The Role of the Network in Distributed Optimization Algorithms: Convergence Rates, Scalability, Communication/Computation Tradeoffs and Communication Delays

Konstantinos I. Tsianos

Doctor of Philosophy

Department of Electrical and Computer Engineering

McGill University

Montréal, Québec

July 2, 2013

A thesis submitted to McGill University in partial fulfillment of the requirements of the degree of Doctor of Philosophy

ACKNOWLEDGEMENTS

Pursuing a PhD is a long and mostly a lonely journey that in many ways defines you from there on. However, no matter how much effort and dedication you put into it, it is very hard to go on without the help of other people. Sometimes their importance is not obvious at the time, but now looking back there is several people that I would like to thank. I have to start with my parents and sister whom I have been away from for so long. They have always looked up for me and supported my choices and my efforts no matter what. I am also most grateful to my advisor Mike who has given me the chance to come to Montréal and provided me with the conditions I needed to work. It was a pleasure working together while observing him handle obstacles and uncertain situations along the way. Many thanks to Babis, Mihalis, Ioan and George who have been maybe my closest friends even though all of them are at least on flight away around the world. Having people you can rely on for their honest opinion about research and life is of uttermost importance. I am also very fortunate to have Véro in my life. She has been the highlight of my last year making the difficult last mile as smooth as possible. Finally, I should acknowledge and thank all of my friends, colleagues, professors and administrative staff whom I have interacted with all these years. I may have been closer to some than others but they have all been valuable to me in this adventure.

ABSTRACT

Many questions of interest in various fields ranging from machine learning to computational biology and finance require the solution of an optimization problem. Frequently such problems are classified as large scale in the sense that they involve complex computations over very large datasets. The increasing interest in distributed optimization algorithms is motivated by two main reasons. First, the problem complexity pushes today's processors to their limits and the need for distributed algorithms arises quite naturally. A second and more practical reason is that sometimes the data is collected in a distributed manner and transmitting it to a single location is either too costly or violates privacy. The starting point of this thesis is the simple realization that the main difference between a serial and a distributed algorithm is that in the latter, a processor needs to exchange messages over a network to access another processor's information. Network communication is in general less reliable and orders of magnitude slower than local disk accesses. Furthermore, in an arbitrary network topology to achieve global performance messages might need to travel over multiple hops. Finally, the hardware's capabilities also limit the ways in which a distributed algorithm may be implemented. All these factors highlight the important role of the network in distributed optimization algorithms. To understand that role we focus on the class of consensus-based distributed optimization algorithms. Those algorithms admit an elegant theoretical analysis while remaining easy to implement. In addition they tend to be scalable and robust to communication delays. The contributions of this work can be grouped into four important areas: 1) understanding the communication/computation tradeoff and its effect on scalability with the network size, 2) understanding the limitations of the network and the necessary features that distributed algorithms need to possess to be practical, 3) understanding the effects on convergence of network-induced communication delays and 4) understanding the theoretically achievable convergence

rates of distributed algorithms. These areas impact the design and deployment of any consensus-based distributed optimization algorithm.

ABRÉGÉ

Il y a une grande variété de domaines d' apprentissage automatique, de la biologie à la finance, où l'on doit résoudre des problèmes d'optimisation. Souvent ces problèmes impliquent des calculs complexes sur de vastes ensembles de données. Pour résoudre ces problèmes le développement d' algorithmes distribués est devenu très populaire pour deux raisons. Premièrement, la complexité des problèmes pousse les processeurs actuels à leurs limites. Naturellement, il devient essentiel d'utiliser des systèmes distribués. La deuxième raison est que la collecte des données est parfois distribuée, et il est difficile, coûteux ou en violation d' accords de confidentialité de transférer toutes les données au même endroit. La fondation de cette thèse est la réalisation simple que la grande différence entre un algorithme centralisé et un algorithme distribué est que ce dernier utilise un réseau pour permettre l' échange d' information d'un processeur à un autre. Généralement, la communication sur un réseau est moins fiable et beaucoup plus lente que l'accès d'information sur un disque local. De plus, pour une topologie de réseau arbitraire, la communication de messages nécessite plusieurs sauts. Finalement, les capacités matérielles aussi limitent les façons par lequelles les algorithmes distribués théoriques peuvent être mis en oeuvre. Tous ces facteurs, rappellent l'importance du réseau. Pour comprendre cette importance, nous nous concentrons, sur la classe des algorithmes de consensus pour optimisation distribuée. Ces algorithmes aussi possédent des analyses théoriques très élégantes tout en restant faciles a mettre en oeuvre. Aussi, ils sont robustes aux délais de communication et extensibles. Les contributions de cette thèse peuvent être classifiées selon les quatres catégories suivantes: 1) comprendre le compromis entre communication et calculs locaux, et l'extensibilité avec la taille du réseau, 2) comprendre les limites posées par le réseau aux fonctionnalités nécessaires que chaque algorithme distribué doit posséder en pratique, 3) comprendre les effets de délais de communication et les propriétés de convergence de ces algorithmes

en présence de délais, 4) comprendre les taux théoriques de convergence des algorithmes d'optimisation distribués. Tous ces domaines affectent la conception et le déploiement de chaque algorithme de consensus d'optimisation distribué.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS ii

ABSTRACT iii

ABRÉGÉ v

LIST OF FIGURES x

1 Introduction 1

1.1 Motivation 1
1.2 Distributed Algorithms 1
1.3 Contributions 3
1.4 Basic Problem Statement 5
1.5 Consensus-based Distributed Optimization 8
1.6 Publications 11
1.7 Notation 12

2 Background and Previous Work 13

2.1 Distributed Consensus 13
2.2 Communication Delays 17
2.3 Non-Linear Convex Optimization 18
2.4 Online and Stochastic Optimization 20
2.5 Distributed Optimization 26
2.5.1 Distributed Dual Averaging (DDA) 30
2.6 Distributed Programming Models 33

3 Communication/Computation Tradeoff and Scalability 36

3.1 Introduction 36
3.2 Scalability in Consensus-based Distributed Optimization 37
3.3 Quantifying the Tradeoff 38
3.4 Time model 40
3.5 Simple Case Analysis: Communicate at Every Iteration 41
3.6 General Case Analysis: Sparse Communication 43
3.6.1 Bounded Intercommunication Intervals 43
3.6.2 Increasingly Infrequent Communication 45
3.7 Experimental Evaluation 47
3.7.1 Application to Metric Learning 48
3.7.2 Nonsmooth Convex Minimization 49
3.8 Conclusions and Future Work 51

4 Parallelization at the Task Level . . . 53

4.1 Introduction . . . 53
4.2 Algorithms . . . 54
4.3 Communication/computation Tradeoff for Multiple Tasks . . . 55
4.4 Experimental Study . . . 58
4.4.1 Cluster description . . . 59
4.4.2 Solving Multiple Identical Jobs . . . 59
4.4.3 Sweeping a Problem Parameter . . . 62
4.4.4 Collisions between MIGD Random Walks . . . 63
4.5 Discussion . . . 64

5 Practical Consensus Algorithms . . . 67

5.1 Introduction . . . 67
5.2 Features of Distributed Consensus Algorithms . . . 68
5.2.1 Averaging . . . 70
5.2.2 One-Directional Communication . . . 72
5.2.3 Time varying protocols . . . 73
5.2.4 Semantics of Different Consensus Matrices . . . 74
5.3 Push-Sum Consensus . . . 75
5.4 Push-Sum Distributed Dual Averaging (PS-DDA) . . . 78
5.5 Implementation Remarks . . . 80
5.5.1 One directional communication . . . 80
5.5.2 Numerical instability . . . 80
5.5.3 Step-size de-synchronization . . . 81
5.5.4 Incoming message handler . . . 82
5.5.5 Communicator saturation . . . 82
5.6 Experimental Evaluation . . . 83
5.6.1 Benchmark Problem and Setup . . . 83
5.6.2 Doubly Stochastic Matrices cannot be maintained in Practice 83
5.6.3 Comparison with AllReduce . . . 84
5.7 Concluding Remarks and Future Work . . . 86
5.8 Proof of Theorem 5.1 . . . 88

6 Communication Delays . . . 94

6.1 Introduction . . . 94
6.1.1 Time Delayed Consensus . . . 95
6.1.2 Main Results . . . 96
6.2 Fixed Communication Delays . . . 97
6.2.1 Fixed Delay Model . . . 97
6.2.2 Stationary Distribution under Fixed Delays . . . 100
6.2.3 Convergence Rate under Fixed Delays . . . 102
6.2.4 Effect of Delays on Second Eigenvalue . . . 104
6.2.5 Distributed Optimization under Fixed Communication Delays . . . 108
6.2.6 Convergence of DDA with Fixed Edge Delays . . . 115
6.3 Time Varying Communication Delays . . . 116

6.3.1 Random Delay Model . . . 116
6.3.2 Convergence under Time-Varying Delays . . . 123
6.4 Push-Sum Consensus with Delays . . . 131
6.4.1 Consensus with Fixed Delays using Push-Sum . . . 132
6.4.2 Consensus with Random Delays using Push-Sum . . . 133
6.4.3 Convergence of Push-Sum Consensus with Random Delays 134
6.5 Simulation . . . 136
6.6 Concluding Remarks and Future Work . . . 137

7 Improved Convergence Rate for Distributed Convex Optimization . . . 139

7.1 Introduction . . . 139
7.2 A Time Optimal Algorithm for General Convex Functions . . . 139
7.3 Analysis and Proof of Theorem 7.1 . . . 141
7.4 Comments . . . 145

8 Distributed Strongly Convex Optimization . . . 146

8.1 Introduction . . . 146
8.2 Distributed Online Gradient Descent (DOGD) . . . 147
8.3 Analysis of DOGD . . . 148
8.3.1 Properties of Strongly Convex Functions . . . 149
8.3.2 The Lazy Projection Algorithm . . . 150
8.3.3 Evolution of Network-Average Quantities in DOGD . . . 151
8.3.4 Analysis of One Round of DOGD . . . 152
8.3.5 Bounding the Network Error . . . 154
8.3.6 Analysis of DOGD over Multiple Rounds . . . 155
8.3.7 Proof of Theorem 8.1 . . . 157
8.4 Extension to Stochastic Optimization . . . 158
8.5 Concluding Remarks . . . 159

9 Summary and Open Questions for Future Work . . . 161

Appendix A: Proof of equation (3.16) . . . 166

Appendix B: Proof of equation (3.19) . . . 167

Appendix C: Proof of Theorem 6.1 . . . 168

References . . . 172

LIST OF FIGURES

Figure page

3–1 Optimal number of processors in complete graphs. 50

3–2 Optimal number of processors in complete graphs, reduced communication. 50

3–3 Sparse communication comparison 51

4–1 DDA vs MIGD for identical tasks 60

4–2 Task interference 61

4–3 Cumulative time to solve multiple identical tasks with DDA and MIGD 61

4–4 Cumulative time to solve multiple different tasks 62

4–5 Cumulative number of collisions, 8 tasks 63

4–6 Cumulative number of collisions, 64 tasks 63

4–7 Average number of collisions per iteration 64

5–1 Illustration of optimization bias 71

5–2 Illustration of deadlock 73

5–3 Receiving unknown number of messages per iteration 74

5–4 Unbalanced graph topology 84

5–5 Loss of convergence due to unbalanced communication 85

5–6 Illustration of asynchronism 85

5–7 Illustration of slow node problem 87

6–1 Adding fixed delay 98

6–2 Canonical paths under fixed delays 106

6–3 Validation of spectral gap bound 109

6–4 Optimization under fixed delays 116

6–5 Effect of fixed delay on optimization bound 117

6–6 Adding random edge delay 118

6–7 Average consensus with random delays 136

CHAPTER 1
Introduction

1.1 Motivation

Distributed algorithms for optimization have received an increasing interest over the last two decades [10, 22]. Besides the academic motivation of how to design algorithms so that a group of computers communicating over a network can be employed to optimize an objective, there is an emerging great practical interest. Consider for example the challenge of understanding and analyzing a physical process. To apply a computational method, one starts by collecting measurements of what are perceived to be the important and defining features of the process' behaviour. The initial lack of knowledge of the complex physical world typically leads to a large number of measurements, large number of features, or both, and it is not uncommon to have extremely large datasets (see for example [1]). To process the data, there exists a large collection of algorithms falling under the general area of machine learning [11, 38]. Machine learning algorithms typically need to perform some kind of calculation on the data which goes beyond analyzing the data statistical properties. Very often the calculation involves or can be cast into an optimization problem where the quality of a model perhaps is measured by a cost function that must be minimized. When the optimization problems push current computers to their limits due to the sheer volume of data but also complexity of the calculations, the need for distributed optimization algorithms arises.

1.2 Distributed Algorithms

Even the fastest single computer could be perceived as "slow" when used to solve a computationally hard problem. Suppose that to solve a problem at hand in a reasonable amount of time, we would need a single computer that is n times more powerful than today's state-of-the art computers. Physical limitations prohibit us from building such a powerful single computer (at least at the moment). Distributed

and parallel computing are used as a surrogate in lack of powerful enough single processors. The goal is to use n "slow" computers that are readily available in place of a single processor, and achieve the same performance as what we would expect from an ideal (n times faster) single processor.

Although the end goal is the same (i.e., increased performance), the terms parallel and distributed computing should not be used interchangeably since they signify different approaches on using multiple computers to solve a problem. As the name suggest, parallelization implies tasks or processes that are running concurrently. This usually means that the tasks have minimal or ideally no interaction with one another. From this point of view, the challenges are on how to separate the problem data and how to wire the network connections between processors to facilitate the parallel task execution. The implementation design is conceptually hierarchical or even centralized. The computation is structured and proceeds in stages acting as synchronization points and a processor may have a different role or stay idle at every stage. The increased control over each node's role may offer opportunities for harnessing speedup through careful engineering at the system's level.

This thesis is concerned with distributed rather than parallel algorithms. The difference is that in distributed systems, the nodes are ideally indistinguishable. There is no master node and the same code is run on each processor. Typically, distributed algorithms are less structured and nodes coordinate by exchanging messages. This design has several advantages. Synchronization points are eliminated and such systems can run asynchronously requiring only minimal communication infrastructure. This offers increased robustness to individual node failures since no particular node is special, and scalability since in principle there is no master node that could be a bottleneck. However, achieving global performance relying only on local interactions becomes the main challenge. Furthermore, as is emphasized in Section 1.3, performance is critically dependent on the properties of the network over which messages are being exchanged.

In general, not all problems are suitable for employing a distributed solution. A good indicator that a problem is distributable is some form of separability. For example, when a dataset is large, it is natural to split the data into subsets and distribute those among the computation nodes. The split may be deliberate or the data itself may be collected in a distributed manner and transferring the data in a single location is too expensive. Splitting the data has a clean interpretation in empirical loss minimization problems which are ubiquitous in machine learning; how much an algorithm has learned from the data is measured in terms of a loss function which is taken to be the sum of loss over all data points. Splitting the data implies assigning a partial loss to each processor and the goal is to minimize the overall sum of losses over all nodes. Alternatively, it has been proposed to have a split in the feature space. In this case each node is responsible for learning only from a subset of the problem dimensions. Finally, it is possible to require the solution of multiple problems of the same nature, e.g., tuning a kernel parameter with a sweep. In that case, it might be beneficial to solve those problems in parallel on the same network harnessing parallelization at the task level.

1.3 Contributions

The discrepancy between an n-times faster single processor and an arrangement of n slow processors is the communication cost. The ideal fast processor is able to access all of the problem's parameters and data via local memory and disk accesses. Each of the n processors inevitably has to use communication over a network to access remote information residing in another processor. If communication over the network was free (i.e., instantaneous between any two processors independent of the message size), a team of n networked processors would be able to perfectly simulate a single processor which is n times faster. This fundamental realization illustrates the critical role of the network which is the central theme of this thesis. With distributed computing and the role of the network in mind, the contributions of this work are grouped into four directions:

1. Communication cost: How much time does it take to transfer information relative to the time it takes to perform local computations? In other words,

how much slower is the network relative to the processor speed? As shown in Chapters 3 and 4, what matters is not the speed of a processor or network latency in isolation, but rather the relative performance of the two.

2. Network infrastructure: Many distributed algorithms are described in a way that facilitates mathematical analysis. This is, to some degree, always necessary if the theoretical findings are to have any generality. However, when faced with an implementation problem, many of the implicit theoretical assumptions do not hold in practice. Chapter 5 investigates the discrepancies between theoretical assumptions and difficulties in practice. We identify averaging, one-directional communication and asynchronism as the three key issues that make the design of distributed algorithms more compatible to the physical limitations of a network. We also describe and analyze both theoretically and experimentally a distributed algorithm that possesses many desirable properties and has good performance.
3. Communication Delays: Even in highly controlled and well-maintained environments such as clusters, the network is a volatile medium whose performance is bound to fluctuate. We can not assume that all messages are delivered in a timely manner and in the order in which they were transmitted. Rather, it is almost certain that messages will be delayed. Chapter 6 studies the questions of how to model communication delays and then how to use delay models to reason about the effects of delay on convergence of distributed algorithms.
4. Convergence Rate Analysis: Convergence rates from gradient based optimization methods are relatively well understood for single processor algorithms. In a distributed setting, the network is the medium for accessing another processor's information in an attempt to still achieve global performance. It is important to understand what convergence rates are achievable by distributed algorithms for different classes of problems and how those rates relate to the

convergence rates for serial algorithms. Chapters 7 and 8 analyze two distributed algorithms that bridge some of the theoretical gaps between the optimal serial algorithm and their distributed counterparts by improving on existing convergence rate results (Chapter 7) and providing new algorithms with faster convergence rates (Chapter 8).

The thesis structure follows the above list in an attempt to keep the chapters self contained. Nevertheless, it should be clear that when deploying an actual system and implementing a specific algorithm all the issues studied here are present and will have to be dealt with. We need to design an algorithm that does not deadlock, is robust to communication delays and uses the right number of processors to fully exploit the network topology and communication/computation tradeoff for the hardware and problem at hand. In that sense, this thesis should not viewed as a linear document where each chapter is improving on the chapter that precedes it. Instead, the chapters are interconnected. In the rest of the introductory chapter we first give a general problem definition in Section 1.4. This problem will be referred to and appropriately customized at each chapter to highlight the particular research question studied in that chapter. Section 1.5 summarizes the important components found in all the algorithms of the consensus-based distributed optimization family that will be the focus of the thesis. The introduction concludes with an enumeration of the relevant publications that have been the results of this research.

1.4 Basic Problem Statement

Before proceeding in analyzing specific research questions it is important to define a prototype problem that needs to be solved. We describe this problem here. In each subsequent chapter, this problem may be customized of further refined to reflect the important relevant aspects in a specific context.

Most of the results apply to the general setting where we are interested in minimizing a separable convex objective of the form

$$F(\boldsymbol{w}) = \frac{1}{m}\sum_{j=1}^{m} f_j(\boldsymbol{w}) \tag{1.1}$$

where each f_j is assumed convex and $\boldsymbol{w} \in \mathcal{W} \subseteq \mathbb{R}^{d'}$ is the optimization variable. This framework fits well in the context of machine learning applications which are one of the motivations for this work. Assume that a dataset $\mathcal{D}$ is given in the form of vectors or measurements: $\mathcal{D} = \{\boldsymbol{x_1}^T, \boldsymbol{x_2}^T, \ldots, \boldsymbol{x_m}^T\}, \boldsymbol{x_i} \in \mathcal{X} \subseteq \mathbb{R}^d$. For a given $\boldsymbol{w}$, a data point $\boldsymbol{x}$ incurs a cost $l(\boldsymbol{w}, \boldsymbol{x}), l : \mathbb{R}^{d'} \mapsto \mathbb{R}$ which is parameterized by the data $\boldsymbol{x}$. The overall cost is taken to be

$$F(\boldsymbol{w}) = \frac{1}{m} \sum_{j=1}^{m} l(\boldsymbol{w}, \boldsymbol{x_j}). \tag{1.2}$$

The problem is to find a minimizer $\boldsymbol{w}^*$ of $F(\boldsymbol{w})$. As an example, consider a linear binary classification problem where we are looking to find a separating hyperplane $\boldsymbol{w}$ that accurately predicts the label of an unseen data point $\boldsymbol{x}$ as positive or negative. For example a commonly used loss function to quantify the misclassification error is the hinge loss which is used in support vector machines. For a data point $\boldsymbol{x} \in \mathbb{R}^d$ with label $y \in \{-1, 1\}$ the hinge loss at $\boldsymbol{w}$ is defined as

$$l(\boldsymbol{w}, (\boldsymbol{x}, y)) = \max(0, 1 - y\ \boldsymbol{w}^T \boldsymbol{x}). \tag{1.3}$$

Given a $\boldsymbol{w}$, a data point's label is estimated as $\tilde{y} = \text{sign}(\boldsymbol{w}^T \boldsymbol{x})$. Notice that if the estimated label matches the true label, the loss is zero.

From the description so far, the problem of minimizing (1.2) is underspecified. To make the discussion concrete we list the basic assumptions that will hold throughout the thesis.

Assumption 1.1. *Norm: From now on, by $\|\cdot\|$ we indicate the Euclidean norm $\|\cdot\|_2$.*

Generalizations to other norms are certainly possible and sometimes just an exercise. However they are not the focus of this thesis and we refer the interested reader to the related literature.

Assumption 1.2. *Bounded domain: We assume that $\boldsymbol{w} \in \mathcal{W} \subset \mathbb{R}^{d'}$ where $d \neq d'$ in general. We require that the domain $\mathcal{W}$ is closed and convex. Furthermore, the*

domain $\mathcal{W}$ has a radius $\sqrt{2}R$ for some positive constant R, i.e.,

$$\max_{\boldsymbol{w_1},\boldsymbol{w_2}\in\mathcal{W}} \|\boldsymbol{w_1}-\boldsymbol{w_2}\| \leq \sqrt{2}R. \tag{1.4}$$

Since the solution $\boldsymbol{w^*}$ should have a meaningful interpretation and practical use e.g., as the best linear classifier of our dataset, it is convenient to restrict the domain so that optimal solutions remain bounded.

Assumption 1.3. *Convexity: A cost function $l(\cdot,\cdot)$ is convex in its first argument*

$$l(\theta\boldsymbol{w_1}+(1-\theta)\boldsymbol{w_2},\boldsymbol{x}) \leq \theta l(\boldsymbol{w_1},\boldsymbol{x})+(1-\theta)l(\boldsymbol{w_2},\boldsymbol{x}) \tag{1.5}$$

for all $\boldsymbol{w_1},\boldsymbol{w_2}\in\mathcal{W},\boldsymbol{x}\in\mathcal{X},\theta\in[0,1]$. This implies that $F(w)$ is also convex.

Assumption 1.4. *Lipschitz continuity: We assume throughout that each cost function and thus $F(w)$ are Lipschitz continuous functions with constant L i.e.,*

$$|l(\boldsymbol{w_1},\boldsymbol{x})-l(\boldsymbol{w_2},\boldsymbol{x})| \leq L\|\boldsymbol{w_1}-\boldsymbol{w_2}\|, \forall \boldsymbol{w_1},\boldsymbol{w_2}\in\mathcal{W},\boldsymbol{x}\in\mathcal{X}. \tag{1.6}$$

It is a direct consequence of the previous assumption that each cost function has bounded sub gradients i.e.,

$$\|\nabla_{\boldsymbol{w}} l(\boldsymbol{w},\boldsymbol{x})\| \leq L, \forall \boldsymbol{w}\in\mathcal{W}. \tag{1.7}$$

Note that in general we need to use the dual norm for sub-gradients but under Euclidean distances from assumption 1.1 the two norms are the same.

Minimizing the objective (1.2) can be challenging if the dataset size m, data dimension d and domain dimension d', are very large. We thus focus on how to perform the minimization using a team of n processors. For this thesis we restrict to distributing the data over the processors since there is no conclusive evidence that distributing features is superior. We assume that the processors have the ability to communicate with one another but in general not every processor can send direct message to every other processor. To represent this restriction, the network of processors is viewed as a graph $G=(V,E)$ with processors being the nodes (so that $|V|=n$) and available direct communication channels being the edges E. The graph G may be directed or undirected. The only requirement for graph G is that

there is a directed path between any pair of nodes (i, j). Graphs with this property are called *strongly connected*.

We create a distributed optimization problem by splitting the data split evenly among the nodes. For simplicity we assume in the rest that the number of data points m is divided exactly by the number of processors n.

$$\underset{\boldsymbol{w}\in\mathcal{W}}{\text{minimize}}\ F(\boldsymbol{w}) = \frac{1}{m}\sum_{j=1}^{m} l(\boldsymbol{w}, \boldsymbol{x_j}) = \frac{1}{n}\sum_{i=1}^{n}\left(\frac{n}{m}\sum_{j=1}^{\frac{m}{n}} l(\boldsymbol{w}, \boldsymbol{x_{j|i}})\right) = \frac{1}{n}\sum_{i=1}^{n} f_i(\boldsymbol{w}), \tag{1.8}$$

where $\boldsymbol{x_{j|i}}$ we refer to the j-th datapoint in processor i's subset (i.e., $j|i = (i-1)\frac{m}{n} + j$). To unclutter notation the local objective is shown as $f_i(\boldsymbol{w})$ and the data dependence will be suppressed except for the sections where it is necessary to show it.

Observation: One way to arrive at a problem of the form (1.8) is through empirical loss minimization for data-driven machine learning problems. However, equation (1.8) describes a general separable convex minimization problem that does not have to depend on data. From that perspective, many of the conclusions in this thesis extend beyond machine learning problems.

Finally, at this point it is instructive to mention another difference between parallel and distributed algorithms. A parallel algorithm will typically maintain only one estimate $\boldsymbol{w}$ that is updated by computations done on all nodes. As already mentioned in Section 1.2, updating a single estimate may cause contention or introduce synchronization points in the system. A distributed algorithm will avoid maintaining a centralized estimate. Instead each node has a local estimate $\boldsymbol{w_i}$. The increased robustness and scalability is traded off for an iterative message passing algorithm such that each $\boldsymbol{w_i}$ approaches the true minimizer $\boldsymbol{w^*}$ asymptotically.

1.5 Consensus-based Distributed Optimization

As described in Section 1.4, in our distributed setting, each processor is a node in a graph G and is responsible for the i-th component $f_i(\boldsymbol{w})$ of a global objective that needs to be minimized. The thesis studies the family of *consensus-based*

distributed optimization algorithms that solve separable convex optimization problems. In this subsection we describe the major components that the reader should anticipate to encounter in all algorithms of this family. Based on the behaviour and interaction between those components, algorithms with different properties can be developed.

- Time: All of the algorithms considered in this work are iterative. Each node has an iteration counter and repeats a sequence of steps in each iteration, updating its local estimate. We do explicitly consider a relationship between iterations and actual wall clock time in Chapters 3 and 4. However, unless stated otherwise, we may use the term time to refer to discrete time and iteration number. Sometimes we will assume that the iteration counter is common for all nodes which will indicate an algorithm is synchronous. Alternatively, each node may maintain its own iteration counter in which case the algorithm is asynchronous. Unsurprisingly, synchronous algorithms are easier to analyze theoretically and many times we focus on the synchronous setting to prove theoretical results and gain insights.
- First-order Optimization: The focus is on general non-linear convex optimization. In lack of any specific assumptions on the structure of the components $f_i(\boldsymbol{w})$, we rely on gradient information to locate the optimum. Furthermore, without any extra smoothness assumptions, the reader should expect to encounter computations of subgradients of the local objectives. It should also not come as a surprise that to ensure convergence, a diminishing step-size has to be used.
- Constraints: Dealing with complex constraint sets $\mathcal{W}$ that may even differ among nodes, is not the main focus of this work. However, we do accommodate a case of simple convex constraints that still remains of great practical interest. We assume that the solution is sought inside a set $\mathcal{W}$ which is known to all nodes. To ensure that the estimates remain within $\mathcal{W}$, a projection operator is used to map estimates back into $\mathcal{W}$. In most cases a simple Euclidean

projection will suffice, i.e.

$$\Pi_{\mathcal{W}}[\boldsymbol{y}] = \operatorname{argmin}_{\boldsymbol{w}\in\mathcal{W}} \|\boldsymbol{w}-\boldsymbol{y}\|^2 . \tag{1.9}$$

For the analysis of computation time of different algorithms it will be assumed that the set $\mathcal{W}$ has a simple enough structure so that projections are easy to compute in the sense that they do not consume the bulk of the iteration's computation time.

- Coordination: We will be studying distributed algorithms with no special master processor. Instead, our algorithms will guarantee that all nodes approach the global optimum. To reach an agreement on the optimal solution, the processors rely on distributed consensus algorithms that involve communication. In a typical consensus step, a node exchanges messages with its neighbours on the graph and forms a convex combination of the received information. Under appropriate conditions on the weights assigned on the incoming messages, it is possible to guarantee convergence to consensus on a value among all the nodes on the graph. Depending on the approach, in our optimization context, the nodes may try to reach consensus on the optimum point directly, or instead, they may try to agree on the direction towards the optimum. The latter can greatly simplify the analysis.

All of the algorithms in the family of consensus-based distributed optimization make use of the above three components in one form or the other. A typical iteration will include computation in the form of a gradient calculation and a projection, and coordination through a consensus step where the main cost comes from communication. The algorithm designer typically has the freedom to decide the step size strategy, the consensus matrix weights, the communication strategy (e.g., nodes only send messages, or nodes both send and block until they receive a message), as well as the number of gradient and consensus steps to be executed within each iteration. As we will see, these design choices will play a role in the presence of different network effects.

1.6 Publications

Parts of the work presented in this thesis have been published in different venues. Below find a full list of the relevant publications:

1. Konstantinos I. Tsianos and Michael G. Rabbat, *The Impact of Communication Delays on Distributed Consensus Algorithms*, preprint arXiv:1011.2235, 2012 (submitted to IEEE Trans. on Automatic Control)
2. Konstantinos I. Tsianos and Michael G. Rabbat, *Simple Iteration-Optimal Distributed Optimization*, European Signal Processing Conference (EUSIPCO), 2013
3. Konstantinos I. Tsianos, Sean Lawlor and Michael G. Rabbat, *Communication/Computation Tradeoffs in Consensus-Based Distributed Optimization*, Neural Information Processing Systems (NIPS), pp 1952–1960, 2012
4. Konstantinos I. Tsianos Sean Lawlor and Michael G. Rabbat, *Consensus-Based Distributed Optimization: Practical Issues and Applications in Large-Scale Machine Learning*, 50th Allerton Conference on Communication, Control and Computing, pp 1543 - 1550, 2012
5. Konstantinos I. Tsianos and Michael G. Rabbat, *Distributed Strongly Convex Optimization*, 50th Allerton Conference on Communication, Control and Computing, pp 593 - 600, 2012
6. Konstantinos I. Tsianos, Sean Lawlor and Michael G. Rabbat, *Push-Sum Distributed Dual Averaging for Convex Optimization*, IEEE Conference on Decision and Control (CDC), pp 5453 - 5458, 2012
7. Konstantinos I. Tsianos and Michael G. Rabbat, *Distributed Dual Averaging for Convex Optimization under Communication Delays*, American Control Conference (ACC), pp 1067 - 1072, 2012
8. Konstantinos I. Tsianos and Michael G. Rabbat, *Distributed Consensus and Optimization under Communication Delays*, 49th Allerton Conference on Communication, Control and Computing, pp 974 - 982, 2011

1.7 Notation

We summarize here the most important symbols and notation. A conscious attempt was made to keep the same meaning for the same symbol. Unavoidable symbol overloading will be explicitly stated wherever necessary.

- $\mathcal{D}$: A dataset
- m : Dataset size
- $\boldsymbol{x_i}$: A data vector
- t : Discrete time variable, iteration
- T : Total number of iterations
- $\boldsymbol{w_i}(t)$: Estimate at node i at time/iteration t
- $\overline{\boldsymbol{w}}_{\boldsymbol{i}}(t)$: Local running average of $\boldsymbol{w_i}$
- $\boldsymbol{z_i}(t)$: Dual variable at node i at time/iteration t
- $\overline{\boldsymbol{z}}(t)$: Average of all $\boldsymbol{z_i}(t)$
- $\mathcal{W}$: Solution domain
- R : Domain $\mathcal{W}$'s radius
- L : Lipchitz constant, gradient magnitude bound
- $F(\boldsymbol{w})$: Objective to be minimized
- $f_i(\boldsymbol{w})$ or $f(\boldsymbol{w}, \boldsymbol{x_i})$: component of the objective $F(\boldsymbol{w})$ residing at node i
- $\nabla_{\boldsymbol{w}} f_i(\boldsymbol{w}), \boldsymbol{g_i}(\boldsymbol{w})$: Sub-gradient of f_i at $\boldsymbol{w}$
- $\partial_{\boldsymbol{w}} f_i(\boldsymbol{w})$: Sub-differential of f_i at $\boldsymbol{w}$
- $G = (V, E)$: A graph of V nodes connected over edges in E
- n : Number of nodes in a network
- P : $n \times n$ Consensus protocol conformant to G. Column,row or doubly stochastic matrix
- P_{ij} or $[P]_{ij}$: Element in row i and column j of matrix P
- $[P]_{i,:}$ and $[P]_{:,j}$: The i-th row and j-th column of P respectively

CHAPTER 2
Background and Previous Work

Before proceeding to the main results of the thesis, it is important to review the related literature and put this work in context. This chapter collects some necessary background and results. These results will be referenced wherever necessary in the rest of the thesis.

2.1 Distributed Consensus

This work studies consensus-based distributed optimization algorithms which have first been discussed in detail by Tsitsiklis, Bertsekas and Athans [10, 94]. Recently these algorithms have received a lot of attention in the context of large-scale optimization and machine learning, as well as in wireless sensor networks [30]. Consensus-based distributed optimization algorithms generally interleave a local optimization step with an iteration of distributed consensus to coordinate or synchronize values across the network. We review the distributed optimization literature in Section 2.5. Here we start with distributed consensus and averaging.

Assume each node $i \in V$ in a strongly connected network $G = (V, E)$ of $|V| = n$ nodes holds a value z_i. We stack the initial values in a vector $\boldsymbol{z}(0) = (z_1, \ldots, z_n)^T$. The general consensus problem asks for a distributed algorithm such that the nodes of the network exchange messages with their neighbours and update their state to reach consensus: $\boldsymbol{z}(t) \to c\mathbf{1}$ as $t \to \infty$. In other words, we want the nodes to agree on a common value c using only local communication. As briefly mentioned in Section 1.5, to reach consensus, each node transmits its local state to its neighbours in G. A node updates its state by forming a convex combination of the incoming messages. If we arrange the weights that node i uses to linearly combine incoming information from its neighbours into the row of a matrix P, consensus can be achieved by

repeating the iteration

$$\boldsymbol{z}(t) = P\boldsymbol{z}(t-1) = P^t\boldsymbol{z}(0) \tag{2.1}$$

for appropriately chosen matrices P. From now on we will refer to P as the *consensus matrix*. If $P_{ij} > 0$, that means that node j transmits information to node i. If $P_{ij} = 0$ then there is no direct communication from j to i. Consequently, P conforms exactly to the structure of the graph G. Equation (2.1) in general represents a broadcast protocol where each node broadcasts its value to its neighbours. Each node as a receiver forms a linear combination of the incoming messages. An important special case of distributed consensus is *averaging*, where we demand that the limit value c is equal to the average of the initial values $\frac{1}{n}\sum_{i=1}^{n} z_i(0)$. Chapter 5 explains in detail why averaging is critical for solving optimization problems.

There is a rich literature on distributed consensus and averaging (see [30, 63] and references therein) and a lot of effort has been devoted to analyzing the rate of convergence to the consensus value through the properties of the consensus matrix P [13, 65].

When $P(t) \equiv P$ for all t, we have a time-homogeneous consensus protocol, which must be implemented using synchronous, blocking communications so that each node receives a message from all of its neighbours before computing the update for each iteration. Such protocols admit fairly straightforward convergence analysis. To reach consensus it suffices to use a matrix P whose rows sum to one i.e., $P\mathbf{1} = \mathbf{1}$. Such a matrix is called row stochastic and corresponds to the transition matrix of a Markov chain. As is emphasized in [15], we can study consensus through the spectral properties of the corresponding Markov chain. For example, recall that for this work P conforms to a strongly connected graph G so that every two processors in our network can exchange information over a path of G. In the Markov chain literature, such a chain P is called *irreducible*. Let us call $\boldsymbol{\pi}$, the unique stationary

distribution of P i.e., $\boldsymbol{\pi}^T P = \boldsymbol{\pi}^T$. If P represents a reversible Markov chain[1] , (e.g., if G is an undirected graph), convergence is established by Perron-Frobenius theory [73]. Specifically, one can verify that 1 is the unique largest eigenvalue of P and thus P converges to a rank 1 matrix all of whose rows are equal to the chain's unique stationary distribution $\boldsymbol{\pi}$. Furthermore convergence is geometric at a rate $O(|\lambda_2(P)|^t)$, where $\lambda_2(P)$ is the second largest eigenvalue of P (see Theorem 4.2 in [73]). It is easy to show that if in addition P is doubly stochastic (i.e., if the columns sum up to one too) then P is an averaging matrix since its stationary distribution can be shown to be uniform and $P^t \to \frac{1}{n}\mathbf{1}\mathbf{1}^T$.

More generally, if P is not reversible (e.g., if G is directed) then the theory for reversible chains can still be applied to obtain bounds on the convergence rate by first *reversibilizing* the chain (akin to a symmetrizing transformation); see, e.g., [34]. In general, the reversibilization transforms require that P be strongly aperiodic (all diagonal elements satisfy $P_{i,i} \geq 1/2$). When this is not true, it is common to study a lazy version of the corresponding chain, $\frac{1}{2}(I + P)$. The lazy chain converges no more than two times slower than the original chain due to the presence of self-loops. We will see an application of these techniques in Chapter 6 when we model communication delays. We remark that more recent results for characterizing the mixing times[2] of non-reversible Markov chains with zero minimum holding probability[3] may lead to tighter results since they do not use the lazy chain [53]. Making use of these results is an interesting direction for future work.

[1] A chain is reversible iff $\pi_i P_{i,j} = \pi_j P_{j,i}$ for all i and j.

[2] The mixing time of a Markov chain is the number of iterations required for the chain to be sufficiently close to its stationary distribution. Intuitively, if a chain P has stationary distribution $\boldsymbol{\pi}$, the probability of the Markov chain being at state i after mixing time number of steps is close to π_i.

[3] The holding probability for state $i = 1, \ldots, n$ of a Markov chain is the probability that after taking a step from state i the chain finds itself returning to state i. The holding probabilities are the diagonal elements of the Markov chain matrix P.

It is of great practical interest to also study time-varying versions of (2.1) by allowing the consensus matrix to change at every iteration using time varying matrices $P(t)$. For example this allows us to implement and study asynchronous protocols where each node decides when to transmit independently. It is generally more difficult to establish convergence properties for time-varying updates because one needs to analyze products of time-varying stochastic matrices. For example, for time-varying protocols, [82] provides necessary conditions under which convergence is achievable while [66] characterizes the expectation and variance of the consensus value. In addition, [4] shows that using asynchronous broadcasts and forming convex combinations of incoming information guarantees convergence to the average only in expectation.

If the matrices $P(t)$ are drawn independently and identically distributed (i.i.d.) according to a known distribution, then the bounds mentioned above for time-homogeneous protocols can be applied to the expected update matrix $\mathbb{E}[P(t)]$. When all matrices $P(t)$ are row stochastic, the process (2.1) gives rise to a backward product

$$\boldsymbol{z}(t) = P(t)P(t-1)\cdots P(1)\boldsymbol{z}(0) \stackrel{\text{def}}{=} T(1,t)\boldsymbol{z}(0). \tag{2.2}$$

Convergence properties of backward products of stochastic matrices are typically obtained by establishing weak ergodicity [73]; i.e., that $|[T(r,t)]_{i,s} - [T(r,t)]_{j,s}| \to 0$ as $t \to \infty$ for all i, j, s, and r, where $[T]_{i,j}$ denotes the entry of the matrix T in row i and column j. For time varying Markov chains, convergence and convergence rates are traditionally obtained using coefficients of ergodicity which are metrics that characterize the distance from convergence of the backward product based on the metrics of the individual product terms. Related to that are also the scrambling properties of the matrices in the process $P(t)$, i.e., the number of steps necessary for any two pairs of nodes to exchange information. For more details consult [20,73,82]. It is worth mentioning that establishing convergence is significantly harder and the derived rates tend to be very conservative. More recently, the PhD thesis of Touri [84] provides rates of convergence for backward products using a suitable

Lyapunov function and the *infinite flow property* which ensures that the graph formed by putting edges between nodes that exchange information infinitely often, is connected. In general, these rates of convergence are also pessimistic, involving a worst-case analysis. For example, when packets can be delayed, the bounds depend on the largest possible delay.

Besides the simple iteration (2.1), as we will emphasize in Chapter 5 it is possible to achieve average consensus using column stochastic matrices P. This might be surprising at first since a column stochastic matrix P alone does not preserve the sum of the value in (2.1). However, this complication is overcome with the exchange of extra scalar information. An example of this family of algorithms is Push-Sum [49]. If we allow the consensus matrix to vary with time, we end up with what is called a forward product of column stochastic matrices and is analogue to (2.2). See [7, 31, 73] for some convergence analysis in the time varying case.

2.2 Communication Delays

As explained in the introduction, communication delays are almost always expected to occur for any real networked system and there exist studies of delays in various contexts. For applications in partial differential equations, distributed control and multi-agent coordination see [12, 71] and [64, 74] which analyze continuous-time delay models where all messages incur the same constant delay. In this work we are interested in the effect of communication delays on consensus algorithms and distributed optimization where both computation and communication happen in rounds and take a significant amount of time. For this reason we focus on discrete-time models. An early treatment of delays in discrete-time distributed averaging algorithms can be found in [10], where it is proved that convergence is not guaranteed if delays are unbounded. An analysis of conditions for convergence in the presence of delays is given in [13]. Closer to our work are [21], [56] and [95] which model delays in discrete time for consensus problems by augmenting the state space with delay nodes. However, in [21] the value to which the consensus algorithm asymptotically converges is not characterized. The model in [95] accumulates all the delayed information in a single delay node and does not allow for delivery of

messages out of order. The model in [56] has the same expressive power as the random delay model introduced in Chapter 6, although the equation describing the consensus dynamics in [56] does not allow for receiving multiple messages from the same sender in one iteration.

2.3 Non-Linear Convex Optimization

The question of finding the minimizer of a generally non-linear convex objective $F(\boldsymbol{w})$ over a convex set $\mathcal{W}$, has been studied for many decades [8]. Several textbooks such as [17] and [62] cover the topic in detail focusing either on the engineering aspect of how to express a given task as an optimization problem, or on what algorithms and technical issues must be addressed when searching for a solution. In the rest of this thesis we will focus on first-order gradient-based methods, and we will not consider other alternatives such as second order and interior point methods. The reason is that gradient-based methods are very simple and can be extended to the distributed setting quite naturally. Furthermore, gradient methods tend to be very efficient in terms of computational overhead and memory requirements, and they remain competitive when the problem is high-dimensional and the objective does not have any special structure that can be exploited.

The basic gradient descent algorithm for solving unconstrained problems (where $\mathcal{W} = \mathbb{R}^{d'}$), performs updates

$$\boldsymbol{w}(t+1) = \boldsymbol{w}(t) - a(t)\nabla_{\boldsymbol{w}} F(\boldsymbol{w}(t)) \tag{2.3}$$

where $a(t) > 0$ is a step-size and $\nabla_{\boldsymbol{w}} F(\boldsymbol{w}(t))$ is the gradient of the objective at the most recent estimate. The two relevant questions are: under what conditions for the objective and the step size sequence does (2.6) converge to the true minimizer $\boldsymbol{w}^*$ of $F(\boldsymbol{w})$, and how many iterations are needed to guarantee that $\boldsymbol{w}(t)$ is an ϵ-accurate solution i.e., after how many iterations the error

$$F(\boldsymbol{w}(t)) - F(\boldsymbol{w}^*) \tag{2.4}$$

is less than some positive ϵ. An excellent reference for answering these questions, depending on the properties of the objective function, is [60].

