University of Novi Sad
Faculty of Sciences
Department of
Mathematics and Informatics

Implementation and analysis of a class of algorithms for distributed convex optimization: Performance evaluation and tradeoffs in practical HPC clusters

-PhD Thesis-

Implementacija i analiza klase algoritama za distribuiranu konveksnu optimizaciju: Evaluacija performansi i osobina na praktičnim HPC klasterima

-Doktorska disertacija-

Mentors:

dr Dušan Jakovetić

dr Danijela Boberić Krstićev

Candidate:

Lidija Fodor

Novi Sad, 2022

УНИВЕРЗИТЕТ У НОВОМ САДУ **ОБРАЗАЦ – 5а**
ПРИРОДНО-МАТЕМАТИЧКИ ФАКУЛТЕТ

КЉУЧНА ДОКУМЕНТАЦИЈСКА ИНФОРМАЦИЈА[1]

Врста рада:	Докторска дисертација
Име и презиме аутора:	Лидија Фодор
Ментор (титула, име, презиме, звање, институција)	др Душан Јаковетић, ванредни професор, Природно-математички факултет, Универзитет у Новом Саду др Данијела Боберић Крстићев, ванредни професор, Природно-математички факултет, Универзитет у Новом Саду
Наслов рада:	Имплементација и анализа класе алгоритама за дистрибуирану конвексну оптимизацију: Евалуација перформанси и особина на практичним HPC кластерима
Језик публикације (писмо):	Енглески језик
Физички опис рада:	Страница 225 Поглавља 4 Референци 163 Табела 19 Слика 52 Графикона 0 Прилога 2
Научна област:	Информатика
Ужа научна област (научна дисциплина):	Рачунарске науке
Кључне речи / предметна одредница:	Рачунарство високих перформанси, Дистрибуирана оптимизација, ADMM, Евалуација перформанси
Резиме на српском језику:	Значај алгоритама дистрибуиране оптимизације манифестује се кроз растуће интересовање у различитим доменима примене. Примењује се у аналитици великих података, дистрибуираном машинском учењу, дистрибуираној контроли, мрежама возила и паметним мрежама, између осталог. Ова теза се фокусира на примарне и дуалне дистрибуиране методе конвексне оптимизације. Најпре се уводи општи алгоритамски оквир метода првог и другог реда примарног типа, које користе концепт

[1] Аутор докторске дисертације потписао је и приложио следеће Обрасце:
5б – Изјава о ауторству;
5в – Изјава о истоветности штампане и електронске верзије и о личним подацима;
5г – Изјава о коришћењу.
Ове Изјаве се чувају на факултету у штампаном и електронском облику и не кориче се са тезом.

	спарсификоване комуникације и израчунавања преко повезаног графа чворова. Поред неколико већ постојећих метода, појављују се и нове варијанте које користе једносмерну комуникацију. Иако у овој области постоји изузетно велика количина теорије и теоријског напретка, практичне евалуације метода над стварним подацима и практичним системима рачунара вискох перформанси (High Performance Computing — HPC) великих размера, су много мањег обима. Стога смо развили имплементације и извршили скуп различитих нумеричких евалуација предложених метода у стварном, паралелном програмском окружењу. Имплементације су развијене коришћењем технологије Message Passing Interface (MPI) и тестиране су на рачунарском кластеру високих перформанси. Ове емпиријске процене резултирају веома корисним увидима и смерницама у вези са перформансама и наглашавају најважније компромисе између комуникације и израчунавања, у стварном окружењу извршавања. С обзиром на постојање широког скупа алгоритама машинског учења који се могу посматрати као оптимизациони проблеми, дистрибуирана оптимизација има врло значајну улогу у овој области. У тези је такође представљен и алгоритам за конвексно кластеровање, заснован на дуалној методи Alternating Direction Method of Multipliers (ADMM), која се ослања на COMPS Superscalar (COMPSs) приступ за паралелизацију. Приказујемо резултате опсежних нумеричких евалуација алгоритма на HPC рачунарском кластеру, како бисмо демонстрирали висок степен скалабилности и ефикасности методе, у поређењу са постојећим алтернативним приступима за конвексно кластеровање. Програмски код за развијене алгоритме је софтвер отвореног кода, и доступан је у одговарајућим репозиторијумима.
Датум прихватања теме од стране надлежног већа:	30.09.2021. (датум прихватања од стране сената)
Датум одбране: (Попуњава одговарајућа служба)	
Чланови комисије: (титула, име, презиме, звање, институција)	Председник: др Милош Стојаковић, редовни професор, Природно-математички факултет, Универзитет у Новом Саду Члан: др Наташа Крејић, редовни професор, Природно-математички факултет, Универзитет у Новом Саду Члан: др Милош Ивановић, ванредни професор, Природно-математички факултет, Универзитет у Крагујевцу Ментор: др Душан Јаковетић, ванредни професор, Природно-математички факултет, Универзитет у Новом Саду Ментор: др Данијела Боберић Крстићев, ванредни професор, Природно-математички факултет, Универзитет у Новом Саду
Напомена:	

UNIVERSITY OF NOVI SAD
FACULTY OF SCIENCES

KEY WORD DOCUMENTATION[2]

Document type:	Doctoral dissertation
Author:	Lidija Fodor
Supervisor (title, first name, last name, position, institution)	Dr Dušan Jakovetić, associate professor, Faculty of Sciences, University of Novi Sad Dr Danijela Boberić Krstićev, associate professor, Faculty of Sciences, Unicersity of Novi Sad
Thesis title:	Implementation and analysis of a class of algorithms for distributed convex optimization: Performance evaluation and tradeoffs in practical HPC clusters
Language of text (script):	English language
Physical description:	Pages 225 Chapters 4 References 163 Tables 19 Illustrations 52 Graphs 0 Appendices 2
Scientific field:	Informatics
Scientific subfield (scientific discipline):	Computer science
Subject, Key words:	High-performance computing, Distributed optimization, ADMM, Performance evaluation
Abstract in English language:	The significance of distributed optimization algorithms manifests through growing interest in various application domains. It finds its use in Big Data analytics, distributed machine learning, distributed control, vehicle networks and smart grid, inter alia. This thesis focuses on primal and dual distributed convex optimization methods. First, it introduces a general algorithmic framework of first and second order methods of primal type, that utilize the concepts of sparsified communications and computations across a connected graph of working nodes. Besides several already existing methods, the

[2] The author of doctoral dissertation has signed the following Statements:
5б – Statement on the authority,
5в – Statement that the printed and e-version of doctoral dissertation are identical and about personal data,
5г – Statement on copyright licenses.
The paper and e-versions of Statements are held at he faculty and are not included into the printed thesis.

	framework also includes novel variants that utilize unidirectional communication. Although there have been a remarkable amount of theory and theoretical advances in this field, practical evaluations of methods on real data and practical large scale and High Performance Computing (HPC) systems are of much smaller volume. Therefore, we developed the implementations and performed various numerical evaluations of the proposed methods in an actual, parallel programming environment. The implementations were developed using the Message Passing Interface (MPI) and tested on a High Performance Computing cluster. These empirical evaluations result with very useful insights and guidelines regarding performance and highlights the most important communication-computational tradeoffs in a real execution environment. As there exists a wide set of machine learning algorithms that can be viewed as optimization problems, distributed optimization plays a significant role in this area. The thesis also presents an algorithm for convex clustering, based on the dual method Alternating Direction Method of Multipliers (ADMM), that relies on COMPS Superscalar (COMPSs) framework for parallelization. We provide results of extensive numerical evaluations of the algorithm on a HPC cluster environment, to demonstrate the high degree of scalability and efficiency of the method, compared to existing alternative solvers for convex clustering. The program code for the developed algorithms is open-source and available in the corresponding repositories.
Accepted on Scientific Board on:	30.09.2021. (accepted on Senate)
Defended: (Filled by the faculty service)	
Thesis Defend Board: (title, first name, last name, position, institution)	President: Dr Miloš Stojaković, full professor, Faculty of Sciences, University of Novi Sad Member: Dr Nataša Krejić, full professor, Faculty of Sciences, University of Novi Sad Member: Dr Miloš Ivanović, associate professor, Faculty of Sciences, University of Kragujevac Mentor: Dr Dušan Jakovetić, associate professor, Faculty of Sciences, University of Novi Sad Mentor: Dr Danijela Boberić Krstićev, associate professor, Faculty of Sciences, University of Novi Sad
Note:	

Abstract

The significance of distributed optimization algorithms manifests through growing interest in various application domains. It finds its use in Big Data analytics, distributed machine learning, distributed control, vehicle networks and smart grid, inter alia. This thesis focuses on primal and dual distributed convex optimization methods. First, it introduces a general algorithmic framework of first and second order methods of primal type, that utilize the concepts of sparsified communications and computations across a connected graph of working nodes. Besides several already existing methods, the framework also includes novel variants that utilize unidirectional communication. Although there have been a remarkable amount of theory and theoretical advances in this field, practical evaluations of methods on real data and practical large scale and High Performance Computing (HPC) systems are of much smaller volume. Therefore, we developed the implementations and performed various numerical evaluations of the proposed methods in an actual, parallel programming environment. The implementations were developed using the Message Passing Interface (MPI) and tested on a High Performance Computing cluster. These empirical evaluations result with very useful insights and guidelines regarding performance and highlights the most important communication-computational tradeoffs in a real execution environment. As there exists a wide set of machine learning algorithms that can be viewed as optimization problems, distributed optimization plays a significant role in this area. The thesis also presents an algorithm for convex clustering, based on the dual method Alternating Direction Method of Multipliers (ADMM), that relies on COMPS Superscalar (COMPSs) framework for parallelization. We provide results of extensive numerical evaluations of the algorithm on a HPC cluster environment, to demonstrate the high degree of scalability and efficiency of the method, compared to existing alternative solvers for convex clustering. The program code for the developed algorithms is open-source and available in the corresponding repositories [1, 2].

Izvod

Značaj algoritama distribuirane optimizacije manifestuje se kroz rastuće interesovanje u različitim domenima primene. Primenjuje se u analitici velikih podataka, distribuiranom mašinskom učenju, distribuiranoj kontroli, mrežama vozila i pametnim mrežama, između ostalog. Ova teza se fokusira na primarne i dualne distribuirane metode konveksne optimizacije. Najpre se uvodi opšti algoritamski okvir metoda prvog i drugog reda primarnog tipa, koje koriste koncept sparsifikovane komunikacije i izračunavanja preko povezanog grafa čvorova. Pored nekoliko već postojećih metoda, pojavljuju se i nove varijante koje koriste jednosmernu komunikaciju. Iako u ovoj oblasti postoji izuzetno velika količina teorije i teorijskog napretka, praktične evaluacije metoda nad stvarnim podacima i praktičnim sistemima računara visokih performansi (High Perfromance Computing - HPC) velikih razmera, su mnogo manjeg obima. Stoga smo razvili implementacije i izvršili skup različitih numeričkih evaluacija predloženih metoda u stvarnom, paralelnom programskom okruženju. Implementacije su razvijene korišćenjem tehnologije Message Passing Interface (MPI) i testirane su na računarskom klasteru visokih performansi. Ove empirijske procene rezultiraju veoma korisnim uvidima i smernicama u vezi sa performansama i naglašavaju najvažnije kompromise između komunikacije i izračunavanja, u stvarnom okruženju izvršavanja. S obzirom na postojanje širokog skupa algoritama mašinskog učenja koji se mogu posmatrati kao optimizacioni problemi, distribuirana optimizacija ima vrlo značajnu ulogu u ovoj oblasti. U tezi je takođe predstavljen i algoritam za konveksno klasterovanje, zasnovan na dualnoj metodi Alternating Direction Method of Multipliers (ADMM), koja se oslanja na COMPS Superscalar (COMPS) pristup za paralelizaciju. Prikazujemo rezultate opsežnih numeričkih evaluacija algoritma na HPC računarskom klasteru, kako bismo demonstrirali visok stepen skalabilnosti i efikasnosti metode, u poređenju sa postojećim alternativnim pristupima za konveksno klasterovanje. Programski kod za razvijene algoritme je softver otvorenog koda, i dostupan je u odgovarajućim repozitorijumima [1, 2].

Preface

Distributed convex optimization represents a field whose application is constantly expanding. This is influenced by a demand to solve large-scale problems, on growing volumes of data. The need for solutions that enable problem partitioning is emerging. Therefore, distributed convex optimization finds its use in a wide variety of domains, as the need for fast processing is rapidly increasing. High performance computing is a great ally to these problems, as it provides a technical background to achieve efficient and scalable execution.

This thesis focuses on the implementation and extensive analysis and evaluations of a set of distributed convex optimization problems, by providing parallel algorithms, evaluated on a computer cluster environment. It considers two directions. The first one is oriented towards a class of primal distributed optimization methods. The second one is dedicated to a well-known dual optimization method, Alternating Direction Method of Multipliers (ADMM) [3].

The thesis is organized into four chapters. In Chapter 1, a short introduction to the main concepts is provided, supported by an overview of the related work, with an emphasis on the motivations and contributions of this work.

In Chapter 2, we focus on a class of primal optimization methods, by providing some theoretical insights, but also stressing out the important implementation aspects. We also carry out a thorough empirical evaluation of the developed algorithms on a computer cluster and derive conclusions about the nature of different methods. These methods are implemented using Message Passing Interface (MPI) [4].

Chapter 3 is dedicated to a dual distributed optimization method, ADMM, with an emphasis on the development and analysis of a parallel, ADMM-based convex clustering approach. We provide the theoretical aspects of the proposed method, and also describe the algorithm itself. A comprehensive evaluation of the method on an HPC cluster is also described, by analysing different aspects of the algorithm and deriving necessary conclusions. We also briefly explain the implementation of additional ADMM-based parallel algorithms. These algorithms are implemented using the COMPSs [5] framework. Therefore we provide a comparison of the framework used in Chapter 2 (MPI) and in Chapter 3 (COMPSs) and derive some insights into their properties. Finally, we derive the conclusions on the proposed methods in Chapter 4.

Acknowledgements

I would like to express my gratitude to my supervisor, Dr. Dušan Jakovetić, for all the amazing ideas and the time spent on working on this thesis and the papers that contributed to it. I thank him for the patience and commitment. I can not emphasize enough how grateful I am for everything I learned through our collaboration. I would also like to thank to my another supervisor, Dr. Danijela Boberić Krstićev for all the discussions, tips and encouragements during the years. I am extremely grateful for the knowledge that she selflessly shared with me all the time, it was always a pleasure to work together. I also thank professor Dr. Nataša Krejić, for the fruitful collaboration, but especially for the kindness, support and patience while working on the contributions of this thesis. I am also grateful to the other member of the committee, Dr. Miloš Stojaković and Dr. Miloš Ivanović for their time spent on reading the thesis and providing valuable feedback. I would also like to thank professor Dr. Srđan Škrbić, for introducing me to the world of parallel programming, and for the tremendous amount of help throughout my academic career. I also thank Dr. Nataša Krklec Jerinkić for the nice collaboration and support. Many thanks to my colleague Aleksandar Armacki, with whom we developed the machine learning algorithms for the I-BiDaaS project, mentioned in the thesis. A special thank goes to Bane Ivošev, Vladimir Jokić and Žarko Bodroški for giving me access and support for the resources of the Axiom cluster, making my extensive experiments possible. Also, I am grateful to the stuff at the Institute of physics, Belgrade, for the access to the PARADOX computing facility. I also thank all of my colleagues for the positive work environment and support.

* * *

My greatest gratitude goes to my dear parents, Virka and Rudi. I am so grateful for all their love and support and for always believing in me unconditionally. Everything that I am and everything that I achieved is inspired by their selfless support. I would also like to thank other family members and relatives, who kept me asking about the progress of the thesis. I also thank Attila, for all the nice moments and the valuable experiences. Finally, I thank my friends, for the talks, support and encouragement.

✳ ✳ ✳

The work presented in the thesis is supported by the I-BiDaaS project, funded by the European Commission under Grant Agreement No. 780787, and also by EU Project CYRENE, which has received funding from the European Unions Horizon 2020 research and innovation programme under grant agreement No 952690. The research presented in the thesis was performed using the AXIOM HPC facility at Faculty of Sciences, University of Novi Sad and the PARADOX supercomputing facility at the Scientific Computing Laboratory, Center for the Study of Complex Systems of the Institute of Physics Belgrade.

Dedicated to my parents...

'The only person who is educated is the one who has learned how to learn and change.'

Carl Rogers

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In this chapter, the introductory concepts are briefly presented, including also the main motivations, expectations and contributions of this PhD thesis.

1.1 Distributed convex optimization

Mathematical optimization [6] is often described as a tool that enables to make the best possible choice to be a solution for a specific problem. In other words, mathematical optimization seeks to maximize or minimize a function, called objective function, by determining the best values, regarding some input data. Convex optimization [6] is a subclass of mathematical optimization, where the final goal is to minimize a convex function over convex sets. The set of areas where convex optimization can find its use is very broad. Nowadays, the constantly increasing demand for handling rapidly growing data volumes requires new solutions, that can solve problems in an efficient manner. Distributed convex optimization [3, 6, 7] provides a way to partition convex optimization problems into sets of connected subsystems. Using a set of workers to solve these sets of problems leads us to the distributed multi-agent convex optimization methods [8, 9, 10, 11, 12, 13], that are able to solve a set of problems, working simultaneously, and hence provide the final solution in less time. Incorporating high performance computing [14] techniques to the described methods, a solution model for a wide range of problems can be established.

1.1.1 Distributed optimization methods

This subsection introduces basic concepts and terminology of distributed convex optimization, at a high level. We refer the reader to [3, 6, 7] for further technical details to the subject. Distributed convex optimization finds its use in a wide range of problem

types. The categorisation of distributed optimization problems [6, 7], can be defined in a few ways:

- **Unconstrained and constrained problems**. Constrained optimization problems tend to optimize an objective function with respect to some variables and set of constraints defined on those variables. The constraints can be of different types, but most often they are equality or inequality constraints. Unconstrained problems do not involve any constraints on variable values. Usually, a constrained problem can be adapted to an unconstrained problem, often using penalty methods. We consider both classes, unconstrained problems in Chapter 2, and a constrained problem in Chapter 3. In fact, the nature of the problems in Chapter 3 is unconstrained, but when solving them, we utilize "cloning variable constraints", in order to adapt the problems to ADMM.

- **Primal and dual problems**. By the principle of duality, optimization problems can be formulated in two different domains, primal and (Lagrangian) dual. Primal optimization methods contain only primal variables. By means of weak duality, the solution of the dual problem is the lower bound to the solution of the primal problem. The difference between the optimal values of the primal and dual variables is called duality gap. In convex optimization, we are dealing with strong duality, i.e., when the original, primal problem is convex, under additional mild technical conditions, the duality gap is zero, and, moreover, the primal solution can be recovered from the dual solution. This means that the duality gap is zero under a defined constraint qualification condition. Chapter 2 considers primal methods, while Chapter 3 is dedicated to a dual method, ADMM.

- **Stochastic and deterministic problems**. Stochastic problems aim to minimize or maximize an objective function when randomness is present. Deterministic problems (mathematical programming) follow a rigorous mathematical approach, where the aim is to obtain the global solution and provide theoretical guarantees for the solution. In Chapter 2, we consider stochastic methods with different communication probabilities, while Chapter 3 covers a deterministic approach.

- When considering the underlying communication model, two different approaches exist: **manager-workers and fully distributed communcation models**. A multi-agent system consists of a set of agents and an underlying network, that determines how the agents (computing units) communicate. This categorization can be made on the level of the nature of algorithms. The manager-workers paradigm means the existence of a manager node, that coordinates the process of computation

and communication inside a network of nodes, where the worker nodes perform some assigned tasks. The principle of fully distributed model implies a generic connected network of nodes, that mutually work on a solution of a problem. In Chapter 2, a fully distributed model is used for the algorithmic framework of primal methods, while Chapter 3 relies on the manager-workers model, applied for the dual, ADMM-based algorithm for convex clustering.

This thesis includes work on different classes of distributed convex optimization methods. Firstly, we focus on primal methods of first and second order, that represent stochastic unconstrained approaches. The implementation of these methods uses a principle of a fully distributed network of nodes. Secondly, we consider a class of dual methods, namely Alternating Direction Method of Multipliers (ADMM) [3], where the implementation relies on the manager-workers approach.

1.2 High performance computing

High performance computing (HPC) [14] is, as its name suggests, a computing strategy that produces results with high performance properties (usually referred to as time efficient). It represents a contemporary approach to aggregate multiple computing units, in order to solve a common problem. In the context of high performance computing, related terms: parallel computing and computing cluster should also be explained. Parallel programming [15] provides the execution of a set of tasks simultaneously, in parallel. Parallelism can be completely transparent to the end users. However, in order to exploit the benefits of it, a completely different programming strategy (compared to serial programming) is required to be employed by the developer. Programs that execute in parallel need to satisfy a few concepts. First of all, the problem being solved should be divisible and the best scheme for dividing it should be used. This means splitting the input data and investing simultaneous computational effort, while expecting the same results as when doing the same task without parallelism. Second, a problem should have a notable volume, in order to benefit from parallelization. Besides the apparent advantages, a parallel program usually has its bottlenecks. This usually means that there should exist synchronization points, or message exchange points, that can use a significant portion of time, depending on the context. For that reason, the program should be designed such that it contains as little as possible highly time consuming points.

Parallel computing has several possible dimensions. Shared memory systems and distributed memory systems can be identified as commonly used. Shared memory systems

utilize the existence of a set of cores, while having access to the same memory locations. This is the basic form of parallelization, that can be achieved on each computer that possesses a CPU with multiple cores. One of the most common frameworks with shared memory approach is OpenMP [16]. Practically, OpenMP represents an extension to the C and Fortran programming languages. It can enhance the performance of an algorithm, until the number of available cores that can use the shared memory is sufficient, and while the shared memory can hold the required amount of data. When any of these conditions can not be satisfied anymore, one should think about a more flexible approach.

Distributed memory systems imply that each computational unit i.e. core has its own memory space. This means that they need a way to communicate somehow, as they do not have access to the same memory locations. The most popular framework with distributed memory approach is the Message Passing Interface (MPI) [4]. MPI is a standard, and there exist several relevant implementations for it, as OpenMPI [17] and MPICH [18]. MPI is usually used with the C programming language, but can also be utilized with C++, Python and other languages. The data exchange happens by sending and receiving messages among the processes. By using MPI, different processes can run on different machines, connected by a network. Besides these approaches, there also exists a possibility to utilize the power of graphics processing units (GPUs). The most common interface for this method is CUDA [19] (Compute Unified Device Architecture), where a CUDA-enabled GPU is used for processing. These different parallelization approaches can also be combined, depending on the needs.

As already mentioned, high performance computing is usually being manifested through the use of a group of connected machines, i.e. cluster of computers. This simply means that instead of using one single unit to solve a problem, one can utilize the power of multiple, mutually connected machines that work together. Each machine in a cluster is called a node. The components i.e. nodes are usually connected by a fast local area network. The network properties play an important role, when it comes to the message exchange speed. A cluster can contain a various number of nodes, but at least two. If a node fails for some reason, the rest of the cluster can continue working undisturbed. Each node in a cluster has an operating system and necessary software and libraries installed. Usually, computer clusters are meant to be used by multiple users for different tasks, i.e. cluster jobs. For that reason, most commonly, a special software called job scheduler is also present on the cluster. This software is responsible to schedule different jobs, by putting them in a queue.

MPI is very commonly used to write parallel programs, that operate on a cluster. Writing parallel programs, that run on a cluster environment, can lead to serious time savings

and to solutions that could not be acquired using serial programs i.e. without parallelism. Today, there also exists a variety of novel frameworks that can also provide parallelism, while being higher level. This means that the process of parallelization is eased and even automated to certain extent.

COMPS Superscalar (COMPSs) [5] is a representative of a high level framework, that is meant for parallel application development and execution. It enables executing applications on distributed infrastructures. This is a task based framework that can be used with Java, Python and C/C++. The parallelization process is simplified and made mainly transparent for the programmer. The code can be written as a serial application implementation, with the addition of task annotations to some portions of the code. This seems related to shared memory systems, but as a matter of fact, COMPSs is able to work on distributed cluster systems by spreading the data among the nodes implicitly, without demanding message exchange handling by the programmer. However, this level of comfort comes with some losses regarding performance, when compared to MPI. In [20], the authors compare the COMPSs framework to Apache Spark [21] and conclude that COMPSs is able to achieve comparable performance. However, in [22], it was shown that Spark does not perform generally better than MPI. Writing parallel programs in lower level languages and frameworks, as C with MPI, provides a high level of control and opens additional opportunities to make the code more efficient. The tradeoff between performance and programming effort is a present issue with mentioned frameworks. The choice of the more suitable one depends on the particular needs and domain.

This thesis considers two different parallelization frameworks: MPI for the class of primal methods in Chapter 2, and COMPSs for the dual, ADMM method, in Chapter 3. Both approaches are being tested on a computer cluster, in order to explore the scaling properties and gains of parallelization in terms of execution time. We also provide a brief comparison of these frameworks in Section 3.5.

1.3 Motivation and Objectives

High performance computing finds its use in a very wide variety of fields. On the other hand, distributed optimization represents a basis that is of interest in different areas. By definition, distributed multi-agent convex optimization problems are divisible to a set of subproblems, that can be solved by multiple agents, as already explained. If we bring together these two sides by utilizing the power of parallel programming in order to make the agents (i.e. nodes) solve their subproblems in parallel, then significant results could be expected, with regard to performance. This means that we could be able to apply

parallel distributed optimization to different domains, on large amount of data.

The high applicability of distributed multi-agent optimization methods in various domains is evident. For example, these include distributed machine learning [23], distributed control [24], vehicular networks [25], smart grid [26], etc. Some relevant applications have been demonstrated in [3]. Distributed multi-agent optimization is nowadays already a mature theoretical area, e.g. [8]. Several first [8, 9, 10] and second order [11, 12, 13] distributed methods of primal type have been proposed in the literature. The theoretical properties of these methods have been well understood, e.g., in terms of theoretical iteration-wise convergence rates. However, there is a very restricted amount of scientific investigation of distributed multi-agent optimization methods in realistic parallel computational systems. Carrying out such studies is extremely important as there is a significant gap between theoretical studies of the methods and actual performance in practical infrastructures. For example, how a derived iteration-wise convergence rate translates into actual communication and computational costs is very hard to understand without empirical evaluation. The results of experimentation on real data sets, performed on a cluster environment, while practically demonstrating the distributed nature of the algorithm by parallelzation, could be of significant interest.

One of the main areas of application of distributed convex optimization methods is in machine learning. Our empirical evaluations are also based on some important machine learning algorithms. We use quadratic and logistic loss functions for our primal methods evaluation. From the aspect of dual methods, we introduce an ADMM-based convex clustering approach and evaluate its performance. Clustering is a widely used unsupervised learning problem in various domains [27, 28, 29, 30]. The k-means algorithm [31] is a well-known and commonly used clustering approach, but other useful methods have been also proposed as K-means++[32], k-medians [33], and Bregman clustering [34]. Convex clustering emerged as a valuable idea, in [35, 36, 37, 30, 38]. It exposes some advances compared to conventional clustering methods, as it formulates the clustering problem as a convex optimization problem, based on a sum-of-norms penalty and therefore it eliminates local minima-related issues. Also, convex clustering does not need a predefined number of clusters to be specified in advance. There exists a set of useful convex clustering approaches in the literature [35, 36, 37, 30, 38], but the main issue is that they do not scale well with the number of input data points. This represents a motivation for developing a scalable, parallel convex clustering approach, by relying on the dual optimization method ADMM.

A very important aspect is that the above mentioned optimization problems can be solved numerically, in an efficient manner. Therefore, our main focus is on creating the appro-

priate implementations for the class of distributed multi-agent methods described later, and performing different aspects of empirical evaluations. This approach enables comparing the behaviour of different methods, as well as to practically prove the theoretical advances. The results will also recognise the bottlenecks, and highlight the most efficient setups for particular domains.

1.4 Contributions

This thesis subsumes several contributions including the development and analysis of parallel implementations of a class of primal and dual methods, a comparison of corresponding methods, a thorough analysis of their properties and a practical proof of the proposed theoretical concepts. We also introduce and provide a description for two novel methods: a primal method with unidirectional sparisifed communication and a dual method for convex clustering, based on ADMM. The provided extensive empirical evaluations of the methods provide important insights into performance, scalability and applicability of the proposed methods.

1.4.1 Contributions regarding primal methods

With respect to primal methods, we first explain the development of parallel MPI implementations for the considered classes of optimization methods of first and second order. In addition, we discuss different approaches regarding implementation strategies and identify the most efficient one. Also, an extensive empirical evaluation of the developed implementations of the methods is discussed. The tests are performed on different data sets, of different volumes, and various aspects are observed during the testing phase. This enables the derivation of different dimensions of conclusions. For example, we are able to identify which methods are more suitable depending on input data size. Also, we can identify the most appealing communication sparsification strategy, regarding performance. We compare first and second order methods and identify the setups where they perform better. The topology of the underlying graph of communication is an important aspect, that affects performance of distributed multi-agent optimization algorithms. More densely connected graph implies faster information flow iteration-wise, whereas more communication links may have a negative effect with respect to the cost incurred by excessive communications. On top of an all-to-all computer cluster wired communication infrastructure, we implement and experiment with multiple communication topologies and study the topology effects on the performance. Furthermore, the scaling properties of the developed

methods are also demonstrated. Other similar additional conclusions are also shown, as explained later.

The underlying implementation framework that we use here is MPI (Message Passing Interface, [4]) running on a computer cluster, as it is a standard and widely adopted computational system. We develop MPI implementations for a class of first and second order distributed multi-agent methods, that provide a framework for parallel, distributed optimization problem solving. Also, we provide open-source code for the described methods [1]. As one of our main concerns is to carry out a thorough and systematic empirical study of the mentioned class of methods, it is extremely important to implement different variants i.e. methods of the algorithm. These methods utilize different variants of sparsification of communications and/or computations along iterations. This means, that we practically allow idling for a subset of nodes during the execution, in order to reduce the amount of resources used to run the algorithm. We consider both first and second order methods that exhibit either unidirectional or bidirectional randomized sparsified communications. The communication sparsification is determined by a probability p_k, that represents the probability that a node communicates at iteration k; the quantity p_k is a design parameter that is either increasing, decreasing, or constant. These strategies give rise to a number of methods. The studied class of methods subsumes several existing algorithms [39, 40, 41, 42, 43, 44] but also includes several methods that have not been studied before, neither numerically nor analytically. We also compare the methods utilizing communication sparsification with methods that do not utilize this mechanism.

As already mentioned, in order to achieve sparsification, the principle of randomized communication with some probability can be successfully used here, as using randomized communication at the level of algorithm design is a well established topic. For example, gossip [45] is an outstanding example for this. An advantage of using this approach is the possibility to explore the case when communication sparsification is not fully determined by the algorithm designer, but instead is dictated by random link failures (e.g., packet dropouts in wireless networks).

Beside exploiting communication sparsification itself, we may have unidirectional or bidirectional communication strategies. The unidirectional strategy allows for inactive nodes to receive messages (but not to send anything), while the bidirectional strategy only allows two-sided communication (an inactive node cannot communicate at all). These strategies are also implemented and combined with the previously discussed aspects.

Observing different aspects that we are interested in, the considered class of methods subsume several existing, already described methods [39, 40, 41, 42, 43, 44], but it also

provides certain methods that are either completely novel, or for which theoretical analysis was not available in the literature yet.

Concisely, the goals that the empirical evaluation is aiming to achieve are the following:

1. An assessment of real benefits of sparsifying communications and/or computations across working nodes, which have been already proved to be beneficial theoretically [39].
2. A comparison of different variants of the sparsification strategies, in order to identify the most efficient ones. We provide evaluations from different aspects, so we can identify the factors that influence the performance of different methods in different setups.
3. A comparison of unidirectional and bidirectional communication strategies. More precisely, we are interested in performance benefits/losses when applying communication sparsification on two different ways: enabling mutual bidirectional communication only between active working nodes vs. also enabling mutual bidirectional communication between active workers, but also allowing idle nodes to receive data from active nodes (unidirectional communication).

One of the main motivations for using sparsified, randomized communication is to reduce the amount of resources used for computation. In this case, the most important resource is time. We strive to reduce the time spent on data exchange, and hence reduce the overall execution time of the algorithm, as the time consumption evaluation of different parts of the algorithm identifies the communication as the bottleneck. This evaluation is performed on the algorithm without sparsification used. It is expected that the communication takes the largest portion of time, and that with data size increase, it also gets even higher. That is the main motivation to sparsify this part of the algorithm. But generally, the choice of omitting to communicate in some cases can also lead to another benefits and savings in other resources as bandwidth or transmission power of wireless devices, when considering wireless networks. Our work on this topic have been published in [46].

1.4.2 Contributions regarding dual methods

When considering dual methods, the focus is on Alternating Directions Method of Multipliers (ADMM) here, as this method is designed for distributed setups. It breaks a problem into smaller pieces and enables solving of those problem parts separately, in parallel. The main contribution is the development of a novel, convex, fully parallel ADMM-based

distributed convex clustering method. It relies on manager-workers (server-clients) communication model. We describe the theoretical properties of the proposed method first and provide a conceptual design for the algorithm.

The implementation of the parallel ADMM-based convex clustering method is written in Python, and the COMPS Superscalar (COPMSs) framework is used for parallelization. It has already been widely adopted and extended in numerous scientific projects and papers (e.g. [47, 48, 20]) offered as a tool to develop scientific applications and optimize their execution in distributed infrastructures. We discuss the implementation details of the proposed clustering method in this framework and provide open-source code for it [2].

As the empirical evaluation of the properties of the methods is one of our main concerns, we perform a thorough analysis oriented towards different aspects. First, we evaluate the accuracy of the developed method, by observing the percentage of accurately clustered data points, when the real labels denoting the clusters are available, or by observing an appropriate accuracy metric, when the real labels are not available. The scaling properties of the proposed method are also an important consideration point. Therefore, an analysis of the execution time of the algorithm for different number of working nodes is also provided. Another important aspect is to asses the proposed method, with regard to other relevant clustering methods, in terms of accuracy and scalability. Therefore, we make a comparison of our method to other clustering approaches and draw some important conclusions.

The extensive set of evaluations implies using different data sets, of different volume. We test the algorithm on both synthetic and real data sets. Moreover, a subset of tests is performed on a real, industrial data set, that was part of use cases defined under a H2020 project Industrial-Driven Big Data as a Self-Service Solution (I-BiDaaS) [47], and is publicly available on the Zenodo repository [49, 50] of the project. We also discuss the implementation of other machine learning algorithms, based on ADMM and using COMPSs framework, by providing some example implementations.

To summarize, the main goals and contributions, related to the ADMM-based approach are:

1. The development and evaluation of a novel parallel, convex clustering approach, that exhibits convex clustering benefits (e.g., no need for a pre-defined number of clusters).

2. The evaluation of the accuracy of the developed method. It turns out that the method exhibits high level of accuracy, comparable to other clustering methods

(e.g., in terms of the silhouette score or percentage of accurately clustered points when the ground truth for expected outcomes is known).

3. The evaluation of the scaling properties of the developed method. The method improves scalability over existing convex clustering solvers and is suitable for use with larger data sets.

Furthermore, we provide two additional implementations of machine learning algorithms that relies on ADMM and utilize COMPSs: logistic regression and lasso regression. These algorithm were developed under the H2020 project I-BiDaaS, and the ADMM-based lasso regression was also incorporated into the dislib library [51]. The work regarding our parallel, ADMM-based convex clustering algorithm has been submitted in EURASIP Journal on Andvances in Signal Processing.

1.5 Related work

We can identify a few threads of work regarding the topics related to this thesis. The first subset is the broadest and it subsumes applications of distributed multi-agent optimization in different areas. The second one is dedicated to theoretical advances in distributed optimization methods. The third subset covers the work that is oriented to empirical evaluation, testing of distributed optimization methods in real, cluster environments. A significant amount of literature dedicated to theoretical development of distributed optimization methods is available nowadays. A proportionally much smaller body of scientific literature focuses on testing and evaluation of the distributed optimization methods over actual distributed infrastructures.

Applications

Regarding the topic of multi-agent optimization, there is a wide variety of domains, that all utilize the concept of distributed workload between a network of agents, and there exists a vast amount of work on this topic. Therefore, we demonstrate only a set of examples here. Distributed machine learning represents one of the main areas of interest. Reference [23] describes a fully decentralized multi-agent reinforcement learning, where the agents are tending to maximize the final result incrementally, while having a step for the internal calculation and a consensus update relying on the network. In [52],the main strategies and principles of distributed machine learning are discussed, wit application on Big Data. The authors emphasize the importance of distributed optimization in the

area of machine learning, as the majority of the considered algorithms can be actually viewed as optimization problems. Reference [53] presents a parameter server framework for distributed machine learning, that represents a communication efficient approach, meant to solve non-convex non-smooth problems with convergence guarantees.

On the other side, distributed control represents an other direction, where distributed optimization plays an important role. Reference [24] introduces important results regarding different dimensions of cooperative control, based on distributed multi-agent systems. The considered dimensions are distributed control and computation, adversarial interactions, uncertain evolution and complexity management. The range of applications for this area is wide, and includes autonomous vehicle systems, cooperative robotics, distributed computing, sensor networks and data network congestion control as well.

In [54], a formation control problem with velocity assignment of networked multi-agent systems with heterogeneous time-delays is in the focus. In this approach, only a subset of agents receive a signal. The paper describes the problems that appear caused by the heterogeneous time-delays and also proposes an algorithm to minimize the error that appears. Reference [55] considers a non-convex problem of optimal power flow for microgrids, where a semidefinite programming relaxation is used to obtain a convex problem. The problem is being solved in a distributed manner, by utilizing ADMM. In [56], the authors focus on distributed estimation of deterministic vector parameters, using ad hoc wireless sensor networks. The problem is being transformed to a set of constrained convex optimization sub-problems, that are meant for distributed implementation. Spectrum sensing represents an other area, where optimization finds its use. In [57], a wideband spectrum sensing technique is introduced. Here, the spectrum sensing problem is considered as a class of optimization problems, which maximize the aggregated opportunistic throughput of a cognitive radio system under some constraints on the interference to the primary users. The application areas of distributed multi-agent optimization include a wide range of other directions, as vehicular networks [25], smart grid [26], etc. In [3], some relevant applications are illustrated as well.

Theoretical advances

The literature dedicated to theoretical development of distributed optimization methods includes an extensive amount of work. As the body of scientific literature is vast, we provide a representative subset of work. In reference [8], the distributed subgradient approach is proposed to solve optimization problems, by using a network of agents. This is a first order, iterative method, that converges event when the objective function is not differen-

tiable. Each agent minimizes its objective function and exchanges data with others, while the topology changes over time. The focus is on convergence, the communication is asynchronous and the connectivity is changing among the agents. Reference [9] also considers the topic, with main focus on distributed stochastic subgradient projection algorithms. It investigates the effects of stochastic subgradient errors on convergence. Reference [11] introduces a distributed network Newton(NN) method that includes second-order updates approximations. This is achieved by means of a distributed implementation of the approximation of the appropriately chosen Newton step. This is a penalty method that replaces a constrained problem with an array of unconstrained problems.

Reference [12] introduces a class of distributed Newton-like methods, referred to as Distributed Quasi Newton (DQN) methods, that are also in the focus in this thesis. It is characterized by the approximation of the Hessian inverse, by splitting it to a diagonal and off-diagonal parts and by inverting the diagonal part. The off-diagonal part is being approximated through a weighted linear function. Reference [39] introduces communication sparsification into distributed second order methods, with increasing probability. Reference [43] is also dedicated to novel methods for zeroth and first order distributed stochastic optimization, based on a probabilistic communication between agents that increasingly sparsifies agent communications over time. The communication probability decreases over time here, and the approach is characterized by a mean square error convergence rate. These papers also introduce an important concept, that is used in the thesis. Additionally, [44] presents a distributed recursive estimator that utilizes directed increasingly sparse communications. In [41], the convergence rates related to distributed optimization with random networks are evaluated. A distributed approach that uses zeroth order optimization, namely a Kiefer-Wolfowitz stochastic approximation approach is introduced in [42]. Also, reference [58] represents a very important theoretical basis, as it is dedicated to the issue of convergence of decentralized gradient descent. Similarly, [59] is dedicated to evaluate the convergence rates of inexact proximal-gradient methods.

In [60], the regularized dual averaging method is introduced. The idea is the use of regularization structure in an online setting, where an optimization problem involves the running average of all past subgradients of the loss functions and the whole regularization term. [61] describes a set of distributed algorithms based on dual subgradient averaging, while defining their convergence rates as a function of the network size and topology. Reference [62] provides an extension to the distributed dual averaging algorithm, that enables communication delays handling. Also, there is work on a new algorithm, that combines dual averaging for convex optimization with a push-sum consensus protocol, called push-sum distributed dual averaging, presented in [63]. In [64], an introduction to

augmented Lagrangian method (ALM) and its variants for solving convex optimization problems is introduced, for large-scale and distributed applications.

Distributed convex optimization by Alternating Direction Method of Multipliers (ADMM) was studied in reference [3], where it is shown that ADMM is an appropriate choice in distributed convex optimization. Furthermore, reference [65] introduces new methods as a combination of stochastic optimization techniques and ADMM, where the dual averaging and proximal gradient descent are used for online ADMM. A specific use of ADMM is presented in [66], where it is applied to a class of total variation regularized estimation problems. Furthermore, the behaviour of ADMM is well investigated. As an example, reference [67] focuses on analysis of the convergence of distributed ADMM, when an additive random node error is present, while also providing numerical results. In [68], a practical, parallel implementation of ADMM is introduced, and the results are again supported by some numerical evaluations. Reference [69] introduces a flexible Alternating Direction Method of Multipliers, called F-ADMM. The algorithm updates blocks of variables by using a Gauss-Seidel scheme. The authors show that the algorithm is globally convergent and that there is a case where the algorithm is partially parallelizable. Working with parallel applications necessarily involves large amounts of data, and there is a huge interest in finding the best approaches to deal with such scenarios. In [70], it is studied how to implement ADMM for large data sets, mathematically.

Reference [71] introduced a stochastic, efficient quasi-Newton method, using the BFGS (Broyden Fletcher Goldforb Shanno) update formula, in order to take advantage of the curvature information during approximation by points. This is an iterative method for solving an unconstrained problem, that determines the direction of the descent by adding curvature conditions to the gradient. In [72], a variance reduction method for stochastic gradient descent is proposed, called stochastic variance reduced gradient (SVRG). Then, as an extension to the previously mentioned two papers, a stochastic L-BFGS algorithm is introduced, and its linear convergence rate is proven in [73]. On the other side, an interesting study on vertical federated learning for logistic regression, based on a quasi-newton method is shown in [74].

A fast distributed proximal gradient method was proposed in reference [75], for the optimization of the average of convex functions. This is also an iterative approach, where agents recompute their estimates incrementally. The approach uses a time varying topology and is relying on Nesterov-type acceleration techniques and uses multiple points for communication inside iterations. The paper also discusses the convergence rate of the method and supports it by numerical experiments. [10] proposes fast distributed gradient algorithms, that are also based on the centralized Nesterov gradient algorithm. The

authors prove fast convergence to the exact solution, with an advanced convergence rate (similar to centralized Nesterov gradient), for convex, coercive, three times differentiable and with Lipschitz continuous first derivative cost functions. This work also introduces a distributed fast gradient algorithm for composite non-differentiable costs, with constant step size. An asynchronous randomized dual proximal gradient method is introduced in [76]. It is meant for large-scale distributed optimization, with the main idea to properly choose the primal variables and hence separate the dual problem into separate blocks. Reference [77] proposes accelerated distributed gradient-like methods, relying on the Nesterov gradient methods. The advantage of these methods is in the faster rates and cheap iterations. The idea of using a variable number of working nodes for distributed gradient methods is proposed in [40]. Our work also relies on these results.

Reference [78] presented an incremental sub-gradient approach, suitable for distributed optimization in networked systems, that uses a fixed step size. A new distributed algorithm for convex optimization problems related to big data is introduced in [79]. This is a randomized block subgradient approach, meaning that the nodes can exchange only blocks of their solutions at once. An equally interesting distributed consensus-based subgradient method is presented in [80]. This approach assumes that each agent accumulates information about past gradients of neighbours, while the underlying undirected graphs are switching. Reference [81] also describes a distributed subgradient method for multi-agent optimization. Here, a time-varying network and quantized communications are considered. On the other hand, reference [82] also proposes a subgradient method, but focuses on solving an optimization problem over an intersection of fixed point sets of nonexpansive mappings in a real Hilbert space.

An important aspect for evaluation regarding distributed optimization is the topology of the network of nodes that underlies communication. Reference [83] highlights the effects of this aspect. In [84], a framework for optimizing a communication network topology, so that it has the smallest number of links, is presented. It assumes that a prescribed decay rate is satisfied, regarding the response of a distributed control system. Reference [85] investigates the role of network topology, in a particular cluster environment in distributed machine learning algorithms running on that infrastructure.

An interesting approach is described in [45], where special variants of distributed algorithms are introduced, namely the gossip algorithms, where the communication is distributed and the communication is organized such that a node communicates with a randomly chosen neighbour. There are additional ideas on this topic. [86] represents a study on broadcasting-based gossip algorithm, for example. [87] is also related to convex optimization problems over a network of nodes and introduces an augmented Langrangian

gossiping algorithm.

Other relevant works include the following: [88] focuses on analysis of convergence of gradient-based optimizations, where the updates are dependent on delayed stochastic gradient information; [89] describes control and coordination algorithms for groups of vehicles, where a vehicle network performs distributed sensing tasks; [90] is also oriented towards distributed multi-agent optimization, but the communication uses state-dependent model; [91] focuses on reinforcement learning, with gradient approximations computing; [92] describes the performance of a consensus-based distributed subgradient method with random communication topologies; and many others.

Practical evaluations

When considering the work related to practical evaluations of distributed optimization algorithms, on cluster environments, the amount of work available is modest. More recently, there have been works that include MPI-based empirical studies of the methods. Reference [93] proposes an asynchronous subgradient-push method and evaluates its performance on an MPI cluster. The workers perform independently and asynchronously here. It is shown that the iterates on the workers converge to the neighbourhood of the solution, that is dependent on the level of asynchrony. As an addition, reference [94] compares empirically several distributed first order methods. It also finds the asynchronous subgradient optimization algorithm advantageous.

Reference [95] proposes an exact distributed asynchronous subgradient-push algorithm (AsySPA) and provides its performance analysis using an MPI cluster. The working nodes can asymptotically converge to the same optimal solution in this approach. However, the update rates among them differ and there are bounded communication delays. These are solved by adaptive step size adjusting.

Reference [96] provides a theoretical and empirical study of communication and computational tradeoffs for the distributed dual averaging method. It focuses on scalability issues. Also, it identifies that the communication reduction over time, can lead to faster execution. Finally, [97] also focuses on the distributed dual averaging method and provide several useful guidelines about practical design and performance of the methods. Additionally, reference [98] is directed towards asynchronous distributed optimization and proposes a family of randomized primal-dual block coordinate algorithms, namely it utilizes doubly stochastic coordinate optimization with variance reduction (DSCOVR).

Regarding primal methods, with respect to existing studies, this thesis contrasts with them along several dimensions. First, it considers a different class of methods with respect to

existing empirical studies, where the considered methods include various strategies for communication sparsification. We investigate both first and second order methods with sparsifed communications, as well as different communication strategies during sparsification (unidirectional vs bidirectional). Second, the thesis provides novel insights into how the different sparsification strategies mutually compare, as well as how much are they beneficial in practical settings over the corresponding always-communicating benchmark. We investigate the performance of different communication probabilities over various data sets. Interestingly, we show that communication sparsification can lead to significant execution time reductions, when compared to always-communicating approach.

To the best of our knowledge, this is the first empirical evaluation about the class of algorithms with sparsified communications in [39]. Besides that, the concept of unidirectional communication strategy is also tested empirically here and compared to the bidirectional alternatives.

While [44] also considers unidirectional communications, it studies the specific problem of distributed estimation, which translates into quadratic objective functions and stochastic gradient updates. In contrast, our analysis considers generic strongly convex costs. Also, an important aspect is that our work is not limited to first order methods only, as we evaluate the behaviour of both first and second order methods with different sparsification strategies. Also, we introduce a novel method that has not been considered before.

When considering our ADMM-based clustering method, a set of related clustering approaches can be identified. The commonly used approach, k-means clustering was originally presented in [99], while reference [31] introduces the algorithm itself (Lloyds algorithm). k-means represents a straightforward solution, that has been widely used due to its useful features. However, it is sensitive to initialization and may converge to a local minimum. The impact of different initialization methods on the algorithm behaviour was empirically evaluated in [100]. The authors investigate four initialization strategies and identify the most advantageous solutions. There have also been several works that develop improved initialization methods. In [32], a randomized seeding technique was added to the algorithm, in order to improve both its speed and accuracy. On the other hand, reference [101] proposes an algorithm for cluster centers initialization.

The need for convex clustering solutions has been recognized by different authors and several convex clustering formulations have been considered. In [102], an exemplar-based likelihood function was introduced, leading to a convex minimization problem for clustering. This represents an efficient algorithm with guaranteed convergence to the globally optimal solution, supported also by a set of experiments. Reference [103] formulates an

unsupervised learning problem as one convex “master“ problem, that includes nonconvex subproblems which can be solved efficiently. There also emerges the idea of supervised convex clustering, proposed in [104], that strives to find more interpretable patterns via a joint convex fusion penalty. The authors introduce several extensions to this approach, in order to integrate different types of supervising auxiliary variables and they also demonstrate the practical advantages of the proposed method.

The growing interest and wider use of convex clustering becomes evident in different settings. For example, in reference [105] the use of convex clustering approach instead of hierarchical clustering in certain scenarios was investigated. This work derives and tests a novel proximal distance algorithm for minimizing the objective function of convex clustering. Reference [106] introduced an approach, where the idea is to perform sparse convex clustering. This means performing the clustering simultaneously with feature selection, in order to enhance performance. The Sum of Norms (SON) clustering was proposed in [37, 30, 38], as a convex relaxation of k-means clustering. A detailed explanation of the algorithm for SON clustering was given in [35]. This work also includes the presentation of the connection of SON clustering to k-means. It is inspired by the group lasso approach [107].

It has been shown beneficial to utilize weighted pairwise differences, including setting up many weights to zero [106, 108, 109, 110, 111, 112, 36], while the original SON clustering formulation involves all-pairwise-differences across cluster candidates in the SON regularization. The approach with weighted and sparse SON regularizations have been shown to yield faster algorithms and good clustering accuracies [108, 111, 112, 36]. For example, [108] proposes an approach based on weighted minimum spanning trees and k-means bipartite graphs and shows high clustering accuracies of such sparse SON regularization methods.

The SON clustering approach exhibits good theoretical cluster recovery guarantees for all-pairwise-differences SON models. Theoretical advances regarding the perfect recovery properties of the convex clustering model with uniformly weighted all-pairwise-differences regularization were proven in [113, 114]. For weighted and sparse SON models, theoretical recovery guarantees are limited. In reference [115], sufficient conditions for the perfect recovery guarantee of a general weighted convex clustering model are established. However, the weights have to be non-zero for all data point pairs within the same clusterinformation not known a priori.

There are several numerical algorithms, proposed for solving SON clustering problems. Reference [115] introduces a semi-smooth Newton based augmented Lagrangian method,

for large-scale convex clustering, that is supported by a set of numerical evaluations. The authors also establish sufficient conditions for the perfect recovery guarantee of the general weighted convex clustering model in this work. Two splitting methods have been proposed, by using the Alternating Direction Method of Multipliers (ADMM) and the Alternating Minimization Algorithm (AMA) to solve convex clustering problems in [36]. In [109], a novel method for convex clustering, using semiproximal ADMM was introduced. This method is suitable for high-dimensional data. It is based on the sparse group lasso penalty and includes a set of numerical experiments in MATLAB. In [110], a networked k-means algorithm is proposed. The algorithm deals with distributed data and a multi-agent approach. It contains a description of an illustrative numerical evaluation, but does not conduct a thorough empirical study. A Scalable cOnvex cLustering AlgoRithm is introduced in [111], via Parallel Coordinate Descent Method (SOLAR-PCDM). The authors combine a parallelizable algorithm with a compression strategy. SOLAR-PCDM includes the development of a method called weighted convex clustering to recover the solution path by formulating a sequence of smaller equivalent optimization problems and the utilization of the Parallel Coordinate Descent Method (PCDM) to solve a specific convex clustering problem. Furthermore, reference [112] introduces an efficient smoothing proximal gradient algorithm (Sproga) for convex clustering.

Regarding dual methods and our parallel ADMM-based convex clustering approach, with respect to the existing literature, our main contribution is on developing a novel parallel and scalable solver for SON-type clustering. We adopt a sparse zero-one weights SON formulation and leverage it to develop an efficient parallel ADMM method. Unlike existing ADMM-based convex clustering methods that are *sequential* [109], our method is parallel and hence well suited to scalable execution on HPC clusters.

Extensive numerical evaluations show a high clustering accuracy and a high scalability of the proposed method on a number of real and synthetic data sets. Specifically, the achieved accuracy is comparable to alternative sequential k-means solvers, while scalability is significantly improved. It is worth noting that our sparse zero-one SON formulation does not guarantee perfect theoretical recovery guarantees. However, extensive numerical results demonstrate a high clustering accuracy of the proposed method. This is a typical scenario with other sparse clustering methods like, e.g., [109, 108].

1.6 Thesis overview

This thesis is organized into four chapters. Chapter 1 contains the introduction to the field of distributed, parallel convex optimization, including a brief description of distributed convex optimization (Section 1.1) with an overview of different optimization methods (Section 1.1.1) and basic concepts of high performance computing (Section 1.2). The main motivations and objectives are described in Section 1.3, while the contributions are explained in Section 1.4 (containing the contributions for both primal methods, in Section 1.4.1 and dual methods, in Section 1.4.2). The related work is presented in Section 1.5.

Chapter 2 is dedicated to the primal class of optimization methods, where we first describe the background theory (in Section 2.1), starting with basic concepts of optimization and network models (Section 2.1.1). We also describe the aspects of the algorithmic framework, that is used here (Section 2.1.2), and show a convergence analysis for a novel method (Section 2.1.3). Section 2.2 is dedicated to the aspects of implementation. It can be divided into several subsections. First, in Section 2.2.1 the algorithm implementation for strongly convex quadratic cost functions is described. This subsumes the description of the input data preparation for the algorithm, the description of the serial implementation and the description of the parallel implementation. Further, Section 2.2.2 is dedicated to the algorithm development for logistic loss functions. This includes the description of the input data preparation again, the explanation of the details of the serial and of the parallel implementation. The description of the parallel implementation starts with the initial implementation, that is followed by the adaptations of the algorithm, i.e. adapting the algorithm to arbitrary input data size and weight matrix distribution, and adapting the implementation regarding the stopping criterion. After that, a few innovations are described regarding the implementation. First, the algorithm variant without second order update computation is described, followed by the introduction of communicators, and then by the introduction of communication sparsification and unidirectional communication principles. Finally, we derive some conclusions on the parallel implementation for logistic loss functions. Section 2.2.3 contains an explanation of an evaluation metric for comparing the implementation for logistic loss functions with an ADMM based implementation, also written in C using MPI. Section 2.2.4 provides some basic ideas on the way of measuring execution time in parallel applications. Section 2.3 is oriented to experimentation. Here, we first describe the infrastructure used for the tests, in Section 2.3.1. Further, we can divide the experiments into two different sets. The first is containing the intermediate experimentation results, obtained during the development of the implementation (Section 2.3.2). These include the results of the experiments, that are performed on the implementation for strongly convex quadratic cost functions, and the results of the

experiments on the implementation for logistic loss functions. Inside the topic containing results regarding quadratic cost functions, two main topics can be identified: the description of the simulation setup, and the description of the experimental results. The results for logistic loss functions also start with the description of the simulation setup for the experiments. After that, the results of the experiments regarding introducing communicators are displayed, followed by the results for testing different possible algorithm variants with communication sparsification, followed by an explanation on choosing a graph type for the experiments first, and then choosing the best performing algorithm methods. Section 2.3.3 is dedicated to the experimental results regarding the selected set of methods. It begins with the description of the methods, that is followed by the presentation of the experimental results. These include an evaluation of graph types for the network, a demonstration of execution times for different methods, an evaluation of the scaling properties of the methods, a description of results regarding convergence percentage of methods and cost reductions, an assessment of the execution time for different sequences of probabilities, a comparison of the algorithm to ADMM, and the performance profiles for the methods. Finally, Section 2.4 concludes this chapter.

Chapter 3 is dedicated to the dual class of optimization methods, particularly ADMM. The main focus in this chapter is on developing a parallel ADMM-based convex clustering algorithm. Section 3.1 provides the necessary theoretical insights, by means of describing the basic concepts behind the ADMM method and SON clustering. Secondly, it introduces the problem model and the proposed parallel clustering solution (Section 3.1.1). The implementational aspects are described in Section 3.2, where a few different concepts are examined: the input data that is used for the tests (Section 3.2.1), the stopping criterion used (Section 3.2.2) and finally some details about the implementation in PyCOMPSs [116]. Section 3.3 contains the experimental results, including the evaluation of different aspects of the algorithm, as the time consumption of different parts of the implementation (Section 3.3.1), an accuracy evaluation (Section 3.3.2), a scalability evaluation (Section 3.3.3), a discussion on choosing a value for the regularization parameter (Section 3.3.4) and a comparison to other clustering approaches (Section 3.3.5). This comparison concerns a set of clustering methods: SON clustering [37, 30, 38], AMA method [36], DBSCAN [117], SSNAL [115] and parallel k-means provided by Apache Spark [21]. Finally, Section 3.3.6 is oriented towards describing the testing of the method on a real, industrial data set from banking sector, provided by the H2020 project I-BiDaaS. In Section 3.4 some further possibilities for implementation enhancement are considered, while Section 3.5 contains a comparison of MPI and COMPSs frameworks for parallelization. In Section 3.6, two additional ADMM-based parallel algorithms are described: ADMM-based lasso regression (Section 3.6.1) and ADMM-based logistic regression (Section 3.6.2). Section

3.7 concludes this chapter. Finally, Chapter 4 represents the conclusions of the thesis, where the summary of the main achievements of the thesis is described (Section 4.1) with possible application scenarios mentioned (Section 4.2) and future work guidelines (Section 4.3).

Chapter 2

Primal distributed optimization methods

The first contribution of this thesis is the evaluation of a class of primal distributed convex optimization methods of first and second order. This chapter is dedicated to the description of all aspects of these methods and their practical evaluation. We first describe the methods theoretically. Then, the details of implementations are explained, and finally the results and conclusions gathered during testing the methods on a cluster environment are provided. The conducted experiments may reveal some important features of the observed class of methods, that directly affects the possibility of their usage in different setups.

2.1 Background theory

The main goal of this section is to provide all the necessary theoretical concepts, needed to understand the nature of the class of observed methods. Besides the definition of the properties of the methods, an algorithmic framework is also described in details. This section also contains a convergence analysis for a novel method.

