.$$
        Choose $\eta_{\text{iter}} \in [0.9, 1)$ and compute
            $$\alpha_{\text{iter}}^{\text{Pri}} \leftarrow \min\{1, \eta_{\text{iter}}\alpha_{\text{iter}}^{\text{Pri}}\},$$
            $$\alpha_{\text{iter}}^{\text{Dual}} \leftarrow \min\{1, \eta_{\text{iter}}\alpha_{\text{iter}}^{\text{Dual}}\}.$$
        Update $x \leftarrow x + \alpha_{\text{iter}}^{\text{Pri}}\Delta x$, and $(\lambda, s) \leftarrow (\lambda, s) + \alpha_{\text{iter}}^{\text{Dual}}(\Delta\lambda, \Delta s)$.
    return $(x, \lambda, s)$
```

and the convergence condition is [33, page 226]

$$\frac{\|r_b\|}{1+\|b\|} < \text{tol} \ \text{ and } \ \frac{\|r_c\|}{1+\|c\|} < \text{tol} \ \text{ and } \ \frac{|c^T x - b^T \lambda|}{1+|c^T x|} < \text{tol}$$

with usual $\text{tol} = 10^{-8}$, where $\|.\|$ is the Euclidean norm.

Chapter 3

MapReduce, Apache Spark and distributed matrix computations

In this chapter, we introduce MapReduce framework for analyzing Big Data and describe some of its advantages and disadvantages. The framework is introduced in Section 3.1. Next, we explain the generalized MapReduce framework and its implementation in Apache Spark project in Section 3.2. Finally, in Section 3.3 we describe how efficient distributed linear algebra support is made possible in Apache Spark.

3.1 MapReduce

In 2004, Google introduced a new computational and programming framework for analyzing Big Data called *MapReduce*. In short, users of MapReduce framework

> specify the computation in terms of a `map` and a `reduce` function, and the underlying runtime system automatically parallelizes the computation across large-scale clusters of machines, handles machine failures (**fault-tolerant**) and **schedules** inter-machine communication to make efficient use of the network and disk [23].

In the rest of this section, we explain the programming model (Section 3.1.1) and the execution model (Section 3.1.2). The advantages and disadvantages of MapReduce particularly for iterative applications are discussed in Section 3.1.4.

3.1.1 Programming model

The major advantage of MapReduce is in hiding intricate but tedious details such as parallelization, fault-tolerance and inter-machine communication from users. Consequently, it allows users to merely think about their applications by expressing computations in terms of two functions: `map` and `reduce`.

This simplistic feature might seem very restrictive at first, but it actually is crucial in solving many problems involving Big Data [23]. We will talk about its advantages and disadvantages later in Section 3.1.4.

The map function accepts a set of pairs of *key/value* as input and produces another set of key/value *intermediate* pairs. Then, under the hood, the MapReduce library is responsible for grouping all intermediate *values* corresponding to a same key and passes them to the reduce function.

The reduce function accepts an intermediate key with its set of corresponding values and outputs a possibly smaller set of values. It is worth noting that the intermediate results are handed to the `reduce` via an `iterator` to allow handling sets of values that might not fit in memory buffer.

3.1.2 Execution model

When a MapReduce application is ready for execution, the following steps will occur [23]

1. One machine in our cluster is designated as *master* and others are considered as *workers*. The input files are divided into *M* pieces (typically each piece in the range of $16 - 64$ MB). Then copies (forks) of the program are sent via the network to all workers.

2. For each MapReduce job, there are *M* map *tasks* and *R* reduce tasks. The master picks idle workers and assigns a map task and a reduce task to each of them. The master also maintains a set of states *(idle, in-progress, completed)* for every map and reduce task in a special data structure.

3. A worker with a map task reads an input piece of data from containing key/-value pairs and feeds them to the user-defined map function. The output (in-

termediate key/value pairs) is stored in memory buffer of the worker. Then, the output is written to *local disk*, partitioned into *R* parts by a partitioning function. Moreover, the locations of these buffered pairs in local memory are sent to the master which is responsible for distributing these locations to idle workers for the reduce phase.

4. When a reduce worker is notified by the master about the mentioned locations, it uses the *Remote Procedure Call* (RPC), which in this particular case is a special protocol used in distributed computing for accessing data in a worker from another worker as if the other worker had the data already stored in its local disk, to read the buffered data stored in local disk. Once this is done, the reduce worker groups together all occurrences of the same key by sorting the intermediate keys. Note that in sorting, if the intermediate data does not fit in memory, an external sort is then used.

5. Once the sorting is done, the reduce worker iterates over the intermediate data and applies the user-defined reduce function to a unique key and its corresponding values. The output of the reduce function is appended to a final output file for this reduce partition.

6. The MapReduce job is completed when there is no more in-progress map or reduce tasks left.

3.1.3 Fault-tolerance

MapReduce has been designed to work with Big Data on large-scale clusters of machines. Therefore, it naturally must handle machine failures. This special mechanism is called *fault-tolerant*.

Fault-tolerant is guaranteed at data layer as well as application layer. That is, at data layer multiple replicas (usually $2-3$ times per piece) of data are stored in the underlying global *distributed file system* and at application layer the mechanism is described as follows:

Periodically, the master pings each worker and waits to get a response back in certain amount of time. If the master does not get back a response from a worker, it marks that worker as failed. Then depending on the worker state, the master behaves differently. In that, any in-progress map or reduce task on a failed worker is reset to *idle* and becomes eligible for rescheduling on other workers. And any completed map tasks on a failed worker are reset back to initial idle state and becomes eligible for rescheduling, because the output of a failed worker is unreachable to others from its local disk.

Furthermore, when a map task is executed by a worker *A* and it fails, then another worker *B* is executing the map task, all other workers executing reduce tasks are notified by the master for this change, and any reduce task that has not already read the data from *A*, will read the data from *B* instead.

3.1.4 Advantages and disadvantages

Some important advantages of MapReduce are listed below:

- Simplicity of the framework: allowing users to only think about the application by abstracting away the harder parts such as fault-tolerance and inter-machine communications.
- Locality of computations: the MapReduce master has the location information of the input data and attempts to schedule a map task on machine that contains a replica of the corresponding data. If the master fails to do so, it tries to schedule a map task near a replica. This lowers the amount of network communications significantly and speeds up the computations.
- Fault-tolerance: as described in Section 3.1.3

F or the purpose of this thesis, the disadvantages of MapReduce are as follows

- Very limited programming model: simplicity of the framework is a double-edged sword. Despite offering great power to tackle Big Data problems, there are still problems that become very hard to program and process with MapReduce.

- I/O bottleneck in iterative applications: due to how MapReduce was designed, for iterative applications that are the subject of this thesis, MapReduce is not suitable. Since in each iteration, data has to be read from and then written back to the underlying distributed file system, therefore, this introduces more bottlenecks to iterative applications and significantly reduces the speed.

- MapReduce does not leverage data persistence enough. In particular, for iterative applications it slows the computations further down.

In the next section, we will see how it is possible to solve the mentioned shortcomings for iterative applications by generalizing the framework.

3.2 Apache Spark as generalized MapReduce

As previously described, traditional MapReduce brings some overheads to iterative computations. In order to remove the overheads, such as relying on multiple reads and writes to the underlying stable distributed file system, a new abstraction, named *Resilient Distributed Datasets* (RDD) has been introduced and implemented in Apache Spark project using Scala [19] on Java Virtual Machine (JVM) environment.

To understand why RDD abstraction is powerful, we first need to explain more about RDD, its representation and the operations involved.

3.2.1 Resilient distributed datasets

In short, RDD is an immutable (read-only), distributed collection of objects. RDDs can be created either from a data in stable storage or from other RDDs through special operations. There are two types of operations acting of RDDs: *transformations* and *actions* (consider transformations as the generalized notion of map in MapReduce and actions as the generalized notion of reduce). More precisely, transformations are *lazy* operations that can be chained together and actions are operations that launch computations and return values.

Exploiting the lazy feature of transformations, RDDs need not to be materialized all the time throughout an application. Moreover, an RDD “knows” how it was created from previous RDDs by tracking its *lineage*. This is so powerful that it not only solves I/O bottleneck in iterative applications, but also it enables further optimizations that is explained in [31]. Note that unlike in MapReduce, no intermediate results need to be stored multiple times to ensure fault-tolerance. That is, a lost partition of an RDD can be automatically recomputed by following back its lineage. In essence, a program cannot reference an RDD that it cannot reconstruct after a failure [28].