For the gradient method, if $F(\boldsymbol{w})$ is smooth with $L-$Lipschitz continuous gradients then convergence is guaranteed at a rate

$$F(\boldsymbol{w}(T)) - F(\boldsymbol{w}^*) = O\left(\frac{1}{T}\right) \tag{2.5}$$

if we use the optimal constant step size $a(t) = \frac{1}{L}$ where L is the Lipschitz constant of the objective [60].

In general we will be interested in solving non-smooth problems. For example the hinge loss from equation (1.3), is not differentiable at zero. For non-differentiable problems, to find a minimizer within a constraint set $\mathcal{W}$ we can use the *Projected Subgradient* method [8] that uses sub-gradients in place of gradients:

$$\boldsymbol{w}(t+1) = \Pi_{\mathcal{W}}\left[\boldsymbol{w}(t) - a(t)\nabla_{\boldsymbol{w}} F(\boldsymbol{w}(t))\right] \tag{2.6}$$

Here $\Pi_{\mathcal{W}}[\cdot]$ is a projection operator such as (1.9) and $a(t) > 0$ is a step-size sequence. If the objective is L-Lipschitz continuous (see definition in Section 1.4), then using a diminishing step size sequence $a(t) = O(\frac{1}{\sqrt{t}})$, the projected sub-gradient algorithm converges as (Theorem 3.2.2 in [60])

$$F(\boldsymbol{w}(T)) - F(\boldsymbol{w}^*) = O\left(\frac{1}{\sqrt{T}}\right) \tag{2.7}$$

if we decide a priori the number of iterations T that the algorithm shall execute. Notice that this rate is significantly slower, indicating that non-smooth problems are much harder to solve. The rate is improved however, if the objective has more structure.

Definition 2.1. *A function $F(\boldsymbol{w})$ is σ-strongly convex if there exists a constant $\sigma > 0$ such that for all $\theta \in [0,1]$ and all $\boldsymbol{u}, \boldsymbol{w} \in \mathcal{W}$,*

$$F(\theta\boldsymbol{u} + (1-\theta)\boldsymbol{w}) \leq \theta F(\boldsymbol{u}) + (1-\theta)F(\boldsymbol{w}) - \frac{\sigma}{2}\theta(1-\theta)\left\|\boldsymbol{u} - \boldsymbol{w}\right\|^2. \tag{2.8}$$

If the objective to be minimized is strongly convex, then as explained in [39,76], the projected subgradient algorithm with a step size sequence $a(t) = O(\frac{1}{T})$ converges

at a rate

$$F(\boldsymbol{w}(T)) - F(\boldsymbol{w}^*) = O\left(\frac{\log(T)}{T}\right). \tag{2.9}$$

The intuition is that strong convexity ensures that the objective does not flatten out too much near $\boldsymbol{w}^*$ and thus the sub-gradient algorithm can keep approaching the minimum quickly.

We conclude this section with another important algorithm attributed to Nesterov [61] called dual averaging. The name comes from the use of a dual variable $\boldsymbol{z}(t)$ which at every iteration is the cumulative sum of past gradients which are vectors that lie in the dual space $\mathcal{W}^*$. At each iteration the algorithm performs the updates

$$\boldsymbol{z}(t+1) = \boldsymbol{z}(t) + \partial_{\boldsymbol{w}} F(\boldsymbol{w}(t)) \tag{2.10}$$

$$\boldsymbol{w}(t+1) = \operatorname{argmin}_{\boldsymbol{w}\in\mathcal{W}} \left[\boldsymbol{w}^T \cdot \boldsymbol{z}(t+1) + \frac{1}{2a(t)}\psi(\boldsymbol{w})\right]. \tag{2.11}$$

The second equation uses a 1-strongly convex function $\psi(\boldsymbol{w})$ to compute a projection. Observe that if $\psi(\boldsymbol{w}) = \frac{\boldsymbol{w}^T\boldsymbol{w}}{2} = \frac{\|\boldsymbol{w}\|}{2}$ then the step reduces to a Euclidean projection and the algorithm is referred to as *lazy projection* in [103]. Using a step size sequence $a(t) = O\left(\frac{1}{\sqrt{t}}\right)$, dual averaging converges at the same optimal rate $F(\boldsymbol{w}(T)) - F(\boldsymbol{w}^*) = O\left(\frac{1}{\sqrt{T}}\right)$ for general non-differentiable, $L-$Lipschitz continuous functions. Finally, it is worth mentioning that the convergence results given above are still achievable when the gradients are noisy and the projection operations are imprecise [72].

2.4 Online and Stochastic Optimization

Online and stochastic optimization considers scenarios where an algorithm is not given all the data in advance [75]. Chapter 8 studies convergence rates of distributed algorithms for such problems so we review some of the recent related literature here. Again we will focus on first order gradient methods. We first define the related problem of online prediction and then proceed to give the convergence results for online optimization.

An online prediction algorithm [23] receives convex and L-Lipschitz continuous cost functions $f_1(\boldsymbol{w}), f_2(\boldsymbol{w}), \ldots$ sequentially one at a time. In general, the cost functions do not have to be related in anyway. With time, the algorithm produces a sequence of estimates $\boldsymbol{w}(1), \ldots, \boldsymbol{w}(T)$ and accrues cost $\sum_{t=1}^{T} f_t(\boldsymbol{w}(t))$. The goal is to accumulate as little cost as possible. To put this abstract setting in context, consider the case where the algorithm needs to process a dataset and the data points are arriving one at a time. In this case, the cost functions may be parameterized by the data, i.e., the algorithm receives $f(\boldsymbol{w}, \boldsymbol{x}(1)), f(\boldsymbol{w}, \boldsymbol{x}(2)), \ldots$ where the temporal dependence is encoded in the data index. This situation could occur either when the data is collected in real time, or deliberately enforced when the dataset is large and is processed sequentially to save memory. An important special case is when the data, and thus the costs, are drawn randomly from an unknown distribution. We call this a stochastic online prediction problem.

The goal in online prediction is to produce a sequence of estimates $\{\boldsymbol{w}(t)\}_{t=1}^{T}$ that accumulates as little cost as possible. To quantify this notion, rather than an objective function the notion of *regret* is used. Consider an offline algorithm that has access to all the data, or more generally, all the cost functions in advance. Based on that privileged information the offline algorithm is allowed to choose the best possible fixed estimate $\boldsymbol{w}^*$. Then, an online algorithm is evaluated based on how much it regrets using its sequence of estimates rather than $\boldsymbol{w}^*$ at every step. Formally for a serial online prediction algorithm regret is defined as

$$R_1(T) = \sum_{t=1}^{T} f(\boldsymbol{w}(t), \boldsymbol{x}(t)) - \min_{\boldsymbol{w} \in \mathcal{W}} \sum_{t=1}^{T} f(\boldsymbol{w}, \boldsymbol{x}(t)). \tag{2.12}$$

This comparative performance is fair in the sense that either we have perfect information but commit to one single optimal predictor $\boldsymbol{w}$, or information is revealed progressively and the predictor $\boldsymbol{w}$ is allowed to change accordingly. Finally, observe that if we analyze an online algorithm but feed the algorithm the same data at every iteration then the resulting rates describe the performance of an offline algorithm

that uses that same data so the reader should recognize here many of the results of the previous section.

Turning now to optimization, in the stochastic case, we can retrieve the optimization setting introduced in Section 1.4. Suppose the goal is to minimize an objective $\bar{F}(\boldsymbol{w}) = \mathbb{E}_{\mathcal{D}}[f(\boldsymbol{w}, \boldsymbol{x})]$ where the expectation is taken with respect to an unknown fixed distribution that generates the data $\boldsymbol{x}$. Using a finite set of T data samples, we can minimize minimize $F(\boldsymbol{w}) = \frac{1}{T}\sum_{t=1}^{T} f(\boldsymbol{w}, \boldsymbol{x}(t))$. The minimizer of $F(\boldsymbol{w})$ approaches the minimizer of the true objective $\bar{F}$ as T grows [59, 76]. This approach is called empirical loss minimization since we use the finite set of observed data (empirical data) to approach the minimum of an objective that is expressed as an expectation and for which we do not have an analytic expression. Notice that with T finite, a subgradient $\partial_{\boldsymbol{w}} f(\boldsymbol{w})$ at some point $\boldsymbol{w}$ is only a noisy unbiased estimate of the true subgradient $\partial_{\boldsymbol{w}} \bar{F}(\boldsymbol{w})$. Consider now the running average $\overline{\boldsymbol{w}}(T) = \frac{1}{T}\sum_{t=1}^{T} \boldsymbol{w}(t)$. From the convexity assumption, the expected optimization error is bounded by the average expected regret:

$$\mathbb{E}_{\mathcal{X}}[F(\overline{\boldsymbol{w}}(T)) - F(\boldsymbol{w}^*)] \leq \frac{1}{T}\mathbb{E}_{\mathcal{X}}[R_1(T)] \tag{2.13}$$

where the expectation is taken with respect to the unknown data distribution (see [100]). In the rest of this section we report some recent results on convergence rates for stochastic optimization as well as the extension to the distributed setting. To describe all of the results in an optimization form, we express the bounds in terms of the average or expected average regret depending on whether the bound refers to a general online or a stochastic problem:

$$\Delta_1(T) = \frac{1}{T}R_1(T) \quad \text{or} \quad \mathbb{E}[\Delta_1(T)] = \frac{1}{T}\mathbb{E}[R_1(T)] \tag{2.14}$$

where the expectation is taken with respect to the unknown data distribution.

The question, of course, is how quickly does $\Delta_1(T)$ or $\mathbb{E}[\Delta_1(T)]$ converge to zero. The answer depends on the convexity and smoothness properties of the objective. When the cost functions $f(\boldsymbol{w}, \boldsymbol{x})$ are convex in $\boldsymbol{w}$ for every $\boldsymbol{x} \in \mathcal{X}$ and the

sub-gradient $\nabla_{\boldsymbol{w}} f(\boldsymbol{w}, \boldsymbol{x})$ is bounded uniformly then stochastic sub-gradient descent:

$$\boldsymbol{w}(t+1) = \Pi_{\mathcal{W}} \left[\boldsymbol{w}(t) - a(t) \partial_{\boldsymbol{w}} f(\boldsymbol{w}(t), \boldsymbol{x}(t)) \right] \tag{2.15}$$

achieves the rate $\Delta_1(T) = \Theta\left(\frac{1}{\sqrt{T}}\right)$ and this algorithm is optimal [14, 39, 103]. The same rate can also be achieved by an online or stochastic implementation of the dual averaging algorithm (2.10) using subgradients [61, 100]. In both cases, the rate is achieved using a step size sequence $a(t) = O\left(\frac{1}{\sqrt{t}}\right)$.

There has also been a lot of recent work on the case where cost functions are strongly convex (recall the definition (2.1)). In this case, the rate for general online problems can be improved to $\Delta_1(T) = \Theta\left(\frac{\log(T)}{T}\right)$ using either stochastic subgradient or dual averaging methods [39, 48, 67, 76, 77, 100]. There are several variants of the basic algorithms we have seen that achieve these rates. Typically, for a σ-strongly convex objective (see definition 2.1) one can get the optimal rate using algorithm (2.15) with a step size sequence $a(t) = \left(\frac{1}{\sigma t}\right)$. If the convexity parameter σ is not known in advance one can achieve the same rates using adaptive methods [5]. Interestingly, for strongly convex objective functions the general online and stochastic scenarios can be solved faster. In particular, [40] describes an algorithm that achieves a rate $\mathbb{E}\left[\Delta_1(T)\right] = O\left(\frac{1}{T}\right)$ for stochastic optimization which is faster by a logarithmic factor.

Example: Training a Classifier. It is instructive to mention an example application for the reader's convenience. Suppose the t-th data point $\boldsymbol{x}(t) \in \mathcal{X} \subseteq \mathbb{R}^d$ is drawn i.i.d. from an unknown distribution $\mathcal{D}$, and let $f(\boldsymbol{w}, \boldsymbol{x}(t))$ denote the cost of the t-th data point with respect to a particular estimate $\boldsymbol{w}$. One would like to find the point $\boldsymbol{w}$ that minimizes the expected loss $\mathbb{E}_{\mathcal{D}}[f(\boldsymbol{w}, \boldsymbol{x})]$, possibly with the constraint that $\boldsymbol{w}$ be restricted to a space $\mathcal{W}$. Since the data are drawn i.i.d., as $T \to \infty$, the objective $F(\boldsymbol{w}) = \frac{1}{T} \sum_{t=1}^{T} f(\boldsymbol{w}, \boldsymbol{x}(t))$ approaches $\mathbb{E}_{\mathcal{D}}[f(\boldsymbol{w}, \boldsymbol{x})]$, and so if the data stream is finite this motivates minimizing the empirical loss $F(\boldsymbol{w})$. The reason the costs are processed sequentially could be either because the data points are collected and arrive in real time one after another, or because we deliberately process them serially to keep memory and computation cost per iteration low. For a

specific example, consider the problem of training an SVM classifier using the hinge-loss with ℓ_2 regularization [77] that we saw earlier (see (1.3)). In this case, the data stream consists of pairs $\{\boldsymbol{x}(t), y(t)\}$ such that $\boldsymbol{x}(t) \in \mathcal{X}$ and $y(t) \in \{-1, +1\}$. The goal is to minimize the misclassification error as measured by the ℓ_2-regularized hinge loss. Formally, we wish to find the $\boldsymbol{w}^* \in \mathcal{W} \subseteq \mathbb{R}^d$ that solves

$$\text{minimize}_{\boldsymbol{w}\in\mathcal{W}} \quad \frac{\sigma}{2}\|\boldsymbol{w}\|^2 + \frac{1}{m}\sum_{t=1}^{m} \max\{0, 1 - y(t)\boldsymbol{w}^T\boldsymbol{x}(t)\} \tag{2.16}$$

which is σ-strongly convex[4] . For problems like (2.16) using a single-processor stochastic gradient descent algorithm, one can achieve $\frac{R_1(T)}{T} = O\left(\frac{\log T}{T}\right)$ [77] or $\frac{R_1(T)}{T} = O\left(\frac{1}{T}\right)$ [40] by using different update schemes.

Let us now turn our attention to the distributed case where we have a network of n processors which all receive data and update their estimates over time. Specifically, processor i receives data $\boldsymbol{x}_i(1), \boldsymbol{x}_i(2), \ldots$, and updates a local variable $\boldsymbol{w}_i(1), \boldsymbol{w}_i(2), \ldots$. In this distributed setting, when each processor receives T data points, performance is measured in terms of the cumulative regret over all processors,

$$R_n(T) = \sum_{i=1}^{n}\sum_{t=1}^{T} f(\boldsymbol{w}_i(t), \boldsymbol{x}_i(t)) - \arg\min_{\boldsymbol{w}\in\mathcal{W}} \sum_{i=1}^{n}\sum_{t=1}^{T} f(\boldsymbol{w}, \boldsymbol{x}_i(t)). \tag{2.17}$$

Furthermore, the processors communicate and coordinate their actions to satisfy three objectives.

1. The processors should collectively minimize the regret $R_n(T)$.
2. The processors should agree on the optimal point $\boldsymbol{w}^*$, and hence we require some form of consensus.
3. The distributed algorithm should be scalable in the sense that the regret $R_n(T)$ should scale well with the number of processors n; i.e., the overhead involved in coordinating the processors should not grow substantially as n increases.

[4] Although the hinge loss itself is not strongly convex, adding a strongly convex regularizer makes the overall cost function strongly convex.

With respect to the third objective, if the n processors act independently then there will be no overhead for coordination, but the regret will inevitably grow at a rate of $nR_1(T)$. On the other hand, processing the entire data stream

$$\boldsymbol{x_1}(1), \ldots, \boldsymbol{x_n}(1), \boldsymbol{x_1}(2), \ldots, \boldsymbol{x_n}(2), \ldots \tag{2.18}$$

sequentially with a single processor would give a regret of $R_1(nT)$. In light of the discussion in Sections 1.2 and 1.3 any reasonable distributed algorithm that uses some coordination will have a regret $R_n(T)$ for which

$$R_1(nT) \leq R_n(T) \leq nR_1(T). \tag{2.19}$$

Observe that for distributed algorithms the convergence rate question has two aspects. The first is to ensure that a distributed algorithm converges with the number of iterations T, at the same rate as its serial counterparts. The second aspect is to optimally utilize the power of n processors and scale with the network size. We say that a distributed online stochastic prediction algorithm achieves optimal scaling if $R_n(T) = \Theta(R_1(nT))$ since in that case the distributed regret with n processors is of the same order as the serial regret of a single processor that sees the same amount of data.

Contrary to the serial case, the existing results for distributed/decentralized algorithms are less complete. Both serial gradient descent and dual averaging can be implemented in a distributed manner using consensus iterations over a network [32, 58, 69]. Similar to the serial case, if we define

$$\Delta_n(T) = \frac{1}{nT} R_n(T) \tag{2.20}$$

then the best consensus-based algorithms scale as $\Delta_n(T) = O\left(\frac{\log(\sqrt{n}T)}{\sqrt{T}}\right)$ when nodes coordinate over a constant-degree expander graph[5] if the cost functions are convex with bounded subgradients. Note that this rate is not optimal with T or

[5] Expanders graphs are graphs with relatively low degree per node but nevertheless remain well connected and allow for fast information diffusion. There is

n due to the logarithmic factor in the numerator. For strongly convex functions Chapter 8 describes an algorithm that achieves a rate $\Delta_n(T) = O\left(\frac{\log(\sqrt{n}T)}{T}\right)$ which is time optimal for online optimization.

As an alternative to consensus-based algorithms, there also exist hierarchical schemes where a system consists of n slave processors and one master node [2, 50]. In such an architecture the master maintains and updates the estimate $\boldsymbol{w}(t)$. The slaves are responsible for computing gradients which they communicate back to the master. Typically, only one node can communicate with the master at a time and consequently the rest of the slaves need to wait or perform gradient computations with stale values of the estimate $\boldsymbol{w}(t)$. It turns out that such a scheme can achieve an optimal rate $\mathbb{E}\left[\Delta_n(T)\right] = O\left(\frac{1}{\sqrt{nT}}\right)$ for convex objectives that are also smooth, even in the presence of bounded maximum delay [2]. A drawback of implementing this approach is that the master node is a bottleneck of the system. At the expense of building a tree hierarchy among the nodes, the problem can be alleviated, but bottlenecks and vulnerability to single points of failure are still present.

Finally, Dekel et al. [28] also describe a distributed algorithm that asymptotically achieves the optimal distributed rate of $\mathbb{E}\left[\Delta_n(T)\right] = O\left(\frac{1}{\sqrt{nT}}\right)$ for smooth convex functions and $\mathbb{E}\left[\Delta_n(T)\right] = O\left(\frac{1}{nT}\right)$ for smooth strongly convex functions. Their algorithm is synchronous, processes the data in mini-batches and relies on global communication for perfect coordination. Specifically, each node performs a local gradient step in the direction of the average gradient of a mini-batch of incoming data. After each gradient update all nodes participate in a global communication step which is synchronous and computes the true average of the estimates.

2.5 Distributed Optimization

Over the last few years a plethora of parallel and distributed algorithms have been developed to use multiple processors for solving difficult optimization and

---

great interest on both existence and construction methods of expander graphs with bounded or even fixed degree per node [70].

machine learning problems . Reference [22] gives a taxonomy of different classes of distributed optimization algorithms and problems. The more recent reference [6, 79] gives a fairly extensive collection of applications of distributed optimization for machine learning. These applications cover whole range of different platforms from clusters to shared memory machines and GPUs following recent hardware trends. This thesis focuses on large scale computations that take place either in clusters or in general networks of computers. We will not be concerned with the growing literature in multi-threaded programming for shared memory machines or the emerging trend of parallelization using GPUs. Section 1.2 talks about the difference between parallel and distributed algorithms which is mostly conceptual and based on how many synchronization points there are and how much communication is needed. In this section we review a few different algorithms for computations over clusters. We review some representative consensus-based optimization algorithms in more detail to equip the reader with the necessary background for what follows.

We have already discussed some distributed algorithms that are also suitable for stochastic optimization. These include hierarchical algorithms with a master-slave architecture [2, 50] where a team of processors is responsible for gradient computations while a master node collects all partial gradients to update the state, and a mini-batch algorithm which relies on global blocking communication to ensure that all processing nodes have access to the same state $\boldsymbol{w}(t)$ at all times. A different strategy to guarantee the existence of only one estimate shared among the nodes is that of incremental methods such as [9, 47]. For example in [47] there is only one estimate $\boldsymbol{w}(t)$ which is known to only one node at any given time. The node that holds the estimate performs a gradient update using its local data and then passes on the estimate to another node. As the estimate goes through the nodes, either in a cyclic or a random order, the global objective is minimized. One could argue that such a scheme is not a distributed algorithm since only one node is performing computations at any given time. However, in Chapter 4 we show that these methods can be superior when we need to solve multiple optimization tasks on the same cluster in parallel.

Over the last few years there has been great interest in an algorithm called the Alternating Direction Method of Multipliers (ADMM) [16, 41, 44, 96]. This method relies on a dual decomposition [57, 83] of the objective to bring the problem in a separable form that can be solved in a distributed manner. The method is iterative and involves local computation and communication. A drawback is that each iteration requires global coordination and a barrier mechanism to synchronize the nodes and update shared global variables. However, the clean separation between a local computation and a synchronized global communication step renders ADMM compatible with a very popular distributed programming model called MapReduce (see [27] and Section 2.6), where a distributed task is mapped onto n workers that perform computation in isolation and then a reduce step coordinates the nodes and updates the state. For similar examples see also [51] which intentionally creates an overlap of the data subsets of different nodes as an implicit form of communication, or Chapter 1 in [6] which report that such an approach can scale well in practice. An extreme example at the moment of this writing is the algorithm in [1] which uses MapReduce to apply a simple gradient descent algorithm and train an SVM classifier with terabytes of data.

A simple but powerful approach is proposed in [104]. This scheme suggests to first optimize the components of the objective locally at each node without communication and then average the estimates only once at the end. This algorithm conforms more to the parallel paradigm as described in Section 1.2. A similar approach, but closer to this thesis, is described in [52] for training neural networks. There the idea of solving the local problems and then fusing the estimates is not just done once, but rather it is repeated for many training epochs.

All the algorithms mentioned so far rely on either maintaining a unique estimate of the solution at all times, or obtaining such an estimate exactly at some point in the computation. This thesis follows a line of work that started with Tsitsiklis and Bertsekas [10, 94] who showed that it is possible to solve separable optimization problems over a network without structuring the computation hierarchically. In that sense, all nodes have the same role throughout the process and the algorithms

become much simpler to implement and analyze while remaining robust to individual node failures. However, this comes at the price of settling for an iterative algorithm where the nodes arrive all at the same solution only asymptotically.

The consensus-based optimization literature has grown significantly in the last few years with several papers exploring the many different questions associated with these algorithms [25, 32, 58, 68]. Besides the practical advantages of scalability, robustness, and asynchronism, it is possible that these algorithms attract attention because they lend themselves to elegant theoretical analysis while maintaining the potential to be useful in real life applications; two aspects that sometimes tend to contradict each other. Maybe the simplest algorithm in this family is Distributed Sub-gradient Descent (DSD) [55] which extends the basic sub-gradient descent algorithm (2.6) to the case of a network. Specifically, after choosing a doubly stochastic matrix P for a connected network G of n nodes, DSD repeats the update

$$\boldsymbol{w_i}(t+1) = \Pi_{\mathcal{W}} \left[\sum_{j=1}^{n} P_{ij} \boldsymbol{w_j}(t) - a_i(t) \boldsymbol{g_i}(\boldsymbol{w_i}(t)) \right] \tag{2.21}$$

at each node where $a_i(t)$ is the step size used by node i at time t and $\boldsymbol{g_i}$ is a sub-gradient of $f_i(\boldsymbol{w})$ that node i computes at point $\boldsymbol{w_i}(t)$. Note that since the estimates $\boldsymbol{w_i}(t)$ are in general vectors, the subscript here is bold to indicate that $\boldsymbol{w_i}(t)$ is node i's estimate. This is not to be confused with the i-th element of a vector that is denoted by w_i and is not bold. Observe that this iteration reduces to the serial algorithm with $n = 1$ and $P = 1$. Otherwise, each node forms a convex combination of its neighbours estimates. The role of consensus is to bring the estimates to an agreement on the optimum. Using a diminishing step size $a(t) = O\left(\frac{1}{\sqrt{t}}\right)$ at all nodes DSG converges to the optimal solution for L-Lipshitz convex functions at the time optimal rate of $O\left(\frac{1}{\sqrt{T}}\right)$. A number of papers explore extensions of DSG such different constraint set per node, noisy communication links and noisy gradients [58, 69, 80].

An interesting direction is to improve the speed of convergence by employing more tools from the serial optimization literature. For example, [45] achieves a

convergence rate of $O\left(\frac{\log(T)}{T}\right)$ for smooth unconstrained problems using Nesterov's accelerated method [85] while [24] builds upon recent work on inexact proximal methods [72] and acceleration techniques to describe a distributed algorithm that converges at a rate $O\left(\frac{1}{T}\right)$ for smooth convex functions. A drawback of the latter is that it requires using a increasing amount of communication after every gradient iteration which could render the algorithm not practical. In the following subsection we describe in more detail the distributed version of dual averaging. The reason is that this algorithm is representative of the consensus-based literature in terms of its assumptions and performance but has a much cleaner mathematical analysis. It will thus be used as a prototype for investigating the role of the network. We conclude this section by noting that there is also an increasing interest on the less well understood topic of non-convex distributed optimization [102].

2.5.1 Distributed Dual Averaging (DDA)

A distributed version of Nesterov's dual averaging algorithm (2.10) (from now on DDA) is developed and analyzed in [32]. Again, a network G of n nodes is given together with a doubly stochastic consensus protocol P. Each node maintains a primal variable $\boldsymbol{w_i}(t)$ and a dual variable $\boldsymbol{z_i}(t)$. At iteration t, node i uses a step size $a(t)$ and performs three steps:

1. *Communicate:* Send $\boldsymbol{z_i}(t)$ and receive $\boldsymbol{z_j}(t)$ from neighbors according to G
2. *Compute:* Update the primal and dual variables by setting

$$\boldsymbol{z_i}(t+1) = \sum_{j=1}^{n} P_{ij}\boldsymbol{z_j}(t) + \boldsymbol{g_i}(t) \tag{2.22}$$

$$\boldsymbol{w_i}(t+1) = \Pi_{\mathcal{W}}^{\psi}\left[\boldsymbol{z_i}(t+1), a(t)\right] \tag{2.23}$$

3. *Update:* The final estimate of the optimum is the local running average at each node:

$$\overline{\boldsymbol{w}}_{\boldsymbol{i}}(t+1) = \frac{1}{t+1}\sum_{s=1}^{t+1}\boldsymbol{w_i}(s). \tag{2.24}$$

In (2.22), $\boldsymbol{g_i}(t) \in \partial f_i(\boldsymbol{w_i}(t))$ is the latest local subgradient and the projection operator $\Pi^{\psi}_{\mathcal{W}}[\cdot,\cdot]$ is defined as

$$\Pi^{\psi}_{\mathcal{W}}[\boldsymbol{z}, a] = \underset{\boldsymbol{w}\in\mathcal{W}}{\operatorname{argmin}} \left[\langle \boldsymbol{z}, \boldsymbol{w} \rangle + \frac{1}{2a}\psi(\boldsymbol{w}) \right]. \tag{2.25}$$

The projection makes use of a 1-strongly convex function $\psi(\boldsymbol{w})$ for which it is assumed that $\psi(0) = 0$. In [48] it is explained that the form of this projection stems from Fenchel duality theory. For this work, it suffices to observe that if we chose $\psi(\boldsymbol{w}) = \frac{1}{2}\|\boldsymbol{w}\|^2$ we retrieve the standard Euclidean projection of the vector $-a\boldsymbol{z}$.

Observe that DDA uses consensus to bring the dual variables $\boldsymbol{z_i}(t)$ variables to an agreement as $t \to \infty$ rather than the estimates themselves. The intuition is that each dual variable $\boldsymbol{z_i}(t)$ represents the direction towards the global optimum according to node i and DDA is trying to bring the directions to an agreement.

Since we will be using and modifying this algorithm in the sequel, it is useful to make some comments about the analysis of the algorithm and collect some intermediate lemmas and theorems with respect to the convergence rate of DDA. These results can be found in [32] but are included here to keep the manuscript self-contained. First of all, as is common in the analysis of consensus-based optimization algorithms, it is convenient to break the overall error into two terms, an optimization error and a network error. This is done by defining an auxiliary centralized sequence that would only be available if the algorithm had the ability to do perfect averaging at every step. In general perfect averaging is not possible so rather than tracking the error at each node directly, we track how far the average sequence is from the solution and how far each node is from the average. The first corresponds to an optimization error very similar to serial algorithms while the second depends on the performance of distributed consensus over this particular network. For DDA, the average sequences of interest are

$$\overline{\boldsymbol{z}}(t) = \frac{1}{n}\sum_{i=1}^{n} \boldsymbol{z_i}(t) \quad \text{and} \quad \boldsymbol{y}(t) = \Pi^{\psi}_{\mathcal{W}}[\overline{\boldsymbol{z}}(t), a(t-1)] \tag{2.26}$$

i.e., the true average direction to the optimum and an estimate sequence $\boldsymbol{y}(t)$ that evolves accordingly.

We are now ready to state the results. Recall that the objective is assumed to be L-Lipschitz continuous and convex as stated in Section 1.4.

Lemma 2.1. *Consider the sequences* $\{\boldsymbol{w_i}(t)\}_{t=1}^{\infty}$, $\{\boldsymbol{z_i}(t)\}_{t=1}^{\infty}$, *defined in* (2.23), (2.22) *together with* (2.26). *For each* $i = 1, \ldots, n$ *and any* $\boldsymbol{w}^* \in \mathcal{W}$ *we have*

$$\sum_{t=1}^{T} F(\boldsymbol{w_i}(t)) - F(\boldsymbol{w}^*) \leq \sum_{t=1}^{T} F(\boldsymbol{y}(t)) - F(\boldsymbol{w}^*) + L \sum_{t=1}^{T} a(t) \left\| \overline{\boldsymbol{z}}(t) - \boldsymbol{z_i}(t) \right\|, \quad (2.27)$$

and with $\overline{\boldsymbol{y}}(T) = \frac{1}{T} \sum_{t=1}^{T} \boldsymbol{y}(t)$,

$$F(\overline{\boldsymbol{w}}_{\boldsymbol{i}}(T)) - F(\boldsymbol{w}^*) \leq F(\overline{\boldsymbol{y}}(T)) - F(\boldsymbol{w}^*) + \frac{L}{T} \sum_{t=1}^{T} a(t) \left\| \overline{\boldsymbol{z}}(t) - \boldsymbol{z_i}(t) \right\|. \quad (2.28)$$

This lemma helps to break down the local error at each node into an optimization error of the centralized true average sequence, and a network error showing the discrepancy between the local estimates and their true average.

Lemma 2.2. *Let* $\{\boldsymbol{g}(t)\}_{t=1}^{\infty} \subset \mathbb{R}^d$ *be an arbitrary sequence of vectors, let* $\{a(t)\}_{t=1}^{\infty}$ *be a non-increasing sequence, and consider the sequence*

$$\boldsymbol{w}(t+1) = \Pi_{\mathcal{W}}^{\psi} \left[\sum_{s=1}^{t} \boldsymbol{g}(s), a(t) \right]. \quad (2.29)$$

For any $\boldsymbol{w}^* \in \mathcal{W}$ *we have*

$$\sum_{t=1}^{T} \langle \boldsymbol{g}(t), \boldsymbol{w}(t) - \boldsymbol{w}^* \rangle \leq \frac{1}{2} \sum_{t=1}^{T} a(t-1) \left\| \boldsymbol{g}(t) \right\|^2 + \frac{1}{a(T)} \psi(\boldsymbol{w}^*). \quad (2.30)$$

This lemma is standard in proofs of serial gradient-based optimization algorithms. It will help bound the optimization error of the centralized true average sequence which behaves like a serial algorithm.

Lemma 2.3. *For an arbitrary pair* $\boldsymbol{u}, \boldsymbol{v} \in \mathbb{R}^d$, *we have*

$$\left\| \Pi_{\mathcal{W}}^{\psi} [\boldsymbol{u}, a] - \Pi_{\mathcal{W}}^{\psi} [\boldsymbol{v}, a] \right\| \leq \left\| \boldsymbol{u} - \boldsymbol{v} \right\|. \quad (2.31)$$

Based on these Lemmas, [32] proves also a theorem that we will use in the sequel.

Theorem 2.1. *(Theorem 1, [32]) Consider the DDA algorithm* (2.22)-(2.24) *solving problem* (1.8)*. Assume without loss of generality that for a minimizer* $\boldsymbol{w}^*$ *we have* $\psi(\boldsymbol{w}^*) \leq R^2$ *for some real constant* $R > 0$*. The error at any node* i *after* T *iterations is bounded by*

$$Err_i(T) = F(\overline{\boldsymbol{w}}_{\boldsymbol{i}}(T)) - F(\boldsymbol{w}^*) \leq \frac{R^2}{Ta(T)} + \frac{L^2}{2T}\sum_{t=1}^{T} a(t-1) + \frac{L}{T}\sum_{t=1}^{T} a(t)\left(\frac{2}{n}\sum_{j=1}^{n} \|\overline{\boldsymbol{z}}(t) - \boldsymbol{z}_{\boldsymbol{j}}(t)\| + \|\overline{\boldsymbol{z}}(t) - \boldsymbol{z}_{\boldsymbol{i}}(t)\|\right). \tag{2.32}$$

Theorem 2.1 illustrates exactly the separation between an optimization error associated with the true average sequence (bounded by the first two terms in the right hand side of (2.32)), and a network error captured by the terms involving $\|\overline{\boldsymbol{z}}(t) - \boldsymbol{z}_{\boldsymbol{j}}(t)\|$. From this expression, we can bound the network error term to obtain concrete expressions for the convergence rate. Specifically, following the derivation of Theorem 2 in [32], if we select a step size sequence $a(t) = \frac{A}{\sqrt{t}}$ and find the constant A that minimizes the bound, we arrive at[6]

$$\mathrm{Err}_i(T) \leq C_1 \frac{\log(T\sqrt{n})}{\sqrt{T}}, \qquad C_1 = 2LR\sqrt{19 + \frac{12}{1-\sqrt{\lambda_2}}}, \tag{2.33}$$

where λ_2 is the second largest eigenvalue of the consensus matrix P. As we see, due to the logarithmic factor, DDA is neither time optimal nor scalable with the network size n because as was discussed, an ideal distributed algorithm would achieve a convergence rate of $O\left(\frac{1}{\sqrt{nT}}\right)$ after T iterations.

2.6 Distributed Programming Models

A significant part of this work is concerned with the practical issues in the implementation of distributed algorithms. In this section we briefly discuss the main programming tools that are available. We start by referring the reader to

[6] A generalized version of this derivation is given in Section 3.6.

the introduction of [36] for a classic reference that describes the semantics of different communication schemes. This is important to understand what is possible in practice. See for example Section 5.2.2 which explains why a very popular and elegant consensus algorithm cannot be implemented out-of-the-box in practice. In short the reason is that this particular consensus algorithm relies on bi-directional communication where two nodes need to both send and receive information and block until the exchange is complete. Without extra assumptions, this blocking communication causes deadlocks.

To have maximum control one of course has to program at a low level and use sockets [81]. Socket programming is supported in all major programming languages and provides functionality for point-to-point communication between processors. Conveniently, asynchronous consensus-based algorithms rely only on pairwise interactions and can be programmed using sockets. However at that low level it is hard to impose global synchronization e.g., using barriers. Another difficulty arises when the message sizes are much larger than typical TCP packet sizes. Even though in a truly asynchronous algorithm nodes needs to decide completely independently when to transmit and receive, with messages of several MB in size (as is common in machine learning problems) it is a non-trivial programming exercise to ensure that messages are received and the sockets do not choke. Fortunately, most programming languages also have an API for the message passing interface (MPI) [37] which is a very versatile framework for implementing parallel and distributed algorithms. MPI offers functionality for both blocking and non-blocking point-to-point communication so it is suitable for implementing asynchronous algorithms. However the issues of delivering large messages are easier to handle through the MPI interface. Furthermore, MPI provides functionality such as barriers and efficient all-to-all communication so it is convenient to code synchronous distributed algorithms that interleave communication with computation.

Finally, we need to mention a programming framework that is at an even higher level than MPI and has received a significant amount of attention in the last few years. As it turns out, many of the distributed optimization algorithms of interest

operate in stages interleaving communication with computation. The computation can take place in parallel independently on all processors while communication is necessary to update some common variables. This structure fits nicely in a framework called MapReduce [27]. MapReduce is specifically designed for large-scale data processing in clusters. It abstracts a lot of the low-level functionality behind two main operations: *map* and *reduce*. Mapping refers to distributing a load of work into different processors, while the reduce step uses communication to collect and fuse the results. By repeated application of map and reduce steps it is possible to implement synchronous versions of most of the distributed algorithms we have described so far. For an extensive list of recent success stories see also [6]. However, MapReduce does not come without shortcomings. In particular, implementations of iterative algorithms like the gradient descent schemes we have discussed earlier, required sequential calls to MapReduce which imposes synchronization of the nodes and makes the system vulnerable to real cluster delays. In particular, Chapter 5 illustrates the slow node problem which is common in large clusters. In short, a synchronous algorithm is bound to run at the speed of the slowest node. In a large cluster where nodes share their computation cycles between many users, it is very common to have a very slow processor that holds back the whole system. This is one of the main arguments in favour of asynchronism that is not natural to implement within the MapReduce framework.

CHAPTER 3
Communication/Computation Tradeoff and Scalability

3.1 Introduction

Communication over the network is an expensive operation that introduces a discrepancy between the performance of a network of n processors as compared to an idealized single processor that is n times faster than any of them. By increasing the number of processors for a given problem, the total computational capacity of the system increases but so does the overall communication cost. This naturally leads to a tradeoff between communication and computation and it is not the case that adding more computers is always better. To understand this tradeoff, we need to refine the existing analysis of consensus-based distributed optimization algorithms. As will be discussed, an analysis based on number of iterations to convergence is not enough and can even be misleading since it does not account for the cost of communication. The simple intuition is that not all iterations cost the same in actual user time, and thus many cheap iterations might be faster than few expensive ones. For a specific problem, the performance will depend on the relative speed of communication and computation on the available hardware. Given that this speed can be estimated relatively easily, we can calculate what kind of performance to anticipate from our algorithm.

We study the importance of the communication/computation tradeoff in two contexts. This chapter studies the tradeoff focusing on scalability with the network size[1] . To make the discussion concrete we present our analysis within the context of DDA [32] which was summarized in Section 2.5.1. However it should be evident that our findings can be generalized to other consensus-based algorithms. In the

[1] The work presented in this chapter is based on [86].

chapter that follows (Chapter 4) we use the idea of quantifying the communication/computation tradeoff to study parallelization at the task level by looking at scenarios where we need to solve many optimization problems on the same cluster. It is shown that based on the characteristics of the hardware and problems to be solved, it might be possible to solve many tasks in parallel instead of preferring to solve them one after the other.

3.2 Scalability in Consensus-based Distributed Optimization

This section focuses on understanding the limitations and potential for scalability of consensus-based optimization. Furthermore our results are of practical interest. There is a growing trend to perform more and more computation "in the cloud" where a user is charged based on the amount of computational resources used. Having a reasonable estimate of what is the optimal number of processors for a given problem could prevent paying for resources that will not be helpful or will not get used.

The driving questions for this section are: *How many processors should we use and how often should they communicate to solve a distributed optimization problem?*

The recent distributed optimization literature contains multiple consensus-based algorithms with similar rates of convergence for solving problem (1.8). We adopt the *distributed dual averaging* (DDA) framework [32] because its analysis admits a clear separation between the standard (centralized) optimization error and the error due to distributing computation over a network. This facilitates our investigation of the communication/computation tradeoff. The algorithm is described in Section 2.5.1. We first repeat the update equations of DDA and then proceed to quantifying the communication/computation tradeoff and show how it can become part of the analysis. The experimental evaluation that follows, validates the theoretical findings.

In DDA, nodes iteratively communicate and update optimization variables to solve

$$\underset{\boldsymbol{w}\in\mathcal{W}}{\text{minimize}}\ F(\boldsymbol{w}) = \frac{1}{n}\sum_{i=1}^{n} f_i(\boldsymbol{w}). \tag{3.1}$$

Nodes only communicate if they are neighbors in a communication graph $G = (V, E)$, with the $|V| = n$ vertices being the number of processors. The communication graph G is user-defined (application layer) and does not necessarily correspond to the physical interconnections between processors. The DDA update equations are

$$\boldsymbol{z_i}(t) = \sum_{j=1}^{n} P_{ij}\boldsymbol{z_j}(t-1) + \boldsymbol{g_i}(t-1) \tag{3.2}$$

$$\boldsymbol{w_i}(t) = \underset{\boldsymbol{w}\in\mathcal{W}}{\text{argmin}} \left[\langle \boldsymbol{z_i}(t), \boldsymbol{w}\rangle + \frac{1}{2a(t)}\psi(\boldsymbol{w})\right] \tag{3.3}$$

$$\overline{\boldsymbol{w}}_{\boldsymbol{i}}(t) = \frac{1}{t}\big((t-1)\cdot\overline{\boldsymbol{w}}_{\boldsymbol{i}}(t-1) + \boldsymbol{w_i}(t)\big) \tag{3.4}$$

To update $\boldsymbol{z_i}(t)$ in (3.2), each node must communicate to exchange the variables $\boldsymbol{z_j}(t)$ with its neighbors in G. Recall from (2.33) that the convergence rate of DDA is

$$\text{Err}_i(T) \le C_1\frac{\log(T\sqrt{n})}{\sqrt{T}}, \qquad C_1 = 2LR\sqrt{19 + \frac{12}{1-\sqrt{\lambda_2}}}, \tag{3.5}$$

The dependence on the communication topology is reflected through λ_2 the second largest eigenvalue of the consensus matrix P which is conformant to the communication network G. According to (3.5), increasing n slows down the rate of convergence even if λ_2 does not depend on n. At first sight, this sounds disappointing since one would hope that employing more processors should yield some computational speedup when solving the same problem of a given size.

3.3 Quantifying the Tradeoff

In consensus-based algorithms such as DDA, the communication graph G and the cost of transmitting a message have an important influence on convergence speed, especially when communicating one message requires a non-trivial amount of time (e.g., if the dimension of the problem is very high and the messages are large in

size). Intuitively, nodes should engage in expensive communication operations only when the value of the exchanged information is worth it. There exists indeed some work that quantify how much information is needed to learn classifiers from data within a specified accuracy [42] (and references therein). Here we start from the iteration-based convergence bound which describes how much the error can drop in the worst case at each iteration.

To evaluate performance, we are interested in the shortest time to obtain an ϵ-accurate solution where $\mathrm{Err}_i(T) \leq \epsilon$. From (3.5), convergence is faster for topologies with good expansion properties, i.e., when the spectral gap $1 - \sqrt{\lambda_2}$ does not shrink too quickly as n grows. In addition, it is preferable to have a balanced network, where each node has the same number of neighbors so that all nodes spend roughly the same amount of time communicating per iteration. Below we focus on two particular cases and take G to be either a complete graph (i.e., all pairs of nodes communicate) or a k-regular expander graph [70]. This assumption simplifies the analysis and allows us to gain a better intuition of the tradeoff. Furthermore, if we have control over designing the communication topology and we do not want to impose a hierarchy, treating some nodes differently, then there is no reason to deviate from a regular graph. Furthermore, a graph with heavily unbalanced node degrees could be bad simply because some nodes would have to communicate significantly more than others.

By using more processors, the total amount of communication inevitably increases but also more data can be processed in parallel in the same amount of time. We focus on the scenario where the size m of the dataset is fixed but possibly very large. To understand whether there is room for speedup, we move away from measuring iterations and employ a time model that explicitly accounts for communication cost. This will allow us to study the communication/computation tradeoff and draw conclusions based on the total amount of time to reach an ϵ-accurate solution.