2.1.1 Optimization and network models

In this chapter, we consider an unconstrained optimization problem, that can be solved in a distributed manner. Assume that a (connected) network of n nodes is given, where each of the nodes has access to a convex cost function $f_i : \mathbb{R}^s \to \mathbb{R}$, where f_i is known only by node i. Each f_i is assumed to be strongly convex, twice differentiable, and with Lipschitz continuous gradient. The goal for the nodes is to solve the following unconstrained optimization problem:

$$minimize \quad f(x) := \sum_{i=1}^{n} f_i(x). \tag{2.1}$$

We associate with problem (2.1) a graph $G = (N, E)$, where $N = \{1, ..., n\}$ is the set of nodes, and E is the set of edges $\{i, j\}$, i.e., pairs of nodes i and j that can directly communicate.

Graph G practically represents a collection of communication links among computational nodes. The algorithms that we consider may use all these links (no sparsification in communications) or utilize subsets of these links over iterations (sparsified communications). Assume that graph G is connected, undirected and simple (no self nor multiple links). Denote by Ω_i the neighbourhood set of node i. With graph G, we associate an $n \times n$ symmetric, (doubly) stochastic matrix W. The matrix W respects the sparsity pattern of graph G, i.e., for $i \neq j$, $W_{ij} = 0$ if and only if $\{i, j\} \notin E$. However, in the cases of unidirectional communication between the computing nodes, the graph instantiations over the iterations (subgraphs of G) can be directed. We also assume that $W_{ii} > 0$, for all i. It can be shown that $\lambda_1(W) = 1$, and $\lambda_2(W) < 1$, where λ_1 is the largest eigenvalue of W, and $\lambda_2(W)$ is the modulus of the eigenvalue of W that is second largest in modulus. Denote by $\lambda_n(W)$ the smallest eigenvalue of W. There also holds $|\lambda_n(W)| < 1$.

With (2.1), the following optimization problem can be associated:

$$\min_{x \in R^{ns}} \Psi(x) := \sum_{i=1}^{n} f_i(x_i) + \frac{1}{2\alpha} \sum_{i<j} W_{ij} ||x_i - x_j||^2. \tag{2.2}$$

Here, $x = (x_1^T, ..., x_n^T)^T \in R^{ns}$ is the optimization variable that is partitioned into $s \times 1$ blocks $x_1, ..., x_n$. The reasoning behind this transformation is the following. Assume that $s = 1$ for simplicity. Under the stated assumptions on matrix W, it can be shown that $Wx = x$ if and only if $x_1 = x_2 = ... = x_n$, so the problem (2.1) is equivalent to

$$\min_{x \in R^{ns}} F(x), \quad \text{s.t.} \quad (I - W)x = 0, \tag{2.3}$$

where $F(x) := \sum_{i=1}^{n} f_i(x_i)$ and I is the identity matrix. Moreover, $I - W$ is positive semidefinite, so $(I - W)x = 0$ is equivalent to $(I - W)^{1/2}x = 0$. Therefore, (2.3) can be replaced by

$$\min_{x \in R^{ns}} F(x), \quad \text{s.t.} \quad (I - W)^{1/2}x = 0, \tag{2.4}$$

In other words, the constraint $Wx = x$ enforces that all the feasible x_i's in optimization problem (2.3) are mutually equal, thus ensuring the equivalence of (2.1) and (2.3) and the equivalence of (2.1) and (2.4). Further, a penalty reformulation of (2.3) can be stated

as

$$\min_{x\in R^{ns}} F(x) + \frac{1}{2\alpha} x^T (I - W)x, \tag{2.5}$$

where $\frac{1}{\alpha}$ is the penalty parameter. Therefore (2.5) represents a quadratic penalty reformulation of the original problem (2.1). After standard manipulations with the penalty part we obtain

$$\min_{x\in R^{ns}} F(x) + \frac{1}{2\alpha} \sum_{i<j} W_{ij}(x_i - x_j)^2, \tag{2.6}$$

which is the same as (2.2) for $s = 1$. These considerations are easily generalized for $s > 1$.

It is well known, [8], that the solutions of (2.1) and (2.2) are mutually close. More specifically, for each $i = 1, ..., n$, $||x_i^\circ - x^*|| = O(\alpha)$ where x^* is the solution to (2.1) and $x^\bullet = ((x_1^\circ)^T, ..., (x_n^\circ)^T)^T$ is the solution to (2.2).

In more details, Theorem 4 in [58] says that under strongly convex local costs f_i's with Lipschitz continuous gradients (see ahead Assumption 2.1.1 for details), the following holds, for all $i = 1, ..., n$:

$$\begin{aligned} \|x_i^\circ - x^\star\| &\leq (\frac{\alpha L D}{1 - \lambda_2(W)})\sqrt{4/c^2 - 2\alpha/c} + \frac{\alpha D}{1 - \lambda_2(W)} \\ &= O(\frac{\alpha}{1 - \lambda_2(W)}), \end{aligned} \tag{2.7}$$

where

$$D = \sqrt{2L(\sum_{i=1}^n f_i(0) - \sum_{i=1}^n f_i(x_i'))}; c = \frac{\mu L}{\mu + L}; \tag{2.8}$$

and x_i' is the minimizer of f_i. , L is the Lipschitz constant of the gradients of the f_i's, and μ is the strong convexity constant of the f_i's.

The usefulness of formulation (2.2) is that it offers a solution that is close (on the order $O(\alpha)$) to the desired solution of (2.1), while, unlike formulation (2.1), it is readily amenable for distributed implementation. A key insight known in the literature (see, e.g. [11, 118]) is that applying a conventional (centralized) gradient descent method on (2.2) precisely recovers the distributed gradient method proposed in [8]. In other words, it has been shown that the distributed method in [8] – that approximately solves (2.1) – actually converges to the solution of (2.2). This insight has been significantly explored in the literature to derive several distributed methods, e.g., [11, 12, 39]. The class of methods considered in this chapter also exploits this insight and therefore harnesses formulation (2.2) to carry out convergence analysis of the considered methods.

2.1.2 Algorithmic framework

The considered algorithmic framework subsumes several existing algorithms [39, 40, 41, 42, 43, 44], and it also provides algorithms that are either novel, or they have not been analyzed in the literature yet. Within the considered framework, each node i in the network maintains, $x_i^k \in R^s$, as its approximate solution to (2.1), where k is the iteration counter. We also associate a Bernoulli random variable z_i^k to each node i, that governs its communication activity at iteration k. If $z_i^k = 1$, node i communicates; if $z_i^k = 0$, node i does not exchange messages with neighbours. When $z_i^k = 1$, node i transmits x_i^k to all its neighbours $j \in \Omega_i$, and it receives x_j^k, from all its active (transmitting) neighbours.

The idea behind the quantities z_i^k is the following. It has been shown (see, e.g., [40]) that distributed methods to solve (2.1) and (2.2) exhibit certain "redundancy" in terms of the utilized communications. In other words, it is not necessary to activate all communication channels at all times for the algorithm to be convergent. Moreover, communication sparsification may lead to convergence speed improvements in terms of communication cost [40]. Communication sparsification and introduction of the z_i^k's leads to less expensive but inexact algorithmic updates. A proper design of the z_i^k's can lead to a positive resolution of the inexact-less expensive updates tradeoff; see, e.g., [40] for details.

We assume that the random variables z_i^k are independent both across nodes and across iterations. We denote by $p_k = Prob(z_i^k = 1)$, assumed equal across all nodes. The quantity p_k is a design parameter of the method. We consider different approaches to set up this parameter, namely we investigate constant, increasing and decreasing communication probabilities. These strategies for setting p_k are discussed further ahead. A large p_k corresponds to "less inexact" updates but also to lower communication savings. With the considered algorithmic framework, solution estimate update at node i is as follows:

$$d_i^k = -\Big[(M_i^k)^{-1}[\alpha \nabla f_i(x_i^k) + \sum_{j \epsilon \Omega_i} W_{ij}(x_i^k - x_j^k)\xi_{i,j}^k]\Big] \tag{2.9}$$

$$x_i^{k+1} = x_i^k - d_i^k \tag{2.10}$$

Here, α is a positive parameter that plays the role of step-size. The value of the parameter α differs for various input data sets (the particular values will be described later). $\xi_{i,j}^k$ is in general a function of z_i^k and z_j^k that encodes the communication sparsification; and M_i^k is a local second order information-capturing matrix, i.e. the Hessian approximation. We rely here on the class of proposed Newton-like methods, namely Distributed Quasi

Newton (DQN) methods [12]. This class of methods defines the way to incorporate second order information in distributed gradient methods. We will base our implementation on this result.

Vector $d_i^{(k)}$ contains information on node i's local function's gradient, node i's local function Hessian approximation, and an adjustment based on the mutual disagreement between node i's solution estimate and the estimates of its neighbours. Steps (2.9) and (2.10) are carried out in parallel by all nodes. Note that step (2.10) assumes that, prior to executing the step, each node i broadcasts its solution estimate $x_i^{(k)}$ to all its neighbours $j \in \Omega_i$ and receives x_j from all its neighbours $j \in \Omega_i$

We consider the following choices of the quantities $\xi_{i,j}^k$ and M_i^k. For $\xi_{i,j}^k$, we consider:

1. $\xi_{i,j}^k = 1$: no communication sparsification. This means that the nodes exchange their solution estimates with their neighbours constantly across the iterations;

2. $\xi_{i,j}^k = z_i^k \cdot z_j^k$ bidirectional communication sparsification (that is, node i includes node j's solution estimate in its update only if both i and j are active in terms of communications). More precisely, if node i is active during the k-th iteration, it sends its solution estimate to all its active neighbours, and also receives the solution estimates from them. On the other hand, if node i is inactive during an iteration, it does not receives nor sends any data;

3. $\xi_{i,j}^k = z_j^k$ (directed communications); that is, node i includes node j's solution estimate in its update whenever node j transmits, irrespective of node i being communication-active or not. This means that, if node i is active during an iteration, it sends its solution estimate to all its neighbours, regardless of their activity, but only receives values from its active neighbours. In the opposite case, if node i is inactive during an iteration, it does not send its solution estimate to other nodes, but receives data from its active neighboring nodes.

Regarding quantity M_i^k, two options can be identified:

1. $M_i^k = I$. This corresponds to first order methods, where local functions' Hessians are not evaluated;

2. $M_i^k = D_i^k$, where

$$D_i^k = \alpha \nabla^2 f_i(x_i^k) + (1 - W_{ii})I. \tag{2.11}$$

This corresponds to second order methods of DQN-type [39]. The method (2.9)-(2.10) corresponds to an inexact first order or an inexact second order method to solve (2.2) – and hence to approximately solve (2.1). The main source of inexactness is due to the sparsification ($\xi_{i,j}^k$'s). The bidirectional communication ($\xi_{i,j}^k = z_i^k \cdot z_j^k$) is appealing as it preserves symmetry in the underlying weight matrix, which is known to be a beneficial theoretical property. On the other hand, the bidirectional sparsification is also wasteful in that a node ignores the received message from a neighbor if its own transmission to the same neighbor is not successful (see formula (2.9)). With respect to the choice first versus second order method (the choice of M_i^k), the second order choice is computationally more expensive per iteration due to the Hessian computations; on the other hand, it can improve convergence speed iteration-wise.

In order to make the algorithm description clearer,a general form of the introduced algorithm can be described with a snippet of pseudocode (Algorithm 1).

Algorithm 1 Pseudocode for the proposed algorithmic framework

Require: at each node i: $\alpha > 0$; $\{W_{ij}\}_{j\in\Omega_i}$; $\{p_k\}_{k\geq 0}$
  repeat
    Each node i generates z_i^k and computes:
    M_i^k and $\xi_{i,j}^k$, $j \in \Omega_i$
    if $\xi_{i,j}^k = 1$ **then**
      Each node i receives x_j^k from node $j, j \in \Omega_i$
    end if
    Each node i updates x_i^k via (2.9) – (2.10)
  until a stopping criterion is met

2.1.3 Convergence analysis

In this section, a convergence analysis of the algorithm variant with unidirectional communications is carried out. This work has been published in [46]. In this section we assume the following choice of M_i^k and ξ_{ij}^k:

$$M_i^k = I, \quad \xi_{ij}^k = z_j^k. \tag{2.12}$$

To the best of our knowledge, except for a different estimation setting [44], this algorithm has not been studied before. The following assumptions are needed.

Assumption 2.1.1 *(a) Each function $f_i : \mathbb{R}^s \to \mathbb{R}$, $i = 1, ..., n$ is twice differentiable, strongly convex with strong convexity modulus $\mu > 0$, and it has Lipschitz continuous gradient with the constant L, $L \geq \mu$.*

(b) The graph G is undirected, connected and simple.

(c) The step size α in (2.2) satisfies $\alpha < \min\{\frac{1}{2L}, \frac{1+\lambda_n(W)}{L}\}$.

By Assumption 2.1.1, Ψ is strongly convex with modulus μ. Moreover, it has a Lipschitz gradient with the constant

$$L_\Psi := L + \frac{1 - \lambda_n(W)}{\alpha}. \tag{2.13}$$

Notice that Assumption 2.1.1 (c) implies that $\alpha < (1 + \lambda_n(W))/L$, which is equivalent to

$$\alpha < \frac{2}{L_\Psi}. \tag{2.14}$$

Let $x^k = ((x_1^k)^T, ..., (x_n^k)^T)^T$. We have the following convergence result for the first order method with unidirectional communications. [46]

Theorem 2.1 *Let $\{x^k\}$ be a sequence generated by Algorithm 1 and assume Assumption 2.1.1 holds. Then, the following results hold:*

(a) Assume that the sequence $\{p_k\}$ converges to one as $k \to \infty$. Then, the sequence of iterates $\{x^k\}$ converges to $x^\bullet$ in the expected error norm, i.e., there holds:

$$\lim_{k\to\infty} E[\|x^k - x^\bullet\|] = 0. \tag{2.15}$$

(b) Assume that the sequence $\{p_k\}$ converges to one geometrically as $k \to \infty$, i.e., $p_k = 1 - \delta^{k+1}$, for all k, Then, there holds:

$$E[\|x^k - x^\bullet\|] = O(\gamma^k), \tag{2.16}$$

where $\gamma < 1$ is a positive constant.

(c) Assume that $p_k \geq p_{min}$ for all k and for some $p_{min} \in (0, 1)$ and that the iterative sequence $\{x^k\}$ is uniformly bounded, i.e., there exists a constant $C_1 > 0$ such that $E[\|x^k\|] \leq C_1$, for all k. Then, there holds:

$$E[\|x^k - x^\bullet\|] \leq \theta^k \|x^0 - x^\bullet\| + (1 - p_{min})^2 C_2, \tag{2.17}$$

where $C_2 = \frac{2nC_1}{\alpha\mu}$.

Theorem 2.1 demonstrates that Algorithm 1 with sparsified communications converges with unidirectional communications. More precisely, as long as the sequence p_k converges to one, even arbitrarily slowly, the sequence $\{x^k\}$ converges to the solution of (2.2) in the expected error norm sense. When the convergence of p_k to one is geometric, we have that x^k converges geometrically, i.e., at a linear rate. Finally, when p_k stays bounded away from one, under the additional assumption that the sequence $\{x^k\}$ is uniformly bounded, the algorithm converges to a neighbourhood of the solution to (2.2), where the neighbourhood size is controlled by parameter p_{min} (the closer p_{min} to one, the smaller the error). This complements the existing results in [39] which concerns bidirectional communications.

Next, the proof of Theorem 2.1 will be carried out. To avoid notation clutter, let the dimension of the original problem (2.1) be $s = 1$. The proof relies on the fact that the method can be written as an inexact gradient method for minimization of Ψ. More specifically, it can be shown that the algorithm determined by (2.9) – (2.12) is equivalent to the following:

$$x^{k+1} = x^k - \alpha[\nabla\Psi(x^k) + e^k], \tag{2.18}$$

where $e^k = (e_1^k, ..., e_n^k)^T$ is given by

$$e_i^k = \frac{1}{\alpha}\sum_{j\in\Omega_i} W_{ij}(z_j^k - 1)(x_i^k - x_j^k) \tag{2.19}$$

and $e^k \in R^n$.Indeed, in view of (2.12), method (2.9)-(2.10) can be represented as

$$x^{k+1} = x^k - \alpha\nabla F(x^k) - (I - W_k)x^k, \tag{2.20}$$

where

$$F : \mathbb{R}^n \to \mathbb{R}, F(x) = \sum_{i=1}^{n} f_i(x_i), \tag{2.21}$$

$$[W_k]_{ij} = \begin{cases} W_{ij}z_j^k, & \text{if } \{i,j\} \in E, i \neq j, \\ 0, & \text{if } \{i,j\} \notin E, i \neq j, \\ 1 - \sum_{l\neq i}[W_k]_{il}, & \text{if } i = j. \end{cases} \tag{2.22}$$

Thus,

$$\begin{aligned} x^{k+1} &= x^k - \alpha(\nabla F(x^k) + \frac{1}{\alpha}(I - W_k)x^k \pm \frac{1}{\alpha}(I - W)x^k) \\ &= x^k - \alpha(\nabla\Psi(x^k) + \frac{1}{\alpha}((I - W_k)x^k - (I - W)x^k)). \end{aligned} \tag{2.23}$$

Therefore, for each component i the error is determined by

$$e_i^k = \frac{1}{\alpha}\Big(\sum_{j\in\Omega_i} W_{ij} z_j^k (x_i^k - x_j^k) - \sum_{j\in\Omega_i} W_{ij}(x_i^k - x_j^k)\Big), \tag{2.24}$$

and (2.19) follows.

Next we state and prove an important result. Here and further on, $||\cdot||$ denotes the vector 2-norm and the corresponding matrix norm.

Lemma 2.2 *Suppose that Assumption 2.1.1 holds. Then for each k we have*

$$||x^k - x^\bullet|| \le \theta^k ||x^0 - x^\bullet|| + \alpha \sum_{t=1}^{k} \theta^{k-t} ||e^{t-1}||, \tag{2.25}$$

where x^0 is the initial iterate and $\theta = \max\{1 - \alpha\mu, \alpha L_\Psi - 1\} < 1$.

Proof. Using (2.18) and the fact that $\nabla\Psi(x^\bullet) = 0$ we obtain

$$x^{k+1} - x^\bullet = x^k - x^\bullet - \alpha e^k - \alpha(\nabla\Psi(x^k) - \nabla\Psi(x^\bullet)). \tag{2.26}$$

Further, there exists a symmetric positive definite matrix B_k such that

$$\nabla\Psi(x^k) - \nabla\Psi(x^\bullet) = B_k(x^k - x^\bullet) \tag{2.27}$$

and its spectrum belongs to $[\mu, L_\Psi]$. Thus, we obtain

$$\|I - \alpha B_k\| \le \max\{1 - \alpha\mu, \alpha L_\Psi - 1\} := \theta. \tag{2.28}$$

Notice that the Assumption 2.1.1 (c) implies that $\theta < 1$ since (2.14) holds and $L \ge \mu$. Moreover, putting together (2.26) - (2.28), we obtain

$$\|x^{k+1} - x^\bullet\| \le \theta\|x^k - x^\bullet\| + \alpha\|e^k\| \tag{2.29}$$

and applying the induction argument we obtain the desired result. □

To complete the proof of parts (a) and (b) of Theorem 2.1, we need to derive an upper bound for $||e^k||$ in the expected-norm sense. In order to do so, it is needed to establish the boundedness of iterates x^k in the expected norm sense.

Lemma 2.3 *Let Assumption 2.1.1 hold, and consider the setting of Theorem 2.1 (a). Then, there holds $E[||x^k||] \leq C_x$ for all k, where C_x is a positive constant.*

Proof. The update rule (2.20) can be written equivalently as follows

$$x^{k+1} = W_k x^k - \alpha \nabla F(x^k). \tag{2.30}$$

Introduce $\widetilde{W_k} = W_k - W$, and rewrite (2.30) as

$$x^{k+1} = W x^k - \alpha \nabla F(x^k) + \widetilde{W_k} x^k. \tag{2.31}$$

Denote by x' the minimizer of F. Then, by the Mean Value Theorem, there holds

$$\begin{aligned} \nabla F(x^k) - \nabla F(x') &= \underbrace{\left[\int_0^1 \nabla^2 F(x' + t(x^k - x'))dt\right]}_{H_k}(x^k - x') \\ &= H_k(x^k - x') = H_k x^k - H_k x', \end{aligned} \tag{2.32}$$

and

$$x^{k+1} = (W - \alpha H_k)x^k + \widetilde{W_k} x^k + \alpha H_k x' - \alpha \nabla F(x'). \tag{2.33}$$

Note that $||H_k|| \leq L$, by Assumption 2.1.1. Also, note that $||W - \alpha H_k|| \leq 1 - \alpha\mu$, for $\alpha \leq \frac{1}{2L}$. Therefore, the following can be stated

$$\begin{aligned} ||x^{k+1}|| &\leq (1 - \alpha\mu)||x^k|| + \underbrace{\alpha(L||x'|| + ||\nabla F(x')||)}_{c'} \\ &\quad + ||\widetilde{W_k}|| \cdot ||x_k|| \\ &= (1 - \alpha\mu)||x^k|| + c' + ||\widetilde{W_k}|| \cdot ||x_k||. \end{aligned} \tag{2.34}$$

Next, $||\widetilde{W_k}||$ will be upper bounded. Note that

$$||\widetilde{W_k}|| \leq \sqrt{n}||\widetilde{W_k}||_1 \leq \sqrt{n}\sum_{i=1}^{n}\sum_{j=1}^{n}|[\widetilde{W_k}]_{ij}|. \tag{2.35}$$

Therefore,

$$||\widetilde{W_k}|| \leq 2\sqrt{n}\sum_{i=1}^{n}\sum_{j=1}^{n} W_{ij}(1 - z_j^k). \tag{2.36}$$

Taking expectation and using the fact that $E[z_j^k] = p_k$, for all k, it can be concluded that

$$E[||\widetilde{W_k}||] \leq \widetilde{C}(1 - p_k) \tag{2.37}$$

for some positive constant $\widetilde{C}$. Now, using independence of $\widetilde{W_k}$ and x_k, the following can be obtained from (2.34),

$$\begin{aligned} E[||x^{k+1}||] &\leq (1 - \alpha\mu)E[||x^k||] + C' + (1 - p_{k+1})\widetilde{C}E[||x^k||] \\ &= (1 - \alpha\mu + \widetilde{C}(1 - p_{k+1}))E[||x^k||] + C'. \end{aligned} \tag{2.38}$$

As $p_k \to 1$, i.e., $(1 - p_k) \to 0$, it is clear that, for sufficiently large k, there holds

$$E[||x^{k+1}||] \leq (1 - \frac{1}{2}\alpha\mu)E[||x^k||] + C'. \tag{2.39}$$

This implies that there exists a constant C_x such that $E[||x^k||] \leq C_x$, for all $k = 0, 1, ...$ □

Applying Lemma 2.3, the following result is obtained.

Lemma 2.4 *Suppose that the Assumption 2.1.1 holds and $E(\|x^k\|) \leq C_1$ for all k and some constant C_1. Then the error sequence $\{\|e^k\|\}$ satisfies*

$$E[||e^k||] \leq (1 - p_k)C_e, \tag{2.40}$$

for the positive constant $C_e = \frac{2n}{\alpha}(1 - p_{min})C_1$.

Proof. The proof follows straightforwardly from (2.19) and Lemma 2.3. Consider (2.24). Then, $|e_i^k|$ can be upper bounded as follows:

$$|e_i^k| \leq \frac{1}{\alpha} \sum_{j \in \Omega_i} w_{ij}|1 - z_j^k|2\|x^k\|. \tag{2.41}$$

This yields:

$$\|e^k\| \leq \|e^k\|_1 = \sum_{i=1}^{n} \frac{2}{\alpha} \sum_{j \in \Omega_i} w_{ij}|1 - z_j^k|\|x^k\|. \tag{2.42}$$

Taking expectation while using independence of z_j^k and x^k, and using $E(\|x^k\|) \leq C_1$; $\sum_{j\in\Omega_i} \leq 1$; and $E(|1-z_j^k|) = 1-p_k$, the result follows. □

Now, Theorem 2.1 can be proved as follows.

Proof of Theorem 2.1. We first prove part (a). Taking expectation in Lemma 2.2, and using Lemma 2.4, the following can be obtained

$$\begin{aligned} E[||x^k - x^\bullet||] &\leq \theta^k ||x^0 - x^\bullet|| + \alpha \sum_{t=1}^{k} \theta^{k-t} E[||e^{t-1}||] \\ &\leq \theta^k ||x^0 - x^\bullet|| + \alpha \sum_{t=1}^{k} \theta^{k-t} \cdot C_e(1-p_{t-1}). \end{aligned} \tag{2.43}$$

Next, applying Lemma 3.1 in [119], it follows that

$$E[\|x^k - x^\bullet\|] \to 0, \tag{2.44}$$

as we wanted to prove.

Let us now consider the part (b). Note that, in this case, we have that $1-p_k = \delta^{k+1}$, for all k. Specializing the bound in (2.43) to this choice of p_k, the following holds

$$E[||x^k - x^\bullet||] \leq \theta^k ||x^0 - x^\bullet|| + \alpha C_e \sum_{t=1}^{k} \theta^{k-t}\delta^t, \tag{2.45}$$

and using the fact that $s_k := \sum_{t=1}^{k} \theta^{k-t}\delta^t$ converges to zero R-linearly (see Lemma II.1 from [39]), we obtain the result.

Finally, we prove part (c). Here, we upper bound the term $(1-p_{t-1})$ in (2.43) with $(1-p_{min})$. For this case we obtain

$$\begin{aligned} E[||x^k - x^\bullet||] \leq \theta^k ||x^0 - x^\bullet|| \\ + (1-p_{min}) C_e \frac{1}{\mu}, \end{aligned} \tag{2.46}$$

which completes the proof of part (c).□

2.2 Implementation

This section is dedicated to the description of different aspects of implementation of the considered class of methods (see (2.9)-(2.10) and Algorithm 1 in Section 2.1). A distributed nature of an algorithm naturally demands a parallelized implementation. A serial algorithm is unable to handle the increase of the volume of data, and it cannot provide a satisfactory performance level. However, a serial implementation is a good starting point, and it serves as a basis for testing the correctness of the results of a parallel implementation. For this reason, we first developed a serial implementation in C programming language. A parallel implementation of the described algorithm was then developed using Message Passing Interface (MPI) also in C programming language. The need for efficient operations on vectors and matrices is effectively fulfilled by using the appropriate routines from the LAPACK [120] and BLAS [121] libraries on each node. The development of the parallel solution went through a number of phases, striving to find the most suitable algorithm variant, based on empirical evaluations. First, the algorithm was implemented for strongly convex quadratic cost functions, using an all-to-all communication strategy. When considering graph G, that represents the communication links among nodes, all the links from G need to be used here over iterations, as there is no communication sparsification. Moreover, the all-to-all communication strategy assumes that each node communicates with all the other nodes, regardless of the links in graph G. The structure of graph G is then used afterwards on each node, to actually utilize only the links defined by graph G. We also developed an implementation for logistic loss functions, that initially also used the all-to-all communication protocol, but in the next stage, we introduced communicators, in order to overcome the drawbacks of the initial approach. This ensured that a node in a graph only exchanged data with its neighbours, so the cost of communication was reduced. Finally, we applied different communication sparsification strategies, both unidirectional and bidirectional, for both first and second order methods. We also examined the possible implementations regarding the input data distribution, stopping criterion and presence or absence of second order update computation. These development stages and alternatives will be described below in details.

Before diving into the details of the solution development, a few concepts should be explained first. The term serial is used to denote a program that is written to run on a single machine, without parallelization. When writing a parallel program, that uses MPI, a few commonly used terms should also be explained. First, when a program runs in parallel, that means that a set of processes is working together, performing some task simultaneously. Each process has an identifier, referred to as the rank (or sometimes id) of the process. As our algorithm is working with a network of connected nodes, each

node in a network is represented by one process physically, so that we will use the terms process and node interchangeably, while explaining the implementation. The processes are physically assigned to different CPU cores on different machines in the cluster. Inside a set of processes, one can be identified as a master process. A master process always has a rank 0, and should be the one responsible for managing certain common tasks as input and output operations, data distribution among the processes and synchronization of results.

When referring to a set of processes that is assigned to a parallel program, a few important concepts should also be mentioned. First, the number of processes that the program uses is being set at the moment of starting the program. All the parallel code, was tested on a cluster environment. This means that we created scripts for submitting the programs to be executed on the cluster batching system, where the required number of cluster nodes and cores, and the time limit for the execution are being specified.

A communicator represents a set of processes that can communicate with each other. By default, when running a parallel program, all the processes belong to a global communicator called `MPI_COMM_WORLD`, so that they can communicate arbitrarily. However, a communicator can also be created by a program. In that case, it can be decided which processes are included in the newly created communicators. This can be very useful in the cases, when we need to split the communication lines in some ways. We will use these concepts while describing the implementation and experimentation phases. By introducing communicators, we ensure that a process communicates as defined by graph G, i.e. actually utilizing the communication links in graph G.

The algorithm is universal with respect to the cost functions f_i, and can be easily utilized for different applications. We describe below the implementations for two different cases: quadratic and logistic loss functions. These implementations basically differ only in the gradient and Hessian calculations. Therefore, another cost functions could be easily accommodated, by replacing the calculations for gradient and Hessian.

2.2.1 Implementing the algorithm for strongly convex quadratic cost functions

Consider the implementation for strongly convex quadratic cost functions first. This means that, each node is assigned a function:

$$f_i(x) = \frac{1}{2}(x - b_i)^\tau a_i (x - b_i), \tag{2.47}$$

where $x \in \mathbb{R}^s$ represents the optimization variable. The input data are given as an $s \times s$ matrix a_i, and an s sized vector b_i on each node.

As the algorithm is iterative, it should converge after some required number of iterations. For quadratic cost functions, we used a fixed number of iterations. During the iterations, the Gradient $\nabla f_i(x_i^{(k)})$, the Hessian approximation $M_i^{(k)}$, and the solution update x are being recalculated as:

$$\nabla f_i(x_i^{(k)}) = a_i(x_i - B_i) \tag{2.48}$$

$$M_I^{(k)} = \alpha \nabla^2 f_i(x_i^{(k)}) + (1 - W_{ii})I \tag{2.49}$$

$$x_i^{(k+1)} = x_i^{(k)} - \epsilon (M_i^{(k)})^{-1} \Big[\alpha \nabla f_i(x_i^{(k)}) + \sum_{j \in \Omega_i} W_{ij}(x_i^{(k)} - x_j^{(k)}) \Big] \tag{2.50}$$

The constant value ϵ in (2.50) is set to 1 in the implementation. The value α, representing the step size of the algorithm is set as $\frac{1}{200L}$, where L is the largest eigenvalue, among the eigenvalues of parts of input matrix a. The values w_{ij} correspond to the appropriate elements of the weight matrix.

Input data preparation

The implementation can be tested with arbitrary input data. We want to have four different binary files at the input. The first two of them are related to the input matrix and vector, and the third and fourth define the graph structure. Here, we use randomly generated values. The data generation process is not part of the implementation of the algorithm. Therefore, different data generation approaches may be utilized.

As already explained, each node should have an $s \times s$ sized matrix a_i, and an s sized vector b_i. The input binary files contain a global matrix and a global vector, i.e. all the submatrices and subvectors for the nodes. This means, that the binary file containing the input matrix should contain $s \times s \times n$ entries, and the binary file with the input vector contains $s \times n$ entries, where n is the number of nodes. This means that the first chunk of size $s \times s$ from the matrix goes to the first process as its own matrix, and the first chunk of vector of size s goes to the first process as its own vector. The same holds for the other chunks.

The graph of nodes may also be created randomly, but ensuring that it remains connected. The final binary files should contain the adjacency matrix and the degree vector for the nodes separately. The adjacency matrix contains the information about the existing edges in the connected graph of nodes, i.e. it contains value 1, when $W_{ij} > 0$, for $i \neq j$. The degree vector is used to obtain the degrees for each node in the graph. This data is the

basis for calculating the weight matrix W for the communicating nodes. This way, the nodes are aware of the list of their neighbours, as well as of the degree of each node.

One way to generate a graph randomly is as follows:

- For a specified number of nodes n, random points for vertices are generated.
- For each pair of vertices with mutual distance smaller than some specified value r, create an edge. The value r is specified as $r = \sqrt{\frac{log(n)}{n}}$
- An adjacency matrix (containing only values 0 an 1 - indicating the existence/lack of an edge) is created and written to a file. A degree vector is also created and written to a file. It contains the degree i.e. number of neighbours for each node.

The serial implementation for quadratic cost functions

We develop a serial implementation in C first, and then use it later as a starting point for parallelization. This enables the comparison between the two implementations, not just performance related (as we naturally expect better performance with parallelization), but also related to the correctness of the resulting values. When referring to correctness, we naturally do not expect some precisely defined value. We expect a result that converged "close" to the solution. As we have 3 binary input files, the first task is to read the content of the files, and create the needed data structures. This is a very simple, standard approach, where we allocate the necessary data structures and read the contents of the binary files into them. The next step is to create the weight matrix W, which is also a straightforward task. Finally, we calculate the gradient, the Hessian and the solution update iteratively as (2.48)-(2.50).

We use routines from LAPACK and BLAS libraries for operations on matrices and vectors, as `cblas_dgemv` for matrix-vector product, `cblas_daxpy` for matrix addition and subtraction, `LAPACKE_dgetrf` for LU factorization of a matrix, and `LAPACKE_dgetri` for computing the matrix inverse.

The parallel implementation for quadratic cost functions

The serial implementation can now be adapted, in order to introduce the parallelization. As usually, the first task to implement is the input data reading and preprocessing to the desired form.

The process of reading the data is the same as for the serial implementation, with the only difference that the master process reads the data and then scatters the input matrix a and vector b to all the processes equally. The scattering of the data is done in the simplest way here, by calling the MPI method `MPI_Scatter`. First, the master process scatters the whole matrix to chunks of size $s \times s$ among the processes (the first chunk stays on process 0, the next one goes to 1 and so on). Each chunk is stored in a local matrix on a process. A separate call to `MPI_Scatter` scatters the input vector in the same manner. The communicator used for scattering is `MPI_COMM_WORLD`, by default. This means that all the processes belong to the default global communicator and can communicate mutually. Any call to MPI collective functions involves all the existing processes. For more details, see the GitHub repository [1].

After this point, each process has the corresponding part of the global matrix a, and of the global vector b. We refer to these values as variable `A` for the matrix and variable `B` for the vector in the code. The symmetric weight matrix `W` is being created and broadcast to the other processes by the master process, by calling the function `MPI_Bcast`.

The parallel implementation naturally differs from the serial one among several dimensions. The program is working with a set of processes, and each of them operates on its own memory space and communicates with other processes by message passing. For that reason, it is necessary to know the number of neighbours and the array of particular neighbours ranks for each process. This way, we can ensure that each process will be aware of which processes it needs to communicate with. A convenient and easy solution here is to maintain an integer variable, containing the number of neighbours for the node, and a vector, containing the ranks of the neighbours. These data can be easily acquired for node i, by observing all the indices j from W, where $W_{ij} > 0$, and $i \neq j$. Listing 2.1 shows the definition of these structures. Each process executes this snippet of code.

The main loop of the algorithm has a fixed number of iterations k. During the iterations, the Gradient $\nabla f_i(x_i^{(k)})$, the Hessian approximation $M_i^{(k)}$, and the solution update x are being recalculated as in equations (2.48)-(2.50).

Listing 2.1: Creating the data structures that contain data about neighbours, C with MPI

```
int *my_neighbours=calloc(n, sizeof(int));
int my_neighbours_count=0;
for(i=0;i<n;++i)
    if(WMatrix[my_rank*n+i]!=0.0 && my_rank!=i){
            my_neighbours[my_neighbours_count]=i;
            my_neighbours_count++;
    }
```

Listing 2.2 represents a snippet of code, where the gradient is being calculated. It is based on calls to LAPACK and BLAS routines, in order to ensure the best performance of the computations. Each process has its own vector for solution x, and its own data chunk. First, we create a copy of the local solution, variable `X` in the code. Then we subtract `B` from `copyX`, by calling `cblas_daxpy`. The result is stored in `copyX`. Finally, we multiply that result with the matrix `A` and store the result in a variable called `Grad`.

Listing 2.2: Calculating the gradient for quadratic functions, C with MPI

```
LAPACKE_dlacpy(LAPACK_ROW_MAJOR,'A',1,s,X,s,copyX,s);
cblas_daxpy(s,-1,B,1,copyX,1);
cblas_dgemv(CblasRowMajor,CblasNoTrans,s,s,1,A,s, copyX,1,0,Grad,1);
```

Next, each process calculates its Hessian inverse approximation. The code that calculates the Hessian inverse approximation is displayed in Listing 2.3. The variable `GMatr` initially holds the values from the local matrix `A`, so its dimension is $s \times s$. The matrix `AWeight` is again initialized as an identity matrix, its dimension is $s \times s$, as we work on one data chunk on a process.

The matrix `Wii` is also a simple diagonal matrix of size $s \times s$ now. It contains the value W_{ii} on the diagonal, where i is equals to the process rank. We add the `GMatr` multiplied by `alpha` to `AWeight`, and then subtract `Wii` from it. Finally, we create the matrix inverse `AWeightInv` using the appropriate LAPACKE routines.

Listing 2.3: Calculating the Hessian inverse for quadratic functions, C with MPI

```
LAPACKE_dlacpy(LAPACK_ROW_MAJOR,'A',s,s,A,Dim,GMatrix,s);
for(i=0;i<s;++i)
    AWeight[i*s+i]=1.0;
cblas_daxpy(s*s,alpha,GMatrix,1,AWeight,1);
cblas_daxpy(s*s,-1,Wii,1,AWeight,1);
LAPACKE_dlacpy(LAPACK_ROW_MAJOR,'A',s,s,AWeight,s,AWeightInv,s);
LAPACKE_dgetrf(LAPACK_ROW_MAJOR,s,s,AWeightInv,s,pivotArray);
LAPACKE_dgetri(LAPACK_ROW_MAJOR,s,AWeightInv,s,pivotArray);
```

At this point, the processes need to exchange their values of resulting vectors with their neighbours, by exchanging messages, in order to ensure the proper access to neighbours solution updates. The simplest way to achieve this is to use the all-to-all communication protocol, by calling the `MPI_Allgather` function. This ensures that all the results all collected on all processes. However, as a process should consume only the results from its neighbours, the vector of neighbours enables to take into account only the needed values. This can be achieved as follows: node i checks in a loop for each value $j \in$

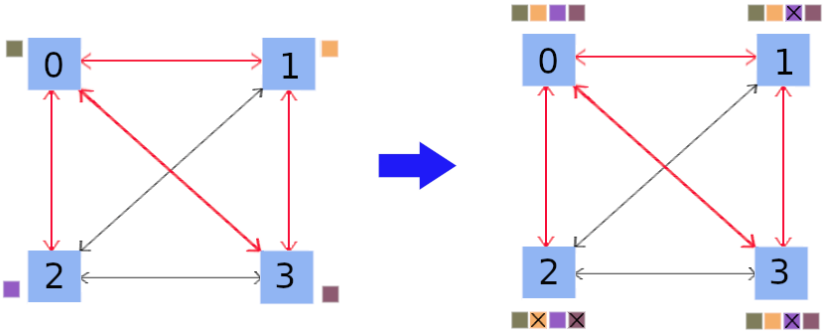


Figure 2.1: The all-to-all communication protocol

$\{0..., n-1\}, j \neq i$ weather node j is its neighbour; if j is its neighbour, then it includes the received data during calculating the update as $\sum_{j\in\Omega_i} w_{ij}(x_i^{(k)} - x_j^{(k)})$; otherwise, it skips node j. Fig. 2.1 displays the all-to-all communication protocol. The red arrows represent the communication links defined by matrix `W`. The black arrows are communication links, that are not of interest for the algorithm, but are still physically utilized. The right side on Fig. 2.1 shows that each node (i.e. process) contains all the data, from all nodes, when the communication is finished. The crossed out data chunks on nodes are those that are not being used by a node, as they came from non-neighbors. After recalculating the update, the final result for the current iteration is being evaluated by incorporating the gradient and the Hessian inverse approximation to the formulation as showed in (2.50). When the iteration counter exceeds the maximal number of iterations, the master process gathers the parts of the results from the other processes, in order to provide the final value.

Listing 2.4: The data exchange and solution update, C with MPI

```
MPI_Allgather(X,s,MPI_DOUBLE,Xremote,s,MPI_DOUBLE,MPI_COMM_WORLD);
for(i=0;i<my_neighbours_count;++i){
        j=my_neighbours[i];
        LAPACKE_dlacpy(LAPACK_ROW_MAJOR,'A',1,s,X,s,Xdiff,s);
        cblas_daxpy(s,-1,Xremote+j*s,1,Xdiff,1);
        cblas_daxpy(s,WMatrix[my_rank*n+j],Xdiff,1,zero,1);
        cblas_daxpy(s,1,zero,1,NablaPsi,1);
}
cblas_daxpy(s,alpha,Grad,1,NablaPsi,1);
cblas_dgemv(CblasRowMajor,CblasNoTrans,s,s,1.0,AWeightInv,s,
            NablaPsi,1,1.0,sDirection,1);
cblas_daxpy(s,-1,sDirection,1,X,1);
```

Listing 2.4 shows the data exchange process and the computation of the solution update. As already mentioned, the call to `MPI_Allgather` ensures that all the processes have the

solutions from all other processes in the variable `Xremote`. Each process then executes a for loop, for each neighbour. The rank of the current neighbour is `j`, and it can be obtained from the array called `my_neighbours`, created initially. Now, a process subtracts the `j`-th process solution from its own value `X` and multiplies it with the appropriate value from `WMatrix`. The result is stored in a vector called `zero` (it initially contains all zeros). This vector is then added to `NablaPsi`. In other words, `NablaPsi` contains the sum of differences from (2.50). After the loop, the gradient multiplied with the step size `alpha` is being added to `NablaPsi`. We then multiply it with `AWeightInv` using `cblas_dgemv`, and store the result in `sDirection`. Finally, the solution update is being computed by subtracting the `sDirection` from `X`. This implements the formula (2.50). Note that we use the value $\epsilon = 1$.

This completes the parallel implementation of the algorithm for quadratic cost functions. All these code snippets, including the gradient, Hessian inverse and solution estimation update, are part of the main loop of the algorithm. After a predefined number of iterations, the algorithm finishes its execution, and the master process gathers the solution estimate from its neighbours. The tests and results of tests, performed for this implementation will be explained in the section dedicated to experimentation.

2.2.2 Implementing the algorithm for logistic loss functions

After implementing the algorithm for a very simple setup with strongly convex quadratic cost functions, the idea is to create an implementation for an another, commonly used approach, in order to demonstrate the possibility for wide applications. Therefore, the algorithm was implemented for logistic loss functions with L2 regularization (that corresponds to distributed learning of a linear classifier), that can be defined as:

$$f_i(x) = \sum_{j=1}^{J} \mathcal{J}_{logis}(b_{ij}(x_1^\top a_{ij} + x_0)) + \frac{\tau}{n}||x||^2. \tag{2.51}$$

Here, $x = (x_1^T, x_0) \in \mathbb{R}^{s-1} \times \mathbb{R}$ represents the optimization variable and τ is the penalty parameter. The input values are $a_i \in \mathbb{R}^{s-1}$ and $b_i \in \mathbb{R}$. Each node i has J data samples. Also, there holds that:

$$\mathcal{J}_{logis}(z) = log(1 + e^{-z}), \tag{2.52}$$

so that, we have:

$$f_i(x) = \sum_{j=1}^{J} log(1 + e^{-b_{ij}(x_1^\top a_{ij} + x_0)}) + \frac{\tau}{n}||x||^2. \tag{2.53}$$

As the algorithm is reusable for different cost functions, the principles regarding the implementation for logistic loss functions are the same as for quadratic cost functions. However, there are naturally some changes, that we illustrate. First, there are two dimensions now, R and s. R represents the total number of rows i.e. samples in the global input matrix a and the number of elements in the global input vector b. The value s is the number of columns i.e. features in matrix a, and there holds that $s + 1$ is the dimension for the solution vector x. The main loop of the algorithm has also a predefined number of iterations, at first. Later, we introduce a stopping criterion, that stops the algorithm if the result is satisfactorily close to the solution. This is explained in more details later in this section. Inside the main loop, each process calculates the gradient, the Hessian and the solution update with data exchange, as:

$$\nabla f_i(x_i^{(k)}) = \sum_{j=1}^{J} \frac{e^{b_{ij}(x_1^\top a_{ij} + x_0)}}{1 + e^{b_{ij}(x_1^\top a_{ij} + x_0)}}(-b_{ij}a_{ij}, -b_{ij}) + \frac{\tau}{n}x_i \tag{2.54}$$

$$M_I^{(k)} = \alpha\nabla^2 f_i(x_i^{(k)}) + (1 - W_{ii})I \tag{2.55}$$

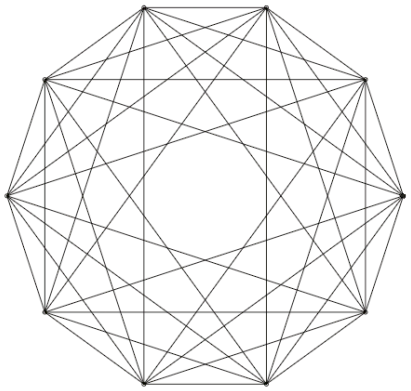
$$x_i^{(k+1)} = x_i^{(k)} - \epsilon(M_i^{(k)})^{-1}\Big[\alpha\nabla f_i(x_i^{(k)}) + \sum_{j\in\Omega_i} W_{ij}(x_i^{(k)} - x_j^{(k)})\Big] \tag{2.56}$$

The step size parameter α is now being set, according on the data set used. This can be determined experimentally. The dimension of the data set influences the value of this parameter. We will define this value for each data set that we use for the experiments in the corresponding section of the thesis. The value of ϵ remains 1, as before.

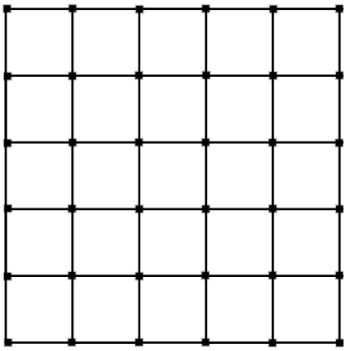
Input data preparation

The input data format is the same here as it was for quadratic cost functions. We expect binary input files for the matrix and vector, and also for the adjacency matrix and degree vector. We expect an input matrix of size $R \times s$, where $r = \frac{R}{n}$ samples of data goes to a single process. Similarly, we expect an input vector b of size R, and each process gets a portion of size $r = \frac{R}{n}$. The input data used for this example will not be synthetic randomly generated data. Instead of that, we use real data sets, obtained from public data repositories. These data sets will be explained in detail later.

The graph structure for the nodes connections is generated, but this time, we do not use random graphs any more. We use regular and grid graphs instead. Fig. 2.2 illustrates examples for these graph types. These graph structures are generated by using auxiliary scripts. We perform tests for both types of graphs and explore the differences in performance, in Section 2.3.3, dedicated to experimentation.



(a) An example for an 8-regular graph, with 10 nodes



(b) An example for a grid graph, with 36 nodes

Figure 2.2: Examples for graph types used during the evaluation

The first type of graphs that we use is d-regular undirected, simple graph type. It means that there are no loops or double edges inside the graph and that each node has d neighbours. The construction of d-regular graphs can be explained in the following way. For 8-regular graphs, for example, for each number of nodes n, we construct an 8-regular graph starting from a ring graph with nodes $1, 2, ..., n$ and then adding to each node i the links to the nodes $i-4, i-3, i-2$, and $i+2, i+3$, and $i+4$, where the subtractions and additions here are modulo n. The same principle was also used for other values of d, for d-regular graphs used during the evaluation.

The second type of graphs we used is a grid graph type. For this type of graph, we used only such numbers of edges n, that are divisible by 4. All of the nodes will be of degree $2, 3$ or 4, depending on their position in the grid. We are only interested in rectangular grid graphs. For that reason, we choose those numbers of nodes, that satisfy this demand. We are striving to have square grid graphs where possible. For example, for $n = 36$, we generate a 6×6 grid. In the cases when this is not achievable, we are working with rectangular variants $t \times v$, wile trying to keep t and v as close as possible. For example, for $n = 48$, we do not want to create a 12×4, or 24×2 grid. Instead of that, we are working with a 8×6 grid structure.

The serial implementation for logistic loss functions

Our first implementation is a serial one, following the same principles as with quadratic cost functions. However, for the logistic loss functions, we will have several adjustments for the parallel algorithm implementation, in order to improve the performance. This incremental development maintains the opportunity to compare the performance of dif-

ferent versions of the algorithm, that use different strategies to solve the same problem. Therefore, the performance of the algorithm can be enhanced.

The first task is the input data reading and processing to the desired form. The master process reads the content from the binary input files, in the same manner as for quadratic cost functions. After that, the main loop performs the updates iteartively, as before. We only need to change the way of computing the gradient and the Hessian inside the main loop, in order to implement (2.54) and (2.55).

The initial parallel implementation for logistic loss functions

The parallel implementation of the algorithm for logistic loss functions can be straightforwardly derived by combining the serial implementation for logistic loss functions with the parallelization for quadratic cost functions. The first task, the input data reading and scattering is the same as for quadratic cost functions. The only difference is that we expect $R \times s \times n$ entries for the input matrix (instead of $n \times s \times s$) and $R \times n$ entries for the input vector (instead of $s \times n$). Similarly, when scattering the data, the size of data chunk for the matrix is $r \times s$ (instead of $s \times s$) and r for the vector (instead of s). The process of creating and distributing the weight matrix is completely identical as for the quadratic cost functions. A vector that contains the list of neighbours for the nodes is also present here, and defined in the same manner as for quadratic cost functions (See Listing 2.1).

The main loop of the algorithm also computes the gradient, the Hessian inverse approximation and the solution update. Listing 2.5 shows the code snippet that calculates the gradient inside the main loop. Here, each process has an $r \times s$ sized matrix `A` and an r sized vector `B`. Also, each node is aware of its neighbours.

The first task is to allocate some auxiliary data structures: a vector `Sum`, that represents the sum in (2.54), a vector for the first s elements from the solution update (variable `ww`) i.e. x_1 and the value x_0 (represented as variable `vv`), i.e. the last element from the current solution update.

Then, we need to implement a loop, that iterates for each row of data. As we have R rows and n nodes, this means that each node needs to handle $r = \frac{R}{n}$ rows. This, of course means, that we assume that the number of rows in the input data is divisible by the number of nodes. This aspect will be discussed further later.

For each row, we calculate the coefficient `coeff`, that is the following: the dot product of the current row from input matrix `A` (corresponds to a_{ij} in the formula) and the variable

ww, or x_1 in the formula, with vv (or x_0 in the formula) added and multiplied by the appropriate element from the input vector B (multiplied by -1), that corresponds to b_{ij} in the formula. Then, we obtain the value of the expression $\frac{e^{coeff}}{1+e^{coeff}}$, as variable coeff2. Then, it remains to calculate the vector called subMatr here. It contains the product of the current row of A and current element of B, multiplied by -1, on the first s positions. The last position simply contains the current element from B, multiplied by -1. The final step inside the loop is to multiply the created subMatr by the coefficient coeff and add the result to the Sum vector.

Listing 2.5: Calculating the gradient in parallel for logistic loss functions, C with MPI

```
double *Sum=calloc(s+1,sizeof(double));
double *ww=calloc(s,sizeof(double)),vv=X[s];
LAPACKE_dlacpy(LAPACK_ROW_MAJOR,'A',1,s,X,s,ww,s);
for(l=0;l<r;l++){
        double dot=cblas_ddot(s, A+(l*(s)),1,ww,1);
        double coeff=(dot+vv)*(-Bdata[l]);
        double coeff2=exp(coeff)/(1+exp(coeff));
        for(h=0;h<s;h++)
            subMatr[h]=A[l*s+h]*(-Bdata[l]);
        subMatr[s] = -Bdata[l];
        cblas_daxpy(s+1,coeff2,subMatr,1,Sum,1);
}
LAPACKE_dlacpy(LAPACK_ROW_MAJOR,'A',1,s+1,Sum,s+1,Grad,s+1);
cblas_daxpy(s+1,lambda_penal,X,1,Grad,1);
```

Finally, we copy the vector Sum to the variable Grad, and then add the current solution update, multiplied by lambda_penal to it. The lambda_penal variable represents the value $\frac{\tau}{n}$ from the formula, i.e. the regularization parameter. Instead of calculating this value, we use a fixed value lambda_penal=0.03. This completes the calculation of the gradient.

Listing 2.6 shows the computation of the Hessian. The values ww and vv are predefined before the gradient calculation, so they can be reused here, as they play the same roles as before. We also use an auxiliary variable, a sum matrix of size $(s+1)\times(s+1)$ here. Again, the coefficient coeff is being calculated in the same way as before. The variable coeff2 is now different and is being calculated as $\frac{e^{coeff}}{(1+e^{coeff})^2}$. We again compute the variable subMatr on the same manner as for the gradient. We then call the routine cblas_dger to multiply the vector subMatr with the transpose of the same vector. The result is being stored in the variable tmp. Finally, we add to the sum matrix SumMatrix the created matrix tmp, multiplied by coeff2.

The matrix eye at the end of code snippet is again an $(s+1) \times (s+1)$ diagonal matrix, as the final matrix GMatrix is of size $(s+1) \times (s+1)$ and we need to add the diagonal matrix multiplied with lampbda_penal to the result. After calculating the gradient and Hessian this way, each process has its own local variable Gradient and its own local variable GMatrix, that are necessary for further computations.

Listing 2.6: Calculating the Hessian approximation in parallel for logistic loss functions, C with MPI

```
double *SumMatrix=calloc((s+1)*(s+1),sizeof(double));
for(l=0;l<r;l++){
        double dot=cblas_ddot(s,A+(l*s),1,ww,1);
        double coeff=(dot+vv)*(-Bdata[l]);
        double coeff2=exp(coeff)/(1+exp(coeff))/(1+exp(coeff));
        double *subMatr=calloc(s+1,sizeof(double));
        double *tmp=calloc((s+1)*(s+1),sizeof(double));
        for(h=0;h<s;h++)
            subMatr[h]=A[l*s+h]*(-Bdata[l]);
        subMatr[s]=(-Bdata[l]);
        cblas_dger(CblasRowMajor,s+1,s+1,1.0,
                    subMatr,1,subMatr,1,tmp,s+1);
        cblas_daxpy((s+1)*(s+1),coeff2,tmp,1,SumMatrix,1);
}
LAPACKE_dlacpy(LAPACK_ROW_MAJOR,'A',s+1,s+1,SumMatrix,s+1,GMatrix,s+1);
cblas_daxpy((s+1)*(s+1),lambda_penal,eye,1,GMatrix,1);
```

The inverse of the Hessian is being calculated in the same way as for quadratic cost functions (see Listing 2.3), with two differences. First, we calculate the GMatrix as shown in Listing 2.6, so we do not need the first line in Listing 2.3 at all. The second thing is that we have the dimension $s+1$ everywhere, instead of s on Listing 2.6. The data exchange between the processes and the solution update calculation is the same as for the quadratic cost functions (see Listing 2.4 and Fig. 2.1), with the difference that we should use $s+1$ instead of s for the dimension now. This completes the initial parallel implementation. Now, we want to explore the possibilities for algorithm enhancement.

Adapting the parallel implementation to arbitrary input data set sizes and weight matrix distribution

The implementation assumed that we have an input data size R as the number of samples, that is divisible by the number of nodes, for now. Then we just defined $r = \frac{R}{s}$ as the number of samples that goes to each process. This is acceptable, as far as we are working with synthetic data, so we can generate data of any size. However, we want to test our

implementation for logistic loss functions on real, publicly available data sets. At this moment, fulfilling this request regarding the dimension is unreasonable. For this reason, the algorithm was modified here, in order to make it able to deal with data set sizes not divisible by the number of processes. In such cases, the overage is assigned to the master process, while also taking care of allocating more space on it. Also, during the computations, the algorithm needs to take care of the possibly larger dimension on the master process.

The master process reads the input matrix and the vector again from binary files, as before. However, the function `MPI_Scatterv` is used, instead of `MPI_Scatter`, when an overage exists. This function enables to set arrays for send counts and displacements for each process, ensuring the proper data distribution. Listing 2.7 shows the code snippet for the data distribution process. The variable `rem` is the remainder when dividing R by n.

Listing 2.7: Scattering the input data with a possible overage, C with MPI

```
if(*rem!=0){
        int *sendcounts=calloc(n,sizeof(int));
        int *displs=calloc(n,sizeof(int));
        sendcounts[0]=s*(r+*rem);displs[0]=0;
        for(int p=1;p<n;p++)
            sendcounts[p]=s*r;displs[p]=p*s*r+s**rem;
        MPI_Scatterv(Adata,sendcounts,displs,MPI_DOUBLE,A,
                sendcounts[my_rank],MPI_DOUBLE,0,MPI_COMM_WORLD);
        sendcounts[0]=r+*rem;displs[0]=0;
        for(int p=1;p<n;p++)
                sendcounts[p]=r;displs[p]=r*p+*rem;
        MPI_Scatterv(Bdata,sendcounts,displs,MPI_DOUBLE,B,
                sendcounts[my_rank],MPI_DOUBLE,0,MPI_COMM_WORLD);
}
else{
        MPI_Scatter(Adata,r*s,MPI_DOUBLE,A,r*s,MPI_DOUBLE,0,
                    MPI_COMM_WORLD);
        MPI_Scatter(Bdata,r,MPI_DOUBLE,B,r,MPI_DOUBLE,0,
                    MPI_COMM_WORLD);
}
```

The overage `rem` is actually a number of rows, where each row contains again s elements. If there is an overage, we call `MPI_Scatterv`. Otherwise, we call `MPI_Scatter` as before. Some data structures need to be prepared for the call to `MPI_Scatterv`. Concretely, we need an array of send counts an array of displacements. The array `sendcounts` contains the number of elements that goes to each process, as the name suggests. It contains

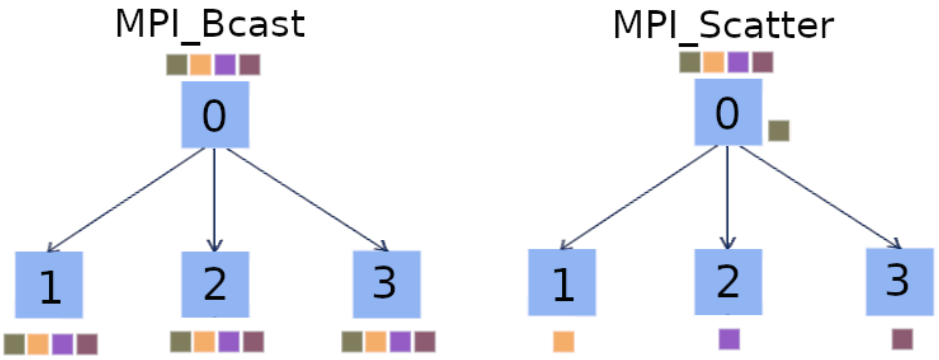


Figure 2.3: The reasoning behind MPI_Bcast and MPI_Scatter

$s \times r + rem$ for the master process, and $s \times r$ for the rest of the processes, regarding the input matrix. The array `displs` contains the starting positions for the chunks to be sent to processes. For the matrix, for the master process, the starting position is 0, for the other processes it is $p \times (s \times r) + s \times rem$ (note that we are representing matrices as one dimensional arrays), where p is the id of a process. When working with the input vector, the same principles hold. The send count for the master is then $r + rem$, and r for the other processes, where the displacement is 0 for the master, and $p \times r + rem$ for the other processes, where p is the id of a process.