Lastly, RDDs have two important customizable features: *persistence* and *partitioning*. That is, RDDs can be explicitly cached/persisted in memory or other storage strategies (including serialized in memory or disk, etc.) for efficient reuse. Additionally, RDD’s elements can be partitioned across machines based on a key in each record to enable further placement optimization.

3.2.2 Representation of RDDs and Spark execution model

Formally, an RDD is characterized by the following five items [28], [31]

1. A set of *partitions*, which are atomic pieces of underlying dataset.
2. A set of *dependencies* on parent RDDs.
3. A function for computing the dataset based on its parents.
4. Optionally, a partitioning function.
5. Optionally, a list of preferred locations for data placement.

To use Spark, an application developer write a *driver program* that connects to a cluster of *workers* through a `SparkContext` instance. Note that Spark code on the driver keeps track of the RDDs’ lineage under the hood.

Any Spark’s application has its own *Directed Acyclic Graph* (DAG) consisting of RDDs as vertices and transformations/actions as edges. When Spark driver wants to evaluate an action, it divides the DAG into *stages* at *DAG-Scheduler* and more importantly, optimization happens through pipelining the transformations (lazy evaluation of transformations plays a crucial role here). Then, each stage is scheduled further into *tasks* at the *Task-Scheduler* where actual computations start to happen.

It is necessary to mention that when driver wants to evaluate a transformation such as `map` on an RDD, it passes closures (function literals) to the underlying data. In Scala, each closure is a Java object, therefore, it is serialized and sent to another worker node in the cluster. Then it will be loaded and deserialized, ready to operate on underlying data.

The following illustrates the split of DAG into stages and tasks. Moreover, it shows how several tasks with narrow dependencies are optimized via pipelining them into a single task.

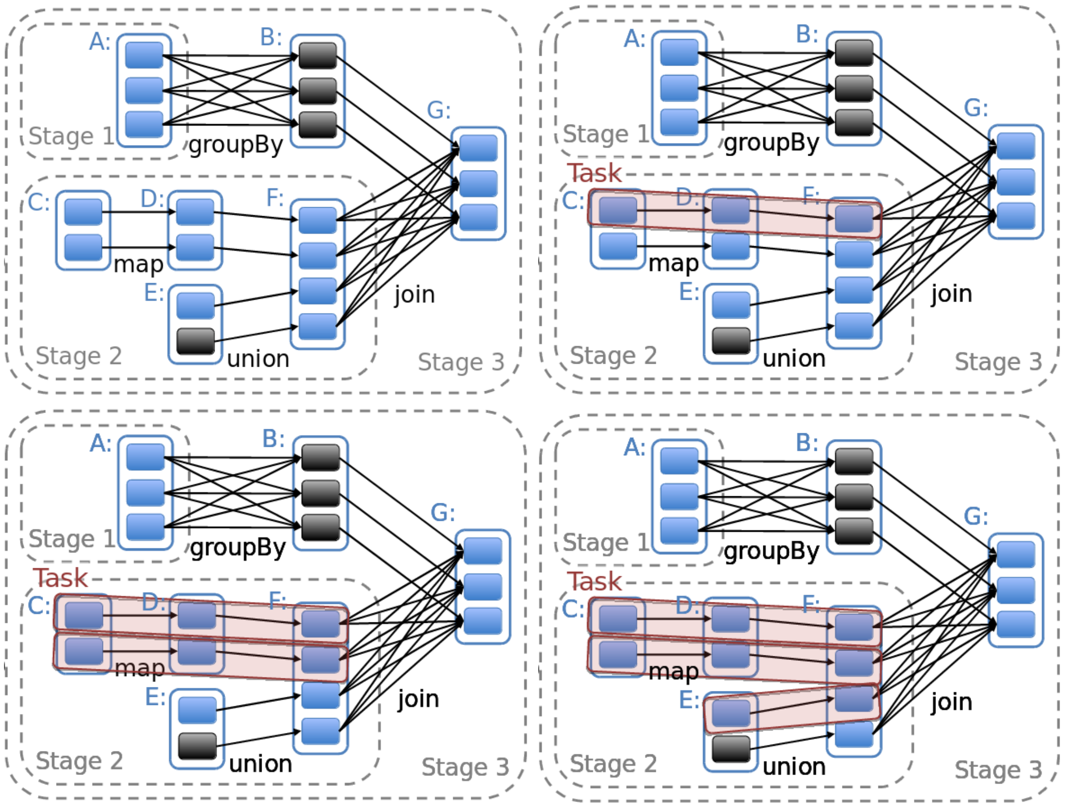


Figure 3.1: [28, Left to right, DAG splits into stages and tasks]

3.2.3 Cluster mode

Spark driver is the *process* where the main method runs. As described earlier, after splitting application DAG into stages then into tasks, the driver schedules tasks for executions. In detail, the Spark driver is first connected to the cluster via `SparkContext` object. More precisely, `SparkContext` can connect to several types of *cluster managers* responsible for resource allocations and coordinating tasks. Second, once connected, Spark acquires worker nodes' *executors*. Executors are workers' processes which are responsible for executing individual tasks and storing data in a given Spark *job*. Finally, the driver sends the serialized application code to executors to run tasks. It is worth mentioning that, if any worker dies or runs slowly, its tasks will be sent to different executors to be processed again.

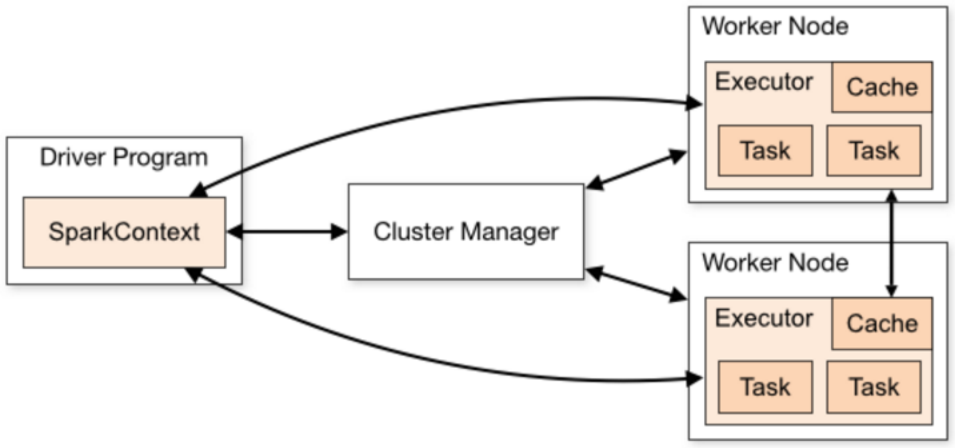


Figure 3.2: [2, Cluster overview]

3.2.4 Overview of Spark memory management

Understanding Spark memory management is critical for getting the best results from an application. We start by specifying that any Spark process that works on a cluster or local machine is a *JVM* process. The user can configure JVM heap size by passing flag options. Spark divides the JVM heap into several parts described as follows

- *Reversed memory*: Spark sets aside 300 MB of memory which will be untouchable and in fact, it sets the limit on what the user can allocate for Spark usage.

- *Spark memory*: This memory pool is directly managed by Spark. It can be modified by the option `spark.memory.fraction`. This part consists of two smaller parts:

  - *Storage memory*: This is used for both storing cached data and for temporary space serialized data. `spark.memory.storageFraction` is controlling by the storage memory with a typical value of 50% or 60% of Spark memory.
  - *Execution memory*: This is used for storing objects required for execution of Spark tasks. When not enough memory is available, it supports spilling to disk. Its size is what is left in `spark.memory.fraction`.