3.4 Time model

At each iteration, in step (3.2), processor i computes a local sub-gradient on its subset of the data:

$$\boldsymbol{g_i} \in \frac{\partial f_i(\boldsymbol{w})}{\partial \boldsymbol{w}} = \frac{n}{m} \sum_{j=1}^{\frac{m}{n}} \frac{\partial l_{j|i}(\boldsymbol{w})}{\partial \boldsymbol{w}}. \tag{3.6}$$

The cost of this computation increases linearly with the subset size. Let us normalize time so that one processor computes a sub-gradient on the full dataset of size m in 1 time unit. Then, using n cpus, each local gradient will take $\frac{1}{n}$ time units to compute since each node's data subset is $\frac{1}{n}$-th of the full dataset size. We ignore the time required to compute the projection in step (3.3); often this can be done very efficiently and requires negligible time when m is large compared to n and d.

We account for the cost of communication as follows. In the consensus update (3.2), each pair of neighbors in G transmits and receives one variable $\boldsymbol{z_j}(t-1)$. Since the message size depends only on the problem dimension d' and does not change with m or n, we denote by r the time required to transmit and receive one message, relative to the 1 time unit required to compute the full gradient on all the data. If every node has k neighbors, the cost of one iteration in a network of n nodes is

$$\frac{1}{n} + kr \quad \text{time units / iteration.} \tag{3.7}$$

Using this time model, we study the convergence rate bound (3.5) after attaching an appropriate time unit cost per iteration. To obtain a speedup by increasing the number of processors n for a given problem, we must ensure that ϵ-accuracy is achieved in fewer time units. We perform an analysis of DDA for two scenarios. First, when each node transmits and computes a new gradient at every iteration. This serves as a base case as it conforms to the original description of the algorithm. Then, we analyze scenarios where nodes transmit less frequently and focus more on local gradient steps. As we will see, the results are sometimes surprising but in the end confirm our intuition and are verified in practice.

3.5 Simple Case Analysis: Communicate at Every Iteration

In the original DDA description (3.2)-(3.4), nodes communicate at every iteration. According to our time model, T iterations will cost $T(\frac{1}{n} + kr)$ time units. From (3.5), the time $\tau(\epsilon)$ to reach error ϵ is found by substituting for T and solving for $\tau(\epsilon)$. In the derivation we ignore the logarithmic factor in (3.5) because in light of the analysis in Chapter 7, the logarithm can be saved. Our experimental results also agree with the theory developed here and there does not seem to be any prominent effect that we smooth out. We have:

$$C_1 \frac{1}{\sqrt{\frac{\tau(\epsilon)}{\frac{1}{n}+kr}}} = \epsilon \quad \Longrightarrow \quad \tau(\epsilon) = \frac{C_1^2}{\epsilon^2}\Big(\frac{1}{n} + kr\Big) \text{ time units.} \tag{3.8}$$

This simple manipulation reveals some important facts. If communication is free, then $r = 0$. If in addition the network G is a k-regular expander, then λ_2 is fixed [26], C_1 is independent of n and $\tau(\epsilon) = C_1^2/(\epsilon^2 n)$. Thus, in the ideal situation, we obtain a linear speedup by increasing the number of processors, as one would expect. In reality, of course, communication is not free.

Complete graph: Suppose that G is the complete graph, where $k = n-1$ and $\lambda_2 = 0$. In this scenario we cannot keep increasing the network size without eventually harming performance due to the excessive communication cost. For a problem with a communication/computation tradeoff r, the optimal number of processors is calculated by minimizing $\tau(\epsilon)$ for n:

$$\frac{\partial \tau(\epsilon)}{\partial n} = 0 \quad \Longrightarrow \quad n_{opt} = \frac{1}{\sqrt{r}}. \tag{3.9}$$

Again, in accordance with intuition, if the communication cost is too high (i.e., $r \geq 1$) and it takes more time to transmit and receive a gradient than it takes to compute it, using a complete graph cannot speedup the optimization. We reiterate that r is a quantity that can be easily measured for a given hardware and a given optimization problem. As we report in Section 5.6, the optimal value predicted by our theory agrees very well with experimental performance on a real cluster.

Expander: For the case where G is a k-regular expander, the communication cost per node remains constant as n increases. From (3.8) and the expression for C_1 in (3.5), we see that n can be increased without losing performance, although the benefit diminishes (relative to kr) as n grows.

Other graphs: For the rest of the chapter we focus on complete and bounded-degree expanders which are the most favourable cases. Before proceeding, we comment here on the communication/computation trade-off for other types of graphs such as k-regular graphs and grids. Using the convergence rates proven in Corollary 1 in [32], we have

1. *k-regular:* If a graph is k-regular but not an expander, then the DDA error decreases at a rate

$$\mathrm{Err}(T) = \frac{C}{\sqrt{T}} \frac{n \log Tn}{k}. \tag{3.10}$$

   The same manipulation as before yields a time to ϵ-accuracy of

$$\tau(\epsilon) = \frac{C^2}{\epsilon^2} \left(\frac{n}{k^2} + \frac{n^2 r}{k} \right). \tag{3.11}$$

   In this case, unless $k > \sqrt{n}$, the time $\tau(\epsilon)$ increases with n and there is no trade-off. We can only lose by increasing n.

2. *k-connected $\sqrt{n} \times \sqrt{n}$ grid:* If a graph is an $\sqrt{n} \times \sqrt{n}$ grid with every node connected to its k nearest horizontal and vertical neighbours, we have $k \leq n^{\frac{1}{4}}$ and a node has in general $4k$ neighbours[2] . The DDA error decreases at a rate

$$\mathrm{Err}(T) = \frac{C}{\sqrt{T}} \frac{\sqrt{n} \log Tn}{k} \tag{3.12}$$

[2] Ignoring the few nodes on the boundaries who have $2k$ neighbours at the corners, and $3k$ neighbours on the edges.

which is faster than k-regular graphs by a factor of $\sqrt{n}$. When taking the communication cost into account, the time to ϵ-accuracy is

$$\tau(\epsilon) = \frac{C^2}{\epsilon^2}\left(\frac{1}{n} + 4rk\right)\frac{n}{k^2} = \frac{C^2}{\epsilon^2}\left(\frac{1}{k^2} + \frac{4rn}{k}\right). \tag{3.13}$$

In this case, if $k = \Theta(1)$ again there is no trade-off and performance is worse as n increases. If we let k grow with n but without making the graph too well-connected, a trade-off can be retrieved. For example, by taking $k = n^{\frac{1}{4}}$, we have

$$\tau(\epsilon) = \frac{C^2}{\epsilon^2}\left(\frac{1}{n^{\frac{1}{2}}} + 4rn^{\frac{3}{4}}\right), \tag{3.14}$$

and the optimal number of processors is

$$n_{opt} = \frac{1}{(6r)^{\frac{4}{5}}} \tag{3.15}$$

which is smaller than in the case of the complete graph.

3.6 General Case Analysis: Sparse Communication

Next we investigate the more general situation where we adjust the frequency of communication. A natural choice is to fix an intercommunication interval h and only transmit every h gradient steps. A more adaptive approach is to increase h with time. The motivation for this is that in the beginning the nodes could start from completely different estimates and they need to communicate frequently to coordinate their search directions for the optimum. However as the estimates approach the solution the nodes may not need that much communication to stay aligned, simply because proximity to the solution implies exploring the most flat region of the convex objective where individual gradient steps have small impact.

3.6.1 Bounded Intercommunication Intervals

Suppose that a consensus step takes place once every $h + 1$ iterations. That is, the algorithm repeats $h \geq 1$ cheap iterations (no communication) of cost $\frac{1}{n}$ time units followed by an expensive iteration (with communication) which takes $\frac{1}{n} + kr$ time units. This strategy clearly reduces the overall average cost per iteration. The

caveat is that the network error $\|\overline{\boldsymbol{z}}(t) - \boldsymbol{z_i}(t)\|$ is higher because of having executed fewer consensus steps.

In a cheap iteration, we replace the update (3.2) by $\boldsymbol{z_i}(t) = \boldsymbol{z_i}(t-1) + \boldsymbol{g_i}(t-1)$. After some straight-forward algebra we can show that (for proofs of (3.16), (3.19) please consult Appendix A and B):

$$\boldsymbol{z_i}(t) = \sum_{w=0}^{H_t-1} \sum_{k=0}^{h-1} \sum_{j=1}^{n} \left[P^{H_t-w}\right]_{ij} \boldsymbol{g_j}(wh+k) + \sum_{k=0}^{Q_t-1} g_i(t-Q_t+k), \tag{3.16}$$

where $H_t = \lfloor \frac{t-1}{h} \rfloor$ counts the number of communication steps in t iterations, and $Q_t = \mathrm{mod}(t,h)$ if $\mathrm{mod}(t,h) > 0$ and $Q_t = h$ otherwise. Using the fact that $P\mathbf{1} = \mathbf{1}$, we obtain

$$\begin{aligned}
\overline{\boldsymbol{z}}(t) - \boldsymbol{z_i}(t) =& \frac{1}{n}\sum_{s=1}^{n} \boldsymbol{z_s}(t) - \boldsymbol{z_i}(t) && (3.17)\\
=& \sum_{w=0}^{H_t-1} \sum_{j=1}^{n} \Big(\frac{1}{n} - \left[P^{H_t-w}\right]_{ij}\Big) \sum_{k=0}^{h-1} \boldsymbol{g_j}(wh+k) \\
&+ \frac{1}{n}\sum_{s=1}^{n} \sum_{k=0}^{Q_t-1} \big(\boldsymbol{g_s}(t-Q_t+k) - \boldsymbol{g_i}(t-Q_t+k)\big). && (3.18)
\end{aligned}$$

Taking norms, recalling that the f_i are convex and Lipschitz, and since $Q_t \leq h$, we arrive at

$$\|\overline{\boldsymbol{z}}(t) - \boldsymbol{z_i}(t)\| \leq \sum_{w=0}^{H_t-1} \left\| \frac{1}{n}\mathbf{1}^T - \left[P^{H_t-w}\right]_{i,:} \right\|_1 hL + 2hL \tag{3.19}$$

Finally, by bounding the ℓ_1 distance of row i of P^{H_t-w} to its stationary distribution as t grows (see Appendix C), we can show that

$$\|\overline{\boldsymbol{z}}(t) - \boldsymbol{z_i}(t)\| \leq 2hL\frac{\log(T\sqrt{n})}{1-\sqrt{\lambda_2}} + 3hL. \tag{3.20}$$

for all $t \leq T$. Comparing (3.20) to equation (29) in [32], the network error after t iterations is no more than h times larger when a consensus step is only performed once every $h+1$ iterations. Finally, we substitute the network error in (2.32). For

$a(t) = \frac{A}{\sqrt{t}}$, we have $\sum_{t=1}^{T} a(t) \leq 2A\sqrt{T}$, and

$$\mathrm{Err}_i(T) \leq \left(\frac{R^2}{A} + AL^2\left(1 + \frac{12h}{1-\sqrt{\lambda_2}} + 18h\right)\right)\frac{\log\left(T\sqrt{n}\right)}{\sqrt{T}} = C_h\frac{\log\left(T\sqrt{n}\right)}{\sqrt{T}}. \tag{3.21}$$

We minimize the leading term C_h over A to obtain

$$A = \frac{R}{L}\left(\sqrt{1 + 18h + \frac{12h}{1-\sqrt{\lambda_2}}}\right)^{-1} \quad \text{and} \quad C_h = 2RL\sqrt{1 + 18h + \frac{12h}{1-\sqrt{\lambda_2}}}. \tag{3.22}$$

Of the T iterations, only $H_T = \lfloor \frac{T-1}{h} \rfloor$ involve communication. So, T iterations will take

$$\tau = (T - H_T)\frac{1}{n} + H_T\left(\frac{1}{n} + kr\right) = \frac{T}{n} + H_T kr \ \text{ time units.} \tag{3.23}$$

To achieve ϵ-accuracy, ignoring again the logarithmic factor, we need $T = \frac{C_h^2}{\epsilon^2}$ iterations, or

$$\tau(\epsilon) = \left(\frac{T}{n} + \left\lfloor\frac{T-1}{h}\right\rfloor kr\right) \leq \frac{C_h^2}{\epsilon^2}\left(\frac{1}{n} + \frac{kr}{h}\right) \text{ time units.} \tag{3.24}$$

From the last expression, for a fixed number of processors n, there exists an optimal value for h that depends on the network size and communication graph G:

$$h_{opt} = \sqrt{\frac{nkr}{18 + \frac{12}{1-\sqrt{\lambda_2}}}}. \tag{3.25}$$

If the network is a complete graph, using h_{opt} yields $\tau(\epsilon) = O(n)$; i.e., using more processors hurts performance when not communicating every iteration. On the other hand, if the network is a k-regular expander then $\tau(\epsilon) = \frac{c_1}{\sqrt{n}} + c_2$ for constants c_1, c_2, and we obtain a diminishing speedup.

3.6.2 Increasingly Infrequent Communication

Next, we consider progressively increasing the intercommunication intervals. This captures the intuition that as the optimization moves closer to the solution, progress slows down and a processor should have "something significantly new to say" before it communicates. Let $h_j - 1$ denote the number of computation-only iterations performed between the $(j-1)$st and jth expensive iteration,i.e., the first

communication is at iteration h_1, the second at iteration $h_1 + h_2$, and so on. We consider schemes where $h_j = j^p$ for $p \geq 0$. The number of iterations that nodes communicate out of the first T total iterations is given by $H_T = \max\{H\colon \sum_{j=1}^{H} h_j \leq T\}$. We have

$$\int_{y=1}^{H_T} y^p dy \leq \sum_{j=1}^{H_T} j^p \leq 1 + \int_{y=1}^{H_T} y^p dy \quad \Longrightarrow \quad \frac{H_T^{p+1} - 1}{p+1} \leq T \leq \frac{H_T^{p+1} + p}{p+1}, \tag{3.26}$$

which means that $H_T = \Theta(T^{\frac{1}{p+1}})$ as $T \to \infty$. Similar to (3.19), the network error is bounded as

$$\|\overline{\boldsymbol{z}}(t) - \boldsymbol{z_i}(t)\| \leq \sum_{w=0}^{H_t-1} \left\| \frac{1}{n}\mathbf{1}^T - \left[P^{H_t-w}\right]_{i,:} \right\|_1 \sum_{k=0}^{h_w-1} L + 2h_t L \tag{3.27}$$

$$= L \sum_{w=0}^{H_t-1} \left\| \frac{1}{n}\mathbf{1}^T - \left[P^{H_t-w}\right]_{i,:} \right\|_1 h_w + 2h_t L. \tag{3.28}$$

We split the sum into two terms based on whether or not the powers of P have converged. Using the split point $\hat{t} = \frac{\log(T\sqrt{n})}{1-\sqrt{\lambda_2}}$, the ℓ_1 term is bounded by 2 when w is large and by $\frac{1}{T}$ when w is small:

$$\|\overline{\boldsymbol{z}}(t) - \boldsymbol{z_i}(t)\| \leq L \sum_{w=0}^{H_t-1-\hat{t}} \left\| \frac{1}{n}\mathbf{1}^T - \left[P^{H_t-w}\right]_{i,:} \right\|_1 h_w \tag{3.29}$$

$$+ L \sum_{w=H_t-\hat{t}}^{H_t-1} \left\| \frac{1}{n}\mathbf{1}^T - \left[P^{H_t-w}\right]_{i,:} \right\|_1 h_w + 2h_t L \tag{3.30}$$

$$\leq \frac{L}{T} \sum_{w=0}^{H_t-1-\hat{t}} w^p + 2L \sum_{w=H_t-\hat{t}}^{H_t-1} w^p + 2t^p L \tag{3.31}$$

$$\leq \frac{L}{T} \frac{(H_t - \hat{t} - 1)^{\frac{1}{p+1}} + p}{p+1} + 2L\hat{t}(H_t - 1)^p + 2t^p L \tag{3.32}$$

$$\leq \frac{L}{p+1} + \frac{Lp}{T(p+1)} + 2L\hat{t}H_t^p + 2t^p L \tag{3.33}$$

since $T > H_t - \hat{t} - 1$. Substituting this bound into (2.32) and taking the step size sequence to be $a(t) = \frac{A}{t^q}$ with A and q to be determined, we get

$$\mathrm{Err}_i(T) \leq \frac{R^2}{AT^{1-q}} + \frac{L^2 A}{2(1-q)T^q} + \frac{3L^2 A}{(p+1)(1-q)T^q} + \frac{3L^2 pA}{(p+1)(1-q)T^{1+q}}$$
$$+ \frac{6L^2\hat{t}A}{T} \sum_{t=1}^{T} \frac{H_t^p}{t^q} + \frac{6L^2 A}{T} \sum_{t=1}^{T} t^{p-q}. \tag{3.34}$$

The first four summands converge to zero when $0 < q < 1$ as $t \to \infty$. Since $H_t = \Theta(t^{\frac{1}{p+1}})$,

$$\frac{1}{T}\sum_{t=1}^{T}\frac{H_t^p}{t^q} \leq \frac{1}{T}\sum_{t=1}^{T}\frac{O(t^{\frac{1}{p+1}})^p}{t^q} \leq O\left(\frac{T^{\frac{p}{p+1}-q+1}}{T}\right) = O\left(T^{\frac{p}{p+1}-q}\right) \tag{3.35}$$

which converges to zero if $\frac{p}{p+1} < q$. To bound the last term, note that $\frac{1}{T}\sum_{t=1}^{T} t^{p-q} \leq \frac{T^{p-q}}{p-q+1}$, so the term goes to zero as $T \to \infty$ if $p < q$. In conclusion, $\mathrm{Err}_i(T)$ converges no slower than $O(\frac{\log(T\sqrt{n})}{T^{q-p}})$ since $\frac{1}{T^{q-\frac{p}{p+1}}} < \frac{1}{T^{q-p}}$. If we choose $q = \frac{1}{2}$ to balance the first three summands, for small $p > 0$, the rate of convergence is arbitrarily close to $O(\frac{\log(T\sqrt{n})}{\sqrt{T}})$, while nodes communicate increasingly infrequently as $T \to \infty$.

Out of T total iterations, DDA executes $H_T = \Theta(T^{\frac{p}{p+1}})$ iterations involving communication and $T - H_T$ iterations without communication, so

$$\tau(\epsilon) = O\left(\frac{T}{n} + T^{\frac{p}{p+1}}kr\right) = O\left(T\left(\frac{1}{n} + \frac{kr}{T^{\frac{1}{p+1}}}\right)\right). \tag{3.36}$$

In this case, the communication cost kr becomes a less and less significant proportion of $\tau(\epsilon)$ as T increases. So for any $0 < p < \frac{1}{2}$, if k is fixed, we approach a linear speedup behaviour $\tau(\epsilon) = \Theta(\frac{T}{n})$. To get $\mathrm{Err}_i(T) \leq \epsilon$, ignoring the logarithmic factor, we need

$$T = \left(\frac{C_p}{\epsilon}\right)^{\frac{2}{1-2p}} \text{ iterations, with } C_p = 2LR\sqrt{7 + \frac{12p+12}{(3p+1)(1-\sqrt{\lambda_2})} + \frac{12}{2p+1}}. \tag{3.37}$$

From this last equation we see that for $0 < p < \frac{1}{2}$ we have $C_p < C_1$, so using increasingly sparse communication can, in fact, be faster than communicating at every iteration in terms of time units .

3.7 Experimental Evaluation

To verify our theoretical findings, we implement DDA on a cluster of 14 nodes with 3.2 GHz Pentium 4HT processors and 1 GB of memory each, connected via Ethernet that allows for roughly 11 MB/sec throughput per node. Our implementation is in C++ using the send and receive functions of OpenMPI v1.4.4 for

communication. The Armadillo v2.3.91 library, linked to LAPACK and BLAS, is used for efficient numerical computations.

3.7.1 Application to Metric Learning

Metric learning [97, 98, 101] is a computationally intensive problem where the goal is to find a distance metric $D(u, v)$ such that points that are related have a very small distance under D while for unrelated points D is large. Following the formulation in [78], we have a data set $\{\boldsymbol{u_j}, \boldsymbol{v_j}, s_j\}_{j=1}^m$ with $\boldsymbol{u_j}, \boldsymbol{v_j} \in \mathbb{R}^d$ and $s_j = \{-1, 1\}$ signifying whether or not $\boldsymbol{u_j}$ is similar to $\boldsymbol{v_j}$ (e.g., similar if they are from the same class). Our goal is to find a symmetric positive semi-definite matrix $A \succeq 0$ to define a pseudo-metric of the form $D_A(\boldsymbol{u}, \boldsymbol{v}) = \sqrt{(\boldsymbol{u} - \boldsymbol{v})^T A (\boldsymbol{u} - \boldsymbol{v})}$. To that end, we use a hinge-type loss function $l_j(A, b) = \max\{0, s_j \left(D_A(\boldsymbol{u_j}, \boldsymbol{v_j})^2 - b\right) + 1\}$ where $b \geq 1$ is a threshold that determines whether two points are dissimilar according to $D_A(\cdot, \cdot)$. In the batch setting, we formulate the convex optimization problem

$$\underset{A,b}{\text{minimize}} \quad F(A, b) = \sum_{j=1}^{m} l_j(A, b) \quad \text{subject to} \quad A \succeq 0, b \geq 1. \tag{3.38}$$

The subgradient of l_j at (A, b) is zero if $s_j(D_A(\boldsymbol{u_j}, \boldsymbol{v_j})^2 - b) \leq -1$. Otherwise

$$\frac{\partial l_j(A, b)}{\partial A} = s_j(\boldsymbol{u_j} - \boldsymbol{v_j})^T(\boldsymbol{u_j} - \boldsymbol{v_j}), \quad \text{and} \quad \frac{\partial l_j(A, b)}{\partial b} = -s_j. \tag{3.39}$$

Since DDA uses vectors $\boldsymbol{w_i}(t)$ and $\boldsymbol{z_i}(t)$, we represent each pair $(A_i(t), b_i(t))$ as a $d^2 + 1$ dimensional vector. The communication cost is thus quadratic in the dimension. In step (3.2) of DDA, we use the proximal function $\psi(\boldsymbol{w}) = \frac{1}{2}\boldsymbol{w}^T\boldsymbol{w}$, in which case (3.3) simplifies to taking $\boldsymbol{w_i}(t) = -a(t-1)\boldsymbol{z_i}(t)$, followed by projecting $\boldsymbol{w_i}(t)$ to the constraint set by setting $b_i(t) \leftarrow \max\{1, b_i(t)\}$ and projecting $A_i(t)$ to the set of positive semi-definite matrices by first taking its eigenvalue decomposition and reconstructing $A_i(t)$ after forcing any negative eigenvalues to zero.

We use the MNIST digits dataset which consists of 28×28 pixel images of handwritten digits 0 through 9. Representing images as vectors, we have $d = 28^2 = 784$ and a problem with $d^2 + 1 = 614657$ dimensions. The goal is to learn a 784×784

matrix A. With double precision arithmetic, each DDA message has a size approximately 4.7 MB. We construct a dataset by randomly selecting 5000 pairs from the full MNIST data. One node needs 29 seconds to compute a gradient on this dataset, and sending and receiving 4.7 MB takes 0.85 seconds. The communication/computation tradeoff value is estimated as $r = \frac{0.85}{29} \approx 0.0293$. According to (3.9), when G is a complete graph, we expect to have optimal performance when using $n_{opt} = \frac{1}{\sqrt{r}} = 5.8$ nodes. Figure 3–1 shows the evolution of the average function value

$$\overline{F}(t) = \frac{1}{n} \sum_i F(\overline{\boldsymbol{w}}_{\boldsymbol{i}}(t)) \tag{3.40}$$

for 1 to 14 processors connected as a complete graph, where $\overline{\boldsymbol{w}}_{\boldsymbol{i}}(t)$ is as defined in (3.4). There is a very good match between theory and practice since the fastest convergence is achieved with $n = 6$ nodes.

In the second experiment, to make r closer to 0, we apply PCA to the original data and keep the top 87 principal components, containing 90% of the energy. The dimension of the problem is reduced dramatically to $87 \cdot 87 + 1 = 7570$ and the message size to 59 KB. Using 60000 random pairs of MNIST data, the time to compute one gradient on the entire dataset with one node is 2.1 seconds, while the time to transmit and receive 59 KB is only 0.0104 seconds. In this case $r = \frac{0.0104}{2.1} = 0.005$ and $n_{opt} = 14.15$. Again, for a complete graph, Figure 3–2 illustrates the evolution of $\bar{F}(t)$ for 1 to 14 nodes. As we see, increasing n speeds up the computation. The speedup we get is close to linear at first, but diminishes since communication is not entirely free.

3.7.2 Nonsmooth Convex Minimization

Next we create an synthetic problem where the minima of the components $f_i(\boldsymbol{w})$ at each node are very different, so that communication is essential in order to obtain an accurate optimizer of $F(\boldsymbol{w})$. We define $f_i(\boldsymbol{w})$ as a sum of a max of high

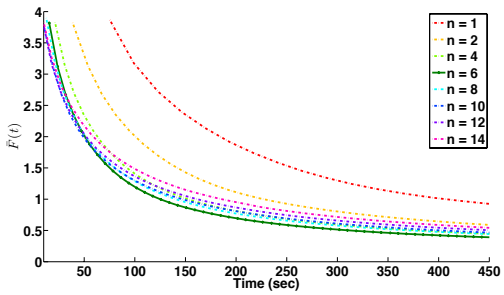

Figure 3–1: In a subset of the Full MNIST data for our specific hardware, $n_{opt} = \frac{1}{\sqrt{r}} = 5.8$. The fastest convergence is achieved on a complete graph of 6 nodes.

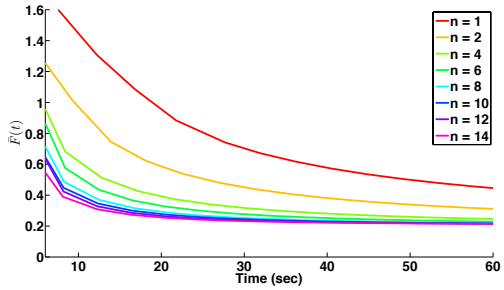

Figure 3–2: In the reduced MNIST data using PCA, the communication cost drops and a speedup is achieved by scaling up to 14 processors.

dimensional quadratics,

$$f_i(\boldsymbol{w}) = \sum_{j=1}^{M} \max\left(l(\boldsymbol{w}, \boldsymbol{x}_{j|i}^1), l(\boldsymbol{w}, \boldsymbol{x}_{j|i}^2)\right), \tag{3.41}$$

$$l(\boldsymbol{w}, \boldsymbol{x}_{j|i}^{\xi}) = (\boldsymbol{w} - \boldsymbol{x}_{j|i}^{\xi})^T(\boldsymbol{w} - \boldsymbol{x}_{j|i}^{\xi}), \quad \xi \in \{1, 2\}, \tag{3.42}$$

where $\boldsymbol{w} \in \mathbb{R}^{10,000}$, $M = 15,000$ and $\boldsymbol{x}_{j|i}^1, \boldsymbol{x}_{j|i}^2 \in \mathbb{R}^{10,000}$ are the data, and in this case, the centers of the quadratics. Figure 3–3 illustrates again the average function value $\bar{F}(t)$ for 10 nodes in a complete graph topology. As a baseline performance, we compare to the performance when nodes communicate at every iteration ($h = 1$). For this problem $r \approx 0.00089$ and, from (3.25), $h_{opt} = 1$. Naturally, communicating every 2 iterations (i.e., $h = 2$) slows down convergence. Over the duration of the experiment, with $h = 2$, each node communicates with its peers 55 times. We select $p = 0.3$ for increasingly sparse communication, and get $H_T = 53$ communications per node. As we see, even though nodes communicate as often as in the $h = 2$ case, convergence is even faster than when communicating at every iteration. This verifies

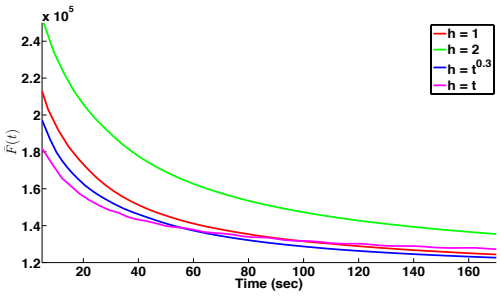

Figure 3–3: Sparsifying communication to minimize (3.41) with 10 nodes in a complete graph topology. When waiting $t^{0.3}$ iterations between consensus steps, convergence is faster than communicating at every iteration ($h = 1$), even though the total number of consensus steps performed over the duration of the experiment is equal to communicating every 2 iterations ($h = 2$). When waiting a linear number of iterations between consensus steps ($h = t$) DDA does not converge to the right solution. Note: all methods are initialized from the same value; the x-axis starts at 5 sec.

our intuition that communication is more important in the beginning. Finally, the case where $p = 1$ is shown. This value is out of the permissible range, and as expected, DDA does not converge to the correct solution. This can be seen on the graph where the line for $p = 1$ does not continue to decrease the objective value and flattens out instead.

3.8 Conclusions and Future Work

This chapter focused on understanding the tradeoff between communication and computation in consensus-based distributed optimization. The main message is that a convergence rate describing the number of iterations required to obtain an ϵ-accurate solution may not be representative of an algorithm's performance as experienced by the end user. Whenever communication is non-negligible it must be taken into account as not all iterations cost the same amount of time. In particular, we saw that for complete graphs there is an optimal number of processors that balances the two costs while for expander graphs we have a diminishing reward in terms of speedup. Furthermore, we showed that communicating less frequently as the algorithm approaches the solution can yield significant time savings. We were able to reason about these findings theoretically and also validate them in experiments on a

real cluster. It would be interesting to have an even more detailed analysis that covers graphs with uneven degree distributions or where the cost of transmitting to k neighbours is something faster than just worst case $O(k)$. Finally, it would be important to extend this analysis to stochastic optimization scenarios where rather than reducing the gradient computation cost when we add more nodes, we are instead able to process more data in the same amount of time. Finally, we claim that our tradeoff analysis extends beyond DDA to other algorithms in the consensus-based optimization family such as DSG as well as various distributed averaging-type algorithms (e.g., [51, 52, 104]). The reason is that this chapter proposes a meta analysis that accounts for communication computation costs. The analysis does not rely on any particular properties of DDA but rather exploits DDA's clean mathematical convergence proofs to draw important conclusions. The meta analysis should be possible for other algorithms as well, even though the mathematical details would depend on the specific convergence proof of each algorithm. Verifying this claim as well as extending the analysis to the case of stochastic optimization, where $h_t = t^p$ could correspond to using increasingly larger mini-batches would be important directions for future work.

CHAPTER 4
Parallelization at the Task Level

4.1 Introduction

There are many machine learning scenarios where the same data needs to be used to solve multiple related optimization problems. Fitting models with free parameters typically requires sweeping through a range of values for parameter tuning. Cross-validation involves performing the same computation on different overlapping subsets of the data. One approach to multiclass classification is to train several "one-versus-all" classifiers, which could be learned in parallel. In exploratory data analysis, one may not know a priori which loss function or kernel function is best suited to the data, and running several learning algorithms with different characteristics can help determine an appropriate model.

To solve S optimization problems using a consensus approach, we must run S instances of the algorithm one after the other. Is there a way to work simultaneously on the S instances to speed up the computation? In other words, is it possible to benefit from parallelizing at the task level? As it turns out, the answer is not straightforward, and as one might guess, it again depends on the relative communication to computation cost.

The main drawback of consensus-based optimization algorithms comes from the potentially high communication cost associated with distributed consensus. The distributed optimization literature includes also incremental algorithms [9,47,54] which can significantly reduce the communication cost by visiting the function components f_i one at a time; communication involves passing a single copy of the optimization variable from one node to the next, as opposed to each node maintaining a local copy that needs to be communicated to all neighbours. References [54] and [9] analyze the situation where every component $f_i(\boldsymbol{w})$ has to be visited once in every cycle, essentially assuming that the nodes are connected as a complete graph. Markov

Incremental Gradient Descent (MIGD) [47] generalizes Nedic and Bertsekas' algorithm for any connected graph assuming that information can only be transmitted between neighboring nodes. The side effect of this type of scheduling is that incremental methods converge slower in practice and have a worse scaling with the size of the network n even though asymptotically the convergence rate, as a function of the number of iterations T, is $O(\frac{1}{\sqrt{T}})$ for both consensus-based and incremental methods when the components f_i are convex and Lipschitz continuous (see, e.g., the comparison in [32]).

A key observation is that, with incremental algorithms, it is possible to run S instances in parallel. For example in MIGD, many instances can move through the network according to independent random walks so in principle MIGD could solve S problems in the same time it would take to solve one. Now the question becomes: *Is it better to run serial consensus-based procedures or parallel incremental procedures?*

To make the comparison concrete, we focus on two specific algorithms, Distributed Dual Averaging (DDA) [32] and Markov Incremental Gradient Descent (MIGD) [47]. These algorithms are chosen as representatives of the consensus and incremental approaches to optimization. The convergence rate of both DDA and MIGD depends on the structure of the communication graph over which they are executed. In [32] it is argued that for well-connected graphs (e.g., expanders), DDA achieves a target error n times faster than MIGD on a network of n nodes. This analysis suggests that parallel copies of MIGD should be much slower than DDA. In addition, since for MIGD the instances are behaving as independent random walks, nothing prevents the instances from colliding, causing additional delays.

4.2 Algorithms

DDA has already been discussed in Section 2.5.1 and earlier in this chapter. The theoretical analysis [32] shows that all nodes achieve accuracy ϵ after $T = O(\frac{1}{\epsilon^2})$ iterations if the nodes communicate over a well-connected graph (complete or expander). The description of DDA in [32] assumes synchronous iterations, requiring nodes to use blocking communication. This is convenient for analysis, but as explained in [87], in practice synchronous DDA is impossible to implement without

using a barrier mechanism to synchronize the nodes. Furthermore, when barriers are used, synchronization forces the algorithm to run at the speed of the slowest node. For this reason, in the experiments reported below, we use an asynchronous version called Push-Sum DDA (PS-DDA) [88], for which the error converges at the same rate but with different constants. PS-DDA has a lot of other advantages and it is the described in detail in Chapter 5.

For MIGD see Algorithm 1. The setup in [47] considers minimizing the function

$$F(\boldsymbol{w}) = \sum_{i=1}^{n} f_i(\boldsymbol{w}), \quad \boldsymbol{w} \in \mathcal{W}. \tag{4.1}$$

To match this setting with our standard objective (1.8), each term $f_i(\boldsymbol{w})$ for MIGD is scaled by $\frac{1}{n}$. Consider the case where we only have one optimization problem. One node in the network has a token, meaning that this node holds the most recent estimate and will perform a projected gradient descent update based on its local objective component f_i. Once the node has updated the estimate, it chooses a neighbour at random according to the corresponding column of a doubly stochastic matrix P, and passes on the token and the estimate to this neighbour. As explained in [32], to reach accuracy ϵ, MIGD requires $T = O(\frac{n}{\epsilon^2})$ iterations on a well-connected graph.

MIGD operates with a fixed step size a which is set to

$$a \leq \frac{2\epsilon n^2}{L^2 K}, \tag{4.2}$$

for ϵ accuracy, and K is a constant that depends on the network connectivity implicitly through P as follows:

$$K = \max_i \left\{ 2n^2 \left[\left(I - P + \frac{\mathbf{1}\mathbf{1}^T}{n} \right)^{-1} \right]_{ii} - n \right\}. \tag{4.3}$$

4.3 Communication/computation Tradeoff for Multiple Tasks

Consider solving $S \leq n$ optimization problems of the form (1.8) using DDA or MIGD on a network of n nodes. To remove the effect of different graph topologies, from now on we assume that the nodes communicate over a well connected graph;

Algorithm 1 MIGD

Initialize: $t = 1, \boldsymbol{w(1)} = \mathbf{0}_d, tok_1 = 1, tok_i = 0, i > 1, \epsilon \in \mathbb{R}, a = \frac{2\epsilon n^2}{L^2 K}$
while $t \leq T$ **do**
  for $i \in [n]$ **do**
    if $tok_i = 1$ **then**
      Compute subgradient $\boldsymbol{g_i}(t) \in \partial_{\boldsymbol{w}} f_i(\boldsymbol{w}(t))$
      $\boldsymbol{w}_{tmp} = \boldsymbol{w}(t) - a\boldsymbol{g_i}(t)$
      $\boldsymbol{w}(t+1) = \Pi_{\mathcal{W}}[\boldsymbol{w}_{tmp}]$
      Sample a neighbour $j \in V$ according to $P_{:,j}$
      Set $t = t + 1$
      Send $\boldsymbol{w}(t)$, tok_i and t to j
      Set $tok_i = 0$
    else
      Poll neighbours for incoming messages
      if There is a message msg_j from j **then**
        $t = msg_j(t)$
        $tok_i = msg_j(tok_j)$
        $\boldsymbol{w}(t) = msg_j(\boldsymbol{w}(t))$
      end if
    end if
  end for
end while

i.e. either a complete graph or a k-regular expander. To solve S problems with DDA, we run the S jobs sequentially. For MIGD we start S random walks at different nodes in the network. We assume that each processor will only work on one problem at a time, so if two random walks arrive at the same node, one is served and the other one is buffered until the node's CPU becomes available to process it.

Let us momentarily assume that all jobs take the same number of iterations to reach the desired accuracy[1] . To solve all problems to ϵ-accuracy, DDA will need $ST = O(\frac{S}{\epsilon^2})$ iterations in total. For MIGD, if the random walks do not collide (so that no job idles in buffer) then the time to solve S problems is identical to the time to solve 1; i.e., the total number of MIGD iterations is $O(\frac{n}{\epsilon^2})$. Immediately we see that, in this idealized scenario, if $S = \Theta(n)$ then MIGD exploits task parallelization to become competitive with DDA. Note, however, that this discussion is in terms

[1] This is the case, e.g., if all jobs are clones of each other. We make this simplifying assumption here for the sake of gaining intuition, and our results do not rely on it.

of the number of *iterations* or gradient steps taken by the nodes. This abstraction ignores the time for communicating the messages between nodes which, for some problems, may be commensurate with the time to perform a single update. Let us thus refine the analysis following the model of [86] presented in Section 3.3.

With $S = 1$, at every iteration, if every node has k neighbours, the cost of one iteration in DDA is,

$$\frac{1}{n} + kr \quad \text{time units / iteration.} \tag{4.4}$$

Observe that the same approach can be used for MIGD for the node that has the token. Since that node will only transmit the estimate to one neighbour, the cost of a MIGD iteration is simply

$$\frac{1}{n} + r \quad \text{time units / iteration.} \tag{4.5}$$

We can use these time models to reason about the *time* to achieve ϵ accuracy rather than the number of iterations. In particular, to reach ϵ accuracy on S problems, DDA will take $S \cdot T$ iterations or

$$\tau_{dda}(\epsilon) = C_{dda} \frac{S}{\epsilon^2} \left(\frac{1}{n} + kr \right) \text{ time units} \tag{4.6}$$

where C_{dda} is a constant that does not depend on n. For MIGD on the other hand, in the ideal case where the random walks do not collide and there are no delays, the total is equivalent to the time it takes to do T iterations; i.e.,

$$\begin{aligned} \tau_{migd}(\epsilon) =& C_{migd} \frac{n}{\epsilon^2} \left(\frac{1}{n} + r \right) \\ =& C_{migd} \frac{1}{\epsilon^2} (1 + nr) \text{ time units.} \end{aligned} \tag{4.7}$$

This simple manipulation reveals that the relative performance of the two algorithms is more delicate than what the iterations bounds suggest. If G is the complete graph, where $k = n - 1$, then asymptotically as n increases, $\tau_{dda}(\epsilon) = \Theta(\tau_{migd}(\epsilon))$; i.e., the algorithms are equivalent. This is not surprising. Even though MIGD needs n times more iterations, each MIGD iteration takes only $\frac{1}{n}$-th of the time since MIGD

requires communication with only one neighbor instead of $n-1$. Of course, if the message size is very small, the network bandwidth is very high, or computation takes significantly longer than a transmission, then $r \to 0$ and in that case MIGD is indeed n times slower.

For a non-vanishing communication cost, if the graph is a k-regular expander, then as n increases we see that the defining factor is the relative size of $S \cdot k$ and n which multiply the communication/computation tradeoff r in (4.6) and (4.7), and it could be the case that either algorithm is faster. Following similar reasoning, these arguments can be generalized to the case where the jobs are not identical. We omit this analysis but refer the reader to the experiments section below.

Finally, before moving on to the empirical study of the communication / computation tradeoffs, we note that, without additional coordination or scheduling, the MIGD random walks are not guaranteed to avoid each other, and so collisions are bound to occur as the number of jobs S increases. Thus, in practice, MIGD suffers from additional delay while jobs are sitting idle, and this additional delay depends on the number of jobs executing relative to the number of nodes.

4.4 Experimental Study

In order to understand the difference between consensus-based and incremental optimization procedures when running many jobs, we implemented DDA and MIGD on a cluster and compared their performance on the metric learning task used in Section 3.7. We emphasize that these experiments are designed to illustrate the different regimes for these two procedures, and that the "best" optimization procedure will depend on the particular task and hardware.

Our experiments involve solving multiple instances of the metric learning problem introduced in Section 3.7. To solve S instances we run DDA S times serially and run S copies of MIGD in parallel. Because the instances in MIGD move according to a random walk, different instances may collide; if an instance i moves to a node which is already busy processing job j, job i waits in a buffer until the node completes processing job j. When there are many instances, collisions can cause a delay in the convergence of MIGD; see Section 4.4.4 for details.

For these experiments we use 32000 randomly selected pairs of MNIST images, partitioned evenly among the processors. To decrease the communication cost and make DDA more competitive, we exploit the symmetry of matrix A to represent each message as $\frac{d(d-1)}{2} + d + 1$ doubles for the upper triangular part of A and the bias b. With each MNIST digit image being a $28^2 = 784$-dimensional vector, each message becomes 307721 doubles or approximately 2.4 MB in size.

4.4.1 Cluster description

The experiments are performed on a cluster with 8 worker nodes running Matlab 2009b. Each node has two 4-core Xeon processors clocked at 2.5GHz with 14GB of RAM, and 300GB of storage. All nodes communicate over 100Mbps Ethernet. Communication among the 64 available CPUs is organized as an expander graph G with average node degree 32 ± 2.6. To create G, we sample graphs from the family of Erdős-Rényi random graphs [33] with $p = 0.5$, which are known to have good expansion properties with high probability, and keep the graph for which P has the smallest second eigenvalue. The implementation of DDA and MIGD is in Matlab using the `labSend` and `labReceive` communication primitives supported by the Parallel Computing Toolbox. No coordination of the nodes is imposed and both algorithms are completely asynchronous and subject to real network conditions and communication delays.

4.4.2 Solving Multiple Identical Jobs

Our first experiment is designed to illustrate the effects of communication cost and delay (due to buffering and network effects beyond our control) on distributed optimization. The theoretical analysis in Section 4.3 made the simplifying assumption that all problems being solved are identical in order to make statements with some level of generality. To keep the conditions of this first experiment somewhat symmetric and stay close to the theoretical models discussed in Section 4.3, we solve a varying number of *identical* instances of the metric learning problem. The more realistic scenario of solving tasks that are different, is studied in Section 4.4.3.

Since all instances are identical, they all take approximately the same time to complete. To limit the overhead of monitoring, we focus on one instance. For DDA

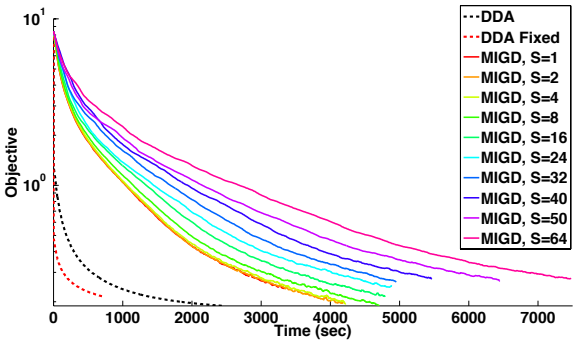

Figure 4–1: Evolution of the objective for DDA for diminishing and fixed step size, and MIGD with a varying number of jobs running in parallel. Experiment run on an expander graph with $n = 64$.

we measure the time it takes to complete one instance of the problem. For MIGD we track one instance of the problem and measure the total time it takes for this job to converge in the presence of other jobs.