This adaptation enables to work with arbitrarily sized data sets. However, we should also keep in mind the loops, where we iterate over the rows in a local data chunk. This kind of loop is present when calculating the gradient and the Hessian. In those places, we will have a loop of the form `for(l=0;l<r+(my_rank==0)*rem; l++)`, instead of `for(l=0;l<r;l++)`. This is a very simple C-style solution: if a rank of the process is zero, it is the master process, so the expression `my_rank==0` is true, i.e. it has value 1. When multiplied by rem, we have exactly what we wanted. For the rest of the processes, this will be $0 \times rem$. The weight matrix `W` is being created as already described. But now, instead of broadcasting the whole matrix `W` to all processes, we can just scatter it, so each process can have its own part of the matrix. The reason behind this is that the processes do not actually need to know the weight values for the processes they are not connected with. So, we replace the call to:

```
MPI_Bcast(WMatrix, n*n, MPI_DOUBLE, 0, MPI_COMM_WORLD)
```

with the call to:

```
MPI_Scatter(WMatrix,n,MPI_DOUBLE,myWMatr,n,MPI_DOUBLE,0,MPI_COMM_WORLD).
```

This means that each process obtains a vector of n elements `myWMatrix`, that contains the weight values for the branches between that node and its neighbours (the value 0

means that the node is not connected to the particular process). It is a convenient fact that the ranks of processes correspond to indices in the resulting vector. The properties of the mentioned functions `MPI_Bcast` and `MPI_Scatter`, can be easily demonstrated graphically, as shown on Fig. 2.3.

Adapting the parallel implementation regarding the stopping criterion

The problem of stopping criterion is a place for reevaluation at this point. We had a fixed number of iterations initially, and it turned out mainly inefficient. For different data sets, the algorithm needs different number of iterations in order to approach the desired values. Defining a large value for the iterations number can lead to overkill in cases when that large number is not needed practically.

For that reason, we let the algorithms run until $||\nabla\Psi(x^k)|| \leq \epsilon$, where $\epsilon = 0.01$. Note that the gradient $\nabla\Psi(x^k)$ is not computable by any node in a distributed graph G in general. In our implementation $\nabla\Psi(x^k)$ is maintained by the master node. We calculate this value at the end of every iteration.

Listing 2.8: Introducing the stopping criterion, C with MPI

```
double *curRes=calloc(s+1,sizeof(double));
double *GradGlob=calloc((s+1)*n,sizeof(double));
for(i=0;i<s+1;i++){
        curRes[i]=(1-myWMatrix[my_rank])*X[i];
        for(j=1;j<=my_neighbours_count-1;j++){
                int my_neighbours_rank=my_neighbours[j-1];
                curRes[i]+=-myWMatrix[my_neighbours_rank]*
                        Xremote[my_neighbours_rank*(s+1)+i];
        }
        curRes[i]+=alpha*GradOld[i];
}
MPI_Gather(curRes,s+1,MPI_DOUBLE,GradGlob,s+1,MPI_DOUBLE,0,
           MPI_COMM_WORLD);
if(my_rank==0)
        euclidean_norm=cblas_dnrm2((s+1)*n,GradGlob,1);
MPI_Bcast(&euclidean_norm,1,MPI_DOUBLE,0,MPI_COMM_WORLD);
if(k<iter && euclidean_norm<epsilon ){
        stop=my_rank;
        continue;
}
```

Listing 2.8 shows the code snippet, that should be inserted into the main loop, before the solution update calculation. The master process gathers the values and determines the

euclidean norm of the gathered vector (containing subvectors from all processes). First, each process calculates its current solution (the variable `curRes`). This means that it first takes its own solution update and multiplies it with $1 - W_{ii}$. Then for each of its neighbours, it finds the rank of the neighbour. The loop starts from 1 here, and we are searching for the neighbour on position $j - 1$. This is because the first neighbour's rank is on position 0, and so on. Then, we subtracts the solution obtained from that neighbour and multiplied by the appropriate value from `myWMatr`, from the current solution. We use the neighbours rank to obtain the needed element from the weight matrix. Similarly, in order to access the right solution update in `Xremote`, we use the rank of the neighbour. Finally, we add the value of the previously calculated gradient, multiplied by the step size `alpha`. We use the value of the previously calculated gradient, i.e. the value that corresponds to the previous iteration. Because of this, before calculating the gradient, we need to copy its current value to `GradOld`. This way, we always have the value of the gradient from the previous iteration.

Finally, the master process gathers these current results into the `GradGlob` variable. Then it calculates the euclidean norm of it, by calling the routine `cblas_dnrm` and broadcasts it to all the processes. At the moment, when the euclidean norm becomes smaller than ϵ (and we are not in the first iteration), the main loop should be terminated. This can be achieved by broadcasting the euclidean norm to all processes, where each process sets its variable `stop` to its rank value, and continues to the top of the main loop.

Listing 2.9: Stopping the main loop, C with MPI

```
MPI_Allreduce(&stop,&stopGlobal,1,MPI_INT,MPI_MAX,MPI_COMM_WORLD);
if(stopGlobal>=0)
    break;
```

At the top of the main loop, we place an `MPI_Allreduce` call, for finding the maximal value for `stop`. All processes will be aware of this maximal value `stopGlobal`. If that value is greater than 0, we should stop (see Listing 2.9). As the variable `stop` is initialized to -1 on all processes, it can be easily seen that any value greater than -1 shows that we have a stopping signal. In fact, the maximal value of the `stop` variable will be the highest rank value in the communicator. In that case, each process breaks the loop. This way, we ensure that all the processes leave the loop, and later finalize in a proper manner. While not being a realistic stopping criterion in a fully distributed setting, this approach allows us to adequately compare different algorithmic strategies. When the algorithm finishes, a log file is being created. This file contains the important data about the algorithm execution, including the total execution time, among others. The total execution time is the execution time of the slowest process. These log files are very important from the

aspect of testing the algorithm, that will be described later.

Introducing the algorithm variant without Hessian inverse approximation

During the test phase, we will investigate the performance of the implementation. One of the aspects to consider will also be the time consumption for different parts of the algorithm. For that reason, as we are aware that the Hessian inverse approximation can take a significant amount of time, we are interested in the results when we omit the Hessian inverse approximation calculation, i.e. when we work with first order methods. We can possibly expect faster iterations, as we do not spend the execution time on second-order update determination. But, we will possibly need a larger number of iterations in order to converge towards the solution. These aspects will be evaluated in detail in Section 2.3.3, that is dedicated to the tests. Here, we want to explain what happens in the code, when we want to omit the calculation of the variable `AWeightInv`.

Listing 2.10: The solution update, without Hessian inverse calculation, C with MPI

```
double *AWeightInv=calloc((s+1)*(s+1),sizeof(double));
for(int i=0;i<s+1;i++)
        AWeightInv[i*(s+1)+i] = 1.0;
cblas_daxpy(s+1,alpha,Grad,1,NablaPsi,1);
cblas_dgemv(CblasRowMajor,CblasNoTrans,s+1,s+1,1.0,AWeightInv,
            s+1,NablaPsi,1,1.0,sDirection,1);
cblas_daxpy(s+1,-1,sDirection,1,X,1);
```

Listing 2.10 shows the code snippet for the solution update, when we do not want to calculate the Hessian inverse approximation. In that case, we replace the matrix representing the inverse of the Hessian, with an identity matrix and use that identity matrix to calculate the solution update. This means that we do not need the computation of the variable `GMatrix` here, that was shown on Listing 2.6. It should be noted here, that we do not want to omit the calculation of the Hessian inverse from the implementation permanently. We just want to investigate both possibilities, i.e. the algorithm with and without this computation (first and second order methods) and compare the performance.

Introducing communicators

The tests performed on the developed algorithm as it was described for now, showed that it had serious performance issues. As we measured the overall execution time, and also the time needed for different parts of code, we concluded that, not surprisingly, the all-to-all communication protocol represents the bottleneck of the implementation. When working with larger data sets, the amount of data that goes to each process is naturally larger.

With the all-to-all communication protocol, the amount of data for exchange in every iteration represents a significant problem. By the nature of the algorithm, the nodes do not need the data from all the other nodes, only from their neighbours. The problem of significant communication time can be possibly solved by reducing the communication channels, so that each node communicates only with its neighbours. This can be achieved by using an array of communicators. We eliminated the all-to-all data exchange, by introducing an array of communicators. This means that n communicators need to be created, one for each node. The i-th communicator contains the node i as the master node. It also contains all the nodes that are connected to the node i. The creation of these communicators should be done once, before the iterative part of the algorithm starts. This way, during the iterations, the value exchange on a node should happen for all the communicators where the node is a master (there is exactly one such case) or a neighbour to the other node. The global communicator is also preserved, as it is needed for synchronization. The part of the implementation regarding the solution update exchange should be also updated, in order to eliminate the call to `MPI_Allgather`, and replace it with a code snippet that enables communicator based connections among the nodes. But first, let us explain the process of creating the communicators, shown on Listing 2.11.

Listing 2.11: The creation of communicators, C with MPI

```
MPI_Group worldGroup, myGroup; MPI_Comm myComm, tmpComm;
MPI_Comm *allComms=calloc(n, sizeof(MPI_Comm));
for(i=0;i<n;i++){
        if(my_rank==i)
                numOfNeighbours=my_neighbours_count;
        MPI_Bcast(&numOfNeighbours,1,MPI_INT,i,MPI_COMM_WORLD);
        int *arrayOfNeighbours=calloc(numOfNeighbours+1,sizeof(int));
        if(my_rank==i){
                arrayOfNeighbours[0]=my_rank;
                for(int k=0;k<my_neighbours_count;k++)
                        arrayOfNeighbours[k+1]=my_neighbours[k];
        }
        MPI_Bcast(arrayOfNeighbours,numOfNeighbours+1,MPI_INT,
                  i,MPI_COMM_WORLD);
        MPI_Comm_group(MPI_COMM_WORLD,&worldGroup);
        MPI_Group_incl(worldGroup,numOfNeighbours+1,
                       arrayOfNeighbours,&myGroup);
        MPI_Comm_create(MPI_COMM_WORLD,myGroup,&tmpComm);
        allComms[i]=tmpComm;
        free(arrayOfNeighbours);
        if(my_rank==i)
                myComm=tmpComm;
}
```

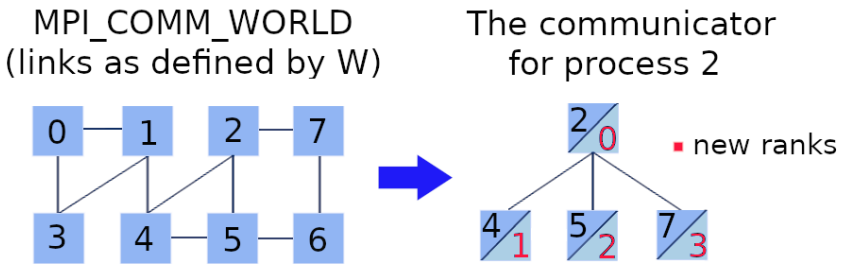


Figure 2.4: Creating a new communicator

As Listing 2.11 shows, we first allocate the variable for the array of communicators, `allComms`. Then, we should iterate a loop `n` times, as we need to create `n` communicators. The size of the `i`-th communicator is equals to the number of neighbours of node with rank `i`, plus one (the node `i` itself). We already have a variable `my_neighbours_count` that is used for defining the communicator size. We need to broadcast this value, so that all the processes are then aware of the size of the communicator that is being created currently. Then, we need to create the array of nodes that will participate in the communicator. A process whose rank is equals to the iteration counter, becomes the master process in the newly created communicator. Therefore, we put the rank of that process to the position 0 in the variable `arrayOfNeighbours`. Then, we copy the ranks of neighbours of that node, that we already had before, to the rest of the positions of `arrayOfNeighbours`.

The order of ranks during this procedure is being maintained. We create an array of ranks that go to the new communicator, ensuring that the first rank corresponds to the process for which we are creating the communicator. The other ranks will actually go in ascending order into the communicator, following the same ordering as in the array of neighbours. This is important, as inside the new communicators, the processes will have new ranks, and the only way for the new master to identify their original ranks is to maintain the ordering. The rank of the first process will be 0 in its own communicator, as it represents the master node in that subset of nodes. The ranks of other processes in the new communicator will be assigned one after the other. Let us illustrate this on an example, shown on Fig. 2.4. Suppose that we are creating a communicator for process 2, and that the neighbours are the processes with ranks 4, 5, 7 in the global communicator, as shown on left on Fig. 2.4. We create an array, that contains the ranks of the nodes, that need to be included as $[2, 4, 5, 7]$. This results with new ranks for the processes in the newly created communicator, i.e. we get $[0, 1, 2, 3]$ (as shown on right, on Fig 2.4).

When the array of the processes for the communicator is being created by the `i`-th process (that will be the master in the new communicator), the `i`-th process broadcasts that array on the level of the global communicator. Then, by calling `MPI_Comm_group`, we

obtain the global group `worldGroup`, that is by default assigned to the `MPI_COMM_WORLD` communicator. The call to `MPI_Group_incl` creates a new group `myGroup`, based on the `worldGroup`, and the new group will contain the defined array of process ranks. Finally, we create the communicator, by calling `MPI_Comm_create`. The new communicator is called `tmpComm`. We put it in the appropriate position in the array of communicators and set it as the own communicator of the process `i`, by assigning the communicator to `myComm`. This way, the communicators are organized in an array, also keeping the ordering. The `i`-th position is reserved for the communicator of the process than has rank `i` in the global communicator.

The next task is to adapt the data exchange so that it utilizes the created communicators. This is shown on Listing 2.12. We replace the call to `MPI_Allgather`, with a loop having `n` iterations. Inside that loop, a process calls `MPI_Gather`, if its rank corresponds to the iteration counter or if it is a neighbour of the node with rank equals to the iteration counter. The `MPI_Gather` is called for the `i`-th communicator, where `i` is the iteration counter. Here, the dimension of the vector `Xremote` corresponds to the size of the communicator. Its size is $(s+1) \times (my\_neighbours\_count + 1)$.

Listing 2.12: The data exchange using communicators, C with MPI

```
for(int c=0;c<n;c++)
    if(my_rank==c || is_my_neighbour(my_rank,c,my_neighbours,
                                     my_neighbours_count))
        MPI_Gather(X,s+1,MPI_DOUBLE,Xremote,s+1,MPI_DOUBLE,0,
                                                  allComms[c]);
for(i=0;i<my_neighbours_count;++i){
        double *zero=calloc(s+1, sizeof(double));
        int my_neighbours_rank=my_neighbours[i];
        LAPACKE_dlacpy(LAPACK_ROW_MAJOR,'A',1,s+1,X,s+1,Xdiff,s+1);
        cblas_daxpy(s+1,-1,Xremote+(i+1)*(s+1),1,Xdiff,1);
        cblas_daxpy(s+1,myWMatr[my_neighbours_rank],Xdiff,1,zero,1);
        cblas_daxpy(s+1,1,zero,1,NablaPsi,1);
        free(zero);
}
```

The source process, gathering the result is always 0 in the current communicator (that corresponds to `i` in the global communicator). This way, we ensure that each process gathers its data from the neighbours, as it has rank 0 in its own communicator. On the other hand, a process sends its data to be gathered on a master process of other communicator, if the nodes are neighbours. In order to check whether a node is a neighbour to the current node, we use an auxiliary function `is_my_neighbour`. This function simply iterates over the array of neighbours searching for the particular rank (see Listing 2.13).

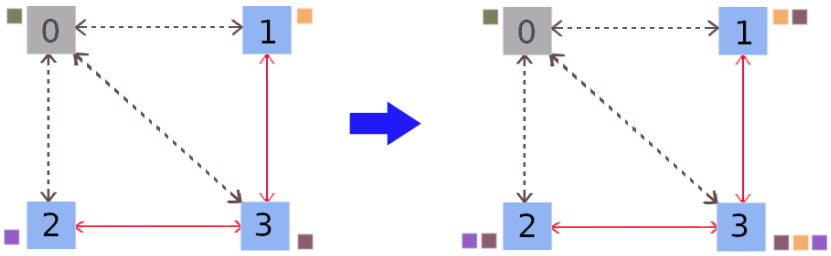


Figure 2.5: An example of sparsified communication

For checking the stopping criterion, we still use the global communicator, as we need the data from all the processes. This enhancement makes the code execute more efficiently, as will be shown in the section, dedicated to the experimental results.

Listing 2.13: The function *is_my_neighbour*, C with MPI

```
int is_my_neighbour(int my_rank,int i,int *my_neighbours,
                                                int my_neighbours_count){
        for(int j=0;j<my_neighbours_count;++j)
                if(my_neighbours[j]==i)
                        return 1;
        return 0;
}
```

Introducing sparsification

In order to additionally enhance the algorithm performance, a communication sparsification can be applied. This means that some nodes become idle during the iterations. This principle reduces the amount of communication, and possibly leads to lower execution time. Fig. 2.5 represents an example for sparsified communication. We consider 4 nodes, where node 0 is meant to be idle in the current iteration. The dashed lines show the communication links for node 0, defined by `W`. However, as the node is idle, its communication links are not being utilized in the current iteration. On the right part of Fig. 2.5, it can be seen that node 0 only has its original data (it did not receive anything) and that the other nodes, that are neighbours of 0 do not have the data from 0. Only active nodes exchanged their data with their active neighbours.

There are different approaches for sparsifying the communication. We investigated different possibilities and made some important conclusions. In order to sparsify communications, we introduce communication probability p_k, as it was already described. The probability can be a fixed value. We first investigated the cases for values $p_k = 0.3$, $p_k = 0.5$, $p_k = 0.8$. Next, the probability value can alternatively change during the iterations,

it can increase as $p_k = 1 - 0.5^k$, or decrease as $p_k = \frac{1}{k+1}$, where k is the iteration counter. We investigate and compare different probability values, combined with other features of the algorithm, in order to make some valuable conclusions. This will be explained in more details in the section dedicated to the experiments.

The solution update formula should now be changed, in order to encode the sparsification:

$$x_i^{k+1} = x_i^k - \Big[(M_i^k)^{-1}[\alpha \nabla f_i(x_i^k) + \sum_{j \epsilon \Omega_i} W_{ij}(x_i^k - x_j^k)\xi_{i,j}^k]\Big], \tag{2.57}$$

If we associate a Bernoulli random variable z_i^k to each node i, that denotes its communication activity at iteration k, then $z_i^k = 1$ means that node i communicates in the k-th iteration, where if $z_i^k = 0$, it means that node i does not exchange messages with neighbors in the k-th iteration. Then, the probability is $p_k = Prob(z_i^k = 1)$. $\xi_{i,j}^k$ is in general a function of z_i^k and z_j^k that encodes the communication sparsification. The value $\xi_{i,j}^k = 1$ means that there is no communication sparsification. This corresponds to the previous implementations. $\xi_{i,j}^k = z_i^k \cdot z_j^k$ corresponds to bidirectional communication sparsification, i.e. the implementation that we are currently describing. $\xi_{i,j}^k = z_j^k$ is for unidirectional communication, that will be described in the next topic.

From the aspect of the implementation, introducing the idling mechanism means that we need to invest some additional effort to the code development. The code snippet, that represents the process of recreating the communicators at the beginning of the main loop is shown on Listing 2.14. First, we introduce a probability bound value, as `probab_bound=pk*10`. This value can be constant all the time, or it can change, according to the described principle of increasing or decreasing communication probability. Consider the case when the probability is not constant. Then, at the beginning of an iteration, we calculate the current `probab_bound` value. We set some limits, so that this value cannot be smaller than 1, when the probability decreases, and it cannot be smaller than 5, when the probability increases, as we want to start from $p_k = 0.5$. Listing 2.14 shows the case when the probability decreases. For constant probability values, we will not have the calculation of the `probab_bound` at the beginning of the loop, but instead, it will be defined as a constant before the main loop.

Next, each process generates a random number `comm_probab` from the range between 1 and 10. It is important to initialize the seed for random number generator on each process for every iteration separately, depending on the iteration counter and node's rank, in order to prevent it from returning always the same values. We need to maintain a new array of communicators `allCurrComms`, that contains only the communicators of the active nodes in the current iteration `k`. However, those communicators will not be the

same as we created them in advance, as we need to remove the inactive nodes from the communicators. The easiest way to achieve this is the use of function `MPI_Comm_split`. This function enables to split a communicator into a set of disjoint communicators. The function is declared as:

`int MPI_Comm_split(MPI_Comm comm,int color,int key,MPI_Comm * newcomm).`

The function splits a communicator into an array of communicators, regarding the flag `color`. The first argument of the function is the current communicator that needs to be split. The last argument is the new communicator where the process belongs, created by splitting the original communicator. The argument `color` determines the split, as all the processes with the same color value will be assigned to the same communicator. The `key` value helps to assign the ranks to the processes in the newly created communicators. In our case, we just want to remove some processes from the communicator, so we will have just one resulting communicator after split. This function enables for processes to set the `color` value to `MPI_UNDEFINED`. This way, the process will be omitted from the resulting communicator (those processes get `MPI_COMM_NULL` for the resulting communicator after split). Following this scheme, the process of splitting the communicators is straightforward.

Listing 2.14: Recreating the communicators at the beginning of an iteration, according to the idling mechanism, C with MPI

```
color=1; activeLoc=0; active=calloc(n, sizeof(int));
allCurrComms=calloc(n, sizeof(MPI_Comm));
probab_bound=(1/(k+1))*10;
if(probab_bound<1)  probab_bound=1;
srand(time(NULL)+my_rank+k);
comm_probab=rand() % 10 +1;
if(comm_probab > probab_bound) color=MPI_UNDEFINED;
else activeLoc=1;
MPI_Allgather(&activeLoc,1,MPI_INT,active,1,MPI_INT,MPI_COMM_WORLD);
MPI_Comm_split(MPI_COMM_WORLD,color,my_rank,&global_active_comm);
for(int c=0;c<n;c++){
        if((my_rank==c && activeLoc) || (is_my_neighbour(my_rank,
                c,my_neighbours,my_neighbours_count) && active[c])){
                int my_local_rank;
                MPI_Comm_rank(allComms[c], &my_local_rank);
                MPI_Comm_split(allComms[c], color,
                                my_local_rank, &allCurrComms[c]);
        }
}
```

If the generated communication probability on a node represents a larger value than the defined probability bound, then the node will be idle in the current iteration. As we decrease the probability bound across the iterations, a smaller number of processes will be included in the communication. With increasing communication probability, it will be the opposite. When a node is inactive, we set its color to `MPI_UNDEFINED`, otherwise it will be 1. We also create an array that indicates the activeness of nodes. The variable `activeLoc` indicates whether a node is active or not. We call `MPI_Allgather` to crate an array, based on all nodes value for `activeLoc`. The result is stored in a vector `active`, that will be useful later. We first split the global communicator, based on the color (only active nodes has color value 1). This way, we can have a communicator that contains all the nodes that are active. Then, we iterate in a loop `n` times (once for each communicator), in order to split the communicators. If the node's rank equals to the iteration counter and the node should be active, then we need to redefine its communicator. The second condition is if the iteration counter is a rank of a node that is a neighbour of the current node and that node is active. In that case, we also redefine its communicator. Basically, whenever a master node of a communicator should be active, we need that communicator. When a master node of a communicator is inactive, then we do not need that communicator. When splitting the current communicator, each process provides its local rank from that communicator, in order to preserve the ordering (even if some processes are omitted now). The inactive nodes will also call the split function, but they pass `MPI_UNDEFINED` for color, so they will be omitted from the new communicator. To summarize, an idle node will not participate in the communicators of its neighbours, and it will not use its own communicator at all.

The data exchange part follows, after the nodes finished their local calculations. It needs to be changed significantly now. This part of the code is shown on Listing 2.15. The data exchange on a node happens only if it is active. If a node is active, it needs to allocate a vector, whose size corresponds to the current size of the communicator for that node. Next, the node runs a loop `n` times. If the iteration counter `c` is equal to the rank of the process, then it should call `MPI_Gather` in order to gather the data from active neighbours. The other case we are interested in is when the iteration counter value `c` is equals to some of the node's active neighbour's rank. In that case, the node also calls `MPI_Gather`. The gathering is always called on `allCurrentCommunicators[c]`. This ensures that, when a node calls gathering for its own communicators, than its rank is 0 in that communicator, so it gets the data. When a node calls gathering on some other node's communicator, than its rank is greater than zero, so it just sends data.

Listing 2.15: The data exchange with sparsified communications, C with MPI

```
if(active[my_rank]){
    my_communcator_size=0;
    MPI_Comm_size(allCurrComms[my_rank], &my_communcator_size);
    Xremote=calloc((my_communcator_size)*(s+1),sizeof(double));
    for(int c=0;c<n;c++)
        if(my_rank==c || (is_my_neighbour(my_rank,c,
                my_neighbours,my_neighbours_count) && active[c]))
            MPI_Gather(X,s+1,MPI_DOUBLE,Xremote,s+1,
                    MPI_DOUBLE,0,allCurrComms[c]);
}
```

There is one more important aspect, after the data exchange finishes. We have to maintain the rule, that the sum of the weights for the nodes should be 1, i.e. the values on the diagonal of the matrix `W` should change now. While computing the sum in equation (2.57), we can determine the sum of weights of active neighbours. Later, we will need this value to get $W_{ii} = 1 - active\_neighbours\_weight$. We sum the weights of the active neighbours, while computing the value of the variable `NablaPsi`, i.e. consuming the solutions gathered from the active neighbours. This is shown on Listing 2.16.

Listing 2.16: Processing the solutions from active neighbours, C with MPI

```
if(active[my_rank]){
    active_neighbours_weight=0.0;
    for(i=0;i<my_communcator_size-1;++i){
        double *zero=calloc(s+1,sizeof(double));
        int my_neighbours_rank=get_my_active_neighbour(i,
                my_rank,my_neighbours,my_neighbours_count,active);
        LAPACKE_dlacpy(LAPACK_ROW_MAJOR,'A',1,s+1,X,s+1,Xdiff,s+1);
        cblas_daxpy(s+1,-1 Xremote+(i+1)*(s+1),1,Xdiff,1);
        cblas_daxpy(s+1,myWMatr[my_neighbours_rank],Xdiff,1,zero,1);
        active_neighbours_weight+=myWMatr[my_neighbours_rank];
        cblas_daxpy(s+1,1,zero,1,NablaPsi,1);
        free(zero);
    }
}
```

An active node iterates in a loop, where the number of iterations is determined by the size of its communicator. Here, we need a special function, that will return the rank of the `i`-th active neighbour. It will return the rank of a neighbour, regarding the global communicator `MPI_COMM_WORLD`. Then, we can subtract the solution of that node from the local solution, and multiply the result by the weight for that neighbour. Note that we pick the `i+1`-th chunk from `Xremote`. This is because, in the current local communicator, the first

neighbours solution comes to position 1, as the own solution comes to position 0, and so on for the other neighbours. Using the global rank of the neighbour, we get the appropriate value from the weight matrix and add it to the sum `active_neighbours_weight`.

Listing 2.17 shows the implementation of the function `get_my_active_neighbour`. It is very straightforward. It iterates through the neighbours array and counts the active ones. When the counter reaches the desired position, it returns the rank of the process that corresponds to that position. This function returns a process rank regarding the global communicator `MPI_COMM_WORLD`.

When checking whether the stopping criterion is met, all the processes are obligated to participate. That means that even if a process is idle at the current iteration, it should send its current result, that of course, does not include any result from the neighbours, just a locally obtained value. On the other hand, active processes should calculate their current result values, but with the modified weight value W_{ii}. We can call this value the node's self confidence. This value has changed now, as the sum of weights for the neighbours changed, because the inactive nodes weight values are not relevant now. Basically, it means that the more inactive neighbours a process has, its self confidence should grow accordingly.

Listing 2.17: The implementation of the function *get_my_active_neighbour*, C with MPI

```
int get_my_active_neighbour(int k,int my_rank,
            int *my_neighbours,int my_neighbours_count,int *active){
        int cnt_active=-1; int neighbour=-1;
        for(int i=0;i<my_neighbours_count;i++){
                if(active[my_neighbours[i]])
                        ++cnt_active; neighbour=my_neighbours[i];
                if(cnt_active==k) return neighbour;
        }
}
```

Listing 2.18 shows the process of gathering the data for checking the stopping criterion. An active node calculates its self confidence, using the sum of weights of active neighbours, that was computed before.

The self confidence value is then `1-active_neighbours_weight`. After this point, it calculates the value of variable `curRes` as before. It iterates through the list of its neighbours and adds the neighbours solutions multiplied with weights to the result. Finally, the gradient value from the previous iteration is also being added, multiplied by the step

size alpha. In the case of an inactive node, the variable curRes only contains the gradient from the previous step, multiplied by the step size. Here, the self confidence is 1.

Listing 2.18: The data preparation and gathering for stopping criterion check, MPI with C

```
double *curRes=calloc(s+1,sizeof(double));
GradijentGlob=calloc((s+1)*n, sizeof(double));
if(active[my_rank]){
    my_self_confidence = 1-active_neighbours_weight;
    for(i=0;i<s+1;i++){
        curRes[i]=(1-my_self_confidence)*X[i];
        for(j=1;j<=my_communcator_size-1;j++){
            int my_neighbours_rank=get_my_active_neighbour(j-1,
               my_rank, my_neighbours, my_neighbours_count, active);
            curRes[i]+=-myWMatrix[my_neighbours_rank]
               *Xremote[j*(s+1)+i];
        }
        curRes[i]+=alpha*GradOld[i];
    }
}else{
    my_self_confidence=1;
    cblas_daxpy(s+1,alpha,GradOld,1,curRes,1);
}
MPI_Gather(curRes,s+1,MPI_DOUBLE,GradijentGlob,s+1,MPI_DOUBLE,0,
           MPI_COMM_WORLD);
```

After the code snippet from Listing 2.18 is being executed, the master node gathers all the curRes vectors and computes the euclidean norm, as already explained before. The rest of the algorithm (gradient, Hessian and solution update computation) is also the same as described before. This mechanism of communication sparsification will be used in different variants during testing, and it will be combined with other aspects of different variants of the algorithm (see more details in Section 2.3.3).

Introducing unidirectional communication

Introducing sparsification as explained means using bidirectional communication protocol, by default. This means that an inactive node does not communicate with its neighbours, while an active node sends and also receives data. However, an interesting approach to reduce the communication amount, and hence the time needed for it, is to implement the idea of unidirectional communication. As already described, this approach ensures that an active node receives data only from active neighbours, but sends data to all of them, regardless of the fact whether they are active or not. An example for unidirectional

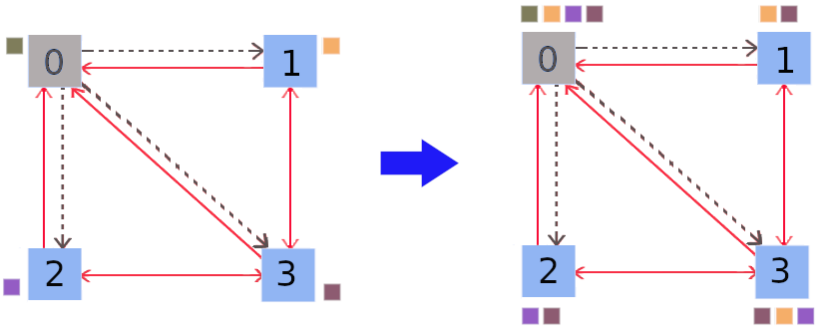


Figure 2.6: An example for unidirectional communication

communication is displayed on Fig. 2.6. Here, we consider 4 nodes, and node 0 is idle in the current iteration, meaning that it does not send its data to neighbours (the dashed lines). However, it still receives data from its active neighbours (the red lines). The right side of Fig 2.6 shows the state of nodes and data, when the communication is finished. This means, that an idle, i.e. inactive node does not sends its solution update to the neighbours, but still receives data from the active ones. In equation (2.57), the value of ξ is now $\xi_{i,j}^k = z_j^k$. This could be a possible point for execution time reduction, but we need to keep in mind the existence of other additional costs during executing the code with this kind of sparsification. Possible bottlenecks could appear due to an additional effort needed to set up the described scenario, and also regarding the required number of iterations to converge. These aspects will be evaluated in the section, that is dedicated to testing the algorithms (particularly Section 2.3.3).

When considering implementing the unidirectional communication with idling, there are a few things that need to be changed, compared to the previously described bidirectional approach. The main difference is that now, for inactive nodes, we still need to keep their communicators. But, those communicators include the node itself, and only the active neighbours. This can be achieved, as shown in Listing 2.19.

We cannot set a flag for activeness of a node in advance, as it will not be the same in different communicators. Actually, an idle node is inactive for the neighbours, but in a sense ‘active‘ for receiving data. We also need a loop, with `n` iterations. Inside it, if a node's rank is equals to the iteration counter `c`, then it should be included in its own communicator (regardless of its activity). Similarly, if a node is a neighbour to the node with rank equals to `c` and it is also active, it should be included into that communicator. These two cases are displayed by the first condition of `if` statement in the code. A special case is when the node is a neighbour to the node with rank `c`, but is not active. In that case, we omit that node from the communicator and set the color flag to `MPI_UNDEFINED`. This is the condition of the `else if` statement. Otherwise, when a node is not a neighbour

to node c, then it was neither included in that communicator initially.

Listing 2.19: The process of recreating the communicators for unidirectional sparsified communication, C with MPI

```
if(comm_probab <= probab_bound)  activeLoc=1;
MPI_Allgather(&activeLoc,1,MPI_INT,active,1,MPI_INT,MPI_COMM_WORLD);
for(int c=0;c<n;c++)
    if(my_rank==c || (is_my_neighbour(my_rank,c,
      my_neighbours, my_neighbours_count) && active[my_rank])){
        int my_local_rank;
        MPI_Comm_rank(allComms[c],&my_local_rank);
        MPI_Comm_split(allComms[c],1,my_local_rank,&allCurrComms[c]);
    }
    else if(is_my_neighbour(my_rank, c,
           my_neighbours, my_neighbours_count) && !active[my_rank]){
        int my_local_rank;
        MPI_Comm_rank(allComms[c],&my_local_rank);
        MPI_Comm_split(allComms[c],MPI_UNDEFINED,my_local_rank,
                       &allCurrComms[c]);
        }
```

The data exchange process is similar as for bidirectional communication. It is shown in Listing 2.20. The main difference is that we do not need to check here whether a node is active, in order to perform communication, as the idle nodes should also receive data. So, each node communicates in a sense here. We again need a loop with n iterations. If a node's rank is equals to the iteration counter, then it should certainly communicate, i.e. receive data from its neighbours, regardless of its activeness. This part is related to receiving data. When thinking about sending the solution update to neighbours, it is a task that only active processes should perform. In other words, if the rank that corresponds to the iteration counter is a neighbour of the current process and the current process is active, then it should communicate, i.e. send data to the neighbour.

Listing 2.20: The data exchange with unidirectional sparsified communication, C with MPI

```
my_communicator_size=0;
MPI_Comm_size(allCurrComms[my_rank],&my_communicator_size);
double *Xremote=calloc((my_communicator_size)*(s+1),sizeof(double));
for(int c=0;c<n;c++)
        if(my_rank==c || (is_my_neighbour(my_rank,
             c, my_neighbours,my_neighbours_count) && active[my_rank]))
                MPI_Gather(X,s+1,MPI_DOUBLE,Xremote,s+1,
                           MPI_DOUBLE,0,allCurrComms[c]);
```

Regarding the processing of the neighbours solutions, this part of the code is the same as for bidirectional communication (see Listing 2.16). The only difference is that we do not need the `if` statement any more, as each node has some solution update from the neighbours. It can happen at some point that a node does not have any active neighbours. In that case, the communicator size is 1, so the loop will not be executed and the sum of weights for the neighbours will remain zero. The data preparation and gathering for determining the stopping criterion occurrence is also almost the same as for the bidirectional communication approach (see Listing 2.18). The difference is that we do not need the check if the node is active. Hence, we do not need the line `if(active[my_rank])`, and we can completely remove the else branch and its content as well.

Conclusion on parallel implementation for logistic loss functions

The described incremental development of parallel implementation of the algorithm for logistic loss functions can be observed in two dimensions. The first one is related to enhancements. This subsumes the possibility for reading input data of arbitrary size, and introducing the stopping criterion, instead of fixed number of iterations. All the tests performed on logistic loss functions will rely on this enhancements. The introduction of communicators can be also put here, because the tests are mainly directed to the version of the algorithm that uses communicators. However, we will briefly mention the tests that did not use communicators, in order to encourage introducing them. The second dimension represent the possibilities to: introduce communication sparsification, to exclude the second order updates and to work with bidirectional or unidirectional communication sparsification. We will create different methods, by combining these properties and base the tests on them. This way, we can explore how these different possibilities influence the performance.

2.2.3 A comparison with an ADMM implementation

In this section, the focus is on a class of primal convex optimization methods and the described implementation details correspond to these methods. However, we are interested in how the performance of these methods relate to the performance of a dual method, applied for the same algorithm, on same data. A fair comparison requires an MPI implementation for logistic regression in C language. Problem (2.1) can be solved using the Alternating Direction Method of Multipliers (ADMM) [3], so we decided to compare our Algorithm 1 to an ADMM implementation for logistic regression [122] (the authors provide the source code on GitHub), also implemented in C, and using MPI for parallelization. The reason for choosing ADMM is twofold. First, ADMM represents a widely

applied, efficient approach. Second, the next chapter of the thesis focuses on ADMM-based algorithms as representatives of dual methods.

A few adaptations need to be applied, in order to make a meaningful comparison. More precisely, the method in [3] solves problem (2.1) assuming the presence of a central node that communicates to all other nodes in the network. Hence, we need to adapt our algorithmic framework to the latter setting by letting the underlying network G to be fully connected and by setting the matrix W to have all its entries equal $\frac{1}{n}$.

We calculate the value of $\Phi^k = \frac{1}{n}\sum_{i=1}^{n} f(x_i^k)$, i.e., the average global cost in (2.1) averaged across all nodes' estimates, at the end of each iteration and we also measure the execution time; where $f(x) = \sum_{i=1}^{n} f_i(x)$. We are interested in the time required to satisfy the condition $\frac{\Phi^k - f^*}{f^*} < 0.1$. Here, $f^\star$ is numerically evaluated by ADMM. The rationale for this comparison is the following. Our algorithm (Algorithm 1) converge to a neighbourhood of the solution to (2.1), while ADMM converges to the exact solution of (2.1). Therefore, it is meaningful to compare the times that each method needs to reach a certain accuracy level, measured with respect to the cost function in (2.1).

In order to be able to measure the time, required to satisfy $\frac{\Phi^k - f^*}{f^*} < 0.1$, we need to add a computation for the following:

$$\Phi^k = \frac{1}{n}\sum_{i=1}^{n}\sum_{j=1}^{J} log(1 + \frac{1}{e^{b_{ij}(x_1^\top a_{ij} + x_0)}}) + \frac{\tau}{n}||x||^2 \tag{2.58}$$

at the end of each iteration, for both implementations (our algorithm and ADMM). Also, we measure the time for each iteration, but that time naturally does not include the calculation of (2.58). This metric requires that we calculate $f(x) = \sum_{i=1}^{n} f_i(x)$, meaning that for each node's solution, we should consume the whole input data, i.e. the data distributed between the nodes at the beginning of the algorithm. This can be done in two ways. We can gather the input data parts on each node once, at the beginning, so that the node can perform the calculation for its solution at the end of each iteration. The second approach is to gather the whole data set only on the master node once, before the main loop, and then send the solution approximations from the nodes to the master, meaning that the master calculates the whole expression. We chose the second option. The reason is that this way, we need the whole data set only on the master node. Also, we chose a smaller data set for these tests, that is suitable to make the comparison, but still does not overload the master node. The dimension of the solution update is also reasonably small, so the computation of the described metric can be done easily. The setup and results of tests will be described in Section 2.3.3.

Listing 2.21: Implementation of the process of obtaining the value of Φ^k, C with MPI

```
MPI_Reduce(&comm_time,&max_comm_time,1,MPI_DOUBLE,MPI_MAX,
           0,MPI_COMM_WORLD);
MPI_Reduce(&iter_elapsed,&max_iter_time,1,MPI_DOUBLE,MPI_MAX,
           0,MPI_COMM_WORLD);
MPI_Gather(X,s+1,MPI_DOUBLE,Xall_k,s+1,MPI_DOUBLE,0,MPI_COMM_WORLD);
if(my_rank==0){
    double sum=0.0;
    for(int i=0; i<n; i++){
        double Fx=0.0; double *Xk=Xall_k+i*(s+1);
        double x0=Xk[s]; double *x1=calloc(s,sizeof(double));
        LAPACKE_dlacpy(LAPACK_ROW_MAJOR,'A',1,s, Xk,s,x1,s);
        for(int j=0; j<n; j++){
            double *Ai=A+(j*r*s)+(j!=0)*rem*s;
            double *Bi=B+(j*r)+(j!=0)*rem;
            double sumForSample = 0.0;
            for(int t=0; t<r+(i==0)*rem; t++){
                double dotProd=cblas_ddot(s,Ai+(t*s),1,x1,1);
                double coeff=(dotProd+x0)*(-Bi[t]);
                double exp_val=exp(coeff);
                double log_val=log(1+exp_val);sumForSample+=val;
            }
            double x_norm=cblas_dnrm2(s+1,Xk,1);
            Fx+=sumForSample+lambda_penal*x_norm;
        }
        sum+=Fx;
    }
    avg=sum/n; Fi[k]=avg;
    iter_timings[k]=max_iter_time; comm_times[k] = max_comm_time;
}
```

The code for the computation of Φ^k is shown in Listing 2.21. This computation is being inserted to the end of main loop of the algorithm. It is used only for evaluation purposes, and the time needed for this is not part of the running time of the algorithm (it is not taken into account when measuring the execution time). First, it is important to record the time required for that iteration as the required time of the slowest process. Similarly, we also compute the total communication time for an iteration. Also, the master node gathers the current solution estimates to `Xall` and initializes the overall sum. Then, the master node computes (2.58), so that the first loop iterates over the solution estimates. It picks `Xk` from `Xall` and initializes `Fx=0`, as it will be computed inside the inner loop. The second loop iterates over data chunks (with `j` as the iteration counter). It takes the `j`-th part of the input data matrix and vector. It also takes into account the possible overage

on the 0-th chunk. Finally, the third loop (where `t` is the iteration counter), computes the sum for one sample, iterating over the current matrix and vector chunk.

We add the norm of the current solution estimate to the value `Fx`, and then `Fx` is added to the final sum. We maintain three arrays globally. The `iter_timings` keeps the required timings for the iterations. Similarly, `communication_times` contains the time required for communications during the iterations. The array `Fi` contains the solution to (2.58) iteration-wise. These arrays are being written to separate files, when the algorithm finishes its execution.

The final step to obtain some meaningful results is to process these resulting files. We created a Python script that reads these files created by different variants of our algorithm, and also by the ADMM variant, and that produces some interesting insights. The script first reads the output files of the ADMM variant, and finds the minimal value of Φ^k, i.e. $f^\star$. It also determines the required number of iterations and the required time to reach that minimum. This process is being repeated for different variants of our algorithm (the exact list of them will be described in Section 2.3.3). Then, the absolute differences between different variants and ADMM, and the average time to reach own minimum are being computed. The script also truncates the vectors to contain only the values until the minimum, as a preparation step for possible plotting. Finally, the point where an algorithm satisfies $\frac{\Phi^k - f^\star}{f^\star} < 0.1$ is being found, and the sum of timings of the iterations to reach that point is being recorded.

2.2.4 Measuring the execution time in a parallel program

The way of measuring the execution time of a serial program is straightforward, as it only requires to check the points for measurement start and end and put the appropriate function calls recording the current time (this is dependent on the programming language) to that positions. However, when we are working with parallel programs, a set of processes is working simultaneously, and their execution times can be various. For that reason, we always should pick the execution time of the slowest process. After all, we need to wait for the slowest process to finish the execution, in order to terminate the application.

When measuring the execution time of an MPI application, we should measure the execution time for each process separately, and then find the largest value.

Listing 2.22 shows a snippet of code for measuring the time in an MPI application. The function `MPI_Wtime()` returns an elapsed time on the calling processor. We should call this routine immediately before and after the code block, that is of interest to measure its

execution time. By subtracting these values, i.e. the beginning from the end, we get the execution time for the particular process. Then, by calling `MPI_Reduce`, we can get the highest execution time among the processes and that is our final value. This approach can be applied to an arbitrary block of the code.

Listing 2.22: Measuring the execution time of an MPI program, C with MPI

```
double start = MPI_Wtime();
//here we put the code that we are interested in
//for measuring the execution time
double end = MPI_Wtime();
double elapsed=end-start;
double max_time;
MPI_Reduce(&elapsed, &max_time, 1, MPI_DOUBLE,MPI_MAX, 0,
           MPI_COMM_WORLD);
```

2.3 Experimentation

The testing of the algorithm includes a broad set of benchmarks. First, we illustrate some experimental results, that we obtained while developing the solution and comparing different aspects of the implementation of the algorithm. Secondly, for the final implementation, we make a thorough comparison of the properties of various methods, that utilize different communication sparsification strategies. In terms of the results of intermediate experiments regarding implementing the algorithm, a few different aspects emerge. First, the algorithm was compared to the serial implementation. These tests were conducted for the algorithm implementation for quadratic cost functions. We then also make a quick comparison between the all-to-all and communicator-based algorithm variants. Then, we test different sparsification strategies, in order to identify those we are mostly interested in. These tests were performed on the algorithm implementation regarding logistic loss functions. They enable the identification of the suitable implementation strategies and best performing sparsification approaches. The second part of the evaluations is oriented towards examining a class of implemented methods. When referring to "methods" through the thesis, we mean algorithm variations with different communication sparsifications, combined with first/second order approaches and with bidirectional/unidirectional communication lines between the nodes. We compare the performance of these methods in different setups, and also make a comparison between the appropriate methods with and without communication sparsification. These tests are performed for logistic loss functions. The final goal of the experimentation phase is to obtain some meaningful information about the nature of the algorithm. This is the empirical proof of the

performance properties of the described concepts.

2.3.1 The infrastructure

A few different infrastructures were used for the tests. First, the tests for serial approaches (MATLAB and plain C) were performed on a 64-bit Linux machine, with Core i5-4590 3.30GHz 4 core CPU and 16GB RAM. Then, for the parallel implementation, we used a real cluster environment for the tests. Particularly, we used two different clusters, at different moments of the evaluation. One of them is the AXIOM computing facility consisting of 16 compute nodes (8 x Intel i7 5820k 3.3GHz and 8 x Intel i7 8700 3.2GHzCPU - 192 cores and 16GB DDR4 RAM/node) interconnected by a 10 Gbps network. The tests were mainly performed on this cluster. However, at one stage (when choosing the appropriate sparsification strategies based on performance), we also used the Paradox cluster consisting of 106 compute nodes (2 x 8 core Sandy Bridge Xeon 2.6GHz processors with 32GB of RAM + NVIDIA Tesla M2090) interconnected by the QDR InfiniBand network.

2.3.2 Intermediate experimentation studies and results

In this subsection, we describe the conclusions that can be drawn, regarding the implementation. Here, we compare different approaches for implementing the algorithm, in order to identify the most suitable one, in terms of performance. We also make a comparison between different communication sparsification techniques, in order to choose the final class of methods, that we evaluate in details.

The tests regarding the algorithm implementation for quadratic cost functions

First, let us illustrate the results of evaluation on the parallel implementation of the algorithm for quadratic cost functions. We observe the scaling properties of the algorithm here. Also, we compare the parallel implementation to a serial MATLAB implementation, in order to assess the benefits of parallelization. All the serial tests regarding this phase were performed on a single machine (see the details in Section 2.3.1 regarding infrastructure). The performance of the parallel implementation was tested on the AXIOM cluster environment (see also Section 2.3.1 for details).

The simulation setup

As already described earlier, the tests for quadratic cost functions were run on synthetically generated data. The graph structure between the nodes is also randomly generated. For detailed explanation, see Section 2.1.1.

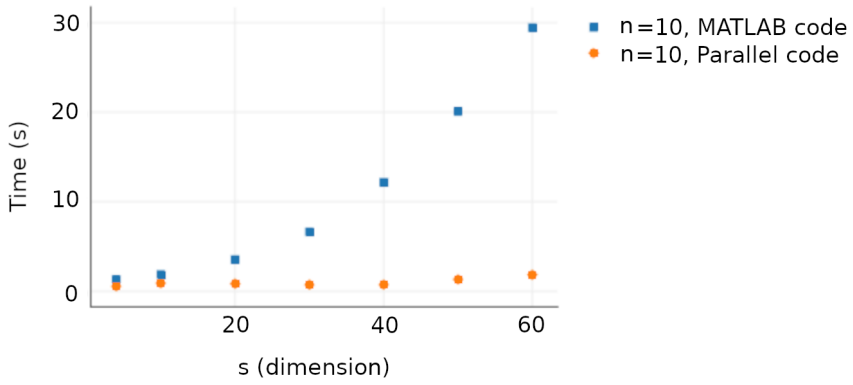


Figure 2.7: Comparing the execution time for MATLAB and parallel, MPI based C code

The experimental results

First, we want to investigate quickly how well an MPI based parallel algorithm, written in C performs, when compared to an implementation, written in MATLAB. As the nature of the algorithm is parallel, and the work is meant to be distributed among a set of nodes, a serial implementation of it means that the workload of each of the nodes is physically executed sequentially over a single machine. This can be done in a loop, that computes the solution update for node i, in the i-th iteration, where $i = 1, 2, ..., n$. We expect that the efforts invested in the parallel solution development pays off, and that this can serve as a good basis for further enhancements.

Fig 2.7 represents the comparison of the time needed for the serial version of the same algorithm written in MATLAB, where the updates of different nodes are carried out sequentially and for the parallel MPI based C version of the code. This comparison should assesses the gains of parallelization. The example illustrates relatively small input data sets. The reason for this is that serial algorithms limit the possibility for data enlargement. The time needed for the serial algorithm raises relatively quickly when increasing the data size. For only $s = 60$ as vector and matrix dimension and with $n = 10$ nodes in a graph, the execution time for MATLAB is 29.5 seconds. For the same input, the parallel program executes in 1.85 seconds. The MATLAB version is able to handle increased input data size to certain extent, but it is naturally very limited and unable to execute all the examples that we can run on a cluster. With larger data size, the serial program can deplete the resources and even block the machine during the execution. In contrast, the parallel implementation can handle these cases. We can provide much bigger amount of data to our algorithm, that leads to opening of possibilities for further work in this direction. In the observed examples on Fig. 2.7, it performs 16 times faster than the MATLAB version. When providing larger data sets, the difference in performance is growing even further. These tests were dedicated to prove the gains of the introduced parallelization. We can

conclude that they showed that introducing the parallel implementation is worthwhile.

The second part of tests on quadratic cost functions is meant to explore whether the implementation is scalable regarding performance. The described initial MPI implementation for quadratic cost functions was tested here for different data dimensions s and different graph sizes n. The execution times on a cluster environment are shown in Figs 2.8 and 2.9. As expected, increasing data size leads to execution time increase for the same number of nodes, as this way each node has a larger portion of work. If we consider different possibilities for number of nodes n, where the dimension is the same, it is noticeable that the time changes only slightly for increased number of nodes. Fig 2.8 shows that increasing the number of nodes n , for the same data dimension s, does not reduce the execution time. The small data dimension values s could influence this results. However, if we consider Fig 2.9, where we have larger data dimensions, we can derive very similar conclusions. We cannot identify the existence of scaling, as mostly the execution time is the same for different numbers of nodes n, or is even higher for higher values of n. This deterioration is most likely due to the costs of communications between the nodes, that can actually influence even execution time increase, while increasing n. The reason is that the gain of parallelzitaion on larger number of nodes is smaller than the losses caused by the communication. This means that our implementation is not scalable and the most possible reason is the all-to-all communication protocol. This approach leads to increased amount of communication lines, while increasing the number of nodes n. Of course, the number of links in different graph instantiations also varies, and this is also reflected in a varying time spent on communications across different graph instances.

We should note that, for the quadratic cost considered here, the problem admits a closed-form solution, only in the centralized setting where a single node has access to all functions f_i 's. However, this is not the case for generic f_i 's. A study of this kind will be also described later.

To summarize, we carried out a performance evaluation of the proposed method for quadratic cost functions when implemented in a distributed MPI cluster environment. We concluded that the advantage of introduced parallel algorithm are evident. The first gaining is the performance improvement, where our program performs significantly faster than the MATLAB version. Also, the MPI implementation can clearly accommodate larger data sizes s. In our setup, the MATLAB implementation breaks down already at $s = 150$, depleting the resources or blocking the machine.

The communication overhead was expected to play a significant role regarding the execution time. From the aspect of the parallelization, the nature of the graph is completely

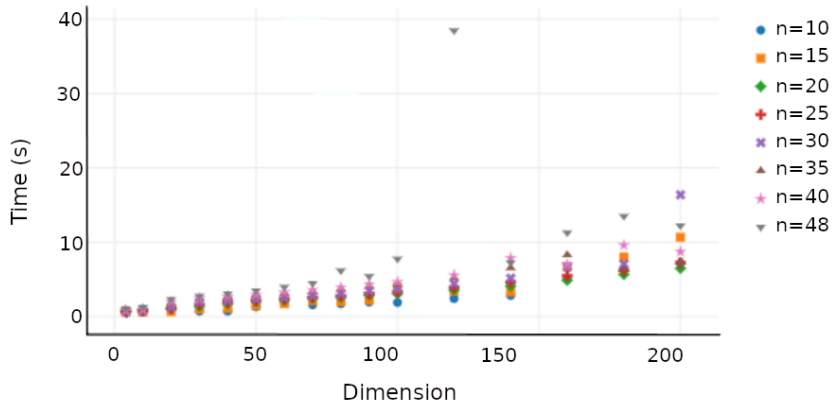


Figure 2.8: Execution time of the distributed MPI implementation for quadratic cost functions for different data sizes s and different number of nodes n, for data dimension 200 and smaller

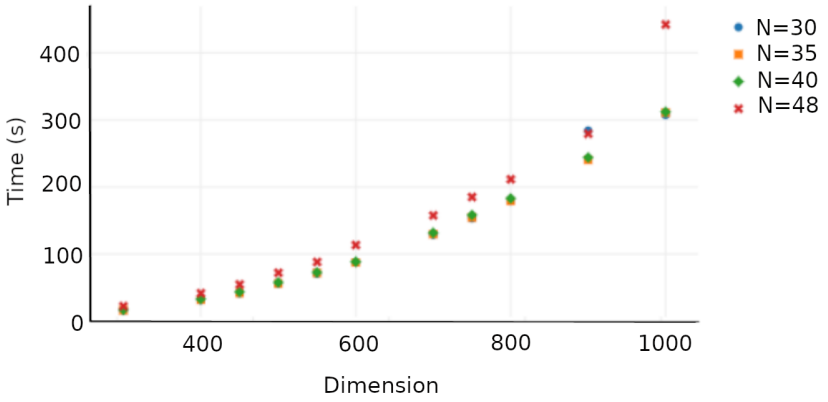


Figure 2.9: Execution time of the distributed MPI implementation for quadratic cost functions for different data sizes s and different number of nodes n, for data dimension larger than 200

transparent, as we use all-to-all communications. Regardless of the edges in the graph, each node communicates with all the other nodes all the time. This can get even worse, as the data size grows. In that case, a node has to exchange larger portions of data with all the other nodes. Therefore, even if we do not have a scalable implementation at this stage, it was of interest to experimentally quantify the parallelization and communication overhead effects. As the results demonstrate significant gains of parallelization and how to moderate losses due to communication overhead, the next steps include tests of non-quadratic cost functions, i.e. for logistic loss and over real data sets, while the implementation will be also accommodated, as already explained in Section 2.2.2.

The tests regarding the algorithm implementation for logistic loss functions

As the algorithm implementation that utilizes the all-to-all communication protocol does not scales well, i.e. the time reduction with increased number of workers on the same problem is not evident, a different approach is needed for the communication implementation. First, we show the difference in execution time, after introducing the communicators, compared to the all-to-all approach. Then, we evaluate the time consumption of different parts of the algorithm, after introducing communicators, in order to see how time-consuming different algorithm parts tend to be. Then, we also explore different possible algorithm variants with sparsified communications and pick the most promising ones for the final class of proposed methods.

Regarding the data distribution process, it does not consume a large amount of the execution time. For example, considering a data set that contains a matrix of 5000×6000 elements and a vector of 5000 elements, the initial setup, including reading and scattering the data, as well as the creation of the communicators, takes about 0.3s per process. When compared to the overall run-time of the tests, it represents a relatively small percentage. Regarding the case with the lowest execution time this percentage is 5%. On the other hand it is only 0.0007%, in the case with the highest execution time.

The simulation setup

For the tests regarding logistic loss functions, we use two different types of graphs for comparison: grid graphs and d-regular graphs, as instances of graph G, introduced in Section 2.1.1. We generate them as already described. See Section 2.2.2 for details.

As already mentioned, the input data are represented as an $R \times s$ sized matrix of features, and an R sized vector of labels. Both the matrix and the vector are then being divided into n parts, where n represents the number of nodes. For different tests in different stages, the following data sets were used:

- The Conll data set that concerns language-independent named entity recognition [123, 124]. It has $R = 220663$ and $s = 20$ as the input data sizes. This data set is only used for comparing the performance of the algorithm between regular and grid graphs, as well as for some initial tests.
- The Gisette data set [125, 126, 127, 128]. It is known as a handwritten digit recognition problem. Its input data sizes are $R = 6000$ and $s = 5001$. The data set is used for testing the different alternatives of the algorithm as well as for determining the most suitable value of d for d-regular graphs.

- The YearPredictionMSD train data set used to predict the release year of a song from audio features [129, 126, 130, 128]. The values R and s are $R = 463715$ and $s = 91$ here. The data set is also used for testing the different alternatives of the algorithm.

- The Mnist data set representing a database of handwritten digits [131, 132], with input data sizes $R = 60000$ and $s = 785$. This data set is also used for testing the different alternatives of the algorithm.

- The Relative location of CT slices on axial axis data set (referred to as CT data set in the further text), containing features extracted from CT images [133, 126, 134, 128]. The data sizes are $R = 53500$ and $s = 386$. This data set is also used for testing the different alternatives of the algorithm.

- The p53 Mutants data set [135, 126, 136, 137, 138, 128] (referred to as p53 data set in the further text), used for modelling mutant p53 transcriptional activity (active or inactive) based on data extracted from biophysical simulations. The data set sizes are $R = 31159$ and $s = 5410$. The data set is also used for testing the different alternatives of the algorithm.

We use these data sets for different kinds of tests, and for each test it will be noted on which data set it was concretely performed.

The value of the step size α in (2.2), can be fine-tuned according to the data set used for the tests. Increasing this value can lead to faster convergence. However, if the value is too large, then the algorithm might converge to a coarse solution neighbourhood. The values of α used for the mentioned data sets are obtained experimentally and are listed below:

- $\alpha = 0.0001$ for the Gisette data set;

- $\alpha = 0.001$ for the p53 data set;

- $\alpha = 0.1$ for the YearPredictionMSD, Mnist and CT data sets.

A larger value of $\alpha = 0.1$ can be applied in the cases of relatively small number of features, compared to the number of instances (i.e. rows of data). Here, in all the 3 cases for $\alpha = 0.1$, the number of features is smaller than 1000.