  The size of Spark memory can be calculated as

  $$(\text{Java heap} - \text{Reserved memory}) \times \texttt{spark.memory.fraction}.$$

- *User memory*: This memory pool remains after the allocation of Spark memory and it is left to the user to use it appropriately. Its size is calculated as

  $$(\text{Java heap} - \text{Reserved memory}) \times (1 - \texttt{spark.memory.fraction}).$$

The following illustration summarizes the above explanations in Figure 3.3

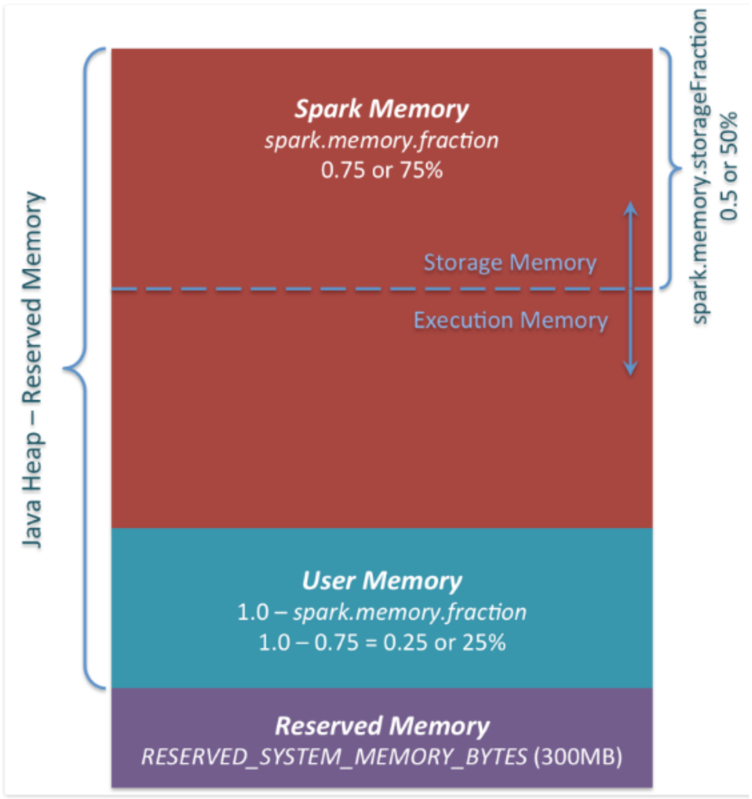


Figure 3.3: [25, Spark 1.6+ memory manamgent]

3.2.5 Advantages and disadvantages

Spark with its distributed memory abstraction (RDD), provides a limited programming interface due to its immutable nature and coarse-grained transformations. But still the creators of Spark have made use of the abstraction for a wide class of applications and made many improvements as of Spark-2.0.0 by adding more fine-grained abstractions which are beyond the scope of our brief descriptions.

It is somewhat surprising that the RDDs have high expressible power over previously defined abstractions. In fact, RDDs can be used to reconstruct the number of cluster programming models with ease [28]. Major applications that have been integrated in Spark are: *SQL*, *MLlib* (machine learning library), *GraphX* (graph processing library) and *Streaming* (batched stream processing library).

Other advantages of Spark are efficient use of data sharing abstraction, avoiding the cost of data replication and leveraging data persistence well enough.

However, some disadvantages of Spark are inevitable high latency (although lower than MapReduce), in particular for applications involving stream of data. The other is for applications that make asynchronous fine-grained updates to shared state, such as an incremental web crawler [28]. From the high performance computing side, the main disadvantage is the speed. Because the mentioned advantages for Big Data applications weigh much higher than only the speed comparing to, for example, MPI [26]. For the record, these are some of the main reasons of slow down in speed: serialization, deserialization, communication costs, bandwidth in the cluster, stragglers (slow workers) and JVM itself.

3.3 Distributed linear algebra and optimization in Apache Spark

Apache Spark as the most well-known open-source, distributed computing platform for Big Data applications, has provided distributed linear algebra and convex optimization supports through one simple assumption that divides vector and matrix operations: vectors are *local* and matrices must be distributed across the cluster. That is, since matrices are quadratically taking more storage space than vectors, then vector operations should be local operations while matrix operations should be distributed and involve the cluster [30]. For example, matrix-matrix operations are done over the cluster and matrix-local vector operations can be efficiently done by broadcasting the local vector to workers containing matrix pieces.

Based on this assumption, there are built-in specially designed distributed linear algebra module (*linalg*) and a distributed convex optimization module (*optimization*) for separable convex objective functions as part of MLlib (Spark's machine learning library) [30], [3].

There have been three main distributed matrix data structures implemented in Spark with sparse and dense supports

- `CoordinatMatrix`: which distributes an RDD of type `MatrixEntry` with underlying data `(i: Long, j: Long, value: Double)` indicating specific position of a value in matrix. It is suitable for very sparse matrix storage and operations.

- `BlockMatrix`: which distributes an RDD of blocks

  ```
  (i: Int, j: Int, mat: Matrix)
  ```

  indicating specific locations of each block in a structured block formated matrix.

- `RowMatrix`: which distributes an RDD of local row vectors. It is suitable for talk-skinny or small-wide matrices and operations involving them.

In addition to distributed matrix data structures, the local vector and matrix supports are provided by the Scala library called Breeze [5]. Breeze supports efficient computations for sparse or dense local vectors or matrices.

3.3.1 Efficient matrix computations

Depending on the underlying data structure for distributed matrices, there are different types of implementations for special matrix operations such as Singular Value Decomposition (SVD), QR decomposition [30] and Gramian matrix computation using DIMSUM [29] in Spark.

For efficiency of computations, it is vital to fully utilize the hardware-specific linear algebraic operations locally on a single node. To do that, Spark uses BLAS (Basic Linear Algebra Subroutines) interface with native *Fortran* speed. Native libraries can be used in Scala through Java Native Interface (JNI) and interoperability of Scala with Java. Netlib-java [17] library provides the JNI and it is wrapped by Breeze library. The variety of BLAS implementations have been tested for *linalg* module and the results were reported in [30].

In addition to native BLAS support, the native Linear Algebra Package (LAPACK) and Arnoldi Package (ARPACK) written in Fortran are available through Netlib-java. Optimized routines from LAPACK and ARPACK have been widely utilized in *linalg* and *optimization* modules in Spark. For more details, see [3].

Chapter 4

Distributed linear programming with Apache Spark

In this chapter, we describe the logic behind our large-scale linear programming solver (Spark-LP) implemented in [20]. We explain in depth our software architecture design and present some of its advantages over other possible choices. Finally, we demonstrate performance results obtained from solving small-scale to large-scale problems in Section 4.2.

4.1 Methodology and software architecture design

As described at the beginning of Chapter 2, the linear programming problem in standard form is given by the Equation 2.1 and for the sake of convenience we reproduce here:

$$\begin{aligned} \min \quad & c^T x \\ \text{subject to} \quad & Ax = b \\ & x \geq 0 \end{aligned}$$

We assume that the real valued constraint matrix A is of dimension $m \times n$, where $m \leq n$ and $\operatorname{rank}(A) = m$.

Our goal is to implement Mehrotra's predictor-corrector algorithm, described in Section 2.2.3, to solve large-scale linear programming problems in a distributed way. To do so, because most of the operations involved in Section 2.2.3 are vector space operations, then the first step is to separate vector space interface and vector space operator interface. These abstractions help us implement operations involving *local* vectors or matrices separately from operations involving *distributed* vectors or matrices.

4.1.1 Local versus distributed design

Vectors

As mentioned in Section 3.3.1, in order to utilize hardware acceleration for local computations, we can use Netlib-java [17] which provides a Native Java Interface (JNI) to directly call BLAS [4] and LAPACK [14] routines written in *Fortran*. This functionality is already provided in Spark. Therefore, as an example, if a, b are two `DenseVectors`, we can define their local inner product $a^T b$ or their linear combinations $\alpha a + \beta b$ with coefficients α, β as follows

```scala
/* local computations: */
import org.apache.spark.mllib.linalg.{BLAS, DenseVector}

def dot(a: DenseVector, b: DenseVector): Double = BLAS.dot(a, b)

def combine(alpha: Double,
            a: DenseVector,
            beta: Double,
            b: DenseVector): DenseVector = {
      val ret = a.copy
      BLAS.scal(alpha, ret)
      BLAS.axpy(beta, b, ret)
      ret
    }
```

Listing 4.1: Excerpts from [20]

As we go from the built-in local vector and matrix data structures in Spark to distributed matrix data structures, the assumption of [30] that all vectors are local but matrices are distributed (as explained in Section 3.3) is no longer sufficient for our purpose. That is, since the objective vector x and its coefficient c do not necessarily fit in a single machine, we need to define our own *distributed vector* data structure.

We notice that there are two viable choices to represent distributed vectors: `RDD[Double]` and `RDD[DenseVector]`. The latter is our choice as it offers some important advantages over the former (also indicated in [21]), most importantly is the computation speed.