Figure 4–1 shows the objective value as a function of the total wall time in the system for the two algorithms with varying numbers of jobs. Since MIGD uses a fixed step size, we also show DDA with a fixed step size even though the default DDA algorithm using a diminishing step size. As we expect, the time for a single DDA instance to complete is very fast. MIGD is slower but not by a factor of n. As the number of parallel instances increases there is an additional slowdown for MIGD per instance as a result of buffering from random walk collisions. Figure 4–2 illustrates this slowdown by graphing how much service (i.e., processor time) task 1 receives in the presences of other tasks. All data points on the plot are taken by running the experiment for the same amount of time. To generate the figure, we plot the fraction of gradient steps that job 1 receives in the presence of other jobs running in parallel, relative to the case of 100% service when job 1 is running alone. In a perfect parallelization scenario, increasing the number of tasks should not affect task 1. However, due to task interference serialization points occur and task 1 receives less and less gradient steps for the duration of the experiment in the presence of other tasks.

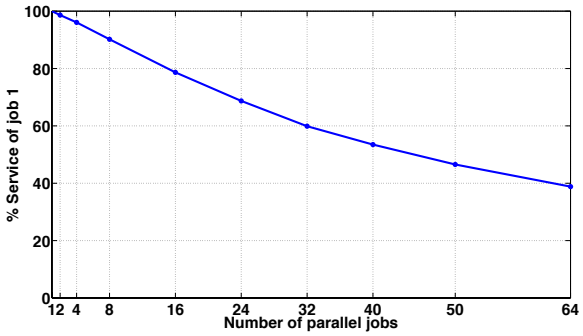

Figure 4–2: Amount of service received by job 1 in the presence of $2-64$ other jobs relative to the service received when the job is running alone. All experiments are run for the same amount of time.

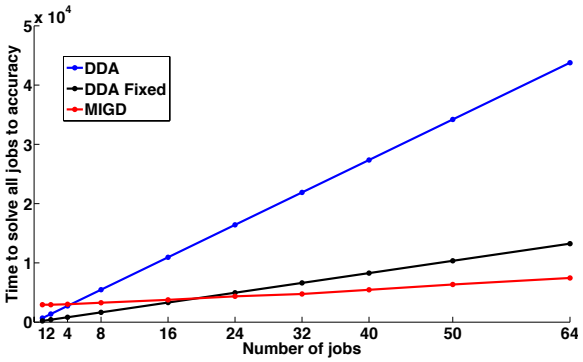

Figure 4–3: Cumulative time to solve $S = 1, \ldots, 64$ jobs with MIGD and DDA with fixed and diminishing step size. In both cases there is a cross over point beyond which MIGD exploits its ability to solve many jobs in parallel to become the faster algorithm.

In Figure 4–3 we plot the cumulative time it takes to solve S tasks with DDA and MIGD up to some desired accuracy ϵ for S between 1 and 64. We can see that there is indeed a tradeoff in wall time between consensus-based and iterative algorithms. For few tasks DDA is indeed fast enough that it has time to solve 4 tasks one after the other before MIGD solves them in parallel. However at that point, MIGD is able to accommodate and solve more and more tasks in parallel and is therefore preferable. Notice also that DDA with fixed step size is faster than DDA with diminishing step size, so the cross over point between DDA and MIGD is moved from 4 to 20 jobs but it still exists.

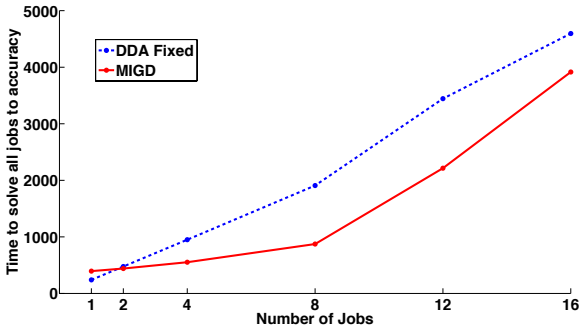

Figure 4–4: Cumulative time to solve $S = 1, \ldots, 16$ jobs to accuracy ϵ using DDA with fixed step size and MIGD.

4.4.3 Sweeping a Problem Parameter

In the previous example we considered running multiple identical instances of the same problem. We now turn to a problem where there is a parameter in the optimization and we wish to run multiple instances for *different* values of the parameter. As an example we chose to do ℓ_2-regularized metric learning as suggested, e.g., in [46]

$$\operatorname*{argmin}_{A,b} \frac{1}{m} \sum_{j=1}^{m} l_j(A, b) + \frac{\lambda}{2} \|A\|_F \qquad s.t.\ b \geq 1, A \succeq 0.$$

In this case, varying the amount of regularization makes the target problem harder (large λ) or easier (small λ). On an expander graph of 16 nodes we solve 1 to 16 different task sweeping values of λ from 1 to 0.0001. Figure 4–4 shows the cumulative amount of time needed to solve all problems with MIGD and DDA with fixed step size. Despite the fact that the jobs are now different from each other, the same behaviour is still evident For very few jobs, DDA is superior, while adding more jobs renders MIGD a better choice. Notice that extrapolating from this figure we see that the benefit of MIGD cannot be sustained. In fact, if we let the number of tasks grow, eventually we expect that due to excessive number of collisions, MIGD would again fall back and become slower than DDA.

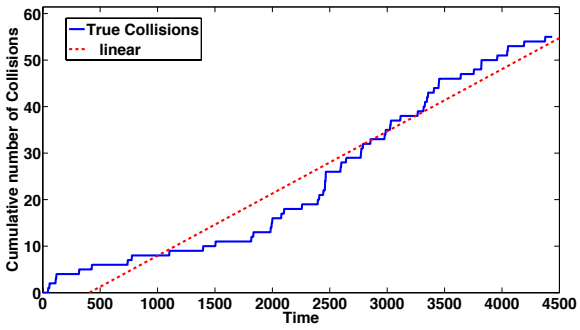

Figure 4–5: Cumulative number of collisions on an expander of size 64 with 8 jobs solved by MIGD simultaneously. A linear fit shows that the collision rate is approximately 0.013 collisions per second.

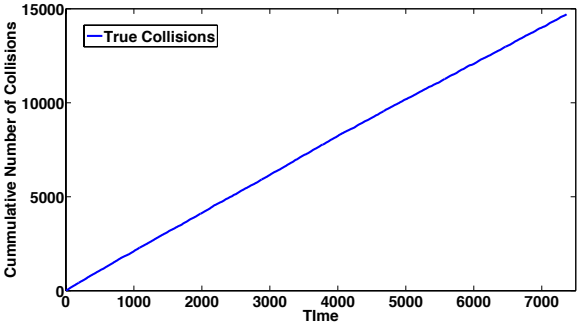

Figure 4–6: Cumulative number of collisions on an expander of size 64 with 64 jobs solved by MIGD simultaneously. The collision rate is 2 collisions per second.

4.4.4 Collisions between MIGD Random Walks

It is clear that collisions between random walks prevent MIGD from achieving perfect parallelization at the task level. Quantifying the expected number of collisions is a challenging combinatorial problem because the jobs move according to the graph structure and we leave the full theoretical investigation for future work. However, we track the frequency of collisions experimentally in order to get a sense of how the collisions affect performance.

To count collisions in our first experiment with a 64-node graph, every time an idle processor receives a new message, it also polls its buffer. If the buffer is not empty, that processor registers a collision since there is at least one other job in its buffer waiting for service. Figures 4–5 and 4–6 show the cumulative number of collisions counted by all 64 nodes when $S = 8$ and 64 respectively.

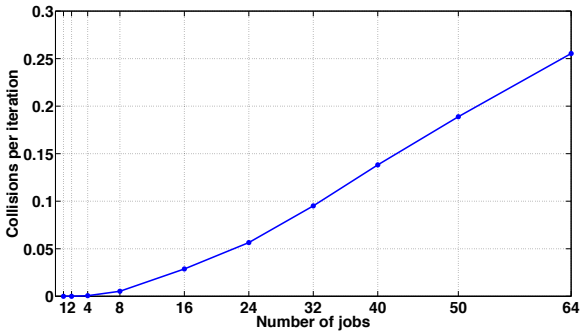

Figure 4–7: Total number of collisions per iteration with $S = 1, \ldots 64$ random walks on the same expander graph.

Figure 4–7 shows the average number of collisions per iteration as the number of jobs increases. With more than one job on the same graph the probability of a collision is non-zero but with few jobs this probability is actually very small. On our graph with 64 nodes, we observe that in practice up to 4 jobs fit without having any interference between them. However increasing the number of jobs further yields a linear increase in the number of collisions per iteration. Figure 4–7 could be used in practice to estimate how long to run MIGD for; if we have S jobs and each job needs T iterations to reach the desired optimization accuracy, without collisions we would run MIGD for T iterations. In practice, from the figures we can estimate a rate q of collisions per iteration, and then run MIGD for $(1 + q)T$ iterations to achieve the desired accuracy.

Finally, for the experiments so far, MIGD was terminated based on a time limit. If we instead terminate MIGD based on an achieved accuracy, it is possible to stop certain walks when the accuracy is reached and reduce the rate of collisions. This would require evaluating the objective regularly and may not be possible if that computation is expensive.

4.5 Discussion

In this chapter we saw another role of the communication/computation trade-off when solving multiple problems on the same cluster. With many optimization tasks to run on distributed data, choosing the right computational algorithm will depend on the cluster node capabilities, network topology, time complexity of local

iterations, communication delays, and the number of tasks. In the case of a single optimization problem we can choose the cluster size to trade off computation and communication time but for a given cluster size, choosing how to optimize for many parallel objectives affects this tradeoff as well. We illustrate that the bounds on the iterations for consensus-based and incremental procedures do not necessarily characterize the overall system time for solving multiple optimizations since there is room for improvement by exploiting task parallelization. In particular, we showed that there are regimes in which running parallel copies of the slower incremental MIGD algorithm is better than running sequential copies of the much faster consensus-based DDA algorithm.

We illustrated this discrepancy via an application in metric learning but we believe that the phenomena we exhibit here are general. We considered two algorithms, DDA and MIGD, that are representative of the consensus-based and incremental approaches. The regimes for different algorithms will vary, but we believe it is relatively clear that the tradeoff may appear in many problems of interest. In particular, there may not be a single best algorithm which works for all optimization problem sizes and clusters. For the future, our study of parallel MIGD has highlighted the importance of collisions and now two related questions arise.

Question 1: What is the probability of a collision occurring and thus what is the expected number extra iterations that we need to account for due to collisions? Our empirical study indicates a regularity in the behaviour of collisions as we increase the number of random walks. We observed similar linear growth in the collision rate for all complete and expander graphs that we tried. Characterizing the behaviour of collisions probabilistically can be a challenging combinatorial problem. For example, consider the simplest case of S uniform random walks running on a complete graph of n nodes. One can view this as a repeated balls-in-bins Markov process. Each random walk is a ball and each node's buffer is a bin. We can start by throwing each ball randomly into a bin. Then, we pick exactly one ball from each non-empty bin, and throw those balls back into the bins randomly. The state of this system is the vector of bin occupancies which evolves

as a non-reversible Markov chain.[2] Characterizing the stationary distribution of this random walk would yield an estimate of how long a job has to wait in a buffer before being serviced.

Question 2: Is it possible to design the random walks so that they remain random and rapidly mixing while they avoid each other and cause as few collisions as possible? Complementary to understanding and anticipating the effects of collisions we might want to be proactive and try to avoid collisions all together. Developing avoiding random walks is another challenging theoretical problem. Recent work [3] has shown that $O(\frac{n}{\log n})$ random walks can be coupled so that they co-exist on a complete graph of n nodes without ever colliding with each other. Although similar results are currently absent for other graph topologies, it would be interesting to use such coupled random walks to schedule the execution of parallel MIGD jobs.

More broadly, developing an analysis of the running time for optimization algorithms which incorporates the effect of running parallel copies would help in head-to-head comparisons. Another challenging problem is in scheduling very heterogeneous jobs which operate on the same distributed data. If some jobs have very short local iterations and others have very long local iterations, the queueing delay from parallel MIGD may become very burdensome. Another interesting constraint to consider is the energy consumption of computation and communication, which may alter the tradeoffs for parallelization. Finally, developing a framework for analyzing the performance of distributed optimization algorithms in a multi-user system with jobs contending for computation time may lead to new ways of approaching distributed optimization.

[2] For example, if each bin has one ball, in one move all balls can be collected in the same bin, but in the next move this cannot be reversed if we have more than two balls.

CHAPTER 5
Practical Consensus Algorithms

5.1 Introduction

Consensus-based optimization algorithms have the appealing feature that they can operate in a peer-to-peer fashion, with minimal coordination between nodes. Much of the existing literature has focused on establishing and analyzing convergence properties of these algorithms. However the theory does not always consider issues that arise when implementing and using consensus-based algorithms for distributed optimization on a real scenario. To be of practical interest, a consensus-based optimization algorithm needs to accommodate the constraints imposed by the network. For example, not all (directed) networks admit a doubly stochastic matrix [35] as required by both DDA and DSD algorithms. However, relinquishing double stochasticity can introduce bias in the optimization [69, 90]. Moreover, it is desirable to only rely on one-directional communication between nodes because in the bi-directional case where each node blocks until it receives a response, deadlocks can occur when the network has cycles. Finally, an algorithm should be able to converge in the presence of network induced communication delays and should allow for an asynchronous implementation to avoid delaying the whole computation if a particular node is very slow.

This chapter describes and analyzes PS-DDA, an algorithm that addresses all the concerns mentioned above. In particular, PS-DDA guarantees convergence to the optimum without knowing the stationary distribution of the averaging matrix or the size of the network, and the convergence rate is the same as standard DDA. Furthermore, the communication semantics of the consensus matrices make PS-DDA truly asynchronous and allow for a clean analysis when modelling varying intercommunication intervals and communication delays. The chapter begins by identifying and justifying what features are critical for consensus-based optimization

algorithms to be useful in practice. Then we proceed to describe PS-DDA, an algorithm that has all the desired properties. Our theoretical analysis shows that convergence is still achieved at a rate $O\left(\frac{1}{\sqrt{T}}\right)$ where T is the number of iterations and thus PS-DDA is competitive with the existing algorithms in the literature. The true power of the algorithm is illustrated via experiments on a cluster under real network conditions at the end of the chapter.

5.2 Features of Distributed Consensus Algorithms

Recall equation (2.22) of the DDA algorithm:

$$\boldsymbol{z_i}(t+1) = \sum_{j=1}^{n} P_{ij}\boldsymbol{z_j}(t) + \boldsymbol{g_i}(t). \tag{5.1}$$

If we disregard for the moment the addition of the latest local gradient $\boldsymbol{g_i}(t)$, we see that the equation involves a consensus step which requires communication between nodes. Each node i transmits its latest dual variable $\boldsymbol{z_i}(t)$ to its neighbours, receives the dual variables $\boldsymbol{z_j}(t)$ from its neighbours, and forms a convex combination. For the network to agree on the vector which minimizes $F(\boldsymbol{w})$, the nodes need to agree on the direction to the optimal value which is locally captured by each variable $\boldsymbol{z_i}$. This notion of agreement, or consensus in a network was described in Section 2.1.

If each node i in a network G holds a value z_i, stacking all the values (treated as scalars for simplicity), into a vector $\boldsymbol{z} = (z_1, \ldots, z_n)^T$, a linear iteration scheme of the form

$$\boldsymbol{z}(t) = P(t)\boldsymbol{z}(t-1). \tag{5.2}$$

can bring the node's values to an agreement. Furthermore, from Perron-Frobenius theory [73], with a time-homogeneous row stochastic matrix $P(t) \equiv P \in \mathbb{R}^{n\times n}$ such that $P\mathbf{1} = \mathbf{1}$ and $p_{ij} > 0$ if $(j,i) \in E$, consensus is achieved almost surely on a value c that is a convex combination of the initial values $\boldsymbol{z}(0)$. A very popular special case is the *average consensus* problem where the limit value must be the average of the initial values i.e., $z_i(t) \to \frac{1}{n}\sum_{i=1}^{n} z_i(0)$ as $t \to \infty$.

In the following subsections, we analyze the consensus iteration from a practical standpoint. To design distributed optimization algorithms, we distinguish the following three key properties that a distributed algorithm should possess to be applicable:

Averaging. The consensus matrix needs to be an averaging matrix. As will be explained shortly, for the purposes of optimization, the communication mechanism should assign equal importance to all messages so as not to bias the objective function being optimized. An important complication arises from the fact that the easiest consensus algorithms to analyze theoretically, are based on doubly stochastic matrices. However such algorithms are difficult to implement especially in presence of the two next requirements.

One-directional Communication. In theory the way information exchange is implemented receives little attention. In practice however, an engineer has available only the functionality that can be supported by the given hardware and software. We argue that algorithms relying on one-directional communication where each node only sends out information and does not need to receive a response, are preferable to algorithms that rely on bi-directional communication. The intuition is that the latter might lead to deadlocks when the network has cycles.

Time-Varying. Real networks, especially in clusters, are quite reliable. However there is still a certain degree of volatility and messages are not delivered instantaneously. Furthermore, different nodes may have different workloads and may not be able to process messages at the same speed. To be able to model random variability in the network and the node's performance, we would like to have asynchronous algorithms where each node makes local decisions of when and with which node to communicate. It becomes clear that a communication protocol that remains fixed in time is too restrictive and cannot capture such complicated dynamics.

Next we proceed to explain and justify the above properties in more detail. As one might expect, to accommodate all the requirements we need to restrict the set of consensus matrices P that can be used. Consequently, much of the theory for

analyzing consensus becomes inapplicable if the matrices do not have nice properties such as symmetry and double stochasticity. For this reason, we also include a section explaining the semantics of different consensus matrices.

5.2.1 Averaging

For the purposes of consensus-based distributed optimization, an averaging matrix is necessary in order not to bias the objective function. The importance of averaging has been mentioned in previous work on distributed optimization (e.g., [69, 90]). To gain some intuition why averaging is important, consider equation (2.22). Unwrapping the recursion and assuming $\boldsymbol{z_i}(t) = 0$ for simplicity, we have

$$\boldsymbol{z_i}(t) = \sum_{s=1}^{t-1} \sum_{j=1}^{n} \left[P^{t-s-1}\right]_{ij} \boldsymbol{g_j}(s) + \boldsymbol{g_i}(t). \tag{5.3}$$

As t grows, P^{t-s-1} converges to its limit $\mathbf{1} \cdot \boldsymbol{\pi}^T$ where $\boldsymbol{\pi}$ is the stationary distribution of P. More importantly, the gradients of different nodes are weighted based on the stationary distribution $\boldsymbol{\pi}$, and this weighting is unequal unless $\boldsymbol{\pi}$ is the uniform distribution. The implication for consensus-based optimization is that instead of minimizing the true objective (1.8) a naïve consensus-based approach will minimize the biased objective $\tilde{F}(\boldsymbol{w}) = \sum_{i=1}^{n} \pi_i f_i(\boldsymbol{w})$.

To avoid this problem, most previous work has insisted on using doubly stochastic update matrices P, i.e., matrices P for which $\mathbf{1}^T P = \mathbf{1}^T$ and $P\mathbf{1} = \mathbf{1}$. Such matrices are averaging by definition. As we will explain below however, doubly stochastic matrices are undesirable to use in practice because they require synchronization and coordination. Furthermore, it turns out that averaging can be achieved without them. For example, [90] and Chapter 6 show that a simple reweighing of the objective removes the bias and achieves averaging. Specifically, for any row-stochastic matrix P with stationary distribution $\boldsymbol{\pi}$ (see also the *scaled agreement algorithm* in [65]):

$$F(\boldsymbol{w}) = \frac{1}{n} \sum_{i=1}^{n} f_i(\boldsymbol{w}) = \sum_{i=1}^{n} \pi_i \left[\frac{f_i(\boldsymbol{w})}{\pi_i n}\right] = \sum_{i=1}^{n} \pi_i h_i(\boldsymbol{w}). \tag{5.4}$$

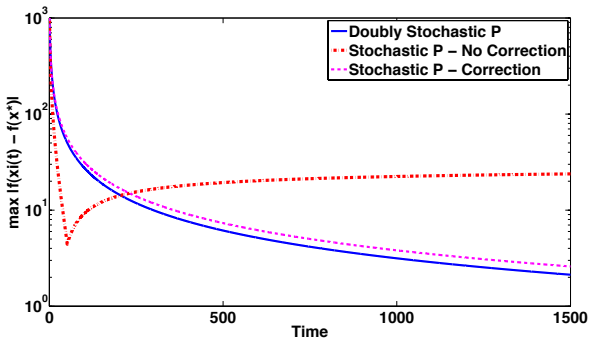

Figure 5–1: Illustration of optimization bias with non-doubly stochastic matrices. The blue curve shows progress of distributed dual averaging with a doubly stochastic consensus matrix P. With a row stochastic P that has a non-uniform stationary distribution we end up solving a biased problem with a different optimum shown by the red curve. By rescaling the objective function components we remove the bias as shown in the purple curve.

In essence, the objective components are reweighed and then the optimization algorithm "uses" the bias to revert the weights back to uniform. To illustrate the effect of bias, consider a simple problem where the objective component at each node i is a simple quadratic: $f_i(\boldsymbol{w}) = (\boldsymbol{w} - i\mathbf{1})^T(\boldsymbol{w} - i\mathbf{1}), \boldsymbol{w} \in \mathbb{R}^5$ and we are interested in minimizing the quadratic sum. For this problem we can easily compute the exact minimizer $\boldsymbol{w}^* = 5.5 \cdot \mathbf{1}$ with $F(\boldsymbol{w}^*) = 412.5$. In Figure 5–1 the blue curve shows the progress of the minimization for DDA which uses a doubly stochastic matrix P. The progress is captured in terms of the maximum error $\max_i |F(\overline{\boldsymbol{w}}_{\boldsymbol{i}}(t)) - F(\boldsymbol{w}^*)|$. Instead of a consensus protocol with a uniform stationary distribution $0.1 \cdot \mathbf{1}$, we generate a stochastic matrix with stationary distribution

$$\boldsymbol{\pi} = (0.06, 0.04, 0.11, 0.10, 0.09, 0.04, 0.10, 0.21, 0.14, 0.11)^T \tag{5.5}$$

giving significant weight to node 8. As a result, the optimal value is biased to be $\boldsymbol{w}^*_{biased} = 6.2847 \cdot \mathbf{1}$ with $F(\boldsymbol{w}^*_{biased}) = 443.286$. This is the value that DDA with a row stochastic matrix converges to as shown by the red curve. Finally, by employing the suggested rescaling of the objective function components, we remove the bias and solve the original problem as the purple curve shows.

Note that an implementation of this approach requires that both $\boldsymbol{\pi}$ and n are known. Moreover, it requires that P be time-homogeneous, which is not reasonable

when the network induces time-varying delays or the algorithms needs to be asynchronous. Later in this chapter we will see a different solution that does not have many of these limitations.

5.2.2 One-Directional Communication

There exist consensus algorithms with bi-directional communication between nodes. For example, a popular algorithm for distributed averaging is Randomized Gossip [15]. This is a very simple asynchronous algorithm where at each iteration a node is activated at random (e.g., according to a global poisson clock), and that node selects a random neighbour. The two nodes, exchange values and set their new values to the pairwise average of their previous values. In the context of distributed optimization, this algorithm is suggested in [32] as a way to reduce the communication overhead when nodes have a large number of neighbours. However, in practice, the bi-directional communication model can be problematic as it creates deadlocks. Consider, for example, three nodes i, j, and k connected in a clique as shown in Figure 5–2. Without coordination, there is no way of enforcing a rule that only two neighbours activate in one time instant and that no other node initiates an averaging update until the one currently in progress is completed. Consequently, suppose i attempts to exchange its value with j. While i blocks, awaiting j's response, node j attempts to exchange values with k. While j is blocked waiting, k, being oblivious to the situation, independently initiates an exchange with i. Now there is a deadlock since i cannot proceed without j's value, j cannot proceed without k's value, and k is waiting for i. In practice, this scenario is highly unlikely if the time it takes to transmit information is much shorter than the frequency of communication. However, in distributed optimization problems, where the amount of data transmitted is non-trivial, this scenario is not so rare and we have certainly encountered it in our experiments. One remedy may be to artificially slow down the rate at which nodes communicate (e.g., by sleeping for a random time before transmitting), but this is undesirable since the ultimate goal is to solve the optimization as quickly as possible. In addition, the added complexity of coordinating two nodes that exchange information via blocking receive operations or idling in non-blocking

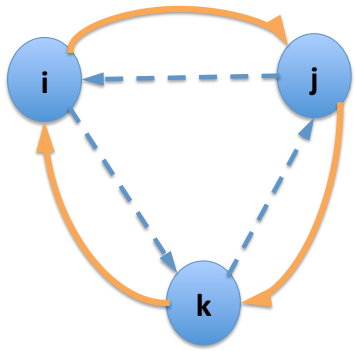

Figure 5–2: Illustration of a deadlock with bi-directional communication. Node i has transmitted to j and is blocked until j responds. In the meantime j has transmitted and is waiting for k's response. Finally, k has transmitted and is waiting for i's response creating a deadlock.

communication is usually undesirable. For these reasons, we argue that a consensus algorithm must rely on one-directional communication where nodes only transmit information and then proceed with their local computations without expecting a response.

5.2.3 Time varying protocols

The main reason to use a time varying consensus matrix $P(t)$ is to add the expressive power needed to model real network conditions. Consider for example the case of network introduced random delays. Figure 5–3 shows a simple scenario where node j does not know in advance how many messages it will receive from node i. Suppose at time t, node i sends message $M1$ to node j and that message is delayed by one time unit. This can be modelled by sending that message to a virtual delay node. Then at time $t+1$ node i transmits a second message $M2$ that is delivered without delay. Node j will receive $M1$ and $M2$ simultaneously. It becomes clear that without knowing in advance how many messages node j will receive, it is impossible to arrange weights and form a convex combination of the incoming information with a fixed matrix P. Of course the justification does not need to come from random delays. We can imagine scenarios where the nodes are not all running at the same speed either due to varying work load or simply because

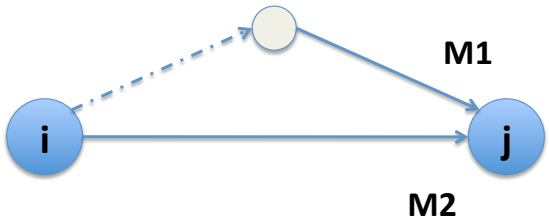

Figure 5–3: Node i transmits two messages $M1$ and $M2$ at times t and $t+1$. If $M1$ is delayed by one time unit, both messages arrive simultaneously at j. Node j has no way of knowing in advance how many messages it will receive.

the processing power is not the same. In that case a node might need to limit its communication overhead by only transmitting to some of its neighbours at time.

5.2.4 Semantics of Different Consensus Matrices

With the above considerations in mind, we turn our attention to consensus matrices $P(t)$ which in general can be time-varying. In discrete time, at each iteration the state vector of the node values evolves as (5.2). Depending on the structure and behaviour of $P(t)$, a matrix can encode the semantics of one- or two-directional communication and they can drive $\boldsymbol{z}(t)$ to average consensus.

An important distinction is made depending on whether or not $P(t)$ varies with time. The case where $P(t) = P$ is time-homogenous implies that the algorithm is *synchronous*; i.e., each node communicates with all of its neighbours exactly once at every iteration. Synchronous protocols require that nodes block until they have received one message from each neighbour, and this is undesirable since then the entire network moves at the pace of the slowest node. Allowing the consensus matrix to be time-dependent provides the freedom to encode *asynchronous* communication where a node may choose whether or not to transmit something to each neighbour at each round. It also becomes possible to encode time-varying communication delays. The topic is studied in detail in Chapter 6.

To achieve average consensus with one-directional communication, previous work has insisted on using doubly stochastic consensus matrices where each row and column of $P(t)$ sums to 1. See for example [63]. However, with asynchronous and time-varying matrices, agreeing on time-varying weights that preserve double

stochasticity requires additional coordination, effectively foregoing asynchronous operations. In addition, in the presence of communication delays, double stochasticity can be lost (see Chapter 6 and [56]). Finally, there might be cases where a directed network does not admit a doubly stochastic consensus matrix [35]. For these reasons we focus on the case where $P(t)$ is stochastic but not doubly stochastic.

With stochastic matrices, the nodes become disentangled and more autonomous. When $P(t)$ is row stochastic, each node controls a row of the consensus matrix (each node applies the weights in its row of $P(t)$ to the messages it receives). At each iteration, the new value at a node is a weighted average of the incoming values. The weights are encoded in the corresponding row of $P(t)$ and need to sum to 1. If on the other hand we use a column stochastic matrix $P(t)$, the semantics are different as each node controls a column of the matrix; each node sends a portion of its current value to each neighbour so that the portion fractions sum to one (as indicated by the stochastic columns).The receiver simply sums up the incoming messages which is convenient when we do not know how many messages will be received at each iteration (e.g., due to random communication delays).

5.3 Push-Sum Consensus

In Section 6.2 it is proven that restricting to doubly stochastic consensus matrices in distributed dual averaging is not necessary and it is still possible to converge to the optimum with a general row stochastic matrix P. However there are multiple reasons why using a row stochastic matrix may not be desirable. The bias correction described in Section 6.2 requires knowledge of the stationary distribution of P in advance which is restrictive. Moreover, with a time-varying consensus matrix $P(t)$, we may not even be able to specify the stationary distribution beyond its expectation and variance [66] or may only be able to achieve average consensus in expectation [4].

We propose the use of a different one directional consensus algorithm called *Push-Sum*. A simple asynchronous version of the algorithm was first analyzed in [49] for complete graphs. In [7] convergence is proven for any graph based on weak ergodicity arguments. Let us focus on the simple synchronous case where

all nodes exchange information with their neighbours simultaneously to gain some understanding of how Push-Sum works. The reader is referred to the cited literature for more detailed proofs on the general case. Given the topology of the network G, choose a column stochastic matrix P conformant to G; i.e., $P_{ij} = 0$ if there is no directed edge (j, i). If $(j, i) \in E$ we may still have $P_{ij} = 0$ meaning that although the channel is available the protocol chooses not to use it[1] . The initial values at the nodes are stacked in a vector $\boldsymbol{z}(0)$ and the goal is to compute the average $z_{ave} = \frac{\mathbf{1}^T \boldsymbol{z}(0)}{n}$. In Push-Sum, each node i maintains two values, a cumulative estimate of the sum $s_i(t)$ and a weight $u_i(t)$. We initialize

$$\boldsymbol{s}(0) = \boldsymbol{z}(0) \qquad \boldsymbol{u}(0) = \mathbf{1} \tag{5.6}$$

and the average estimate at each iteration is the ratio $\frac{s_i(t)}{u_i(t)}$. At each iteration, a node j splits its total sum $s_j(t)$ and weight $u_j(t)$ into shares $S_{j\to i}(t) = \big(P_{ij}s_j(t), P_{ij}u_j(t)\big)$ where $\sum_{i=1}^{n} P_{ij} = 1$, and sends the corresponding share $S_{j\to i}(t)$ to each neighbour i. A receiving node just sums up all the incoming shares from its neighbours. At each time, the estimate of the average at each node is $\tilde{z}_i(t) = \frac{s_i(t)}{u_i(t)}$. In vector form the state evolves as

$$\boldsymbol{s}(t) = P\boldsymbol{s}(t-1) \tag{5.7}$$

$$\boldsymbol{u}(t) = P\boldsymbol{u}(t-1) \tag{5.8}$$

$$\tilde{\boldsymbol{z}}(t) = \frac{\boldsymbol{s}(t)}{\boldsymbol{u}(t)} \tag{5.9}$$

where the division of $\boldsymbol{s}(t)$ and $\boldsymbol{u}(t)$ in (5.9) is element-wise. The algorithm relies on a mass conservation property to converge to the correct average. Specifically, for all iterations we can verify that through the updates (5.7), mass is conserved in the

[1] As long as this does not break strong connectivity of G.

sense that for all t

$$\sum_{i=1}^{n} s_i(t) = \sum_{i=1}^{n} z_i(0) = \mathbf{1}^T \boldsymbol{z}(0) = n z_{ave} \tag{5.10}$$

$$\sum_{i=1}^{n} u_i(t) = n. \tag{5.11}$$

To see why Push-Sum correctly computes the average, notice that since G is assumed to be strongly connected and P conforms to G, matrix P is a scrambling matrix[2] and P^t converges to a rank-1 matrix exponentially fast [34, 73]. Let P^∞ be the limit of P^t as $t \to \infty$. Matrix P^∞ will be column stochastic with all columns the same. At any node i we will have

$$\tilde{z}_i(\infty) = \frac{\left[P^\infty \boldsymbol{s}(0)\right]_i}{\left[P^\infty \boldsymbol{u}(0)\right]_i} = \frac{\left[P^\infty \boldsymbol{z}(0)\right]_i}{\left[P^\infty \mathbf{1}\right]_i} = \frac{\sum_{j=1}^{n} P_{ij}^\infty z_j(0)}{\sum_{j=1}^{n} P_{ij}^\infty} \tag{5.12}$$

$$= \frac{P_{i1}^\infty \sum_{j=1}^{n} z_j(0)}{P_{i1}^\infty \sum_{j=1}^{n} 1} = \frac{\sum_{j=1}^{n} z_j(0)}{n} = \frac{\mathbf{1}^T \boldsymbol{z}(0)}{n}. \tag{5.13}$$

We used the fact that elements in the same row of P^∞ are the same i.e. $P_{i1}^\infty = P_{ij}^\infty, \forall j$. For a formal proof see [7]. Observe that convergence is achieved without the need to know the stationary distribution of P or the size of the network n at every node. This is possible because the weights $\boldsymbol{u}(t)$ are approaching the stationary distribution of P as the algorithm progresses. Notice also that even though we assumed that P is fixed for simplicity, the same argument for convergence to the average applies if $P = P(t)$. The only difference is that we need the forward product $T(1,t) = P(t)P(t-1)\cdots P(1)$ to approach a limit P^∞ as $t \to \infty$.

For the sequel, it will be useful to also note that from (5.10) and (5.9), convergence implies that

$$\frac{s_j(t)}{u_j(t)} \to \frac{1}{n} \sum_{i=1}^{n} s_i(t). \tag{5.14}$$

[2] A stochastic matrix is scrambling if any two rows share a column where both rows have a positive entry [43].

Finally, for the case where P remains fixed, we can obtain an eigenvalue bound on the convergence rate of the algorithm. For more details refer to Chapter 6. Here we just mention the result. In general, matrix P represents a non-reversible, irreducible Markov chain and we have

$$\left\| \boldsymbol{\pi} - \left[P^t\right]_{:,i} \right\|_1 \leq \sqrt{\frac{\lambda_2^t}{\pi_i}}, \tag{5.15}$$

where $\boldsymbol{\pi}$ is the stationary distribution vector of P and λ_2 is the second largest eigenvalue of the *lazy additive reversibilization* of P as explained in [89] and [34].

5.4 Push-Sum Distributed Dual Averaging (PS-DDA)

Based on the criteria set in this Chapter, neither DDA nor DSG or other similar schemes are practical algorithms since they are synchronous and rely on bidirectional communication with doubly stochastic consensus matrixs P. Here we propose a new algorithm called Push-Sum Distributed Dual Averaging or PS-DDA, an algorithm which employs Push-Sum for the coordination of the nodes. PS-DDA converges at the same asymptotic rate as DDA and DSG but at the same time it has all the desired practical features. The description of the algorithm is followed by a discussion of issues that one needs to consider from an implementation stand-point together with an experimental evaluation. The mathematical analysis of convergence is deferred until the end of the chapter.

Equipped with the Push-sum averaging protocol, the *Push-sum Distributed Dual Averaging* (PS-DDA) algorithm works as follows:

$$u_i(t+1) = \sum_{j=1}^{n} P_{ij} u_j(t) \tag{5.16}$$

$$\boldsymbol{z_i}(t+1) = \sum_{j=1}^{n} P_{ij} (\boldsymbol{z_j}(t) + \boldsymbol{g_j}(t)) \tag{5.17}$$

$$\boldsymbol{w_i}(t+1) = \Pi_{\mathcal{W}}^{\psi} \left[\frac{\boldsymbol{z_i}(t+1)}{u_i(t+1)}, a(t) \right] \tag{5.18}$$

$$\overline{\boldsymbol{w}}_{\boldsymbol{i}}(t+1) = \frac{1}{t+1} \sum_{s=1}^{t+1} \boldsymbol{w_i}(s) \tag{5.19}$$

where as usual $\boldsymbol{g_i}(t)$ is a subgradient of $f_i(\boldsymbol{w})$ at point $\boldsymbol{w_i}(t)$ and $a(t)$ is a non-increasing sequence of step sizes. Observe that to retrieve the correct cumulative

gradient we need to divide the dual variable $\boldsymbol{z_i}(t)$ at every node by the appropriate weight. The weight variables $u_i(t)$ automatically rescale the dual variables to account for the case where the stationary distribution of P is non-uniform. In this way, PS-DDA more naturally accommodates the challenges posed by a real implementation. However, as we discuss in the next section, there still exist implementation subtleties that need to be addressed. Notice also that, contrary to DDA, we first locally integrate the most recent gradient and then execute the consensus step. This leads to a cleaner derivation, but also has another advantage. In the case of a complete graph with a doubly stochastic matrix P, (5.17) performs perfect averaging and the network error is zero as one would expect. This is not the case with DDA. We elaborate on this in Chapter 7.

Remark: We describe and analyze here PS-DDA with a fixed consensus matrix P. This is done because it allows for a more elegant closed-form expression for the convergence rate. If the fixed consensus matrix is replaced by a time varying matrix $P(t)$, an asynchronous version of PS-DDA can be obtained where each node decides independently when to transmit a new message and to which neighbour. For our experiments the asynchronous version is used and as discussed in Chapter 6, PS-DDA converges to the right solution with time varying consensus matrices as well. Finding a clean expression for the rate is more difficult in that case however, since the analysis requires bounding a forward product of random matrices.

In Section 5.8 we prove that PS-DDA converges to the solution of (1.8) at a rate given in the following theorem:

Theorem 5.1. *Consider a strongly connected graph G and a consensus matrix P that respects the structure of G. There exists a value $c > 0$ such that for all t and all i, we have $\sum_{s=1}^{n}[P^t]_{is} \geq c$. Let also $\pi^* = \min_s\{\pi_s\} > 0$ be the minimum entry in $\boldsymbol{\pi}$, the stationary distribution of P. Let λ_2 be the second largest eigenvalue of P. Suppose we want to solve the L-Lipschitz continuous problem* (1.8)*. The PS-DDA algorithm* (5.16)-(5.19) *using a strongly convex function $\psi(\boldsymbol{w})$ with respect to norm*

$\|\cdot\|$ *such that* $\psi(\boldsymbol{w}^*) \leq R^2$ *and choosing step sizes*

$$a(t) = \frac{R}{L\sqrt{1 + \frac{8+4n}{c\sqrt{\pi^*}(1-\sqrt{\lambda_2})}}} \frac{1}{\sqrt{t}}, \tag{5.20}$$

converges for every node $i \in V$ *to the optimum* $\boldsymbol{w}^* \in \mathcal{W}$ *of* (1.8) *as*

$$F(\overline{\boldsymbol{w}}_{\boldsymbol{i}}(T)) - F(\boldsymbol{w}^*) \leq 2RL\sqrt{1 + \frac{8+4n}{c\sqrt{\pi^*}(1-\sqrt{\lambda_2})}} \frac{1}{\sqrt{T}}. \tag{5.21}$$

For a fixed network and consensus protocol, the convergence rate is $O\left(\frac{1}{\sqrt{T}}\right)$ which is time optimal. The constant term reveals the dependence on the connectivity and specific consensus protocol through λ_2 as well as the network size n. The dependence on n, c and π^* is pessimistic and arises due to the fact that we are analyzing products or column rather than row stochastic matrices.

5.5 Implementation Remarks

Here we summarize a number of issues that arise and must be addressed in a real implementation. We assume that each node has the ability to send and receive messages and also to poll its buffer for incoming messages that have not been received yet. In our implementation, discussed in Section 5.6 below, these features are provided by the *message passing interface* (MPI) library [37].

5.5.1 One directional communication

PS-DDA uses one-directional communications so that each node sends information without expecting replies before it updates. Specifically, node i rescales $u_i(t)$ and $\boldsymbol{z_i}(t)$ by P_{ji} and send the rescaled values to its neighbor j. The receiver forms the sum of the received messages u and $\boldsymbol{z}$. Notice that this mode of operation may not be obvious from the algorithmic description (5.16)-(5.19). However, the one directional nature of the algorithm is revealed by the fact that the only restriction on matrix P is that it is column stochastic, and that node i only has control over the values in the i-th row of the matrix.

5.5.2 Numerical instability

In an asynchronous implementation, each node independently decides when to send a message to its neighbours. Moreover, a message from node i to node

j is a quantity ($\boldsymbol{z_i}$ or $\boldsymbol{u_i}$) discounted by P_{ji}. If, for some reason, node i finishes its iterations faster than its neighbours, its incoming message buffer will be empty most of the time and it will not update (5.16). It is then possible that $u_i(t)$ will become very small due to repeated rescaling by $P_{ji} < 1$ whenever node i transmits something. In practice after a few thousand iterations we hit the limits of numerical precision. To prevent this from happening, it is sufficient to add a condition that node i does not transmit if $u_i(t)$ is too small. We use a threshold of 10^{-4} in our experiments and find that this value suffices to avoid any numerical instabilities. If transmitting is prevented consensus slows down. How often this condition becomes true and consequently how much Push-Sum is delayed will depend on the relative difference of processing speed between nodes. A theoretical characterization of the effect could be very interesting.

5.5.3 Step-size de-synchronization

By allowing the consensus matrix P to be time-varying, the algorithm becomes asynchronous. Note, however, that the description (5.16)-(5.19) is in terms of iterations and the step size at each node is set as $a(t) = O\left(\frac{1}{\sqrt{t}}\right)$. If each node operates at its own pace, maintaining a local iteration counter, the step sizes can end up being very different at different nodes. This situation is problematic in practice because the incoming messages will be discounted by step sizes that differ by orders of magnitude at different nodes simply because the nodes are at different stages of the computation. To prevent this from happening, we update the step size based on actual wall clock time instead of iterations. Each node maintains a secondary iteration counter τ which is incremented by 1 every 100ms. This way if a node experiences a delay and finishes an iteration in, say, 1 second, then it will set $\tau^{\text{new}} = \tau^{\text{old}} + 10$ and $a(\tau^{\text{new}}) = \frac{1}{\sqrt{\tau^{\text{new}}}}$. In a somewhat controlled environment like a cluster, where nodes begin the computation at effectively the same time, this solution suffices. A theoretical analysis of the effect of de-synchronized step-sizes is an important topic for future work. Note also that the increment of 100ms is somewhat adhoc for our specific cluster and might need some configuration in general. To comply with the

theory, the increment should be long enough for a node to be able to complete one iteration involving one gradient step.

5.5.4 Incoming message handler

Clusters are shared resources, and it is not uncommon for one node to be simultaneously assigned to process multiple tasks for different users. Moreover, network throughput may vary significantly depending on other background traffic. Both of those factors can result in some nodes transmitting more frequently than others. Consequently, when the (slow) receiver polls its incoming message buffer it may find multiple messages from the same neighbor. Those messages would have been sent at different moments in time, but they arrive and are processed during the same update due to, e.g., communication delays. It is an interesting question how to handle such incoming information. At the one extreme, a node can wait until it receives at least one message per neighbour. We have found that this approach does not work well in practice and goes against the desired asynchronous operation. At the other extreme, a node may empty its incoming buffer and sum all the incoming messages at the beginning of each iteration. Then the sums in (5.16) and (5.17) are over all messages in the buffer, not over all nodes. The danger in this case is to run into a producer-consumer scenario where one node continuously sends new information while another node continuously receives it. In this scenario, the receiver may never exit the receive mode to continue with local computations. Although this could happen in principle, in practice we did not observe this behaviour. It would be interesting to explore if assigning less weight to older messages can speedup convergence.

5.5.5 Communicator saturation

Depending on the CPU and the problem being solved, it is possible that the local gradient and projection computations will be very quick, and a node may poll its communicator (at the lowest level a TCP socket) very frequently for new messages. This is not uncommon in practice and can result in the system stalling simply because the network cannot be polled too frequently. This issue can also

be easily prevented by making sure that a minimal amount of time always lapses between polls. A value of 10ms was found be sufficient in our implementation.