The experimental results regarding the benefits of introducing communicators

The tests on quadratic cost functions already showed the lack of scaling properties when using all-to-all communication protocol. We get the same conclusion for logistic loss functions as well. Therefore, after implementing the communication based on a list of communicators, it was of interest to prove weather the cost of creating this structure of communicators is smaller than the cost of communicating to all nodes all the time. We will illustrate this concept with a couple of examples, where we compare the two approaches.

Table 2.1: Examples of comparing the execution times for all-to-all communication and using communicators

Data set	**number of nodes**	**Time(s): all-to-all**	**Time(s): communicators**
Conll	26	2.81	2.61
Gisette	42	6926.05	6628.73

Table 2.1 illustrates two examples to compare the execution timings for the two implementations with different communication strategies: all-to-all and using communicators. The first example is on the Conll data set with 26 nodes, and the second one is on the Gisette data set with 42 nodes. These tests were performed on the AXIOM computing facility. We can see that the execution time reduction is evident after introducing communicators. It is naturally of smaller volume for a smaller data set and is more significant for a larger one. This supports the idea that the introduction of communicators represents a payable solution.

As one of the main concerns with the all-to-all communication approach is the lack of scaling properties, it is extremely important to investigate this property on the communicators-based implementation.

Fig. 2.10 shows the scaling properties of the algorithm implementation for logistic loss functions on the Gisette data set, after introducing communicators. These tests are performed on the AXIOM computing facility, for 8-regular graphs. It is now evident that the algorithm scales well. The most optimal number of nodes for the Gisette data set is 38 here. The execution time decreases while increasing the number of nodes n, until reaching this optimal point. After that, it starts to grow again. This is a completely normal occurrence, as after the optimal number of nodes was reached, increasing further this number is unnecessary and leads to poorer performance, as now the communication costs can be more extensive than the gains of parallelization.

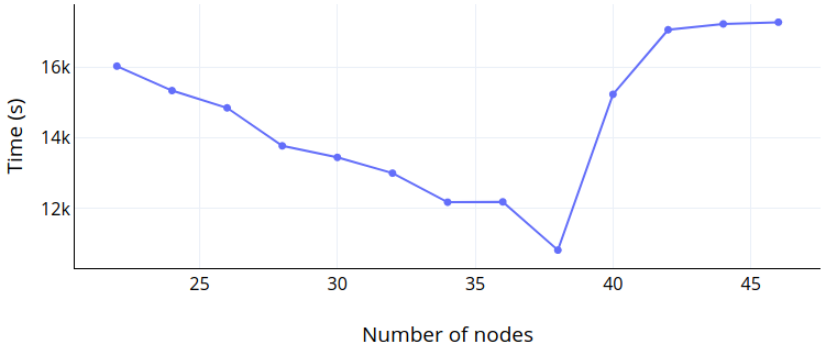


Figure 2.10: Scaling properties of the algorithm with communicators, on the Gisette data set

For now, we showed that the parallelization reduces the execution time and provides broader use than a serial implementation, as it can work on much larger data sets. Furthermore, we showed that introducing the concept of communicators leads to further algorithm improvements regarding performance, and that this implementation is also scalable. Now, we are interested in the time consumption of different parts of the algorithm, in order to direct our further enhancements of the implementation. For that reason, we conducted an experiment on the algorithm that uses communicators for the Gisette data set, on the PARADOX cluster this time (this cluster can provide a larger number of cores, i.e. processes for the execution). We run our tests for the following numbers of nodes $n = \{50, 60, 70, 80, 90, 100, 120, 140\}$. We measured the execution time of the following parts of the algorithm here: the communication, i.e. data exchange; the gradient computation; the Hessian computation; the checking if the stopping criterion is satisfied; the solution update computation. Table 2.2 shows the results.

Table 2.2: The percentages of execution time for different parts of the algorithm

Algorithm part	**percentage(%)**
Communication	71.39
Gradient	0.003
Hessian	61.56
Stop check	0.16
Solution update	20.91

The percentages in Table 2.2 are calculated in the following way: for each number of nodes n, we calculate the exact percentage for each part of the algorithm, and then we calculate the average on different parts of the algorithm. Each timing represents the time for the slowest process. For this reason, the sum of percentages in the table is not 100%. It serves

as a starting point for possibly considering further enhancements. First, although the algorithm performs much better after introducing the communicators, it still consumes the most of execution time on communication. That is the main motivation for communications sparsifications in the next iteration of development. Also, the computation of the Hessian is also very time consuming. This fact motivates the investigation of using first order methods, that omit this computation. The solution update is taking about 20% of the execution time, but it should be mentioned that we also incorporated the inverse matrix computation for the Hessian here. So, without that, the solution update is not time consuming at all. If we omit the Hessian inverse computation, then this percentage will be also negligible. The rest of the code does not seem to influence the performance significantly.

The experimental results for testing different possible methods with sparsification

At this stage, the main idea is to choose a set of possible algorithm variants, i.e. methods, that will be used for detailed performance evaluation later. There are three dimensions for creating these methods:

- The first one is the probability p_k for communication. We want to test the cases with increasing, decreasing and constant probability values. In other words, the communication probability can be defined to grow as $p_k = 1 - 0.5^k$, or to decrease as $p_k = (k+1)^{-1}$. We also investigate 3 different cases for constant probability values $p_k = 0.3, p_k = 0.5$ and $p_k = 0.8$.
- The second dimension is the second order information presence. The Hessian inverse approximation can be included in the computation during the iterations, or it can be replaced by an identity matrix.
- The third dimension is the type of the communication. This means that the algorithm can use bidirectional or unidirectional communicating principle.

While combining these dimensions, we get a list of possible methods to be tested. Table 2.3 lists them. For this stage of tests, the PARADOX cluster environment is used. The reason behind this is the possibility of this cluster to offer a larger number of processes. This is necessary, as we do not know at this stage the exact number of nodes that can satisfy our test requirements, so we cannot claim for now that the AXIOM cluster will be satisfactory. The implementation details for these dimensions were already described in detail in Section 2.2.2.

Table 2.3: Different methods to be tested

Method	Probability	Hessian inverse	Direction
1	increasing	yes	bidirectional
2	decreasing	yes	bidirectional
3	constant 0.3	yes	bidirectional
4	constant 0.5	yes	bidirectional
5	constant 0.8	yes	bidirectional
6	increasing	no	bidirectional
7	decreasing	no	bidirectional
8	constant 0.3	no	bidirectional
9	constant 0.5	no	bidirectional
10	constant 0.8	no	bidirectional
11	increasing	yes	unidirectional
12	decreasing	yes	unidirectional
13	constant 0.3	yes	unidirectional
14	constant 0.5	yes	unidirectional
15	constant 0.8	yes	unidirectional
16	increasing	no	unidirectional
17	decreasing	no	unidirectional
18	constant 0.3	no	unidirectional
19	constant 0.5	no	unidirectional
20	constant 0.8	no	unidirectional

There are 20 different methods of the algorithm to be tested, listed in Table 2.3. The goal of testing these 20 variants of the algorithm is to eliminate those that perform worse, and keep those that show better performance for further testing. In order to do so, we need to choose a graph type first to be used for these tests.

Choosing the graph type for the tests

We simply consider two types of graphs for graph G here, regular and grid graphs. The easiest and probably best way to choose the better option is empirically, to perform a set of tests regarding these graph types. These tests are performed on the algorithm that uses communicators, but without any sparsification. Here, we tested 7-regular, 8-regular, 9- regular, 10-regular and grid graphs, in order to identify the most appealing alternative. The Conll data set is used here for the tests.

Fig 2.11 represents the comparison between grid and regular graphs. For regular graphs value, we use the average for the mentioned 4 variants for d, when working with d-regular graphs. For a low number of nodes, regular graphs already perform better, but this difference is not so obvious. However, as the number of nodes grow, the difference in performance between these two types of graphs is evident. This leads to the choice of

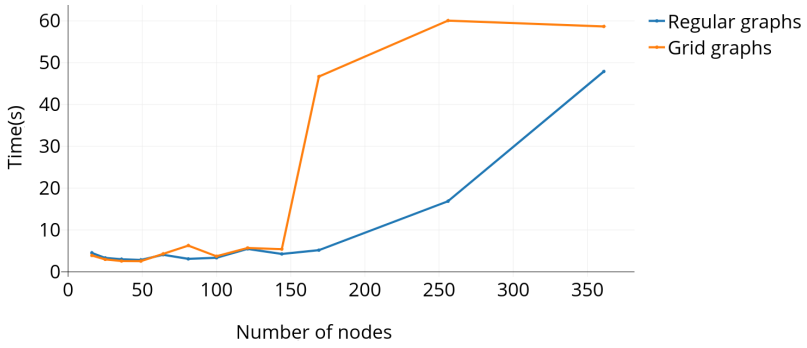


Figure 2.11: Comparing the execution time using regular and grid graphs for the same number of nodes, using the Conll data set

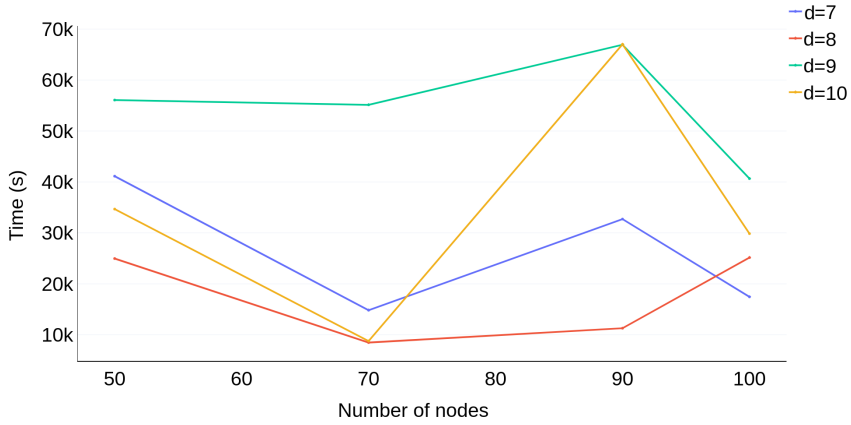


Figure 2.12: Comparing the execution time related to the value of d, using d-regular graphs and the Gisette data set

working with d-regular graph. In order to choose the most appropriate value for d, we can examine Fig 2.12, where the execution times for different variants of d-regular graph are presented. It can easily be concluded, that the best performing option is with 8-regular graphs. For that reason, we will work with them for the rest of the experiments. An extensive analysis of the graph topology is out of scope of this thesis, and represents a possible future research direction.

Choosing the best performing methods for the algorithm

Now, we can run the tests for the 20 mentioned methods of the algorithm with sparsified communication. We run these tests with 80 nodes first (in order to choose the "good" and eliminate the "bad" performing variants), with 8-regular graphs, on the Gisette data set again. As we need to set some time limit for execution, we set it to 5 hours for each job on the cluster. We will not display the results of all tests, but instead some of the interesting ones. The first important conclusion is that the methods that work

with constant probability values do not converge close enough to the solution for the defined time limit. They do approach it, as we can see from the log files, but they need extensively more time to converge. For that reason, we will omit to use these constant probability values for the rest of our tests, with one exception later, where we will explore the convergence. But for the relevant methods, that will be used for further tests, we will use only cases with increasing and decreasing probabilities.

Before moving on to the next section, where we will explore further the performance of these relevant algorithm methods, let us demonstrate a few interesting conclusions, based on the already performed tests. Table 2.4 shows a comparison between the 4 best performing methods (on the Gisette data set), for 50 nodes. The first value is present as a baseline and represents the algorithm without sparsification. It is obvious that introducing any sparsification leads to execution time reduction, that can be very extensive.

Table 2.4: Execution time for different methods for 50 nodes in the network

Method	**Execution time (s)**
No sparsification, second order method	24966.11
Bidirectional, decreasing probability, second order method	8166.08
Bidirectional, decreasing probability, first order method	789.92
Bidirectional, increasing probability, first order method	314.47
Unidirectional, increasing probability, first order method	16.28

The maximal execution time, i.e. the time for the slowest process, is taken into account for all the cases. It is evident that the second order method, i.e. the case with the Hessian inverse included, requires a larger amount of time, as the nodes spent more time on computation than the other variations where the Hessian inverse was excluded. Only the first version includes the Hessian inverse, the others use an identity matrix instead of it, as its computation is time consuming. As this amount of time can vary on different processes, all processes are waiting for the slowest one in the communicator in order to successfully exchange the data. When the Hessian inverse is excluded, all parts of the algorithm perform relatively fast, so there is no such large delay present. However, second order methods are of interest when the dimension of the optimization variable is sufficiently small or moderate sized and when the problem is difficult, so second order information pays off. The greatest time reduction occurs for unidirectional communication for this particular case, displayed in Table 2.4. This means that introducing the concept of idling, so that an idle process performs computation and gathers results from active neighbours without sending its own results, drastically decreases the communication time here. However, the data set used for testing highly influences the fact which methods

perform better on it. This aspect will be evaluated further.

Table 2.5: The average percentage of time spent on communication for different versions of the algorithm

Method	**Communication time**
No sparsification, second order method	71.39%
Bidirectional, decreasing probability, second order method	13.22%
Bidirectional, decreasing probability, first order method	25.7%
Bidirectional, increasing probability, first order method	11.27%
Unidirectional, increasing probability, first order method	9.7%

As we performed the tests, we also measured the execution time for different parts of the algorithm. The results show that the most time consuming part is the communication, which is not a surprising fact. Table 2.5 represents the average percentage of communication time, related to the overall execution time, for the different 4 best performing methods of the algorithm, and the base case, without sparsification (same as in Table 2.4). These percentages represent the average percentage of time spent on communicating, relative to the overall execution time, for a set of tests with different numbers of nodes in the network. Without sparsification, the algorithm still spends more than 70% on communication, although the all-to-all communication protocol was replaced with introducing a set of communicators. The idea that the data exchange is not necessary in every iteration for every node might seem interesting from this aspect. Decreasing the frequency of exchanging the results, the average communication time falls to 25.7% in the worst case. For the other methods, this percentage is around 10%. The value of 25.7% is unexpected, as the algorithm does not differ from the others in an extent to influence more communication. However, it should be kept in mind that each process generates a random value in the predefined range between 0 and 1. If the randomly generated value falls into the scope of the probability bound for that iteration, than the node is active. As a consequence, it can happen that even if the probability is small, the majority of processes still satisfy the condition to be active. It is evident that the performance significantly increases with the introduction of communication probability. Moreover, introducing directed communication could possibly make the algorithm even more efficient.

2.3.3 The experimental results for the selected set of methods

The previously conducted set of tests revealed the algorithm methods that are possibly worth for further analysis, in the sense that they may show good performance features.

However, the class of methods that we propose does not contain only those solutions that showed the best performance for the described test cases. Instead, it also includes the appropriate pairs of them, in order to ensure the ability of comparing the adequate possible approaches. From here, we consider the selected class of methods.

The methods

Here, we provide a complete list of algorithm methods, that we are interested in for further analysis. When considering the solution update:

$$x_i^{k+1} = x_i^k + d_i^k, \tag{2.59}$$

the following alternatives of algorithm (2.9)-(2.10) are considered.

- ***Method 0***: The initial version of the algorithm, used as the benchmark here, without sparsification (all the nodes are active all the time), the communication is always bidirectional, and the Hessian is included in the computation, so it is a second order method. More precisely, the method is defined by the following. For all $i = 1, \dots, n$, given x_i^k, we have

$$\xi_{i,j}^k = 1, p_k = 1, M_i^k = \alpha \nabla^2 f_i(x_i^k) + (1 - W_{ii})I, \tag{2.60}$$

$$d_i^k = -\Big[(M_i^k)^{-1}[\alpha \nabla f_i(x_i^k) + \sum_{j \epsilon \Omega_i} W_{ij}(x_i^k - x_j^k)]\Big]. \tag{2.61}$$

- ***Method 1***: Bidirectional communication, with increasing communication probability. Here, the Hessian approximation is replaced with the identity matrix, resulting in the following first order method:

$$\xi_{i,j}^k = z_i^k \cdot z_j^k, p_k = 1 - 0.5^k, M_i^k = I, \tag{2.62}$$

$$d_i^k = -\Big[\alpha \nabla f_i(x_i^k) + \sum_{j \epsilon \Omega_i} W_{ij}(x_i^k - x_j^k)\xi_{i,j}^k\Big]. \tag{2.63}$$

- ***Method 2***: Bidirectional communication, with decreasing communication proba-

bility and first order updates,

$$\xi_{i,j}^k = z_i^k \cdot z_j^k, p_k = \frac{1}{k+1}, M_i^k = I, \tag{2.64}$$

$$d_i^k = -\Big[\alpha \nabla f_i(x_i^k) + \sum_{j \epsilon \Omega_i} W_{ij}(x_i^k - x_j^k)\xi_{i,j}^k\Big]. \tag{2.65}$$

- ***Method 3***: Unidirectional communication, with increasing communication probability and first order method updates,

$$\xi_{i,j}^k = z_j^k, p_k = 1 - 0.5^k, M_i^k = I, \tag{2.66}$$

$$d_i^k = -\Big[\alpha \nabla f_i(x_i^k) + \sum_{j \epsilon \Omega_i} W_{ij}(x_i^k - x_j^k)\xi_{i,j}^k\Big]. \tag{2.67}$$

- ***Method 4***: Unidirectional communication with decreasing communication probability and first order updates,

$$\xi_{i,j}^k = z_j^k, p_k = \frac{1}{k+1}, M_i^k = I, \tag{2.68}$$

$$d_i^k = -\Big[\alpha \nabla f_i(x_i^k) + \sum_{j \epsilon \Omega_i} W_{ij}(x_i^k - x_j^k)\xi_{i,j}^k\Big]. \tag{2.69}$$

- ***Method 5***: Bidirectional communication, with increasing communication probability and second order updates,

$$\xi_{i,j}^k = z_i^k \cdot z_j^k, p_k = 1 - 0.5^k, M_i^k = \alpha \nabla^2 f_i(x_i^k) + (1 - W_{ii})I, \tag{2.70}$$

$$d_i^k = -\Big[(M_i^k)^{-1}[\alpha \nabla f_i(x_i^k) + \sum_{j \epsilon \Omega_i} W_{ij}(x_i^k - x_j^k)\xi_{i,j}^k]\Big]. \tag{2.71}$$

- ***Method 6***: Bidirectional communication, with decreasing communication probability and second order updates,

$$\xi_{i,j}^k = z_i^k \cdot z_j^k, p_k = \frac{1}{k+1}, M_i^k = \alpha \nabla^2 f_i(x_i^k) + (1 - W_{ii})I, \tag{2.72}$$

$$d_i^k = -\Big[(M_i^k)^{-1}[\alpha\nabla f_i(x_i^k) + \sum_{j\epsilon\Omega_i} W_{ij}(x_i^k - x_j^k)\xi_{i,j}^k]\Big]. \tag{2.73}$$

- ***Method 7***: Unidirectional communication, with increasing communication probability and second order updates,

$$\xi_{i,j}^k = z_j^k, p_k = 1 - 0.5^k, M_i^k = \alpha\nabla^2 f_i(x_i^k) + (1 - W_{ii})I, \tag{2.74}$$

$$d_i^k = -\Big[(M_i^k)^{-1}[\alpha\nabla f_i(x_i^k) + \sum_{j\epsilon\Omega_i} W_{ij}(x_i^k - x_j^k)\xi_{i,j}^k]\Big]. \tag{2.75}$$

- ***Method 8***: Unidirectional communication, decreasing communication probability and second order updates,

$$\xi_{i,j}^k = z_j^k, p_k = \frac{1}{k+1}, M_i^k = \alpha\nabla^2 f_i(x_i^k) + (1 - W_{ii})I, \tag{2.76}$$

$$d_i^k = -\Big[(M_i^k)^{-1}[\alpha\nabla f_i(x_i^k) + \sum_{j\epsilon\Omega_i} W_{ij}(x_i^k - x_j^k)\xi_{i,j}^k]\Big]. \tag{2.77}$$

- ***Method 9***: Bidirectional communication without communication sparsification. This is a first order method. It corresponds to *Method 0* without second order information.

$$\xi_{i,j}^k = 1, p_k = 1, M_i^k = I, \tag{2.78}$$

$$d_i^k = -\Big[\alpha\nabla f_i(x_i^k) + \sum_{j\epsilon\Omega_i} W_{ij}(x_i^k - x_j^k)\xi_{i,j}^k\Big]. \tag{2.79}$$

In order to make things clearer, we introduce a naming convention for the mentioned methods. Our convention for abbreviating the methods uses a three letter system, where the first letter represents whether the method is first or second order (F or S); the second letter represents the type of the communication (B for bidirectional and U for unidirectional); the third letter represents the communication sparsification type, i.e. the probability used for communication (I for increasing, D for decreasing and C for constant)

It can be seen that *Method SBC* represents the initial version of the algorithm, used as the benchmark here, where *Method FBC* is its first order equivalent. These are the

Table 2.6: Different alternatives of Algorithm 1

Method name	**Method number**	**Type**	M_i^k	$\xi_{i,j}^k$	p_k	**Relevant reference**
FBI	Method 1	First order	I	$z_i^k \cdot z_j^k$	$p_k = 1 - 0.5^k$	[40]
FBD	Method 2	First order	I	$z_i^k \cdot z_j^k$	$p_k = (k+1)^{-1}$	[43]
FUI	Method 3	First order	I	z_j^k	$p_k = 1 - 0.5^k$	novel method [46]
FUD	Method 4	First order	I	z_j^k	$p_k = (k+1)^{-1}$	[44]
FBC	Method 9	First order	I	1	1	[41]
SBC	Method 0	Second order	D_i^k	1	1	[12]
SBI	Method 5	Second order	D_i^k	$z_i^k \cdot z_j^k$	$p_k = 1 - 0.5^k$	[39]
SBD	Method 6	Second order	D_i^k	$z_i^k \cdot z_j^k$	$p_k = (k+1)^{-1}$	[39, 43]
SUI	Method 7	Second order	D_i^k	z_j^k	$p_k = 1 - 0.5^k$	novel method [46]
SUD	Method 8	Second order	D_i^k	z_j^k	$p_k = (k+1)^{-1}$	novel method [46]

only methods that do not utilize any communication sparsification. *Methods FBI, FBD, FUI, FUD, SBI, SBD, SUI, SUD* use sparsification with either increasing or decreasing communication probabilities p_k. The reason for choosing a linearly increasing p_k and a sub-linearly decreasing p_k is adopted according to insights available in the literature; see, e.g., [39], [44]. While it is possible to consider other choices and fine-tuning of the sequence p_k, this topic is outside of the thesis scope. Our primary aim is to investigate the feasibility and performance of increasing and of decreasing sequence of p_k's relative to the always-communicating strategy (*Method SBC* and *Method FBC*), as well as relative to the unidirectional versus bidirectional communication, and the first order versus second order methods.

The considered communication probability in [12] in Table 2.6, is 1. References [39, 40], in addition to communication sparsification, also consider sparsification in search directions (of second and first order, respectively). However, the analysis therein can be extended to also cover communication sparsification-only. Also, reference [44] considers a distributed estimation setting, but we also include it in Table 2.6 for completeness. The convergence analysis for the novel method with unidirectional communication *Method FUI* was presented in Section 2.1.3, while *Methods SUI* and *SUD*, that also rely on unidirectional communication, remain open for theoretical analysis in the future. The *Methods FBI, FBD, SBI* and *SBD*, using bidirectional communication are already analysed in the literature (see [39, 40, 41, 42, 43, 44]).

The listed methods and data sets described before, are used to derive some empirical conclusions. As expected, the analysis of obtained results provides some insights about the optimal number of nodes for different setups. Also, the advantages of particular

methods are clearly visible and one can estimate the usefulness of sparsification based on these results, keeping in mind that the tests might be influenced by the selection of data sets. Nevertheless, we believe that the obtained insights are useful.

The experimental results

The tests performed on the PARADOX cluster environment enabled to run our tests for a large number of nodes. However, the optimal number of nodes for a data set and particular method could not be clearly detected. Also, the scaling was not detectable above 80 nodes in a graph. This is quite easy to explain - it means that the optimal number of nodes is lower and that we do not need such a large number of processes in order to expose the scaling properties, for our setups. For that reason, we decided to explore the performance of the relevant methods, using a smaller number of nodes, i.e. less than 50. For these purposes, the AXIOM cluster is a reasonable choice. All the tests which follow are performed on this cluster environment. First, we will investigate further the behaviour of Algorithm 1 for two types of graphs - d-regular graphs and grid graphs. After that, we perform a sequence of tests using all the methods and the data sets stated above on d-regular graphs. These test are used to gain insight into effectiveness of different sparsification alternatives and differences between the first and second order methods in the framework of Algorithm 1.

The experiments regarding graph types

We already made a set of tests regarding the nature of the graph, that is the most appropriate for our experiments. We concluded that the 8-regular graphs represent the best choice. However, we wanted to test this property again for a lower number of nodes. We run these tests again for grid and d-regular graphs. However, we choose different values for d this time. The reason behind this is that the values for d were quite "close" in the previous set of tests. We want to explore more distant values now.

Fig. 2.13 and 2.14 represent a performance comparison between the executions of the algorithm using different d-regular ($d = 4$, $d = 8$, $d = 16$) and grid graphs with the *SBC* method on CT and Conll data sets, respectively. Observing Fig. 2.13, it can be clearly concluded that d-regular graphs perform better than grid graphs, which becomes more evident when increasing the number of nodes. This is equivalent to our previous conclusion, regarding this comparison. However, d-regular graphs perform similarly on this data set for different values of d, so it is hard to identify one of them as a best choice. The execution times for $d = 4$ and $d = 8$ are almost the same here. Therefore,

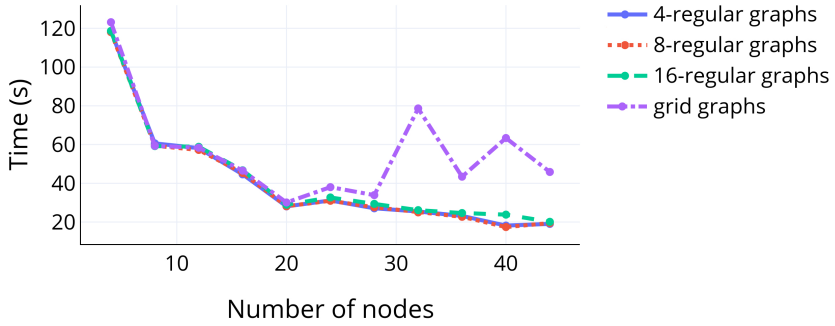


Figure 2.13: Comparing the execution time using regular and grid graphs for the same number of nodes, using the CT data set, for the *SBC* method

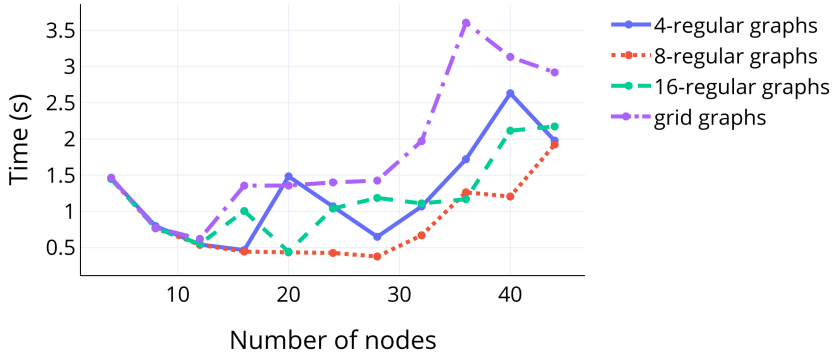


Figure 2.14: Comparing the execution time using regular and grid graphs for the same number of nodes, using the Conll data set, for the *SBC* method

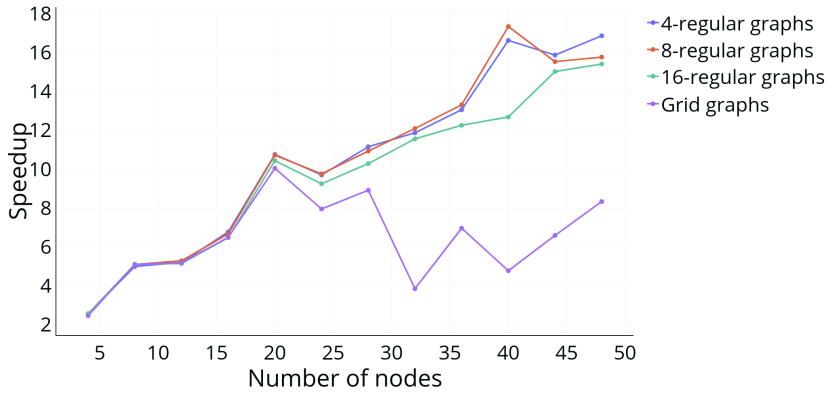


Figure 2.15: Speedup for the *SBC* method for different graph types on the Conll data set

it is important to examine the performance for different graphs on another data set, in order to open the possibility to gain additional information regarding the differences

between variants of d-regular graphs. From Fig. 2.14, it is evident that the execution time decreases until the optimal number of nodes is reached, and starts to grow after that point. The same trend is present in Fig. 2.13, but the optimal number of nodes is higher here. Fig. 2.14 clearly shows the difference between d-regular and grid graphs again. It also identifies 8-regular graphs as the most suitable choice for different number of nodes, as these graphs mostly have the lowest execution time. This is again the same conclusion, as we made for larger number on nodes and different values of d previously.

Let us now consider the speedup of the algorithm for these alternatives. The speedup of an algorithm can be also presented graphically, e.g in [139]. Consider an example for this on Fig. 2.15, that displays the speedup for the SBC method, on the Conll data set, for the mentioned underlying graph structures. The speedup value on y axis can be calculated using Amdahl's Law [140], as a ratio between the execution time required for one worker, divided by the execution time for n workers. In an ideal case, we could expect a linear speedup. Fig. 2.15 shows that a certain speedup exists for all considered graph types. Grid graphs have the weakest acceleration, where the considered different variants of d-regular graphs perform much better. In fact, 4-regular and 8-regular graphs have very similar speedups, but 8-regular graphs still perform slightly better, in most cases. All the tests performed in order to determine the most suitable graph type, resulted in the empirical conclusion that $d = 8$ is the most appropriate value when the number of nodes is at least 8. For the cases, where the number of nodes n is less than 8, the value $d = n - 1$ is used, leading to all-to-all graphs for $n < 8$.

From this point, we perform a sequence of tests using the described set of methods and the data sets stated in Section 2.3.3, on 8-regular graphs. These test are used to gain insight into effectiveness of different sparsification alternatives and differences between first and second order methods in the framework (2.9) - (2.10).

The experiments regarding execution times of methods

Let us start the examination by comparing the execution times for all the considered methods, for the same concrete example. Table 2.7 lists the execution time for each of the 10 methods for the p53 data set and 20 nodes in the network. As always, the maximal execution time, i.e. the time for the slowest process, was taken into account for all the cases. As this amount of time can vary on different processes, all processes are waiting for the slowest one in the communicator in order to successfully exchange the data. As the table shows, all first order methods introduce significant execution time reduction. In this case, the *Method FBD* has the best performance. When comparing *Method FBC* to

Table 2.7: The execution time for different variations of Algorithm 1 (2.9)-(2.10), for 20 nodes in the network, on the p53 data set

Method	**Execution time (s)**
FBI	4.64
FBD	1.89
FUI	6.04
FUD	3.56
FBC	3.16
SBC	9661.42
SBI	43126.71
SBD	22683.84
SUI	22029.20
SUD	9651.77

Table 2.8: The execution time for different variations of algorithm (2.9)-(2.10), for 12 nodes in the network, on the Mnist data set

Method	**Execution time (s)**
FBI	336.31
FBD	118.16
FUI	353.31
FUD	342.59
FBC	161.33
SBC	19045.00
SBI	3124.56
SBD	11853.99
SUI	12259.79
SUD	N/A

Method SBC, it is clear that the computation of second order direction d_i^k significantly increases the execution time, as these methods differ only in this dimension. Reducing the amount of communication across the iterations with *Method FBD* leads to even faster execution here. However, this behaviour may be highly dependent on the nature of the data set itself. The algorithms for p53 data set converge generally fast, within relatively small number of iterations. An equally important aspect here is also the fact that *Method FUD*, using unidirectional communication and decreasing communication probability performs better than *Method FBI*, with bidirectional and increasing communication probability. Observing the execution times for the second order methods proves that introducing communication sparsification mostly does not pay off as the computation of the second order direction is time consuming here.

As the nature of the data can highly influence the results, let us consider an another example that compares the execution timings for the methods, but on a different data

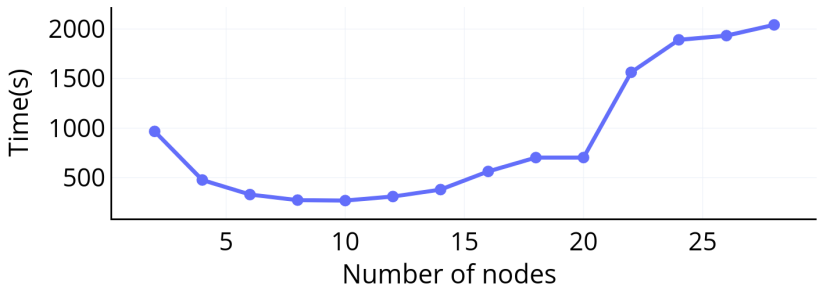


Figure 2.16: Scaling properties of Method FBI, for the YearPredictionMSD data set

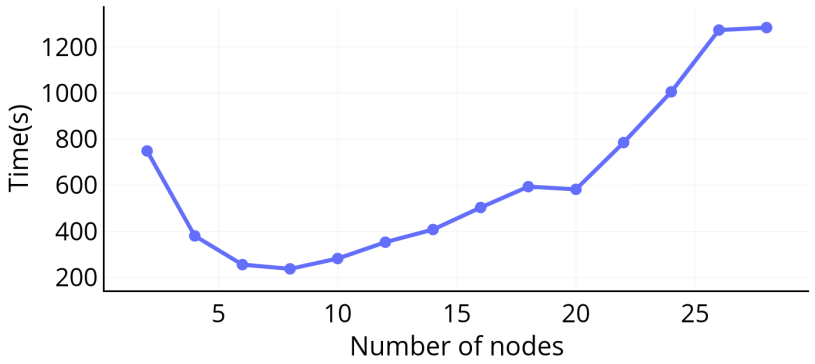


Figure 2.17: Scaling properties of Method FUI, for the Mnist data set

set and with different number of nodes in the network. Table 2.8 contains the execution times for each of the 10 methods, with 12 nodes for the Mnist data set. In this example (*Method SUD* does not converge for the given execution time limit).

The behaviour of methods on this data set differs from the behaviour on the p53 data set, observed in Table 2.7, as now we have longer execution timings. This is not surprising, as these data sets produce quite different behaviour. For example, for 12 nodes *Method FBD* requires 4795 iterations to converge for the Mnist data set. When considering the p53 data set for the same setup with 12 nodes, it converges after only 3 iterations.

However, the conclusions based on Table 2.8 are very similar to those from Table 2.7. In fact, it seems that the properties of particular methods are similar as long as the data sets are of similar volume. Generally, there exists a large decrease in execution time, when using communication sparsification. This corresponds also to the conclusions made on Table 2.4. The next important task is to examine the scaling properties of the methods.

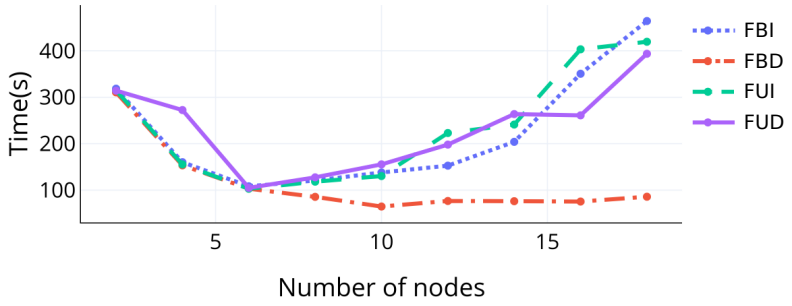


Figure 2.18: Execution times for the first order methods on CT data set

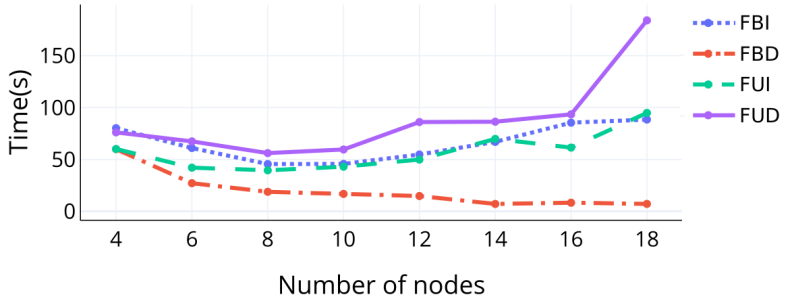


Figure 2.19: Execution times for the first order methods on Gisette data set

The experiments regarding scaling properties

A results of sequences of tests with different number of computational nodes n are shown next to give an insight into the most suitable number of nodes for a particular data set. Fig. 2.16 and Fig. 2.17 represent examples of the scaling properties of the algorithm, for *Method FBI* on the YearPredictionMSD data set and for *Method FUI* on the Mnist data set, respectively. Here, when varying n we keep the graph structure to the 8-regular graph all the time. The optimal number of nodes can be identified in both cases. These graphs obviously show the usual and expected trend where the execution time decreases until the optimal number of nodes is reached, while after that, further enhancement in number of nodes leads to time increase. Intuitively, the larger number of workers n means that the same overall workload is parallelized over more workers, leading to time reduction. However, the benefit effect is lost for a sufficiently large n when the communication overhead time starts to dominate. Interestingly, the optimal number of nodes is mostly constant for the first order methods as well as for the second order methods irrespective of the data set.

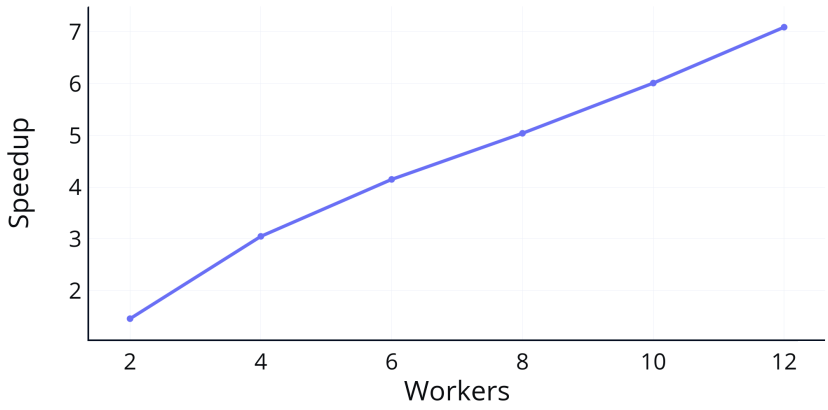


Figure 2.20: Speedup for the FBD method on the Mnist data set

In order to examine the performance of first order methods in one place, let us take a look at two different examples. Fig. 2.18 and Fig. 2.19 represent the execution times for first order methods with communication sparsification for the CT and Gisette data sets, respectively. From Fig. 2.18, it can be concluded that the optimal number of nodes for *Methods FBI, FUI* and *FUD*, is the same value $n = 6$. However, *Method FBD* performs differently. It shows lower execution time values generally, and its optimal number of nodes is $n = 10$. Similar conclusions could be made based on Fig. 2.19. Here, the optimal number of nodes for *Methods FBI, FUI* and *FUD* is again the same value, $n = 8$. *Method FBD* also performs differently here, with lower execution time values, compared to other first order methods. The optimal number of nodes for the second order methods tends to be a larger number generally. This is a direct consequence of the fact that the time consuming computations for the direction are faster with smaller portions of data on a node.

Fig. 2.20 shows the speedup of the FBD method on the Mnist data set. The number of nodes on x axis is in range from 2 to 12. The reason for choosing this interval is that the "sweet spot", i.e. the optimal number of nodes for this test case is 12. Considering the results on Fig. 2.20, it can be seen that the speedup of the FBD method is satisfactory. Similar conclusions can be made for the rest of the methods, as well.

The results regarding convergence percentage and cost reduction

We should also take into consideration to what extend the methods converge, when observing all the performed tests. Table 2.9 shows the percentages of successful tests for all methods, i.e., of tests that satisfy the stopping criteria $\|\Phi(x^k)\| < 0.01$ within maximal execution time of 15 hours. We enlarged the time limit here (compared to the tests for large numbers of nodes from the previous stage of testing), in order to provide a broader possibility for tests on different data sets to converge. The failed tests were also

Table 2.9: Percentages of successful test with respect to the overall number of tests

Method	Percentage
FBI	99.1
FBD	100
FUI	100
FUD	100
FBC	100
SBC	98.3
SBI	84.1
SBD	95.8
SUI	95.8
SUD	35

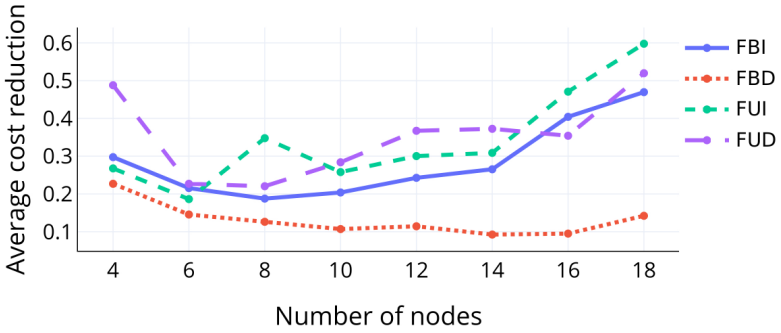


Figure 2.21: Average cost reduction compared to the worst relevant tested method for each problem, for *Methods FBI, FBD, FUI* and *FUD*

approaching the solution, but they did not reach it within the given time limit. The results indicate that the first order methods are better choice in this environment. *Method SUD* is the method with the smallest number of successful tests. This fact can be easily explained as it computes the expensive second order direction and the communication probability decreases while the communications are unidirectional. All this lead to the lack of communication epochs in order to ensure convergence during the time consuming iterations.

In order to make an additional comparison between first order methods with communication sparsification, let us examine Fig. 2.21. It represents the average cost reduction for different number of nodes, compared to the method of the weakest performances for each data set, where the average is taken across different data sets. Basically, this figure covers all the tests performed on first order methods with communication sparsification, i.e., *Methods FBI, FBD, FUI* and *FUD*. For each data set, we divide the execution time for a given number of nodes with the worst execution time on the same data set, and

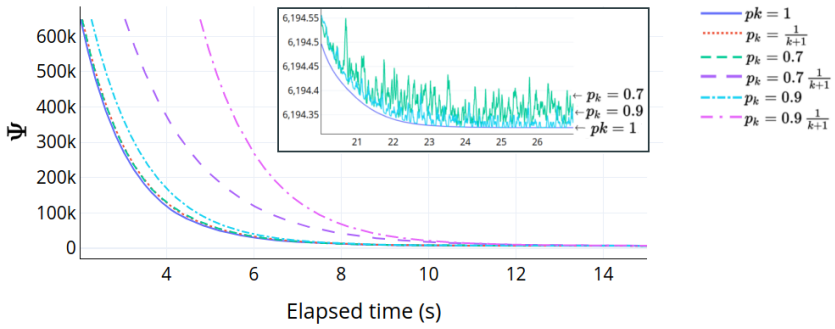


Figure 2.22: The comparison between using different values of $p_k \leq 1$ for directed first order method with unidirectional communication, on Conll data set

compute the average value over methods for all the data sets, for different numbers on nodes.

This is how we compute the average cost reduction on y axis of Fig 2.21. The conclusions based on this figure are consistent with the ones in Fig. 2.18 and Fig. 2.19. *Method FBD* has the best performance properties. Also, for each method, an optimal number of nodes can be easily identified.

The evaluation of the execution with different sequences of p_k

In order to examine the convergence properties and behaving of the algorithm for different probabilities, we can observe Fig 2.22. An evaluation of the algorithm execution with different sequences $\{p_k\}$ that stay bounded away from one as k grows large is presented in Fig. 2.22.

The unidirectional, first order method was tested on the Conll data set, using the step size value $\alpha = 0.1$. We observed the value of Ψ as in (2.2) during the execution of the algorithm. The value of Ψ decreases over time for all choices of p_k, as expected. The zoomed part of the figure is included in order to present the last few seconds of the execution before reaching the minimal values of Ψ. Fig. 2.22 shows that for different values of p_k the iterative sequences do not converge to the same value, but also that for the constant p_k choices the obtained limiting values are close.

Comparing the algorithm to ADMM

As problem (2.1) can be solved using the Alternating Direction Method of Multipliers (ADMM) [3], we compare Algorithm 1 to an ADMM implementation for logistic regression [122], on the Conll data set. We want to investigate how the performance of our algorithm relates to the performance of an ADMM implementation, when the implementations are using the same framework. More precisely, the method in [3] solves problem

Table 2.10: Comparison of the second order *Methods SBC* and *SBI* with ADMM

Method	**Execution time**
ADMM	4.487
Method SBC	0.247
Method SBI	0.226

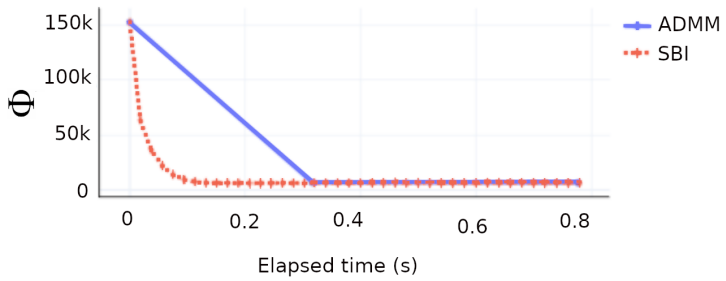


Figure 2.23: The comparison between ADMM and Method SBI on Conll data set

(2.1) assuming the presence of a central node that communicates to all other nodes in the network. Henceforth, we adapt our algorithmic framework to the latter setting by letting the underlying network G to be fully connected and by setting the matrix W to have all its entries equal $1/n$. This is easy to set up in our implementation.

The comparison between the second order *Methods SBC* and *SBI* and ADMM is shown in Table 2.10. We calculate the value of $\Phi^k = \frac{1}{n}\sum_{i=1}^{n} f(x_i^k)$, i.e., the average global cost in (2.1) averaged across all nodes' estimates, at the end of each iteration and we also measure the execution time. The implementation of this cost calculation was already explained in Section 2.2.3. The second column in Table 2.10 represents the time required to satisfy the condition $\frac{\Phi^k - f^*}{f^*} < 0.1$. Here, $f^\star$ is numerically evaluated by ADMM. The rationale for this comparison is the following. Our proposed methods converge to a neighbourhood of the solution to (2.1), while ADMM converges to the exact solution of (2.1). Therefore, it is meaningful to compare the times that each method needs to reach a certain accuracy level, measured with respect to the cost function in (2.1). We tested all the methods, and finally included the results for the best performing second order methods, i.e. *Methods SBC* and *SBI*. More precisely, *Method SBI* (a second order method with sparsification)

is here the best performing method across all methods, while *Method SBC* is taken as the baseline (second order) method without sparsification. The fact that second order methods perform better than first order methods here is consistent with our previous conclusion that for smaller data sets, second order methods perform better than first order methods. It is clear that our second order methods converge faster than ADMM. Fig. 2.23 shows the comparison between *Method SBI* and ADMM. *Method SBI* takes a larger number of significantly faster iterations, compared to ADMM, and hence results with shorter execution time needed to approach the solution.

The performance profiles

Fig. 2.24 - 2.31 displays the performance profile [141] for the described methods. Performance profiles enable evaluating the performance of different solvers running on a large number of tests. The performance measure is the execution time here. It is noticeable that the value range on the x axis is large, on these figures. This is due to the fact that there are very large differences in execution times, ranging from a few seconds to values larger that 18000 seconds. So, we consider the execution time as the comparison criterion. To compute the performance profile let us denote the execution time for a method M_i and test problem j by T_i^j. Then, given the value on the x-axis $\beta \geq 1$, the method M_i obtains a point for the performance on test j if there holds $T_i^j \leq \beta T_{min}^j$, where T_{min}^j is the smallest execution time of all tested methods considering that problem, i.e., $T_{min}^j = \min_i T_i^j$. The performance profile for a given β of the method M_i is then calculated as the number of points divided by the number of the performed tests. For example, on the y-axis where the parameter $\beta = 1$, we obtain the statistical probability that the method is the best one among all the tested methods in terms of the execution time.

Fig. 2.24 shows the performance profile for all the test on all data sets for the all 10 methods, where Fig. 2.25 and Fig. 2.26 display the performance profile for first and second order methods, respectively. Fig. 2.24 and Fig. 2.25 identify *Method FBD* as the best choice within the framework for Algorithm 1. This is accordant with the previous conclusions we made.

The performance profile evaluation gives us the opportunity to compare the pairs of first and second order methods, that use the same sparsification strategy. Observing the methods without sparsification, i.e. *Methods SBC* and *FBC*, Fig. 2.24 indicates that the first order method, *Method FBC*, performs better than the second order method, *Method SBC*. The same is true if we consider the methods with sparsification. Considering methods with decreasing communication probability and using bidirectional communication, *Method FBD* performs clearly much better than *Method SBD*. When comparing the other

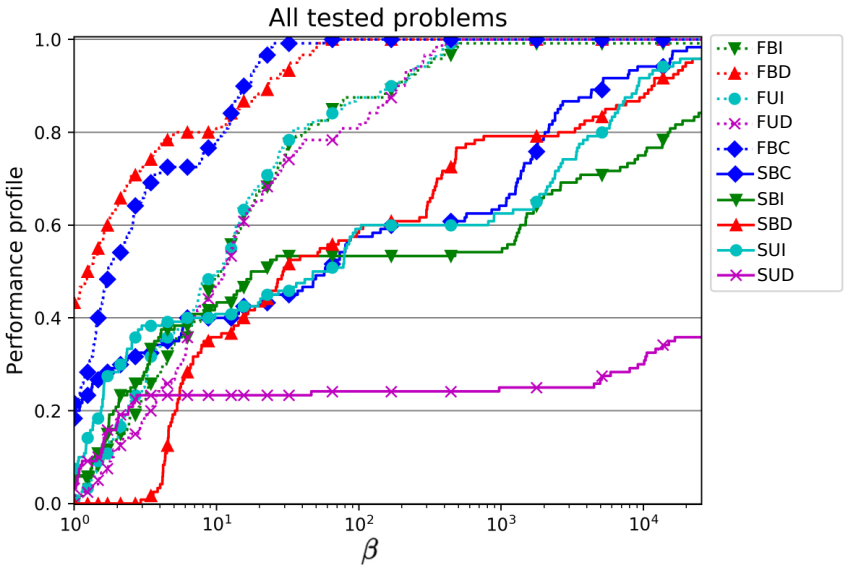


Figure 2.24: The performance profile for the all 10 methods, based on all the performed tests

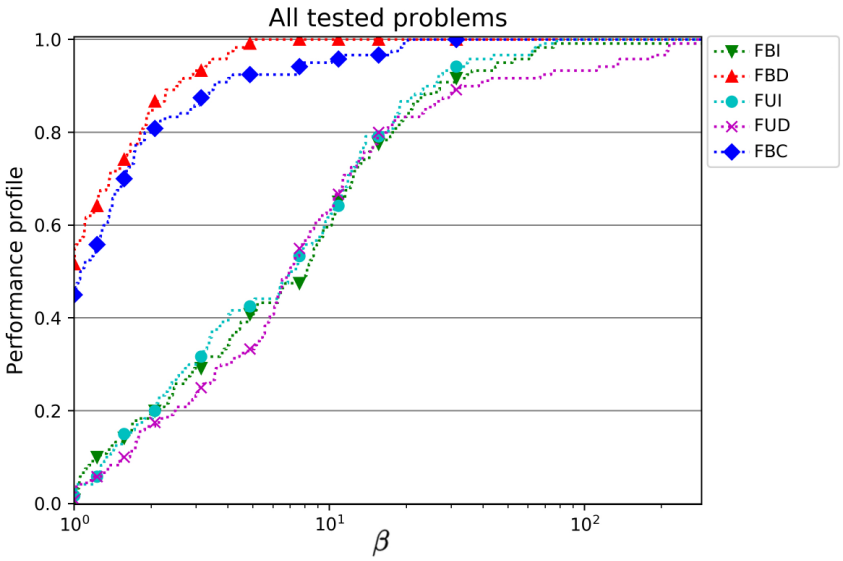


Figure 2.25: The performance profile for first order *Methods FBI, FBD, FUI, FUD* and *FBC*, based on all the performed tests

first and second order methods using the same sparsification (*Method FBI* and *Method SBI*, *Method FUI* and *Method SUI*, *Method FUD* and *Method SUD*), first order methods performs better in 61% of test cases. Also, the convergence rate for first order methods is higher (See Table 2.9).

It can also be concluded that the sparsification of second order methods gives no significant advantages probably because the computation of the second order direction is time consuming by itself (from Table 2.5, it can be seen that the average percentage

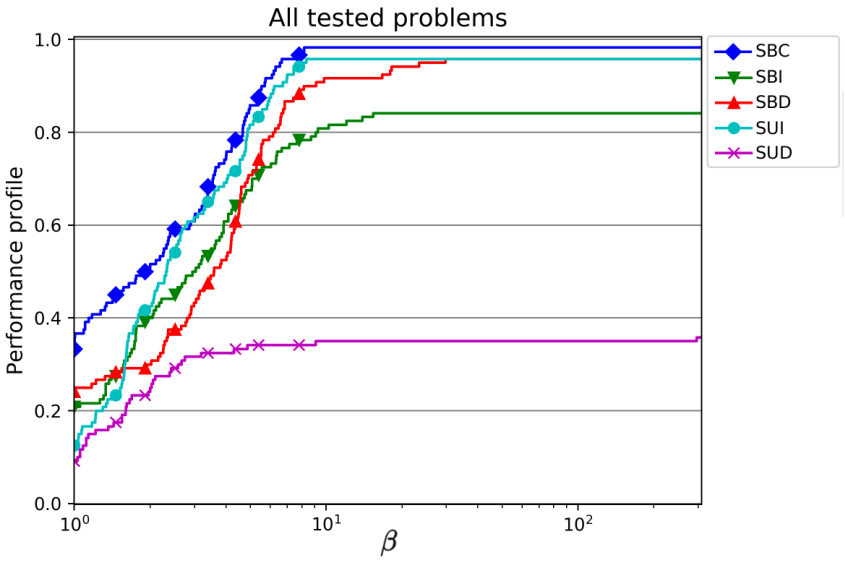


Figure 2.26: The performance profile for second order *Methods SBC, SBI, SBD, SUI* and *SUD*, based on all the performed tests

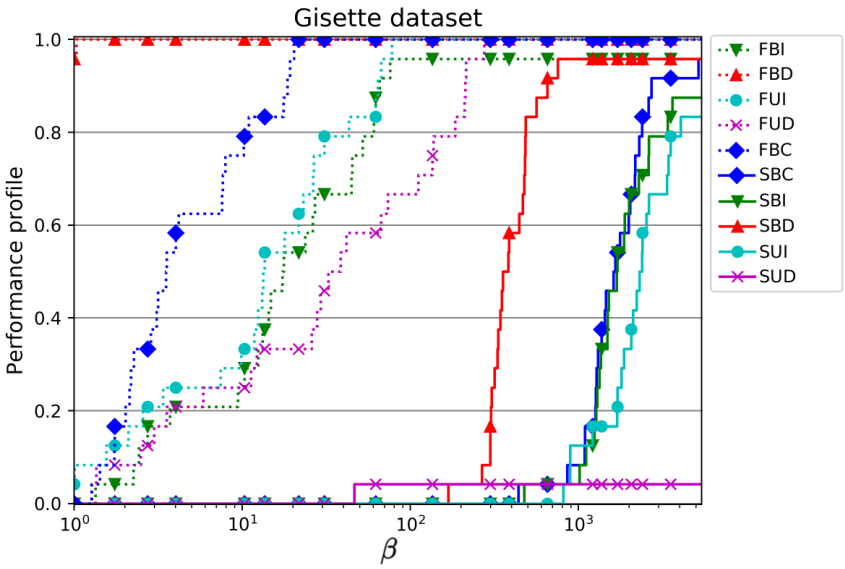


Figure 2.27: The performance profile for the all 10 methods, based on the tests performed on the Gisette data set

of time, spent on communication does not depend on whether the method is of first or second order). Furthermore, with communication sparsification the second order information is incorporated only partially and hence it does not provide enough advantage to compensate for computational load. On the other hand, communication sparsification can be beneficial for the first order methods, as evidenced by *Method FBD*. Generally, the best performing method is a first order method using the appropriate sparsification (bidirectional with decreasing communication probability), *Method FBD*.

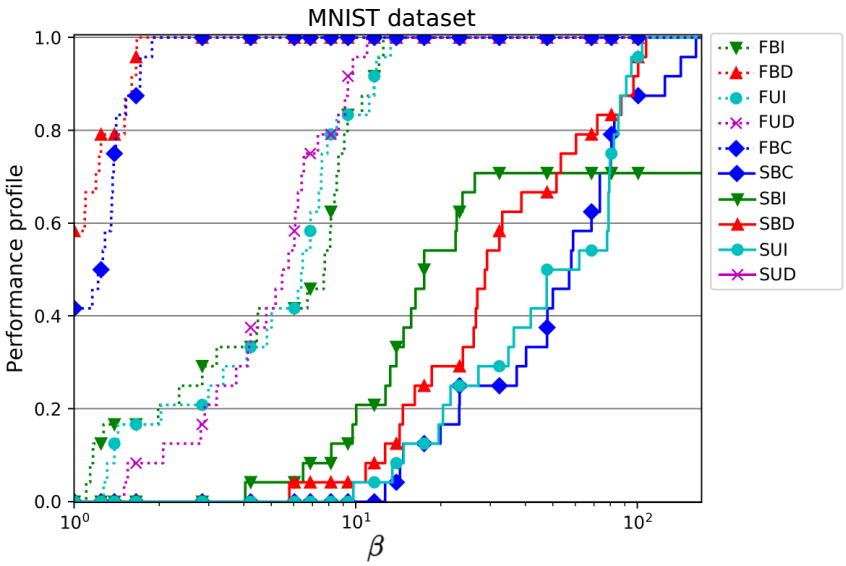


Figure 2.28: The performance profile for the all 10 methods, based on the tests performed on the Mnist data set

Now, we will take a look at performance profile graphs regarding different data sets separately. Fig. 2.27 represents the performance profile for the tests on the Gisette data set. Here, *Method FBD* can be also identified as the most suitable, followed by *Method FBC*, and later by *Method FUI*, *Method FBI* and *Method FUD*, where the second order methods show poorer performance profiles. The dimension s for this data set is a large value $s = 5001$, resulting with time consuming calculations in the second order methods as the Hessian approximation matrices are of large dimensions. Therefore, the first order methods perform better than second order methods. Here, the only sparsification strategy that truly pays off is the approach of *Method FBD*. Fig. 2.29 displays the performance profile for the tests on the p53 data set. The conclusions for this data set, are very similar to those for Fig. 2.27. Similarly, the dimension s is also a larger value here, $s = 5410$, so the first order methods also performs better than the second order methods and again, *Method FBD* represents the best choice. Similar conclusions are emerging from Fig. 2.28, that represents the performance profile for the Mnist data set. Here, the dimension $s = 785$ is around 6 times smaller, compared to Gisette and p53 data sets, but the dimension $r = 60000$ is 10 times larger than for Gisette, and 2 times larger than for p53. This results with similar load when distributing the data and calculation of the second order direction is too costly again.