Because partitions of an `RDD[DenseVector]` consist of some number of local `DenseVectors`, operations involving `RDD[DenseVector]` can be executed much faster than the normal choice `RDD[Double]` by exploiting BLAS operations locally. For example, following up our previous local example, the inner product of two `RDD[DenseVector]` a, b with *equal number of partitions* and their linear combinations can be computed by using the previously defined `dot` and `combine` over partitions and then aggregating the results. The aggregation can be done via the `aggregate` function as follows (for simplicity we use `DVector` as type alias for `RDD[DenseVector]`)

```
/* distributed computations: */
def dot(a: DVector, b: DVector): Double = {
      a.zip(b).aggregate(0.0)(
        seqOp = (acc: Double, ab: (DenseVector, DenseVector)) => {
          acc + DenseVectorSpace.dot(ab._1, ab._2)
        },
        combOp = (acc1: Double, acc2: Double) => acc1 + acc2
      )
}

def combine(alpha: Double, a: DVector, beta: Double, b: DVector):
    DVector =
      if (alpha == 1.0 && beta == 1.0) {
        a.zip(b).map {
          case (aPart, bPart) => {
            BLAS.axpy(1.0, aPart, bPart) // bPart += aPart
            bPart
          }
        }
      } else {
        a.zip(b).map {
          case (aPart, bPart) =>
            DenseVectorSpace.combine(alpha, aPart, beta, bPart).
  toDense
        }
      }
```

Listing 4.2: Excerpts from [20]

Spark's great versatility lets us easily go from local to distributed computations as shown above. Since the aggregate efficiency is critical for us, as an optimization to our `aggregate` function we can use `treeAggregate` function with pre-specified *depth* to compute the aggregated results more efficiently than `aggregate` as depicted below in

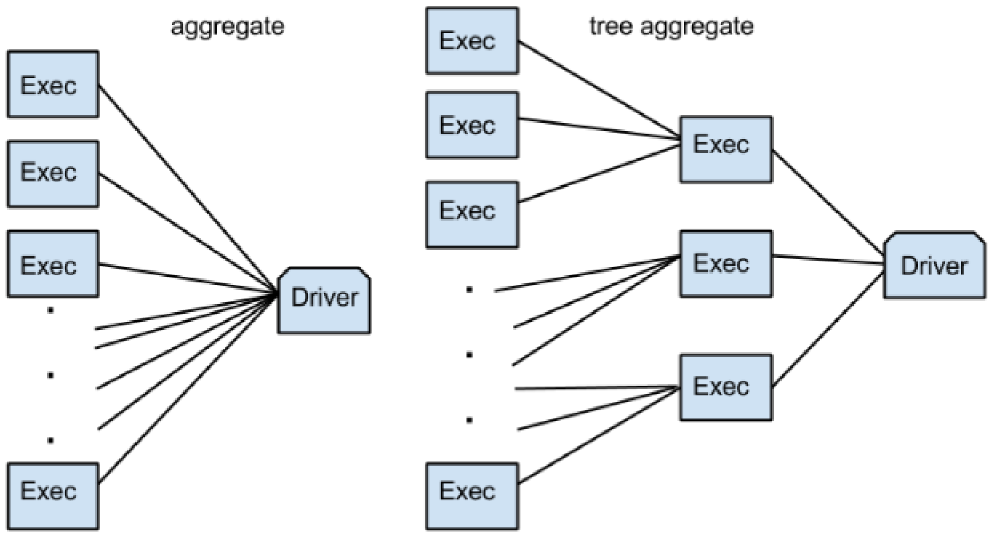


Figure 4.1: [7, spark-1-1-mllib-performance-improvements]

The main difference is that `treeAggregate` need not wait for all the results to be computed and sent back to driver. Instead it aggregates the partial results when ready, up to the final result. Therefore, the optimized distributed inner product is as follows

```scala
def dot(a: DVector, b: DVector, depth: Int = 2): Double =
    a.zip(b).treeAggregate(0.0)((sum, x) => sum + BLAS.dot(x._1, x._2), _ + _, depth)
```

Listing 4.3: Excerpts from [20]

Matrices

In all interesting linear programming problems that the number of unknowns exceeds the number of equations, we notice if we assume that the number of rows in A of Equation 2.1 is small, then columns of A can be considered as local (sparse or dense) vectors. Thus, we can choose `RDD[Vector]` (with type alias `DMatrix`) as our distributed matrix data type. However, we can leverage `RowMatrix` in Section 3.3 with underlying distributed data type to our advantage and instead of working with A, we can consider its transpose $B = A^T$ and exploit the implemented functionalities greatly. For our purpose, we have modified `RowMatrix` to `LPRowMatrix` in [20] to compute the *Gramian* matrix with less overhead for our computations. The Gramian matrix computation such as $B^T B = AA^T$ for initialization, described in Section 2.2.2 and $B^T D^2 B = (DB)^T (DB)$ in Mehrotra's algorithm, described in Section 2.2.3, is critical for us. Based on our earlier assumption, the result of $B^T B$ is a local symmetric positive definite matrix, therefore, we can compute its inverse with two LAPACK routines [14] described below as follows

1. `dpptrf`: computes the *Cholesky decomposition* of a symmetric positive definite matrix in packed format

2. `dpptri`: computes the *inverse* of a symmetric positive definite matrix from its Cholesky decomposition computed by `dpptrf`

```scala
import com.github.fommil.netlib.LAPACK.{getInstance => lapack}
import org.netlib.util.intW

def posSymDefInv(B: Array[Double], m: Int): Unit = {
    val info1 = new intW(0)
    lapack.dpptrf("U", m, B, info1)
    val code1 = info1.`val`
    assert(code1 == 0, s"lapack. returned $code1.")
    val info2 = new intW(0)
    lapack.dpptri("U", m, B, info2)
    val code2 = info2.`val`
    assert(code2 == 0, s"lapack. returned $code2.")
 }
```

Listing 4.4: Excerpts from [20]

Moreover, to solve

$$(B^T D^2 B)\Delta\lambda = -r_b + B^T D^2(-r_c + s + X^{-1}\Delta X^{\text{aff}}\Delta S^{\text{aff}} e - \sigma\mu X^{-1} e)$$

in Section 2.2.3 via Cholesky decomposition, we have used the LAPACK `dppsv` routine designed for solving symmetric positive definite linear system of equations in-place. Fortunately, this functionality exists in Spark *linalg* module.

Note that there are other possible choices for distributed matrix data structures, such as `BlockMatrix`, as introduced in Section 3.3, which is only suitable if *a priori*, we know that B is composed of block matrices. That is the reason we have decided to use the more general data structure.

Matrix-matrix product

As mentioned above, the only distributed matrix-matrix product that we care about is the Gramian matrix computation in both the initialization Section 2.2.2 and the main algorithm Section 2.2.3. Since we represent B as an `RDD[Vector]` where rows are local vectors, we need to be careful about the maximum size of one-dimensional `Vector` to avoid overflow in our computation. Unfortunately, the underlying array of numbers in Scala/Java is limited to 32 bits in length. As we will explain more, that results into having the maximum number of 65535 columns in B

(rows in A). A priori, there is no limitations for the number of rows in B (columns in A) and it all depends on the number of available memory in our cluster.

The Gramian matrix is computed via the BLAS routine `spr` which adds the outer product vv^T of a local vector row v in each partition inplace locally and the results are aggregated from the executors with `treeAggregate`. Formally,

$$B^T B = \sum_{i=1}^{n} v_i v_i^T$$

where v_i's are local row vectors in B as `RDD[Vector]`.

Matrix-vector product

The matrix-vector product captures most of the computations in Section 2.2.3, depending on the output data type, we need to take care of three different cases. Note that for simplicity, we use `DVector` and `DMatrix` as type aliases for `RDD[DenseVector]` and `RDD[Vector]`, respectively. Moreover, for all the operations we require the number of partitions in `DMatrix` and `DVector` be *equal*. The details are as follows:

- `DMatrix` $\times$ `Vector` = `DVector`

  This is the case where a distributed matrix is multiplied to a local vector and the result is a distributed vector. We can take advantage of the local vector and *broadcast* it to all the executors where each contains part of `DMatrix`. Therefore, all the inner product computations happen locally within executors and no further communication is required.

  For more details, see `LinopMatrix.scala` in [20].