5.6 Experimental Evaluation

5.6.1 Benchmark Problem and Setup

To complement the discussion above, we report the results of experiments with DDA and PS-DDA on a cluster of 15 nodes. Each node has a 3.2 GHz Pentium 4HT processor and 1 GB of memory, and they are physically connected in a star topology through an Ethernet switch that allows for roughly 11 MB/sec throughput per node. Our implementation is in C++ using the send and receive functions of OpenMPI v1.4.4 for communication. The Armadillo v2.3.91 library, linked to LAPACK and BLAS, is used for efficient numerical computations.

As a benchmark problem we seek to minimize a sum of quadratics with

$$f_i(\boldsymbol{w}) = \sum_{j=1}^{M} (\boldsymbol{w} - \boldsymbol{x}_{j|i})^T (\boldsymbol{w} - \boldsymbol{x}_{j|i}) \tag{5.22}$$

where $\boldsymbol{w} \in \mathbb{R}^{5,000}$, $M = 500$ and $\boldsymbol{x}_{j|i}$ is the centre of the j-th quadratic of node i. The data $\boldsymbol{x}_{j|i}$ are chosen so that the minima of the components $f_i(\boldsymbol{w})$ at each node are very different and coordination is essential to obtain an accurate optimizer of $F(\boldsymbol{w})$.

5.6.2 Doubly Stochastic Matrices cannot be maintained in Practice

The first experiment aims to illustrate that it is not possible to guarantee that the updates $P(t)$ are doubly stochastic due to network delays and nodes that experience different amounts of communication overhead and workload, even if the consensus protocol is initially designed to be doubly stochastic. Consequently, the standard consensus algorithm is not an averaging algorithm anymore and convergence to the right solution is lost.

To illustrate the point, we solve problem (5.22) on a 15-node graph with neighborhood structure defined through P so that node 1 has a much higher degree than all other nodes, as shown in Figure 5–4. We expect that node 1 will spend more time communicating than the others, and its iterations will take more time to complete.

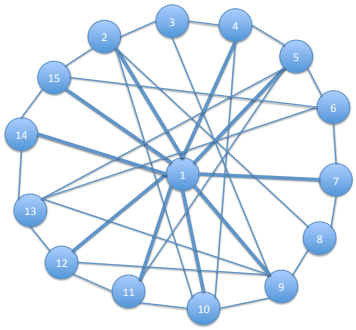

Figure 5–4: The unbalanced communication topology used in the first set of experiments. In this topology, node 1 has many more neighbors than the others, and consequently, it spends more time communicating than other nodes. Edges connected to node 1 have a heavier weight than other edges.

We select $P = I - \frac{D-A}{d_{max}+1}$ where, D is a diagonal matrix containing the node degrees (excluding self loops), A is the symmetric graph adjacency matrix and d_{max} is the maximum node degree. It can be verified that P is doubly stochastic. Figure 5–5 shows the evolution of the objective value $F(\overline{\boldsymbol{w}}_{\boldsymbol{i}}(t))$ at each node when we solve the problem using DDA (i.e., with asynchronous consensus updates that are not doubly stochastic, not asynchronous Push-Sum). Note that if the true consensus matrix used at each iteration was indeed the doubly stochastic matrix P, then all nodes would converge to the average consensus solution. However, this is clearly not the case in practice. As the figure shows, node 1 being slower than the rest, cannot coordinate with the team. The resulting consensus protocol is no longer averaging and we have disagreement. On the other hand, if we make use of the asynchronous Push-Sum weights and run PS-DDA, then as Figure 5–6 shows consensus does work and the objective is minimized as desired.

5.6.3 Comparison with AllReduce

In the next experiment, we compare the performance of consensus-based PS-DDA with a solution that uses MPI's specialized AllReduce communication capabilities. The latter is available in high-performance computing clusters supporting MPI and allows for all the nodes to exchange information with each other and obtain the true average $\boldsymbol{z}$ at each iteration. AllReduce is designed as an efficient primitive

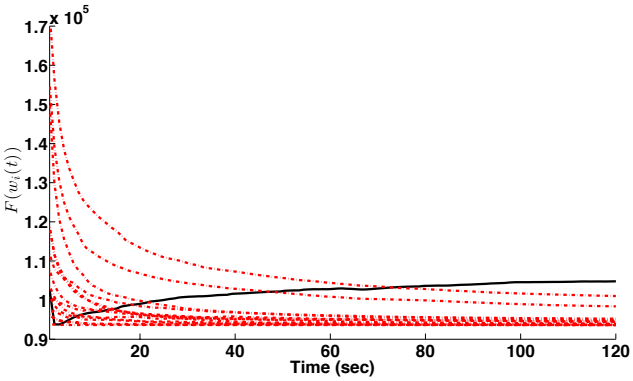

Figure 5–5: Network of 15 nodes solving problem (5.22) on a graph where node 1 has many more neighbours that the rest of the nodes. Because of asynchronism and delays, the resulting updates do not correspond to a doubly stochastic matrix, and the nodes do not reach consensus on the average. Consequently, node 1 (solid black line) does not converge to right solution and the algorithm does not solve the problem.

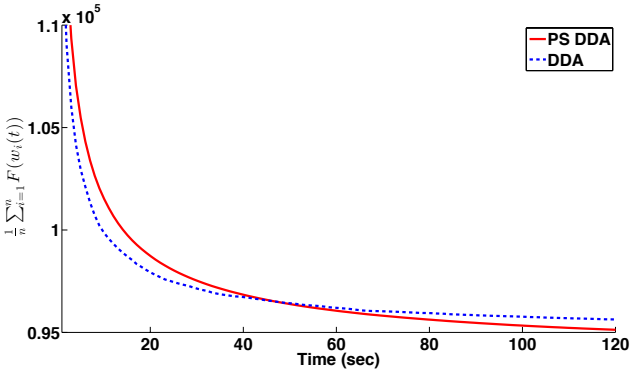

Figure 5–6: (Blue dashed) The average performance of the team produced by running DDA and ignoring the weights (average of plots inFigure 5–5). (Red Line) When the weights w are not kept to 1 and true asynchronous PS-DDA is running, the network converges to the right solution despite the asymmetry of communication overhead of its nodes.

to allow all nodes in the network to exchange information in one function call. However, the call is blocking. Hence, using AllReduce implicitly requires a synchronous approach and all nodes must call AllReduce simultaneously. This solution resembles algorithms based on the popular Map-Reduce approach that have been developed recently for distributed optimization [27]. We compare this solution with our PS-DDA implementation operating in blocking and asynchronous mode, executing over a communication topology G corresponding to the complete graph. In blocking mode, at each iteration each node blocks in a receive call until one message from each neighbour arrives. This turns PS-DDA into a synchronous algorithm and each node computes the exact average $z(t)$ at every iteration.

Figure 5–7 shows two sets of experiments. Initially, we solve problem (5.22) with all three algorithms in a delay-free environment. As we see from the solid lines at the bottom of the figure, the AllReduce implementation, is slightly faster than the blocking PS-DDA which again is just slightly faster than the asynchronous PS-DDA implementation. Then, to illustrate the benefits of asynchronism, we artificially slow down one of the nodes by adding a 0.5 second pause after each local gradient computation. In practice, a node could be slowed down because it is spending cycles on unrelated tasks, it could have more data to process, or it could simply have a less powerful CPU. It should come to no surprise that the synchronous algorithms (purple and red dashed lines) are both severely impacted as their overall computation runs at the speed of the slowest node. However, the asynchronous algorithm (green dashed line) is not as affected. The fast nodes quickly converge to the solution and the slow node is pulled to the right solution.

5.7 Concluding Remarks and Future Work

Consensus-based distributed optimization algorithms are an attractive alternative for solving large-scale problems over peer-to-peer networks. The advantages of such an approach are increased robustness to node failures, scalability and asynchronism. The main difficulty comes from the lack of sophisticated communication infrastructure, and performance heavily depends on the network properties.

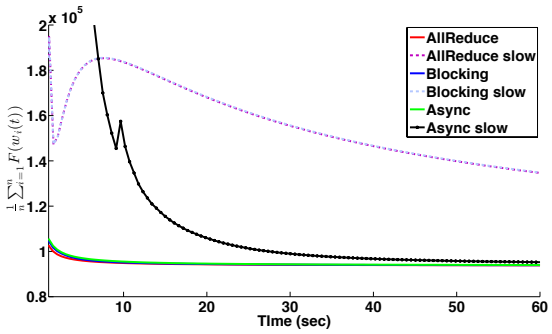

Figure 5–7: (Red,Blue,Green solid) Average progress towards the solution of problem (5.22) with AllReduce, PS-DDA blocking and PS-DDA asynchronous algorithms when all nodes are running at full speed. (Red,Blue,Green dashed) Average progress when one node is artificially slowed down to have a 0.5 second delay at each local iteration. In this case the synchronous algorithms are running at the speed of the slow node and their curves are overlapping. The asynchronous algorithms is significantly less affected by the slow node.

We identify averaging, one directional communication, and asynchronism as the three key ingredients that a consensus algorithm should contain for a reliable and efficient practical implementation. We note that the benefit of an algorithm that accommodates real network constraints comes at the price of more difficult mathematical analysis. We also discuss implementation issues we have observed/experienced, including numerical instabilities and de-synchronizing the step sizes of different nodes. For these issues we explain the root cause and side effects as well as ways to prevent them from happening. Our discussion is complemented by simulations on a real cluster that illustrate how the key ingredients mentioned above yield practical algorithms that work correctly even in adverse circumstances where the communication overhead of the nodes is not equally distributed or where nodes experience different and unbalanced workloads.

In the future, it would be important to conduct a more thorough theoretical investigation of the effect of de-synchronizing step sizes, as well as thresholding the Push Sum weights to avoid numerical instabilities. Moreover, it would be interesting to scale the experiments to a significantly larger cluster with many more than the 15 nodes used in the results reported here.

5.8 Proof of Theorem 5.1

We start by noting that Lemmas 2.2 and 2.3 are derived based on basic convexity arguments and properties of the objective function and thus hold unaltered for PS-DDA as well. Let $\pi^* = \min_s\{\boldsymbol{\pi}_s\}$ denote the minimum entry in the stationary distribution of P. We know there exists a constant $c > 0$ such that for all t and all i, $\sum_{s=1}^{n}[P^t]_{is} \geq c$ since P is assumed to be a scrambling matrix (this is necessary since G is strongly connected so that there is a directed path between any two nodes). Without loss of generality we initialize $\boldsymbol{z_i}(0) = 0$ and $\boldsymbol{g_i}(0) = 0$ for all i.

We start by finding expressions for $\boldsymbol{z_i}(t)$ and $\overline{\boldsymbol{z}}(t) = \frac{1}{n}\sum_{i=1}^{n}\boldsymbol{z_i}(t)$ based on the gradients. It is convenient to use matrix notation to derive the recursion. Let us just momentarily treat the $\boldsymbol{z_i}$ variables as scalars to keep the notation simple. We stack all the z variables in a vector $\boldsymbol{Z}(t)$ and all local gradients in a vector $\boldsymbol{G}(t)$. From (5.17) we have

$$\begin{aligned}
\boldsymbol{Z}(t) =& P(\boldsymbol{Z}(t-1) + \boldsymbol{G}(t-1)) && (5.23)\\
=& P^2\boldsymbol{Z}(t-2) + P^2\boldsymbol{G}(t-2) + P\boldsymbol{G}(t-1) && (5.24)\\
=& \cdots && (5.25)\\
=& \sum_{r=1}^{t-1} P^{t-r}\boldsymbol{G}(r) && (5.26)
\end{aligned}$$

where we used the fact that the initial conditions are zero. Separating the i-th row for node i we have

$$\boldsymbol{z_i}(t) = \sum_{r=1}^{t-1}\sum_{j=1}^{n}[P^{t-r}]_{ij}\boldsymbol{g_j}(r). \tag{5.27}$$

Next, using (5.27), we derive an expression for the average dual variable $\overline{\boldsymbol{z}}(t)$:

$$\overline{\boldsymbol{z}}(t) = \frac{1}{n}\sum_{i=1}^{n} \boldsymbol{z_i}(t) \tag{5.28}$$

$$= \frac{1}{n}\sum_{i=1}^{n}\sum_{r=1}^{t-1}\sum_{j=1}^{n}[P^{t-r}]_{ij}\boldsymbol{g_j}(r) \tag{5.29}$$

$$= \sum_{r=1}^{t}\frac{1}{n}\sum_{j=1}^{n}\boldsymbol{g_j}(r)\sum_{i=1}^{n}[P^{t-r}]_{ij} \tag{5.30}$$

$$= \sum_{r=1}^{t-1}\frac{1}{n}\sum_{j=1}^{n}\boldsymbol{g_j}(r). \tag{5.31}$$

For the last equality we used the fact that each P^{t-r} is a column stochastic matrix as a product of column stochastic matrices. We also need the sequence $\{\boldsymbol{y(t)}\}_{t=1}^{\infty}$ defined by the projection of $\overline{\boldsymbol{z}}(t)$:

$$\boldsymbol{y(t)} = \Pi_{\mathcal{W}}^{\psi}\left(\overline{\boldsymbol{z}}(t), a(t)\right) \tag{5.32}$$

and the running average $\overline{\boldsymbol{y}}(T) = \frac{1}{T}\sum_{t=1}^{T}\boldsymbol{y}(t)$. Using standard convexity arguments and Lemma 2.1, we can show that for any $\boldsymbol{w^*} \in \mathcal{W}$ (for a detailed derivation see e.g. [32])

$$F(\overline{\boldsymbol{w}}_{\boldsymbol{j}}(T)) - F(\overline{\boldsymbol{w}}^{*}) \leq \frac{1}{T}\sum_{t=1}^{T}\frac{1}{n}\sum_{i=1}^{n}\left\langle \boldsymbol{g_i}(t), \boldsymbol{w_i}(t) - \boldsymbol{w^*}\right\rangle \tag{5.33}$$

$$+ \frac{1}{T}\sum_{t=1}^{T}\frac{L}{n}\sum_{i=1}^{n}\left\|\boldsymbol{y}(t) - \boldsymbol{w_i}(t)\right\| \tag{5.34}$$

$$+ \frac{L}{T}\sum_{t=1}^{T}\left\|\boldsymbol{w_j}(t) - \boldsymbol{y}(t)\right\|. \tag{5.35}$$

We will bound the three RHS terms separately. For (5.33) we split the inner product by adding and subtracting $\boldsymbol{y}(t)$:

$$\sum_{i=1}^{n} \langle \boldsymbol{g_i}(t), \boldsymbol{w_i}(t) - \boldsymbol{w^*} \rangle = \sum_{i=1}^{n} \langle \boldsymbol{g_i}(t), \boldsymbol{y}(t) - \boldsymbol{w^*} \rangle \tag{5.36}$$

$$+ \sum_{i=1}^{n} \langle \boldsymbol{g_i}(t), \boldsymbol{w_i}(t) - \boldsymbol{y}(t) \rangle \tag{5.37}$$

$$= \left\langle \sum_{i=1}^{n} \boldsymbol{g_i}(t), \boldsymbol{y}(t) - \boldsymbol{w^*} \right\rangle \tag{5.38}$$

$$+ \sum_{i=1}^{n} \langle \boldsymbol{g_i}(t), \boldsymbol{w_i}(t) - \boldsymbol{y}(t) \rangle . \tag{5.39}$$

To bound (5.38) we recall the definition (5.32) of $\boldsymbol{y}(t)$ and the expression (5.31) to see that

$$\boldsymbol{y}(t) = \Pi_{\mathcal{W}}^{\psi} \left[\sum_{r=1}^{t-1} \frac{1}{n} \sum_{i=1}^{n} g_i(r), a(t-1) \right]. \tag{5.40}$$

Now, observe that

$$\sum_{t=1}^{T} \frac{1}{n} \left\langle \sum_{i=1}^{n} \boldsymbol{g_i}(t), \boldsymbol{y}(t) - \boldsymbol{w^*} \right\rangle$$

$$= \sum_{t=1}^{T} \left\langle \frac{1}{n} \sum_{i=1}^{n} \boldsymbol{g_i}(t), \Pi_{\mathcal{W}}^{\psi} \left[\sum_{r=1}^{t-1} \frac{1}{n} \sum_{i=1}^{n} \boldsymbol{g_i}(r), a(t-1) \right] - \boldsymbol{w^*} \right\rangle. \tag{5.41}$$

Invoke Lemma 2.2 with $\frac{1}{n} \sum_{i=1}^{n} \boldsymbol{g_i}(t)$ as the vector sequence to get

$$\sum_{t=1}^{T} \frac{1}{n} \left\langle \sum_{i=1}^{n} \boldsymbol{g_i}(t), \boldsymbol{y}(t) - \boldsymbol{w^*} \right\rangle \leq \frac{L^2}{2} \sum_{t=1}^{T} a(t-1) + \frac{1}{a(T)} \psi(\boldsymbol{w^*}). \tag{5.42}$$

For term (5.39), using the gradient magnitude bound, the definition (5.18) of $\boldsymbol{w_i}(t)$ and Lemma 2.3 we get

$$\sum_{i=1}^{n} \langle \boldsymbol{g_i}(t), \boldsymbol{w_i}(t) - \boldsymbol{y}(t) \rangle \leq \sum_{i=1}^{n} \|\boldsymbol{g_i}(t)\| \, \|\boldsymbol{w_i}(t) - \boldsymbol{y}(t)\| \tag{5.43}$$

$$\leq \sum_{i=1}^{n} L \left\| \Pi_{\mathcal{W}}^{\psi} \left[\frac{\boldsymbol{z_i}(t)}{u_i(t)}, a(t-1) \right] - \Pi_{\mathcal{W}}^{\psi} \left[\overline{\boldsymbol{z}}(t), a(t-1) \right] \right\| \tag{5.44}$$

$$\leq \sum_{i=1}^{n} L a(t-1) \left\| \overline{\boldsymbol{z}}(t) - \frac{\boldsymbol{z_i}(t)}{u_i(t)} \right\|. \tag{5.45}$$

Terms (5.34) and (5.35) are bounded similarly. Combining (5.33)-(5.35), (5.42), (5.45) and Lemma 2.3 we can show that

$$\begin{aligned} F(\overline{\boldsymbol{w}}_{\boldsymbol{j}}(T)) - F(\boldsymbol{w}^*) \leq & \frac{L^2}{2T}\sum_{t=1}^{T} a(t-1) + \frac{1}{Ta(T)}\psi(\boldsymbol{w}^*) \\ & + \frac{2L}{nT}\sum_{t=1}^{T}\sum_{i=1}^{n} a(t-1)\left\|\overline{\boldsymbol{z}}(t) - \frac{\boldsymbol{z_i}(t)}{u_i(t)}\right\| \\ & + \frac{L}{T}\sum_{t=1}^{T} a(t-1)\left\|\overline{\boldsymbol{z}}(t) - \frac{\boldsymbol{z_j}(t)}{u_j(t)}\right\|. \end{aligned} \tag{5.46}$$

To complete the proof we thus need to bound each network error term $\left\|\overline{\boldsymbol{z}}(t) - \frac{\boldsymbol{z_k}(t)}{u_k(t)}\right\|$ for any node k. From (5.16) we see that $u_k(t) = \sum_{s=1}^{n}[P^t]_{ks}$. Now we proceed with the bound:

$$\left\|\overline{\boldsymbol{z}}(t) - \frac{\boldsymbol{z_k}(t)}{u_k(t)}\right\| = \left\|\sum_{r=1}^{t-1}\frac{1}{n}\sum_{j=1}^{n}\boldsymbol{g_j}(r) - \frac{\sum_{r=1}^{t-1}\sum_{j=1}^{n}[P^{t-r}]_{kj}\boldsymbol{g_j}(r)}{\sum_{s=1}^{n}[P^t]_{ks}}\right\|_* \tag{5.47}$$

$$\leq \sum_{r=1}^{t-1}\sum_{j=1}^{n}\|\boldsymbol{g_j}(r)\|_*\left|\frac{1}{n} - \frac{[P^{t-r}]_{kj}}{\sum_{s=1}^{n}[P^t]_{ks}}\right| \tag{5.48}$$

$$\leq L\sum_{r=1}^{t-1}\sum_{j=1}^{n}\left|\frac{[P^{t-r}]_{kj}}{\sum_{s=1}^{n}[P^t]_{ks}} - \frac{1}{n}\right|. \tag{5.49}$$

We now show that the term in the absolute value remains bounded.

$$\left|\frac{[P^{t-r}]_{kj}}{\sum_{s=1}^{n}[P^t]_{ks}} - \frac{1}{n}\right| = \left|\frac{n[P^{t-r}]_{kj} - \sum_{s=1}^{n}[P^t]_{ks}}{n\sum_{s=1}^{n}[P^t]_{ks}}\right| \tag{5.50}$$

$$= \left|\frac{\sum_{s=1}^{n}[P^{t-r}]_{kj} - \sum_{s=1}^{n}[P^t]_{ks}}{n\sum_{s=1}^{n}[P^t]_{ks}}\right| \tag{5.51}$$

$$= \left|\frac{\sum_{s=1}^{n}\left([P^{t-r}]_{kj} - \boldsymbol{\pi}_k + \boldsymbol{\pi}_k - [P^t]_{ks}\right)}{n\sum_{s=1}^{n}[P^t]_{ks}}\right| \tag{5.52}$$

$$\leq \frac{\sum_{s=1}^{n}\left(\left|[P^{t-r}]_{kj} - \boldsymbol{\pi}_k\right| + \left|\boldsymbol{\pi}_k - [P^t]_{ks}\right|\right)}{n\sum_{s=1}^{n}[P^t]_{ks}} \tag{5.53}$$

The bound (5.15) gives also an exponential convergence rate for each individual element of P^t, so

$$\left|\frac{[P^{t-r}]_{kj}}{\sum_{s=1}^{n}[P^t]_{ks}}-\frac{1}{n}\right| \leq \frac{\sum_{s=1}^{n}\sqrt{\frac{\lambda_2^{t-r}}{\boldsymbol{\pi}_j}}+\sum_{s=1}^{n}\sqrt{\frac{\lambda_2^{t}}{\boldsymbol{\pi}_s}}}{n\sum_{s=1}^{n}[P^t]_{ks}} \tag{5.54}$$

$$\leq \frac{2n\frac{1}{\min_s\{\sqrt{\boldsymbol{\pi}_s}\}}\sqrt{\lambda_2^{t-r}}}{n\sum_{s=1}^{n}[P^t]_{ks}} \tag{5.55}$$

$$\leq \frac{2\sqrt{\lambda_2^{t-r}}}{c\sqrt{\pi^*}} \tag{5.56}$$

where we used the fact that $\lambda_2 < 1$. We thus conclude that

$$\|\overline{\boldsymbol{z}}(t)-\boldsymbol{z_k}(t)\|_* \leq L\sum_{r=1}^{t-1}\sum_{j=1}^{n}\frac{2\sqrt{\lambda_2^{t-r}}}{c\sqrt{\pi^*}} \tag{5.57}$$

$$= \frac{2Ln}{c\sqrt{\pi^*}}\sum_{r=1}^{t-1}\sqrt{\lambda_2^{t-r}} \tag{5.58}$$

$$= \frac{2Ln}{c\sqrt{\pi^*}}\frac{\sqrt{\lambda_2}-(\sqrt{\lambda_2})^t}{1-\sqrt{\lambda_2}} \tag{5.59}$$

$$\leq \frac{2Ln}{c\sqrt{\pi^*}}\frac{1}{1-\sqrt{\lambda_2}} \tag{5.60}$$

$$\tag{5.61}$$

where we used the formula for a finite geometric sum. Now we can go back to (5.46) to get

$$\begin{aligned} F(\overline{\boldsymbol{w}}_{\boldsymbol{j}}(T))-F(\boldsymbol{w}^*) \leq & \frac{L^2}{2T}\sum_{t=1}^{T}a(t-1)+\frac{1}{Ta(T)}\psi(\boldsymbol{w}^*) \\ & +\frac{2L}{T}\frac{2L}{c\sqrt{\pi^*}}\frac{1}{1-\sqrt{\lambda_2}}\sum_{t=1}^{T}a(t-1) \\ & +\frac{L}{T}\frac{2Ln}{c\sqrt{\pi^*}}\frac{1}{1-\sqrt{\lambda_2}}\sum_{t=1}^{T}a(t-1). \end{aligned} \tag{5.62}$$

With $\psi(\boldsymbol{w}^*) \leq R^2$

$$\begin{aligned} F(\overline{\boldsymbol{w}}_{\boldsymbol{j}}(T))-F(\boldsymbol{w}^*) \leq & \frac{L^2}{2T}\sum_{t=1}^{T}a(t-1)+\frac{R^2}{Ta(T)} \\ & +\frac{1}{T}\frac{2L^2(2+n)}{c\sqrt{\pi^*}}\frac{1}{1-\sqrt{\lambda_2}}\sum_{t=1}^{T}a(t-1). \end{aligned} \tag{5.63}$$

Finally, if we choose $a(t) = \frac{A}{\sqrt{t}}$ and minimize for A, noticing that $\sum_{t=1}^{T} \frac{1}{\sqrt{t}} \leq 2\sqrt{t}$ we arrive at the result in Theorem 5.1.

CHAPTER 6
Communication Delays

6.1 Introduction

This chapter studies an important and uncontrollable way in which a physical network can affect the behaviour of a distributed algorithm: communication delays[1] . For implementations of consensus-based optimization algorithms running on clusters, the issue of communication delays arises quite naturally. For example, in typical machine learning problems, the decision variable (and hence the message size) can quickly exceed many megabytes in size. During the time it takes to transmit such large messages, a modern processor can perform a significant amount of local processing of its own data, and the received information always appears to be delayed. In addition, cluster computing resources are typically shared among many users, and delays to one task are introduced if processors devote some of their cycles to other unrelated tasks. Finally, any network infrastructure is bound to have some fluctuation in its performance for reasons beyond our control. It is thus important first to model communication delays, and then incorporate those models in the analysis of consensus algorithms to understand the effects of delays.

We begin with a high level description of delays in a discrete time consensus algorithm and then summarize the results of this chapter. Then we proceed to develop these results in detail.

[1] This chapter is based on the previously published work [89, 90, 92].

6.1.1 Time Delayed Consensus

In its most common form, a consensus algorithm in discrete time updates the network state through iterations of the form

$$\boldsymbol{z}(t+1) = P\boldsymbol{z}(t) \tag{6.1}$$

where P is the consensus matrix. As already discussed in Section 2.1, a row stochastic matrix P is sufficient to drive the state $\boldsymbol{z}(t)$ to consensus on a value c that depends on the initial values $\boldsymbol{z}(0)$. If in addition P is doubly stochastic, the value c becomes the average of the initial values. For most of this chapter we will assume that P is row or doubly stochastic while at the end we will turn the discussion to Push-Sum a different consensus algorithm that was presented in Section 5.3. Push-Sum differs since it requires the exchange of extra information and uses column stochastic matrices.

We say that a message from node i to node j is delayed by b if it is received b time steps after it has been sent. For simplicity, let us first assume that each directed edge (i, j) experiences a fixed delay b_{ij} in the sense that each message leaving node i takes b_{ij} iterations to reach j. This suggests linear update consensus equations of the form

$$z_i(t+1) = \sum_{j=1}^{n} P_{ij} z_j(t - b_{ij}) \tag{6.2}$$

where $0 \leq b_{ij} \leq B$ for a network with bounded maximum delay $B < \infty$. This model describes a system with *fixed delays* and is analyzed in Section 6.2. Each b_{ij} can be thought of as the average delay experienced on that link.

A more realistic model relaxes the fixed delay assumption. Assume there exists a maximum delay bound B so that all messages are eventually delivered. We model random time-varying delays by sampling in $\{0, \ldots, B\}$ every time a new message is sent. Under this *random delay* model analysis is more complicated since in one iteration a node can receive multiple messages from the same sender. This situation is analyzed in Section 6.3. For both models we assume that there is no delay in self loop messages; i.e., each node always has access to its most recent local estimate.

For both cases we develop a formulation to describe the consensus update equations with a new matrix $\widehat{P}$ that is constructed from the initial matrix P. The construction involves an augmentation of the communication graph G with logical delay nodes. The idea itself is not new and appears in other work such as [21] and [56]. However our formulation is significantly different both in terms of the number of delay nodes we introduce as well as the equations it leads to. In our model, the intuition is that for every added unit of delay, the message is forced to pass through one more intermediate logical delay node before reaching its destination.

As a last comment, we emphasize that in this work all delays occur in discrete time. The reason is that both the consensus and the optimization algorithms we study are iterative and proceed in discrete iterations so the amount of delay for a message is also measured in terms of how many iterations of an algorithm are performed during the time it takes from transmission to delivery of a message.

6.1.2 Main Results

This chapter studies communication delays in discrete time and effect of delays on convergence of consensus algorithms; an integral part of distributed optimization algorithms. The results revolve around modelling delays, either assuming they are fixed or that they are random, and then analyzing how convergence rates are affected. In summary, this chapter discusses the following:

Fixed delay model. We first introduce a fixed delay model where transmissions over each directed link of a network experience some fixed amount of delay that does not exceed B. Starting with a consensus matrix P it is shown that consensus is still achieved in the presence of fixed delays at an exponential rate which depends on the second largest eigenvalue of $\widehat{P}$, the modified consensus algorithm accounting for delays. Furthermore, we use geometric arguments to show that the rate of convergence does not get worse by more than an factor of $O(B^2)$. In addition, we present an analysis of DDA in the presence of fixed delays to show how delays can affect distributed optimization.

Random delay model. Given a strongly connected graph G and any stochastic matrix P, we describe a construction for building a matrix $\widehat{P}$ that describes the consensus updates on G when each message experiences a random amount of delay that does not exceed B iterations.

Random delay consensus. If the initial matrix P on a graph G without delays is row stochastic, using the proposed random delay model, the consensus dynamics are captured by a sequence of matrices $\widehat{P}(t)$ which may contain all-zero rows. This means that although the consensus updates remain linear, convergence cannot be established based on standard theory for stochastic matrix products. We give a complete proof of convergence under the proposed random delay model.

Push-Sum consensus with delays. Push-Sum consensus is presented in Chapter 5 as the base for a new distributed optimization algorithm called PS-DDA. We show here that convergence properties of Push-Sum are not affected in the presence of delays. In particular, it is noteworthy that consensus on the average is guaranteed even in the presence of bounded random delays which is not the case for row stochastic matrices.

6.2 Fixed Communication Delays

We first analyze a model where the delay over each communication link does not vary with time. This is generally not true in practice but a fixed delay model can be appropriate in an average sense when the true delay does not fluctuate too much. Furthermore, the fixed delay model is instructive in showing how the delays affect rates of convergence for consensus and distributed optimization. For the rest of this section, whenever we talk about a quantity Q, such as a graph or a matrix, we use a hat (i.e., $\widehat{Q}$) to denote the transformed version of Q in the presence of delays.

6.2.1 Fixed Delay Model

Assume that in a given network G, for a directed link (i, j), every message from i to j is delayed by b_{ij} time units. We model this delay by replacing the link (i, j) with a chain of b_{ij} virtual *delay nodes* in the network, acting as relays between i

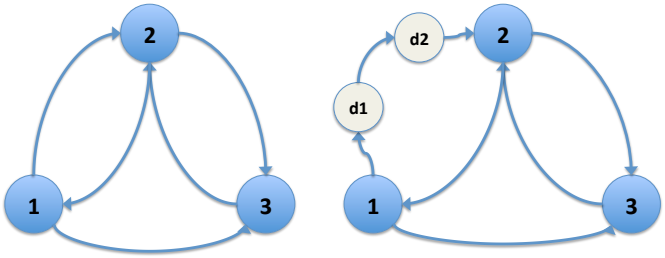

Figure 6–1: (left) A network with 3 nodes. (right) The network when we add a delay of 2 on the edge $(1, 2)$.

and j. This leads to a network $\widehat{G}$ which contains the original compute nodes, V, as well as $b = \sum_{(i,j)\in E} b_{ij}$ delay nodes. Our goal is to study the corresponding consensus algorithm running over $\widehat{G}$. We assume that a consensus matrix P in the delay-free network G is given so that in the presence of delays, the compute nodes still transmit and combine incoming messages using the weights provided by P. We begin by describing how to construct a stochastic matrix $\widehat{P}$ in the augmented space of $n + b$ nodes starting from a delay-free consensus matrix P. The matrix $\widehat{P}$ encodes communication of information between delay and compute nodes and has a stationary distribution $\widehat{\boldsymbol{\pi}}$ which is not uniform and depends on both P and the edge delays. We clarify that the augmentation of G with delay nodes is done just for the purpose of modelling and analysis; no physical delay nodes are actually added to the network.

To illustrate the construction of $\widehat{P}$ from P , consider a graph G with 3 nodes as in Figure 6–1. Suppose that the delay-free consensus algorithm is specified by the matrix

$$P = \begin{bmatrix} \frac{2}{3} & \frac{1}{3} & 0 \\ \frac{1}{6} & \frac{1}{3} & \frac{1}{2} \\ \frac{1}{6} & \frac{1}{3} & \frac{1}{2} \end{bmatrix}. \tag{6.3}$$

To model a fixed delay of $b_{12} = 2$ incurred by messages transmitted from node 1 to node 2, we augment G with two delay nodes $d_1^{1\to 2}, d_2^{1\to 2}$ so that information from 1 to 2 must pass through them *en route*. In the augmented graph $\widehat{G}$, the consensus algorithm is described by a row stochastic matrix $\widehat{P}$. Using the rows of P we write

$\widehat{P}$ as

$$\widehat{P} = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ d_1^{1\to 2} \\ d_2^{1\to 2} \end{array} \begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & d_1^{1\to 2} & d_2^{1\to 2} \end{array} \\ \left[\begin{array}{ccccc} \frac{2}{3} & \frac{1}{3} & 0 & 0 & 0 \\ 0 & \frac{1}{3} & \frac{1}{2} & 0 & \frac{1}{6} \\ \frac{1}{6} & \frac{1}{3} & \frac{1}{2} & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{array}\right] \end{array}. \tag{6.4}$$

Each compute node forms a convex combination of the incoming messages; e.g., in $\widehat{P}$, node 2 receives information from node $d_2^{1\to 2}$ with weight $\frac{1}{6}$ because $p_{2,1} = \frac{1}{6}$.

As it turns out, we can describe in matrix form an algorithm that will generate $\widehat{P}$ starting with any row stochastic P by inserting delays on edges one at a time. Suppose that after some delay insertions we have a matrix $\widehat{P}$ and want to add a delay of b_{ij} on edge (i, j). We replace edge (i, j) by a delay chain $d_1, d_2, \ldots, d_{b_{ij}}$ and re-route all messages from i to j through that chain. Define an $n \times n$ matrix M_1 responsible for setting to 0 the entry of $\widehat{P}$ corresponding to sending information from i to j without delay. Instead, i sends its message to the first delay node d_1. A message is delivered to j by delay node $d_{b_{ij}}$. Node j has to assign a weight to this message equal to the weight that would be used to receive from i directly without delay, i.e., P_{ji}. This is achieved by an $n \times b_{ij}$ matrix M_2. A $b_{ij} \times n$ matrix M_3 delivers the message from i to the first delay node in the chain d_1 and we use a $b_{ij} \times b_{ij}$ matrix M_4 to pass messages along the delay chain. We use e_i to denote the i-th column of the $n \times n$ identity matrix and l_i to denote the i-th column of the $b_{ij} \times b_{ij}$ identity matrix. With these definitions, we initialize $\widehat{P} = P$. To add b_{ij}

delay nodes on the edge $i \to j$ we apply the transformation

$$\widehat{P} \leftarrow \begin{bmatrix} \widehat{P} + M_1 & M_2 \\ M_3 & M_4 \end{bmatrix} \tag{6.5}$$

$$M_1 = -e_j e_j^T \widehat{P} e_i e_i^T \tag{6.6}$$

$$M_2 = -M_1 e_i l_{b_{ij}}^T \tag{6.7}$$

$$M_3 = l_i e_j^T \tag{6.8}$$

$$M_4 = \sum_{k=1}^{b_{ij}-1} l_{k+1} l_k^T. \tag{6.9}$$

Using $\widehat{P}$ we can analyze the effect of delays on convergence based on the update equations for row stochastic consensus

$$\widehat{\boldsymbol{z}}(t) = \widehat{P}\widehat{\boldsymbol{z}}(t-1), \tag{6.10}$$

where $\widehat{\boldsymbol{z}}(t)$ is the augmented state vector of dimension $n + b$ containing values for the compute nodes and virtual delay nodes.

6.2.2 Stationary Distribution under Fixed Delays

We have already seen that without delays a doubly stochastic matrix P has a uniform stationary distribution. In the presence of fixed delays this is no longer true since $\widehat{P}$ is only row stochastic. However, as it turns out, if the original matrix P is doubly stochastic the stationary distribution $\widehat{\boldsymbol{\pi}}$ of $\widehat{P}$ can be computed analytically. Let us index the directed edges of G (without delays) by $r = 1, 2, \ldots, |E|$. For any directed edge r from node i to node j, by $i(r)$ we indicate the origin of the edge and $j(r)$ the destination of the edge. The weight used by the receiving node j is $P_{j(r)i(r)}$. Moreover, let b_r denote the amount of delay on edge r. If a random walk leaves node i through edge r, it will have to traverse all delay nodes that replace edge r so intuitively the value of the stationary distribution vector for all delay nodes on the delay chain replacing edge r will be the same. With a slight abuse of notation we denote that value $\widehat{\pi}_r$. For the same reason, since doubly stochastic matrices are averaging matrices, we expect all of the original compute nodes to have the same

value in the stationary distribution say $\widehat{\pi}_V$. Based on those observations we wish to find the stationary distribution of $\widehat{P}$ which has structure

$$\widehat{\boldsymbol{\pi}} = [\widehat{\pi}_V \mathbf{1}_n^T \;\; \widehat{\pi}_1 \mathbf{1}_{b_1}^T \;\; \cdots \;\; \widehat{\pi}_{|E|} \mathbf{1}_{b_{|E|}}^T]^T. \tag{6.11}$$

Assume the delay matrix $\widehat{P}$ is built as explained before and focus on the balance equations $\widehat{\boldsymbol{\pi}}^T \widehat{P} = \widehat{\boldsymbol{\pi}}^T$. For each edge r of G there is one column in $\widehat{P}$ whose only non-zero value is $P_{j(r)i(r)}$; the weight that node j assigns to incoming messages from node i over edge r. This column is indexed by $d_{b_{ij}}^{i\to j}$. See for example column $d_2^{1\to 2}$ in (6.4). From all such columns we obtain equations of the form

$$\pi_V P_{j(r)i(r)} = \pi_r. \tag{6.12}$$

Moreover, the elements of the stationary distribution $\boldsymbol{\pi}$ must sum to 1; i.e.,

$$\mathbf{1}^T \cdot \boldsymbol{\pi} = 1 \quad \Rightarrow \quad n\pi_V + \sum_{r=1}^{m} b_r \pi_r = 1. \tag{6.13}$$

Substituting π_r from (6.12) to (6.13), we first compute π_V and then go back to (6.12) to get the stationary distribution values:

$$\pi_V = \frac{1}{n + \sum_{r=1}^{|E|} b_r P_{j(r)i(r)}} \tag{6.14}$$

$$\pi_r = \frac{p_{i(r)j(r)}}{n + \sum_{r=1}^{|E|} b_r P_{j(r)i(r)}}. \tag{6.15}$$

One can easily verify that the rest of the equations in (6.13) are satisfied with the computed $\boldsymbol{\pi}$. The stationary distribution depends both on the weight we use to send messages through every edge r and also on the amount of delay b_r.

An intriguing corollary is that in the special case where P is the *max-weight* doubly stochastic matrix[2] , the entries of $\widehat{\boldsymbol{\pi}}$ only take one of two values, one for the compute nodes in the set V and one for the delay nodes; i.e., it does not matter

[2] For an undirected graph G without self loops, with adjacency matrix A and node degrees $\boldsymbol{v} = [deg_1, \ldots, deg_n]$ the max-weight matrix is defined as $P = I - \frac{diag(\boldsymbol{v}) - A}{\max_i deg_i + 1}$ and is doubly stochastic.

how the delays are distributed over the links. Specifically, denoting by D the set of delay nodes we have

$$\widehat{\pi}_V = \frac{d_{max}+1}{b+n(d_{max}+1)}, \quad \widehat{\pi}_D = \frac{1}{b+n(d_{max}+1)} \tag{6.16}$$

where d_{max} is the maximum degree of G viewed as undirected ignoring self-loops.

Notice that even when P is doubly stochastic (and thus is an averaging matrix), the row stochastic delayed matrix $\widehat{P}$ does not converge to the average in general, since its stationary distribution is not uniform. To converge to the average as required for distributed optimization we have two options. If we must use matrix P with delays, we can rescale the initial values by the stationary distribution of $\widehat{P}$ as described in Section 5.2.1. Alternatively, we can use Push-Sum as the consensus algorithm. Push-Sum has already been presented in Section 5.3. Later in this chapter we will show that Push-Sum is an averaging algorithm that is immune to communication delays if no messages get lost.

6.2.3 Convergence Rate under Fixed Delays

A question of great interest is how quickly does a consensus algorithm converge in the presence of delays. We have already seen that the rate of convergence of consensus and distributed optimization algorithms such as DDA and PS-DDA depends on the second largest eigenvalue of the consensus matrix (see e.g., (5.15) in Section 5.3). Obtaining eigenvalue bounds becomes more challenging when the consensus matrix is not symmetric nor doubly stochastic. The analysis is much easier when the consensus matrix has non-zero return probabilities; i.e., non-zero entries in the diagonal. However, by construction, the delay nodes only relay information and have no self loops. Thus, the diagonal entries in $\widehat{P}$ corresponding to delay nodes are zero. This makes $\widehat{P}$ a non-reversible Markov chain that is not strongly aperiodic[3] , and the majority of the known convergence rate results for Markov chains do not

[3] A Markov chain is strongly aperiodic if all the diagonal entries of its transition matrix are at least 1/2.

apply. To get a bound on the convergence rate under fixed delays, we apply the result from [34] with the lazy version $\widehat{P}_{lazy} = \frac{1}{2}(I + \widehat{P})$ of $\widehat{P}$. First, the *additive reversibilization* of a Markov chain with transition matrix P is defined as

$$U(P) = \frac{P + \tilde{P}}{2}, \tag{6.17}$$

where $\tilde{P}$ is the *time-reversed* chain,

$$\tilde{P}_{ij} = \frac{\pi_j P_{ji}}{\pi_i}, \tag{6.18}$$

when P has a stationary distribution $\boldsymbol{\pi}$. Since $\widehat{P}_{lazy}$ is strongly aperiodic and non-reversible, Corollary 2.9 in [34] states that

$$\left\| [\widehat{P}^t]_{i,:} - \widehat{\boldsymbol{\pi}} \right\|_{TV}^2 \leq \left\| [\widehat{P}_{lazy}^t]_{i,:} - \widehat{\boldsymbol{\pi}}_{lazy} \right\|_{TV}^2 \leq \frac{(\lambda_2(U(\widehat{P}_{lazy})))^t}{4[\widehat{\pi}_{lazy}]_i} \tag{6.19}$$

with $\widehat{\boldsymbol{\pi}}_{lazy} = \widehat{\boldsymbol{\pi}}$. In the last equation $\left\| P_{i,:} - \boldsymbol{\pi}^T \right\|_{TV} = \frac{1}{2} \sum_{j=1}^{n} |P_{ij} - \pi_j|$ is the total variation distance of any row of stochastic matrix P and its stationary distribution.

At this point we are ready to ask three questions:

1. How do the delays affect the convergence rate of average consensus algorithms? One way to answer is to understand how much larger is $\lambda_2(U(\widehat{P}_{lazy}))$ in comparison to $\lambda_2(P)$ since a larger eigenvalue implies slower convergence based on bound (6.19).
2. How is the convergence of distributed optimization algorithms affected by the delays? Using the results developed so far, we present an analysis for DDA to characterize the effect of delays.
3. Finally one may ask a design question. Given a network G and the fixed delays on its edges, what is the optimal set of weights? In other words, what is the best possible matrix P that achieves the smallest possible second eigenvalue $\lambda_2(\widehat{P})$? We do not have an analytic solution. However, this question can be formalized as a non-convex minimization problem for which experimental evidence in [89] illustrates that even a numerically computed local minimum can offer substantially faster convergence to consensus.

We answer the first two questions below and leave the third as a direction for future work.