The performance profile for the CT data set is displayed on Fig. 2.30. Here, the second order method, *Method SUI* dominates, as the data set dimension $s = 386$ enables faster calculations of the second order direction, and therefore it pays of. Comparison between first and second order methods with the same communication sparsification yields the

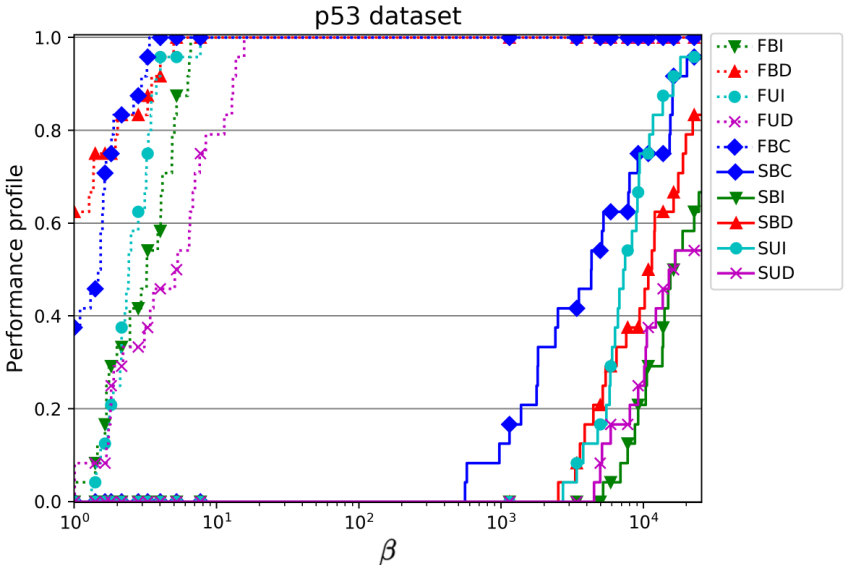


Figure 2.29: The performance profile for the all 10 methods, based on the tests performed on the p53 data set

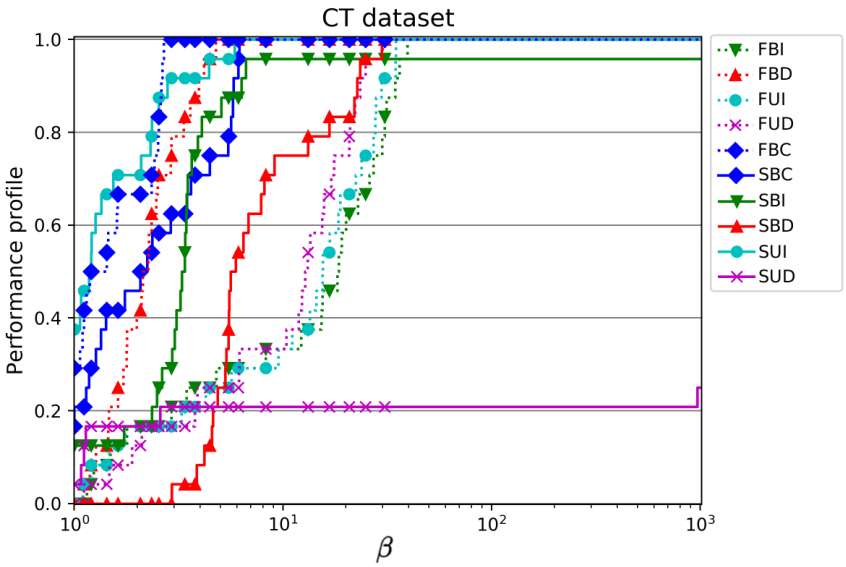


Figure 2.30: The performance profile for the all 10 methods, based on the tests performed on the CT data set

following conclusion - with the increasing communication probability the second order methods (*Methods SBI* and *SUI*) perform better (for both unidirectional and bidirectional communication). With the decreasing communication probability the first order methods (*Mehods FBD* and *FUD*) give better results. This gives us a new insight, as we can see that data sets with lower data dimensions can benefit from second order information.

Fig. 2.31 represents the performance profile for the YearPredictionMSD data set. Here, the dimension $s = 91$ is the smallest among the observed data sets. Therefore the second

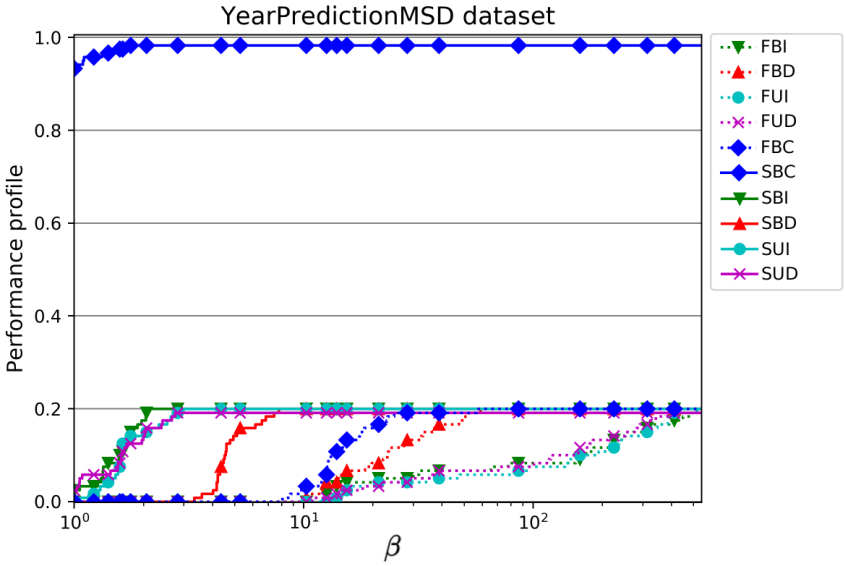


Figure 2.31: The performance profile for the all 10 methods, based on the tests performed on the YearPredictionMSD data set

order methods performs better, as we already saw for the Mnist data set also. But the sparsification does not improve the first order nor the second order methods for these data. This fact might be explained by the large dimension $R = 463715$. Each node gets a large subset and sparsifying the communication means ignoring a large portion of data on idle nodes, even if there is only one idle node. Thus, the gradient and Hessian are poorly approximated with idling.

2.4 Conclusions on the proposed class of primal methods

The tests were performed on an MPI cluster with a usual configuration, where each cluster node contains one processor with 6 CPU cores, and the nodes are connected by an Ethernet network with speed of 10Gbps. Execution and results may depend on the speed of the cores themselves and on the speed of the network. Given that we used a cluster with eighth-generation Core i7 cores, a performance jump can be expected if newer-generation CPUs and / or more powerful Xeon processors were used. This effect would refer to the shortening of the absolute execution time per core, but overall performance characteristics would remain the same. The scaling properties would still be present, as well as the preferences of certain methods for the specified scenarios regarding the data. The factors that can mostly affect the execution of the program are the speed and the latency of the network. In the case of clusters with higher network speeds and lower latency, the general expectation is to achieve good program performance with more nodes than in our experiments. In these cases, communication saturation, which we have shown to be present in this type of algorithm implementations, could only occur with more nodes involved than in our experiments (see Fig. 2.16 and 2.17, as well as Fig. 2.18 and 2.19 and the corresponding descriptions, that show these results for our experiments). In other words, increasing the network speed, with reduced latency, would be a crucial factor that would increase the number of nodes on which the proposed implementations can be executed efficiently. This could result in different values for the optimal number of cores in different setups, compared to the results on Fig. 2.16, 2.17, 2.18 and 2.19.

This chapter contains a detailed explanation of developing a parallel, scalable implementation of a set of distributed optimization methods. It subsumes several existing methods, but also discusses some new directions, both theoretically and empirically. The different phases and possibilities of the solution development are shown, while focusing on overcoming the detected bottlenecks regarding performance. We showed how the implementation can be refined and make more efficient in terms of resolving its most critical parts. The implementation was developed for strongly convex quadratic cost functions and for logistic loss functions as well.

We considered a class of first and second order distributed optimization methods which utilize different versions of the communication sparsification strategy. This means using different communication probabilities, combined with unidirectional or bidirectional communication between the nodes. The concept of unidirectional communication represents a novelty in the considered class of methods.

The thesis provides a comprehensive empirical evaluation of various communication sparsification strategies. The experimentation was performed on a cluster environment. We discuss the results for different implementation strategies, and for the different methods implemented. The overall execution time is observed for different data sets in order to identify the most suitable methods for different setups. Also, an evaluation of the influence of the nature of graph used to connect the nodes was performed. Additionally, a comparison to an ADMM implementation was also performed, in order to gain insights into the way how primal and dual methods relate, when solving the same problem.

The analysis of the methods showed that they posses the expected scaling properties, while the differences in the optimal number of nodes for a particular data set in consideration are evident for various methods. The comparison of different properties of methods was also performed. The performance profile showed the comparison between the proposed methods on different data sets separately, and for all tests together. It was clearly identified that the first order methods perform much better with larger volumes of data, where for smaller data sets the second order methods are more suitable. For data sets with larger number of features (10^3 or more in our tests), the portions of data that the processes work on demand a significant amount of time to calculate the second order updates. If the number of samples is also larger (larger than 10^3 for our tests), it additionally burdens the execution. This is the reason why the first order methods perform better on larger data sets. The first order methods converge within a larger number of iterations, but those iterations are multiple times faster than for the second order methods. When the data set is smaller, obtaining the second order information is not costly as the processes are working on small data portions. On these data sets the second order methods perform better as they converge within smaller number of iterations than the first order methods, while the second order iterations are negligibly slower than for the first order methods.

The method with bidirectional communication and decreasing communication probability, i.e. *Method FBD*, was identified as the best performing first order method. This method also shows the best performance globally, when observing all the tests on all 5 data sets. The fact that the bidirectional method performs better than the unidirectional method in most of the cases is a consequence of enabling exchange only between active nodes. Unidirectional methods require additional communication lines, in order to enable receiving data for idle nodes from their neighbors. The gain from solution update for the idle nodes can be slightly smaller than the cost of the communication to achieve that update. The decreasing probability leads to more communication in the beginning of the execution. Later, the communication becomes sparse, but at the same time the solution becomes

closer to the desired one, so that it does not require much communication any more. This is the reason why decreasing communication probability with a bidirectional method represents an optimal choice. However, the other methods with communication sparsification also showed satisfactory performance. The tests showed that, in general, communication sparsification can significantly improve performance. This serves as motivation for using communication sparsification in the described framework.

An important aspect of tests was the comparison between bidirectional and unidirectional communication. One conclusion is that unidirectional communication strategy works in the framework of (2.9)-(2.10), and thus confirm the theoretical results. Besides that, this strategy yields lower execution time than the bidirectional communication strategy for some test cases. All these conclusions might be influenced by the considered data sets but nevertheless provide significant empirical evidence.

Chapter 3

A dual distributed optimization method

The Alternating direction method of multipliers (ADMM) represents a dual method, that is in common use nowadays when it comes to distributed optimization problems. The reason behind this is that ADMM is a general purpose method that works under a very general setting on the underlying optimization problem and usually exhibits good performance on a wide variety of problems. This fact goes hand in hand with the growing interest in parallel application development, resulting with more efficient and scalable programs, applicable to larger amounts of input data. Therefore, this thesis also focuses on ADMM as a representative of a dual optimization method. Today, the need for efficient machine learning algorithms is growing with the need to process large amount of data as fast and accurate as possible. Clustering is an important unsupervised learning method, that find its use in different domains. Therefore, it is of great interest to provide distributed, parallel clustering solutions. In this chapter, a parallel, ADMM-based clustering algorithm is being described and practically evaluated. The implementation utilizes the manager-workers communication model, unlike the fully distributed model, used in Chapter 2.

3.1 Background theory

First, the theoretical aspects behind the parallel ADMM-based convex clustering algorithm need to be explained. The algorithm utilizes and adapts the idea of Sum Of Norms (SON) clustering algorithm and applies the ADMM approach.

SON clustering. SON clustering represents a convex relaxation of k-means clustering. It can be defined as follows. Consider the problem of clustering a set of observations

$\{a_j\}_{j=1}^N$, $a_j \in R^d$, where the number of clusters is not known in advance. The SON (Sum Of Norms) clustering is formulated as:

$$\min_x \sum_{j=1}^{N} ||a_j - x_j||^2 + \gamma \sum_{i<j} ||x_i - x_j||, \tag{3.1}$$

where $x = \left((x_1)^\top, (x_2)^\top, ..., (x_N)^\top\right)^\top \in \mathbb{R}^{Nd}$ is the optimization variable. Here, $x_i \in \mathbb{R}^d$ plays the role of the i-th cluster center candidate, $i = 1, ..., N$ and $\gamma > 0$ is a regularization parameter. The first sum corresponds to fidelity measure, while the second sum represents the regularization term. It enforces zeros for $||x_i - x_j||$ across a subset of pairs i, j, and can be seen as a generalisation of the fused Lasso penalty [142]. This means that, at the solution $x^* = ((x_1^*)\top, ..., (x_N^*)^\top)^\top$ of (3.1), there will be only a subset of $K, K < N$, mutually distinct vectors $x_1^*, ..., x_N^*$; these K distinct vectors, say $x_{i_1}^*, ..., x_{i_K}^*$, where $\{i_i, ..., i_K\} \subset \{1, ..., N\}$ are the cluster centers obtained through convex clustering. The cost function in (3.1) is strongly convex, and (3.1) has the unique solution x^*. Practically, this means that the algorithm is able to find the cluster centers, as some of the candidate centers "overlap" as the algorithm progresses.

ADMM. As already stated, the proposed parallel clustering method is based on ADMM. ADMM [3] is an iterative algorithm that solves the following type of problems:

$$\text{minimize} f(x) + g(z) \;\; \text{s.t.} \;\; Ax + Bz = c, \tag{3.2}$$

where $f : \mathbb{R}^n \to \mathbb{R}$ and $g : \mathbb{R}^m \to \mathbb{R}$ are convex functions, i.e., ADMM assumes an objective function, that is separable to two components, $x \in \mathbb{R}^n$ and $z \in \mathbb{R}^m$. $A \in R^{p \times n}$, $B \in R^{p \times m}$ are real-valued matrices, where $c \in R^p$. The augmented Lagrangian, $L_\rho : (\mathbb{R}^{m+n}) \times (\mathbb{R}^p) \to \mathbb{R}$ associated with (3.2) is:

$$L_\rho(x, y; \lambda) = f(x) + g(y) + \lambda^\top (Ax + By - c) + \frac{\rho}{2}||Ax + By - c||^2, \tag{3.3}$$

where $\rho > 0$ is a penalty parameter, and λ is the dual variable. Then, the ADMM algorithm consists of the following updates during the iterations:

$$x^{k+1} = argmin_x L_\rho(x, y^k, \lambda^k), \tag{3.4}$$

$$y^{k+1} = argmin_y L_\rho(x^{k+1}, y, \lambda^k), \tag{3.5}$$

$$\lambda^{k+1} = \lambda^k + \rho(Ax^{k+1} + By^{k+1} - c). \tag{3.6}$$

Here, $k = 0, 1, ...$, is the iteration counter, $x^k \in \mathbb{R}^n$ and $y^k \in \mathbb{R}^m$ are the primal variables, and $\lambda^k \in \mathbb{R}^p$ is the dual variable. It is well known that (x^k, y^k) converges to a solution of (3.2) under mild conditions; see [3].

Regarding the stopping criterion, a common way to terminate the algorithm is to introduce threshold values ϵ^{pri} and ϵ^{dual}, as feasibility tolerances, for the primal and dual feasibility conditions:

$$||r^k|| = ||Ax^k + By^k - c|| \leq \epsilon^{pri}, \tag{3.7}$$

$$||s^k|| = ||\rho A^\top B(y^k - y^{k-1})|| \leq \epsilon^{dual}, \tag{3.8}$$

so that the algorithm terminates if both conditions (3.7) and (3.8) are satisfied.

3.1.1 Problem model and the proposed parallel method

Based on the given definitions of SON clustering and ADMM method, the parallel ADMM-based convex clustering approach can be defined as follows. Assume we have N observations $\{a_j\}_{j=1}^N \in R^d$ as already stated before. As the intention is to develop a parallel algorithm, the work needs to be divided and delegated to a set of K working nodes. We assume a manager - workers (nodes) computational and sharing model with $K-1$ worker nodes that store data, perform calculations, and communicate with the master. Without loss of generality, we index the master as the first node, and the workers as nodes $2, ..., K$. Each node (including master) has a chunk of the input data a of size $N/K \times d$ [1]. To facilitate presentation, we introduce a two index notation, where $a_{ij} \in \mathbb{R}^d$ represents the j-th data point available at node $i, i = 1, ..., K, j = 1, ..., \frac{N}{K}$. We introduce a modification of standard convex clustering in (3.1), as follows. Note that the second term in (3.1) involves the differences across all pairs $(i, j), i < j$, of "candidate" cluster centers. Here, we also start by letting $x_{ij} \in \mathbb{R}^d, i = 1, ..., N, j = 1, ..., \frac{N}{K}$, be a "candidate" cluster center that corresponds to the (i, j)-th data point. However, unlike (3.1), we do not penalize the differences across all pairs of candidate clusters. Instead, we assign to the master

[1] For simplicity, assume that N is divisible by K; otherwise, node 1 can take $\lfloor \frac{N}{K} \rfloor + r$ data points, and the remaining nodes take $\lfloor \frac{N}{K} \rfloor$ data points, where r is the remainder when dividing N by K.

a "center" candidate cluster x_{11}. Similarly, we assign to each worker $i, i = 2, ..., K$, a local "center" candidate cluster x_{i1}. Then, we replace the second sum in (3.1) with the following sum:

$$\sum_{i=1}^{K} \sum_{j=2}^{N/K} ||x_{i1} - x_{ij}|| + \sum_{i=2}^{K} ||x_{11} - x_{i1}||. \tag{3.9}$$

In other words, within the data points at each node $i, i = 1, ..., N$, we penalize the difference between the local center x_{i1} and the remaining data points $x_{ij}, j = 2, ..., \frac{N}{K}$, at that node. This corresponds to the first sum in (3.9). In addition, regarding cross-node penalization, we penalize the differences between the master center x_{11} and the local centers $x_{i1}, i = 2, ..., K$. This corresponds to the second sum in (3.9). Finally, accounting for the sum of squared distances between each point a_{ij} and each candidate cluster center x_{ij}, we arrive at the following formulation for convex clustering:

$$\underset{x_{ij}}{minimize} \sum_{i=1}^{K} \sum_{j=1}^{\frac{N}{K}} ||a_{ij} - x_{ij}||^2 + \gamma \sum_{i=1}^{K} \sum_{j=2}^{\frac{N}{K}} ||x_{i1} - x_{ij}|| + \gamma \sum_{i=2}^{K} ||x_{11} - x_{i1}||, \tag{3.10}$$

where the minimization is with respect to the variables $x_{ij} \in \mathbb{R}^d, i = 1, ..., K, j = 1, ..., \frac{N}{K}$, and $\gamma > 0$. Note that problem (3.10) is not equivalent to the analog of (3.1) below:

$$minimize \sum_{i=1}^{K} \sum_{j=1}^{N/K} ||a_{ij} - x_{ij}||^2 + \gamma \sum ||x_{ij} - x_{lm}||, \tag{3.11}$$

where the second sum includes the differences between each pair of variables x_{ij} and x_{lm}. Implementing (3.11) in a parallel environment would result with high costs of communication and ineffective parallelization. However, extensive numerical results show that solving (3.10) yields effective clustering methods.

It is also useful to associate to problem (3.10) a graph $G = (\mathcal{N}, E)$, where $\mathcal{N}$ is the set of N nodes, each corresponding to a single variable $x_{ij}, i = 1, ..., K, j = 1, ..., \frac{N}{K}$, and E is the set of edges $(i, j) \sim (l, m)$, such that there is an edge between nodes (i, j) and (l, m) if the second sum in (3.10) involves the term $||x_{ij} - x_{lm}||$. Figure 3.1 illustrates graph G on an example with $K = 4$ nodes and $\frac{N}{K} = 4$ data points per node. In Fig. 3.1, x_{11} corresponds to the master center; $x_{i1}, i > 1$ correspond to node (worker) i center; and $x_{ij}, i > 1, j > 1$ correspond to the remaining candidate clusters. Formulation (3.10) penalizes differences $||x_{ij} - x_{lm}||$ for those pairs of x_{ij} and x_{lm} for which an edge in G exists.

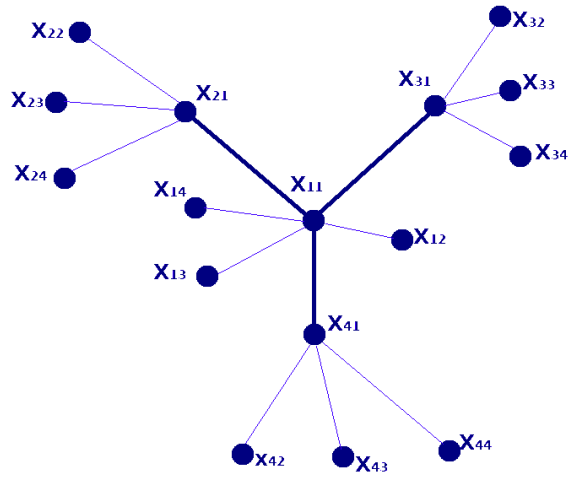


Figure 3.1: Illustration of graph G and structure of problem (3.10)

In view of the graph-based representation of (3.10), the original convex clustering in (3.1) is recovered when G is replaced with the full (complete) graph. Similarly, (3.10) may be seen as a weighted convex clustering [115], where unit weights are added for those pairs of x_{ij}'s and x_{lm}'s where $(i,j) \sim (l,m)$, and zero weights are added elsewhere.

Note that we do not assume beforehand any knowledge of the "structure" or "distribution" of data across different nodes. Also, the graph construction is independent of the actual values of the data points a_{ij}'s. In other words, the graph construction is arbitrary with respect to the available data. Extensive numerical results (See Section 3.3), show that this leads to accurate clustering solutions. We adapt this approach because the alternative, "data-driven" graph (weights) construction, as e.g. done in [109], incurs high computational cost and communication coordination among nodes. Data-driven centers and graph assignments are left for future work.

The problem can now be reformulated, in order to apply ADMM, as follows:

$$\begin{aligned} minimize & \sum_{i=1}^{K}\sum_{j=1}^{\frac{N}{K}} ||a_{ij} - x_{ij}||^2 + \gamma \sum_{i=1}^{K}\sum_{j=2}^{\frac{N}{K}} ||x_{i1} - x_{ij}|| + \gamma \sum_{i=2}^{K} ||x_{11} - y_{i1}|| \\ & \text{s.t. } y_{i1} = x_{i1}, i = 2..K. \end{aligned} \tag{3.12}$$

In other words, we introduce, for each node's center $x_{i1}, i = 1, ..., K$, an auxiliary variable y_{i1} and add the constraint $y_{i1} = x_{i1}$ to keep problem (3.12) equivalent to (3.10). Clearly, variables in (3.12) are then $\{x_{ij}\}, i = 1, ..., K, j = 1, ..., \frac{N}{K}$, and $y_{i1}, i = 1, ..., K$. We

now dualize the constraints in (3.12) and form the Augmented Lagrangian function $L_\rho : \mathbb{R}^{Nd} \times \mathbb{R}^{Kd} \times \mathbb{R}^{Kd} \to \mathbb{R}$ as follows:

$$\begin{aligned} L_\rho(x, y; \lambda) = &\sum_{i=1}^{K} \sum_{j=1}^{\frac{N}{K}} ||a_{ij} - x_{ij}||^2 + \gamma \sum_{i=1}^{K} \sum_{j=2}^{\frac{N}{K}} ||x_{i1} - x_{ij}|| + \gamma \sum_{i=2}^{K} ||x_{11} - y_{i1}|| \\ &+ \sum_{i=2}^{K} \lambda_i^T (y_{i1} - x_{i1}) + \frac{\rho}{2} \sum_{i=2}^{K} ||y_{i1} - x_{i1}||^2, \end{aligned} \tag{3.13}$$

where $\rho > 0$ is a penalty parameter. In (3.13), we denote by $x \in \mathbb{R}^{Nd}$ the vector that stacks all the x_{ij}'s one in top of another, and by $y \in \mathbb{R}^{Kd}$ the vector that collects all y_{i1}'s one on top of another. We now apply ADMM in (3.4)-(3.6) with respect to the Lagrangian L_ρ in (3.13) to solve (3.12). After decomposing x and y back to blocks x_{ij}'s and y_{i1}'s, it can be verified that (3.4)-(3.6) translates into the set of updates in (3.14) - (3.17), as follows:

- x update on each worker node $i = 2..K$ in parallel:

$$\begin{aligned} x_{ij}^{k+1} = argmin &\sum_{j=1}^{\frac{N}{K}} ||a_{ij} - x_{ij}||^2 + \gamma \sum_{j=2}^{\frac{N}{K}} ||x_{i1} - x_{ij}|| \\ &+ (\lambda_i^k)^T (y_{i1}^k - x_{i1}) + \frac{1}{2} \rho ||y_{i1}^k - x_{i1}||^2 \end{aligned} \tag{3.14}$$

  In (3.14), the optimization is (jointly) with respect to $x_{ij}, j = 1, ..., \frac{N}{K}$.

- x update on the master node:

$$x_{1j}^{k+1} = argmin \sum_{j=1}^{\frac{N}{K}} ||a_{1j} - x_{1j}^k||^2 + \gamma \sum_{j=1}^{\frac{N}{K}} ||x_{1j} - x_{11}|| + \gamma \sum_{i=2}^{k} ||x_{11} - y_{i1}^k|| \tag{3.15}$$

  Note that the optimization in (3.15) is (jointly) with regard to $x_{1j}, j = 1, ..., \frac{N}{K}$.

- y update on master node:

$$y_{i1}^{k+1} = argmin \sum_{i=2}^{K} (\lambda_i^k)^T (y_{i1} - x_{i1}^{k+1}) + \frac{1}{2} \rho \sum_{i=2}^{k} ||y_{i1} - x_{i1}^{k+1}||^2 + \gamma \sum_{i=2}^{K} ||x_{11}^{k+1} - y_{i1}||. \tag{3.16}$$

  Note that minimization (3.16) is with respect to (jointly) variables $y_{i1}, i = 2, ..., K$.

- λ update on master node:

$$\lambda_i^{k+1} = \lambda_i^k + \rho(y_{i1}^{k+1} - x_{i1}^{k+1}) \tag{3.17}$$

  Note that all the λ_i's, $i = 2, ..., K$, are updated at the master independently, in parallel.

Regarding inter-node communications (variable exchange), the procedure is as follows. Assume for simplicity that all the λ's, x's and y's are initialized to zero. Then, after the master and the worker nodes update x according to (3.14) and (3.15), each worker i sends its new center variable x_{i1}^{k+1} to the master. After the master performs y and λ updates as in (3.16) - (3.17), it sends variables y_i^{k+1} and λ_i^{k+1} to worker $i, i = 2, ..., K$. The "sending" of results does not have to be explicit sending, it can also be a synchronization, depending on the parallelization framework (see more details in Section 3.2).

Regarding (3.14)-(3.17), at each iteration, each node and the master solve problems (3.14) and (3.15). These problems are of SON type, but with a sparse, star graph of SON penalties, and of variable size that is K times smaller than (3.10) and (3.11), hence enabling scalability. Hence, for sufficiently large K, an efficient solver for moderate-sized SON problems can be adapted to solve (3.14) and (3.15), e.g., [112]. Actually, as detailed later, we used CVXPY [143, 144] as a general convex solver to solve (3.14)-(3.15). Update (3.17) is clearly a cheap update. Finally, (3.16) is done closed form by evaluating a proximal operator block-thresholding for the 2-norm [109].

Note that formulation (3.10), unlike (3.11), does not guarantee perfect cluster recovery for any $\gamma > 0$. However, we observe numerically that, for the solution $\{x_{ij}^*\}$ of (3.10), an approximate clustering structure emerges. That is, the $\{x_{ij}^*\}$'s cluster into a number, say K', different groups, such that the x_{ij}^*'s within the same group are mutually very close. This motivates the following merging procedure.

Algorithm 2 shows the merging procedure, that is being applied after (3.14) - (3.17) converges. The first stage of merging is applied locally on each node. The threshold values ϵ_i and ϵ are positive numbers, used to filter the possible centers x_{ij}^*. We assign the first candidate point as a first center, and check the rest of the points. All those points that are close (within ϵ_i distance) to the points already marked as centers are being ignored. In the opposite case, a point is denoted as a new center. The second step is to merge the obtained local centers on the master node. This means that all the obtained local centers need to be synchronized on the master node first. The value ϵ is calculated by using the average distance between the obtained local centers. Then, the

Algorithm 2 Merging procedure of possible centers

On each node i locally, in parallel:
Require: $\epsilon_i, i = 1, ..., K; \epsilon$; initialize the list of local cluster centers $\mathcal{C}_i = \{\}$
for all possible center candidates x_{ij}^* **do**
for already accepted centers $c_{il} \in \mathcal{C}_i$ **do**
if $\|x_{ij}^* - c_{il}\| \leq \epsilon_i$ **then**
omit x_{ij}^* from centers $\mathcal{C}_i$
else
include x_{ij}^* to centers $\mathcal{C}_i$
end if
end for
end for
return the found local centers $\mathcal{C}_i = \{c_{i1}, ..., c_{iP_i}\}$, where P_i is the number of local centers found at node i
On master node:
Require: ϵ;, All possible center candidates from the nodes: $\mathcal{C}_i, i = 1, ..., K$
Require: initialize the list of final centers $\mathcal{C} = Null$
for all possible center candidates from all nodes $c_{ij} \in \mathcal{C}_i, i = 1, ..., K$ **do**
for already accepted centers $c_l \in \mathcal{C}$ **do**
if $\|c_{ij} - c_l\| \leq \epsilon$ **then**
omit c_{ij} from centers $\mathcal{C}$
else
include c_{ij} to centers $\mathcal{C}$
end if
end for
end for
return the found centers $\mathcal{C} = c_1^\star, ..., c_{P'}^\star$, where P' is the final number of centers

same merging procedure is applied as before. The result is a set of centroids on the output of the algorithm. It is important to perform this twofold merging procedure, as the result of first phase of merging is a local set of centers that a worker found, that may and should partially overlap with the sets of centers found by other workers.

This means that we keep each next candidate for centroid only if its distance to previously kept centroids is larger than ϵ. Then, this whole set of already reduced subsets of possible centers from all nodes is being analyzed by the master node in the same manner. The output of this process is the set of resulting centers.

The pseudocode for the described clustering algorithm is shown in Algorithm 3, where the overall proposed clustering method is summarized. After (3.14)-(3.17) converges and the merging procedure is applied as described in Algorithm 2, the master node makes the final centers available to the worker nodes. Then, each worker assigns its local data points to the cluster that corresponds to the nearest center, by assigning the corresponding labels.

Algorithm 3 Pseudocode for the proposed algorithm

Require: global tuning parameters γ and $\epsilon^\star$, and at each node i: $a_{ij}, j = 1, ..., \frac{N}{K}$
  repeat
    Compute x_{1j}^k on master as in (3.15)
    Compute x_{ij}^k for each worker $i = 2..K$ in parallel as in (3.14)
    Compute y_{i1}^k on master as in (3.16)
    Compute λ_i^k on master as in (3.17)
  until a stopping criterion is met
  Merge the possible centers as described in Algorithm 2
Require: on each node i the final list of global centers $\mathcal{C} = c_1^\star, ..., c_{P'}{}^\star$
  for all local data points a_{ij} on each worker i, in parallel **do**
    assign point a_{ij} to cluster $c_l^\star$ where $min_{t=1,...,P'} \|a_{ij} - c_t^\star\| = \|a_{ij} - c_l^\star\|$
  end for

The algorithm works with 2 tunable parameters, γ and ϵ. Choosing a large value for the regularization parameter γ naturally enforces more overlapping centroids. On the other hand, choosing a small value for the parameter can result in only slightly moving the centers, producing large distances between the gathered points, and hence a too large number of clusters. A discussion on choosing the value for parameters γ is given later, in the section dedicated to experimentation.

The adjustable parameter ϵ_i, needed during the merging procedure, can be chosen on different ways. The proposed approach here to choose $\epsilon_i, i = 1, ..., K$ is the following:

$$\epsilon_i = \frac{2K^2}{\epsilon^\star \times N(N-K)} \sum_{j=1}^{\frac{N}{K}} \sum_{l=j}^{\frac{N}{K}} \|a_{ij} - a_{il}\|, \tag{3.18}$$

where $\epsilon^\star$ is a tunable input parameter, that can also be set to a universal, data and problem independent, constant value, e.g., equal to 5 or 10. The formulation in (3.18) means that each node calculates the average euclidean distance between the points in its own data chunk and divides the result by a constant value $\epsilon^\star$. Similarly, the parameter ϵ can be calculated on master, based on the average euclidean distance between the locally obtained centers as follows:

$$\epsilon = \frac{2}{\epsilon^\star \times P(P-1)} \sum_{j=1}^{P} \sum_{l=j}^{P} \|c_j - c_l\|, \tag{3.19}$$

where $P = P_1 + ... + P_K$ is the overall number of locally obtained centers on all nodes and the sum in (3.19) involves all elements from the union of sets $\mathcal{C}_i, i = 1, ..., K$. Calculating ϵ_i as in (3.18) involves only pairwise distances within single workers data, which is only $O(\frac{N^2}{K^2})$ pairwise distances. That is, across all nodes, we calculate $O(K \times \frac{N^2}{K^2})$ pairwise

distances. Compared with $O(N^2)$ pairwise distances, needed with sequential weighted SON clustering approaches, like for example AMA [36], it is significantly cheaper for sufficiently large number of workers K. Alternatively, the sums in (3.18) may be replaced with minimum, i.e. to consider minimal within-workers data distances. This means that, when performing the merging of the local centers, the value of ϵ on the master can be used by calculating it in the same manner as in (3.18), with the only difference that the distances of local centers are considered instead of the distances of data points.

Algorithm 2 may be seen as a simple instance of a pairwise clustering method; see, e.g., [145]. Intuitively, solving (3.10) usually brings the $x_{ij}^{\star}$'s that correspond to the data points a_{ij}'s within a single "true" cluster very close to each other, but it may not make them exactly equal up to the full accuracy. Therefore, a simple pairwise clustering method in Algorithm 2 is introduced to fine-tune the results achieved by solving (3.10). Note that Algorithm 2 involves $O(K \times \frac{N^2}{K^2})$ pairwise comparisons across all nodes, and hence again it scales well when K is large. We also report that Algorithm 2 allows for a cheap polishing of the results, typically incurring 11.6% of the overall execution time of Algorithm 3, on average.

3.2 Implementation

The implementation of Algorithm 3 is developed in Python, using the COMPSs [5] framework for parallel execution. COMPSs offers a simple programming model with the aim to facilitate the parallelization process. It has been widely adopted and extended in numerous scientific projects offered as a tool to develop scientific applications and optimize their execution on distributed infrastructures. The testing has been performed using the AXIOM computing facility, as well as for the primal methods described in the previous chapter.

The Python implementation relies on the CVXPY [143, 144] package, used for the minimizations described in (3.14)-(3.15). The PyCOMPSs framework [116] (COMPSs for Python) enables a convenient way for parallelization, by simply annotating a function as a task. However, this requires a proper data format and distribution, as well as a synchronization point, where the results of execution on different processes are being collected into a predefined data structure. As the parallelization for the primal methods was developed in MPI, it is of interest to exploit another, alternative approach. The parallelization offered by PyCOMPSs is of higher level and more user-oriented than MPI. This transparency reduces the level of control in the hands of programmers and naturally may be less efficient than MPI. However, it is valuable to collect impressions about these

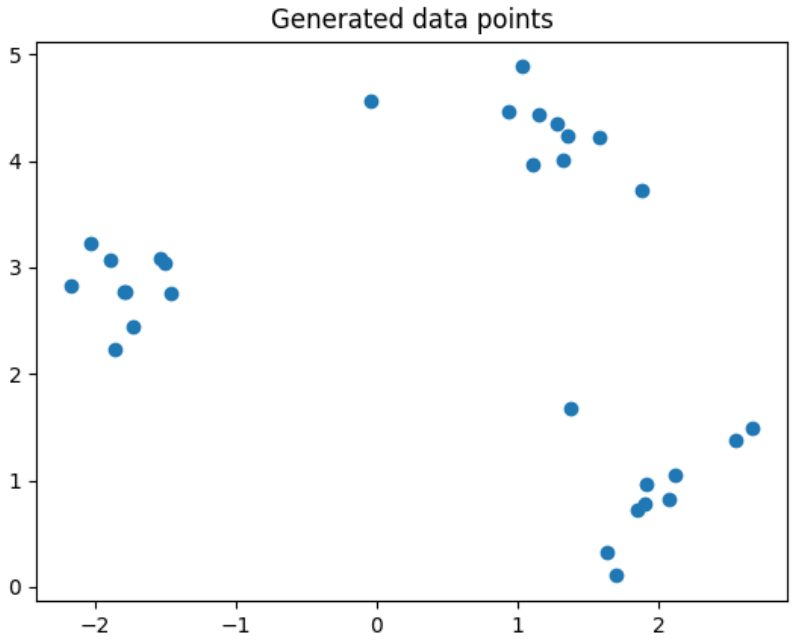


Figure 3.2: Example of a 2-dimensional data set of a small volume.

frameworks and compare them from different aspects. This section provides a detailed explanation of implementing the described algorithm in Python, using PyCOMPSs as a tool for parallelization. The implementation of the algorithm is open-source and can be found on GitHub [2].

3.2.1 The input data

The input data is read from a file and resized from its 2-dimensional form to a 3-dimensional form $workers \times chunk\_size \times d$, where $workers$ is the number of nodes operating (defined as an input parameter) and $chunk\_size = \frac{N}{K}$. The data points are being distributed in consecutive chunks as read from files. This means that the data distribution across workers is not "informed" and is arbitrary, i.e., each worker usually contains a mixture of data points that should belong to different clusters. The tests are based on both synthetical and real data sets. The synthetical data sets were generated in order to test the scalability and the accuracy of the algorithm. The data sets were generated by using a samples generator from the *scikit-learn* package [146]. Fig. 3.2 represents an example of a generated 2-dimensional data set of small volume. It contains 30 points, with clearly distinguishable clustering into 3 clusters.

Large synthetic data sets are generated as Gaussian mixture models [147]. We con-

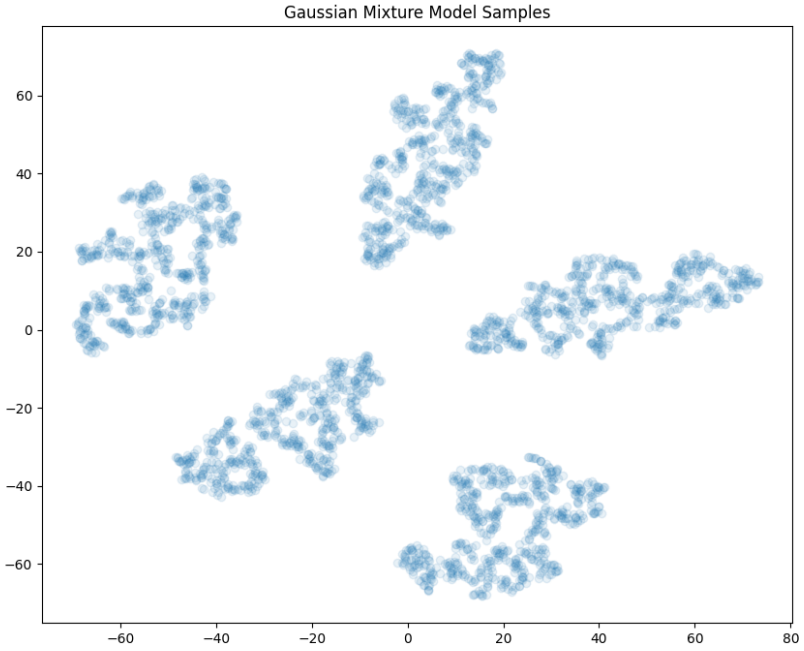


Figure 3.3: t-SNE for an example of the generated 3-dimensional data set of larger volume.

sider mixtures of k multivariate Gaussian distributions with mean μ_i and covariance $\mathcal{E}_i, i = 1...k$. The values for μ are d-dimensional points generated randomly, but from different intervals for each Gaussian k, in order to ensure that they will be distant enough to represent separate clusters. Also, the values for σ are generated randomly, from one interval for the diagonal part and from another one for the upper triangular part. Regarding the value of π, the same value of $\pi = \frac{1}{k}$ was used for each Gaussian. Fig. 3.3 shows the t-SNE embedding [148] for an example of generated data. The data set size is 3000 points here, with dimension 3 and 5 clusters.

3.2.2 The stopping criterion

The stopping criterion implemented here is the usual stopping approach for ADMM. It requires implementing (3.7)-(3.8).

The residuals are being calculated at the end of each iteration, when the master process has access to all results, obtained by the workers. The threshold values ϵ^{pri} and ϵ^{dual} are also being recalculated at each iteration as follows:

$$\epsilon^{pri} = \alpha\sqrt{N} + \beta max\{||x^k||, || - y^k||\}, \tag{3.20}$$

$$\epsilon^{dual} = \alpha\sqrt{N} + \beta||\lambda^k||. \tag{3.21}$$

Here, $\alpha \in R$ and $\beta \in R$ represent the absolute and relative tolerance values, respectively. These values can be set as input parameters and their default values are $\alpha = 10^{-4}$ and $\beta = 10^{-2}$. This means that both the residuals and the threshold values are being updated at the end of each iteration k, based on the values x^k, y^k and λ^k.

3.2.3 The parallel implementation of the ADMM-based convex clustering algorithm

Developing a parallel algorithm in PyCOMPSs is nearly as simple as developing a serial implementation. A function or method needs to be annotated with a decorator `@task`. This means that the function/method that follows represents a place for parallel execution. The `@task` decorator can have parameters, that for example describe the input/output types for tasks. Before invoking a task, there are a few important aspects to note. First, we need to prepare the input data for a task properly. This means that we should have a data structure, for example a list of arrays, that can be divided between the workers, so that each worker does the computation on a separate data chunk. Practically, this means reshaping the data structure so that it contains K sub structures for the workers. Particularly, the input data a, the primal variables x and y, as well as the dual variable λ are organized as lists of arrays. Every list contains K arrays, one for each worker, where the shapes of these arrays correspond to definitions given before. Listing 3.1 shows the setup of these structures.

Listing 3.1: Preparing the data structures for parallel execution, Python

```
N=A.shape[0]
d=A.shape[1]
chunk_size=N//workers
a=[A[i*chunk_size:(i+1)*chunk_size] for i in range(workers)]
x=[np.zeros((chunk_size, d)) for i in range(workers)]
y=[np.zeros((d)) for i in range(workers-1)]
y=np.asarray(y)
lambdaVal=[np.zeros((d)) for i in range(workers-1)]
```

We use `functools` from Python standard library, that offers higher-order functions and operations on callable objects. Specifically, we use the function `partial`, that is able to return a partial object. When that object is called, it behaves like the function from its arguments. When combined with `map`, the following call can be made:

```
list(map(functools.partial(update_x, rho, gamma, d),
                a[1:workers], x[1:workers], y, lmbd))
```

This enables calling the function `update_x` on each worker, with scalar valued parameters `rho`, `gamma` and `d`, that are the same values for all workers. The rest of the parameters, `a[1:workers]`, `x[1:workers]`, `y` and `lmbd`, are being sent "partially", by chunks, so that each worker gets the following chunk. This makes the synchronization process eased afterwards.

The code snippet in Listing 3.2 shows the definition of the function `update_x`, that is being executed as a task. The `@task` decorator specifies the return type as array. The function creates the CVXPY variable for the solution and sets up the minimization problem. The function `objective_x` directly implements the previously defined objective function. Finally, the CVXPY solver solves the problem and returns the solution on each worker separately.

When the workers complete the task, they can return the results to a joint list. However, it is still needed to do an explicit synchronization after task execution, in order to collect the results correctly. This ensures that all workers finished the execution and put the data to the resulting structure. Different function exist for synchronization in the framework. We used the `@compss_wait_on` in our implementations for this purpose.

Listing 3.2: The definition of a task function, in PyCOMPSs

```
@task(returns=np.array)
def update_x(rho, lmbd, d, a, x, y, lambdaVal):
  sol=cp.Variable(a.shape, value=x)
  problem=cp.Problem(cp.Minimize( \
                    objective_x(a,sol,lambdaVal,y,rho,lmbd,d)))
  problem.solve()
  return sol.value
```

The main loop of the algorithm is displayed in Listing 3.3. During the main loop of the algorithm, the update of x is performed in parallel as a task, and the rest of updates is carried out afterwards, on a single (master) node.

When the convergence criteria are met, the algorithm breaks the loop. It is convenient that the synchronization is already done at this point, so there is no need for spreading a break signal among processes, as we are working with only one process when checking the termination condition.

Listing 3.3: The main loop of Algorithm 3

```
for i in range(num_iter):
  x[0]=update_x_zero(a[0],x[0],y,lmbd,workers)
  x[1:workers]=list(map(functools.partial(update_x,rho,lmbd,d), \
                        a[1:workers],x[1:workers],y,lambdaVal))
  x[1:workers] = compss_wait_on(x[1:workers])
  y_old=y
  y=y_update(lambdaVal,x,y,rho,workers-1,d,lmbd)
  lambdaVal=lambda_update(lambdaVal,rho,x,y,workers-1)
  #The stopping criterion
  primal_res=np.sqrt(np.sum([(np.linalg.norm(x[i][0]-y[i-1])**2) \
                     for i in range(1, workers)]))
  dual_res=np.linalg.norm(- rho*(y-y_old))
  eps_pri=np.sqrt(N)*abstol+reltol* \
                     max(np.linalg.norm(x),np.linalg.norm(-y))
  eps_dual=np.sqrt(N)*abstol+reltol*np.linalg.norm(lambdaVal)
  if primal_res<=eps_pri and dual_res<=eps_dual:
    req_iter=i+1
    break
```

The merging procedure is performed on each worker separately, as another `task`, as shown in Listing 3.4. This corresponds to the merging procedure, described in Algorithm 2. The final merging of locally obtained centers is done serially, on master.

Listing 3.4: The local merging on workers

```
@task(returns=np.array)
def merge_centers_locally(epsilon, x, chunk_size, a, data_size):
  sumVal=0.0
  cnt=0
  for i in range(data_size):
    for j in range(i+1, data_size):
      sumVal+=np.linalg.norm(a[i]-a[j])
      cnt=cnt+1
  avg_dist = sumVal/cnt
  eps = avg_dist/epsilon
  centers=[]
  removed_indices=[]
  for i in range(0, chunk_size):
    if i not in removed_indices:
      centers.append(x[i])
      for j in range(i+1, chunk_size):
        if np.linalg.norm(x[i] - x[j]) <= eps:
          removed_indices.append(j)
  return centers
```

The function for the final merging is similar as the local function from Listing 3.4, see the GitHub repository for more details [2]. COMPSs enables running this kind of code on different distributed infrastructures. The fact that the algorithm is operating on a computing cluster without the need for explicit data exchange inside the code is directly showing the advantageous ease of development. All the data exchange within cluster nodes is completely transparent, which makes this high-level approach for parallel application development very appealing.

3.3 Experimentation

In this section, the aim is to assess the quality of different aspects of the distributed ADMM-based convex clustering algorithm. Particularly, the tests performed on small, 2-dimensional data sets are appealing for plotting and evaluating the properties of the algorithm. One of the main ideas is to monitor the accuracy of the solution, which is straightforward with data sets generated under controlled conditions. For these synthetic data sets, the number of expected clusters is known in advance, which makes the first stage of the evaluation simpler. However, we will also include other accuracy metrics, as silhouette score and comparison with the results of plain k-means clustering. The second key aspect is the evaluation of performance. The scaling properties of the algorithm will be demonstrated through a set of tests on different data sets. A real, industrial, highly relevant data set is also used for evaluation. The real data used for the tests are gained through collaborations on a H2020 project, I-BiDaaS [47].

A few different sets of experiments can be identified during the evaluation process. First, we discuss the time consumption of different parts of the algorithm. This is useful in order to identify possible bottlenecks. Further, the focus is on accuracy evaluation, where these analyzes include measuring the percentage of accurately clustered points, where a ground truth is known. We use a small, synthetic data set and the Iris data set [149, 150] here. Additionally, we observe the accuracy of some large, synthetic data set, by means of silhouette score values and comparison to k-means. We also evaluate the scalability of the proposed method, on some large, synthetic data sets, generated as Gaussian mixtures. The evaluation also includes a discussion on some aspects of choosing the value for the regularization parameter γ. A comparison of the proposed method to other clustering approaches is also given in this section. We compare our approach to the following alternatives: the SON clustering [37, 30, 38], the AMA method [36], DBSCAN [117], SSNAL [115] and parallel k-means provided by Apache Spark [21]. We perform the comparison on the synthetic data sets, generated as Gaussian mixtures. Finally, we also

include the results of tests on a mentioned real, industrial data set.

3.3.1 Time consumption of different segments of the algorithm

When considering the time consumption of the different actions during the algorithm execution, it naturally emerges that the iterative part of algorithm consumes 88.3% of the overall execution time, on average. The average time spent on reading the input data is only 0.05%, while the average time needed for the process of merging the possible centers is 11.6%. These average values are calculated over all the experiments conducted and mentioned in this chapter.

3.3.2 Accuracy evaluation

In order to gain some insights into the level of accuracy of the developed clustering implementation, a few approaches are used here. First, we investigate the solution for a small, generated 2-dimensional data sets, where it is straightforward to plot the results and gain visual insights. Fig. 3.4 shows an example with 30 2-dimensional data points. The points belong to 3, clearly separable clusters, as shown by the blue dots on Fig. 3.4. The results of clustering are also shown on Fig. 3.4. Clearly, the algorithm is able to identify the centers correctly. Fig. 3.4 displays the found centroids, as shown by the orange dots. These centers are the output from the merging mechanism. When the stopping criterion is met, the center candidate points, that are "close", by the definition of ϵ_i (3.18) are being aggregated. It can be seen that the algorithm properly detected the existence of 3 clusters and assigned the points to the clusters accurately.

It is also interesting to examine the points that are candidates for centroids, i.e. the values of the primal variables x and y, after the algorithm converges, before applying the merging of centroids. Fig. 3.5 shows all the candidate points for centroids. It can be observed that the points are moving towards each other on the level of one cluster, but also globally. This example meets the termination condition in only 7 iterations, and after applying the merging, it results with 3 centers.

Beside these visual inspections of the results, some reliable accuracy proofs are also needed, in order to ensure the algorithm is working as intended, especially for larger data sets. The best approach is to use the ground truth, when available. Comparing the results with the clearly defined expected cluster labels provides the most reliable clustering accuracy, that can be expressed through the percentage of accurately clustered points. When obtaining the percentage of accuracy, it should be taken into account that we are interested in

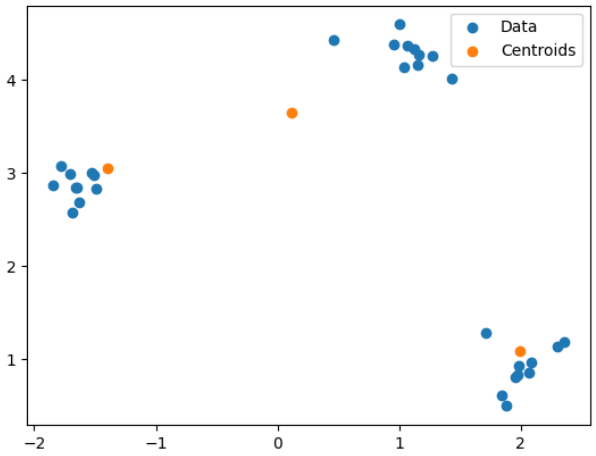


Figure 3.4: Results of clustering for a generated data set 30x2, $\gamma = 0.3$,$\epsilon^{\star} = 2$

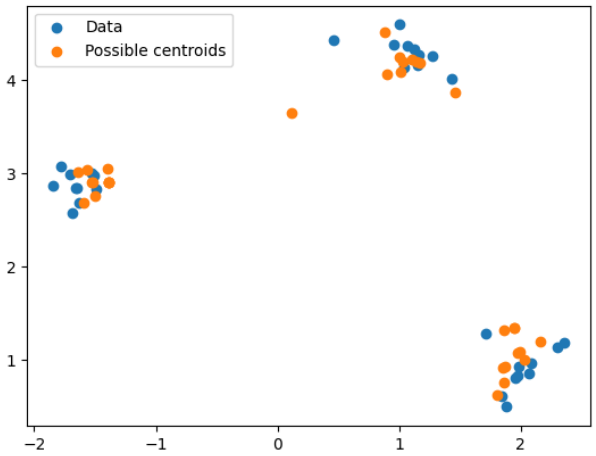


Figure 3.5: All centroid candidates for a generated 30x2 data set

assigning the points to the same cluster, when they have the same real cluster label. Naturally, the found label can differ regarding the real label. For example, the algorithm can converge and assign the points to clusters accurately, but denoting the points with label 'x', where the label for that cluster was 'y' in original. This can be easily solved by designing the accuracy checking algorithm so that it tries all the combinations for cluster labels and takes the one with the best accuracy. The accuracy evaluation, when the ground truth is known, can be illustrated on a common example of the Iris [149, 150] data set. This means dealing with a higher-dimensional data set, where the clusters are mainly distinguishable by the nature of the underlying data.

As already stated, when the ground truth is available, we evaluate the clustering accuracy

Table 3.1: Accuracy comparison for different clustering algorithms on the Iris data set

Algorithm	Parameters	Number of clusters	Accuracy (%)
ADMM-based convex clustering	$\gamma = 40$, $\epsilon^\star = 5$	3	93.33%
k-means	$k = 3$	3	88.66%
AMA	$\gamma \in [4.3, ..., 9.1]$	3	90.66%

as the percentage of correctly classified data points. The Iris data set [149, 150], is available in *scikit-learn*. It contains 3 different classes of the plant Iris. It has 4 attributes and 150 samples. Based on these attributes, a clustering algorithm could identify the existence of 3 clusters (see Table 3.1). We executed the ADMM-based convex clustering algorithm on this data set and obtained the 3 clusters. In order to further analyse the results, we compared the obtained labels to the real, known labels. It turns out that our algorithm assigned 93.33% of the data points to the correct cluster. It should be also mentioned that it assigns all data points belonging to the first cluster accurately, while it makes some 'mistakes' with the second and third cluster. The nature of the data directly affects this, as the mentioned two clusters are 'close' to each other. In fact, running the standard k-means algorithm on this data set for different values of k, results with a highest silhouette score value of 0.68 for $k = 2$. This can be also easily seen on the t-SNE embedding of the data set, in Fig 3.6. Table 3.1 also contains the percentages of accurately clustered points for the standard k-means algorithm (with the preset value of k) and for the AMA method [36] as well, for reference. When running plain k-means for k=3, the percentage of points accurately clustered is 89.33%. As these evaluation showed, our method can perform as accurately as (or even more accurately than) k-means with a correctly predefined value for k. In order to further investigate this test case, we also run the AMA method [36] on this data set. By setting the parameter γ appropriately, it is able to find the 3 clusters with 90.66% of points clustered accurately. This shows that the 3 different clustering algorithms perform with a similar degree of accuracy on this data set. This is illustrated on Fig. 3.7. It shows the accuracy percentage related to the value of the input parameter γ. The accuracy percentage of k-means is independent regarding the value of the input parameter γ. Different initializations for k-means, meaning the usage of alternate algorithms as 'elkan' and 'full' or number of runs with different centroid seeds set to {5, 10, 100}, always result with the same level of accuracy of 89.33% on this data set. The AMA method has its range $\gamma \in [4.3, ..., 9.1]$, where it produces the highest accuracy. All values of γ that are out of this range affect significant decrease in clustering accuracy. The similar holds for ADMM-based convex clustering, except that this range of values giving the highest accuracy is significantly broader. Intuitively, even outside

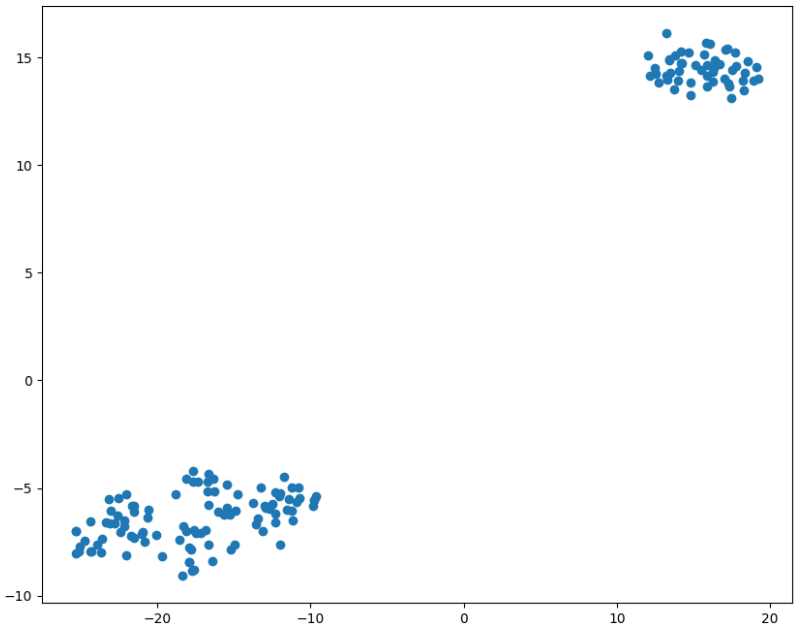


Figure 3.6: The t-Sne embedding of the Iris data set

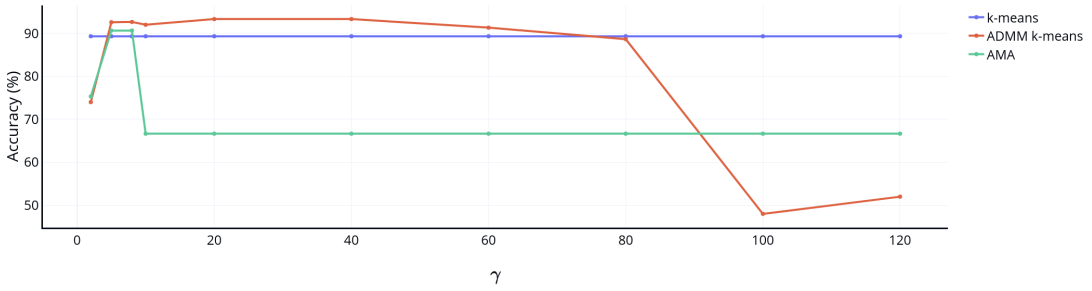


Figure 3.7: The accuracy values of different methods on Iris data set

of the range of γ for which a SON-like clustering is exact, in a vicinity of this range, "SON-like clustering still produces nearly-exact" clustering - that is then harnessed for correct clustering - via the merging procedure.