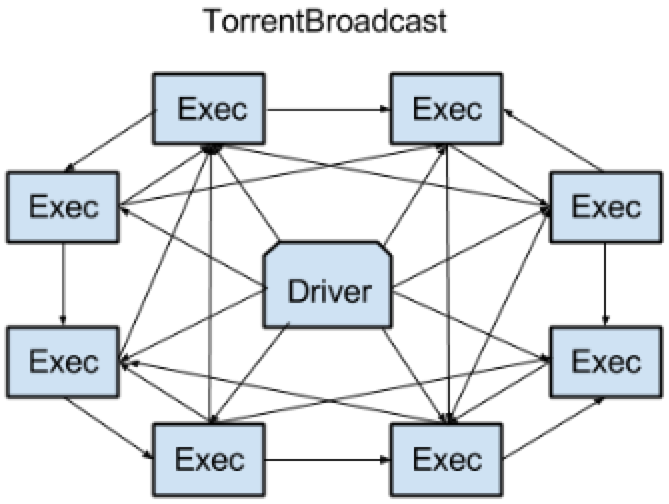


Figure 4.2: [7, spark-1-1-mllib-performance-improvements]

- `DMatrix` $\times$ `DVector` = `Vector`

  This is the case where the adjoint of a distributed matrix is multiplied to a distributed vector (all must have the equal number of partitions) and outputs a local vector. The implementation is done via `zipPartitions` which aligns corresponding partitions of `DMatrix` and `DVector` together. Then, by an *iterator* partial multiplications are computed and the final sum is done via `treeAggregate`. For more details, see `LinopMatrixAdjoint.scala` in [20].

- `DVector` $\times$ `DMatrix` = `DMatrix`

  This very special case happens when a diagonal matrix D in Section 2.2.3 which is represented as a `DVector` containing its diagonal elements, is multiplied to a distributed matrix B and the result DB is a distributed matrix that will be later used in Gramian matrix computation $(DB)^T DB$ as described in Section 4.1.1. Given that D and B have equal number of partitions, after using `zipPartitions` to align corresponding partitions in one place, then elements of each partition of D is aligned with elements of each partition of B via `checkedZip` and an element of the result is a `Double` precision number matched by a `Vector`, therefore, the scalar-vector multiplication is done by

BLAS `scal` routine. Similar to other distributed computations, iterators play a significant role here. For more details, see `SpLinopMatrix.scala` in [20].

4.1.2 Unified interface

With described different implementations for local and distributed vectors and matrices, having a unified interface is very convenient for further development. This unification can be obtained by the use of Scala *implicits* [19]. That is, through implicits we can manage different implementations of local and distributed cases uniformly. Our choice directly uses Spark-TFOCS design [21] by separating local and distributed implementations into different *name spaces* and adding `package object` modifier so that they can be visible to all members of our solver [20] package. For more details, see [20].

4.2 Benchmark results

In this section, we present our benchmark results both for small problems that fit in a single machine in Section 4.2.1, and the large-scale results obtained from cluster of machines in Section 4.2.2.

4.2.1 Local mode

Spark is designed for large-scale applications and its main advantage is seen when the application utilizes the complete parallelization power of a cluster of machines. Therefore, our main purpose is to test our linear programming solver [20] for large-scale problems, however, we have decided to include the local results on small problems essentially for two reasons:

- Local tests are important for detecting bugs in our program prior to launching it over a cluster.
- Local tests help to some extent in finding computational bottlenecks for our main application.

Historically, linear programming problems are presented in *Mathematical Programming System* (MPS) file formats. Netlib [18] provides linear programming problems at various levels of difficulties in compressed MPS formats. With the help of [12] and [21] we parsed some of Netlib small problems in MPS format and transformed them into the standard linear programming format Equation 2.1 suitable for our solver.

We have compared the results to the following solvers:

- GLPK [10] which is an open-source linear programming solver written in C language and provides the baseline results.
- Spark-TFOCS package [21] which is an optimization package imitating Matlab TFOCS (Templates for First-Order Conic Solvers) library on top of Spark and has (ℓ_2 regularized) smoothed standard linear programming solver support.

Our local hardware specifications are: *Intel-Core i7-4810MQ CPU-2.80GHz* $\times$ *8* and 8GB *RAM*. As for the software specifications, we have used *Scala 2.10.4*, *Java 1.8*, *Ubuntu 14.04 operating system*, *OpenBLAS enabled* with the following *Garbage Collection* (GC) flags

- `-XX:MaxMetaspaceSize=3G`
- `-XX:+UseCompressedOops` to compress the object pointers to 4-bytes
- `-XX:ConcGCThreads=8`
- `-XX:+UseConcMarkSweepGC`

Table 4.1 contains the name of the Netlib problems, number of non-zeros (nn) and execution time measured in *seconds*. m, n are dimensions of problems and p is the number of partitions. Also, *netlib sol* denotes the solution provided by Netlib and *rel error* is the relative error of the “official” solution and the solution found by the solver.

solver	problem name	nn	m	n	p	nc	netlib sol	rel error	time ratio
spark-lp	AFIRO	88	27	33	1	1	-464.75314286	3.24e-9	× 486
					2	2		3.24e-9	× 490
spark-tfocs					1	1		1.35e-8	× 17393
					2	2		4.61e-8	× 18813
GLPK					1	1		0	× 1
spark-lp	AGG2	4515	517	302	1	1	-20239252.356	2.21e-11	× 111
					2	2		2.67e-11	× 231
GLPK					1	1		0	× 1
spark-lp	SCFXM2	5229	661	914	1	1	36660.261565	4.20e-12	× 68
					2	2		4.52e-12	× 154
GLPK					1	1		0	× 1
spark-lp	BNL2	16124	2325	3489	1	1	1811.2365404	3.49e-10	× 150
					2	2		3.47e-10	× 156
GLPK					1	1		0	× 1
spark-lp	MAROS-R7	151120	3137	9408	1	1	1497185.1665	2.06e-11	× 39
					2	2		2.06e-11	× 40
GLPK					1	1		0	× 1

Table 4.1: Local result

From Table 4.1, it is evident that Spark-TFOCS (smoothed) linear programming solver is very slow, even for small problems and clearly is not an option for large-scale linear programming problems. We did not include the Spark-TFOCS results for slightly bigger problems because of the very large execution time exceeding the 500 seconds threshold.

As expected, our Spark-LP solver [20] is much slower when solving small problems than GLPK. The reasons are *not only* the clear speed differences between Scala/Java and C languages, but also the inevitable overheads that Spark has for distributed computations versus sequential ones. One clear observation from the local results is that as problems are becoming bigger the execution time ratio between Spark-LP and GLPK is *decreasing*. That is mainly happening because the computations themselves are taking longer thus the overheads slowdown effects are becoming smaller and for large-scale problems (if tuned properly) they will become negligible.

The following are Spark's inevitable computational overheads for distributed large-scale computations that have direct effects on our Spark-LP solver being slower than GLPK on small problems.