6.2.4 Effect of Delays on Second Eigenvalue

The rate at which the powers of a consensus matrix P converge to its limit $\mathbf{1}\boldsymbol{\pi}^T$, as measured by the total variation distance, can be bounded by $\lambda_2(P)$, the second largest eigenvalue of P. The second largest eigenvalue, in turn, can be bounded using a geometric argument based on the *Poincaré inequality* [29,34]. The intuition is to look for the bottleneck edge which limits the flow of information and consequently the convergence speed. Let P be the transition matrix of a discrete-time finite-state Markov chain with state space Ω, and suppose that P has stationary distribution $\boldsymbol{\pi}$. For a pair of states $x, y \in \Omega$, a path γ_{xy} from x to y is a sequence of states $w_0 = x, w_1, \ldots, w_l = y$ such that $P_{w_j, w_{j+1}} > 0$ for all $j = 1, \ldots, l$. The Poincaré inequality requires that we define a canonical path γ_{xy} in P for every pair of states $x, y \in \Omega$. Let also Γ be the set of all selected canonical paths γ_{xy}. To identify bottlenecks we consider how many paths γ_{xy} go through each edge $e \in \{(v, w) \in \Omega^2 \colon P_{vw} > 0\}$. A measure of bottlenecks in P is given by the *Poincaré constant*,

$$K = \max_{e=(v,w)} \left[\frac{1}{\pi_v P_{vw}} \sum_{x,y:\ e \in \gamma_{xy}} |\gamma_{xy}| \, \pi_x \pi_y \right], \tag{6.20}$$

where $|\gamma_{xy}|$ is the length (number of edges) of the path γ_{xy}. The constant K quantifies the load on the most heavily used edge. Paths are assigned a weight $\pi_x \pi_y$ based on the value of the stationary distribution value at the endpoints. Note that the value of K is affected by the choice of the paths $\{\gamma_{xy}\}$ used. The Poincaré constant gives a bound on the second eigenvalue of P:

$$\lambda_2 \leq 1 - \frac{1}{K}. \tag{6.21}$$

Our goal is to use a given set of canonical paths Γ for G to construct a set of canonical paths in $\widehat{G}$, the augmentation of G after adding fixed edge delays. This will reveal how the delays affect the convergence rate of the delayed consensus

algorithms. To that end, we compute the Poincaré constant for $\widehat{G}$ as a function of the Poincaré constant of the original graph G.

Since $\widehat{P}$ represents a non-reversible Markov Chain on the state space $\Omega = \widehat{V}$, we consider the lazy additive reversibilization $U(\widehat{P}_{lazy})$ which is strongly aperiodic, reversible, has the same stationary distribution as $\widehat{P}$, and whose convergence rate bounds that of $\widehat{P}$. With the exception of some added self loops on the delay nodes, the graph structure compatible with $U(\widehat{P}_{lazy})$ is the same as that of $\widehat{P}$. We assume that the delay on any edge is bounded by $b_{ij} \leq B$ for some $B > 0$, and we use subscripts to index the nodes on a delay chain. To compute the Poincaré constant $\widehat{K}$ for $\widehat{G}$ we start by listing observations and consequences of the procedure described above for augmenting G with fixed delays.

1. We claim that if $e = (v, w)$ is the bottleneck edge in G with no delays, then all edges on the delay chain $v \to d_1 \to \cdots \to d_{B'} \to w, B' \leq B$, that replaces e in $\widehat{G}$ are bottlenecks in $\widehat{G}$. The reason is that if a flow needs to go through e in G, it will have to go through all of the delay edges replacing e in $\widehat{G}$. This is true because the degrees of the compute nodes do not change by adding fixed delays in the augmentation procedure described above; the paths between the compute nodes in $\widehat{G}$ are elongated without offering new path alternatives. Consequently, to compute the Poincaré constant of $U(\widehat{P}_{lazy})$ we do not need to maximize over all edges in $\widehat{G}$. Instead we only examine edges in the middle of delay chains. If a delay chain connecting compute nodes a and b has length $B' \leq B$, we focus on the edge $\widehat{e} = (d^{ab}_{\lfloor \frac{B'}{2} \rfloor}, d^{ab}_{\lfloor \frac{B'}{2} \rfloor + 1})$.

2. We intend to use the given collection of canonical paths Γ in G to derive a bound on the Poincaré constant of $\widehat{G}$. A Markov chain on $\widehat{G}$ has $|\widehat{V}| = n + b$ states, but the paths Γ in G are only defined for the n states in V. However, to apply the Poincaré inequality for $\widehat{P}$, we need to define a set of paths between all pairs of states in $\widehat{V}$. We can design a collection of paths in $\widehat{G}$ by associating each delay state in $\widehat{V}$ with a compute node in $V \subset \widehat{V}$. The key point is to ensure that if a path γ_{xy} goes through an edge e of G, then in $\widehat{G}$ we will have a set of paths $\{\widehat{\gamma}_{xy}\}$ corresponding to all states identified with the compute nodes x, y, and all of the paths in $\{\widehat{\gamma}_{xy}\}$

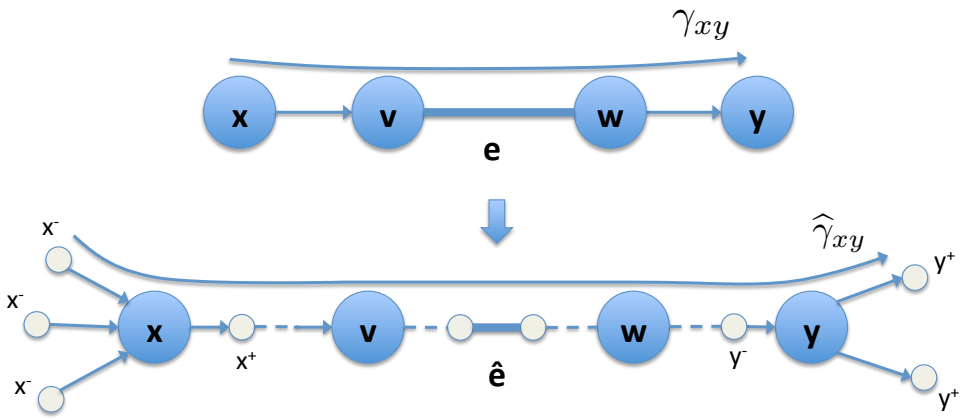

Figure 6–2: (Top) A path γ_{xy} in G. (Bottom) After adding delays in $\widehat{G}$, all paths from nodes $\{x^-, x, x^+\}$ towards nodes $\{y^-, y, y^+\}$ are associated with the same path γ_{xy}. If $e = (v, w)$ was a bottleneck edge in G, edge $\widehat{e}$ in the middle of the delay chain that replaced e will be a bottleneck edge in $\widehat{G}$.

go through $\widehat{e}$, the edge in the middle of the delay chain that replaced e in $\widehat{G}$. By forming this path association, the expression for K will appear in the bound for $\widehat{K}$. Let us use the notation x^- to denote delay nodes before x associated with paths through x, and x^+ to denote delay nodes after x. Figure 6–2 illustrates the path association.

We distinguish the following nine cases. If x, y are compute nodes in $\widehat{G}$, we associate $\widehat{\gamma}_{xy} \sim \gamma_{xy}$. Note that $|\widehat{\gamma}_{xy}| \leq (B+1)\,|\gamma_{xy}|$ when the maximum possible delay per edge is B. Next, to consider paths to or from delay nodes, we associate a delay node with the compute node that is closest to it in the direction of the path. For each path γ_{xy} of G going through edge e, we identify different cases of paths in $\widehat{G}$ going through $\widehat{e}$. We have eight possibilities: $x \to y^-, x \to y^+, x^- \to y^-, x^- \to y, x^- \to y^+, x^+ \to y^-, x^+ \to y$, and $x^+ \to y^+$.

3. To get a cleaner expression for the bound, assume that P is doubly stochastic. In that case, from (6.14) we see that the stationary distribution of the compute nodes in the presence of delays is $\widehat{\pi}_x = \frac{\pi_x}{c}$ where $c = \frac{n+\sum_r b_r P_{r(i)r(j)}}{n}$. Moreover, for all compute nodes x, we have $\widehat{\pi}_x \geq p\widehat{\pi}_{x^-}$ and $\widehat{\pi}_x \geq p\widehat{\pi}_{x^+}$ where $p = \max_{i \neq j} P_{ij}$.

With the above considerations in mind, we start from the definition of the Poincaré constant for $\widehat{G}$:

$$\widehat{K} = \max_{h=(a,b)} \Big[\frac{1}{\widehat{\pi}_a U(a,b)} \sum_{x,y:\ h \in \widehat{\gamma}_{xy}} |\widehat{\gamma}_{xy}|\, \widehat{\pi}_x \widehat{\pi}_y \Big]. \tag{6.22}$$

Let $e = (v, w)$ be a bottleneck edge of G. This means that the edge $\widehat{e}$ in the middle of the delay chain that replaces e will be the bottleneck in $\widehat{G}$. After some algebra we can bound $\widehat{K}$ with an expression that involves K (from (6.20)). Besides the leading constant involving the bottleneck edge, we need to break the sum over the canonical paths into summands according to the nine cases we described in consideration 2 above. We state here the final result and the proof can be found in Appendix D.

Theorem 6.1. *Let G be a network endowed with a doubly stochastic consensus matrix P and a set of canonical paths Γ yielding a Poincaré constant K, and let $e = (v, w)$ denote the corresponding bottleneck edge. Let $\{b_{ij}\}_{(i,j)\in E}$ denote the network delays and assume $b_{ij} \le B$ for all i, j. Then the corresponding augmented weight matrix $\widehat{P}$ defined on $\widehat{G}$ has a Poincaré constant $\widehat{K}$ for which*

$$\begin{aligned} \widehat{K} \le ZK, \quad Z = &\frac{P_{vw}}{4c} \Big[p^2(2d_{max}^2 + 3d_{max} + 1)B^3 \\ &+ p(2pd_{max}^2 + 2pd_{max} + 8d_{max} + 6)B^2 \\ &+ (8pd_{max} + p + 8)B + 8 \Big], \end{aligned} \tag{6.23}$$

where $p = \max_{i \neq j} P_{ij}$, $c = \frac{n + \sum_r b_r P_{r(i)r(j)}}{n}$ and d_{max} is the maximum degree in the undirected graph G ignoring self-loops.

Theorem 6.1 yields a bound in the second eigenvalue and thus the spectral gap of $\widehat{P}$.

Corollary 6.1. *Suppose a doubly stochastic matrix P on a graph G has a spectral gap $1 - \lambda_2(P) \ge \frac{1}{K}$, and assume that messages over the edges of G experience arbitrary fixed delays of up to B iterations. Then the spectral gap of $\widehat{P}$ is reduced by at most a factor $\Theta(B^2)$; i.e.,*

$$1 - \lambda_2(\widehat{P}) \ge \frac{1}{ZK}, \quad Z = \Theta(B^2). \tag{6.24}$$

Proof. From Theorem 6.1 we have $\lambda_2(\widehat{P}) \leq \lambda_2(U) \leq 1 - \frac{1}{ZK}$. Since $b_r \leq B, r = 1, 2, \ldots, m$ we see that $c = \frac{n+\sum_r b_r p_{r(i)r(j)}}{n} = \Theta(B)$ and thus $Z = \Theta(B^2)$. □

To the best of our knowledge this is the first result to describe the effect of a bounded fixed delay on the convergence rate of average consensus. It shows that the delays cannot slow down consensus by more than a polynomial factor and convergence remains exponentially fast.

Experimental Validation of Theorem 6.1

We conclude the discussion about the effect of delays on the convergence rate of consensus algorithms by verifying Theorem 6.1 and Corollary 6.1 in simulation. One difficulty with validating these results numerically is that Theorem 6.1 describes the effect of fixed delays relative to a consensus matrix P on a graph G without delays. To compute the Poincaré constant $\widehat{K}$ explicitly we still need to find a set of canonical paths in G and apply (6.20) which can be computationally intractable. Instead, we estimate $\widehat{K}$ as follows. For a given network of 15 nodes, matrix P, and delay bound B, we randomly select delays for all edges, construct $U(\widehat{P}_{lazy})$ as explained in Section 6.2.3 and compute the second eigenvalue of U. For each bound B we repeat this procedure 50 times. Since $\widehat{K} \geq \frac{1}{1-\lambda_2(U(\widehat{P}_{lazy}))}$ we keep the largest λ_2 out of the 50 trials to approximately maximize the lower bound on $\widehat{K}$. Figure 6–3 illustrates that the inverse spectral gap increases almost quadratically with B. It appears that $O(B^2)$ might be increasing faster than $\widehat{K}$ so our bound might be loose but not dramatically so. The mismatch could also be a result of poor approximation on $\widehat{K}$ since for larger B, 50 trials might not be enough to capture the worst possible scenario.

6.2.5 Distributed Optimization under Fixed Communication Delays

The previous section studied how communication delays effect the convergence rate of consensus algorithms. However, as one might expect, exchanging delayed information has an even more severe effect on distributed optimization algorithms. We illustrate this effect in this section by studying synchronous DDA under fixed delays. By following similar steps, this analysis can be extended to other algorithms.

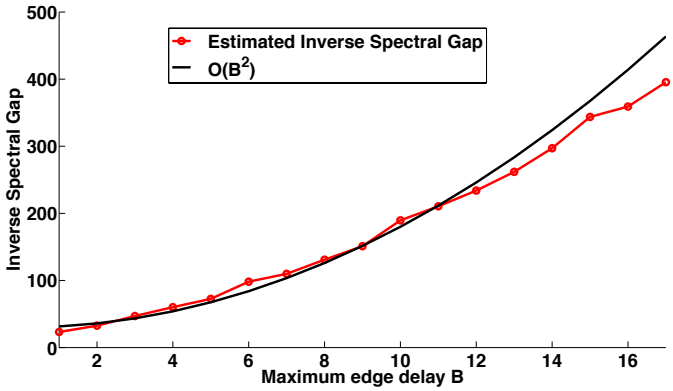

Figure 6–3: (Red) Estimated inverse spectral gap $\frac{1}{1-\lambda_2(U(\widehat{P}_{lazy}))}$ for a network G of 15 nodes when increasing the upper bound B of fixed delays. Each data point is the maximum over 50 randomly selected delay distributions over the edges of G. (Black) An approximate fit of an $O(B^2)$ curve to show that the inverse spectral gap does not deteriorate by worse than a quadratic factor as we increase B.

Let us introduce b delay nodes in the network G. We associate with each delay node a function $f_i(\boldsymbol{w}) = 0, i = n+1, \ldots, n+b$ so that the subgradients on the delay nodes are zero as well. To analyze distributed dual averaging with delays, we use $\widehat{P}$ as a transition matrix instead of P in equation (2.22).

$$\boldsymbol{z_i}(t+1) = \sum_{j=1}^{n+b} \widehat{P}_{ij} \boldsymbol{z_j}(t) + \boldsymbol{g_i}(t), \quad i = 1, \ldots, n+b. \tag{6.25}$$

The matrix $\widehat{P}$ is not doubly stochastic and has a non-uniform stationary distribution $\widehat{\boldsymbol{\pi}}$ given in (6.14). As discussed in Section 5.2.1, the result is an undesired bias since we end up optimizing the wrong objective function $\tilde{F}(\boldsymbol{w}) = \sum_{i=1}^{n+b} \widehat{\pi}_i f_i(\boldsymbol{w})$. To correct the bias, if we know the network size n and $\widehat{\boldsymbol{\pi}}$, we rewrite our objective function as

$$F(\boldsymbol{w}) = \frac{1}{n} \sum_{i=1}^{n+b} f_i(\boldsymbol{w}) = \sum_{i=1}^{n+b} \widehat{\pi}_i \left[\frac{f_i(\boldsymbol{w})}{\widehat{\pi}_i n} \right] = \sum_{i=1}^{n+b} \widehat{\pi}_i h_i(\boldsymbol{w}). \tag{6.26}$$

With this definition, in (6.25) and in the sequel we use subgradients $\boldsymbol{g_i}(t) \in \partial h_i(\boldsymbol{w_i}(t))$ for which the Lipschitz constant is $L_h = \max_i \frac{L}{\pi_i n}$.

First let us adapt the auxiliary sequences from (2.26):

$$\overline{\boldsymbol{z}}(t) = \sum_{i=1}^{n+b} \widehat{\pi}_i \boldsymbol{z_i}(t) \tag{6.27}$$

$$\boldsymbol{y}(t) = \Pi_{\mathcal{W}}^{\psi}(\overline{\boldsymbol{z}}(t), a(t)). \tag{6.28}$$

The weighted average cumulative gradient $\overline{\boldsymbol{z}}$ evolves as follows:

$$\begin{aligned}
\overline{\boldsymbol{z}}(t+1) &= \sum_{i=1}^{n+b} \widehat{\pi}_i \boldsymbol{z_i}(t+1) \\
&= \sum_{i=1}^{n+b} \widehat{\pi}_i \left(\sum_{j=1}^{n+b} \widehat{P}_{ij} \boldsymbol{z_j}(t) + \boldsymbol{g_i}(t) \right) \\
&= \sum_{j=1}^{n+b} \boldsymbol{z_j}(t) \left(\sum_{i=1}^{n+b} \widehat{\pi}_i \widehat{P}_{ij} \right) + \sum_{i=1}^{n+b} \pi_i \boldsymbol{g_i}(t).
\end{aligned} \tag{6.29}$$

Since $\widehat{\boldsymbol{\pi}}$ is a left eigenvector of $\widehat{P}$, we know that $\widehat{\boldsymbol{\pi}}^T \widehat{P} = \widehat{\boldsymbol{\pi}}^T$ which implies $\sum_{i=1}^{n+b} \widehat{\pi}_i \widehat{P}_{ij} = \widehat{\boldsymbol{\pi}}^T \widehat{P}_{:,j} = \widehat{\pi}_j$. Using this fact,

$$\overline{\boldsymbol{z}}(t+1) = \sum_{j=1}^{n+b} \boldsymbol{z_j}(t) \widehat{\pi}_j + \sum_{i=1}^{n+b} \widehat{\pi}_i \boldsymbol{g_i}(t), \tag{6.30}$$

and finally

$$\overline{\boldsymbol{z}}(t+1) = \overline{\boldsymbol{z}}(t) + \sum_{i=1}^{n+b} \widehat{\pi}_i \boldsymbol{g_i}(t). \tag{6.31}$$

Using the last recursion, with $\boldsymbol{z_i}(0) = 0$, we rewrite (6.27) as

$$\overline{\boldsymbol{z}}(t) = \sum_{s=1}^{t-1} \sum_{i=1}^{n+b} \widehat{\pi}_i \boldsymbol{g_i}(s), \; \boldsymbol{y}(t) = \Pi_{\mathcal{W}}^{\psi} \left[\sum_{s=1}^{t-1} \sum_{i=1}^{n+b} \widehat{\pi}_i \boldsymbol{g_i}(s), a(t) \right]. \tag{6.32}$$

At this point we have all we need to proceed with the convergence proof. Since $F(\boldsymbol{w})$ is convex, for any $\boldsymbol{w^*} \in \mathcal{W}$ we have

$$F(\overline{\boldsymbol{w}}_{\boldsymbol{i}}(T)) - F(\boldsymbol{w^*}) \leq \frac{1}{T} \sum_{t=1}^{T} \big[F(\boldsymbol{w_i}(t)) - F(\boldsymbol{w^*}) \big]. \tag{6.33}$$

Using Lemma 2.1 we obtain

$$F(\overline{\boldsymbol{w}}_{\boldsymbol{i}}(T)) - F(\boldsymbol{w}^*) \leq \frac{1}{T}\sum_{t=1}^{T}\left[F(\boldsymbol{y}(t)) - F(\boldsymbol{w}^*)\right] + \frac{L_h}{T}\sum_{t=1}^{T} a(t)\left\|\overline{\boldsymbol{z}}(t) - \boldsymbol{z}_{\boldsymbol{i}}(t)\right\|. \tag{6.34}$$

To bound the first term in (6.34), we add and subtract $\sum_{t=1}^{T}\sum_{i=1}^{n+b}\widehat{\pi}_i h_i(\boldsymbol{w}_{\boldsymbol{i}}(t))$ and since $\sum_{i=1}^{n+b}\pi_i h_i(\boldsymbol{w}^*) = F(\boldsymbol{w}^*)$ we get

$$\begin{aligned}\sum_{t=1}^{T}\left[F(\boldsymbol{y}(t)) - F(\boldsymbol{w}^*)\right] \leq & \sum_{t=1}^{T}\sum_{i=1}^{n+b}\widehat{\pi}_i\left[h_i(\boldsymbol{w}_{\boldsymbol{i}}(t)) - h_i(\boldsymbol{w}^*)\right] \\ & + \sum_{t=1}^{T}\sum_{i=1}^{n+b}\widehat{\pi}_i\left|h_i(\boldsymbol{y}(t)) - h_i(\boldsymbol{w}_{\boldsymbol{i}}(t))\right|. \end{aligned} \tag{6.35}$$

Using convexity of each component $h_i(\boldsymbol{w})$ with $\boldsymbol{g}_{\boldsymbol{i}}(t) \in \partial h_i(\boldsymbol{w}_{\boldsymbol{i}}(t))$,

$$\begin{aligned}\sum_{t=1}^{T} F(\boldsymbol{y}(t)) - F(\boldsymbol{w}^*) \leq & \sum_{t=1}^{T}\sum_{i=1}^{n+b}\widehat{\pi}_i\langle \boldsymbol{g}_{\boldsymbol{i}}(t), \boldsymbol{y}(t) - \boldsymbol{w}^*\rangle \\ & + \sum_{t=1}^{T}\sum_{i=1}^{n+b}\widehat{\pi}_i\langle \boldsymbol{g}_{\boldsymbol{i}}(t), \boldsymbol{w}_{\boldsymbol{i}}(t) - \boldsymbol{y}(t)\rangle \\ & + \sum_{t=1}^{T}\sum_{i=1}^{n+b}\widehat{\pi}_i\left|h_i(\boldsymbol{y}(t)) - h_i(\boldsymbol{w}_{\boldsymbol{i}}(t))\right|. \end{aligned} \tag{6.36}$$

Focusing on the first term of (6.36) and recalling the definition (6.32) of $\boldsymbol{y}(t)$ we have

$$\begin{aligned}\sum_{t=1}^{T}\sum_{i=1}^{n+b}\widehat{\pi}_i\langle \boldsymbol{g}_{\boldsymbol{i}}(t), \boldsymbol{y}(t) - \boldsymbol{w}^*\rangle &= \sum_{t=1}^{T}\Big\langle \sum_{i=1}^{n+b}\widehat{\pi}_i \boldsymbol{g}_{\boldsymbol{i}}(t), \boldsymbol{y}(t) - \boldsymbol{w}^*)\Big\rangle \\ &= \sum_{t=1}^{T}\Big\langle \sum_{i=1}^{n+b}\widehat{\pi}_i \boldsymbol{g}_{\boldsymbol{i}}(t), \Pi_{\mathcal{W}}^{\psi}\left[\sum_{s=1}^{t-1}\sum_{i=1}^{n+b}\widehat{\pi}_i \boldsymbol{g}_{\boldsymbol{i}}(s), a(t)\right] - \boldsymbol{w}^*\Big\rangle. \end{aligned} \tag{6.37}$$

With $\sum_{i=1}^{n+b}\widehat{\pi}_i \boldsymbol{g}_{\boldsymbol{i}}(s)$ playing the role of the arbitrary vector sequence, the last equation can be bounded using Lemma 2.2 after applying the Cauchy-Schwartz inequality

and remembering that $\|\boldsymbol{g_i}\|_* \leq L_h$:

$$\begin{aligned}\sum_{t=1}^{T}\sum_{i=1}^{n+b}\widehat{\pi}_i\langle \boldsymbol{g_i}(t), \boldsymbol{y}(t)-\boldsymbol{w}^*\rangle \leq & \frac{1}{2}\sum_{t=1}^{T}a(t-1)\Big\|\sum_{i=1}^{n+b}\widehat{\pi}_i\boldsymbol{g_i}(t)\Big\|_*^2+\frac{1}{a(T)}\psi(\boldsymbol{w}^*)\\ \leq & \frac{L_h^2}{2}\sum_{t=1}^{T}a(t-1)+\frac{1}{a(T)}\psi(\boldsymbol{w}^*). \end{aligned} \tag{6.38}$$

For the last two terms in (6.36) we use *L-Lipshitz* continuity of $F(\boldsymbol{w})$ and Lemma 2.3 to get after some straightforward algebra that

$$\begin{aligned}&\sum_{t=1}^{T}\sum_{i=1}^{n+b}\widehat{\pi}_i\Big(\big\langle \boldsymbol{g_i}(t), \boldsymbol{w_i}(t)-y(t)\big\rangle+\big|h_i(\boldsymbol{y}(t))-h_i(\boldsymbol{w_i}(t))\big|\Big)\\ &\leq\sum_{t=1}^{T}\sum_{i=1}^{n+b}\widehat{\pi}_i\Big(L_h\left\|\boldsymbol{y}(t)-\boldsymbol{w_i}(t)\right\|+L_h\left\|\boldsymbol{y}(t)-\boldsymbol{w_i}(t)\right\|\Big)\\ &\leq 2L_h\sum_{t=1}^{T}\sum_{i=1}^{n+b}\widehat{\pi}_i\Big\|\Pi_{\mathcal{W}}^{\psi}\left[\overline{\boldsymbol{z}}(t),a(t)\right]-\Pi_{\mathcal{W}}^{\psi}\left[\boldsymbol{z_i}(t),a(t)\right]\Big\|\\ &\leq 2L_h\sum_{t=1}^{T}\sum_{i=1}^{n+b}\widehat{\pi}_i a(t)\left\|\overline{\boldsymbol{z}}(t)-\boldsymbol{z_i}(t)\right\|. \end{aligned} \tag{6.39}$$

Going back to (6.34), we replace the bounds we derived for the first and last two terms to obtain a generalized version of the bound in Theorem 2.1 for the modified version of the algorithm:

$$\begin{aligned}F(\overline{\boldsymbol{w}}_{\boldsymbol{i}}(T))-F(\boldsymbol{w}^*)\leq & \frac{L_h^2}{2T}\sum_{t=1}^{T}a(t-1)+\frac{1}{Ta(T)}\psi(\boldsymbol{w}^*)\\ &+\frac{2L_h}{T}\sum_{t=1}^{T}\sum_{i=1}^{n+b}\widehat{\pi}_i a(t)\left\|\overline{\boldsymbol{z}}(t)-\boldsymbol{z_i}(t)\right\|\\ &+\frac{L_h}{T}\sum_{t=1}^{T}a(t-1)\left\|\overline{\boldsymbol{z}}(t)-\boldsymbol{z_i}(t)\right\|. \end{aligned} \tag{6.40}$$

Next we need to bound the network error $\|\overline{\boldsymbol{z}}(t)-\boldsymbol{z_i}(t)\|_*$. If we define for convenience $\Phi(t,s)=\widehat{P}^{t-s+1}$ and back-substitute in the recursion (6.25) we can see that (see also (5.27))

$$\boldsymbol{z_i}(t)=\sum_{s=1}^{t-1}\sum_{j=1}^{n+b}[\Phi(t-1,s)]_{ij}\cdot\boldsymbol{g_j}(s-1)+\boldsymbol{g_i}(t-1). \tag{6.41}$$

Recalling the definition (6.27) for $\overline{\boldsymbol{z}}(t)$, after rearranging some terms and using the fact that $\sum_{k=1}^{n+b} \widehat{\pi}_k [\Phi(t-1,s)]_{kj} = \widehat{\pi}_j$, we see that

$$\begin{aligned}
\overline{\boldsymbol{z}}(t) - \boldsymbol{z_i}(t) =& \sum_{k=1}^{n+b} \pi_k \boldsymbol{z_k}(t) - \boldsymbol{z_i}(t) \\
=& \sum_{k=1}^{n+b} \widehat{\pi}_k \Big[\sum_{s=1}^{t-1} \sum_{j=1}^{n+b} \big[\Phi(t-1,s)\big]_{kj} \cdot \boldsymbol{g_j}(s-1) + \boldsymbol{g_k}(t-1) \Big] \\
& - \sum_{s=1}^{t-1} \sum_{j=1}^{n+b} \big[\Phi(t-1,s)\big]_{ij} \cdot \boldsymbol{g_j}(s-1) - \boldsymbol{g_i}(t-1) \\
=& \sum_{s=1}^{t-1} \sum_{j=1}^{n+b} \boldsymbol{g_j}(s-1) \sum_{k=1}^{n+b} \pi_k \big[\Phi(t-1,s)\big]_{jk} + \sum_{k=1}^{n+b} \widehat{\pi}_k \boldsymbol{g_k}(t-1) \\
& - \sum_{s=1}^{t-1} \sum_{j=1}^{n+b} \big[\Phi(t-1,s)\big]_{ji} \cdot \boldsymbol{g_j}(s-1) - \boldsymbol{g_i}(t-1) \\
=& \sum_{s=1}^{t-1} \sum_{j=1}^{n+b} \Big(\widehat{\pi}_j - \big[\Phi(t-1,s)\big]_{ij} \Big) \boldsymbol{g_j}(s-1) \\
& + \sum_{k=1}^{n+b} \widehat{\pi}_k \big[\boldsymbol{g_k}(t-1) - \boldsymbol{g_i}(t-1) \big].
\end{aligned}$$

Taking norms on both sides and using the bound L_h on gradient magnitudes, we obtain

$$\begin{aligned}
\|\overline{\boldsymbol{z}}(t) - \boldsymbol{z_i}(t)\| \leq& \sum_{s=1}^{t-1} \sum_{j=1}^{n+b} \Big| \widehat{\pi}_j - \big[\Phi(t-1,s)\big]_{ij} \Big| \cdot \|\boldsymbol{g_j}(s-1)\| \\
& + \sum_{k=1}^{n+b} \widehat{\pi}_k \, \|\boldsymbol{g_k}(t-1) - \boldsymbol{g_i}(t-1)\| \\
\leq& L_h \sum_{s=1}^{t-1} \Big\| \widehat{\pi}^T - \big[\Phi(t-1,s)\big]_{i,:} \Big\|_1 + 2L_h. \qquad (6.42)
\end{aligned}$$

The last expression reduces to exactly the bound obtained in Theorem 2 in [32] if $\widehat{P}$ is doubly stochastic and there are no delays, since in that case $\widehat{\pi}_i = \frac{1}{n}$ and $L_h = L$.

Instead of using the bounding technique of [32], we provide a different bound that is tighter in the number of iterations. From (6.19) we know that for all i,

$$\left\|\widehat{\boldsymbol{\pi}} - \left[\Phi(t-1,s)\right]_{i,:}\right\|_1 = 2\left\|\widehat{\boldsymbol{\pi}} - \widehat{P}_{i,:}^{t-s+1}\right\|_{TV} \leq \sqrt{\frac{\lambda_2^{t-s+1}}{\pi_i}} \tag{6.43}$$

where $\|\cdot\|_{TV}$ denotes total variation distance and λ_2 is the second largest eigenvalue of the *lazy additive reversibilization* of $\widehat{P}$ (see Section 6.2.3 and [89], [34]). Using this result and applying the formula for a finite geometric series (since $\lambda_2 < 1$), we bound the network error by:

$$\begin{aligned}
\|\overline{\boldsymbol{z}}(t) - \boldsymbol{z_i}(t)\| &\leq L_h \sum_{s=1}^{t-1} \sqrt{\frac{\lambda_2^{t-s+1}}{\widehat{\pi}_i}} + 2L_h \qquad (6.44)\\
&= \frac{L_h}{\sqrt{\widehat{\pi}_i}} \sum_{s=1}^{t-1} \left(\sqrt{\lambda_2}\right)^{t-s+1} + 2L_h \\
&= \frac{L_h}{\sqrt{\widehat{\pi}_i}} \sum_{s=2}^{t} \left(\sqrt{\lambda_2}\right)^{s} + 2L_h \\
&= \frac{L_h}{\sqrt{\widehat{\pi}_i}} \frac{\left(\sqrt{\lambda_2}\right)^2 - \left(\sqrt{\lambda_2}\right)^{t+1}}{1-\sqrt{\lambda_2}} + 2L_h \\
&\leq \frac{L_h}{\sqrt{\widehat{\pi}_i}} \frac{\lambda_2}{1-\sqrt{\lambda_2}} + 2L_h \triangleq K_i. \qquad (6.45)
\end{aligned}$$

This bound is tighter than the one obtained in [32] since for fixed n it is constant and does not increase logarithmically with time. This also changes the dependence on the network size, through $\widehat{\pi}_i$ and λ_2. Since the network error is bounded, from (6.40) we can guarantee the converge of DDA to the right solution using any row stochastic matrix and in the presence of fixed delays as long as we choose an appropriate step size sequence. Furthermore, from bound (6.45) we can derive a convergence rate that illustrates the effect of the delay. As a side effect of the analysis, we show that DDA does not need a doubly stochastic matrix to converge to the right solution. Any fixed row stochastic matrix with the appropriate rescaling of the objective components suffices. The latter follows since all of the derivations treated $\widehat{P}$ as row stochastic without explicitly requiring that is has the special structure of a fixed delay matrix.

6.2.6 Convergence of DDA with Fixed Edge Delays

To derive a convergence rate that corresponds to standard DDA but in the presence of fixed edge delays, suppose P is a stochastic matrix whose stationary distribution $\boldsymbol{\pi}$ assigns equal probabilities to all the compute nodes. We can derive a precise expression for the convergence rate by first observing from (6.14) that $\pi_i \geq \frac{1}{n+b}, i \in V$. We use this fact to get

$$\begin{aligned} K_i &\leq \sqrt{n+b}\frac{\lambda_2}{1-\sqrt{\lambda_2}} + 2)L_h \\ &= \underbrace{\left(\sqrt{n+b}\frac{\lambda_2}{1-\sqrt{\lambda_2}} + 2)\right)}_{Q} L_h \triangleq QL_h. \end{aligned} \tag{6.46}$$

By replacing the bound QL_h in (6.40), using the fact that $\sum_{t=1}^{T} t^{-0.5} \leq 2\sqrt{T}-1$ and selecting $a(t) = O(\frac{1}{\sqrt{t}})$, after some algebraic manipulations we prove the following.

Theorem 6.2. *Under the conditions of Theorem 2.1, using update* (6.25) *in place of* (2.22)*, assuming* $\psi(\boldsymbol{w}^*) \leq R^2$*, using the step size sequence* $a(t) = \frac{R}{L_h\sqrt{1+6Q}\sqrt{t}}$*, assuming that* P *is doubly stochastic and in a network with fixed edge delays, for all* $\boldsymbol{w}^* \in \mathcal{W}$

$$F(\overline{\boldsymbol{w}}_{\boldsymbol{i}}(T)) - F(\boldsymbol{w}^*) \leq 2RL_h\sqrt{1+6Q}\frac{1}{\sqrt{T}} \tag{6.47}$$

$$\leq 2RL\frac{(n+b)\sqrt{1+6Q}}{n}\frac{1}{\sqrt{T}}. \tag{6.48}$$

The influence of the network topology is captured by λ_2 in Q and the effect of the network size is $O(n^{\frac{1}{4}})$ since $Q = O(\sqrt{n})$. If b is the total amount of delay cumulatively on all the links, the bound grows like $O(b^{\frac{5}{4}})$. Figure 6–4 illustrates the effect of delays in a toy example. We create a random network topology of 10 nodes. The objective for node i is a simple quadratic: $f_i(\boldsymbol{w}) = (\boldsymbol{w} - i\mathbf{1})^T(\boldsymbol{w} - i\mathbf{1}), \boldsymbol{w} \in \mathbb{R}^5$. For this problem we can easily compute the exact minimizer $\boldsymbol{w}^* = 5.5 \cdot \mathbf{1}$ with $F(\boldsymbol{w}^*) = 412.5$. The blue curve shows the progress of the minimization without delays as the evolution of the maximum error $\max_i |F(\overline{\boldsymbol{w}}_{\boldsymbol{i}}(t)) - F(\boldsymbol{w}^*)|$. The red curve shows that the algorithm is slowed down when we inject a random fixed delay

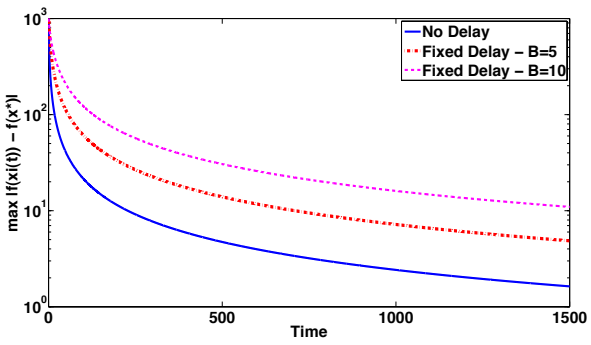

Figure 6–4: Illustration of the effect of fixed edge delays on distributed dual averaging. Blue curve: Performance without delays. Red curve: Performance using a fixed delay up to $B = 5$ time steps per directed link. Purple curve: Performance using a fixed delay up to $B = 10$ time steps per directed link.

of up to $B = 5$ on each directed link (i, j). The purple curve allows for a maximum possible delay $B = 10$.

To verify that the dependence in the total amount of delay appearing in the bound (6.47) is in the right order of magnitude, in Figure 6–5 we record the amount of time it takes to bring the optimization error below a threshold for varying amounts of delay. Specifically, in our problem with 10 nodes, we measure the time until $\max_i |F(\overline{\boldsymbol{w}}_{\boldsymbol{i}}(t)) - F(\boldsymbol{w}^*)| < 180$. We also plot the dominant term in our theoretical bound as $O(\frac{(n+b)^{\frac{5}{4}}}{n}) = 10\frac{(n+b)^{\frac{5}{4}}}{n} + 102$. The bound and simulation are relatively well matched. The discrepancy between the two is explained by the fact that the theory predicts the worse case while the experimental bound illustrates the performance for a specific instantiation of the edge delays.

6.3 Time Varying Communication Delays

To capture the volatility apparent in real networks, it is more appropriate to assume that link delays vary randomly with time. We propose a discrete-time random delay model that starts with any row-stochastic matrix and present a formal convergence proof. Furthermore, we show how using Push-Sum and column stochastic matrices simplifies both the construction and the convergence proof.

6.3.1 Random Delay Model

Similar to the fixed delay model, we add virtual delay nodes. We assume again that delays are finite and upper bounded by a maximum delay B. To model random

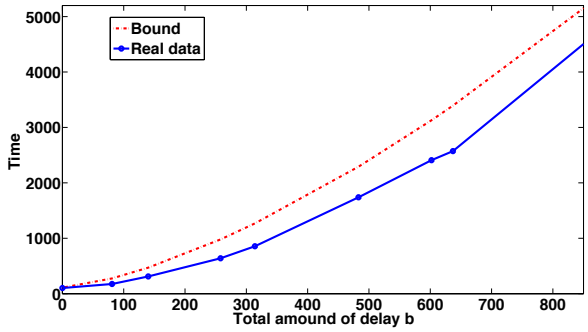

Figure 6–5: Blue curve: Time it takes for a network of 10 nodes to reduce the objective function error $\max_i |F(\overline{\boldsymbol{w}}_{\boldsymbol{i}}(t)) - F(\boldsymbol{w}^*)|$ below 180 as we increase the total amount of delay b in the network. The theoretical bound (red curve) is in the right order of magnitude.

delays in discrete time we need to be careful. Others have previously analyzed a consensus update of the form

$$z_i(t+1) = \sum_{j=1}^{n} P_{ij} z_j(t - b_{ij}(t)), \tag{6.49}$$

where $b_{ij}(t)$ is the random delay experienced by link (i, j) at time t [56,64]. However, this type of update implies that at time t each node i will only receive a single (possibly delayed) message from each neighbour j. In practice this may not be true. We have seen an example in Section 5.2.3 where, due to time varying communication delays, a node does not know how many messages it will receive from a neighbour at each iteration. This scenario can easily occur in practice when messages are large in size and receiving a message takes a non-trivial amount of time during which more messages can arrive. When this happens, the receiving node polling its buffer experiences the arrival of many messages during the same time slot.

To model random bounded delays, we replace each directed edge of the original graph with multiple delay chains of varying lengths to model varying amounts of delay. Every time a message is sent, a random decision is made for which delay

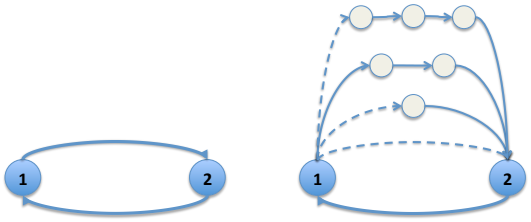

Figure 6–6: Adding a random bounded delay on edge $(1, 2)$. At this particular instant, 1 sends with delay 2 since the connections to delays 1 and 3 are deactivated.

chain the message will take to reach its destination[4] . If a communication network with n computing nodes has m directed edges (not counting the self loops), each edge delivers messages with some bounded delay that is randomly chosen between 0 and B. For example for an edge (i, j) with a maximum delay of 3 we augment (i, j) in G with three parallel delay chains $(d_1^1), (d_1^2, d_2^2), (d_1^3, d_2^3, d_3^3)$ in $\widehat{G}$; see Figure 6–6. We avoid indexing the delay nodes by edge number to not clutter notation. We augment the graph with $\frac{B(B+1)}{2}$ delay nodes per edge or $b = \frac{mB(B+1)}{2}$ delay nodes total, where m is the number of edges in G. We also allow for messages to be delivered without delay, by including the directed edges (i, j) of the original graph G.

Next we seek to write an expression for the matrix $\widehat{P}(t)$ that describes the linear consensus dynamics under random delays. We assume that we are given a row stochastic matrix P for the graph G, and we construct $\widehat{P}(t)$ using the weights suggested by P.

Every time a message is sent, it is routed randomly through one of the B delay chains or the direct edge with zero delay. Outgoing edges to the other chains leading to the same recipient are cut off. Here we consider a time-varying delay model where the delays experienced by messages sent from compute node i to compute node j are i.i.d. and are independent of the delays incurred by messages sent on other links.

[4] Of course in reality this random choice is made by the environment, i.e., the network, and is beyond our control. For modelling purposes to emulate and understand the effect of delays, we can draw a random sample from a distribution that we believe resembles how real network conditions fluctuate.

We associate with each edge in G a discrete probability distribution on the integers $0, \ldots, B$ to govern the delays of messages sent on that edge.

As we see, the augmented graph topology changes at every iteration based on which outgoing edges to delay chains are active. To describe the consensus update equations we need to model the changing topology. At each iteration, a delay is sampled for each message to be transmitted. Based on these delays, at iteration t the graph adjacency matrix $A(t)$ is a sample from the set $\{A^1, \ldots, A^{(B+1)^m}\}$ of possible adjacency matrices. Notice that a delay node could either contain a message or be empty, and a zero message is not the same as the node being empty. To keep track of which delay nodes are empty, we define the sequence of indicator vectors $\{\phi(t)\}_{t=1}^{\infty}, \phi(t) \in \{0,1\}^b$. Using $A(t)$ and $\phi(t)$ we show how to write a transition matrix $\widehat{P}(t)$ at each iteration t.

We begin by observing that the adjacency matrices $A(t)$ have the block structure

$$A(t) = \begin{bmatrix} I_{n\times n} + L(t) & J_{n\times b} \\ R(t) & C_{b\times b} \end{bmatrix}. \tag{6.50}$$

Matrix $A(t)$ should be interpreted as the adjacency matrix of a directed graph. Element $[A(t)]_{ij}$ is 1 if there is a directed link from j to i. Its constituent parts $L(t)$, $J_{n\times b}$, $R(t)$, and $C_{b\times b}$ are described next.

The upper left block is an identity matrix to represent the self-loops, plus a random $n \times n$ square matrix $L(t)$ with zeros on the diagonal and a one at position (i, j) if compute node j sends a message to compute node i with zero delay[5] at iteration t. Matrix $R(t)$ is $b \times n$ and is also a random matrix. When a compute node i transmits to another compute node j at iteration t using the delay chain of length $r \in \{1, \ldots, B\}$, the matrix $R(t)$ encodes that random delay choice in the following manner. For example, if at iteration t node j sends a message to i which is delayed by 2 steps (so that it will arrive at time $t + 3$), $R(t)$ will contain a block for edge

[5] Note that zero delay means that a message sent at iteration t will be delivered at iteration $t + 1$, i.e., without any delay.