Let us consider an additional example, where a graph G of data points is as on Fig. 3.8. The different colors of points denote different clusters, where the points should be assigned. This data set contains only 10 2-dimensional points, in order to make it easier to plot and evaluate. 3 different clusters of points can be identified, based on the coordinates of the points. It is envisaged to work with 4 workers in this case. The first worker only has one point, while the rest of them have 3 data points each. Each of the 3 data points

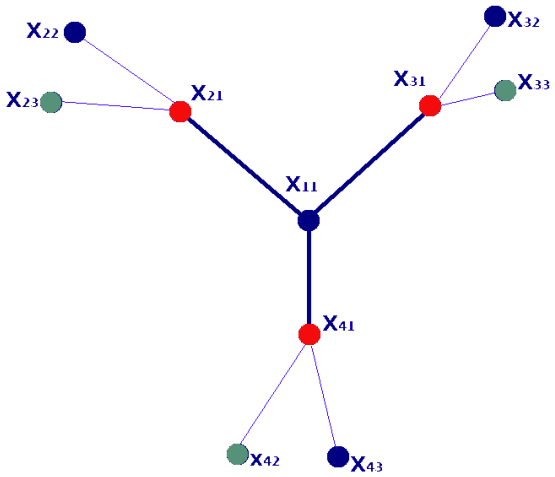
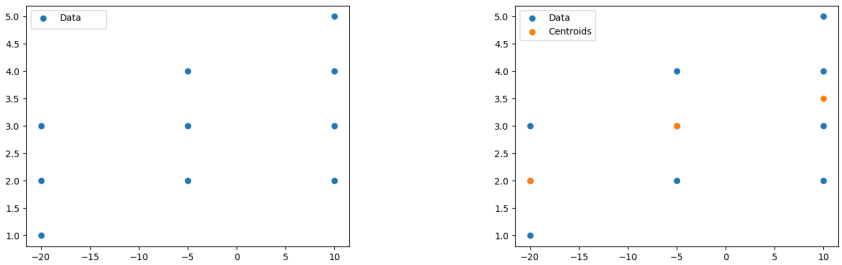


Figure 3.8: The graph for evaluation



(a) The data set for evaluation

(b) The resulting clustering for $\gamma = 1.5$ and $\epsilon^\star = 5$

Figure 3.9: The data set and clustering

on a worker should belong to a different cluster, which makes the problem challenging. The 3 clusters are clearly separable, as shown on Fig. 3.9 (a). The algorithm is able to cluster the points with 100% accuracy, for $\epsilon^\star = 5$ and $\gamma \leq 10$. Fig. 3.9(b) demonstrates the resulting cluster centers.

As the ground truth for clustering is not always available, some additional accuracy metrics can be used in order to assess the outcomes of clustering. Silhouette score is a common and widely-used way to evaluate a clustering approach, so it is included as an accuracy metric in our experiments. First, after obtaining the cluster centers, a label that corresponds to one of the resulting clusters is needed to be assigned to each data point. This can be simply achieved by computing the Euclidean distance of each data point to each center, and taking the least one to label the data point. This can be done in parallel on the workers, when each workers has access to the final centers. Then, the silhouette score

Table 3.2: Accuracy evaluation for higher dimensional data sets

Data size	γ	$\epsilon^\star$	clusters ADMM-based convex clustering	clusters k-means	ADMM-based convex clustering s.score	k-means s.score
1000×3	5.0	4	8	8	0.76	0.76
5000×3	6.6	2	4	4	0.77	0.78
5000×5	6.6	2	5	4	0.62	0.75
10000×3	6	5	10	10	0.69	0.75

value can be easily obtained.

The accuracy of the algorithm is also tested for large higher-dimensional data sets. These data set are generated as Gaussian mixture models, as described earlier (see Section 3.2.1). In order to visualize the results, t-distributed stochastic neighbor embedding (t-SNE) [148] will be used. Let us consider a few higher-dimensional data sets. Table 3.2 shows the results for these experiments. It can be seen that the algorithm is mostly able to identify the expected number of clusters, with high silhouette score values. Additionally, the *scikit-learn* k-means algorithm results mostly with the same number of clusters and similar values for silhouette score.

Fig. 3.10 shows the t-SNE embedding for a 1000×3 generated data set. The data points are colored according to their cluster labels, obtained by our ADMM-based convex clustering algorithm. It represents an example where our clustering algorithm clusters the data points accurately to the expected clusters, that also corresponds to a high silhouette score value.

Based on the described experiments, it can be concluded that ADMM-based convex clustering can perform with high accuracy. The accuracy of the algorithm is compatible with the accuracy level of k-means clustering and the AMA method.

Additional accuracy metrics

The accuracy level of the performed tests was represented by silhouette score, which is a commonly used approach for clustering. However, there exists a set of additional metrics for this purpose. One of them is the Dunn Index (DI) [151]. Dunns Index is equal to the minimum inter-cluster distance divided by the maximum cluster size. This practically means that large inter-cluster distances (better separation) and smaller cluster sizes (more compact clusters) lead to a higher DI value, which represents better clustering. The `validclust` package provides this metric. We can test this metric on an example.

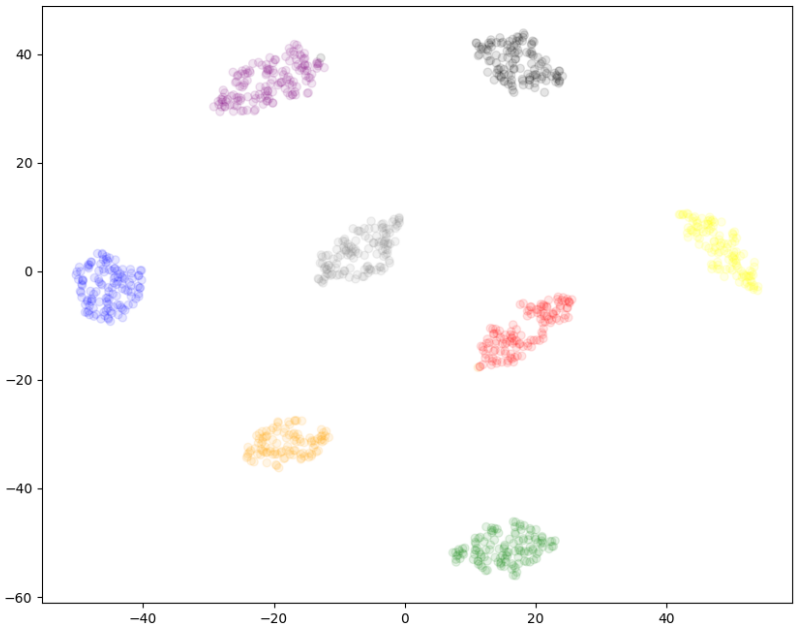


Figure 3.10: t-SNE embedding for clustering over a synthetical data set of size 1000x3

For instance, let us take our smallest data set with size 30×2 data points. The DI is 2.53, in the setup where the silhouette score is also high, 0.89.

Another useful metric is the Davies-Bouldin score [152]. It represents the average similarity measure of each cluster with its most similar cluster. Similarity is the ratio of within-cluster distances to between-cluster distances. This means that further clusters that are less dispersed will result in a better score, i.e. lower values represent better clustering here. The metric is available in the *scikit-learn* library. Let us demonstrate the metric with the same example as for DI. For the 30×2 data set, we get 0.13 for Davies-Bouldin score, when the silhouette score is 0.89. These approaches support the trends presented by silhouette score, and can be used as additional metrics here. However, the silhouette score is still a baseline for our tests, as its value is limited from -1 to 1, so it gives a clear representation of the quality of the results.

3.3.3 Scalability evaluation

In order to evaluate the scaling properties of the parallel ADMM-based convex clustering algorithm, different data sets will be used to run on a computer cluster with different numbers of working nodes. All the data sets in these experiments are generated as Gaus-

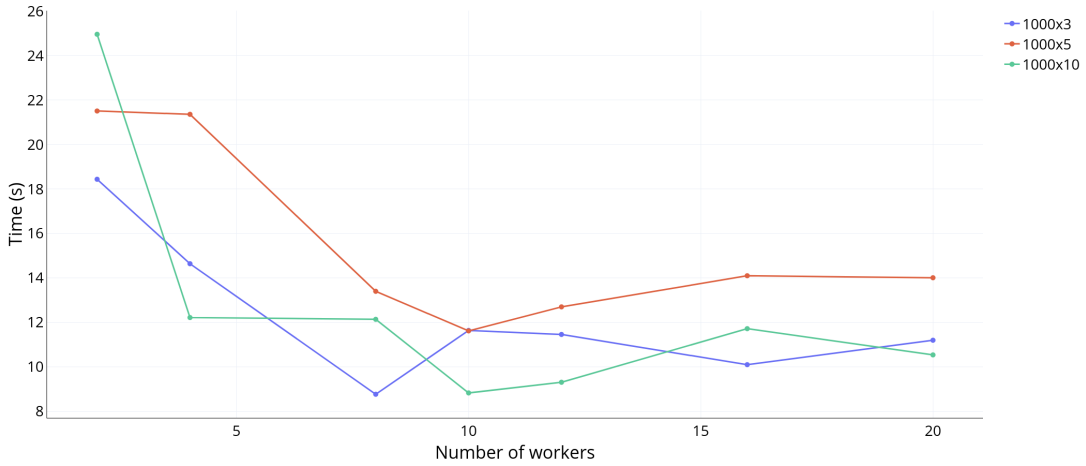


Figure 3.11: Scaling properties for the data sets with 1000 samples and 3, 5 and 10 features

sian mixtures. One aspect is to evaluate run time with respect to the number of workers. Here, we assume a fixed-sized data set is partitioned into an increasing number of workers. Another aspect is to see how the changes in number of features influences the execution. Finally, these tests can provide an insight into the most appropriate number of working nodes for each data set, i.e. the number of nodes that produces the lowest execution time on the data set. There is always a tradeoff between communication and computation in parallel systems. Splitting the computation to smaller chunks, i.e. adding more nodes, reduces the time required for computation. However, the process of communication/synchronization is becoming more time consuming when increasing the number of nodes. Therefore, an optimal point (a particular number of workers) can be found, where these two aspects are best balanced.

Figs. 3.11, 3.12 and 3.13 represent the scaling properties of the algorithm, by showing the execution timings for different data sets and different number of workers. Fig. 3.11 displays the experiments for 3 data sets, where the number of samples is the same value 1000, but the number of features is different for each data set (3,5 and 10 features). It can be seen that the algorithm scales well here. Increasing the number of worker nodes reduces the execution time until reaching the optimal point of workers for the data set. After that point, the execution time starts to increase, as the cost of having more nodes is then higher than the gains of parallelization. For the 1000×3 data set, the optimal number of nodes is 8, for 1000×5 is 10 and for 1000×10 is 10 as well.

Fig. 3.12 displays the tests for 3 data sets with 5000 samples and 3,5 and 10 features. It can be seen that the algorithm scales well and it is straightforward to identify that the optimal number of nodes for each data set: 12, 20 and 30 respectively. However, there

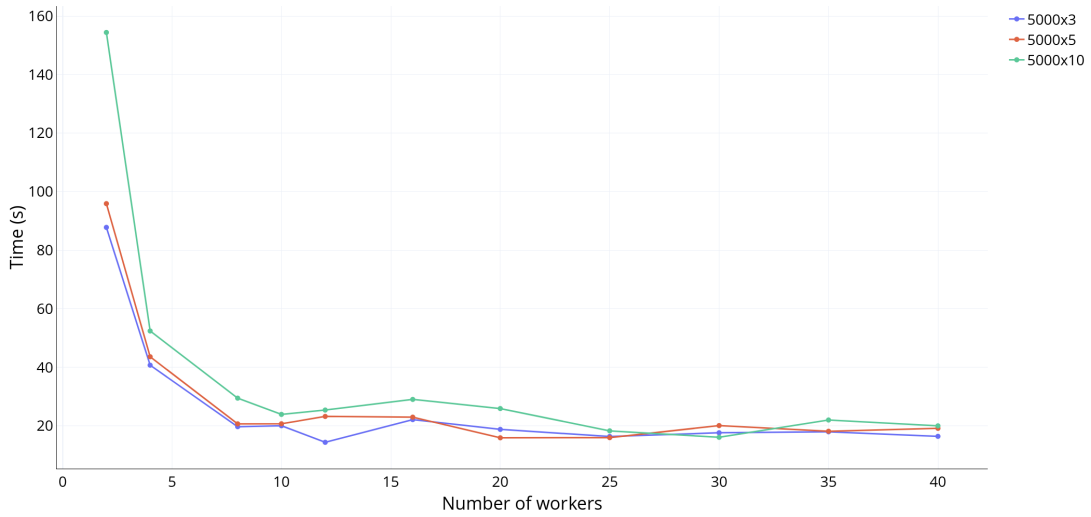


Figure 3.12: Scaling properties for the data sets with 5000 samples and 3, 5 and 10 features

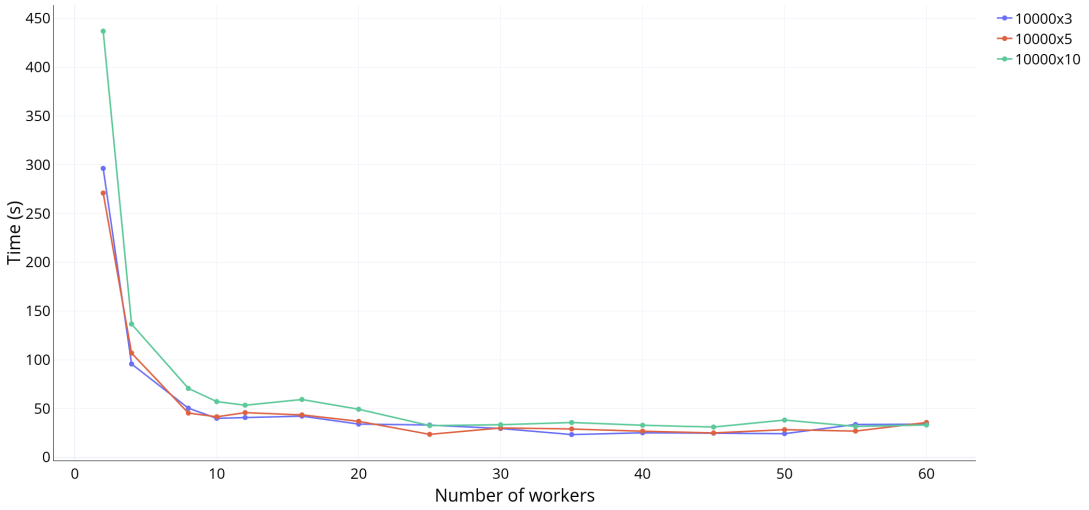


Figure 3.13: Scaling properties for the data sets with 10000 samples and 3, 5 and 10 features

is a whole range of number of workers where the execution time remains similarly low. Increasing the number of working nodes reduces the execution time until reaching the optimal range of workers for the data set. The execution time remains approximately the same, within the frames of the conducted experiments. Further increasing the number of workers could lead to execution time increase at some point, as the cost (in terms of communication/synchronization) of having more nodes would then be higher than the gains of parallelization. At some points, the execution time of a larger data set can be slightly lower than the execution time for a smaller data set. The reasons for this could be various, including latency caused by a particular node, but also the nature of the

particular data in the data set. It should be kept in mind that measuring the execution time of a synchronous parallel program always subsume waiting for the slowest process to terminate. The difference compared to Fig 3.11 is that there is a larger gap between the optimal number of points when changing the number of features, as we have a data set with larger number of samples now. The behaviour of the algorithm in these 3 cases is similar.

Fig. 3.13 shows the scaling properties for 3 data sets with 10000 samples and again 3, 5 and 10 features. The algorithm scales well here, as expected. The optimal number of workers is 35, 25 and 25 respectively, but also stays close to this optimal value for a range of different number of workers. It seems strange that we do not have the distribution for optimal points as expected: lower value for less features, higher value for more features. However, this observation is not completely true. The reason is the following: we start with small number of workers and very high execution times. As we increase the number of workers, the execution time rapidly drops, and once it is reduced to certain level, it remains close to that value for further increased number of workers. As a result, the mention optimal number of workers corresponds to the lowest execution time, but that timing is only slightly different for a whole range of tests with different number of nodes.

The displayed experiments showed that the developed algorithm exposes good scaling properties and that the gains of parallelization are evident. The execution time decreases nearly linearly with the number of workers in the experimental range of workers considered.

3.3.4 Choosing the value for the parameter γ

It is well-known that, with SON clustering, parameter γ critically influences performance. The magnitude of this parameter determines the size of steps the algorithm will take during the iterations. The number of clusters for a very small γ equals the number of data points. As γ increases, the number of clusters typically reduces. For gamma above a large threshold, the number of clusters equals one. For all-pair-penalty in [113, 114], SON clustering guarantees to find exact clustering structure, if it exists, for a range of γ. In practice, one can start with a small γ and re-solve the SON problem multiple times, each time increasing γ by a multiplicative functor. This is also known as clusterpath [38]. With our approach, for a fixed ϵ, we observe a similar behavior: the number of clusters reduces when we increase γ, while for a range of γ, we obtain exact recovery (Figs. 3.4, 3.7, 3.10).

We illustrate clusterpath on an example with 30×2 data points, that is suitable for

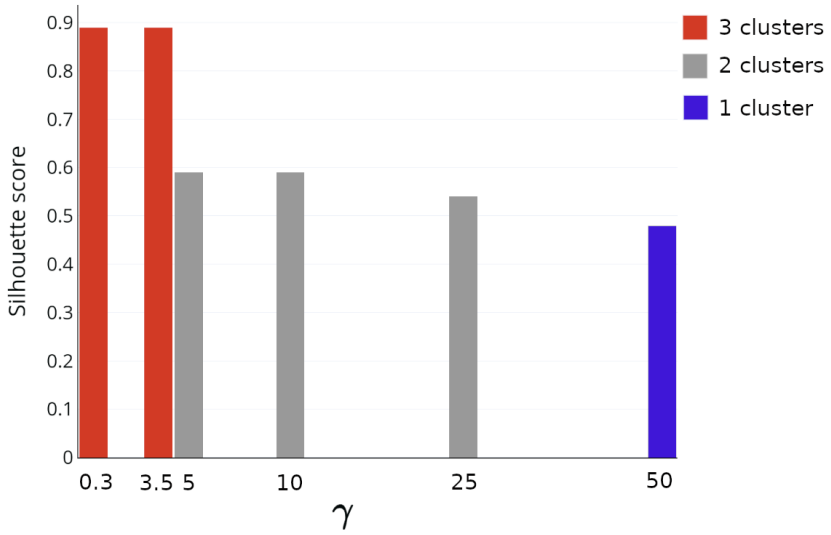
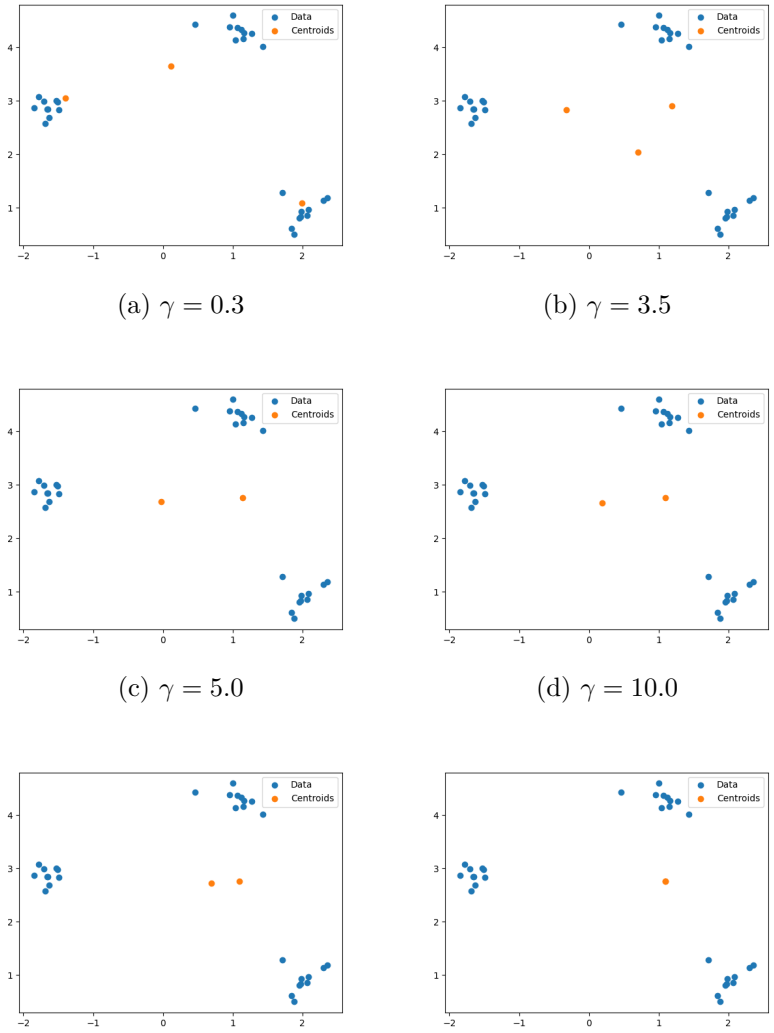


Figure 3.14: The impact of choosing different values for γ

easily plotting the results, as effects of changes in the value of γ. This data set was generated using samples generator from the *scikit-learn* package [146], that generates isotropic Gaussian blobs for clustering. Fig. 3.14 shows the silhouette score values and the number of obtained clusters for a set of experiments, performed for different values of γ. For the value $\gamma = 0.3$, the algorithm results with 3 centers and high silhouette score value. As we increase the regularization parameter, the silhouette score value is getting lower, as well as the number of clusters. For $\gamma = 3.5$, we still have 3 centers, but they are obviously closer to each other than for $\gamma = 0.3$. The silhouette score value is preserved, as the clustering of the data points in this case remain the same as for $\gamma = 0.3$. Increasing the value further as $\gamma = 5.0$, two centers already overlap, resulting with 2 clusters and lower silhouette score value accordingly. Choosing even larger values for γ forces the two centers to become even more close to each other. At the end, for $\gamma = 50.0$, only 1 center remains, as the candidate points for centers overlap. This can be also seen when plotting the resulting centers, as shown on Fig. 3.15. This trend can also be illustrated in a different way. As discussed in [35], a graphical interpretation similar to hierarchical clustering plot can be created that shows the dependency between different values of γ and number of centroids. Fig 3.16 shows the effects of γ to the number of clusters, for the 30×2 generated data set. It identifies the values of γ, where the number of cluster changes and illustrates the ranges of values where the number of cluster centers remains the same.

An approach to choose γ is to evaluate an initial value γ^*, and search in a neighborhood of that value. One way to compute γ^* is as follows:



(a) $\gamma = 0.3$ (b) $\gamma = 3.5$

(c) $\gamma = 5.0$ (d) $\gamma = 10.0$

(e) $\gamma = 25.0$ (f) $\gamma = 50.0$

Figure 3.15: Centroids for different values of γ

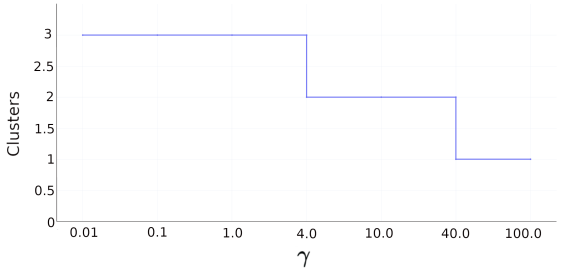


Figure 3.16: The effect of changing the value of γ on the number of clusters

$$\gamma^* = \max_{i=1...K} \frac{\max_{j \neq l}\{||a_{ij} - a_{il}||\}}{\frac{N}{K}}, \tag{3.22}$$

Then, we can consider the values $\gamma = \{\frac{\gamma^\star}{100}, \frac{\gamma^\star}{10}, \gamma^\star 10 \times \gamma^\star 100 \times \gamma^\star\}$.

3.3.5 Comparison with other clustering methods

In this section, we provide a comparison of the developed parallel ADMM-based convex clustering algorithm, to other clustering solutions. The comparison is made for the following clustering approaches: SON clustering, AMA method, DBSCAN, SSNAL and k-means from Apache Spark.

Comparison with SON clustering

As it was already stated, the parallel ADMM-based convex clustering algorithm relies on the idea of Sum Of Norms (SON) clustering, as a convexification of k-means with ADMM incorporated. A parallel implementation is naturally meant to run faster and to be able to handle larger data sets. However, in order to investigate the limitations of the serial SON clustering algorithm, we make a quick comparison here. We implemented the SON clustering approach as (3.1), in Python and tried to run it on a single computer. The machine used for the tests has 24 GB RAM and an Intel i5-4590 CPU with 4×3.30GHz. It turns out that the serial approach breaks down quite fast. Using a data set with only 500 samples and 2 features already depletes the resources of the machine and is not able to be executed.

Furthermore, in order to illustrate the speedup with the parallel approach, we decided to run the comparison for a data set, that is small enough to produce results with SON clustering inside a reasonable time frame. A data set of dimension 100×2 was generated and used here. The data set was generated with 2, clearly distinguishable clusters, as shown on Fig. 3.17. It turns out that SON clustering is able to find the solution during the first iteration already. The algorithm produced a silhouette score value of 0.79, with the 2 clusters found. The execution time was 8.95 seconds. We then ran the parallel ADMM-based clustering algorithm on our cluster environment, although it may seem as an overkill to expect gains of parallelization on such a small data set. However, the algorithm converged in 3 iterations with 2 workers, obtaining the silhouette score of 0.79 with 2 clusters found, and with overall execution time of 4.89 seconds. Clearly, the ADMM-based convex clustering is twice as fast as SON clustering for this small volume example with 2 workers. It is an expected advantage, that emerges from the nature of

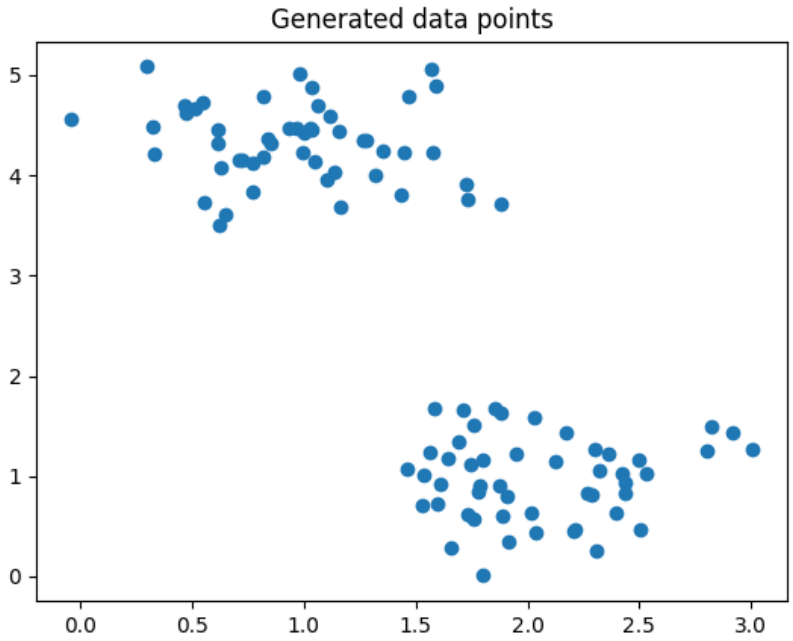


Figure 3.17: The 100 × 2 generated data set

the algorithm. On the other hand, the most important conclusion here is the same level of accuracy, that our approach exposes when compared to SON clustering.

Comparison with splitting method for convex clustering

In [36], two convex clustering methods are introduced, one based also on ADMM and the other based on alternating minimization algorithm (AMA). The authors provide a rich set of results, based on different tests on synthetic and real data. They clearly identify AMA significantly more efficient. However, the tests described are performed on at most 500 data points, as the subgradient algorithm, used as a benchmark takes a large portion of time to converge on larger data. The R code for these algorithms is available in earlier versions of the CRAN [153] repository.

In order to compare the performance of the parallel ADMM-based convex clustering to this approach, we used only the AMA splitting method, as it is more efficient. We created an R script that utilizes the AMA approach.

The AMA method defines a function named `cvxclust_path_ama` as the starting point for calling the clustering path algorithm. This function expects a set of arguments, at least the following: a data matrix to be clustered, a vector of prepared non-negative weights and a sequence of regularization parameters γ. After setting up the weights, the algorithm

Table 3.3: The comparison of execution time (in seconds) for AMA and ADMM-based convex clustering methods

Data set	**AMA method**	**ADMM -based convex clustering 4 workers**	**ADMM -based convex clustering 8 workers**	**ADMM -based convex clustering 10 workers**	**ADMM -based convex clustering 20 workers**	**ADMM -based convex clustering 25 workers**
1000×3	2.8	12.65	**9.73**	13.79	11.45	12.24
5000×3	17.55	38.38	19.95	16.45	19.35	**14.15**
10000×3	45.46	96.0	51.17	40.84	37.3	**22.01**
5000×5	22.74	39.29	19.65	16.87	16.15	**14.37**
10000×10	100.2	136.76	72.63	59.33	50.23	**30.29**
200000×3	N.A.	–	–	–	–	**564.1**

estimates the convex clustering path. Particularly, it takes each value from the sequence γ and calls `cvxclust_ama`, the function that actually performs the convex clustering via AMA method. In order to be able to compare the performance of the AMA method with ADMM-based convex clustering, we measure the execution time of `cvxclust_path_ama`, for only one value of γ, as it corresponds to our setup, where we run our algorithm once for some defined parameter value.

We run a set of tests for the AMA method. The machine used for the tests has 24 GB RAM and an Intel i5-4590 CPU with 4×3.30GHz. We also run ADMM-based convex clustering on the same data sets on our cluster, for different number of workers. The results of comparison regarding the Iris data set with known ground truth were already described previously. The results for other data sets are displayed in Table 3.3. It can be seen that for data sets of smaller volume, the AMA method can perform even better than our approach. This is observable for the data set with 1000×3 points. The AMA method, available in R language is actually a wrapper around C code, which is a low level programming language, so the performance is expectably good. Running a smaller example on a cluster naturally does not pay off as in the cases with higher volume of data. For a little larger data set with 5000×3 points, the performance of the AMA method is still very close to the performance, that can be obtained on the cluster. However, as we increase the volume of the data, it becomes obvious that the parallel ADMM-based convex clustering can perform much better. For the 10000×3 data set, it performs 2 times faster, while for the 10000×10 data set, it performs 3 time faster, when using an appropriate number of workers. Enlarging the data set size even further, leads us to cases where the AMA method cannot be run, as it cannot allocate a data structure of the defined volume, due to its serial nature. This is the case for the 200000×3 data set. The

AMA method cannot obtain the results, but the ADMM-based convex clustering methods solves the problem for 9.4 minutes with 25 workers. This illustrates the advantage of a parallel approach, that can solve large scale problems.

Comparison with DBSCAN

The ADMM-based convex clustering algorithm should also be compared to another algorithm where the number of clusters is also unknown in advance. We decided to use Density-based spatial clustering of applications with noise (DBSCAN) [117] for this purpose. We use the implementation of DBSCAN from the *scikit-learn* library. By default, the algorithm uses euclidean distance as a metric for obtaining the distance values. It accepts a parameter ϵ, representing the maximum distance between two samples for one to be considered as in the neighborhood of the other. This value can be set according to the data set. We set the number of samples in a neighborhood for a point to be considered as a core point to value 2 for all the tests, and run all the tests mentioned before with DBSCAN in order to catch the results for those ϵ values, that produce the highest silhouette score.

Table 3.4 lists the results of experiments, containing the number of clusters and silhouette score for ADMM-based convex clustering, *scikit-learn* k-means and DBSCAN respectively. The values for DBSCAN silhouette score assume that there are points labeled as noisy by the algorithm, that are not assigned to any of the clusters. For the smallest example (30×2 data set), DBSCAN performs in the same manner as our proposed method and k-means, as the number of clusters and the silhouette scores are the same. Let us consider an example of a noisy data set, where the clusters are not clearly separated. For an example of a 40×2 data set, where the data is noisy, DBSCAN performs slightly better than our method, resulting with the highest silhouette score value and (expected) 3 clusters, but also leaving some points unlabeled. For the bigger, generated data sets, DBSCAN performs very similar to ADMM-based convex clustering and k-means. For a clearer view, the silhouette score values for the mentioned test cases are also represented graphically, on Fig. 3.18. Based on these tests, we can conclude that when a clear cluster structure exists, both methods perform satisfactorily.

Comparison with SSNAL method

In [115], a semismooth Newton based augmented Lagrangian method for solving large-scale convex clustering problems was introduced, called SSNAL. It represents an efficient and robust approach for large-scale problems. The algorithm is developed in MATLAB.

Table 3.4: Comparison of ADMM-based convex clustering with DBSCAN

Data set	no clust. ADMM	ADMM s.sc.	no clust. k-means	k-means s.sc.	DB-SCAN ϵ	no clust. DB-SCAN	DB-SCAN s.sc.
30×2	3	0.89	3	0.89	0.5	3	0.89
40×2	6	0.39	5	0.57	0.8	3	0.65
1000×3	8	0.76	8	0.76	2.5	8	0.75
5000×3	4	0.77	4	0.78	2.5	4	0.76
5000×5	5	0.62	4	0.75	5.0	4	0.75
10000×3	10	0.69	10	0.75	2.5	10	0.75

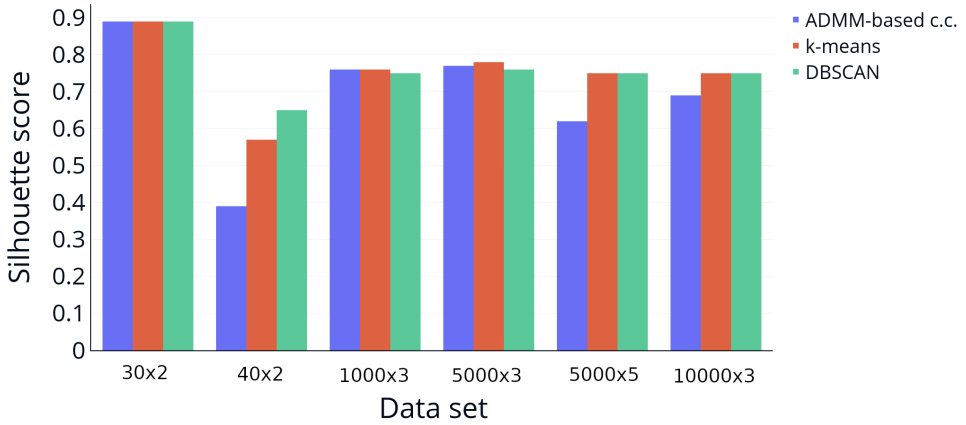


Figure 3.18: The silhouette score values for a set of tests, for ADMM-based convex clustering, k-means and DBSCAN

A comparison with the AMA method shows great advantage of SSNAL over AMA, in execution time. In order to compare ADMM-based convex clustering with SSNAL, we need a reasonably larger data set, as the performance of a parallel algorithm does not come to expression, when the expenses of parallelization and syncrhronization are higher than the gains gathered in computation. The authors in [115] mention the following large scale problem they tested: they generated a data set with 200000×3 points, representing semi spherical data. For n = 200000, they report the execution time of 374 seconds to solve the model, for the right choice of input parameters. In order to gain some insight into the timings of ADMM-based convex clustering, for a data set of same volume, let us first consider a data set of volume 200000×3 generated as a Gaussian mixture model, as we do not have access to the same data set that was used for SSNAL. The nature of these data sets differs, but this example serves just to roughly evaluate the behavior of ADMM-based convex clustering on this volume of data. Let us illustrate the results in Table 3.5. It can be seen that our proposed approach performs better than SSNAL for

100 workers. It should also be noted, that the presented time for our method also includes the time spent on merging the cluster centers when the algorithm terminates.

We expect that SSNAL cannot scale as effectively as the proposed approach on larger data sets due to the serial implementation and calculation of weights that are pair-wise across all pairs of data in the data set. To further demonstrate this, we evaluated execution times for weights calculation for data sets of different sizes. Note that the weights calculation time represents a lower bound on the execution time of the overall SSNAL method. Therefore, if the execution time of the proposed method is smaller than that of the weight calculation, it follows that the execution time of the proposed method is smaller than that of SSNAL overall. We also implement the weight assignment as in [115], and measure the time required for this kind of preprocessing. Assigning weights according to the nature of the data is very useful, but the process have a certain cost. We want to identify how much time is needed for this kind of preprocessing, in order to compare it to our execution time. The calculation of the weights, as stated in [115], can be done as follows:

$$w_{ij} = \begin{cases} exp(-0.5||a_i - a_j||^2 & if(i,j) \in E, \\ 0 & otherwise \end{cases} \tag{3.23}$$

Here, $E = U_{i=1}^{N}\{(i,j)|a_j$ is among a_i's k nearest neighbours, $i < j \leq N\}$. This kind of preprocessing is not applicable in this form to ADMM-based convex clustering, due to its distributed nature. Computing the pairwise distances among points that are assigned to different workers could be very expensive. However, it is indisputable that assigning weights could seriously affect the performance of the algorithm afterwards. Therefore, we analyze the execution time required for obtaining the weights, as well as the execution time of ADMM-based convex clustering itself. We wrote a Python script, that works sequentially with sparse data structures, and tested it for 2 data sets and a few different values of k, used for k-nearest neighbors.

The execution time for different values of k, for the mentioned 2 data sets, are shown in Table 3.6. For the smaller data set with 200000×3 data points and k=10 and k=100, the required time for weights calculation is low, compared to the overall execution time of 9.4 minutes on the cluster. However, when increasing k to $k = 1000$, the time grows to 15 minutes, that is longer than the time required to solve the problem by ADMM-based convex clustering on the cluster. Considering the larger data set, with 2000000×3 data points, the execution time for weight calculation is extensive. For $k = 100$, it takes almost an hour. Setting the parameter k for KNN is always an open issue. However, for a data

Table 3.5: The comparison of SSNAL and ADMM-based convex clustering methods, in terms of execution time

SSNAL (on 200000×3 semi spherical data)	**ADMM-based convex clustering on 200000×3 gaussian mixture data with 25 workers**	**ADMM-based convex clustering on 200000×3 gaussian mixture data with 50 workers**	**ADMM-based convex clustering on 200000×3 gaussian mixture data with 100 workers**
374s	564.14s	744.01s	315.76s

Table 3.6: The execution time required for obtaining the weights

Data set size	**k for KNN**	**Time**
200000x3	10	24.4s
200000x3	100	61.38s
200000x3	1000	959.32s
2000000x3	10	18.46 min
2000000x3	100	59.05 min

set of large volume, as the data sets displayed here, it is likely that a larger value of k will be needed. This could result with a very time consuming preprocessing step, that can actually be higher than the execution time for ADMM-based convex clustering on a cluster, that does not use any proprocessing. Our approach uses pairwise distances only within workers, when the value of ϵ is being computed, for the merging process.

Comparison with another parallel implementation

The state of the art regarding parallel clustering algorithms is mainly oriented towards utilizing the Apache Spark framework, as described in [154]. As it represents a completely different technology, it is interesting to examine the performance of a Spark-based clustering algorithm, and compare it to ADMM-based convex clustering, that is based on COMPSs. We use Python in both cases. Apache Spark provides a parallel version of k-means algorithm, that is part of the MLlib [155] library. We created a Python script, that loads the data and calls k-means from MLlib. The disadvantage of this approach is that we need to specify the number of clusters in advance. However, in this case, we already know the desired number of clusters for our generated data sets, so we can use these values to test the performance. As our cluster currently does not support Spark,

Table 3.7: Comparison of ADMM-based convex clustering to Spark based k-means from MLlib, in terms of execution time

Data set	**Spark k-means time for 2 workers**	**Spark k-means time for 4 workers**	**ADMM conv. clust. COMPSs time for 2 workers**	**ADMM conv. clust. COMPSs time for 4 workers**	**ADMM conv. clust. Best time / no workers**
1000×3	21.27s	26.14s	18.44s	14.64s	8.77s/8
5000×5	24.31s	30.31s	95.9s	43.6s	15.92s/20
10000×10	26.79s	33.26s	437s	136.57s	31.61s/55

we set up a standalone Spark Server on our computer and tested the performance for different number of workers, specifically for 2 cases: 2 and 4 workers. Each worker is assigned a CPU unit and 4GB of memory. As the CPU has 4 cores, 4 is the maximal number of workers used for the tests. Table 3.7 lists the execution time for each test, for a few different data sets. It also includes the results for the same number of workers, obtained for the ADMM-based convex clustering COMPSs implementation. It should be noted that these tests were run on a cluster, so the comparison can be made only roughly.

From Table 3.7, it can be concluded that the ADMM-based convex clustering COMPSs based algorithm performs better for the 1000×3 data set, than MLlib k-means. However, as the data size grows, the MLlib k-means shows better performance for 2 and 4 workers. It is also noticeable that the addition of workers is expensive in these cases, so the Spark based clustering does not scales well for these data sets. The reason is probably the moderate volume of these data sets. The best obtained time for ADMM-based convex clustering for larger number of workers is interesting for observation here, as it can be seen that in the first two cases, it is far better than the Spark timings, but for the third case, the Spark based k-means performs better than ADMM-based convex clustering in the best case. This comparison is only for illustration. It should be kept in mind that these two algorithms are relying on completely different principles and are implemented in different technologies. Finally, the tests are not made on the same machine, although the machine used for Spark is similar to the cluster nodes. However, it can be seen that the Spark based implementation is expected to perform better for even larger data sets, and also that the COMPSs based ADMM-based convex clustering implementation needs a larger number of workers to achieve the optimal performance, where the Spark implementation can accomplish good results with less workers. The reasons behind this are lying in the differences in the natures of the algorithms and frameworks.

3.3.6 Testing on a real, industrial data set

The developed algorithm was also tested on a real, industrial data set, gained through collaborations on a H2020 project, I-BiDaaS [47]. The data set is from the banking sector and is publicly available on the Zenodo repository [49]. The data set was generated collecting relevant information of bank transfers, done by a bank employee. The aim was to identify anomalies that lead to potential frauds or bad practices.

Therefore, the data set was already used for similar clustering analysis under the I-BiDaaS project, the results are available in the public Deliverable 3.3 of the project [156]. The parallel ADMM-based convex clustering algorithm is able to find the same number of clusters, as reported in the deliverable, with a silhouette score value that corresponds to the results in the deliverable, so the accuracy level of this evaluation corresponds to the previously obtained results.

Fig. 3.19 displays the scaling properties of the algorithm on this data set. It can be seen that the algorithm scales well again. Although the data set is not very large, the execution is more efficient with parallelization. The best choice for the number of workers appears to be 10 here. Adding more than 10 workers does not increase performance any more, as the cost of parallelization then becomes more demanding than the gains. However, these tests showed that the ADMM-based clustering algorithm has good scaling properties, and an acceptable level of accuracy as well. The speedup of the algorithm on this data set is displayed on Fig. 3.20. It is necessary to mention here that the algorithm is designed to work in parallel, and it requires a separation to x update on master and x update on the rest of the nodes, as in (3.14) and (3.15). This means that it assumes the existence of at least 2 worker nodes. In order to run the code serially, we need to keep this separation. The updates are then executed serially, but we need to split x to x_0, that is being solved as (3.15) and $x_{1..k}$, that is being solved as (3.14). However, the way of splitting this value significantly influences the execution time. Therefore, a fair comparison of the serial and parallel execution time is to keep the separation as in the parallel test during serial execution, for the observed number of workers. This results with the speedup as shown on Fig. 3.20. It can be observed that the speedup is increasing until reaching the optimal number of workers, and after that it decreases a bit, that corresponds to the conclusions made on Fig. 3.19.

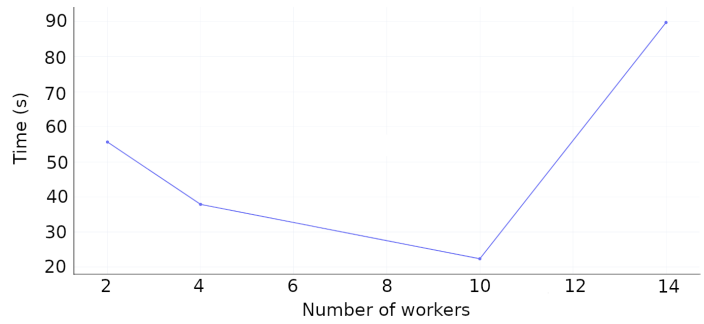


Figure 3.19: The scaling properties on the Caixa Bank data set

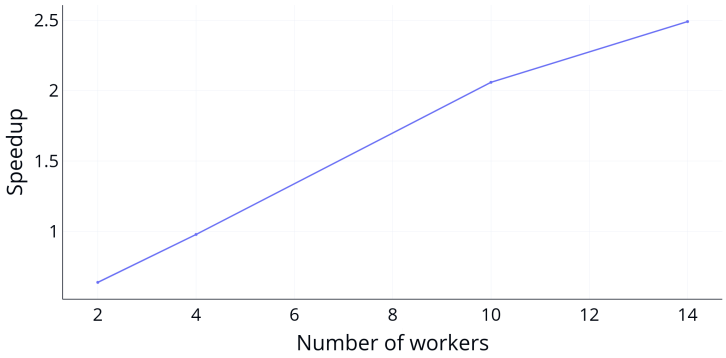


Figure 3.20: Speedup on the Caixa Bank data set

3.4 Further implementation considerations

Enhancing the CVXPY solver performance

The current implementation of the proposed method utilizes CVXPY to solve sub-problems and it can be set to use the so called "warm start" option. Generally, improvements in execution time can be expected if we use a "warm start" in updates (3.14), (3.15) and (3.16) and initialize the new variables with their values from the previous iteration. In addition, as commonly used with ADMM, the sub-problems (3.12) may not be solved to full accuracy, i.e., they can be solved inexactly. We examine these two strategies on the CVXPY implementation. This way the solver starts with the solution from the previous iteration, which can reduce the execution time. Another approach is to set a solver property such that the expected accuracy becomes lower during the first few iterations. The ECOS solver, used by CVXPY in this case, has the property *abstol*, which corresponds to absolute accuracy tolerance and it has a default value of $1e-8$. By enlarging this value, we permit greater difference i.e. lower accuracy. We tested this approach by using 0.0001 during the first 3 iterations of the algorithm.

Let us illustrate the impacts of these enhancements to performance. Table 3.8 shows the

Table 3.8: The impact of solver enhancement on performance

Data set	No enhancement	Warm start	Accuracy adjustment	Warm start and accuracy adjustment
10000x3	38.65 s	37.5 s	37.7 s	35.88 s

execution time for different variants of the algorithm: without enhancement, with warm start, with tolerance set up and with both warm start and tolerance set up. Apparently, these enhancements can reduce the execution time to certain extent. For the data set with 10000×3 points, the time reduction is 2.7 seconds, i.e. 7% roughly. This is not a drastic difference, but it certainly represents an improvement in performance.

An alternative problem solving approach

In addition, sub-problems (3.12) can be solved by adopting efficient moderate-size problem SON clustering solvers like [109], instead of using the general-purpose solver like CVXPY. This approach is described as follows. Let us redefine the solution update on the worker nodes, by introducing inner iterations $t = 0, 1, ...$ In order to update x (see Algorithm 3), let us define, for each node $i = 2, ..., K$:

$$x^{t+1} = z^{t+1} + \frac{t-1}{t+2}(z^{t+1} - z^t), \tag{3.24}$$

where z is being computed as:

$$z^{t+1} = x^t - \frac{1}{L}g^t. \tag{3.25}$$

Here, $x^t = (x_{i1}, x_{i2}, ..., x_{i\frac{N}{K}})$ and $L = 1 + \frac{2\gamma\frac{N}{K}}{\mu}$, $\mu = \frac{2\epsilon}{\gamma\frac{N}{K}}$, $\epsilon = \{10^{-1}, 10^{-2}\}$. x^t and z^t are being initialized with x^{t-1}. The gradient $g^t = (g_1^t, g_2^t, ..., g_{\frac{N}{K}}^t)$ is being calculated for nodes $i = 2, ..., K$ as follows:

$$g_1^t = -\lambda_i^k + \rho(x_{i1}^t - y_{i1}^k) + (x_{i1}^t - a_{i1}) + \gamma \sum_{j=2}^{\frac{N}{K}} P\left(\frac{x_{i1}^t - x_{ij}^t}{\mu}\right), \tag{3.26}$$

and

$$g_j^t = (x_{ij}^t - a_{ij}) - \gamma P\left(\frac{x_{i1}^t - x_{ij}^t}{\mu}\right), j = 2, ..., \frac{N}{K}. \tag{3.27}$$

where:

$$P(a) = \begin{cases} \frac{a}{||a||} & \text{if} ||a|| > 1 \\ a & \text{else.} \end{cases} \tag{3.28}$$

The x update on the master node is the same as for the others, except for g_1^t:

$$g_1^t = (a_{11} - x_{11}^t) + \gamma \sum_{j=2}^{\frac{N}{K}} P\left(\frac{x_{11}^t - x_{1j}^t}{\mu}\right) + \gamma \sum_{j=2}^{K} P\left(\frac{x_{11}^t - y_{i1}^k}{\mu}\right). \tag{3.29}$$

The y update on the master node is now:

$$y_{i1} = x_{i1}^{k+1} + \delta_{i1}, \tag{3.30}$$

where $\delta_{i1} = Prox\left(\frac{-\lambda_i^k}{\rho}\right)$ and $Prox(a) = \left[1 - \frac{\gamma}{\rho ||a||}\right]_+ a$, and $[b]_+ = (max\{0, b1\}, max\{0, b2\}, ...)$.

The implementation of this approach is also available on GitHub [2]. The parallelization and synchronization points remain the same as before, only the computation during the x and y update change according to (3.24) and (3.30).

This approach can exhibit equally good or even better performance than CVXPY. For example, some initial tests show that the data set with 10000×3 points (generated as a Gaussian mixture model), can be solved for 6.7 seconds with 25 workers, where with the CVXPY solver, the same problem was solved for 22 seconds, with the same number of workers. The accuracy of the solution is the same as for CVXPY. Therefore, this approach represents a promising direction for further enhancement.

3.5 A comparison of MPI and COMPSs parallel applications

The development of parallel algorithms always brings up the question about choosing the best possible parallelization framework. There does not exist a unique criterion for making this kind of choice. Through this thesis, we discussed the use of two different parallelization approaches, MPI and COMPSs. We developed the implementations for primal methods with MPI in C programming language, and for dual methods with COMPSs in Python. Both approaches have their own advantages. In order to make a fair comparison of these approaches regarding performance, we need the same algorithm (based on

Table 3.9: Comparing the execution time (in seconds) of ADMM-based convex clustering with MPI and COMPSs

	4 workers		8 workers		10 workers		20 workers		25 workers	
Data set	**MPI**	**COMPSs**	**MPI**	**COMPSs**	**MPI**	**COMPSs**	**MPI**	**COMPSs**	**MPI**	**COMPSs**
1000×3	5.83	12.65	4.99	9.73	3.18	13.79	2.59	11.45	4.26	12.24
5000×3	34.17	38.38	15.36	19.95	10.85	16.45	5.49	19.35	10.05	14.15
5000×5	38.71	39.29	18.82	19.65	13.26	16.87	10.08	16.15	7.94	14.37
10000×3	95.13	96.0	29.89	51.17	29.08	40.84	13.31	37.3	15.53	22.01
10000×10	153.09	136.76	42.57	72.63	47.44	59.33	29.24	50.23	15.97	30.29

the same optimization method), implemented in the same programming language, parallelized with the two frameworks, and tested on same data. In order to achieve this, we created an implementation of the parallel ADMM-based convex clustering algorithm in MPI for Python. This practically means replacing COMPSs with MPI in the code. This is not as straightforward as it may seem, as the frameworks work on different principles. The source code for this approach is also available on ADMM-based clustering GitHub repository [2].

The main difference is that, with COMPSs, we assumed that one process (the master) performs everything, until we invoke a task. After that, with one call, we were able to synchronize the results to make them available again to a single operating process. With MPI, we need to specify explicitly that the master process performs the logic meant for serial execution. Also, the initializations are now different. Instead of having a list of data chunks that is being split by calling a task, we need to initialize local data structures on the processes. The functions that used to be tasks with COMPSs are now executed locally on the processes on local data. Instead of having a line of code that synchronizes the results to be used further by the main (master) process, we need to call a corresponding MPI routine to gather the results on the master. Similarly, in order to make the global values available to the processes, we need explicit broadcasts. All these differences arise from different setups of the frameworks. After adapting the algorithm to use MPI, we run a set of tests in order to compare performance.

Table 3.9 displays the results of tests, meant for comparing MPI with COMPSs. We observe the execution times of the two implementations for a set of worker numbers, on a few different data sets. We use the same data sets, that we already utilized for the tests on ADMM-based convex clustering with COMPSs. Both implementations show good scaling properties, and achieve the same level of accuracy, as described before for the mentioned data sets. For the smallest data set, with 1000×3 data points, there exists a difference in the optimal number of points, as COMPSs has the best performance for 8 workers, where MPI performs the best with 20 workers. It is also obvious that for this

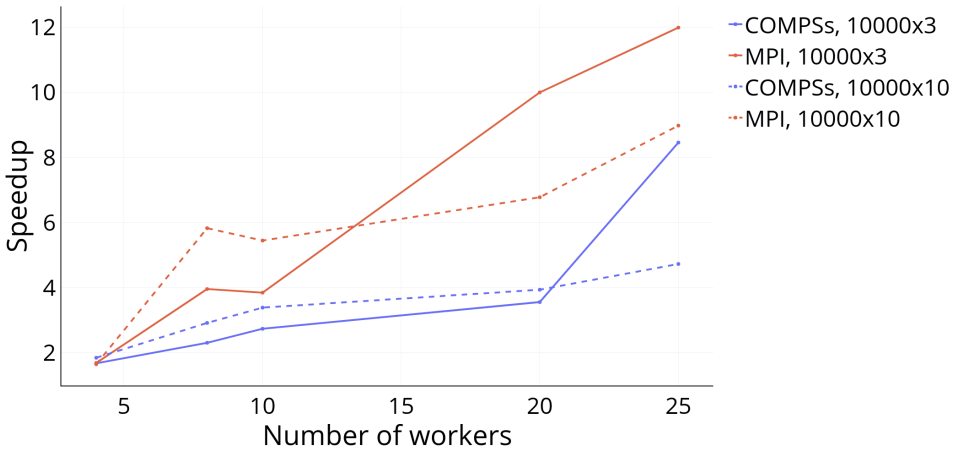


Figure 3.21: Speedup for MPI and COMPSs implementations of ADMM-based convex clustering

data set, the timing does not change a lot when changing the number of workers, only a few seconds, for both implementations. However, when comparing the performance of the two implementations, there exists a serious difference, MPI is 2 to 4 times faster, when observing the same number of workers. On the other hand, the best time for MPI is 2.59 seconds, where for COMPSs is 9.73, which also represents an important variance. For the data sets 5000×3, 5000×5 and 10000×3, the implementations perform very similar for 4 workers. In fact, the differences in timings for 8 and 10 workers are of smaller volume, in most of the cases. Still, MPI performs better. For the largest data set with 10000×10 data points, COMPSs has a lower execution time than MPI for 4 workers. When increasing the number of workers, both implementations show a decrease in execution time. However, in case of MPI, this decrease is steeper, and the best timing for MPI is 2 times faster than for COMPSs, for the same number of workers. The speedup for both implementations, MPI and COMPSs, is displayed on Fig. 3.21, for the 2 largest data sets, considered in Table 3.9, i.e 10000×3 and 10000×10. Both implementations show a certain level of speedup, for both test cases. However, the advantage of MPI over COMPSs is clearly visible. The speedup is generally higher for MPI than for COMPSs, and is also growing faster.

As the tests showed, MPI is able to achieve better performance than COMPSs. This is not surprising, as MPI represents a lower-level approach. Earlier works also showed that COMPSs has comparable performance with Apache Spark [20], and other results proved the advantage of MPI over Apache Spark [22], in terms of performance. Now, we showed directly that MPI performs a few times faster than COMPSs in most of the cases. Although the advantage of MPI is inviolable in terms of performance, higher-level approaches, as COMPSs, have other advantages over MPI. The most important one is the

ease of parallelization with COMPSs, as it provides a high level of transparency. There is no need for managing communication or possible deadlocks. We do not have to deal with complex operations to achieve parallelization. Adding a `@task` annotation simply makes a function parallel. This automatically means less control than with MPI, but COMPSs is convenient for a broader number of developers, as it does not require extensive experience.

The choice of a parallelization framework hence depends on the particular needs, but the benefits of different approaches should be kept in mind. When the aim is to create a parallel algorithm quickly and easily, COMPSs is a reasonable choice. It scales well, and accelerates the execution properly. However, when the speed of an algorithm is a critical demand, MPI remains the best choice.

3.6 Additional ADMM-based machine learning algorithms

The broad usability of the ADMM approach can be illustrated by applying it to implement some well-known machine learning algorithms. We describe two additional algorithms, beside the already mentioned convex clustering approach: ADMM-based lasso regression and ADMM-base logistic regression. The parallelization framework used here is also PyCOMPSs. These implementations were developed under the H2020 project I-BiDaaS as part of the projects' pool of machine learning algorithms and are publicly available on the GitHub knowledge repository of the project [157].

3.6.1 ADMM-based lasso regression

The Least Absolute Shrinkage and Selection Operator (lasso) represents a regression model, that is suitable for both variable selection and regularization. Particularly, lasso is a type of linear regression that uses shrinkage, where data values are being shrunk towards some central point. The algorithm performs L1 regularization and can be defined as:

$$minimize \frac{1}{2}\|Ax - b\|_2^2 + \lambda\|x\|_1, \tag{3.31}$$

where $\lambda > 0$ is the regularization parameter and $A \in \mathbb{R}^{n \times p}$ and $b \in \mathbb{R}^n$. Then, keeping in mind the already introduced definition of ADMM, the formulation can be rewritten to ADMM form as:

$$minimize \frac{1}{2}\|Ax - b\|_2^2 + \lambda\|z\|_1 \quad subject\ to\ x - z = 0. \tag{3.32}$$

Finally, the ADMM steps follow the standard ADMM form, as defined in [3]:

$$x^{k+1} = (A^\top A + \rho I)^{-1}(A^\top b + \rho(z^k - u^k)), \tag{3.33}$$

$$z^{k+1} = S_{\lambda/\rho}(x^{k+1} + u^k), \tag{3.34}$$

$$u^{k+1} = u^k + x^{k+1} - z^{k+1}, \tag{3.35}$$

where $\rho > 0$ is the penalty parameter for constraint violation, and x and z are the primal variables and u is the dual variable here; $S_{\lambda/\rho}$ is a soft thresholding (shrinkage) operator, that moves a point towards zero. We compute x^{k+1} by minimizing the objective function in each iteration as 3.32. Further, the soft threshold operator is being computed as:

$$S_k(a) = \begin{cases} a - k, a > k \\ 0, |a| \leq k \\ a + k, a < -k \end{cases} \tag{3.36}$$

The implementation of the algorithm in Python with PyCOMPSs can be found on the GitHub knowledge repository of the I-BiDaaS project [157]. The parallelization is based on a similar idea, as for ADMM-based convex clustering. The main loop performs the updates in each iteration and checks whether the stopping criterion is met. The function that updates x is defined as a task and being executed in parallel, by passing the already prepared split data structures. A maximum number of iterations is defined, but the algorithm could stop earlier, when the stopping conditions are met. We initialize x and u as lists of n vectors of size p, meaning that we have a vector of p elements for each of n workers, for both variables. z is initialized as a p-sized zero vector. Additionally, the update for u can also be defined as a task, although its computation is not demanding.