- *Serialization* of lambdas to be distributed via the network to the executors
- *Deserialization* of lambdas inside executors for running tasks
- *Scheduler delay*
- *Garbage Collection (GC)*
- *Network bandwidth*

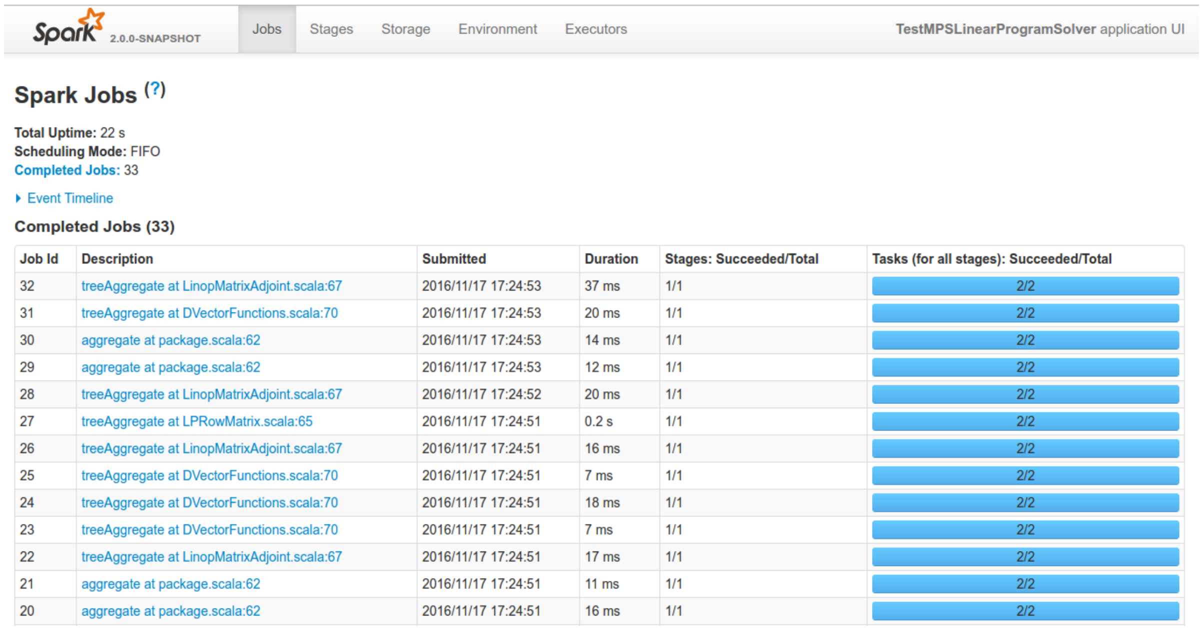
Spark offers many configuration parameters for tuning an application, such as the choice of fast serialization with `Kryo` [13] and specifying GC flags. One of the best ways for choosing a suitable set of parameters is via Spark-UI. It is a common practice to use Spark-UI prior to launching an application to a cluster that we will explain in Section 4.2.1. In fact, the above local results Table 4.1 were obtained by the following additional configurations

- `"spark.executor.extraJavaOptions"` was set to `-XX:NewRatio=6`.
- `"spark.locality.wait"` was set to 0. That means, all tasks must run locally and there is no wait for other less local options.

For the complete set of configuration settings, please see [2].

Spark-UI

Spark-UI is a tool which visualizes some of the most important information of runtime. For example, it shows the specifications of machines that are running tasks, the number of stages that were completed and are still running. We can focus on a particular stage or task and see the underlying DAG of operations, the execution time, serialization and de-serialization time, scheduler delay and the GC time. As an example, upon solving `BNL2.mps` as noted before in Table 4.1, we can see



Spark Jobs (?)

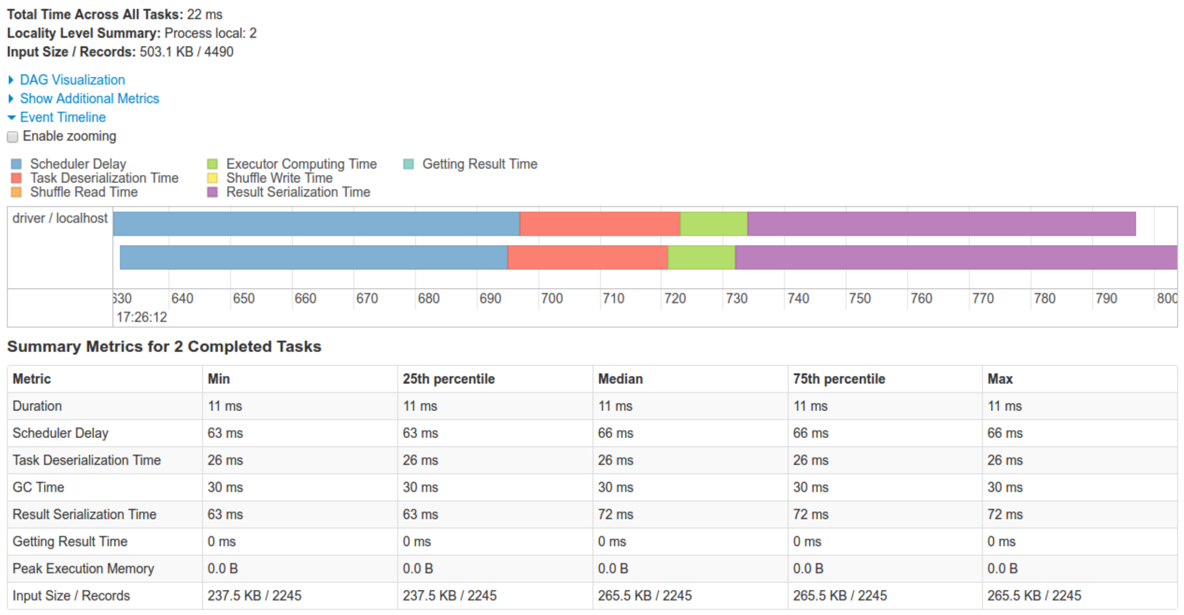
Total Uptime: 22 s
Scheduling Mode: FIFO
Completed Jobs: 33

Completed Jobs (33)

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
32	treeAggregate at LinopMatrixAdjoint.scala:67	2016/11/17 17:24:53	37 ms	1/1	2/2
31	treeAggregate at DVectorFunctions.scala:70	2016/11/17 17:24:53	20 ms	1/1	2/2
30	aggregate at package.scala:62	2016/11/17 17:24:53	14 ms	1/1	2/2
29	aggregate at package.scala:62	2016/11/17 17:24:53	12 ms	1/1	2/2
28	treeAggregate at LinopMatrixAdjoint.scala:67	2016/11/17 17:24:52	20 ms	1/1	2/2
27	treeAggregate at LPRowMatrix.scala:65	2016/11/17 17:24:51	0.2 s	1/1	2/2
26	treeAggregate at LinopMatrixAdjoint.scala:67	2016/11/17 17:24:51	16 ms	1/1	2/2
25	treeAggregate at DVectorFunctions.scala:70	2016/11/17 17:24:51	7 ms	1/1	2/2
24	treeAggregate at DVectorFunctions.scala:70	2016/11/17 17:24:51	18 ms	1/1	2/2
23	treeAggregate at DVectorFunctions.scala:70	2016/11/17 17:24:51	7 ms	1/1	2/2
22	treeAggregate at LinopMatrixAdjoint.scala:67	2016/11/17 17:24:51	17 ms	1/1	2/2
21	aggregate at package.scala:62	2016/11/17 17:24:51	11 ms	1/1	2/2
20	aggregate at package.scala:62	2016/11/17 17:24:51	16 ms	1/1	2/2

Figure 4.3: Spark-UI example 1

and more specific information about a completed task



Total Time Across All Tasks: 22 ms
Locality Level Summary: Process local: 2
Input Size / Records: 503.1 KB / 4490

Summary Metrics for 2 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	11 ms	11 ms	11 ms	11 ms	11 ms
Scheduler Delay	63 ms	63 ms	66 ms	66 ms	66 ms
Task Deserialization Time	23 ms	26 ms	26 ms	26 ms	26 ms
GC Time	30 ms	30 ms	30 ms	30 ms	30 ms
Result Serialization Time	63 ms	63 ms	72 ms	72 ms	72 ms
Getting Result Time	0 ms	0 ms	0 ms	0 ms	0 ms
Peak Execution Memory	0.0 B	0.0 B	0.0 B	0.0 B	0.0 B
Input Size / Records	237.5 KB / 2245	237.5 KB / 2245	265.5 KB / 2245	265.5 KB / 2245	265.5 KB / 2245

Figure 4.4: Spark-UI example 2

As seen, when the number of tasks is much bigger than the number of available cores, the scheduler delay becomes one of the major bottlenecks. Therefore, for large-scale applications we have to make sure that the ratio between the number of tasks and the available cores is small enough.

4.2.2 Cluster mode

Here we will present the results for large-scale problems. The main difficulty in measuring the execution time of our solver for large-scale linear programming problems is finding the problems that meet the specification for the number of equations that our solver can handle.

As explained in Section 4.1.1, due to the fact that the core Java only supports arrays of 32-bits long, the maximum number of columns in B is limited to 65535. Therefore, we needed to generate our own large-scale linear programming problems from Gaussian distribution $\mathscr{N}(\mu, \sigma)$ with mean μ and standard deviation σ. The problem creation is controlled by the number of rows n, columns m and sparsity density $\delta \in (0,1]$ parameters.

For the cluster resources, we have used [7, Databricks cloud] which uses Amazon's Elastic Computing instances with pre-installed and configured Apache Spark-1.6.1 machines where each machine is *memory optimized* (`r3.xlarge`) with 4 cores, 30 GB RAM and *moderate* network bandwidth. Note that it is technically possible to use workers having more cores, RAM and much higher bandwidths but at the higher costs. However, at the time of writing this thesis, the `r3.xlarge` instances offer the least cost. Also, the number of executors in our clusters varies between $16, 32$ and 64.