(j, i) indicating the delay chain that is active, as illustrated in equation (6.51).

$$R(t) = \begin{array}{c} \\ \vdots \\ d_1^1 \\ d_1^2 \\ d_2^2 \\ d_1^3 \\ d_2^3 \\ d_3^3 \\ \vdots \end{array} \begin{array}{c} \begin{array}{ccccc} 1 & \cdots & j & \cdots & n \end{array} \\ \left[\begin{array}{ccccc} \vdots & & \vdots & & \vdots \\ \hline 0 & \cdots & 0 & \cdots & 0 \\ 0 & \cdots & 1 & \cdots & 0 \\ 0 & \cdots & 0 & \cdots & 0 \\ 0 & \cdots & 0 & \cdots & 0 \\ 0 & \cdots & 0 & \cdots & 0 \\ 0 & \cdots & 0 & \cdots & 0 \\ \hline \vdots & & \vdots & & \vdots \end{array}\right] \end{array}. \tag{6.51}$$

Element (d_1^2, j) of $R(t)$ is 1 since j will transmit to the first delay node in the chain of length 2 towards i. The entries that are not shown within each block are all zero.

Matrix $J_{n\times b}$ describes the connections between the delay nodes d_r^r at the end of each delay chain delivering messages to the compute nodes. The part of $J_{n\times b}$ corresponding to the edge (j, i) of $R(t)$ just discussed has the form

$$J_{n\times b}^T = \begin{array}{c} \\ \vdots \\ d_1^1 \\ d_1^2 \\ d_2^2 \\ d_1^3 \\ d_2^3 \\ d_3^3 \\ \vdots \end{array} \begin{array}{c} \begin{array}{ccccc} 1 & \cdots & j & \cdots & n \end{array} \\ \left[\begin{array}{ccccc} \vdots & & \vdots & & \vdots \\ \hline 0 & \cdots & 1 & \cdots & 0 \\ 0 & \cdots & 0 & \cdots & 0 \\ 0 & \cdots & 1 & \cdots & 0 \\ 0 & \cdots & 0 & \cdots & 0 \\ 0 & \cdots & 0 & \cdots & 0 \\ 0 & \cdots & 1 & \cdots & 0 \\ \hline \vdots & & \vdots & & \vdots \end{array}\right] \end{array}. \tag{6.52}$$

i.e., for edge $j \to i$, the entries $(j, d_1^1), (j, d_2^2)$ and (j, d_3^3) in $A(t)$ are all 1. Finally, we define the matrix $C_{b\times b}$ for forwarding messages from one delay node to the next on each chain. On a specific delay chain of length h, messages are forwarded through the action of an $h \times h$ Toeplitz forward shift matrix with 1s on the first lower

diagonal, i.e.,

$$S_h = \begin{bmatrix} 0 & 0 & \cdots & 0 & 0 & 0 \\ 1 & 0 & & & & 0 \\ 0 & 1 & & & & 0 \\ \vdots & & \ddots & & & \vdots \\ 0 & & & 1 & 0 & 0 \\ 0 & 0 & \cdots & 0 & 1 & 0 \end{bmatrix}. \tag{6.53}$$

For any edge $r = 1, \ldots, m$, to forward messages through all delay chains we use a block diagonal matrix $K_r = \text{diag}(S_1, S_2, \ldots, S_B)$. Finally, since we have m edges

$$C_{b\times b} = \text{diag}(K_1, K_2, \ldots, K_m). \tag{6.54}$$

Recalling (6.50), observe that every row of $[R(k) \;\; C_{b\times b}]$ contains at most one non-zero element and there are rows that are all zero.

Next, we define an indicator vector $\phi(t) \in \{0,1\}^b$ that keeps track of whether a delay node on any delay chain contains a message or is empty. Initially we have $\phi(0) = \mathbf{0}_b$. At iteration t, the first nodes in the delay chains may receive new information depending on which edges are activated by $R(t)$. The rest of the delay nodes will be non-empty only if their predecessors in the chains were non-empty in the previous iteration. In other words, $\phi(t)$ evolves as

$$\phi(t) = R(t)\mathbf{1}_n + C_{b\times b}\phi(t-1). \tag{6.55}$$

After understanding the structure of the time-varying adjacency matrices $A(t)$, to describe the consensus transition matrices $\widehat{P}(t)$ we need to specify the weights used to combine incoming messages. Recall that a compute node may receive multiple messages from a neighboring compute node in one iteration, with each message arriving via a different delay chain. We assign equal weight to all incoming messages from the same sender, and messages from different senders will receive weights according to P. For example, suppose compute node i receives $\widehat{w}_{ij} + L_{ij}(t)$ messages from node j where $0 \le \widehat{w}_{ij} \le B$ are the delayed messages and $L_{ij}(t) = 0$ or 1 is a message without delay. Then node i assigns a weight $\frac{P_{ij}}{\widehat{w}_{ij}+L_{ij}(t)}$ to each of

those messages. In this setting, the self-loop message from i to itself takes weight $P_{ii} + \sum_{k=1}^{n} \mathbb{1}[\widehat{w}_{ik} + L_{ik}(t) = 0]P_{ik}$ where the sum is over all neighboring nodes k from which i does not receive anything at iteration t. Define $\Phi(t) = \text{diag}(\phi(t))$. We can determine which delay nodes at the ends of delay chains have information to be delivered by taking the product $J_{n\times b}\Phi(t-1)$ and locating the entries which are equal to 1. Thus, to construct $\widehat{P}$ we locate all the entries equal to 1 in matrix $J_{n\times b}\Phi(t-1)$ at row i and columns corresponding to deliveries from j, and replace them by $\frac{P_{ij}}{\widehat{w}_{ij}+L_{ij}(t)}$. If $L_{ij}(t) = 1$ we also replace the corresponding entry with $\frac{P_{ij}}{\widehat{w}_{ij}+L_{ij}(t)}$. Let us denote by $\bar{P}[L(t)]$ and $\bar{P}[\phi(t-1)]$ the operators that replace the 1s in $L(t)$ and $J_{n\times b}\Phi(t-1)$ respectively with weights using P. If node i receives no messages from its neighbor j, then $\widehat{w}_{ij} + L_{ij}(t) = 0$ and the weight P_{ij} is transferred to the self-loop (i.e., diagonal weight) of i. The transition matrix $\widehat{P}(t)$ is now written as

$$\widehat{P}(t) = \begin{bmatrix} \widehat{P}_{UL}(t) & \bar{P}[\phi(t-1)] \\ R(t) & C_{b\times b} \end{bmatrix} \tag{6.56}$$

$$\widehat{P}_{UL}(t) = I - \text{diag}(\bar{P}[\phi(t-1)]\mathbf{1}_b + \bar{P}[L(t)]\mathbf{1}_n) + \bar{P}[L(t)]. \tag{6.57}$$

The upper left block of $\widehat{P}(t)$ has this form since, for any row stochastic matrix P, we have $P_{ii} + \sum_{k=1}^{n} \mathbb{1}[\widehat{w}_{ik} + L_{ik}(t) = 0]P_{ik} = 1 - \sum_{k=1}^{n} \mathbb{1}[\widehat{w}_{ik} + L_{ik}(t) > 0]P_{ik}$ for each compute node i. This is just another way of saying that the portion of the weight not used on incoming messages at compute node i from other neighbours is reassigned to the self-loop message.

Observe that the rows of $\widehat{P}(t)$ either sum to zero or to one. Each row i for $i \leq n$ (corresponding to a compute node) is stochastic by construction, while each row i for $n < i \leq n + b$ (corresponding to a delay node) contains at most a single 1 and all other elements are 0. A row $i > n$ corresponding to a delay node d_1^r will contain all zeros if the compute node at the source of the corresponding edge did not send a message through the delay chain r. Let $\widehat{\boldsymbol{z}}(t) \in \mathbb{R}^{n+b}$ denote the augmented state vector of compute and delay nodes. The consensus update equations using $\widehat{P}(t)$ are

given by

$$\widehat{\boldsymbol{z}}(t+1) = \widehat{P}(t+1)\widehat{\boldsymbol{z}}(t), \quad t \geq 0 \tag{6.58}$$

where to construct $\widehat{P}(t+1)$ using (6.56) we need to first update the vector $\phi(t)$ according to (6.55).

The presence of all-zero rows makes the transition matrices $\widehat{P}(t)$ not stochastic, so we need a convergence proof specific to this family of matrices. As we see later, one advantage of the Push-Sum algorithm is that the analysis of the random delay model simplifies and we no longer have to deal with this complication.

6.3.2 Convergence under Time-Varying Delays

To show the iterations generated by the random delay model (6.58) we inspect fundamental properties of the matrices $\{\widehat{P}(t) \colon t = 1, 2, \dots\}$. First we need two standard definitions [99].

Definition 6.1. *A square matrix M is non-expansive with respect to a norm $\|\cdot\|$ if for any vector $\boldsymbol{x}$, we have $\|M\boldsymbol{x}\| \leq \|\boldsymbol{x}\|$.*

Definition 6.2. *A square matrix M is paracontracting with respect to a norm $\|\cdot\|$ if for any vector $\boldsymbol{x}$, we have $\|M\boldsymbol{x}\| < \|\boldsymbol{x}\|$ whenever $M\boldsymbol{x} \neq \boldsymbol{x}$.*

From the construction of the random delay matrices, it is easy to see that the graphs represented by the adjacency matrices $A(t)$ are all connected, and in addition, every compute node performs an averaging operation of the incoming messages. We can thus show that the product of sufficiently many consecutive matrices $\widehat{P}(t)$ is a contractive mapping, leading to convergence.

Theorem 6.3. *The product $\widehat{P}_{2B+1}(t) = \prod_{s=0}^{2B} \widehat{P}(t+s)$ of $2B+1$ consecutive random delay matrices is non-expansive with respect to the infinity norms $\|\cdot\|_{\infty}$ and $\|\cdot\|_{-\infty}$. Moreover, for some integer $r \geq 1$ that depends on the network topology and is not greater that the graph diameter, the product $\widehat{P}_{r(2B+1)}(t)$ is paracontracting. As a result, at every time t, every non-empty node i such that $1 \leq i \leq n+b$ and $\phi_i(t) > 0$ converges to the same value; i.e. $\widehat{z}_i(t) \to c$ as $t \to \infty$.*

Proof. Consider the linear random delayed consensus updates subsampled at intervals of $2B+1$ iterations:

$$\widehat{\boldsymbol{z}}(t) = \widehat{P}_{2B+1}(t)\widehat{\boldsymbol{z}}(t-1), \quad t = 1, 2, \dots \ . \tag{6.59}$$

Recall that the vector $\boldsymbol{\phi}(t)$, which indicates which delay nodes are empty, evolves in parallel to $\widehat{\boldsymbol{z}}(t)$. To focus on the non-empty nodes, define the vector $\boldsymbol{y}(t)$ such that $y_i(t) = \widehat{z}_i(t)$ if $\phi_i(t) > 0$ and $y_i(t) = -\infty$ if $\phi_i(t) = 0$.

We claim that the maximum value of $\boldsymbol{y}(t)$ is less than or equal to the maximum value of $\boldsymbol{y}(t-1)$. If a compute node $i \leq n$ holds the maximum value of $\boldsymbol{y}(t-1)$, in $B+1$ iterations it is certain that i will receive a message from a neighbouring compute node $j \leq n$. If at least one neighbour of i has a smaller value than i, then the value of i will be reduced because i will set its new value to a convex combination of incoming messages (including the self message). However, i may send its (maximum) value to a node $k \leq n$ through the delay chain of length B at iteration t. Regardless of whether the value at i is reduced or not, the maximum of $\boldsymbol{y}(t-1)$ will not change while it is traversing the delay chain towards k. When the message reaches k, node k's value will be reduced unless all of its neighbours have sent messages to k equal to the maximum. To summarize, the maximum value of $\boldsymbol{y}(t-1)$ after $2B+1$ iterations will either stay the same or be reduced. The maximum value will not change if multiple nodes hold that value and there exists at least one node with no neighbors that contain a smaller value. As a result, the maximum value of the state vector will certainly be reduced after $r(2B+1)$ where r is an integer defined as follows. Assume a node i holds the maximum value of $\boldsymbol{y}(t-1)$. If at least one neighbour of i holds a smaller value, then $r = 1$. If all nodes in the distance 1 neighbourhood $N^1(i)$ of i also contain the maximum value then $r = 2$. If the neighbours of the neighbours $N^2(i) = N^1(N^1(i))$ of i contain the maximum value then $r = 3$ and so on. Hence, r is bounded by the diameter of G. Notice also that if the delay nodes were real nodes initialized with random values such that a delay node contained the maximum value in $\boldsymbol{y}(t-1)$, then that value would reach a compute node and would be reduced via an averaging update in at

most $B+1$ iterations. We have shown that $\widehat{P}_{2B+1}(t)$ is non-expansive with respect to $\|\cdot\|_\infty$.

Similarly, since averaging updates of the form (6.59) do not decrease the smallest number in the state vector $\widehat{\boldsymbol{z}}(t)$, $\widehat{P}_{2B+1}(t)$ is also non-expansive with respect to $\|\cdot\|_{-\infty}$ if we define $\boldsymbol{y}'(t)$ so that $y_i'(t) = +\infty$ if $\phi_i(t) = 0$. Moreover, for a given network, we have shown that there exists an integer r such that $\widehat{P}_{r(2B+1)}(t)$ certainly reduces the maximum value of $\boldsymbol{y}(t-1)$ and increases the minimum value of $\boldsymbol{y}'(t-1)$. In other words, every product $\widehat{P}_{r(2B+1)}(t)$ is paracontracting and thus after every $r(2B+1)$ iterations the minimum and maximum values in the graph come close together. Even more importantly, in light of definition 6.2, the minimum keeps increasing and the maximum keeps decreasing, until the minimum equals the maximum. In other words, all values in vector $\widehat{\boldsymbol{z}}(t)$ must converge to the same limit $c \in \mathbb{R}$. □

Even though Theorem 6.3 establishes convergence to consensus under random delays, the actual consensus value c is difficult to characterize since it depends on the specific realization of the process—i.e., on the random matrices $\{\widehat{P}(t)\}_{t=1}^{\infty}$. As future work, it might be possible to extend the results of [66] to describe the statistics of c. However the extension is non-trivial since the results of [66] are based on the assumption that all the matrices $\widehat{P}(t)$ have non-zero diagonals, which is not the case in our model. Here, we show that, as one might expect, c is a convex combination of the initial values v_i at each node $i \in V$. We do this by showing that the upper left $n \times n$ submatrix $\widehat{P}_{UL}(t)$ of $\widehat{P}(t)$ is a row stochastic matrix for all t.

After t iterations we have

$$\widehat{\boldsymbol{z}}(t) = \widehat{P}(t)\widehat{P}(t-1)\cdots\widehat{P}(1)\widehat{\boldsymbol{z}}(0). \tag{6.60}$$

The product $\prod_{k=1}^{t}\widehat{P}(k)$ is a matrix with block structure

$$\prod_{k=1}^{t}\widehat{P}(k) = M(t) = \begin{bmatrix} M_1(t) & M_3(t) \\ M_2(t) & M_4(t) \end{bmatrix} \tag{6.61}$$

where matrix $M_1(t)$ is $n \times n$ and $M_2(t)$ is $b \times n$. After one more iteration we have

$$\begin{aligned}\widehat{\boldsymbol{z}}(t+1) =& \widehat{P}(t+1)M(t)\widehat{\boldsymbol{z}}(0) \\ =& \begin{bmatrix} \widehat{P}_{UL}(t+1) & \bar{P}[\phi(t)] \\ R(t+1) & C_{b\times b} \end{bmatrix} \begin{bmatrix} M_1(t) & M_3(t) \\ M_2(t) & M_4(t) \end{bmatrix} \widehat{\boldsymbol{z}}(0). \end{aligned} \tag{6.62}$$

From the last equation, we obtain the two recursions

$$\begin{aligned} M_1(t+1) =& \Big(I_{n\times n} - \operatorname{diag}\left(\bar{P}[\phi(t)]\mathbf{1}_b + \bar{P}[L(t+1)]\mathbf{1}_n\right) \\ & + \bar{P}[L(t+1)]\Big) M_1(t) + \bar{P}[\phi(t)]M_2(t) \end{aligned} \tag{6.63}$$

$$M_2(t+1) = R(t+1)M_1(t) + C_{b\times b}M_2(t). \tag{6.64}$$

We will show that $M_1(t)$ is row stochastic for all t and that it converges to a rank-1 matrix. We begin by proving three intermediate lemmas and then proceed with the proof of the claim.

Lemma 6.1. *For all $t \geq 1$, $M_2(t)$ and $\phi(t)$ have non-zero rows in exactly the same positions.*

Proof. We will proceed inductively, using the expressions for how $M_2(t)$ and $\phi(t)$ evolve. We have $\phi(1) = R(1)\mathbf{1}_n + C_{b\times b}\phi(0) = R(1)\mathbf{1}_n$ and $M_2(1) = R(1)$, so clearly the non-zero rows of $R(1)$ are the non-zero rows of $M_2(1)$, and they also result in non-zero entries of $\phi(1)$. For the inductive step, let us assume that $\phi(t)$ and $M_2(t)$ have non-zero rows in the same positions. At step $t+1$ we have $\phi(t+1) = R(t+1)\mathbf{1}_n + C_{b\times b}\phi(t)$ and $M_2(t+1) = R(t+1)M_1(t) + C_{b\times b}M_2(t)$. If row i of $\phi(t)$ and $M_2(t)$ is non-zero, then due to multiplication by the shift matrix $C_{b\times b}$, row $i+1$ of $\phi(t+1)$ and $M_2(t+1)$ will be non-zero. Moreover, if a row i of $R(t+1)$ is non-zero then obviously row i of $\phi(t+1)$ will be non-zero. For $M_2(t+1)$, we look at the term $R(t+1)M_1(t)$. Observe that $M_1(t)$ has non-zero diagonal entries for all t. This is easy to see by the update equation (6.63) for $M_1(t)$. As a result, the product $R(t+1)M_1(t)$ will yield non-zero rows of $M_2(t+1)$ wherever a row of $R(t+1)$ is non-zero. This completes the inductive step of the proof. □

The next two lemmas are also inductive, and they are coupled in the sense that their proofs use each other's inductive hypothesis. Specifically, assuming that $M_1(t)$ is row stochastic and the non-zeros rows of $M_2(t)$ sum to 1, we show that the non-zeros rows of $M_2(t+1)$ sum to 1 and $M_1(t+1)$ is row stochastic respectively, establishing that both properties are true for all t.

Lemma 6.2. *The non-zero rows of $M_2(t)$ sum to 1 for all t.*

Proof. Initially, $M_2(1) = R(1)$, and the base case is true. Suppose for every non-zero row $1 \leq i \leq b$ of $M_2(t)$ that $\sum_{j=1}^{n}[M_2(t)]_{ij} = 1$. Also, by inductive hypothesis, suppose that $M_1(t)$ is row stochastic. We will show that the non-zero rows of $M_2(t+1)$ sum to 1. Take any row $1 \leq i \leq b$ of $M_2(t+1)$. We have

$$\begin{aligned}\sum_{j=1}^{n}[M_2(t+1)]_{ij} &= \sum_{j=1}^{n}[R(t+1)M_1(t) + C_{b\times b}M_2(t)]_{ij} \\ &= \sum_{j=1}^{n}[R(t+1)M_1(t)]_{ij} + \sum_{j=1}^{n}[C_{b\times b}M_2(t)]_{ij}. \end{aligned} \tag{6.65}$$

Given the way the delay nodes are arranged in the random delay model, row i of $R(t+1)$ corresponds to a delay node $d_{r_1}^{r_2}$ such that $1 \leq r_2 \leq B$ and $r_1 \leq r_2$. By definition, row i of $R(t+1)$ will be zero if $r_1 > 1$ and may be non-zero if $r_1 = 1$. We thus distinguish two cases:

- *Case* $r_1 = 1$: By definition all rows of $C_{b\times b}$ corresponding to delay nodes at the beginning of delay chains (identified as $d_1^{r_2}$), are zero. If row $i = d_1^{r_2}$ of $R(t+1)$ is non-zero, it will have all entries equal to zero except one entry equal to 1 at some position $1 \leq q \leq n$. As a result

$$\begin{aligned}\sum_{j=1}^{n}[M_2(t+1)]_{ij} &= \sum_{j=1}^{n}[R(t+1)M_1(t)]_{ij} + \sum_{j=1}^{n}[C_{b\times b}M_2(t)]_{ij} \\ &= \sum_{j=1}^{n}[M_1(t)]_{qj} + \sum_{j=1}^{n}\mathbf{0}_b^T[M_2(t)]_{:,j} = \sum_{j=1}^{n}[M_1(t)]_{qj} = 1, \end{aligned} \tag{6.66}$$

since, by inductive hypothesis, $M_1(t)$ has stochastic rows. Of course, if row i of $R(t+1)$ happens to contain only zeros, then the i-th row of $M_2(t+1)$ will be a zero row too.

- *Case* $r_1 > 1$: In this case $\sum_{j=1}^{n}[R(t+1)M_1(t)]_{ij} = 0$ and

$$\sum_{j=1}^{n}[M_2(t+1)]_{ij} = \sum_{j=1}^{n}[C_{b\times b}M_2(t)]_{ij}. \tag{6.67}$$

Since $C_{b\times b}$ is just a shift matrix, each row $i > 1$ of $M_2(t+1)$ will equal to the row $i-1$ of $M_2(t)$ which by inductive hypothesis sums to 1. The first row of $M_2(t+1)$ will be a zero row. □

Lemma 6.3. *Matrix $M_1(t)$ is row stochastic for all t.*

Proof. Proceeding inductively, the base case is true since $M_1(1) = I$. Assume at step $t > 1$ that $\sum_{j=1}^{n}[M_1(t)]_{ij} = 1$ for every row $1 \le i \le n$. At step $t+1$ assume that compute node i receives $\widehat{w}_{ij}$ messages from node j through different delay chains plus possibly a message without delay if $L_{ij}(t+1) = 1$. Since the self loop message is always delivered without delay we know that $\widehat{w}_{ii} = 1$. We have

$$\begin{aligned}
&\sum_{j=1}^{n}[M_1(t+1)]_{ij} \\
&= \sum_{j=1}^{n}\Big[\Big(I_{n\times n} - \mathrm{diag}(\bar{P}[\phi(t)]\mathbf{1}_b + \bar{P}[L(t+1)]\mathbf{1}_n) \\
&\qquad + \bar{P}[L(t+1)]\Big)M_1(t) + \bar{P}[\phi(t)]M_2(t)\Big]_{ij} \qquad (6.68)\\
&= \underbrace{\sum_{j=1}^{n}\Big[\Big(I_{n\times n} - \mathrm{diag}(\bar{P}[\phi(t)]\mathbf{1}_b + \bar{P}[L(t+1)]\mathbf{1}_n)\Big)M_1(t)\Big]_{ij}}_{T_1} \\
&\quad + \underbrace{\sum_{j=1}^{n}\big[\bar{P}[L(t+1)]M_1(t) + \bar{P}[\phi(t)]M_2(t)\big]_{ij}}_{T_2}. \qquad (6.69)
\end{aligned}$$

Consider the term T_1 first, and observe that $I_{n\times n} - \mathrm{diag}(\bar{P}[\phi(t)]\mathbf{1}_b + \bar{P}[L(t+1)]\mathbf{1}_n)$ is a diagonal matrix so we have

$$\begin{aligned}
T_1 =& (1 - [\mathrm{diag}(\bar{P}[\phi(t)]\mathbf{1}_b + \bar{P}[L(t+1)]\mathbf{1}_n]_{ii})\sum_{j=1}^{n}[M_1(t)]_{ij} \\
=& 1 - \sum_{j=1}^{n}\mathbb{1}[\widehat{w}_{ij} > 0 \text{ or } L_{ij}(t+1) > 0]P_{ij}. \qquad (6.70)
\end{aligned}$$

Next let us focus on term T_2 which is composed of two summands. For the first summand we have

$$\begin{aligned}
&\sum_{j=1}^{n}[\bar{P}[L(t+1)]M_1(t)]_{ij} \\
&\quad=\sum_{j=1}^{n}\sum_{k=1}^{n}\bar{P}[L(t+1)]_{ik}[M_1(t)]_{kj} && (6.71)\\
&\quad=\sum_{k=1}^{n}\bar{P}[L(t+1)]_{ik}\sum_{j=1}^{n}[M_1(t)]_{kj} && (6.72)\\
&\quad=\sum_{k=1}^{n}\bar{P}[L(t+1)]_{ik} && (6.73)\\
&\quad=\sum_{k=1}^{n}L_{ik}(t+1)\frac{P_{ik}}{\widehat{w}_{ik}+L_{ik}(t+1)} && (6.74)\\
&\quad=\sum_{j=1}^{n}\mathbb{1}[L_{ij}(t+1)>0]L_{ij}(t+1)\frac{P_{ij}}{\widehat{w}_{ij}+L_{ij}(t+1)}. && (6.75)
\end{aligned}$$

To compute the second summand in T_2, from Lemma 6.1 we know that the non-zero rows of $M_2(t)$ are at the same position as those of $\phi(t)$. Observe now that those positions are the same as the non-zero rows of $J_{n\times b}\Phi(t)$ and thus the non-zero rows of $\bar{P}[\phi(t)]$. Assume that at iteration t node i receives delayed messages only from the compute nodes in the set $\mathcal{N}_i(t)\subseteq V$. Moreover, assume node i receives $\widehat{w}_{in_r}\geq 1$ messages from neighbour $n_r\in\mathcal{N}_i(t)$ through different delay chains. We have

$$\begin{aligned}
\sum_{j=1}^{n}[\bar{P}[\phi(t)]M_2(t)]_{ij}&=\sum_{j=1}^{n}\bar{P}[\phi(t)]_{i,:}[M_2(t)]_{:,j} && (6.76)\\
&=\sum_{j=1}^{n}\sum_{n_r\in\mathcal{N}_i(t)}\sum_{l=1}^{\widehat{w}_{in_r}}\frac{P_{in_r}}{\widehat{w}_{in_r}+L_{in_r}(t+1)}[M_2(t)]_{n_rj} && (6.77)\\
&=\sum_{n_r\in\mathcal{N}_i(t)}\sum_{l=1}^{\widehat{w}_{in_r}}\frac{P_{in_r}}{\widehat{w}_{in_r}+L_{in_r}(t+1)}\sum_{j=1}^{n}[M_2(t)]_{n_rj} && (6.78)\\
&=\sum_{n_r\in\mathcal{N}_i(t)}\frac{P_{in_r}}{\widehat{w}_{in_r}+L_{in_r}(t+1)}\widehat{w}_{in_r} && (6.79)\\
&=\sum_{j=1}^{n}\mathbb{1}[\widehat{w}_{ij}>0]\frac{P_{ij}}{\widehat{w}_{ij}+L_{ij}(t+1)}\widehat{w}_{ij}. && (6.80)
\end{aligned}$$

So now we see that

$$T_2 = \sum_{j=1}^{n} \mathbb{1}[L_{ij}(t+1) > 0] L_{ij}(t+1) \frac{P_{ij}}{\widehat{w}_{ij} + L_{ij}(t+1)}$$
$$+ \sum_{j=1}^{n} \mathbb{1}[\widehat{w}_{ij} > 0] \frac{P_{ij}}{\widehat{w}_{ij} + L_{ij}(t+1)} \widehat{w}_{ij} \tag{6.81}$$
$$= \sum_{j=1}^{n} \mathbb{1}[\widehat{w}_{ij} > 0 \text{ or } \ L_{ij}(t+1) > 0] \tag{6.82}$$
$$\times \frac{P_{ij}}{\widehat{w}_{ij} + L_{ij}(t+1)} (\widehat{w}_{ij} + L_{ij}(t+1)) \tag{6.83}$$
$$= \sum_{j=1}^{n} \mathbb{1}[\widehat{w}_{ij} > 0 \text{ or } \ L_{ij}(t+1) > 0] P_{ij}, \tag{6.84}$$

and finally

$$\sum_{j=1}^{n} [M_1(t+1)]_{ij} = T_1 + T_2 = 1. \tag{6.85}$$

Therefore $M_1(t)$ is row stochastic for all t. □

Finally, we can state the result as follows.

Theorem 6.4. *Given a graph G and a row stochastic consensus matrix P, if we run consensus on G with random delays up to B using updates* (6.58) *with $\widehat{P}(t)$ given by* (6.56)*, all compute nodes of G asymptotically reach consensus on a value c that is a convex combination of their initial values.*

Proof. After t iterations we have $\widehat{\boldsymbol{z}}(t) = M(t)\widehat{\boldsymbol{z}}(0)$ where $\widehat{\boldsymbol{z}}(t)$ is the augmented vector containing the values of the compute nodes followed by all the delay nodes. The delay nodes do not initially contain any information, so we have $[\widehat{\boldsymbol{z}}(0)]_{n+1:n+b} = 0$. After t iterations,

$$\widehat{z}_i(t) = M_1(t)[\widehat{\boldsymbol{z}}(0)]_{1:n} + M_3(t)[\widehat{\boldsymbol{z}}(0)]_{n+1:n+b} \tag{6.86}$$
$$= M_1(t)[\boldsymbol{z}(0)]_{1:n}. \tag{6.87}$$

Therefore, since $M_1(t)$ is row stochastic from Lemma 6.3, we have that $\widehat{z}_i(t) \to c$ as $t \to \infty$ where c is a convex combination of the initial values at the compute nodes. □

As a final comment, observe that the values at compute nodes converge to a consensus even though the overall matrix $M(t)$ does not converge to a limit. Specifically, the rows of $M(t)$ corresponding to delay nodes oscillate between zero and non-zero values. However, this does not affect the upper left $n \times n$ sub-matrix corresponding to the compute nodes. Notice, also, that from this analysis we cannot say anything concrete about the rate of convergence. A convergence rate bound in expectation could be obtained by applying the Poincaré technique from the previous section on $\mathbb{E}[\widehat{P}(t)]$. Alternatively, it might be possible to derive a more accurate bound by analyzing the recursions (6.63), (6.64). After realizing that $C^B = 0$, $M_2(t)$ can be eliminated given enough past terms, and the evolution of $M_1(t)$ resembles that of the impulse response of a multivariate $AR(B)$ model.

6.4 Push-Sum Consensus with Delays

The previous sections study the behavior of general consensus algorithms using row stochastic matrices in the presence of fixed and random delays. In the random delay case the model is a bit involved due to the fact that we need to keep track of which delay nodes are empty, and also a compute node does not know how many messages it will receive at each iteration. Moreover, the convergence proof needs to be tailored specifically to the model because the resulting matrices $\widehat{P}(t)$ are not row stochastic. Even more importantly, we do not have a statement characterizing the convergence rate, and the limiting state is a convex combination of the initial values at each node which is not necessarily the average. In this section we study fixed and random delays with Push-Sum a different consensus algorithm that we already in Chapter 5. As we will see, Push-Sum is a more natural algorithm for distributed averaging in networks with delay; it alleviates all the aforementioned complications, simplifies the delay models, and always converges to the true average.

Recall that Push-Sum makes use of column stochastic consensus matrices and each node i maintains two values: a cumulative estimate of the sum $s_i(t)$ and a weight $u_i(t)$. The local estimate of the average at each iteration is the ratio

$z_i(t) = \frac{s_i(t)}{u_i(t)}$. The algorithm is initialized by setting

$$\boldsymbol{s}(0) = \boldsymbol{z}(0) \text{ and } \boldsymbol{u}(0) = \mathbf{1} \tag{6.88}$$

In vector form the states of all nodes evolve as

$$\boldsymbol{s}(t) = P(t)\boldsymbol{s}(t-1) \text{ and } \boldsymbol{u}(t) = P(t)\boldsymbol{u}(t-1) \tag{6.89}$$

$$\boldsymbol{z}(t) = \frac{\boldsymbol{s}(t)}{\boldsymbol{u}(t)}, \tag{6.90}$$

where the division of $\boldsymbol{s}(t)$ and $\boldsymbol{u}(t)$ in (6.90) is element-wise.

6.4.1 Consensus with Fixed Delays using Push-Sum

In the case of fixed delays, the construction of a matrix with delays $\widehat{P}$ based on an initial matrix P is the same as in Section 6.2. The only difference is that we start with a column stochastic matrix P and convert it to a new column stochastic matrix $\widehat{P}$ by adding delays one edge at a time. For example, if we start with the matrix (6.3), after adding a delay of 2 on the edge $(1, 2)$ we have

$$\widehat{P} = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ d_1 \\ d_2 \end{array} \begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & d_1 & d_2 \end{array} \\ \begin{bmatrix} \frac{2}{3} & \frac{1}{3} & 0 & 0 & 0 \\ 0 & \frac{1}{3} & \frac{1}{2} & 0 & 1 \\ \frac{1}{6} & \frac{1}{3} & \frac{1}{2} & 0 & 0 \\ \frac{1}{6} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix} \end{array}. \tag{6.91}$$

In the case of Push-Sum, delay node d_1 receives $\frac{1}{6}$ of the share of node 1. Using $\widehat{P}$, average consensus is achieved by iterating

$$\widehat{\boldsymbol{s}}(t) = \widehat{P}\widehat{\boldsymbol{s}}(t-1),\ \widehat{\boldsymbol{w}}(t) = \widehat{P}\widehat{\boldsymbol{w}}(t-1). \tag{6.92}$$

The delay nodes are initialized with $s_i(0) = u_i(0) = 0, n+1 \leq i \leq n+b$ since they should not contain any information that might influence the final average. In vector

form

$$\widehat{\boldsymbol{s}}(0) = [\boldsymbol{z}(0)^T \quad \mathbf{0}_b^T]^T \tag{6.93}$$

$$\widehat{\boldsymbol{u}}(0) = [\mathbf{1}_n^T \quad \mathbf{0}_b^T]^T. \tag{6.94}$$

Suppose that Push-Sum iterations are executed using the delay-augmented matrix $\widehat{P}$, and let $\widehat{P}^\infty = \lim_{t\to\infty} \widehat{P}^t$. Observe that, in the limit, the estimate of the average z_i at each node i converges to the average of the initial values,

$$z_i(\infty) = \frac{\left[\widehat{P}^\infty \widehat{\boldsymbol{s}}(0)\right]_i}{\left[\widehat{P}^\infty \widehat{\boldsymbol{u}}(0)\right]_i} = \frac{\left[\widehat{P}^\infty [\boldsymbol{z}(0)^T \quad \mathbf{0}_b^T]^T\right]_i}{\left[\widehat{P}^\infty [\mathbf{1}_n^T \quad \mathbf{0}_b^T]^T\right]_i} \tag{6.95}$$

$$= \frac{\sum_{j=1}^n \widehat{P}_{ij}^\infty z_j(0)}{\sum_{j=1}^n \widehat{P}_{ij}^\infty} = \frac{\widehat{P}_{i1}^\infty \sum_{j=1}^n z_j(0)}{\widehat{P}_{i1}^\infty \sum_{j=1}^n 1} = \frac{\sum_{j=1}^n z_j(0)}{n} \tag{6.96}$$

since $\widehat{P}$ is column stochastic and $\widehat{P}^\infty$ will have identical columns. Obviously, the convergence rate bound (6.19) applies here as well.

6.4.2 Consensus with Random Delays using Push-Sum

In row stochastic matrices with random delays, we need an indicator vector $\phi(t)$ to know whether a delay node contains information or is empty. We also need to assign the portion of the weight that is being unused to the self-loop message. Both of those complications arise from the fact that we do not know how many messages will be received at each iteration. With Push-Sum consensus however, the semantics suggest that the sending node decides how much weight to assign to each outgoing message, and each receiving node simply adds up the values of s and w for all incoming messages without caring about the number of messages received from each neighbor. This fact simplifies both the model and the convergence analysis when we account for time-varying delays.

Recall from the random delay model construction that the adjacency matrix $A(t)$ is given by (6.50). However, now we are given a column stochastic matrix P and need to construct an augmented column stochastic matrix $\widehat{P}(t)$. Since P

indicates the outgoing weights, the construction is straightforward:

$$\widehat{P}(t) = \begin{bmatrix} \mathrm{diag}(P) + P \circ L(t) & J_{n\times b} \\ \bar{P}[R(t)] & C_{b\times b} \end{bmatrix}, \tag{6.97}$$

where $\mathrm{diag}(P)$ denotes a matrix with diagonal entries the same as those of P and off-diagonal entries set to zero, and where $\circ$ denotes entry-wise (Hadamard) matrix multiplication. For the analysis of Push-Sum we also define the operator $\bar{P}[R(t)]$ a bit differently than in the previous section. If $[R(t)]_{d_1^r,j} = 1$, where d_1^r is the first node on a delay chain from compute node j to compute node i, then we set $\bar{P}[R(t)]_{d_1^r,j} = P_{ij}$. Again for the purpose of analysis we initialize the values $\widehat{s}_i(0)$ and $\widehat{u}_i(0)$ for the delay states $i \in \widehat{V} \setminus V$ to zero.

With Push-Sum, the model is simplified because we no longer need the vector ϕ to indicate which delay nodes contain information. Instead, we use the weights $\widehat{\boldsymbol{u}}(t)$, and an empty delay node is represented by having a weight of zero. Notice, in addition, that $\widehat{P}(t)$ is column stochastic by construction and does not contain zero columns. This allows us to use weak ergodicity theory [73, 82] to establish convergence.

6.4.3 Convergence of Push-Sum Consensus with Random Delays

Using the random delay model with column stochastic matrices yields a forward product, and to prove that the iterates $\boldsymbol{z}(t)$ converge we need to establish weak ergodicity. Since each matrix $\widehat{P}(t)$ in (6.97) contains zeros on the diagonal, we cannot apply known results directly. In this section we derive a worst case (pessimistic) geometric convergence rate. We first need the following lemma.

Lemma 6.4. *If a strongly connected graph G has diameter D, the graph $\widehat{G}$ obtained by adding arbitrary delays of up to B on each edge has diameter at most $\widehat{D} \leq (B+1)D + B + 1$.*

Proof. Let $K = v \to v_1 \to \cdots \to v_{D-1} \to w$ be a path in G with length equal to D. By adding at most B delay nodes per directed edge, each edge of G is replaced by $B+1$ edges in $\widehat{G}$ and the corresponding path $\widehat{K}$ has length $(B+1)D$ in $\widehat{G}$. All neighbours of v and w in G belong to K or else the diameter would be longer.

Suppose that in the worst case, v has a neighbor $z_1 \neq v_1$ and w has a neighbor $z_2 \neq v_{D-1}$ in G. After adding delays, the longest path in $\widehat{G}$ goes from the delay node in the middle of the longest delay chain between z_1 and v and the delay node in the middle of the longest delay chain between z_2 and w and has length at most $\widehat{D} \leq (B+1)D + \frac{B+1}{2} + \frac{B+1}{2} = (B+1)D + B + 1$. □

Now we can state the main convergence result of this section.

Theorem 6.5. *If we run Push-Sum on a strongly connected graph G using a column stochastic weight matrix P, then in the presence of bounded time-varying delays modelled by* (6.97)*, consensus on the average is achieved at a geometric rate.*

Proof. Since G is strongly connected, due to the way we model random delays, at each iteration t there exists a path between any two compute nodes i and j. As a consequence, due to Lemma 6.4, every column $j \leq n$ of every sub-product matrix $F(r, r+\widehat{D}) = \widehat{P}(r)^T \widehat{P}(r+1)^T \cdots \widehat{P}(r+\widehat{D})^T$ contains positive entries.[6] This means that for the (improper) coefficient of ergodicity $\chi(\cdot)$ defined in [73](p. 137) as

$$\chi\big(F(r, r+\widehat{D})\big) = 1 - \max_{1 \leq s \leq n+b} (\min_k [F(r, r+\widehat{D})]_{ks}) \tag{6.98}$$

$$\leq 1 - \max_{1 \leq s \leq n} (\min_k [F(r, r+\widehat{D})]_{ks}) < 1 \tag{6.99}$$

since the maximum over the minimum values in the compute node columns is certainly not zero. After running Push-Sum for $t > \widehat{D}$ iterations, divide the forward product $F(1,t)$ into intervals of $\widehat{D}$ iterations,

$$F(1,t) = \prod_{k=1}^{\lceil \frac{t}{\widehat{D}} \rceil} F((k-1)\widehat{D}+1, k\widehat{D}) \tag{6.100}$$

$$= F(1,\widehat{D}) F(\widehat{D}+1, 2\widehat{D}) \cdots F(t-\widehat{D}+1, t). \tag{6.101}$$

[6] In other words after $\widehat{D}$ iterations every compute node has received a message from all other compute nodes.

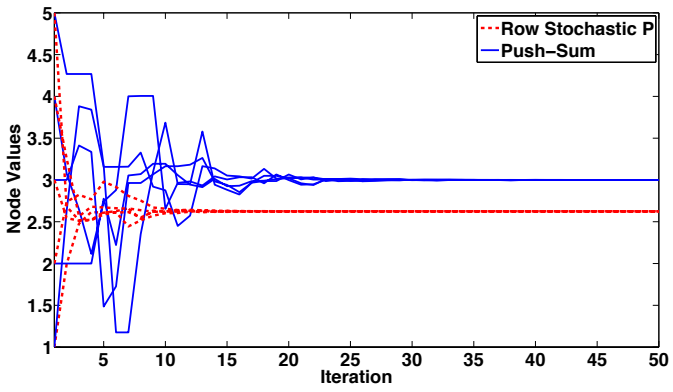

Figure 6–7: Evolution of the node values on a graph of 5 nodes with random delays no more than $B = 5$. The true average is $x_{ave} = 3$. (Blue) With Push-Sum all nodes reach consensus to the correct average. (Red) Using a row stochastic matrix, as expected consensus is reached but not to the average and the consensus value varies between executions.

As explained above, $\chi\big(F((k-1)\widehat{D}+1, k\widehat{D})\big) < 1$ for each term. It follows that $\sum_{k=1}^{\infty}\Big[1-\chi\big(F((k-1)\widehat{D}+1, k\widehat{D})\big)\Big] = \infty$, and from Theorem 4.9 in [73], the product $F(1,t)$ is weakly ergodic. Based on a derivation similar to (6.95), after initializing $\widehat{s}_i(0) = 0$ and $\widehat{u}_i(0) = 0$ for delay nodes i, the Push-Sum values at each compute node converge to the true average. Furthermore, if $\max_k \big(F((k-1)\widehat{D}+1, k\widehat{D})\big) \leq \chi_0 < 1$, the forward product converges geometrically at a rate no worse than χ_0. □

6.5 Simulation

To illustrate that Push-Sum is resilient to time-varying delays, we simulate a random network with 5 nodes and a maximum random delay of $B = 5$. We plot the evolution of the node values when running consensus with equation (6.58) and Push-Sum using consensus matrices of the form (6.97). We initialize the node values to be the node ids 1 through 5. In both cases we start with a random row stochastic matrix P without delays and use its transpose to generate the Push-Sum weights. Figure 6–7 illustrates that since P is not doubly stochastic, the compute nodes reach consensus as Theorem 6.4 suggests, but the consensus value is not the average. Even worse, if we run the simulation again, the different random delays at each iteration will yield a different consensus value. With Push-Sum, on the other hand, the compute nodes always converge to the true average.

6.6 Concluding Remarks and Future Work

In this chapter we analyze the effect of communication delays in distributed consensus and optimization algorithms. Initially we assume that each directed link of a communication network G delivers messages with some fixed delay B. Delays on different links need not be equal. We show how to model the effect of delays by augmenting G with artificial delay nodes, and then we use geometric arguments to show that a bound on the inverse spectral gap of a consensus matrix $\widehat{P}$ in the presence of delays does not increase faster than $\Theta(B^2)$. Thus, delayed consensus iterations converge exponentially fast to a value which, in general, is not the average. For a given row stochastic weight matrix with known stationary distribution, consensus on the average could be achieved by rescaling the initial values.

Next, we show how to model time-varying delays—a scenario that is more realistic but also harder to analyze. For general row stochastic consensus weight matrices we show that convergence to a consensus is still guaranteed, although the consensus value is itself a random variable depending on the delay realizations. If however we use Push-Sum as the consensus algorithm we are guaranteed to converge to a consensus on the average, and the analysis of the time-varying delay model is significantly simplified. These facts are together with Chapter 5, suggest that Push-Sum is more suitable for practical implementations.