The task method for updating x is called, using mapping on `partial` (as it was also the case for convex clustering), meaning that this call is being executed in parallel, for each worker on a separate data and label chunks. The vector u is also being divided between the workers, as it contains a list of vectors, as explained. z and rho are the values that are

used by each worker as they are. The stopping criterion and the synchronizations are done in a same way as it was described for ADMM-based clustering. It is, again, important to synchronize the result of parallel execution, in order to gather all the elements of x. We achieve this by `compss_wait_on`. Then, the update of z is executed sequentially, by calculating a soft threshold value. The rest of the code is dedicated to the evaluation of the stopping criterion. We calculate the square roots of the sum of squared norm of x and u, respectively. The primal and dual residual actually show how much progress the algorithm is doing from one iteration to the other in both the primal and dual domain. These values tend to be smaller than the primal and dual feasibility tolerance values.

The input data reading is performed in parallel here, as it is assumed that the data set is previously split to a number of input files, that corresponds to the number of workers. That means that the functions for reading the input matrix and vector are defined as tasks. This approach is convenient, as we save time with reading data chunks separately, without the need to split them. However, this usually requires a preprocessing step, that actually splits the data and creates the needed number of chunks saved to separate files.

The CVXPY Python package is used to solve the optimization problem here. We create the solution variable first. Then, we formulate the problem as a minimization of our objective function. Finally, the problem is being solved by CVXPY. When solving the problem, the *warm_start* option can be also used. It represents a logical value, that indicates whether to use the warm start option. This means that the solver can use some previous solution or initial value in order to produce the solution faster.

In order to evaluate the performance of the developed algorithm, The ADMM lasso implementation was tested for a synthetically generated data set. The results of this evaluation are available in Deliverable 3.3 of the I-BiDaaS project, where it can be observed that this example illustrate the high rate of speedup with increased number of workers of the parallel ADMM-based lasso implementation on a computer cluster. The implementation of the parallel ADMM-based lasso regression was also included to the *dislib* repository [51], through a collaboration between the I-BiDaaS and *dislib* teams. *Dislib* represents an effort to adapt the highly efficient *scikit-learn* machine learning implementations to the highly parallelizable COMPSs runtime.

3.6.2 ADMM-based logistic regression

As a well known binary classification approach, the logistic regression algorithm can be also implemented as an ADMM-based solution, as defined in [3]. We used ADMM for

logistic regression with L2 regularization, therefore it ensures preserving convexity. The starting point is an L2 regularized logistic regression model, defined as:

$$minimize \sum_{i=1}^{m} log(1 + exp(-b_i(a_i^\top x_1 + x_0))) + \frac{1}{2}\lambda\|x\|_2^2, \tag{3.37}$$

where $a_i \in \mathbb{R}^n$ is a feature vector, $b_i \in \{-1, 1\}$ represents the corresponding label, $x_0 \in \mathbb{R}^n$ and $x_1 \in \mathbb{R}$ are the optimization variables and $\lambda > 0$ is the regularization parameter. By applying ADMM to this problem as in [3], we obtain the steps of the algorithm as follows:

$$x^{k+1} = minimize\ l(-bAx) + \frac{\rho}{2}\|x - z^k + u^k\|^2, \tag{3.38}$$

$$z^{k+1} = S_{\lambda/\rho N}(x^{k+1} + u^k), \tag{3.39}$$

$$u^{k+1} = u^k + x^{k+1} - z^{k+1}, \tag{3.40}$$

where the variables have the same meaning as for lasso regression. l represents a logistic loss function, where $l(x) = log(1 + e^{-x})$. The code for this algorithm is also available on the GitHub knowledge repository of the I-BiDaaS project [157]. Mostly, the code for lasso regression can be reused here, as the structure is the same, it only requires some slight changes, that correspond to (3.38). This particularly means that only the computation of the objective function needs to be changed in the code.

In order to evaluate the developed approach, the ADMM logistic regression was tested on a synthetic data set, provided by CRF, under the project I-BiDaas. The data set is publicly available on the Zenodo repository [50]. The tests showed a level of accuracy, comparable to the accuracy of the *scikit-learn* logistic regression on the same data. The evaluations also showed good scaling properties of the algorithm on the mentioned data set. The results of these experiments are publicly available in Deliverable 3.3 [156] of the project.

The ADMM solver represents a good choice for distributed problem solving and is well suited for different well-known machine learning algorithms. The developed implementations can serve as a base for introducing other ADMM-based solutions, parallelized by PyCOMPSs. In fact, the Python code, available in the I-BiDaaS knowledge repository is

highly reusable. This makes the development and application of parallel machine learning algorithms eased and more transparent.

3.7 Conclusions on the proposed utilization of the dual ADMM method

In this chapter, we introduced a parallel ADMM-based algorithm for convex clustering and provided a parallel implementation for it. Also, a detailed description of a thorough testing of the algorithm was provided. The tests were performed on the AXIOM computing facility. The configuration of the AXIOM computing facility consists of 16 nodes, where each node has a processor with 6 CPU cores (eighth-generation Core i7 cores), and the nodes are connected by an Ethernet network with speed of 10Gbps. The described behaviour of the algorithm during tests should be preserved when tested on other cluster environment. The execution time may be shorter on a cluster with newer generation of processors, but the overall performance characteristics as the scaling properties and the advantages of parallel execution are expected to be the same. A higher network speed and lower latency is expected to produce good performance with more nodes than in our experiments, so the range of the number of nodes with lowest execution time would be different, but still detectable.

The framework used for parallelization is COPMSs, that represents a convenient, task-based programming model. The comprehensive empirical evaluations prove that the algorithm satisfies a similar level of accuracy as the other widely used clustering approaches. It was also shown that the algorithm can work with large data sets efficiently, exhibiting good scaling properties on a cluster environment. The algorithm was also tested on a real, industrial data-set, and exposed the expected outcomes. Additionally, a description of the implementation of other ADMM-based solutions for machine learning is also provided and supported by a set of evaluations. This motivates the utilization of ADMM to obtain parallel solutions, that are efficient and scalable. Finally, we use the developed dual optimization clustering solution to compare two different parallelization strategies, MPI and COMPSs.

Chapter 4

Conclusion

This thesis focuses on development and practical evaluation of a set of parallel distributed convex optimization algorithms. It consists of two main directions. The first one is directed towards developing a class of primal methods of first and second order. The focus is on the practical evaluation of the properties of the proposed methods, their mutual comparison and also on development of novel methods based on empirical evaluation insights. This includes the utilization of different communication sparsification strategies, resulting with a novel approach of unidirectional sparsified communication during parallel execution. The second direction is focused on the class of dual methods. Particlualry, it relies on ADMM as a representative of this class, in order to develop and evaluate a novel parallel, convex clustering approach. It also introduces two additional ADMM-based implementations. The thesis uses two different frameworks for parallelization, MPI and COMPSs. The class of primal methods is developed using MPI, while the dual, ADMM-based solutions are developed with COMPSs. This provides a deep insight into the different possibilities for parallel application development. A comparison of the performances of the two approaches is also provided, by implementing the same algorithm, the parallel ADMM-based convex clustering, with the two frameworks. All the developed code, discussed in the thesis is publicly available on GitHub [2, 1].

The results show that the developed class of primal methods can find its use in different scenarios. Currently, the methods are developed for logistic and quadratic loss, but can be easily extended to other, widely used algorithms, in order to provide scalable and fast solutions to real-world problems. The methods are being analysed from different aspects and the thesis also includes a discussion on the properties of input data sets, that influence the choice of particular methods, that could produce the best performance. The parallel ADMM-based convex clustering algorithm can be applied to arbitrary clustering problems, as its level of accuracy on the scenarios we considered is comparable with the accuracy of other clustering solutions. Additionally, it represents a parallel, scalable

approach, that is able to solve large-scale problems. The usefulness of the method was demonstrated on a real data set, provided by Caixa Bank during the I-BiDaaS project [49]. This represents an example of the utilization of the algorithm and its possibilities for a broader usage in emerging areas of data analysis. Regarding the properties of the developed algorithms, both primal and dual, they could evidently find their use as Big Data solutions, for real-world problems. Their extensible nature enables widening the set of supported algorithms and hence the area of their use.

4.1 Summary of Thesis Achievements

The achievements presented in this thesis are multiple. First, it represents an overview of parallel distributed optimization algorithms development. It shows some interesting implementational aspects, while highlighting the possible performance bottleneck and providing solutions for them. It uses two different parallelization frameworks with extensive explanations of their different aspects. Further, it introduces some novel methods with sparsified communication, based on the experimental insights, as well as the novel ADMM-based convex clustering method implementation.

A detailed empirical evaluation of the developed methods is also shown. It analyses a wide variety of aspects regarding performance and properties of the algorithms. The class of primal methods considered is design to work under a given fixed arbitrary connected network. On the other hand, the underlying cluster infrastructure allows for all-to-all communications. The experimental evaluation shows that introducing communicators that comply with the underlying (sparse) graph topology, improves performance over an all-to-all communication protocol. The use of communicators enables utilizing only a subset of wired links available, which replaces physical data exchange among nodes in an all-to-all fashion, that ignores the underlying graph topology. Moreover, it was observed that the amount of time required for the data exchange drastically decreased further by introducing a probabilistic sparsification to these methods. It was also shown that ADMM represents a valuable choice for developing a distributed, parallel solution for clustering implemented over HPC infrastructures in the first place, but also for other machine learning algorithms. Furthermore, it was shown that the developed methods all scale well, and that the optimal number of nodes exists for each of the test cases. Regarding the class of primal methods, it was also detected in which cases the second order methods perform better, and in which ones are the first order methods dominant. The best performing method regarding all the tests was also identified. The thesis also highlights the communication sparsification strategies that are of interest, as they reduce

the execution time of the algorithm. It also provides comparisons among different aspects of the developed methods and makes conclusions based on them, in order to identify the suitable methods for particular setups. From the aspect of dual methods, it was shown that the proposed convex clustering approach is capable of solving the assigned clustering tasks in an accurate and fast manner.

The thesis covers a set of distributed convex optimization methods, and makes comparisons on different levels. It makes a comparison of the developed primal methods with the ADMM-based solution for the same problem. It also compares and discusses the performance and ease of development for the mentioned two parallelization frameworks, MPI and COMPSs. The main contribution includes an introduction and analysis of a set of distributed, parallel and scalable solutions, that can be applied on a wide range of problems.

4.2 Applications

The results described in the thesis open a few application possibilities. First of all, the developed implementations can be used on real data sets to solve some real problems. An example was demonstrated by applying the parallel ADMM-based convex clustering algorithm to the I-BiDaaS use case for the Caixa Bank data set. This data set was analysed with the aim to identify anomalies that lead to potentially fraudulent bank transfers or bad practices. Similar applications may arise, not only for the clustering solution, but also for the developed class of primal methods. The application of these parallel solutions can be achieved in an efficient, scalable manner, on a computer cluster. The parallel, distributed nature of the algorithms opens the possibility to utilize them in real-world Big Data scenarios.

Secondly, as already mentioned, the implementation process of the algorithm can be applied to the development process of similar algorithms, as the main bottlenecks were empirically found and the solutions for them were proposed, implemented and tested on a cluster environment, in order to prove the performance gains. This means that the class of the proposed algorithms is extensible to even broader use. The set of applications for the class of primal methods can be widen for different cost functions, by replacing the computations for the gradient and Hessian in the implementation. Also, the described ADMM-based approach can be accommodated for different objective function specifications.

4.3 Future Work

There are a few possible directions for future work, from the aspects of both primal and dual methods. Regarding the class of primal methods, there are several possibilities. First, further evaluation of unidirectional communication can be an interesting task, theoretically and practically as well. We provided some initial empirical results on the algorithm that is utilizing unidirectional communication, that prove that this principle produces good performance during execution, in some cases even better than the bidirectional approach. This could be expanded further, by additional experimentation, in order to get more detailed insights into the properties of the algorithm setup that lead to performance increase when using unidirectional communications. Also, second order methods could be also enhanced, as they have good performance characteristics for smaller data sets. As for larger data sets, exact second order updates are time consuming, it might be of interest to consider Quasi-Newton methods with cheaper second-order-type updates, as they can possibly reveal new insights towards efficient execution. An equally useful task for the future could also be to expand further the algorithm implementation for other convex cost functions, in order to make it applicable for a broader use. From the aspect of dual methods, the speed of the ADMM-based convex clustering algorithm could possibly be enhanced by further designing efficient sub-problem solvers. We already proposed an approach that seems promising, and that could be further elaborated. In addition, data-dependent sparse graph construction and weighted sparse SON penalty can be considered with an additional preprocessing cost. Another useful enhancement for both types of methods might be adapting the implementation for very large data sets that cannot be held in memory at once. This means that the input data should be divided during the preprocessing, before running the algorithm. This way, the processes could read their own chunks separately. This is feasible to accomplish, similarly as in the described alternative implementations. The second aspect of this enhancement is the possibility for the process to not consume the whole data chunk it possesses, at once. This could require some further modifications in the algorithm behaviour as well, but could be a very useful innovation.

Source code and reproducibility

The source code for our methods can be found on the following links:

- The class of primal optimization methods:
  https://github.com/lidijaf/parallel_primal_optimization_methods
- The ADMM-based convex clustering algorithm:
  https://github.com/lidijaf/Parallel-ADMM-based-convex-clustering
- ADMM-based lasso and logistic regression:
  https://github.com/ibidaas/knowledge_repository

Some of the data sets used in the thesis are publicly available, and after performing a needed set of preprocessing tasks (where needed), they can be used with the provided implementations. The data sets and graph structures, synthetically generated for the tests, can be recreated using the provided code. Alternatively, the data sets are available on request.

Bibliography

[1] "A class of parallel primal optimization methods of first and second order." https://github.com/lidijaf/parallel_primal_optimization_methods. Accessed: 2022-06-28.

[2] "Parallel admm-based convex clustering." https://github.com/lidijaf/Parallel-ADMM-based-convex-clustering. Accessed: 2022-06-10.

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

[4] M. P. I. Forum, *MPI: A Message-passing Interface Standard, Version 3.1 ; June 4, 2015*. High-Performance Computing Center Stuttgart, University of Stuttgart, 2015.

[5] L. F., E. Tejedor, J. Ejarque, R. Rafanell, J. Álvarez, F. Marozzo, D. Lezzi, R. Sirvent, D. Talia, and R. M. Badia, "Servicess: an interoperable programming framework for the cloud," *Journal of Grid Computing*, vol. 12, p. 6791, March 2014.

[6] S. Boyd and L. Vandenberghe, *Convex Optimization*. Cambridge University Press, 2004.

[7] T. Yang, X. Yi, J. Wu, Y. Yuan, D. Wu, Z. Meng, Y. Hong, H. Wang, Z. Lin, and K. H. Johansson, "A survey of distributed optimization," *Annual Reviews in Control*, vol. 47, pp. 278–305, 2019.

[8] A. Nedic and A. Ozdaglar, "Distributed subgradient methods for multi-agent optimization," *IEEE Transactions on Automatic Control*, vol. 54, no. 1, pp. 48–61, 2009.

[9] S. S. Ram, A. Nedich, and V. V. Veeravalli, "Distributed stochastic subgradient projection algorithms for convex optimization," *Journal of Optimization Theory and Applications*, vol. 147, pp. 516–545, Dec 2010.

[10] D. Jakovetic, J. M. F. Xavier, and J. M. F. Moura, "Fast distributed gradient methods," *CoRR*, vol. abs/1112.2972, 2011.

[11] A. Mokhtari, Q. Ling, and A. Ribeiro, "Network Newton Distributed Optimization Methods," *IEEE Transactions on Signal Processing*, vol. 65, no. 1, pp. 146–161, 2017.

[12] D. Bajović, D. Jakovetić, N. Krejić, and N. K. Jerinkić, "Newton-like method with diagonal correction for distributed optimization," *SIAM Journal on Optimization*, vol. 27, no. 2, pp. 1171–1203, 2017.

[13] H.-U. JB., S. JJ., and N. V.H, "Generalized hessian matrix and second-order optimality conditions for problems withc 1,1 data," *Applied Mathematics and Optimization*, vol. 11, pp. 43–56, 1984.

[14] G. Hager and G. Wellein, *Introduction to high performance computing for scientists and engineers*. CRC Press, 2010.

[15] P. Pacheco, *An Introduction to Parallel Programming*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1st ed., 2011.

[16] L. Dagum and R. Menon, "Openmp: An industry-standard api for shared-memory programming," *IEEE Comput. Sci. Eng.*, vol. 5, p. 4655, Jan. 1998.

[17] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall, "Open mpi: Goals, concept, and design of a next generation mpi implementation," in *In Proceedings, 11th European PVM/MPI Users Group Meeting*, pp. 97–104, 2004.

[18] *MPICH*, accessed on: July 20, 2020.

[19] S. Cook, *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1st ed., 2012.

[20] J. Conejero, S. Corella, R. M. Badia, and J. Labarta, "Task-based programming in compss to converge from hpc to big data," *The International Journal of High Performance Computing Applications*, vol. 32, pp. 45 – 60, 2018.

[21] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica, "Apache spark: A unified engine for big data processing," *Commun. ACM*, vol. 59, p. 5665, oct 2016.

[22] S. Ekanayake, S. Kamburugamuve, P. Wickramasinghe, and G. C. Fox, "Java thread and process performance for parallel machine learning on multicore hpc clusters," in *2016 IEEE International Conference on Big Data (Big Data)*, pp. 347–354, 2016.

[23] K. Zhang, Z. Yang, H. Liu, T. Zhang, and T. Basar, "Fully decentralized multi-agent reinforcement learning with networked agents," *CoRR*, vol. abs/1802.08757, 2018.

[24] J. Shamma, *Cooperative Control of Distributed Multi-Agent Systems*. USA: Wiley-Interscience, 2008.

[25] A. Salkham, R. Cunningham, A. Garg, and V. Cahill, "A collaborative reinforcement learning approach to urban traffic control optimization," in *2008 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology*, vol. 2, pp. 560–566, 2008.

[26] R. Roche, B. Blunier, A. Miraoui, V. Hilaire, and A. Koukam, "Multi-agent systems for grid energy management: A short review," in *IECON 2010 - 36th Annual Conference on IEEE Industrial Electronics Society*, pp. 3341–3346, 2010.

[27] T. Warren Liao, "Clustering of time series data - a survey," *Pattern Recognition*, vol. 38, no. 11, pp. 1857–1874, 2005.

[28] X. Dai and T. Kuosmanen, "Best-practice benchmarking using clustering methods: Application to energy regulation," *Omega*, vol. 42, no. 1, pp. 179–188, 2014.

[29] T. Chaira, "A novel intuitionistic fuzzy c means clustering algorithm and its application to medical images," *Applied Soft Computing*, vol. 11, no. 2, pp. 1711–1717, 2011.

[30] F. Lindsten, H. Ohlsson, and L. Ljung, "Clustering using sum-of-norms regularization: With application to particle filter output computation," in *2011 IEEE Statistical Signal Processing Workshop (SSP)*, pp. 201–204, 2011.

[31] S. Lloyd, "Least squares quantization in pcm," *IEEE Transactions on Information Theory*, vol. 28, pp. 129–137, March 1982.

[32] D. Arthur and S. Vassilvitskii, "K-means++: The advantages of careful seeding," in *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '07, (USA), p. 10271035, Society for Industrial and Applied Mathematics, 2007.

[33] S. Arora, P. Raghavan, and S. Rao, "Approximation schemes for euclidean k-medians and related problems," in *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*, STOC '98, (New York, NY, USA), p. 106113, Association for Computing Machinery, 1998.

[34] A. Banerjee, S. Merugu, I. S. Dhillon, and J. Ghosh, "Clustering with bregman divergences," *J. Mach. Learn. Res.*, vol. 6, p. 17051749, dec 2005.

[35] F. Lindsten, H. Ohlsson, and L. Ljung, "Just relax and come clustering! : A convexification of k-means clustering," in *Technical report, Department of Electrical Engineering, Linkopings Universitet*, 2011.

[36] E. C. Chi and K. Lange, "Splitting methods for convex clustering," *Journal of Computational and Graphical Statistics*, vol. 24, no. 4, pp. 994–1013, 2015. PMID: 27087770.

[37] K. Pelckmans, J. D. Brabanter, B. D. Moor, and J. A. K. Suykens, "Convex clustering shrinkage," in *Workshop on Statistics and Optimization of Clustering Workshop (PASCAL)*, 2005.

[38] T. D. Hocking, A. Joulin, F. Bach, and J.-P. Vert, "Clusterpath: An algorithm for clustering using convex fusion penalties," in *Proceedings of the 28th International Conference on International Conference on Machine Learning*, ICML'11, (Madison, WI, USA), p. 745752, Omnipress, 2011.

[39] N. K. Jerinkić, D. Jakovetić, N. Krejić, and D. Bajović, "Distributed second-order methods with increasing number of working nodes," *IEEE Transactions on Automatic Control*, vol. 65, no. 2, pp. 846–853, 2020.

[40] D. Jakoveti, D. Bajovi, N. Kreji, and N. Krklec Jerinki, "Distributed gradient methods with variable number of working nodes," *IEEE Transactions on Signal Processing*, vol. 64, no. 15, pp. 4080–4095, 2016.

[41] D. Jakovetić, D. Bajović, A. K. Sahu, and S. Kar, "Convergence rates for distributed stochastic optimization over random networks," in *2018 IEEE Conference on Decision and Control (CDC)*, (Miami Beach, FL, USA), pp. 4238–4245, 2018.

[42] A. Sahu, D. Jakovetić, D. Bajović, and S. Kar, "Distributed zeroth order optimization over random networks: A kiefer-wolfowitz stochastic approximation approach," in *2018 IEEE Conference on Decision and Control (CDC)*, (Miami Beach, FL, USA), pp. 4951–4958, 03 2018.

[43] A. K. Sahu, D. Jakovetic, D. Bajovic, and S. Kar, "Communication-efficient distributed strongly convex stochastic optimization: Non-asymptotic rates," 2018.

[44] A. K. Sahu, D. Jakovetic, D. Bajovic, and S. Kar, "Communication efficient distributed estimation over directed random graphs," in *IEEE EUROCON 2019 -18th International Conference on Smart Technologies*, (Novi Sad, Serbia), pp. 1–5, 2019.

[45] S. Boyd, A. Ghosh, B. Prabhakar, and D. Shah, "Randomized gossip algorithms," *IEEE Transactions on Information Theory*, vol. 52, pp. 2508–2530, June 2006.

[46] L. Fodor, D. Jakovetic, N. Krejic, N. K. Jerinkic, and S. Skrbic, "Performance evaluation and analysis of distributed multi-agent optimization algorithms with sparsified directed communication," *EURASIP J. Adv. Signal Process*, vol. 25, 2021.

[47] "I-bidaas project, funded by the european commission under grant agreement no.780787." https://www.ibidaas.eu/. Accessed: 2022-06-15.

[48] M. Bancheri, F. Fusco, D. D. Torre, F. Terribile, P. Manna, G. Langella, P. De Vita, V. Allocca, H. Loishandl-Weisz, T. Hermann, C. De Michele, A. Coppola, F. A. Mileti, and A. Basile, "The pesticide fate tool for groundwater vulnerability assessment within the geospatial decision support system landsupport," *Science of The Total Environment*, vol. 807, p. 150793, 2022.

[49] R. M. de Pozuelo Genis and M. M. Marcos, "I-BiDaaS - CAIXA - Bank Transfer - Tokenised Dataset," Oct. 2020.

[50] J. Mascolo, G. Genchi, and G. D. Spennacchio, "I-BiDaaS - CRF - Aluminium die-casting Synthetic Dataset," Dec. 2020.

[51] "Distributed computing library," accessed on June 1, 2022.

[52] E. P. Xing, Q. Ho, P. Xie, and D. Wei, "Strategies and principles of distributed machine learning on big data," *Engineering*, vol. 2, no. 2, pp. 179–195, 2016.

[53] M. Li, D. G. Andersen, A. J. Smola, and K. Yu, "Communication efficient distributed machine learning with the parameter server," in *Advances in Neural Information Processing Systems* (Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K. Weinberger, eds.), vol. 27, Curran Associates, Inc., 2014.

[54] K. Sakurama, "Leader selection via lasso for formation control of time-delayed multi-agent systems," *Neurocomputing*, vol. 270, pp. 18 – 26, 2017. Distributed Control and Optimization with Resource-Constrained Networked Systems.

[55] E. Dall'Anese, H. Zhu, and G. Giannakis, "Distributed optimal power flow for smart microgrids," *IEEE Transactions on Smart Grid*, vol. 4, 11 2012.

[56] I. D. Schizas, A. Ribeiro, and G. B. Giannakis, "Consensus in ad hoc wsns with noisy linkspart i: Distributed estimation of deterministic signals," *IEEE Transactions on Signal Processing*, vol. 56, no. 1, pp. 350–364, 2008.

[57] Z. Quan, S. Cui, A. H. Sayed, and H. V. Poor, "Optimal multiband joint detection for spectrum sensing in cognitive radio networks," *IEEE Transactions on Signal Processing*, vol. 57, no. 3, pp. 1128–1140, 2009.

[58] K. Yuan, Q. Ling, and W. Yin, "On the convergence of decentralized gradient descent," *SIAM Journal on Optimization*, vol. 26, no. 3, p. 18351854, 2016.

[59] M. Schmidt, N. L. Roux, and F. Bach, "Convergence rates of inexact proximal-gradient methods for convex optimization," in *Advances in Neural Information Processing Systems (NIPS'11)*, pp. 1458–1466, Dec 2011.

[60] L. Xiao, "Dual averaging method for regularized stochastic learning and online optimization," in *Advances in Neural Information Processing Systems 22* (Y. Bengio, D. Schuurmans, J. D. Lafferty, C. K. I. Williams, and A. Culotta, eds.), pp. 2116–2124, Curran Associates, Inc., 2009.

[61] J. C. Duchi, A. Agarwal, and M. J. Wainwright, "Dual averaging for distributed optimization: Convergence analysis and network scaling," *IEEE Transactions on Automatic Control*, vol. 57, no. 3, pp. 592–606, 2012.

[62] K. I. Tsianos and M. G. Rabbat, "Distributed dual averaging for convex optimization under communication delays," in *2012 American Control Conference (ACC)*, pp. 1067–1072, 2012.

[63] K. I. Tsianos, S. Lawlor, and M. G. Rabbat, "Push-sum distributed dual averaging for convex optimization," in *2012 IEEE 51st IEEE Conference on Decision and Control (CDC)*, pp. 5453–5458, 2012.

[64] D. Jakoveti, D. Bajovi, J. Xavier, and J. M. F. Moura, "Primaldual methods for large-scale and distributed convex optimization and data analytics," *Proceedings of the IEEE*, vol. 108, no. 11, pp. 1923–1938, 2020.

[65] T. Suzuki, "Dual averaging and proximal gradient descent for online alternating direction multiplier method," in *Proceedings of Machine Learning Research* (S. Dasgupta and D. McAllester, eds.), vol. 28, (Atlanta, Georgia, USA), pp. 392–400, PMLR, 17–19 Jun 2013.

[66] B. Wahlberg, S. Boyd, M. Annergren, and Y. Wang, "An admm algorithm for a class of total variation regularized estimation problems*," *IFAC Proceedings Volumes*, vol. 45, no. 16, pp. 83 – 88, 2012. 16th IFAC Symposium on System Identification.

[67] L. Majzoobi, F. Lahouti, and V. Shah-Mansouri, "Analysis of distributed admm algorithm for consensus optimization in presence of node error," *IEEE Transactions on Signal Processing*, vol. 67, no. 7, pp. 1774–1784, 2019.

[68] J. Yan, F. Guo, C. Wen, and G. Li, "Parallel alternating direction method of multipliers," *Information Sciences*, vol. 507, pp. 185 – 196, 2020.

[69] R. D.P. and T. R., "A flexible admm algorithm for big data applications," *Journal of Scientific Computing*, vol. 71, p. 435467, 2017.

[70] H. Yue, Q. Yang, X. Wang, and X. Yuan, "Implementing the alternating direction method of multipliers for big datasets: A case study of least absolute shrinkage and selection operator," *SIAM J. Sci. Comput.*, vol. 40, no. 5, pp. A3121–A3156, 2018.

[71] R. H. Byrd, S. L. Hansen, J. Nocedal, and Y. Singer, "A stochastic Quasi-Newton method for Large-Scale Optimization," *SIAM Journal on Optimization*, vol. 26, no. 2, pp. 1008–1031, 2016.

[72] R. Johnson and T. Zhang, "Accelerating stochastic gradient descent using predictive variance reduction," in *NIPS*, 2013.

[73] P. Moritz, R. Nishihara, and M. I. Jordan, "A linearly-convergent stochastic l-bfgs algorithm," in *AISTATS*, 2016.

[74] K. Yang, T. Fan, T. Chen, Y. Shi, and Q. Yang, "A quasi-newton method based vertical federated learning framework for logistic regression," *ArXiv*, vol. abs/1912.00513, 2019.

[75] I. A. Chen and A. Ozdaglar, "A fast distributed proximal-gradient method," in *2012 50th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, (Monticello, IL, USA), pp. 601–608, 2012.

[76] I. Notarnicola and G. Notarstefano, "Randomized dual proximal gradient for large-scale distributed optimization," in *2015 54th IEEE Conference on Decision and Control (CDC)*, pp. 712–717, 2015.

[77] D. Jakoveti, J. Xavier, and J. M. F. Moura, "Distributed nesterov gradient methods for random networks: Convergence in probability and convergence rates," in

2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), pp. 1508–1511, 2014.

[78] B. Johansson, M. Rabi, and M. Johansson, "A Randomized Incremental Subgradient Method for Distributed Optimization in Networked Systems," *SIAM Journal on Optimization*, vol. 20, pp. 1157–1170, Jan 2009.

[79] F. Farina and G. Notarstefano, "A randomized block subgradient approach to distributed big data optimization," in *2019 IEEE 58th Conference on Decision and Control (CDC)*, pp. 6362–6367, 2019.

[80] Y. KAJIYAMA, N. HAYASHI, and S. TAKAI, "Distributed constrained convex optimization with accumulated subgradient information over undirected switching networks," *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, vol. E102.A, no. 2, pp. 343–350, 2019.

[81] J. Li, G. Chen, Z. Wu, and X. He, "Distributed subgradient method for multi-agent optimization with quantized communication," *Mathematical Methods in the Applied Sciences*, 06 2016.

[82] H. Iiduka, "Incremental subgradient method for nonsmooth convex optimization with fixed point constraints," *Optimization Methods and Software*, vol. 31, no. 5, pp. 931–951, 2016.

[83] A. Nedi, A. Olshevsky, and M. G. Rabbat, "Network topology and communication-computation tradeoffs in decentralized optimization," *Proceedings of the IEEE*, vol. 106, no. 5, pp. 953–976, 2018.

[84] N. Gaeini, A. Amani, M. Jalili, and X. Yu, "Optimization of communication network topology in distributed control systems subject to prescribed decay rate," *IEEE Transactions on Cybernetics*, vol. PP, pp. 1–9, 07 2019.

[85] G. Neglia, G. Calbi, D. Towsley, and G. Vardoyan, "The role of network topology for distributed machine learning," in *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*, pp. 2350–2358, 2019.

[86] T. C. Aysal, M. E. Yildiz, A. D. Sarwate, and A. Scaglione, "Broadcast gossip algorithms for consensus," *IEEE Transactions on Signal Processing*, vol. 57, no. 7, pp. 2748–2761, 2009.

[87] D. Jakovetic, J. Xavier, and J. M. F. Moura, "Cooperative convex optimization in networked systems: Augmented lagrangian algorithms with directed gossip communication," *IEEE Transactions on Signal Processing*, vol. 59, no. 8, pp. 3889–3902, 2011.

[88] A. Agarwal and J. C. Duchi, "Distributed delayed stochastic optimization," in *Advances in Neural Information Processing Systems 24* (J. Shawe-Taylor, R. S. Zemel, P. L. Bartlett, F. Pereira, and K. Q. Weinberger, eds.), pp. 873–881, Curran Associates, Inc., 2011.

[89] J. Cortes, S. Martinez, T. Karatas, and F. Bullo, "Coverage control for mobile sensing networks," *IEEE Transactions on Robotics and Automation*, vol. 20, no. 2, pp. 243–255, 2004.

[90] I. Lobel, A. Ozdaglar, and D. Feijer, "Distributed multi-agent optimization with state-dependent communication," *Mathematical Programming*, vol. 129, p. 255284, Jun 2011.

[91] J. Baxter and P. L. Bartlett, "Direct gradient-based reinforcement learning," in *2000 IEEE International Symposium on Circuits and Systems (ISCAS)*, vol. 3, pp. 271–274 vol.3, 2000.

[92] I. Matei and J. S. Baras, "Performance evaluation of the consensus-based distributed subgradient method under random communication topologies," *IEEE Journal of Selected Topics in Signal Processing*, vol. 5, no. 4, pp. 754–771, 2011.

[93] M. Assran and M. Rabbat, "Asynchronous subgradient-push," *CoRR*, vol. abs/1803.08950, 2018.

[94] M. Assran and M. Rabbat, "An empirical comparison of multi-agent optimization algorithms," in *2017 IEEE Global Conference on Signal and Information Processing (GlobalSIP)*, pp. 573–577, 2017.

[95] J. Zhang and K. You, "Asyspa: An exact asynchronous algorithm for convex optimization over digraphs," *CoRR*, vol. abs/1808.04118, 2018.

[96] K. I. Tsianos, S. F. Lawlor, and M. G. Rabbat, "Communication/computation tradeoffs in consensus-based distributed optimization," *CoRR*, vol. abs/1209.1076, 2012.

[97] K. I. Tsianos, S. Lawlor, and M. G. Rabbat, "Consensus-based distributed optimization: Practical issues and applications in large-scale machine learning," in *2012 50th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, pp. 1543–1550, 2012.

[98] L. Xiao, A. W. Yu, Q. Lin, and W. Chen, "Dscovr: Randomized primal-dual block coordinate algorithms for asynchronous distributed optimization," *Journal of Machine Learning Research*, vol. 20, no. 43, pp. 1–58, 2019.

[99] H. Steinhaus, "Sur la division des corps matériels en parties," *Bull. Acad. Pol. Sci., Cl. III*, vol. 4, pp. 801–804, 1957.

[100] J. Peña, J. Lozano, and P. Larrañaga, "An empirical comparison of four initialization methods for the k-means algorithm," *Pattern Recognition Letters*, vol. 20, pp. 1027–1040, October 1999.

[101] S. S. Khan and A. Ahmad, "Cluster center initialization algorithm for k-means clustering," *Pattern Recognition Letters*, vol. 25, no. 11, pp. 1293–1302, 2004.

[102] D. Lashkari and P. Golland, "Convex clustering with exemplar-based models," *Advances in neural information processing systems*, vol. 20, pp. 825–832, 01 2007.

[103] S. Nowozin and G. Bakir, "A decoupled approach to exemplar-based unsupervised learning," in *Proceedings of the 25th International Conference on Machine Learning*, ICML '08, (New York, NY, USA), p. 704711, Association for Computing Machinery, 2008.

[104] M. Wang, T. Yao, and G. I. Allen, "Supervised convex clustering," *arXiv preprint arXiv:2005.12198*, 2020.

[105] C. GK, C. EC, R. JM, and L. K, "Convex clustering: an attractive alternative to hierarchical clustering," *PLoS Comput Biol*, vol. 11, May 2015. PMID: 25965340; PMCID: PMC4429070.

[106] B. Wang, Y. Zhang, W. Sun, and Y. Fang, "Sparse convex clustering," *Journal of Computational and Graphical Statistics*, vol. 27, pp. 393–403, April 2018. Publisher Copyright: © 2018, © 2018 American Statistical Association, Institute of Mathematical Statistics, and Interface Foundation of North America.

[107] M. Yuan and Y. Lin, "Model selection and estimation in regression with grouped variables," *Journal of the Royal Statistical Society Series B*, vol. 68, pp. 49–67, 02 2006.

[108] Q. Qian, *On Algorithmic Regularization And Convex Clustering*. PhD thesis, SOhio State University, 2019. OhioLINK Electronic Theses and Dissertations Center.

[109] Y. L. Huangyue Chen, Lingchen Kong, "A novel convex clustering method for high-dimensional data using semiproximal admm," *Mathematical Problems in Engineering*, vol. 2020, 2020. Article ID 9216351, 12 pages.

[110] S. Kar and B. Swenson, "Clustering with distributed data," 2019.

[111] W. Zhou, H. Yi, G. Mishne, and E. Chi, "Scalable algorithms for convex clustering," in *2021 IEEE Data Science and Learning Workshop (DSLW)*, pp. 1–6, 2021.

[112] X. Zhou, C. Du, and X. Cai, "An efficient smoothing proximal gradient algorithm for convex clustering," 2020.

[113] C. Zhu, H. Xu, C. Leng, and S. Yan, "Convex optimization procedure for clustering: Theoretical revisit," in *Advances in Neural Information Processing Systems* (Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K. Weinberger, eds.), vol. 27, (New York, USA), pp. 1619–1627, Curran Associates, Inc., 2014.

[114] A. Panahi, D. Dubhashi, F. D. Johansson, and C. Bhattacharyya, "Clustering by sum of norms: Stochastic incremental algorithm, convergence and cluster recovery," in *Proceedings of the 34th International Conference on Machine Learning* (D. Precup and Y. W. Teh, eds.), vol. 70 of *Proceedings of Machine Learning Research*, (Sydney, Australia), pp. 2769–2777, PMLR, 06–11 Aug 2017.

[115] D. Sun, K. chuan Toh, and Y. Yuan, "Convex clustering: Model, theoretical guarantee and efficient algorithm," *J. Mach. Learn. Res.*, vol. 22, pp. 9:1–9:32, 2021.

[116] E. Tejedor, Y. Becerra, G. Alomar, A. Queralt, R. M. Badia, J. Torres, T. Cortes, and J. Labarta, "Pycompss: Parallel computational workflows in python," *The International Journal of High Performance Computing Applications*, vol. 31, no. 1, pp. 66–82, 2017.

[117] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu, "A density-based algorithm for discovering clusters in large spatial databases with noise," in *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, KDD'96, (Palo Alto, California), p. 226231, AAAI Press, 1996.

[118] D. Jakoveti, J. M. F. Moura, and J. Xavier, "Distributed nesterov-like gradient algorithms," in *2012 IEEE 51st IEEE Conference on Decision and Control (CDC)*, pp. 5459–5464, 2012.

[119] S. Sundhar Ram, A. Nedi, and V. V. Veeravalli, "Distributed stochastic subgradient projection algorithms for convex optimization," *Journal of Optimization Theory and Applications*, vol. 147, p. 516545, Jul 2010.

[120] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK Users' Guide*, vol. 9. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics (SIAM), 3 ed., 1999.

[121] L. B. et al., "An Updated Set of Basic Linear Algebra Subprograms (BLAS)," *ACM Trans. Math. Softw.*, vol. 28, p. 135151, June 2002.

[122] *ADMM l1 and l2 logistic regression*, accessed on: May 15, 2020.

[123] E. F. Tjong Kim Sang and F. De Meulder, "Language-independent named entity recognition ii," 2005; accessed on: May 30, 2019.

[124] E. F. Tjong Kim Sang and F. De Meulder, "Introduction to the conll-2003 shared task: Language-independent named entity recognition," in *Proceedings of the Seventh Conference on Natural Language Learning at HLT-NAACL 2003 - Volume 4*, CONLL 03, (USA), p. 142147, Association for Computational Linguistics, 2003.

[125] UCI Machine Learning Repository, "Gisette data set," 2008; accessed on: May 29, 2019.

[126] D. Dua and C. Graff, "UCI machine learning repository," 2017. University of California, Irvine, School of Information and Computer Sciences.

[127] I. Guyon, S. Gunn, A. Ben-Hur, and G. Dror, "Result analysis of the nips 2003 feature selection challenge," in *Proceedings of the 17th International Conference on Neural Information Processing Systems*, vol. 17 of *NIPS04*, (Cambridge, MA, USA), p. 545552, MIT Press, 01 2004.

[128] "Uci machine learning repository." hhttp://archive.ics.uci.edu/ml. University of California, Irvine, School of Information and Computer Sciences, 2017, Accessed: 2022-03-20.

[129] UCI Machine Learning Repository, "Yearpredictionmsd data set," 2011; accessed on: September 01, 2019.

[130] T. Bertin-Mahieux, D. P. Ellis, B. Whitman, and P. Lamere, "The million song dataset," in *Proceedings of the 12th International Conference on Music Information Retrieval (ISMIR 2011)*, 2011.

[131] Y. LeCun and C. Cortes, "The mnist database of handwritten digits," 2005; accessed on: September 01, 2019.

[132] L. Deng, "The mnist database of handwritten digit images for machine learning research [best of the web]," *IEEE Signal Processing Magazine*, vol. 29, pp. 141–142, 2012.

[133] UCI Machine Learning Repository, "Relative location of ct slices on axial axis data set," 2011; accessed on: September 08, 2019.

[134] F. Graf, H.-P. Kriegel, M. Schubert, S. Pölsterl, and A. Cavallaro, "2d image registration in ct images using radial image descriptors," in *International Conference on Medical Image Computing and Computer-Assisted Intervention*, vol. 6892, pp. 607–614, Springer, Springer, Berlin, Heidelberg, 2011.

[135] UCI Machine Learning Repository, "p53 mutants data set," 2010; accessed on: September 03, 2019.

[136] S. Danziger, R. Baronio, L. Ho, L. Hall, K. Salmon, G. Hatfield, P. Kaiser, and R. Lathrop, "Predicting positive p53 cancer rescue regions using most informative positive (mip) active learning," *PLoS computational biology*, vol. 5, p. e1000498, 09 2009.

[137] S. A. Danziger, J. Zeng, Y. Wang, R. K. Brachmann, and R. H. Lathrop, "Choosing where to look next in a mutation sequence space: Active Learning of informative p53 cancer rescue mutants," *Bioinformatics*, vol. 23, pp. 104–114, 07 2007.

[138] S. Danziger, S. J. Swamidass, J. Zeng, L. Dearth, Q. Lu, J. Chen, J. Cheng, V. Hoang, H. Saigo, R. Luo, P. Baldi, R. Brachmann, and R. Lathrop, "Functional census of mutation sequence spaces: The example of p53 cancer rescue mutants," *IEEE/ACM transactions on computational biology and bioinformatics / IEEE, ACM*, vol. 3, pp. 114–25, 05 2006.

[139] V. Simic, B. Stojanovic, and M. Ivanovic, "Optimizing the performance of optimization in the cloud environmentan intelligent auto-scaling approach," *Future Generation Computer Systems*, vol. 101, pp. 909–920, 2019.

[140] R. E. Bryant and D. R. O'Hallaron, *Computer systems.* Prentice Hall, 2011.

[141] E. D. Dolan and J. J. Moré, "Benchmarking optimization software with performance profiles," *Mathematical Programming*, vol. 91, p. 201213, Jan 2002.

[142] R. Tibshirani, M. Saunders, S. Rosset, J. Zhu, and K. Knight, "Sparsity and smoothness via the fused lasso," *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, vol. 67, no. 1, pp. 91–108, 2005.

[143] S. Diamond and S. Boyd, "CVXPY: A Python-embedded modeling language for convex optimization," *Journal of Machine Learning Research*, vol. 17, no. 83, pp. 1–5, 2016.

[144] A. Agrawal, R. Verschueren, S. Diamond, and S. Boyd, "A rewriting system for convex optimization problems," *Journal of Control and Decision*, vol. 5, no. 1, pp. 42–60, 2018.

[145] Y. Gdalyahu, D. Weinshall, and M. Werman, "A randomized algorithm for pairwise clustering," in *NIPS*, 1998.

[146] "scikit-learn: Machine learning in python." https://scikit-learn.org/stable/. Accessed: 2022-03-20.

[147] C. Rasmussen, "The infinite gaussian mixture model," in *Advances in Neural Information Processing Systems* (S. Solla, T. Leen, and K. Müller, eds.), vol. 12, (Cambridge, Massachusetts, USA), MIT Press, 1999.

[148] L. van der Maaten and G. Hinton, "Visualizing data using t-SNE," *Journal of Machine Learning Research*, vol. 9, pp. 2579–2605, 2008.

[149] E. Anderson, "The species problem in iris," *Annals of the Missouri Botanical Garden*, vol. 23, no. 3, pp. 457–509, 1936.

[150] R. A. Fisher, "The use of multiple measurements in taxonomic problems," *Annals of Human Genetics*, vol. 7, pp. 179–188, 1936.

[151] J. C. Dunn, "Well-separated clusters and optimal fuzzy partitions," *Journal of Cybernetics*, vol. 4, no. 1, pp. 95–104, 1973.

[152] D. L. Davies and D. W. Bouldin, "A cluster separation measure," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. PAMI-1, no. 2, pp. 224–227, 1979.

[153] "The comprehensive r archive network." https://cran.r-project.org/. Accessed: 2022-06-20.

[154] W. Xiao and J. Hu, "A survey of parallel clustering algorithms based on spark," *Scientific Programming*, vol. 2020, pp. 1–12, 09 2020. Article ID 8884926, 12 pages.

[155] E. R. Sparks, A. Talwalkar, V. Smith, J. Kottalam, X. Pan, J. Gonzalez, M. J. Franklin, M. I. Jordan, and T. Kraska, "Mli: An api for distributed machine learning," in *2013 IEEE 13th International Conference on Data Mining*, pp. 1187–1192, IEEE, October 2013.

[156] "I-bidaas - d3.3: Batch processing analytics module implementation final report," 2021; accessed on June 1, 2022.

[157] "The knowledge repository of the i-bidaas project." https://github.com/ibidaas/knowledge_repository. Accessed: 2022-06-15.

[158] MATLAB, *version 7.10.0 (R2010a)*. Natick, Massachusetts: The MathWorks Inc., 2010.

[159] B. R. M., J. Conejero, C. Diaz, J. Ejarque, D. Lezzi, F. Lordan, C. Ramon-Cortes, and R. Sirvent, "Comp superscalar, an interoperable programming framework," *SoftwareX*, vol. 34, p. 3236, December 2015.

[160] G. Mateos, J. A. Bazerque, and G. B. Giannakis, "Distributed sparse linear regression," *IEEE Transactions on Signal Processing*, vol. 58, no. 10, pp. 5262–5276, 2010.

[161] T. Gowda and C. A. Mattmann, "Clustering web pages based on structure and style similarity (application paper)," in *2016 IEEE 17th International Conference on Information Reuse and Integration (IRI)*, pp. 175–180, 2016.

[162] "H2020 i-bidaas project - industrial-driven big data as a self-service solution," 2021; accessed on June 1, 2022.

[163] A. Alexopoulos, Y. Becerra, O. Boehm, G. Bravos, V. Chatzigiannakis, C. Cugnasco, G. Demetriou, I. Eleftheriou, L. Fodor, S. Fotis, S. Ioannidis, D. Jakovetic, L. Kallipolitis, V. Katusic, E. Kavakli, D. Kopanaki, C. Leventis, M. M. Marcos, R. M. de Pozuelo, M. Martínez, N. Milosevic, E. P. P. Montanera, G. Ristow, H. Ruiz-Ocampo, R. Sakellariou, R. Sirvent, S. Skrbic, I. Spais, G. Vasiliadis, and M. Vinov, *Big Data Analytics in the Banking Sector: Guidelines and Lessons Learned from the CaixaBank Case*, pp. 273–297. Cham: Springer International Publishing, 2022.

Appendix A

Prošireni izvod

Uvod

Distribuirana konveksna optimizacija [3, 6, 7] predstavlja pristup, koji omogućuje particionisanje konveksnog optimizacionog problema u skup povezanih podsistema. Primenom skupa agenata za rešavanje ovakvih problema, pojavljuju se multi-agent distribuirane metode konveksne optimizacije, koje omogućuju rešavanje širokog skupa problema [8, 9, 10, 11, 12, 13]. Metode distribuirane konveksne optimizacije, mogu se podeliti na više načina. U ovoj tezi razmatramo dve klase metoda: stohastičke metode primarnog tipa, bez ograničenja i determinističke metode dualnog tipa sa ograničenjem (posmatrani problem bez ograničenja prevodimo u metodu sa ograničenjem). Iz aspekta implementacije, posmatraju se dva modela komunikacije: manager-workers princip, koji u okviru teze koristimo sa dualnom ADMM [3] metodom, i princip mreže ravnopravno povezanih čvorova, koji koristimo sa metodama primarnog tipa u tezi.

Primenom računarstva visokih performansi, problemi iz oblasti distribuirane konveksne optimizacije se mogu rešavati na efikasan način. Unutar teze, koristimo dva različita pristupa za paralelizaciju: Message Passing Interface (MPI) [4], i COMPS Superscalar (COMPSs) [5]. Primenom ovih tehnologija, razvijamo paralelne implementacije metoda, nad kojima sprovodimo sveobuhvatnu empirijsku evaluaciju na računarskom klasteru. Rezultati ovih analiza omogućuju uvid u osobine posmatranih metoda, njihovo međusobno poređenje, kao i detekciju daljih pravaca razvoja. Metode primarnog tipa, u prvom delu teze, razvijamo u MPI tehnologiji, u programskom jeziku C, dok u drugom delu teze razvijamo algoritam za konveksno klasterovanje, baziran na ADMM-u, u COMPSs tehnologiji. Pokazaćemo osobine skalabilnosti razvijenih algoritama, uz zaključke o njihovoj primenjivosti u raznim scenarijima. Osim toga, teza sadrži i poređenje pomenute dve tehnologije.

Metode distribuirane optimizacije primarnog tipa

U prvom delu teze, u težištu pažnje su metode distribuirane optimizacije primarnog tipa, bez ograničenja. Akcenat je na razvoju što efikasnije implementacije ovih metoda i njihovoj praktičnoj evaluaciji na računarskom klasteru.

Teorijske osobine metoda

Posmatramo mrežu povezanih čvorova (agenata), gde svaki čvor u mreži poseduje konveksnu funkciju cilja $f_i : \mathbb{R}^p \to \mathbb{R}$, gde je funkcija f_i poznata samo čvoru i, i svaka funkcija f_i je strogo konveksna, dvaput diferencijabilna, sa Lipschitz kontinuiranim gradijentom. Svaki čvor u mreži teži rešavanju sledećeg problema:

$$minimize \quad f(x) := \sum_{i=1}^{n} f_i(x). \tag{A.1}$$

Problemu (A.1), pridružujemo graf $G = (N, E)$, gde $N = \{1, ..., n\}$ predstavlja skup čvorova, a E predstavlja skup grana $\{i, j\}$, odnosno parove čvorova, koji mogu direktno komunicirati. Drugim rečima, graf G predstavlja kolekciju komunikacionih veza između čvorova. Algoritmi koje ovde razmatramo mogu da koriste sve ove veze (slučaj kada ne koristimo sparsifikaciju komunikacije), ili mogu da koriste samo podskup ovih veza tokom iteracija (slučajevi primene sparsfikovanja komunikacije). Opisani problem se može rešavati iterativno, na sledeći način:

$$x_i^{k+1} = x_i^k - d_i^k, \tag{A.2}$$

$$d_i^k = -\Big[(M_i^k)^{=1}[\alpha \nabla f_i(x_i^k) + \sum_{j \epsilon \Omega_i} W_{ij}(x_i^k - x_j^k)\xi_{i,j}^k\Big]. \tag{A.3}$$

U ovoj formulaciji, $x_i \in \mathbb{R}^p$, predstavlja optimizacionu varijablu, a $(M_i^k)^{-1}$ je aproksimacija vrednosti inverza Hessian-a, tj. informacije drugog reda. Ω_i je skup suseda čvora i, a matrica W je dvostruko stohastička matrica susedstva, koja prati strukturu grafa G, tako da je $W_{ij} = 0$ za $i \neq j$, ako i samo ako je $\{i, j\} \notin E$. α predstavlja pozitivnu vrednost, koja ima ulogu dužine koraka.

Parametar ξ_{ij}^k enkodira informaciju o sparsifikaciji komunikacije. Označimo sa z_i^k slučajne promenljive, koje su nezavisne na nivou čvorova i iteracija. Tada je verovatnoća data sa

$p_k = Prob(z_i^k = 1)$, jednaka na nivou svih čvorova. Vrednost p_k predstavlja dizajn parametar metoda, i posmatramo različite pristupe za podešavanje njegove vrednosti, što će biti opisano u nastavku. $\xi_{i,j}^k$ je funkcija z_i^k i z_j^k i smatramo da može imati sledeće vrednosti: $\xi_{i,j}^k = 1$, kada nema sparsifikacije komunikacije; $\xi_{i,j}^k = z_i^k \cdot z_j^k$, u slučaju dvosmerne sparsifikacije komunikacije; i $\xi_{i,j}^k = z_j^k$, u slučaju jednosmerne sparsifikacije komunikacije. Dvosmerna sparsifikacija komunikacije podrazumeva da čvor i uključuje procenu rešenja susednog čvora j, samo ako su oba čvora i i j aktivna za komunikaciju. Drugim rečima, ukoliko je čvor i aktivan u iteraciji k, on šalje svoju porcenu rešenja svim svojim aktivnim susedima i takođe prima lokalne procene rešenja od njih. S druge strane, jednosmerna sparsifikacija komunikacije podrazumeva da čvor i uključuje procenu rešenja svog suseda j, ukoliko je čvor j aktivan, bez obzira na aktivni status čvora i. Ovo znači da aktivan čvor i šalje svoje rešenje susedima, bez obzira na to da li su oni aktivni, ali prima vrednosti samo od aktivnih suseda. U protivnom, kada je čvor i neaktivan, ne šalje svoju procenu rešenja susedima, ali prihvata njihova rešenja.

Vrednost M_i^k može da ima vrednost $M_i^k = I$, što podrazumeva metode prvog reda. Druga mogućnost je $M_i^k = D_i^k$, gde je:

$$D_i^k = \alpha \nabla^2 f_i(x_i^k) + (1 - W_{ii})I, \tag{A.4}$$

što odgovara metodama drugog reda. Na osnovu definisanih principa, osnovna forma algoritma za metode primarnog tipa se može prikazati pseudokodom, datim unutar Algoritma 4.

Algoritam 4 Pseudokod algoritma za metode primarnog tipa

Potrebno na svakom čvoru i: $\alpha > 0$; $\{W_{ij}\}_{j \in \Omega_i}$; $\{p_k\}_{k \geq 0}$
  ponovi
    Svaki čvor i generise z_i^k i računa:
    M_i^k i $\xi_{i,j}^k$, $j \in \Omega_i$
    ako $\xi_{i,j}^k = 1$ **onda**
      Svaki čvor i prima x_j^k od čvora $j, j \in \Omega_i$
    kraj ako
    Svaki čvor i racuna x_i^k kao (2.9) – (2.10)
  dok nije ispunjen kriterijum zaustavljanja

Implementacija metoda

Paralelna implementacija predloženog algoritma, razvijena je upotrebom MPI tehnologije, u programskom jeziku C. Kada govorimo o MPI tehnologiji, potrebno je deinisati nekoliko pojmova. Pre svega, prilikom izvršavanja paralelnog programa, skup procesa radi

istovremeno na rešavanju problema. Svaki proces ima svoj identifikator, odnsono redni broj. Proces sa rednim brojem 0 nazivamo master proces, i on je zadužen za koordinaciju paralelnog izvršavanja. Kada je reč o implementaciji, pojmove proces i čvor koristimo naizmenično. Za potrebe efikasnih operacija nad vektorima i matricama, koriste se odgovarajuće rutine iz biblioteka LAPACK [120] i BLAS [121], na svakom čvoru. Algoritam je razvijen za dva različita slučaja funkcije cilja: kvadratne funkcije i funkcije logističkog gubitka (logistic loss). Implementacija je razvijana u nekoliko etapa, gde je nakon svake od njih urađena empirijska analiza, kako bi se utvrdila i eliminisala potencijalna "uska grla" po pitanju performansi. Prvobtino, implementacija je zasnovana na principu komunikacije svako-sa-svakim. Ovo podrazumeva da svaki čvor komunicira sa svim ostalim čvorovima fizički, bez obzira na strukturu grafa G. Naknadno, svaki čvor uzima u obzir rešenja u skladu sa definicijom grafa G.

Kako se ovaj pristup komunikacije svako-sa-svakim pokazao neefikasnim i neskalabilnim, zamenjen je upotrebom komunikatora. Komunikatori su koncept iz MPI tehnologije, koji omogućuju komunikaciju između određenog skupa čvorova. Podrazumevani, globalni komunikator, koji se implicitno kreira, pri pokretanju MPI aplikacije, sadrži sve procese. Međutim, moguće je programski kreirati komunikator, koji sadrži samo određene procese. Praktično, za potrebe algoritma, kreiramo niz komunikatora, po jedan za svaki proces, gde je dati proces master proces, i osim datog procesa, sadrži sve procese, koji su njemu susedni. Na ovaj način, obezbeđuje se da svaki čvor komunicira isključivo sa susednim čvorovima, shodno definiciji grafa G.

Nakon toga, istražujemo i implementacije algoritma, sa sparsifikovanom komunikacijom, na opisan način. Kako bi se realizovao koncept sparsifikovane komunikacije, definišemo verovatnoću p_k. Verovatnoća može imati fiksnu vrednost tokom iteracija, ili može da se menja. U okviru teze, testirane su fiksne vrednosti $p_k = 0.3$, $p_k = 0.5$ i $p_k = 0.8$. Takođe, razmatramo rastuću, $p_k = 1 - 0.5^k$, i opadajuću verovatnoću, $p_k = \frac{1}{k+1}$, gde je k brojač iteracije. Iz aspekta implementacije, ovo znači da je na početku svake iteracije potrebno ponovo kreirati komunikatore, s obzirom da sastav svakog komunikatora zavisi od toga da li su potencijalni kandidati za njega aktivni. Aktivnost procesa u datoj iteraciji, određuje se generisanjem slučajne vrednosti iz zadatog intervala i određivanjem da li je ta vrednost u granicama zadate verovatnoće. U skladu sa ovim je potrebno prilagoditi i implementaciju razmene podataka među procesima. Kada je u pitanju koncept jednosmerne komunikacije, kreiranje komunikatora na početku iteracije je potrebno prilagoditi tako, da sada i neaktivni čvorovi zadrže svoj komunikator, u kome će učestvovati aktivni susedi, kako bi čvor mogao primati podatke od aktivnih suseda, i onda kada je neiaktivan.

Ulazni podaci su organizovani u 4 ulazna fajla, za matricu, vektor i 2 fajla za podatke o grafu, gde su sadržane informacije o matrici susedstva i stepenima čvorova. Master proces učitava podatke i raspoređuje ih svim procesima. Ukoliko dimenzija ulaznih podataka nije deljiva sa brojem procesa, ostatak se dodeljuje master procesu. Po pitanju prirode grafa G, razmatraju se jednostavni slučajevi upotrebe regularnih i mrežnih grafova. Kriterijum zaustavljanja algoritma se definiše na sledeći način: iteracije algoritma se nastavljaju sve dok nije ispunjen uslov $\|\nabla\phi(x^k)\| \leq \epsilon$, gde je $\epsilon = 0.01$. Gradijent $\nabla\phi(x^k)$ ne može da se izracuna na nivou jednog čvora. Na kraju svake iteracije, master čvor sakuplja vrednosti od ostalih čvorova i određuje euklidsku normu vektora. Ukoliko je kriterijum zaustavljanja ispunjen, šalje se odgovarajući signal svim procesima.