To obtain the results, we have tested different values for n, m, δ and varied the number of partitions p to obtain the complete parallelization of the tasks.

The common practice for determining the number of partitions to obtain the complete parallelization is 2-3 times the total number of available cores in cluster. However, we found out that for our application where distributed matrix by distributed matrix multiplication has the largest overhead among other parts, even when re-used distributed matrices and vector are cached in memory, if the number of partitions is high then communications become a huge bottleneck.

We have measured the total execution times as well as other parameters for the best possible number of partitions by trial and error. Table 4.2 shows the results

$n^\star$	$m^\star$	δ	p	cores	creation (sec)	init (sec)	avg iter (sec)	total (sec)
5	3	0.01	8	64	13.0	6.2	2.0	56.3
5	3	0.01	16	64	29.5	2.1	2.1	50.7
5	3	0.1	8	64	10.8	7.0	5.1	136.0
5	3	0.1	16	64	21.1	3.4	1.8	44.6
6	3	0.01	32	128	37.1	4.0	4.0	205.0
6	3	0.1	32	128	38.9	6.2	5.9	262.4
7	3	0.01	64	128	43.2	8.1	19.8	1871.5
7	3	0.1	64	128	33.0	15.8	29.1	2465.6
8	3	0.01	1024	256	101.1	60.3	200.9	exceeds
5	4	0.01	128	128	37.6	260.2	260.0	7021.7
5	4	0.1	128	128	37.9	314.7	326.0	8465.3
6	4	0.01	256	256	77.4	312.3	312.1	9677.2
6	4	0.1	256	256	76.8	398.9	422.1	13484.5

Table 4.2: Distributed results: $n^\star, m^\star$ are logarithm values of n, m respectively, δ is the sparsity density, p is the number of partitions

We required that all convergence measurements, mentioned in Section 2.2.3, must fall under the tolerance of 10^{-8} and at most the maximum of 100 iterations limit. In our measurements, the case with $n = 10^8, m = 10^3$ exceeds the 100 iterations.

For simplicity, we denote each record in Table 4.2 by $n^\star$-$m^\star$-δ-p_no. machines. As an example, 5-3-0.1-16_64m denotes the generated linear programming problem with $n = 10^5, m = 10^3$, sparsity density $\delta = 0.1$, number of partitions $p = 16$ and 64 machines where one is a master and other 63 are workers.

Figure 4.5 illustrates the performance comparisons of the best found results

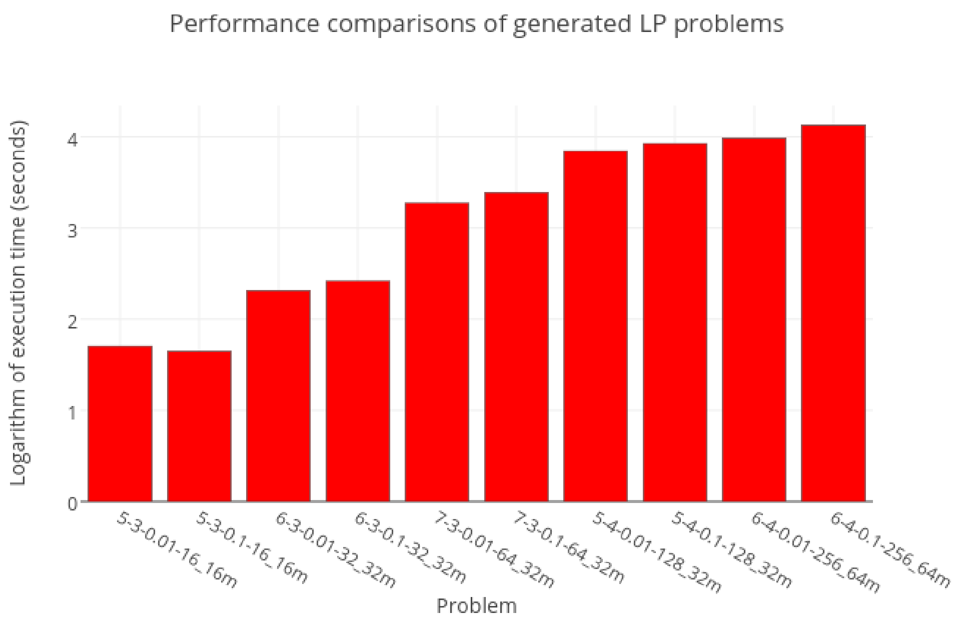


Figure 4.5: Distributed performance results

Figure 4.5 shows that increasing m affects the performance more than increasing n. This is due to two reasons:

1. Increasing m makes row vectors longer. Therefore, that restricts each worker to store more number of row vectors. Hence, local computations take longer.

2. The number of partitions and cores (machines) must increase (in most cases by a factor of 4) thus, the communications take much longer.

The same argument justifies the increase in average time per iteration when m becomes larger, shown in Figure 4.6

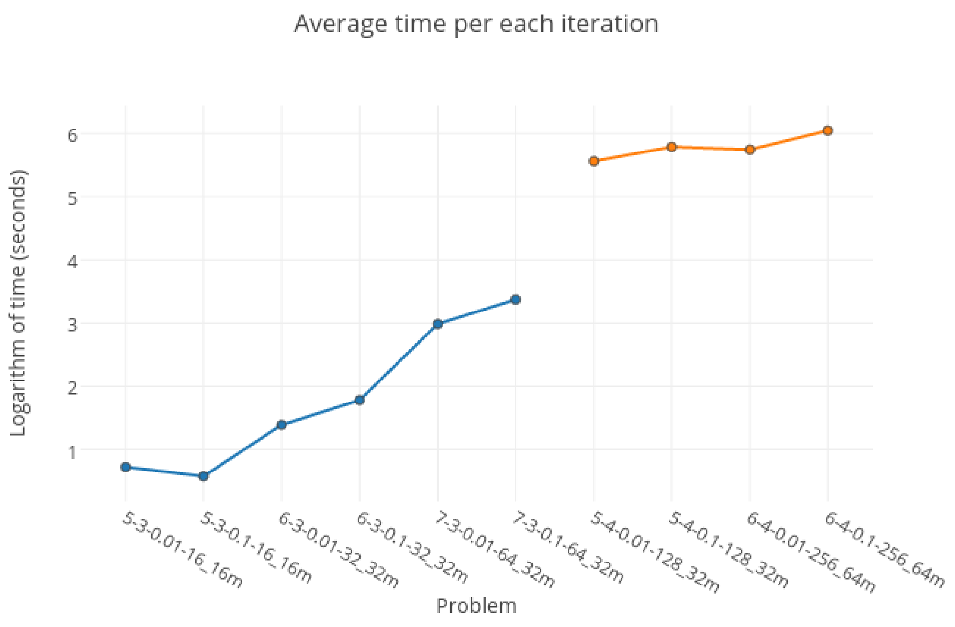


Figure 4.6: Average iteration time

We have also observed that as n becomes larger the convergence rate drops for large-scale problems as opposed to small-scale problems as shown in Figure 4.7.

For further tuning, when visualizing the stages with Spark-UI, we used *local-checkpoints* that sacrifices some fault-tolerance features for increasing the speed. In fact, it shortens the lineage after every iteration and makes the application DAG a lot less complicated. For example, it took 3 seconds to compute Gramian matrix of size 300 GB stored in memory of 32 machines for the case with $n = 10^7, m = 10^3, \delta = 0.1, p = 64$.

In next chapter, we will describe a number of important suggestions for further developments.