Algoritam je univerzalan u odnosu na funkciju cilja f_i, i može se lako prilagoditi za implementaciju nove funkcije cilja, jednostavnom zamenom izračunavanja gradijenta i Hessian-a. Unutar teze, razvili smo implementaciju za dve funkcije cilja. Implemetacija za kvadratne funkcije podrazumeva sledeću funkciju cilja:

$$f_i(x) = \frac{1}{2}(x - b_i)^\tau a_i(x - b_i), \tag{A.5}$$

gde $x \in \mathbb{R}^s$ predstavlja optimizacionu varijablu, a_i je ulazna matrica, a b_i je ulazni vektor na svakom čvoru. Implementacija za funkcije logističkog gubitka sa L2 regularizacijom podrazumeva:

$$f_i(x) = \sum_{j=1}^{J} \mathcal{J}_{logis}(b_{ij}(x_1^\top a_{ij} + x_0)) + \frac{\tau}{n}||x||^2. \tag{A.6}$$

U ovoj formulaciji, $x = (x_1^\top, x_0) \in \mathbb{R}^{s-1} \times \mathbb{R}$ predstavlja optimizacionu varijablu, τ je kazneni (penalty) parametar, dok su a_i i b_i ponovo ulazni podaci, i važi da je $\mathcal{J}_{logis}(z) = log(1 + e^{-z})$.

Evaluacija metoda

Kada je u pitanju evaluacija osobina opisanog algoritma, mogu se definisati dve celine. Prva se tiče testiranja algoritma tokom razvoja implementacije i poređenje različitih aspekata implementacije. Druga celina se tiče temeljne analize formirane klase posmatranih metoda, koje koriste različite strategije sparsifikovanja komunikacije. Za potrebe testiranja, korišceno je nekoliko različitih infrastruktura. Pre svega, za testiranje serijskih implementacija, korišćena je 64-bitna Linux konfiguracija, sa Core i5-4590 3.30GHz procesorom sa 4 jezgra i 16 GB RAM memorije. Paralelne implementacije, testirane se na realnom računarskom klasteru. Za ovu svrhu, korišćena su 2 klastera: AXIOM, koji sadrži 16 računarskih jedinica (8 x Intel i7 5820k 3.3GHz i 8 x Intel i7 8700 3.2GHzCPU - 192

jezgra i 16GB DDR4 RAM-a po čvoru), povezanih 10 Gbps mrežom; i Paradox, koji sadrž 106 računarskih jedinica (2 x 8 jezgara Sandy Bridge Xeon 2.6GHz procesori sa 32GB RAM-a + NVIDIAő Tesla M2090), povezanih QDR InfiniBand mrežom.

Rezultati i zaključci na osnovu analize implementacionih aspekata

Prvobitni skup analiza, tiče se evaluacije razvijene implementacije i ulaganje napora u njeno poboljšanje. Pre svega, prvobitna paralelna implementacija, sa upotrebom komunikacionog protokola svako-sa-svakim za kvadratne funkcije, upoređena je sa serisjkom implementacijom, razvijenom u MATLAB-u. Kako je priroda algoritma takva da podrazumeva paralelno izvršavanje, serijska implementacija izvršava deo posla svakog čvora sekvencijalno, u jednoj petlji. Ova inicijalna analiza je potvrdila da je paralelna implementacija daleko efikasnija po pitanju performansi, što je očekivani zaključak. Međutim, ovakva implementacija se ne skalira, a razlog je u pristupu komunikacije, baziranom na principu svako-sa-svakim. Ovi rezultati su motivisali adaptaciju implementacije, tako da se inkorporira upotreba komunikatora i time se smanje troškovi komunikacije. Za testiranje algoritma za kvadratne funkcije, korišćeni su sintetički podaci.

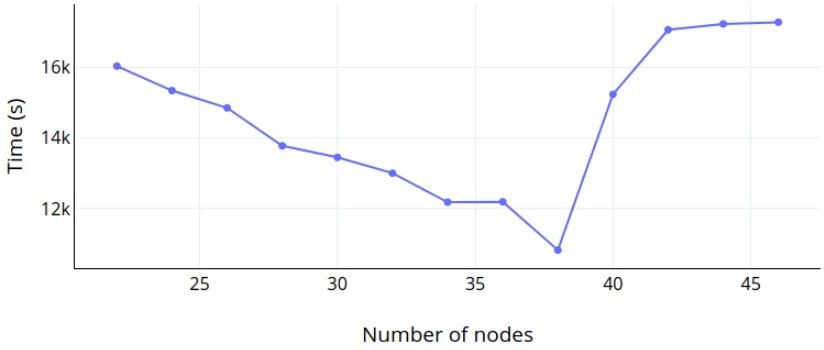
U nastavku, testovi su bazirani na funkcijama logističkog gubitka, nad realnim, javno dostupnim skupovima podataka. Tabela A.1 prikazuje poređenje vremena izvršavanja za pristup sa komunikacijom po principu svako-sa-svakim i upotrebom komunikatora, za dva skupa podataka: Conll [123] i Gisette[125]. Evidentno je da se upotrebom komunikatora smanjuje vreme izvršavanja, što je za veći skup podataka izraženije.

Tabela A.1: Primer poređenja vremena izvršavanja za algoritam sa komunikacijom svako-sa-svakim i sa upotrebom komunikatora

Skup podatala	broj čvorova	Vreme(s): svako-sa-svakim	Vreme(s): komunikatori
Conll	26	2.815006	2.610187
Gisette	42	6926.05	6628.73

Algoritam sa komunikatorima se dobro skalira. Primer skaliranja, prikazan je na Slici A.1. Odavde se jasno vidi da se vreme izvršavanja algoritma smanjuje sa povećanjem broja čvorova, dok se ne dostigne optimalan broj čvorova za dati primer. Nakon toga, vreme izvršavanja počinje da raste, s obzirom da troškovi paralelizacije sa većim brojem čvorova postaju skuplji, u odnosu na dobit, koji dobijamo paralelizacijom.

Analizom potrebne količine vremena za izvršavanje raznih delova algoritma, jasno se vidi da najveći procenat vremena oduzima komunikacija. Stoga se pojavljuje motivacija za uvođenje različitih pristupa sparsifikacije komunikacije. U ovom kontekstu, testiran je



Slika A.1: Skaliranje algoritma sa komunikatorima, na Gisette skupu podataka

Tabela A.2: Metode Algoritma 1

Naziv metode	Broj metode	Tip	M_i^k	$\xi_{i,j}^k$	p_k	Relevantna referenca
FBI	Metod 1	Prvog reda	I	$z_i^k \cdot z_j^k$	$p_k = 1 - 0.5^k$	[40]
FBD	Metod 2	Prvog reda	I	$z_i^k \cdot z_j^k$	$p_k = (k+1)^{-1}$	[43]
FUI	Metod 3	Prvog reda	I	z_j^k	$p_k = 1 - 0.5^k$	nova metoda [46]
FUD	Metod 4	Prvog reda	I	z_j^k	$p_k = (k+1)^{-1}$	[44]
FBC	Metod 9	Prvog reda	I	1	1	[41]
SBC	Metod 0	Drugog reda	D_i^k	1	1	[12]
SBI	Metod 5	Drugog reda	D_i^k	$z_i^k \cdot z_j^k$	$p_k = 1 - 0.5^k$	[39]
SBD	Metod 6	Drugog reda	D_i^k	$z_i^k \cdot z_j^k$	$p_k = (k+1)^{-1}$	[39, 43]
SUI	Metod 7	Drugog reda	D_i^k	z_j^k	$p_k = 1 - 0.5^k$	nova metoda [46]
SUD	Metod 8	Drugog reda	D_i^k	z_j^k	$p_k = (k+1)^{-1}$	nova metoda [46]

skup od 20 metoda, koje predstavljaju kombinacije različitih pristupa sparsifikaciji (koriste različite verovatnoće komunikacije, konkretno 0.3, 0.5, 0.8, kao i rastuću i opadajuću), mogu biti prvog ili drugog reda i mogu biti jednosmerne ili dvosmerne. Nakon analize izvršavanja ovih metoda, identifikovane su one, koje pokazuju najbolje performanse, a osim njih, u konačnu klasu predloženih metoda, ulaze i odgovarajući parovi metoda.

Klasa posmatranih metoda

Konačna klasa predloženih metoda, može da se definiše na način, prikazan u Tabeli A.2. Kako bi identifikacija metoda bila jednostavnija, uveden je princip imenovanja metoda, koji se sastoji od 3 slova: prvo slovo označava da li je metoda prvog ili drugog reda (F za first i S za second order, na engleskom), drugo slovo označava da li je komunikacija jednosmerna ili dvosmerna (B za bidirectional i U za unidirectional, na engleskom), treće slovo označava tip verovatnoće (I za increasing, D za decreasing i C za constant, na engleskom). Jedine metode sa konstantnom verovatnoćom su FBC i SBC. Metoda SBC

je inicijalna verzija algoritma, koja ne sparsifikuje komunikaciju. Ona je drugog reda, a metoda FBC je odgovarajuća metoda prvog reda, takođe bez sparsifikacije. Varijante metoda sa konstantnom verovatnoćom 0.3, 0.5, 0.8 se nisu pokazale dovoljno efikasnim, kako bismo ih uključili u ovaj skup.

Experimentalni rezultati nad klasom odabranih metoda

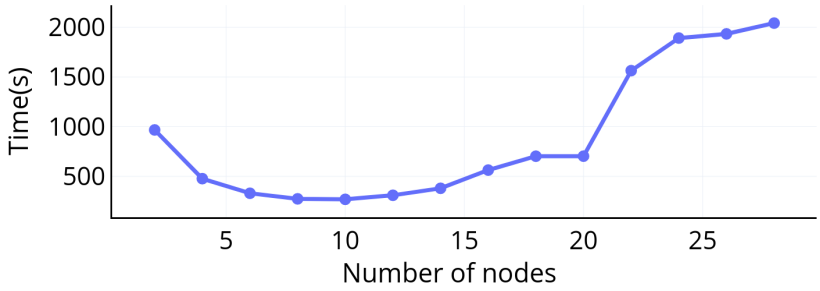
Prilikom testiranja klase predloženih metoda, korišćeni su 8-regularni grafovi. Testirano je nekoliko alternativa regularnih i mrežnih grafova, a 8-regularni su se pokazali kao najpogodniji među njima. Kada su u pitanju vremena izvršavanja različitih metoda, razmotrimo Tabelu A.3. Unutar nje su prikazana vremena izvršavanja za svih 10 metoda, nad p53 [135] skupom podataka, i mrežu od 20 čvorvoa. Kada je u pitanju vreme izvršavanja paralelnih programa, konačna vrednost vremena je vreme potrebno da najsporiji proces završi svoj deo posla. Iz Tabele A.3 se može videti da metode prvog reda uvode značajno smanjenje potrebne količine vremena izvršavanja. U ovom slučaju, metoda FBD ima najbolje performanse. Kada se uporedi metod FBC sa metodom SBC, jasno je da izračunavanje informacije drugog reda značajno povećava vreme izvršavanja, jer se ove metode razlikuju samo u toj dimenziji. Smanjenje koliine komunikacije tokom iteracija kroz opadajuću verovatnoću pomoću metode FBD ovde dovodi do bržeg izvršenja, u odnosu na rastuću vrednost verovatnoće. Međutim, ovo ponašanje može biti veoma zavisno od prirode samog skupa podataka. Algoritmi za skup podataka p53 generalno brzo konvergiraju, unutar relativno malog broja iteracija. Jednako važan aspekt ovde je takođe činjenica da Metod FUD, koristeći jednosmernu komunikaciju i opdajuću verovatnoću, radi bolje od metode FBI, sa dvosmernom komunikacijom i rastućom verovatnoćom komunikacije. Posmatranje vremena izvršenja za metode drugog reda dokazuje da uvođenje sparsifikacije komunikacije uglavnom nije isplativo, jer se računanjem informacija drugog reda oduzima značajna količina vremena, koja se zbog sparsifikacije ne može kompenzovati.

Potrebno je razmotriti i osobinu skalabilnosti predloženih metoda. Posmatramo Sliku A.2, gde je prikazano skaliranje za metodu FBI, na skupu podataka YearPredictionMSD [129]. Na slici se može identifikovati optimalan broj čvorova za ovaj slučaj i uočava se uobičajen trend za paralelne programe: vreme izvršavanja opada, dok se ne dostigne optimalan broj čvorova, nakon čega počinje da raste, jer dodavanje novih čvorova više nije isplativo i vreme komunikacije postaje dominantno.

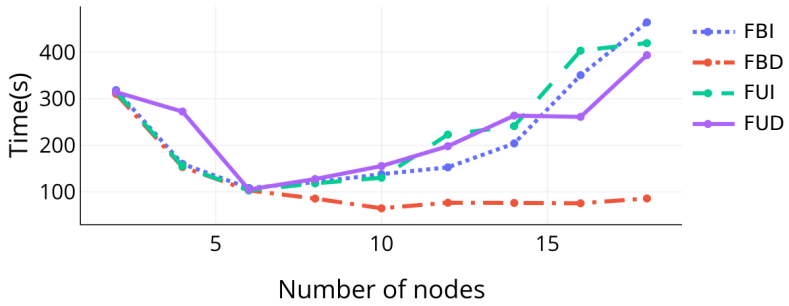
Slika A.3 prikazuje vremena izvršavanja metoda prvog reda, koje koriste sparsifikaciju komunikacije, na skupu podataka CT [133]. Jasno se vidi da je optimalan broj čvorova za metode FBI, FUI i FUD ista vrednost, $n = 6$. Metoda FBD se ponaša malo drugačije. Generalno, ona rezultuje nižim vrednostima vremena izvršavanja, a optimalan broj čvorova

Tabela A.3: Vremena izvršavanja različitih metoda Algoritma 4 (A.3)-(A.2), za 20 čvorova u mreži, na p53 skupu podataka

Metod	Vreme izvršavana (s)
FBI	4.64
FBD	1.89
FUI	6.04
FUD	3.56
FBC	3.16
SBC	9661.42
SBI	43126.71
SBD	22683.84
SUI	22029.20
SUD	9651.77



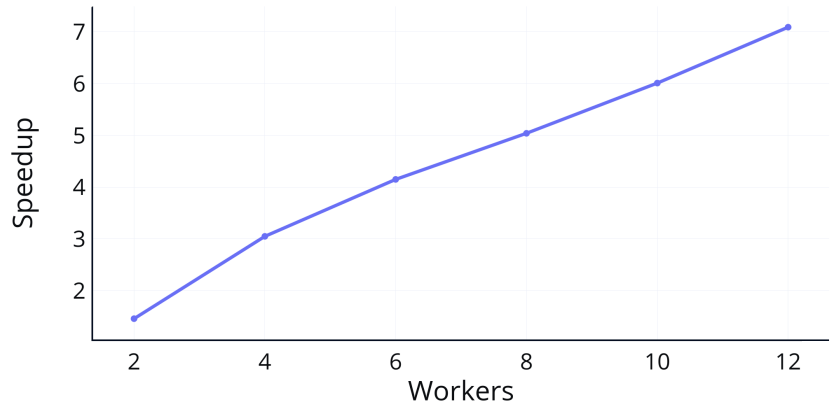
Slika A.2: Osobine skaliranja Metode FBI, za YearPredictionMSD skup podataka



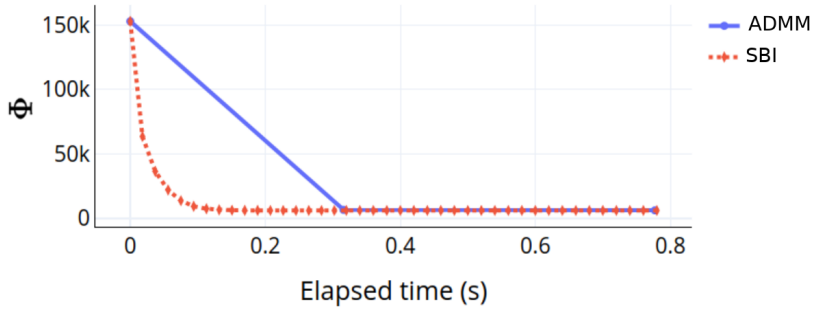
Slika A.3: Vremena izvršavanja za metode prvog reda na skupu podataka CT

za nju je $n = 10$.

Ubrzanje algoritma je moguce predstaviti i grafički, kao na primer u [139]. Primer za ovo, prikazan je na Slici A.4, gde je iscrtano ubrzanje metode FBD na Mnist skupu podataka, za broj čvorova od 2 do 12, s obzirom da je 12 optimalna broj čvorova za ovaj test slučaj. Ubrzanje na y osi, može se računati upotrebom Amdahl-ovog pravila [140], kao odnos



Slika A.4: Ubrzanje za FBD metodu na Mnist skupu podataka



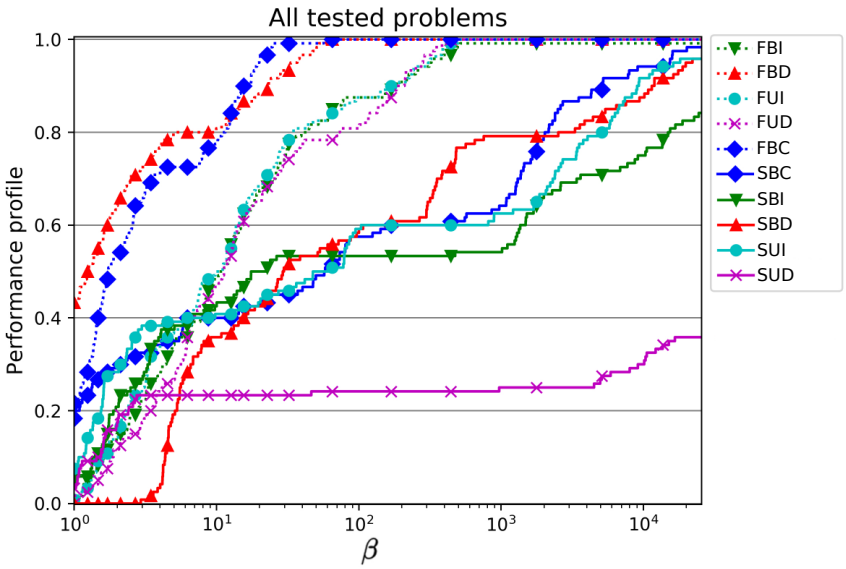
Slika A.5: Poređenje ADMM pristupa sa SBI metodom na Conll skupu podataka

vremena izvršavanja programa za jedan čvor i vremena izvršavanja za n čvorova. U idealnom slučaju, očekuje se linearno ubrzanje. U ovom slučaju, evidentno je da metoda FBD pokazuje veoma zadovoljavajući nivo ubrzanja sa povećanjem broja čvorova.

Kako se opisani problem može rešiti korišćenjem ADMM metode [3], uporedili smo algoritam sa ADMM implementacijom za logističku regresiju, na Conll skupu podataka. Tačnije, ADMM metoda rešava problem pretpostavljajući prisustvo centralnog čvora koji komunicira sa svi ostalim čvorovima u mreži. Zbog toga, prilagođavamo naš algoritam, tako da graf G bude potpuno povezan i da matrica W sadrži sve jednake vrednosti $1/n$. Rezultati poređenja su prikazani na Slici A.5

Računamo vrednost $\Phi^k = \frac{1}{n}\sum_{i=1}^{n} f(x_i^k)$, tj. prosečni globalni trošak u (A.1) u proseku, kroz sve čvorove, na kraju svake iteracije. Merimo vreme potrebno da se zadovolji $\frac{\Phi^k - f^*}{f^*} < 0.1$. Ovde, $f^\star$ se numerički evaluaira od strane ADMM-a. U ovom poređenju, kao najbolja, pokazala se metoda SBI.

Činjenica da metoda drugog reda ima bolje performanse u ovom slučaju, u odnosu na metode prvog reda, konzistentna je sa ranijim zaključcima. U principu, za manje skupove podataka, metode drugog reda imaju tendenciju da budu efikasnije. Slika A.5 pokazuje

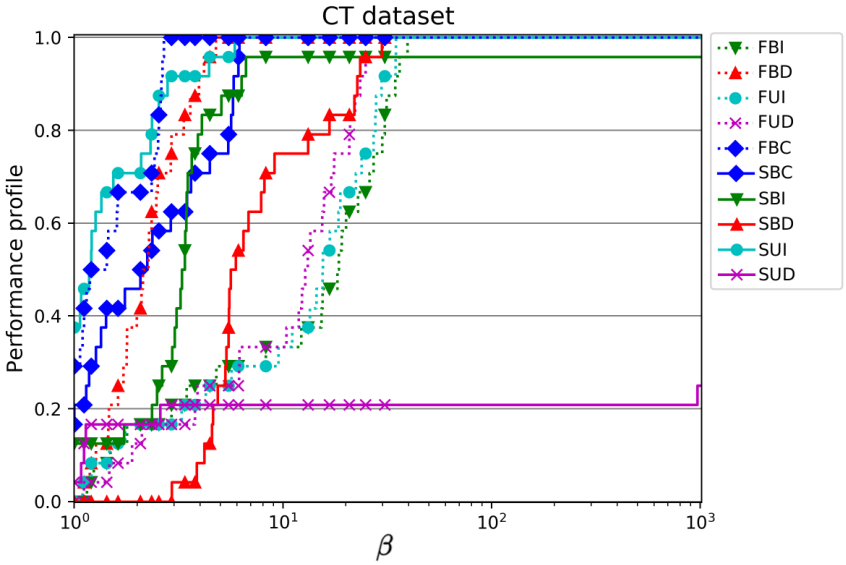


Slika A.6: Profil performansi za svih 10 metoda, na osnovu svih izvršenih testova

da SBI metoda koristi veći broj znatno bržih iteracija, u odnosu na ADMM, i rezultuje nižim celokupnim vremenom izvršavanja.

Kada je reč o performansama i njihovom međusobnom poređenju, profil performansi [141] predstavlja vrlo pregledan i koristan pristup za evaluaciju. Slika A.6 prikazuje profil performansi za svih 10 metoda, na osnovu svih izvršenih testova. Ovakva evaluacija nam, između ostalog, daje priliku da uporedimo parove metoda prvog i drugog reda, koje koriste istu strategiju sparsifikacije. Posmatrajući metode bez sparsifikacije, odnosno metode SBC i FBC, slika pokazuje da metoda prvog reda, metod FBC, ima bolje rezultate od metode drugog reda, metode SBC. Isto važi i ako razmatramo metode sa sparsifikacijom. Uzimajući u obzir metode sa smanjenom verovatnoćom komunikacije i korišćnjem dvosmerne komunikacije, ispostavlja se da metoda FBD ima mnogo bolje perfromanse od metode SBD. Prilikom uporeivanja ostalih metoda prvog i drugog reda koristeći istu sparsifikaciju (metod FBI i SBI, metod FUI i SUI, metod FUD i SUD), metode prvog reda imaju bolji učinak u 61% test slučajeva. Postoje slučajevi u kojima metode drugog reda dominiraju. Ako posmatramo Sliku A.7, gde je prikazan profil performansi za skup podataka CT, vidimo da je u ovom slučaju metoda sa najboljim performansama, metoda drugog reda, SUI. Razlog za ovo je dimenzionalnost skupa podataka. Za manje skupove podataka, informacija drugog reda se računa brže, potreban je manji broj iteracija, i time dolazi do brže konvergencije. Ovaj trend važi generalno, za skupove podata manje dimenzionalnosti.

Prikazane analize su pokazale da metodi iz predloženog skupa imaju osobine skalabilnosti. Pokazano je da za svaku metodu, može da se identifikuje optimalan broj čvorova na datom



Slika A.7: Profil perofmansi za svih 10 metoda, na osnovu testova izvršenim na CT skupu podataka

test skupu. Međusobnim poređenjem metoda, zaključuje se da su metode prvog reda bolji izbor po pitanju performansi, kada skup podataka ima veliki broj osobina (10^3 ili više za prikazane rezultate). S druge strane, kod manjih dimenzija podataka, metode drugog reda pokazuju bolje performanse. Metodama prvog reda je potreban veći broj iteracija za konvergenciju, ali su te iteracije brze, što je suština razlike u ponašanju. Pokazano je da je metoda FBD, metoda sa najboljim performansama unutar prezentovanog skupa testova. Međutim, evidentno je da i metode sa jednosmernom komunikacijom mogu da imaju veoma dobre performanse, u nekim slučajevima i bolje od onih sa dvosmernom komunikacijom, pa stoga predstavljaju takođe značajan doprinos.

Metoda distribuirane optimizacije dualnog tipa

U drugom delu teze, razmatra se dualna metoda, Alternating direction method of multiplyers (ADMM) [3]. Metoda je u širokoj primeni u domenu problema distribuirane optimizacije. Danas, potreba za efikasnim algoritmima mašinskog učenja neprestano raste, praćena težnjom da se obradi velika količina podataka, na što efikasniji način. Klasterovanje (grupisanje) predstavlja značajan metod učenja bez nadzora, koji pronalzi primenu u raznim domenima. Stoga je od interesa da se razvijaju distribuirana, paralelna rešenja za grupisanje podataka. U drugom delu teze, opisujemo i praktično evaluairamo paralelni algoritam za konveksno grupisanje podataka, baziran na ADMM-u.

Teorijske osobine metode

Predložena metoda za grupisanje podataka, zasniva se na ideji SON klasterovanja [35], primenom ADMM metode. Posmatramo skup N observacija $\{a_j\}_{j=1}^{N} \in \mathbb{R}^d$. S obzirom da razvijamo paralleni algoritam, potrebno je sav posao podeliti i delegirati skupu od K čvorova, tj. procesa. Primenjujemo manager-workers princip model komunikacije. Povezane čvorove označavamo indeksima na sledeći način: mater čvor označavamo sa 1, a ostale čvorove sa $2, ..., K$. Ulazni podaci se dele jednako među čvorovima, i sa $a_{ij} \in \mathbb{R}^d$ označavamo j-tu tačku, dostupnu na čvoru i, gde je $i = 1, ..., K, j = 1, ..., \frac{N}{K}$. Uvodimo nekoliko izmena, u odnosu na standardnu formu konveksnog grupisanja. Prvo, ovde ne penalizujemo razlike po svim parovima klaster kandidata. Umesto toga, dodeljujemo master čvoru "centralni" klaster kandidat, koji označavamo sa x_{11}. Slično tome, svim ostalim čvorovima (workers) $i = 2, ..., K$, dodeljujemo lokalni "centralni" klaster kandidat x_{i1}. Drugim rečima, unutar tačaka podataka na jednom čvoru $i, i = 1, ..., N$, penalizujemo razliku između lokalnog centra x_{i1} i ostalih tačaka $x_{ij}, j = 2, ..., \frac{N}{K}$ na tom čvoru. Osim toga, penalizujemo i razlike između master centra x_{11} i lokalnih centara $x_{i1}, i = 2, ..., K$. Na osnovu navedenog, predložena formulacija konveksnog kalsterovanja ima sledeću formu:

$$\underset{x_{ij}}{minimize} \sum_{i=1}^{K}\sum_{j=1}^{\frac{N}{K}} ||a_{ij} - x_{ij}||^2 + \gamma \sum_{i=1}^{K}\sum_{j=2}^{\frac{N}{K}} ||x_{i1} - x_{ij}|| + \gamma \sum_{i=2}^{K} ||x_{11} - x_{i1}||, \tag{A.7}$$

gde se minimizacija vrši u odnosu na varijable $x_{ij} \in \mathbb{R}^d, i = 1, ..., K, j = 1, ..., \frac{N}{K}$, i važi $\gamma > 0$. Definisanom problemu, može se dodeliti graf $G = (\mathcal{N}, E)$, gde je $\mathcal{N}$ skup čvorova, gde svaki odgovara jednoj varijabli $x_{ij}, i = 1, ..., K, j = 1, .., \frac{N}{K}$, a E je skup grana $(i, j) \sim (l, m)$, tako da među čvorovima (i, j) i (l, m) postoji grana, ako druga suma u (A.7) uključuje term $\|x_{ij} - x_{lm}\|$. U smislu grafovske reprezentacije, originalan problem konveksnog klasterovanja, dobija se zamenom grafa G kompletnim grafom. Takođe, (A.7) mogao bi dodatno da se posmatra i kao grupisanje sa dodatom težinskom komponentom. Predloženi algoritam ne pretpostavlja nikakvno poznavanje strukture ili raspodele podataka po čvorovima unapred i konstrukcija grafa je nezavisna od vrednosti ulaznih podataka. Nakon primene ADMM pristupa, problem može da se reformuliše na sledeći način:

$$\begin{aligned} &minimize \sum_{i=1}^{K}\sum_{j=1}^{\frac{N}{K}} ||a_{ij} - x_{ij}||^2 + \gamma \sum_{i=1}^{K}\sum_{j=2}^{\frac{N}{K}} ||x_{i1} - x_{ij}|| + \gamma \sum_{i=2}^{K} ||x_{11} - y_{i1}|| \\ &\text{s.t. } y_{i1} = x_{i1}, i = 2..K. \end{aligned} \tag{A.8}$$

Drugim rečima, za centar svakog čvora $x_{i1}, i = 1, ..., K$, uvodimo pomoćnu varijablu y_{i1} i dodajemo ograničenje $y_{i1} = x_{i1}$ kako bi problem (A.8) bio ekvivalentan sa (A.7). Varijable u (A.8) su tada $\{x_{ij}\}, i = 1, ..., K, j = 1, ..., \frac{N}{K}$, i $y_{i1}, i = 1, ..., K$. Dualizovanjem ograničenja u (A.8), formira se funkcija proširenog Lagranžijana $L_\rho : \mathbb{R}^{Nd} \times \mathbb{R}^{Kd} \times \mathbb{R}^{Kd} \rightarrow \mathbb{R}$, na sledeći način:

$$\begin{aligned} L_\rho(x, y; \lambda) = &\sum_{i=1}^{K} \sum_{j=1}^{\frac{N}{K}} ||a_{ij} - x_{ij}||^2 + \gamma \sum_{i=1}^{K} \sum_{j=2}^{\frac{N}{K}} ||x_{i1} - x_{ij}|| + \gamma \sum_{i=2}^{K} ||x_{11} - y_{i1}|| \\ &+ \sum_{i=2}^{K} \lambda_i^T (y_{i1} - x_{i1}) + \frac{\rho}{2} \sum_{i=2}^{K} ||y_{i1} - x_{i1}||^2, \end{aligned} \tag{A.9}$$

gde je $\rho > 0$ kazneni (penalty) parametar. Konačno, dobijamo skup sledećih ažuriranja:

- ažuriranje vrednosti x, na svakom čvoru $i = 2..K$, paralelno:

$$\begin{aligned} x_{ij}^{k+1} = argmin &\sum_{j=1}^{\frac{N}{K}} ||a_{ij} - x_{ij}||^2 + \gamma \sum_{j=2}^{\frac{N}{K}} ||x_{i1} - x_{ij}|| \\ &+ (\lambda_i^k)^T (y_{i1}^k - x_{i1}) + \frac{1}{2} \rho ||y_{i1}^k - x_{i1}||^2 \end{aligned} \tag{A.10}$$

- ažuriranje vrednosti x na master čvoru:

$$x_{1j}^{k+1} = argmin \sum_{j=1}^{\frac{N}{K}} ||a_{1j} - x_{1j}^k||^2 + \gamma \sum_{j=1}^{\frac{N}{K}} ||x_{1j} - x_{11}|| + \gamma \sum_{i=2}^{k} ||x_{11} - y_{i1}^k|| \tag{A.11}$$

- ažuriranje vrednosti y na master čvoru:

$$y_{i1}^{k+1} = argmin \sum_{i=2}^{K} (\lambda_i^k)^T (y_{i1} - x_{i1}^{k+1}) + \frac{1}{2} \rho \sum_{i=2}^{k} ||y_{i1} - x_{i1}^{k+1}||^2 + \gamma \sum_{i=2}^{K} ||x_{11}^{k+1} - y_{i1}||. \tag{A.12}$$

- ažuriranje vrednosti λ na master čvoru:

$$\lambda_i^{k+1} = \lambda_i^k + \rho (y_{i1}^{k+1} - x_{i1}^{k+1}) \tag{A.13}$$

Komunikacija, odnosno sinhronizacija među čvorovima, vrši se nakon ažuriranja vrednosti za varijablu x, kada su svi čvorovi ažurirali svoju varijablu. Nakon toga, master

čvor izračunava ažurirane vrednosti y i λ i saopštava ih ostalim čvorovima. Ovakva formulacija ne garantuje savršeno grupisanje. Međutim, numeričke evaluacije potvrđuju da algoritam rezultuje aproksimativnim klasterovanjem za rešenje $\{x_{ij}^*\}$. Drugim rečima, $\{x_{ij}^*\}$ se grupišu u određen broj različitih klastera, K', tako da su $\{x_{ij}^*\}$ tačke unutar iste grupe veoma bliske. Ovo predstavlja motivaciju za razvoj procedure spajanja bliskih tačaka, nakon što algoritma konvergira. Ova procedura je prikazana u okviru Algoritma 5.

Algoritam 5 Procedura spajanja potencijalnih centara

Na svakom čvoru i **lokalno, paralelno:**
Potrebno $\epsilon_i, i = 1, ..., K; \epsilon$; inicijalizovati listu lokalnih centara $\mathcal{C}_i = \{\}$
za sve moguće kandidate centara x_{ij}^* **radi**
za vec prihvacene centre $c_{il} \in \mathcal{C}_i$ **radi**
ako $\|x_{ij}^* - c_{il}\| \leq \epsilon_i$ **onda**
izostavi x_{ij}^* iz skupa centara $\mathcal{C}_i$
else
uključi x_{ij}^* u skup centara $\mathcal{C}_i$
kraj ako
kraj za
kraj za
vrati pronađene lokalne centre $\mathcal{C}_i = \{c_{i1}, ..., c_{iP_i}\}$, gde P_i je broj lokalnih centara, pronađenih na čvoru i
Na master čvoru:
Potrebno ϵ; Svi mogući kandidati centara čvorova: $\mathcal{C}_i, i = 1, ..., K$
Potrebno inicijalizovati listu konacnih centara $\mathcal{C} = Null$
za sve moguce kandidate centara svih čvorova $c_{ij} \in \mathcal{C}_i, i = 1, ..., K$ **radi**
za vec prihvacene centre $c_l \in \mathcal{C}$ **radi**
ako $\|c_{ij} - c_l\| \leq \epsilon$ **onda**
izostavi c_{ij} iz skupa centara $\mathcal{C}$
else
uključi c_{ij} u skup centara $\mathcal{C}$
kraj ako
kraj za
kraj za
vrati pronađene centre $\mathcal{C} = c_1^\star, ..., c_{P'}^\star$, gde je P' konačan broj centara

Prva faza spajanja centara, vrši se lokalno, na svakom čvoru. Vrednosti ϵ_i i ϵ su pozitivni brojevi, koji se koriste za filtriranje potencijalnih centara. Prva tačka kandidat za centar se proglašava prvim centrom, i nakon toga se proverava ostatak tačaka. Sve one tačke koje su blizu (u smislu ϵ_i), tačkama koje su već označene kao centri se ignorišu. U suprotnom, tačka se označava kao novi centar. Drugi korak je spajanje dobijenih lokalnih centara unutar čvorova, na master čvoru. Ovo znači da svi lokalni centri treba prvo da se sinhronizuju na masteru. Nakon toga, primenjuje se isti postupak za spajanje. Ovo

je značajno, s obzirom da je moguće da se neki centri, pronađeni na različitim čvorovima preklapaju. Nakon ove procedure, dobija se konačan skup centroida. Pseudokod za algoritam grupisanja možemo opisati, kao što je predstavljeno unutar Algoritma 6.

Algoritam 6 Pseudokod za predloženi algoritam grupisanja

Potrebno globalni parametri γ i $\epsilon^\star$, i na svakom čvoru i: $a_{ij}, j = 1, ..., \frac{N}{K}$
  ponovi
    Izračunati x_{1j}^k na masteru, kao u (3.15)
    Izracunati x_{ij}^k na svakom čvoru $i = 2..K$ paralelno, kao u (3.14)
    Izracunati y_{i1}^k na masteru, kao u (3.16)
    Izracunati λ_i^k na masteru, kao u (3.17)
  dok nije ispunjen kriterijum zaustavljanja
  Spojiti potencijalne centre, kao što je opisano u Algoritmu 6
Potrebno na svakom čvoru i finalna lista globalnih centara $\mathcal{C} = c_1^\star, ..., c_{P'}{}^\star$
  za sve lokalne tačke iz ulaznog skupa a_{ij} na svakom čvoru i, paralelno **radi**
    dodeliti tačku a_{ij} u klaster $c_l^\star$ gde $min_{t=1,...,P'} \|a_{ij} - c_t^\star\| = \|a_{ij} - c_l^\star\|$
  kraj za

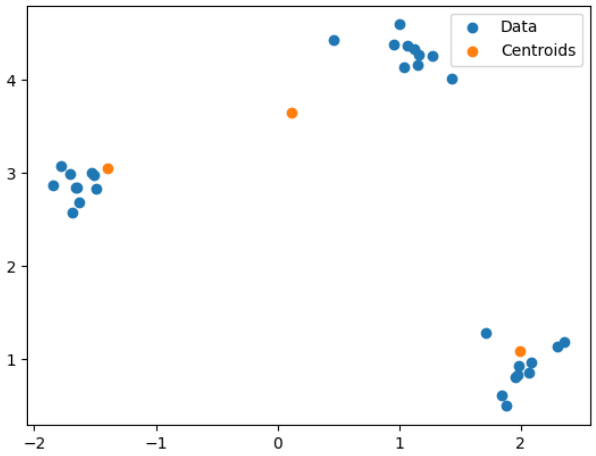
Kao što je prikazano unutar Algoritma 6, nakon što iterativan deo algoritma konvergira i primeni se procedura spajanja, kao što je opisano Algoritmom 5, master čvor sinhronizuje finalnu listu centara sa svim čvorovima. Na taj način, svaki čvor može da dodeli svoje lokalne tačke iz ulaznog skupa podataka u odgovarajući najbliži klaster, odnosno da dodeli odgovarajuću labelu tački.

Implementacija metode

Implementacija predložene metode grupisanja, razvijena je u programskom jeziku Python, upotrebom COMPSs [20] tehnologije za paralelizaciju. COMPSs nudi jednostavan model programiranja sa ciljem da olakša proces paralelizacije. Široko je prihvaćen i proširen u brojnim naučnim projektima, kao alat za razvoj aplikacija u nauci i optimizaciju njihovog izvršavanja na distribuiranim infrastrukturama. Testiranje implementacije je sprovedeno na raunarskom klasteru AKSIOM, koji je korišćen i za primarne metode, na opisan način.

Python implementacija, oslanja se na CVXPY paket [143, 144], koji koristimo za opisane minimizacije. PyCOMPSs tehnologija [116], namenjena je za rad sa programskim jezikom Python i omogućuje pristupačan način razvoja paralelne implementacije, jednostavnim anotiranjem funkcija, koje je potrebno izvršiti u paraleli. Ovakav pristup je višeg nivoa u odnosu na MPI, zbog čega je posebno interesantno istražiti oba pristupa.

Ulazni podaci se čitaju iz fajla, od strane master procesa. Prilikom evaluacija, koristimo sintetički generisane podatke, ali i neke primere realnih skupova podataka. Sintetičke po-



Slika A.8: Rezultat grupisanja sa sintetički skup podataka dimenzije 30x2, $\gamma = 0.3$,$\epsilon^\star = 2$

datke manjeg obima generišemo upotrebom generatora iz scikit-learn [146] paketa. Velike skupove sintetičkih podataka generišemo kao modele Gausovih mešavina [147].

Evaluacija metode

Cilj evalucije razvijenog pristupa za konveksno, ADMM-bazirano paralelno klasterovanje je procena kvaliteta raznih aspekata algoritma. Pre svega, evaluiramo nivo preciznosti rezultujućeg grupisanja. Ovde možemo govoriti o dva tipa testova: testiranje nad podacima za koje je poznat očekivani ishod, odnsno realne labele tačaka; i testiranje nad podacima za koje nemamo očekivane labele. U prvom slučaju, preciznost grupisanja, možemo izraziti kao procenat tačno klasterovanih tačaka. U drugom slučaju, koristimo metriku *silhouette score*.

Tačnost grupisanja

Prvo ćemo evaluirati tačnost grupisanja nad malim skupom podataka, gde možemo lako grafički prikazati rezultate. Posmatrajmo Sliku A.8, gde razmatramo skup podataka dimenzije 30×2. Jasno se može videti da su tačke organizovane u tri odvojena klastera. Takođe, sa slike se vidi da algoritam uspeva da identifikuje centre korektno. Prikazani centri su rezultat primene procedure spajanja.

Kako bismo dobili pouzdane dokaze preciznosti grupisanja, primenićemo algoritam nad skupom podataka, gde su poznate realne, očekivane labele i gde je dimenzija veća od 2. Posmatramo Iris skup podata [149, 150], koji je dostupan u *scikit-learn* biblioteci. Ovaj skup podataka sadrži 3 različite klase biljke Iris, i ima 4 atributa i 150 uzoraka.

Tabela A.4: Poređenje tačnosti za različite algoritme grupisanja, na Iris skupu podataka

Algoritam	Parametri	Broj klastera	Tačnost (u procentima)
ADMM-bazirano konveksno klasterovanje	$\gamma = 40$, $\epsilon^\star = 5$	3	93.33%
k-means	$k = 3$	3	88.66%
AMA	$\gamma \in [4.3, ..., 9.1]$	3	90.66%

Tabela A.5: Evaluacija tačnosti algoritma za podatke većih dimenzija

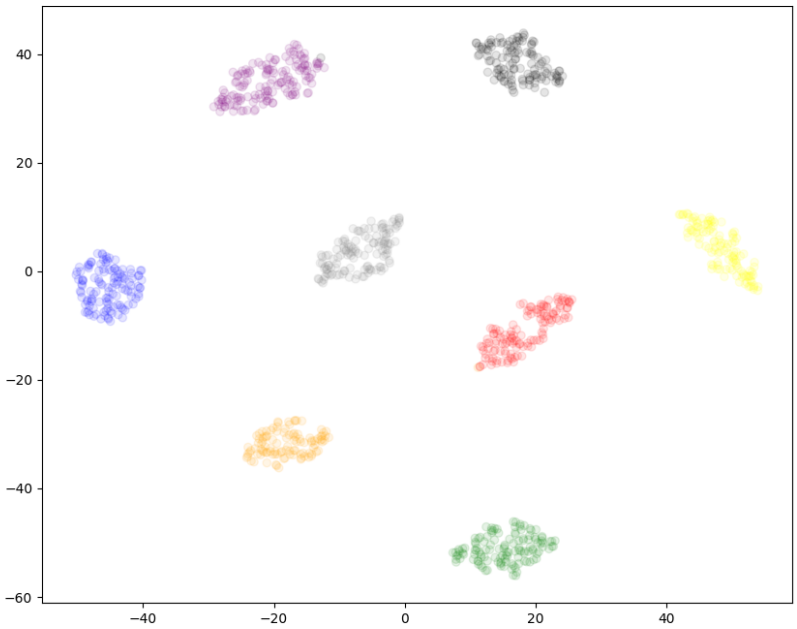
Skup podataka	γ	$\epsilon^\star$	klasteri ADMM-bazirano konveksno grupisanje	klasteri k-means	ADMM-bazirano konveksno grupisanje s.score	k-means s.score
1000×3	5.0	4	8	8	0.76	0.76
5000×3	6.6	2	4	4	0.77	0.78
5000×5	6.6	2	5	4	0.62	0.75
10000×3	6	5	10	10	0.69	0.75

Izvršili smo naš algoritam nad ovim skupom podataka i dobili smo očekivana 3 klastera. Kako bismo analizairali rezultate dodatno, uporedili smo dobijene labele sa očekivanim ishodom. Ispostavlja se da je algoritam grupisao 93.33% tačaka tačno. U principu, sve tačke koje pripadaju prvom klasteru su grupisane tačno. Do 'grešaka' je došlo kod drugog i trećeg klastera, koji su po prirodi blizu jedan drugog. Tabela A.4 prikazuje tačnost grupisanja za ovaj slučaj, ali pored rezultata za naš pristup, sadrži i podatke za standardni k-means (sa predefinisanom vrednošću parametra k), i za AMA metodu [36], radi poređenja. Evidentno je da naš metod može da postigne jednak (ili čak viši nivo) tačnosti, u odnosu na ove alternativne pristupe.

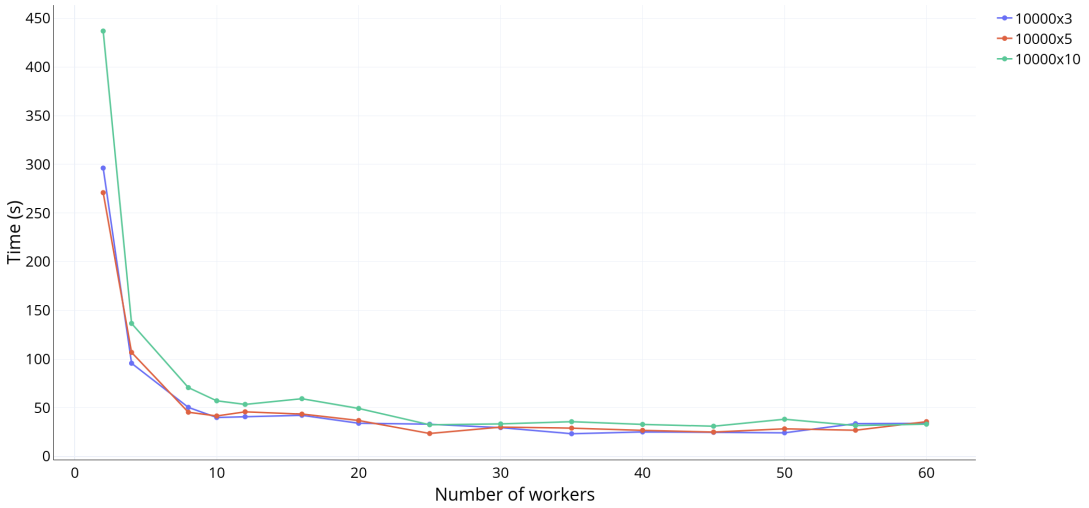
Kada je reč o većim skupovima podataka, algoritam takođe daje dobre rezultate, po pitanju tačnosti. Tabela A.5 prikazuje ovakve slučajeve, za razne sintetički generisane podatke. Evidentno je da naš algoritam postiže slične vrednosti za silhouette score, kao k-means. Grafički prikaz rezultata grupisanja za skup podataka veće dimenzije, prikazan je na slici A.9. Slika predstavlja t-SNE [148] embedovanje rezultata klasterovanja, gde se jasno vidi da je algoritam rasporedio tačke u klastere na adekvatan način.

Evaluacija skalabilnosti

U smislu evaluacije osobina skalabilnosti algoritma, koristićemo veće skupove, sintetički generisanih podataka. Slika A.10 prikazuje skaliranje algoritma za 3 različita skupa podataka, sa 10000 uzoraka i 3, 5 i 10 karaktersitika.



Slika A.9: t-SNE embedovanje za grupisanje sintetičkog skupa podataka veličine 1000x3



Slika A.10: Skaliranje algoritma nad skupom podataka sa 10000 uzoraka i 3, 5 i 10 karakteristika

Algoritam se ovde dobro skalira, kao što se i moglo očekivati. Optimalan broj čvorova se može odrediti za svaki slučaj, ali se primećuje da vreme izvršavanja ostaje blizu ove optimalne vrednosti za veći opseg različitog broja čvorova. Kada bismo dalje povećavali broj čvorova, u nekom momentu, bi vreme izvršavanja počelo rasti, na uobičajen način. Prikazani eksperimenti su pokazali da razvijeni algoritam pokazuje dobra svojstva skali-

Tabela A.6: Poređenje vremena izvršavanja (u sekundama) za AMA i ADMM-baziran metod konveksnog grupisanja

Skup podataka	AMA metoda	ADMM -bazirano konveksno klasterovanje 4 čvora	ADMM -bazirano konveksno klasterovanje 8 čvorova	ADMM -bazirano konveksno klasterovanje 10 čvorova	ADMM -bazirano konceksno klasterovanje 20 čvorova	ADMM -bazirano konveksno klasterovanje 25 čvorova
1000×3	2.8	12.65	**9.73**	13.79	11.45	12.24
5000×3	17.55	38.38	19.95	16.45	19.35	**14.15**
10000×3	45.46	96.0	51.17	40.84	37.3	**22.01**
5000×5	22.74	39.29	19.65	16.87	16.15	**14.37**
10000×10	100.2	136.76	72.63	59.33	50.23	**30.29**
200000×3	N.A.	–	–	–	–	**564.1**

ranja i da su prednosti paralelizacije evidentni. Vreme izvršavanja se smanjuje skoro linearno sa brojem čvorova u eksperimentalnom opsegu koji se razmatra.

Poređenje sa drugim pristupima za grupisanje podataka

Od posebnog je značaja napraviti poređenje razvijenog algoritma sa drugim, prihvaćenim pristupima za grupisanje podataka. Na taj način, stiče se jasniji uvid u karatkeristike razvijene metode, kako po pitanju tačnosti, tako i po pitanju performansi.

U radu [36], pojavljuju se dve nove metode za konveksno grupisanje. Mi ćemo iskoristiti efikasniju, da bismo je uporedili sa našim pristupom. U pitanju je metoda, bazirana na alternating minimzation algoritmu (AMA), koji smo već pomenuli prilikom evaluacije tačnosti. Metoda je dostupna u jeziku R.

Tabela A.6 prikazuje rezultate poređenja pomenute dve metode, AMA i konveksno grupisanje, bazirano na ADMM-u. Za manje obimne skupove podataka, AMA metoda može imati bolje performanse, što je ovde slučaj za skup podataka obima 1000×3 tačaka. AMA metoda, dostupna u R-u, je zapravo omotač oko C koda, što objašnjava ovaj trend. Pokretanje manjih primera na klasteru računara prirodno nije toliko isplativo, kao sa većim obimom podataka. Iz Tabele A.6 se može videti da za veće skupove podataka konveksno klasterovanje, bazirano na ADMM-u, ima znatno bolje performanse. Takođe, evidentno je da AMA, kao serijska metoda, poseduje ograničenje nad veličinom problema koji može da rešava. Ovakvo ograničenje se u paralelnim aplikacijama može prevazići povećanjem broja procesa, koji rešavaju problem.

U nastavku ćemo uporediti našu metoda i sa DBSCAN [117] metodom za grupisanje, s obzirom da je i DBSCAN metoda, kod koje nije potrebno unapred specificirati broj

Tabela A.7: Poređenje konveksnog grupisanja, baziranog na ADMM-u sa DBSCAN algoritmom

Skup podataka	br klast. ADMM	ADMM s.sc.	br klast. k-means	k-means s.sc.	DB-SCAN ϵ	br klast. DB-SCAN	DB-SCAN s.sc.
30×2	3	0.89	3	0.89	0.5	3	0.89
40×2	6	0.39	5	0.57	0.8	3	0.65
1000×3	8	0.76	8	0.76	2.5	8	0.75
5000×3	4	0.77	4	0.78	2.5	4	0.76
5000×5	5	0.62	4	0.75	5.0	4	0.75
10000×3	10	0.69	10	0.75	2.5	10	0.75

klastera. Tabela A.7 prikazuje rezultate ovog poređenja. Ove dve metode se razlikuju po načinu na koji se nose sa šumovima u podacima, ali kada postoji jasna struktuira grupisanja, obe metode imaju zadovoljavajuću preciznost.

U radu [115], pojavljuje se definicija metoda pod nazivom "Semismooth Newton based augmented Lagrangian method" (SSNAL), za rešavanje problema konveksnog grupisanja. Poređenjem prijavljenog vremena izvršavanja SSNAL metode nad skupom polusferičnih podataka obima 200000×3 i vremena izvršavanja našeg algoritma nad podacima istog obima za različit broj čvorova, dolazimo do zaključka da naša metoda može da se izvršava efikasnije. S druge strane, SSNAL metoda, podrazumeva veoma koristan, ali vremenski zahtevan proces pretprocesiranja, u cilju određivanja težinskih vrednosti. Naša metoda ne koristi ovakav vid pretprocesiranja. Implementirali smo izračunavanje težinskih vrednosti iz [115], i zaključili da ovaj proces može biti vremenski zahtevniji, nego izvršavanje našeg algoritma na kalsteru. Osim pomenutih testova, naš predloženi algoritam je testiran i nad realnim industrijskim podacima, dostupnim zahvaljujući saradnji na projektu I-BiDaaS [47]. Rezultati ovih testova konzistentni su sa rezultatima, dobijenim u okviru samog projekta. Ovim se direktno potvrđuje primenjivost algoritma u relanim slučajevima korišćenja.

Poređenje MPI i COMPSs paralelnih aplikacija

Unutar teze, koristimo 2 različite tehnologije za paralelizaciju algoritma, MPI i COMPSs. MPI je tehnologija nižeg nivoa, koja daje veći stepen kontrole programeru, ali je i mnogo zahtevnija za primenu. S druge strane, COMPSs je tehnologija višeg nivoa, gde je paralelizacija velikim delom transparenta i implicitna, tako da je razvoj implementacije

znatno olakšan. Kako bismo uporedili efikasnost pomenutih tehnologija, razvili smo MPI implementaciju našeg algoritma za konveksno klasterovanje, u Python-u. Na ovaj način, rešavamo isti problem, u istom programskom jeziku, nad istim podacima. Razlikuje se samo pristup paralelizaciji.

Tabela A.8 prikazuje rezultate ovog poređenja. Evidentno je da MPI postiže bolje performanse u odnosu na COMPSs. Nad nekim skupovima podataka, za manji broj čvorova, razlike u vremenima izvršavanja za ova dva pristupa nisu drastične, ali se za MPI vreme mnogo brže smanjuje, sa povećanjem broja čvorova. U nekim slučajevima iz Tabele A.8, MPI se izvršava 2 do 4 puta brže u odnosu na COMPSs. Ovo nije iznenađujuće, s obzirom na opisanu prirodu ovih tehnologija. Izbor jedne od ovih tehnologija zavisi od konkretnih potreba i prioriteta, u smislu pitanja, da li je akcenat na brzom i lakom razvoju paralelizacije ili na postizanju što veće brzine izvršavanja.

Tabela A.8: Poređenje vremena izvršavanja (u sekundama) konveksnog grupisanja baziranog na ADMM-u u MPI i COMPSs tehnologijama

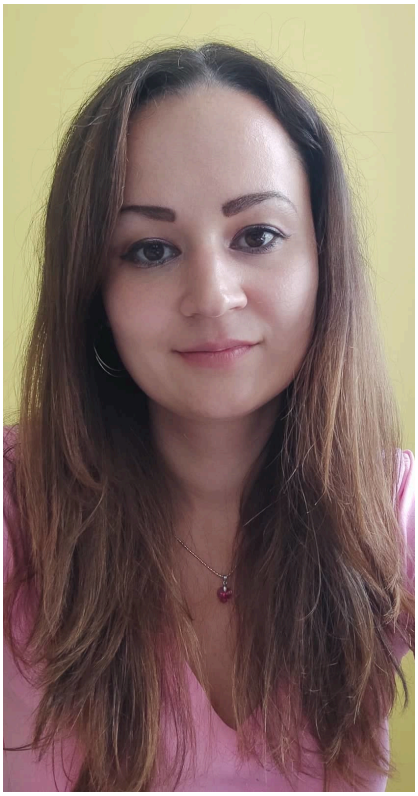
	4 čvora		**8 čvorova**		**10 čvorova**		**20 čvorova**		**25 čvorova**	
Skup podataka	**MPI**	**COMPSs**	**MPI**	**COMPSs**	**MPI**	**COMPSs**	**MPI**	**COMPSs**	**MPI**	**COMPSs**
1000×3	5.83	12.65	4.99	9.73	3.18	13.79	2.59	11.45	4.26	12.24
5000×3	34.17	38.38	15.36	19.95	10.85	16.45	5.49	19.35	10.05	14.15
5000×5	38.71	39.29	18.82	19.65	13.26	16.87	10.08	16.15	7.94	14.37
10000×3	95.13	96.0	29.89	51.17	29.08	40.84	13.31	37.3	15.53	22.01
10000×10	153.09	136.76	42.57	72.63	47.44	59.33	29.24	50.23	15.97	30.29

Zaključak

U fokusu ove teze, nalaze se razvoj i praktična evaluacija skupa paralelnih algoritama konveksne optimizacije. Teza obuhvata dva glavna pravca: razvoj i analizu klase metoda prvog i drugog reda, primarnog tipa, koje korsite razne tehnike sparsfikovanja komunikacije; i razvoj i analizu dualne, ADMM metode za paralelno konveksno grupisanje podataka. Temeljna evaluacija pomenutih metoda, omogućila je izvođenje nekih važnih zaključaka. Pre svega, pokazano je da su implementacije skalabilne i preimenjive na velike skupove podataka. Poređenje raznih aspekata metoda primarnog tipa, omogućilo je identifikaciju scenarija, u kojima određene metode imaju bolje performanse od drugih. Analizom konveksnog grupisanja, baziranog na ADMM-u, pokazano je da algoritam produkuje zadovoljavajući nivo tačnosti. Metode primarnog tipa predstavljaju proširiv skup, jer jednostavnom zamenom gradijenta i Hessian-a, mogu se implementirati slučajevi za dodatne funkcije cilja. Slično, ADMM pristup se može lako prilagoditi za različite slučajeve funckije cilja. Pokazano je da su razvijeni algoritmi primenjivi i u realnim scenarijima, nad realnim podacima, što otvara mogućnost njihove kasnije šire primene.

Appendix B

Short biography



Lidija Fodor was born on 25.07.1989. in Bačka Topola. She finished "Čaki Lajoš" elementary school in Bačka Topola in 2004. After that, she finished high school "Gimnazija i ekonomska škola Dositej Obradović", Bačka Topola, in 2008. She enrolled studies of information technologies at Faculty of Sciences, University of Novi Sad, in 2008 and finished her bachelor studies in 2011 with grade point average 9.85. In 2013, she also finished her master studies at Faculty of Sciences, with grade point average 10.0. After that, she started her PhD studies at Faculty of Sciences. She works as a teaching and research assistant and was involved in teaching on various courses, including: Databases, NoSql databases, High performance computing, Advanced programming, Information systems development, Information systems modelling. She coauthored six publications in international conferences and journals. She participated in national and international projects, including two H2020 projects: I-BiDaas - Industrial-Driven Big Data as a Self-Service Solution and Cyrene). Besides that, she attended numerous scientific visits across Europe, including PRACE "Parallel programming workshop" in Barcelona, Spain and "Introduction to high-performance machine learning", in Amsterdam, Netherlands, among others.

Novi Sad, 2022 Lidija Fodor

Овај Образац чини саставни део докторске дисертације, односно докторског уметничког пројекта који се брани на Универзитету у Новом Саду. Попуњен Образац укоричити иза текста докторске дисертације, односно докторског уметничког пројекта.

План третмана података

Назив пројекта/истраживања
Имплементација и анализа класе алгоритама за дистрибуирану конвексну оптимизацију: Евалуација перформанси и особина на практичним HPC кластерима (Implementation and analysis of a class of algorithms for distributed convex optimization: Performance evaluation and tradeoffs in practical HPC clusters)
Назив институције/институција у оквиру којих се спроводи истраживање
а) Природно-математички факултет, Универзитет у Новом Саду
Назив програма у оквиру ког се реализује истраживање
1. Опис података
1.1 Врста студије У овој студији нису прикупљани подаци.
2. Прикупљање података
3. Третман података и пратећа документација
4. Безбедност података и заштита поверљивих информација
5. Доступност података
6. Улоге и одговорност