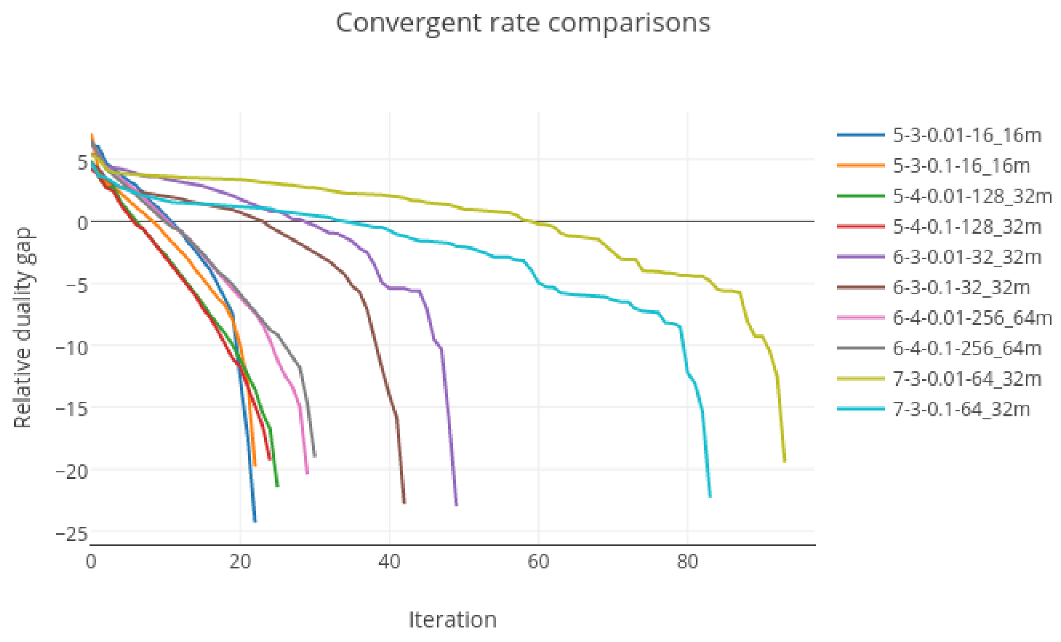


Figure 4.7: Distributed convergent results

Chapter 5

Conclusions

The goal in this thesis project was to investigate Apache Spark's capabilities and potential for scientific computing and high performance computing tasks and assess its advantages and disadvantages. For this, we have developed a *large-scale* distributed linear programming solver, Spark-LP [20], on top of Apache Spark optimization module. Spark-LP supports efficient *sparse and dense* linear algebra computations, it is *open-source*, *fault-tolerant* and can be used on *commodity clusters of machines*.

The software architecture design was explained in depth and results were provided in Section 4.2.

However, we have faced a number of structural and performance limitations that we try to address in the following lists of recommendations:

- Our solver [20] only accepts linear programming problems in the standard format Equation 2.1. One possible extension is to add a *preprocessing* unit so that more general formats can be supported by transforming them into the standard format. However, that will add more overheads to the computations which we tried to avoid as much as possible until now that we have promising results for the standard format case.

- Moreover, having a *feasibility detection* unit is necessary for making it a mature large-scale solver.

- Furthermore, because of the somewhat similar nature of quadratic programming it is also feasible to generalize our solver for such problems.

- The results of convergent rate shows that Mehrotra's predictor-corrector algorithm, described in Section 2.2.3, for large-scale problems is slower than when it is used for small-scale problems. Therefore, finding an optimal algorithm for large-scale problems with our relatively strict settings is an important challenge for the future work.

- To resolve the limitations on the number of columns , described in Section 4.1.1, it is possible to extend core Java 32-bits arrays to 64-bits with [9, fastUtil] library. But that will result in more inevitable changes in the underlying Spark optimization module.

- Last but not least, we have considered the use of Graphical Processing Units (GPU) to have expected faster local linear algebra computations in the cluster. In fact, Spark can work with [6, CUDA]-enabled GPUs. More specifically, one can use [1, Apache Mesos] as the resource manager to emulate NVIDIA [8, docker] on the Spark cluster.

  Currently, the best data structure designed for similar use cases is provided in [16, ND4J] library. It is called `INDArray` and it interplays nicely with Apache Spark, when having a heterogeneous cluster of multiple CPUs and GPUs. Therefore, one could use `RDD[INDArray]` as the underlying distributed structure for such clusters. Note also that `INDArray`s are 64-bits in length, then it automatically solves the limitations on the number of columns described in Section 4.1.1.

  ND4J uses vectorized C++ code through [11, JavaCpp] library for all numerical operations. It handles the stored data off-heap (with its own GC implementation) and it utilizes the optimized native BLAS libraries. The JavaCpp library plays a crucial here as it automatically generates the required JNI with *zero overhead* to access all native codes. However, ND4J is not yet mature enough for high performance scientific computing use cases and further developments are required to add more BLAS and LAPACK supports.

Bibliography

[1] Apache Mesos: A distributed systems kernel. *http://mesos.apache.org/*. December 2016.

[2] Apache Spark: Lightening-fast cluster computing. *http://spark.apache.org/*. December 2016.

[3] Apache Spark Source Code. *http://github.com/apache/spark*.

[4] BLAS: Basic Linear Algebra Subprograms. *http://www.netlib.org/blas/*. November 2016.

[5] Breeze: Numerical Processing Library for Scala. *http://github.com/scalanlp/breeze*.

[6] CUDA: Parallel Computing Platform and Programming Model for GPU. *https://developer.nvidia.com/cuda-zone*. December 2016.

[7] Databricks. *https://databricks.com*. December 2016.

[8] Docker: Software Containerization Platform. *https://www.docker.com/*. December 2016.

[9] FastUtil: Fast and Compact Type Specific Collections for Java. *http://fastutil.di.unimi.it/*. December 2016.

[10] GLPK: GNU Linear Programming Kit. *https://www.gnu.org/software/glpk/*. December 2016.

[11] JavaCpp: The missing bridge between Java and native C++. *https://github.com/bytedeco/javacpp*. December 2016.

[12] JOptimizer. *http://www.joptimizer.com/*. November 2016.

[13] Kryo serialization: Java serialization and cloning (fast, efficient, automatic). *https://github.com/EsotericSoftware/kryo*. December 2016.

[14] LAPACK: Linear Algebra PACKage. *http://www.netlib.org/lapack/*. November 2016.

[15] MPI: A Message-Passing Interface Standard. *http://mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf*. December 2016.

[16] ND4J: N-dimensional arrays and scientific computing for the JVM. *http://nd4j.org/*. December 2016.

[17] Netlib-java: High Performance Linear Algebra (low level). *http://github.com/fommil/netlib-java*.

[18] Netlib Linear Programming Data. *http://www.netlib.org/lp/data*. November 2016.

[19] Scala Programming Language. *http://scala-lang.org*.

[20] Spark-LP. *https://github.com/ehsanmok/spark-lp*. December 2016.

[21] Spark-TFOCS: A Spark port of Templates for First-Order Conic Solvers. *http://github.com/databricks/spark-tfocs*.

[22] T. Terlaky C. Roos and J.-Ph. Vial. *Theory and Algorithms for Linear Optimization, An Interior Point Approach*. John Wiley and Sons, Chichester, UK, 1997.

[23] J. Dean and S. Ghemawat. *MapReduce: Simplified Data Processing on Large Clusters*. *Communications of ACM*, 2008.

[24] G. Farkas. *A Fourier-fele mechanikai elv alkalmazasai (in Hungarian)*. pages 457–472, 1894.

[25] Alexey Grishchenko. *Distributed Systems Architecture*. *https://0x0fff.com/spark-memory-management/*.

[26] D. Anguita J. L. Reyes-Ortiz, L. Oneto. *Big Data Analytics in the Cloud: Spark on Hadoop vs MPI/OpenMP on Beowulf*. *Procedia Computer Science, Volume 53*, pages 121–130, 2015.

[27] S. Wright J. Nocedal. *Numerical optimization*. Springer, second edition, 2006.

[28] T. Das A. Dave J. Ma M. McCauley M. J. Franklin S. Shenker I. Stoica M. Zaharia, M. Chowdhury. *Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing*. 2010.

[29] A. Goel R. B. Zadeh. *Dimension Independent Similarity Computation. The Journal of Machine Learning Research*, pages 1605–1626, 2013.

[30] A. Staple B. Yavuz L. Pu S. Venkataraman E. Sparks A. Ulanov M. Zaharia R. B. Zadeh, X. Meng. *Matrix Computations and Optimization in Apache Spark.* 2016.

[31] M. Zaharia PhD Thesis. *An Architecture for Fast and General Data Processing on Large Clusters. Electrical Engineering and Computer Sciences University of California at Berkeley*, 2014.

[32] R.J. Vanderbei. *Linear Programming: Foundations and Extensions.* Kluwer Academic Publishers, 1997.

[33] S. Wright. *Primal-dual interior point methods.* Society for industrial and applied mathematics, 1987.