Following the example in Section 6.2.5, we can analyze PS-DDA with random delays as well. This derivation is not shown because it will look exactly like the results in Chapter 5. To obtain a more informative result we would need a precise expression of the convergence rate for random delays; a result which is currently missing in the literature. If that result existed, we could use it in the bound for the network error such as $\|\overline{\boldsymbol{z}}(t) - \boldsymbol{z_k}(t)\|$ for DDA or $\left\|\overline{\boldsymbol{z}}(t) - \frac{\boldsymbol{z_k}(t)}{u_k(t)}\right\|$ for PS-DDA.

In the future, for the fixed delay scenario we would like to investigate the following optimization problem: Given a network G, find the consensus weights P that respect the structure of G and reach consensus as fast as possible in the presence of known fixed delays b_{ij}. Observe that since we can use Push-Sum, any column stochastic matrix that does not add edges to G will compute the average

and we are looking for the matrix with the smallest second eigenvalue. It would be interesting to investigate if the techniques used for second eigenvalue optimization for symmetric consensus matrices (see e.g., [15]) could be extended to answer this question.

For the time-varying delay models considered in this paper, the analysis only guarantees convergence with a loose geometric bound on the convergence rate for the Push-Sum algorithm. It would be useful to have a more precise characterization of the convergence rate and to extend the Poincaré technique presented in this paper to understand the extent to which time-varying delays slow down convergence.

CHAPTER 7
Improved Convergence Rate for Distributed Convex Optimization

7.1 Introduction

The last part of this thesis focuses on the theoretical analysis of consensus-based distributed optimization. Chapter 2 discusses the discrepancies between the optimal convergence rates for serial algorithms and their distributed counterparts. This and the next chapter bridge some of the gaps between serial gradient based optimization algorithms and their distributed counterparts. In particular, our results improve the time complexity for fixed networks. An important direction for future work would be to see if the dependence on the network size n can also be improved.

This chapter specifically, focuses on general possibly non-smooth convex optimization problems which is exactly the setup of DDA[1] . We improve on DDA, by describing a distributed algorithm that achieves a time-optimal rate of $O\left(\frac{1}{\sqrt{T}}\right)$ without increasing the communication cost or worsening the dependence on the network size n.

7.2 A Time Optimal Algorithm for General Convex Functions

Looking at the original DDA algorithm from Section 2.5.1 we notice two things. In the special case where there is only one node, DDA reduces to Nesterov's dual averaging algorithm [61]. However, for centralized dual averaging, it is known [61] that the error $F(\overline{\boldsymbol{w}}(T)) - F(\boldsymbol{w}^*)$ decays at a rate $O\left(\frac{1}{\sqrt{T}}\right)$, whereas when we take $n = 1$ and $\lambda_2(P) = 0$ in (2.33), the bound on the error is only $O\left(\frac{\log(T)}{\sqrt{T}}\right)$, and so the bound is loose by a factor of $\log(T)$. We remark though that to save the logarithmic factor in dual averaging, one needs to decide the number of iterations T in advance in order to select the step size appropriately [60]. Even though predetermining the

[1] This chapter is based on the recently submitted work [93]

number of iterations does not seem to benefit DDA we will see that in our proposed algorithm pre deciding T helps tune the amount of communication to save a $\log(T)$ factor.

Also consider the case where G is the complete graph on n nodes (i.e., $E = V \times V$). Then with $P = \frac{1}{n}\mathbf{1}\mathbf{1}^T$, we have $\lambda_2(P) = 0$ regardless of n. In this case, each time nodes perform the consensus update their values should be exactly synchronized, i.e., $\sum_{j=1}^{n} P_{ij}\boldsymbol{z_j}(t)$ is the same at all nodes i. However, in DDA the dual variables $\boldsymbol{z_i}(t)$ are not all identical because the subgradients $\boldsymbol{g_i}(t)$ at each node are not necessarily the same. Although one would hope to achieve the $O(\frac{1}{\sqrt{T}})$ error rate equivalent to the centralized method in this case, the bound given by (2.33) is $O\left(\frac{\log(T\sqrt{n})}{\sqrt{T}}\right)$.

Motivated by these two observations, we propose the following consensus-based proximal gradient distributed optimization algorithm. Each node $i \in V$ still maintains primal and dual variables $\boldsymbol{w_i}(t), \boldsymbol{z_i}(t) \in \mathbb{R}^d$. Similar to DDA, we make use of a sequence of non-increasing positive step sizes, $\{a(t)\}_{t=0}^{\infty}$, and each node initializes $\boldsymbol{z_i}(0) = \mathbf{0}$.

Each iteration of the proposed algorithm proceeds with the following steps. First, each node i computes a subgradient $\boldsymbol{g_i}(t) \in \partial f_i(\boldsymbol{w_i}(t))$, and sets

$$\widetilde{\boldsymbol{z}}_{\boldsymbol{i}}(t) = \boldsymbol{z_i}(t) + \boldsymbol{g_i}(t). \tag{7.1}$$

Next, for an integer $h \geq 1$, the nodes perform h consensus steps on the intermediate dual variables $\widetilde{\boldsymbol{z}}_{\boldsymbol{i}}(t)$, by setting

$$\boldsymbol{z_i}(t+1) = \sum_{j=1}^{n} [P^h]_{ij}\widetilde{\boldsymbol{z}}_{\boldsymbol{j}}(t), \tag{7.2}$$

Finally, the primal variables are updated by a proximal projection,

$$\boldsymbol{w_i}(t+1) = \Pi_{\mathcal{X}}^{\psi}\left[\boldsymbol{z_i}(t+1), a(t)\right]. \tag{7.3}$$

The difference between the proposed algorithm and DDA is subtle, but it has non-trivial impact on the resulting performance bounds. In the proposed approach,

nodes incorporate their gradient and then perform the consensus iterations, while in DDA information is diffused through consensus first. This is analogous to the distinction between the "adapt-then-combine" and "combine-then-adapt" approaches of [25]. Furthermore, we allow execution of more than one communication between the gradient steps.

Our main results for the proposed algorithm are summarized in the following theorem.

Theorem 7.1. *Let $\boldsymbol{w}^* \in \mathcal{W}$ and select a 1-strongly convex function $\psi(\boldsymbol{w})$ such that $\psi(\boldsymbol{w}^*) \leq R^2$. Let $\{\boldsymbol{w_i}(t)\}_{t=0}^{\infty}$ and $\{\boldsymbol{z_i}(t)\}_{t=0}^{\infty}$ be generated by the proposed updates (7.1)–(7.3). If $\lambda_2(P) = 0$ (i.e., G is the complete graph on n nodes), if nodes perform $h = 1$ consensus steps per iteration, and if nodes use the step-size sequence $a(t) = \frac{R}{L\sqrt{t}}$, then for every node $i \in V$,*

$$F(\overline{\boldsymbol{w}}_{\boldsymbol{i}}(T)) - F(\boldsymbol{w}^*) \leq \frac{L(1+R)}{\sqrt{T}}. \tag{7.4}$$

Alternatively, if $\lambda_2(P) > 0$ and nodes perform $h \geq \frac{\log(T\sqrt{n})}{1-\lambda_2(P)}$ consensus steps per iteration, and if they use the step-size sequence $a(t) = \frac{R}{L\sqrt{19t}}$, then for every node $i \in V$,

$$F(\overline{\boldsymbol{w}}_{\boldsymbol{i}}(T)) - F(\boldsymbol{w}^*) \leq \frac{\sqrt{19}LR}{\sqrt{T}}. \tag{7.5}$$

We note that, regardless of the topology, the error rate, in terms of iterations, of the proposed method is optimal, i.e., the error in the objective decays at a rate $O\left(\frac{1}{\sqrt{T}}\right)$. When G is not the complete graph, we achieve this by applying $h > 0$ consensus steps per iteration to drive the network error down to a desired accuracy. The total amount of communications (i.e., the total number of consensus steps) is identical to that of DDA, but the proposed approach requires fewer iterations.

7.3 Analysis and Proof of Theorem 7.1

At a high level, the proof of Theorem 7.1 follows similar steps to those for the proof of Theorem 2 in [32]. We have already seen two similar derivations in Chapters 5 and 6 so here we just emphasize those steps which are different.

The nodes apply h consensus steps in the dual variable update (7.2), which is equivalent to setting

$$\boldsymbol{z_i}(t+1) = \sum_{j=1}^{n} \widetilde{P}_{ij}\big(\boldsymbol{z_j}(t) + \boldsymbol{g_j}(t)\big), \tag{7.6}$$

where $\widetilde{P} = P^h$. Compared to (2.22), each node incorporates its more recent gradient to the dual variable *before* communicating. Since P is doubly stochastic, $\widetilde{P}$ is also doubly stochastic, and with the initialization $\boldsymbol{z_i}(0) = \mathbf{0}$ we have

$$\begin{aligned}
\overline{\boldsymbol{z}}(t+1) &= \frac{1}{n}\sum_{j=1}^{n} \boldsymbol{z_i}(t+1) && (7.7)\\
&= \frac{1}{n}\sum_{j=1}^{n}\left(\sum_{i=1}^{n}\widetilde{P}_{ij}\right)\big(\boldsymbol{z_j}(t) + \boldsymbol{g_j}(t)\big) && (7.8)\\
&= \frac{1}{n}\sum_{j=1}^{n}\big(\boldsymbol{z_j}(t) + \boldsymbol{g_j}(t)\big) && (7.9)\\
&= \overline{\boldsymbol{z}}(t) + \frac{1}{n}\sum_{j=1}^{n}\boldsymbol{g_j}(t) && (7.10)\\
&= \sum_{r=1}^{t+1}\left(\frac{1}{n}\sum_{j=1}^{n}\boldsymbol{g_j}(r)\right). && (7.11)
\end{aligned}$$

The average dual variable $\overline{\boldsymbol{z}}(t+1)$ should be thought of as the update one would get by performing centralized updates, and the corresponding primal variables are

$$\boldsymbol{y}(t+1) = \Pi_{\mathcal{X}}^{\psi}(\overline{\boldsymbol{z}}(t+1), a(t)), \tag{7.12}$$

which should be compared with the primal variables $\boldsymbol{w_i}(t+1)$ at each node. We also define the centralized running average after T iterations, $\overline{\boldsymbol{y}}(T) = \frac{1}{T}\sum_{t=1}^{T}\boldsymbol{y}(t)$, which is to be compared with $\overline{\boldsymbol{w}}_{\boldsymbol{i}}(T)$.

Similar to (5.27), by back-substituting for $\boldsymbol{z_j}(t)$ in (7.6), the dual variable $\boldsymbol{z_i}(t)$ at node i is expressed as

$$\boldsymbol{z_i}(t+1) = \sum_{r=1}^{t}\sum_{j=1}^{n}\left[\widetilde{P}\right]_{ij}\boldsymbol{g_j}(r). \tag{7.13}$$

As will be seen, the network errors $\|\overline{\boldsymbol{z}}(t) - \boldsymbol{z_i}(t)\|$ play a key role in bounding the error of the proposed algorithm.

Via standard convexity arguments (see [32, 61])) and using steps similar to Section 6.2.5 we can retrieve Theorem 2.1, which can be written as

$$F(\overline{\boldsymbol{w}}_{\boldsymbol{j}}(T)) - F(\boldsymbol{w}^*) \leq \textsc{Opt} + \textsc{Net}, \tag{7.14}$$

where

$$\textsc{Opt} = \frac{\psi(\boldsymbol{w}^*)}{Ta(T)} + \frac{L^2}{2T}\sum_{t=1}^{T} a(t), \text{ and}$$
$$\textsc{Net} = \frac{L}{T}\sum_{t=1}^{T} a(t)\left[\|\overline{\boldsymbol{z}}(t) - \boldsymbol{z}_{\boldsymbol{j}}(t)\| + \frac{2}{n}\sum_{i=1}^{n}\|\overline{\boldsymbol{z}}(t) - \boldsymbol{z}_{\boldsymbol{i}}(t)\|\right].$$

The two terms in Opt are similar to those typically arising in serial subgradient methods [61], and the terms in Net quantify the error incurred due to running the algorithm over a network [32]. In particular, the terms $\|\overline{\boldsymbol{z}}(t) - \boldsymbol{z}_{\boldsymbol{i}}(t)\|$ measure how far the dual variable at node i is from the dual variable $\overline{\boldsymbol{z}}(t)$ of the centralized sequence after t iterations.

The main (subtle) difference from the DDA bound in [32] comes in the form of $\boldsymbol{z}_{\boldsymbol{j}}(t)$ which affects the network error. For the proposed method we have the expression (7.13), whereas for DDA, we get

$$\boldsymbol{z}_{\boldsymbol{i}}^{\mathrm{DDA}}(t+1) = \boldsymbol{g}_{\boldsymbol{i}}(t) + \sum_{r=0}^{t-1}\sum_{j=1}^{n}[P^{t-r}]_{ij}\boldsymbol{g}_{\boldsymbol{j}}(r). \tag{7.15}$$

Consequently, even when G is the complete graph so that one consensus step suffices for all nodes to reach a consensus on the average, the network error for DDA is still

$$\overline{\boldsymbol{z}}(t+1) - \boldsymbol{z}_{\boldsymbol{i}}^{\mathrm{DDA}}(t+1) = \frac{1}{n}\sum_{j=1}^{n}\big(\boldsymbol{g}_{\boldsymbol{j}}(t) - \boldsymbol{g}_{\boldsymbol{i}}(t)\big), \tag{7.16}$$

which is non-zero in general.

On the other hand, if G is the complete graph, from (7.13) the network error vanishes and the iterates $\{(\boldsymbol{w}_{\boldsymbol{i}}(t), \boldsymbol{z}_{\boldsymbol{i}}(t))\}_{t=0}^{\infty}$ of proposed algorithm are equivalent to those of a centralized dual averaging algorithm for solving (1.8). To see why,

observe that, in general, we have

$$\|\boldsymbol{z_i}(t) - \overline{\boldsymbol{z}}(t)\| = \left\| \sum_{r=0}^{t-1} \sum_{j=1}^{n} \left(\left[\widetilde{P}^{t-r} \right]_{i,j} - \frac{1}{n} \right) \boldsymbol{g_j}(r) \right\|$$
$$\leq L \sum_{r=0}^{t-1} \left\| \widetilde{P}^{t-r} \boldsymbol{e}_i - \mathbb{1}/n \right\|_1, \tag{7.17}$$

where $\boldsymbol{e}_i$ is the ith canonical vector. When G is the complete graph, $P = \widetilde{P} = \frac{1}{n}\mathbf{1}\mathbf{1}^T$, in which case the right-hand side of the expression above vanishes.

When G is not the complete graph, we obtain a bound in terms of the second largest eigenvalue $\lambda_2(P)$ of P. Since $\widetilde{P}$ is doubly stochastic, for any probability vector $\boldsymbol{q} \in \mathbb{R}^n$ (i.e., with $q_i \geq 0$ for all i, and $\sum_{i=1}^n q_i = 1$) and $s \geq 1$,

$$\left\| \widetilde{P}^s \boldsymbol{q} - \frac{\mathbf{1}}{n} \right\|_1 \leq \lambda_2(\widetilde{P})^s \sqrt{n} = (\lambda_2(\widetilde{P}))^s \sqrt{n}. \tag{7.18}$$

Therefore, if $s \geq s' = \left\lceil \frac{\log(T\sqrt{n})}{\log(\lambda_2(\widetilde{P})^{-1})} \right\rceil$ then

$$\left\| \widetilde{P}^s \boldsymbol{q} - \frac{\mathbf{1}}{n} \right\|_1 \leq \frac{1}{T}. \tag{7.19}$$

Now, let us rewrite (7.17) with the substitution $s = t - r$,

$$\|\boldsymbol{z_i}(t) - \overline{\boldsymbol{z}}(t)\| \leq L \sum_{s=1}^{s'} \left\| \widetilde{P}^s \boldsymbol{e_i} - \frac{\mathbf{1}}{n} \right\|_1 + L \sum_{s=s'}^{t} \left\| \widetilde{P}^s \boldsymbol{e}_i - \frac{\mathbf{1}}{n} \right\|_1.$$

For $s < s'$, we use the bound $\left\| \widetilde{P}^s \boldsymbol{e_i} - \frac{\mathbf{1}}{n} \right\|_1 \leq 2$. When $s \geq s'$, the right hand sum is less than $L \sum_{s=s'}^{t} \frac{1}{T} < L$ since $t \leq T$. These bounds only apply when $\lambda_2(\widetilde{P}) > 0$ since for a complete graph $\lambda_2(P) = 0$. We have,

$$\|\boldsymbol{z_i}(t) - \overline{\boldsymbol{z}}(t)\| \leq \left(\frac{2L \log(T\sqrt{n})}{\log(\lambda_2(\widetilde{P})^{-1})} + L \right) \mathbb{I}\{\lambda_2(\widetilde{P}) > 0\}, \tag{7.20}$$

where $\mathbb{I}\{\cdot\}$ is the 0/1 indicator function. Consequently, if $\lambda_2(P) = 0$ the only error is due to Opt. Next observe that

$$\sum_{t=1}^{T} \frac{A}{\sqrt{t}} \leq 1 + \int_0^T t^{1/2} dt = 2\sqrt{T} - 1 < 2\sqrt{T}.$$

Therefore, using the step sizes as defined in Theorem 7.1, we have proven the first claim.

Consider now $\lambda_2(P) > 0$. Since $\lambda_2(\widetilde{P}) = \lambda_2(P)^h$, take

$$h \geq \frac{\log(T\sqrt{n})}{1-\lambda_2(P)} \geq \frac{\log(T\sqrt{n})}{\log(\lambda_2(P)^{-1})}. \tag{7.21}$$

so that $\frac{\log(T\sqrt{n})}{\log(\lambda_2(\widetilde{P})^{-1})} < 1$ and $\|\boldsymbol{z_i}(t) - \overline{\boldsymbol{z}}(t)\| \leq 3L$. In this case, we obtain

$$\textsc{Net} \leq \frac{18L}{T} \sum_{t=1}^{T} a(t). \tag{7.22}$$

Using a step size sequence $a(t) = \frac{A}{\sqrt{t}}$, and minimizing the bound for A yields the step size defined in Theorem 7.1, which completes the proof.

7.4 Comments

We described a distributed optimization algorithm whose error decays at the same rate as the optimal centralized algorithm, in terms of the number of iterations. Our algorithm relies on more communication per iteration that standard DDA. When the communication topology is the complete graph, our scheme improves upon existing bounds in the literature. For other topologies the proposed method uses the same total amount of communications. Whereas other approaches communicate once per iteration and require a number of iterations which grows with the network size, the proposed approach requires the same number of iterations regardless of the network size, and the number of consensus steps per iteration grows with the network size. Alternatively, nodes could communicate "fewer than once" per iteration (i.e., take multiple gradient steps before communicating). This approach is explored in Chapter 3.

CHAPTER 8
Distributed Strongly Convex Optimization

8.1 Introduction

In this last part of the thesis we study the problem of online strongly convex optimization[1] where the optimization algorithm may receive a different cost function at every iteration. As mentioned in [77] if we feed the same cost to the algorithm at all iterations, we immediately obtain a bound for an offline optimization problem. The setup has already been discussed in Section 2.4 but we quickly review the basic concepts here as well.

The problem we want to solve is

$$\underset{\boldsymbol{w}\in\mathcal{W}}{\text{minimize}}\ F(\boldsymbol{w}) = \frac{1}{m}\sum_{t=1}^{m} f(\boldsymbol{w}, \boldsymbol{x}(t)) \tag{8.1}$$

where the cost functions arrive as a stream $f(\boldsymbol{w}, \boldsymbol{x}(1)), f(\boldsymbol{w}, \boldsymbol{x}(2)), \ldots$ and each of them is L-Lipschitz continuous and strongly convex (see definition 2.1). In general the cost functions are not related to one another and do not need to have the same functional form. With a slight abuse of notation, we denote each cost by $f(\boldsymbol{w}, \boldsymbol{x}(t))$ to describe a function that is parameterized by data $\boldsymbol{x}(t)$. Recall that if the data is generated i.i.d. from some (unknown distribution) then the problem is referred to as stochastic optimization. Stochastic optimization is more relevant to machine learning and empirical loss minimization but our algorithm does not rely on the i.i.d. data assumption.

In a distributed setting, each of our n processors has access to its own independent stream. The performance of the algorithm is quantified by the cumulative

[1] This chapter is based on the previously published work [91]

regret which is defined as

$$R_n(T) = \sum_{i=1}^{n}\sum_{t=1}^{T} f(\boldsymbol{w}_i(t), \boldsymbol{x}_i(t)) - \operatorname*{argmin}_{\boldsymbol{w}\in\mathcal{W}} \sum_{i=1}^{n}\sum_{t=1}^{T} f(\boldsymbol{w}, \boldsymbol{x}_i(t)). \tag{8.2}$$

As already discussed in Section 2.4 for optimization we can bound the optimization error by the average regret

$$F(\overline{\boldsymbol{w}}(t)) - F(\boldsymbol{w}^*) \leq \frac{R_n(T)}{nT}. \tag{8.3}$$

The algorithm presented in this chapter achieves a rate $O\left(\frac{\log(\sqrt{n}T)}{T}\right)$ distributedly i.e., with each processor processing its own data stream while the processors converge to the global minimum. Consulting Chapter 2, this rate is optimal in the number of iterations T for strongly convex problems.

8.2 Distributed Online Gradient Descent (DOGD)

Assume again that we have a network of n processors endowed with a $n \times n$ consensus matrix P which respects the structure of G, in the sense that $[P]_{ji} = 0$ if $(i, j) \notin E$. We assume that P is doubly stochastic, although generalizations to the case where P is just row or column stochastic are possible based on the discussion of chapters 5, 6.

We propose the *distributed online gradient descent* (DOGD) algorithm to solve problem (8.1). A pseudo-code description is given in Algorithm 2. In the algorithm, each node performs a total of T updates to process $nT = m$ data points. One update involves processing a single data point $\boldsymbol{x_i}(t)$ at each processor. The updates are performed over k rounds, and T_s updates are performed in round $s \leq k$. The main steps within each round (lines 10–12) involve updating an accumulated gradient variable, $\boldsymbol{z_i^k}(t)$, by simultaneously incorporating the information received from neighboring nodes and taking a local gradient-descent like step. The accumulated gradient is projected onto the constraint set to obtain $\boldsymbol{w_i^k}(t)$, where

$$\Pi_{\mathcal{W}}[\boldsymbol{z}] = \operatorname{argmin} \boldsymbol{w} \in \mathcal{W} \|\boldsymbol{w} - \boldsymbol{z}\| \tag{8.4}$$

denotes the Euclidean projection of $\boldsymbol{z}$ onto $\mathcal{W}$, and then this projected value is merged into a running average $\overline{\boldsymbol{w}}_{\boldsymbol{i}}(r)$. The step size parameter a_k remains constant within each round, and the step size is reduced by half at the end of each round. The number of updates per round doubles from one round to the next.

Algorithm 2 DOGD

Initialize: $T_1 = \left\lceil \frac{2}{\sigma} \right\rceil, a_1 = 1, k = 1, \boldsymbol{z}_{\boldsymbol{i}}^1(1) = \boldsymbol{w}_{\boldsymbol{i}}^1(1) = 0$

while $\sum_{s=1}^{k} T_s \leq T$ **do** ▷ Each node i repeats
  for $t = 1$ to T_k **do**
    Send/receive $\boldsymbol{z}_{\boldsymbol{i}}^k(t)$ and $z_j^k(t)$ to/from neighbors
    Obtain next subgradient $\boldsymbol{g}_{\boldsymbol{i}}(t) \in \partial_{\boldsymbol{w}} f(\boldsymbol{w}_{\boldsymbol{i}}^k(t), \boldsymbol{x}_{\boldsymbol{i}}(t))$
    $\boldsymbol{z}_{\boldsymbol{i}}^k(t+1) = \sum_{j=1}^{n} P_{ij} \boldsymbol{z}_{\boldsymbol{j}}^k(t) - a_k \boldsymbol{g}_{\boldsymbol{i}}(t)$
    $\boldsymbol{w}_{\boldsymbol{i}}^k(t+1) = \Pi_{\mathcal{W}} \left[\boldsymbol{z}_{\boldsymbol{i}}^k(t+1) \right]$
  end for
  $\boldsymbol{w}_{\boldsymbol{i}}^{k+1}(1) = \boldsymbol{w}_{\boldsymbol{i}}^k(T_k)$
  $\boldsymbol{z}_{\boldsymbol{i}}^{k+1}(1) = \boldsymbol{z}_{\boldsymbol{i}}^k(T_k)$
  $\overline{\boldsymbol{w}}_{\boldsymbol{i}}^{k+1} = \frac{1}{T_k} \sum_{t=1}^{T_k} \boldsymbol{w}_{\boldsymbol{i}}^k(t)$

  $T_{k+1} \leftarrow 2T_k$
  $a_{k+1} \leftarrow \frac{a_k}{2}$
  $k = k + 1$
end while

Note that the algorithm proposed here uses a Euclidean rather than a proximal projection. Also, in contrast to the distributed subgradient algorithms described in [69], DOGD maintains an accumulated gradient variable in $\boldsymbol{z}_{\boldsymbol{i}}^k(t+1)$ which is updated using $\{\boldsymbol{z}_{\boldsymbol{j}}^k(t)\}$ as opposed to the primal feasible variables $\{\boldsymbol{w}_{\boldsymbol{j}}^k(t)\}$. Finally, key to achieving fast convergence is the exponential decrease of the learning rate after performing an exponentially increasing number of gradient steps together with a proper initialization of the learning rate. The next section provides theoretical guarantees on the performance of DOGD.

8.3 Analysis of DOGD

Our main convergence result, stated below, guarantees that the average regret decreases at a rate which is nearly linear.

Theorem 8.1. *Let $\boldsymbol{w}^*$ be the minimizer of $F(\boldsymbol{w})$. Under strong convexity and Lipschitz continuity, and using a doubly stochastic P with $\lambda_2(P)$ not depending on*

n, the sequence $\{\overline{\boldsymbol{w}}_{\boldsymbol{i}}^k\}$ produced by nodes running DOGD to minimize $F(\boldsymbol{w})$ obeys

$$F(\overline{\boldsymbol{w}}_{\boldsymbol{i}}^{k+1}) - F(\boldsymbol{w}^*) = O\left(\frac{\log\left(\sqrt{n}T\right)}{T}\right), \tag{8.5}$$

where $k = \lfloor \log_2(T/2+1) \rfloor$ is the number of rounds executed during a total of T gradient steps per node, and $\overline{\boldsymbol{w}}_{\boldsymbol{i}}^k$ is the running average maintained locally at each node.

Remark 1: We state the result for the case where λ_2 is constant. This is the case when G is, e.g., a complete graph or an expander graph [70]. For other graph topologies where λ_2 shrinks with n and consensus does not converge fast, the convergence rate dependence on n is going to be worse due to a factor $1-\sqrt{\lambda_2}$ in the denominator; see the proof of Theorem 8.1 below for the precise dependence on the spectral gap $1-\sqrt{\lambda_2}$.

Remark 2: The theorem characterizes performance of the online algorithm DOGD, where the data and cost functions $f(\boldsymbol{w}, \boldsymbol{x}_{\boldsymbol{i}}(t))$ are processed sequentially at each node in order to minimize an objective of the form

$$F(w) = \frac{1}{n}\sum_{i=1}^{n}\frac{1}{T}\sum_{t=1}^{T} f(\boldsymbol{w}, \boldsymbol{x}_{\boldsymbol{i}}(t)). \tag{8.6}$$

However, as pointed out in [77], if the entire dataset is available in advance, we can use the same scheme to do batch minimization by effectively setting $f(\boldsymbol{w}, \boldsymbol{x}_{\boldsymbol{i}}(t)) = f(\boldsymbol{w}, \boldsymbol{x}_{\boldsymbol{i}}(1))$, where $f(\boldsymbol{w}, \boldsymbol{x}_{\boldsymbol{i}}(1))$ is the objective function accounting for the entire dataset available to node i. Thus, the same result holds immediately for a batch version of DOGD.

The remainder of this section is devoted to the proof of Theorem 8.1. Our analysis follows arguments that can be found in [32, 40, 103] and references therein. We first state and prove some intermediate results.

8.3.1 Properties of Strongly Convex Functions

Recall the definition of σ-strong convexity (2.1). A direct consequence of this definition is that if $F(\boldsymbol{w})$ is σ-strongly convex then

$$F(\boldsymbol{w}) - F(\boldsymbol{w}^*) \geq \frac{\sigma}{2}\left\|\boldsymbol{w} - \boldsymbol{w}^*\right\|^2. \tag{8.7}$$

Strong convexity can be combined with the assumptions above to upper bound the difference $F(\boldsymbol{w}) - F(\boldsymbol{w}^*)$ for an arbitrary point $\boldsymbol{w} \in \mathcal{W}$.

Lemma 8.1. *Let $\boldsymbol{w}^*$ be the minimizer of $F(\boldsymbol{w})$. For all $\boldsymbol{w} \in \mathcal{W}$, we have $F(\boldsymbol{w}) - F(\boldsymbol{w}^*) \leq \frac{2L^2}{\sigma}$.*

Proof. For any subgradient $\nabla \boldsymbol{g} \in \partial F(\boldsymbol{w})$, by convexity we know that $F(\boldsymbol{w}) - F(\boldsymbol{w}^*) \leq \langle \nabla \boldsymbol{g}, \boldsymbol{w} - \boldsymbol{w}^* \rangle$. It follows from L-Lipschitz continuity that $F(\boldsymbol{w}) - F(\boldsymbol{w}^*) \leq L \|\boldsymbol{w} - \boldsymbol{w}^*\|$. Furthermore, from strong convexity we obtain $\frac{\sigma}{2} \|\boldsymbol{w} - \boldsymbol{w}^*\|^2 \leq L \|\boldsymbol{w} - \boldsymbol{w}^*\|$ or $\|\boldsymbol{w} - \boldsymbol{w}^*\| \leq \frac{2L}{\sigma}$. As a result, $F(\boldsymbol{w}) - F(\boldsymbol{w}^*) \leq \frac{2L^2}{\sigma}$. □

8.3.2 The Lazy Projection Algorithm

The analysis of DOGD below involves showing that the average state, $\frac{1}{n} \sum_{i=1}^{n} \boldsymbol{w}_{\boldsymbol{i}}^k(t)$, evolves according to the so-called (single processor) *lazy projection* algorithm [103], which we discuss next. The lazy projection algorithm is an online convex optimization scheme for the serial problem (8.1). A single processor sequentially chooses a new variable $\boldsymbol{w}(t)$ and receives a subgradient $\boldsymbol{g}(t)$ of $f(\boldsymbol{w}(t), \boldsymbol{x}(t))$. The algorithm chooses $\boldsymbol{w}(t+1)$ by repeating the steps

$$\boldsymbol{z}(t+1) = \boldsymbol{z}(t) - a\boldsymbol{g}(t) \tag{8.8}$$

$$\boldsymbol{w}(t+1) = \Pi_{\mathcal{W}} \left[\boldsymbol{z}(t+1) \right]. \tag{8.9}$$

By unwrapping the recursive form of (8.8), we get

$$\boldsymbol{z}(t+1) = -a \sum_{s=1}^{t} \boldsymbol{g}(t) + \boldsymbol{z}(1). \tag{8.10}$$

The following is a typical result for subgradient descent-style algorithms, and is useful towards eventually characterizing how the regret accumulates. Its proof can be found in the appendix of the extended version of [103].

Theorem 8.2 (Zinkevich [103])**.** *Let $\boldsymbol{w}(1) \in \mathcal{W}$, let $a > 0$, and set $\boldsymbol{z}(1) = \boldsymbol{w}(1)$. After T rounds of the serial lazy projection algorithm* (8.8)–(8.9)*, we have*

$$\sum_{t=1}^{T} \langle \boldsymbol{g}(t), \boldsymbol{w}(t) - \boldsymbol{w}^* \rangle \leq \frac{\|\boldsymbol{w}(1) - \boldsymbol{w}^*\|^2}{2a} + \frac{TaL^2}{2}. \tag{8.11}$$

Theorem 8.2 immediately yields the same bound for the regret as lazy projection [103].

8.3.3 Evolution of Network-Average Quantities in DOGD

We turn our attention to Algorithm 2. A standard approach to studying convergence of distributed optimization algorithms, such as DOGD, is to keep track of the discrepancy between every node's state and an average state sequence defined as

$$\overline{\boldsymbol{z}}^k(t) = \frac{1}{n}\sum_{i=1}^{n} \boldsymbol{z}_{\boldsymbol{i}}^k(t) \quad \text{and} \quad \overline{\boldsymbol{y}}^k(t) = \Pi_{\mathcal{W}}\left[\overline{\boldsymbol{z}}^k(t)\right]. \tag{8.12}$$

We have encountered this concept before. Observe that $\overline{\boldsymbol{z}}^k(t)$ evolves recursively as follows:

$$\begin{aligned}
\overline{\boldsymbol{z}}^k(t+1) =& \frac{1}{n}\sum_{i=1}^{n} \boldsymbol{z}_{\boldsymbol{i}}^k(t+1) && (8.13)\\
=& \frac{1}{n}\sum_{i=1}^{n}\left[\sum_{j=1}^{n} p_{ij}\boldsymbol{z}_{\boldsymbol{j}}^k(t) - a_k \boldsymbol{g}_{\boldsymbol{i}}(t)\right] && (8.14)\\
=& \frac{1}{n}\sum_{j=1}^{n} \boldsymbol{z}_{\boldsymbol{j}}^k(t)\sum_{i=1}^{n} p_{ij} - \frac{a_k}{n}\sum_{i=1}^{n}\boldsymbol{g}_{\boldsymbol{i}}(t) && (8.15)\\
=& \overline{\boldsymbol{z}}(t) - \frac{a_k}{n}\sum_{i=1}^{n}\boldsymbol{g}_{\boldsymbol{i}}(t) && (8.16)\\
=& -a_k\sum_{s=1}^{t}\frac{1}{n}\sum_{i=1}^{n}\boldsymbol{g}_{\boldsymbol{i}}(s) + \frac{1}{n}\sum_{i=1}^{n}\boldsymbol{z}_{\boldsymbol{i}}^k(1) && (8.17)
\end{aligned}$$

where equation (8.16) holds since P is doubly stochastic. Keep in mind that due to the nature of the algorithm we need to also keep track of the initial condition $\boldsymbol{z}_{\boldsymbol{i}}(1)$. Notice also (cf. eqn. (8.10)) that the states $\{\overline{\boldsymbol{z}}^k(t), \overline{\boldsymbol{y}}^k(t)\}$ evolve according to the lazy projection algorithm with gradients $\overline{\boldsymbol{g}}(t) = \frac{1}{n}\sum_{i=1}^{n}\boldsymbol{g}_{\boldsymbol{i}}(t)$ and learning rate a_k. In the sequel, we will also use an analytic expression for $\boldsymbol{z}_{\boldsymbol{i}}^k(t)$ derived by back substituting in its recursive update equation. After some algebraic manipulation,

we obtain

$$\begin{aligned}\boldsymbol{z}_{\boldsymbol{i}}^{k}(t) =& - a_k \sum_{s=1}^{t-1} \sum_{j=1}^{n} \left[P^{t-s+1}\right]_{ij} \boldsymbol{g}_{\boldsymbol{j}}(s-1) - a_k \boldsymbol{g}_{\boldsymbol{i}}(t-1) \\ &+ \sum_{j=1}^{n} [P^t]_{ij} \boldsymbol{z}_{\boldsymbol{j}}^{k}(1), \end{aligned} \tag{8.18}$$

and since the projection in non-expansive and $\boldsymbol{z}_{\boldsymbol{i}}^{1}(1) = 0, \forall i$,

$$\begin{aligned}\left\|\boldsymbol{z}_{\boldsymbol{i}}^{k+1}(1)\right\| = \left\|\boldsymbol{w}_{\boldsymbol{i}}^{k+1}(1)\right\| = \left\|\boldsymbol{w}_{\boldsymbol{i}}^{k}(T_k)\right\| &= \left\|\Pi_{\mathcal{W}}\left[\boldsymbol{z}_{\boldsymbol{i}}^{k}(T_k)\right]\right\| \\ &\leq \left\|\boldsymbol{z}_{\boldsymbol{i}}^{k}(T_k)\right\| \\ &\leq \left\| -a_k \sum_{t=1}^{T_k-1} \sum_{i=1}^{n} \left[P^{T_k-s+1}\right]_{ij} \boldsymbol{g}_{\boldsymbol{i}}(s-1) \right\| \\ &\quad + \left\|-a_k \boldsymbol{g}_{\boldsymbol{i}}(T_k-1)\right\| + \sum_{j=1}^{n} \left[P^{T_k}\right]_{ij} \left\|\boldsymbol{z}_{\boldsymbol{j}}^{k}(1)\right\| \\ &\leq a_k T_k L + \sum_{j=1}^{n} \left[P^{T_k}\right]_{ij} \left\|\boldsymbol{z}_{\boldsymbol{j}}^{k}(1)\right\| \leq \cdots \\ &\leq L \sum_{s=1}^{k} a_s T_s. \end{aligned} \tag{8.19}$$

8.3.4 Analysis of One Round of DOGD

Next, we bound the regret accumulated at round k of DOGD (lines 5–12 of Algorithm 2) during which the learning rate remains fixed at a_k. From convexity,

Lipschitz continuity, and the triangle inequality

$$
\begin{aligned}
&\sum_{t=1}^{T_k}[F(\boldsymbol{w}_{\boldsymbol{i}}^k(t)) - F(\boldsymbol{w}^*)] \\
&= \sum_{t=1}^{T_k}\left[F\left(\overline{\boldsymbol{y}}^k(t)\right) - F(\boldsymbol{w}^*) + F(\boldsymbol{w}_{\boldsymbol{i}}^k(t)) - F\left(\overline{\boldsymbol{y}}^k(t)\right)\right] \\
&\leq \sum_{t=1}^{T_k}\left[F(\overline{\boldsymbol{y}}^k(t)) - F(\boldsymbol{w}^*) + L\left\|\boldsymbol{w}_{\boldsymbol{i}}^k(t) - \overline{\boldsymbol{y}}^k(t)\right\|\right] \\
&\leq \sum_{t=1}^{T_k}\frac{1}{n}\sum_{i=1}^{n}\left[f(\boldsymbol{w}_{\boldsymbol{i}}^k(t), \boldsymbol{x}_{\boldsymbol{i}}^k(t)) - f(\boldsymbol{w}^*, \boldsymbol{x}_{\boldsymbol{i}}^k(t))\right] \\
&\quad + \sum_{t=1}^{T_k}\frac{1}{n}\sum_{i=1}^{n}\left[f(\overline{\boldsymbol{y}}^k(t), \boldsymbol{x}_{\boldsymbol{i}}^k(t)) - f(\boldsymbol{w}_{\boldsymbol{i}}^k(t), \boldsymbol{x}_{\boldsymbol{i}}^k(t))\right] \\
&\quad + \sum_{t=1}^{T_k} L\left\|\boldsymbol{w}_{\boldsymbol{i}}^k(t) - \overline{\boldsymbol{y}}^k(t)\right\| \\
&\leq \underbrace{\sum_{t=1}^{T_k}\frac{1}{n}\sum_{i=1}^{n}\langle \boldsymbol{g}_{\boldsymbol{i}}(t), \boldsymbol{w}_{\boldsymbol{i}}^k(t) - \boldsymbol{w}^*\rangle}_{A_1} \\
&\quad + \sum_{t=1}^{T_k}\frac{1}{n}\sum_{i=1}^{n} L\left\|\overline{\boldsymbol{y}}^k(t) - \boldsymbol{w}_{\boldsymbol{i}}^k(t)\right\| \\
&\quad + \sum_{t=1}^{T_k} L\left\|\boldsymbol{w}_{\boldsymbol{i}}^k(t) - \overline{\boldsymbol{y}}^k(t)\right\|. \qquad (8.20)
\end{aligned}
$$

For the first summand we add and subtract $\overline{\boldsymbol{y}}^k(t)$ to obtain

$$
\begin{aligned}
A_1 &= \sum_{t=1}^{T_k}\frac{1}{n}\sum_{i=1}^{n}\langle \boldsymbol{g}_{\boldsymbol{i}}(t), \boldsymbol{w}_{\boldsymbol{i}}^k(t) - \boldsymbol{w}^*\rangle \\
&\leq \sum_{t=1}^{T_k}\frac{1}{n}\sum_{i=1}^{n}\langle \boldsymbol{g}_{\boldsymbol{i}}(t), \overline{\boldsymbol{y}}^k(t) - \boldsymbol{w}^* + \boldsymbol{w}_{\boldsymbol{i}}^k(t) - \overline{\boldsymbol{y}}^k(t)\rangle \\
&\leq \underbrace{\sum_{t=1}^{T_k}\frac{1}{n}\sum_{i=1}^{n}\langle \boldsymbol{g}_{\boldsymbol{i}}(t), \overline{\boldsymbol{y}}^k(t) - \boldsymbol{w}^*\rangle}_{A_2} \\
&\quad + \sum_{t=1}^{T_k}\frac{1}{n}\sum_{i=1}^{n} L\left\|\boldsymbol{w}_{\boldsymbol{i}}^k(t) - \overline{\boldsymbol{y}}^k(t)\right\|. \qquad (8.21)
\end{aligned}
$$

Invoking Theorem 8.2 for the sequences $\{\overline{\boldsymbol{y}}^k(t)\}$ and $\{\overline{\boldsymbol{z}}^k(t)\}$ and noticing that $\|\overline{\boldsymbol{g}}(t)\| \leq L^2$,

$$A_2 = \sum_{t=1}^{T_k} \frac{1}{n} \sum_{i=1}^{n} \langle \boldsymbol{g_i}(t), \overline{\boldsymbol{y}}^k(t) - w^* \rangle \tag{8.22}$$

$$= \sum_{t=1}^{T_k} \left\langle \frac{1}{n} \sum_{i=1}^{n} \boldsymbol{g_i}(t), \Pi_{\mathcal{W}} \left[\overline{\boldsymbol{z}}^k(t) \right] - \boldsymbol{w^*} \right\rangle \tag{8.23}$$

$$= \sum_{t=1}^{T_k} \left\langle \overline{\boldsymbol{g}}(t), \Pi_{\mathcal{W}} \left[\overline{\boldsymbol{z}}^k(t) \right] - \boldsymbol{w^*} \right\rangle \tag{8.24}$$

$$\leq \frac{\left\| \overline{\boldsymbol{y}}^k(1) - \boldsymbol{w^*} \right\|^2}{2a_k} + \frac{T_k a_k L^2}{2}. \tag{8.25}$$

Collecting the bounds (8.20) and (8.22), so far we have shown that

$$\begin{aligned} \sum_{t=1}^{T_k} [F(\boldsymbol{w}_{\boldsymbol{i}}^k(t)) - F(\boldsymbol{w^*})] \leq & \frac{\left\| \overline{\boldsymbol{y}}^k(1) - \boldsymbol{w^*} \right\|^2}{2a_k} + \frac{T_k a_k L^2}{2} \\ & + \sum_{t=1}^{T_k} \frac{2}{n} \sum_{i=1}^{n} L \left\| \boldsymbol{w}_{\boldsymbol{i}}^k(t) - \overline{\boldsymbol{y}}^k(t) \right\| \\ & + \sum_{t=1}^{T_k} L \left\| \boldsymbol{w}_{\boldsymbol{i}}^k(t) - \overline{\boldsymbol{y}}^k(t) \right\|, \end{aligned} \tag{8.26}$$

and since the projection operator is non-expansive, we have

$$\begin{aligned} \sum_{t=1}^{T_k} [F(\boldsymbol{w}_{\boldsymbol{i}}^k(t)) - F(\boldsymbol{w^*})] \leq & \frac{\left\| \overline{\boldsymbol{y}}^k(1) - \boldsymbol{w^*} \right\|^2}{2a_k} + \frac{T_k a_k L^2}{2} \\ & + \sum_{t=1}^{T_k} \frac{2}{n} \sum_{i=1}^{n} L \left\| \boldsymbol{z}_{\boldsymbol{i}}^k(t) - \overline{\boldsymbol{z}}^k(t) \right\| \\ & + \sum_{t=1}^{T_k} L \left\| \boldsymbol{z}_{\boldsymbol{i}}^k(t) - \overline{\boldsymbol{z}}^k(t) \right\|. \end{aligned} \tag{8.27}$$

The first two terms are standard for subgradient algorithms using a constant step size. The last two terms depend on the error between each node's iterate $\boldsymbol{z}_{\boldsymbol{i}}^k(t)$ and the network-wide average $\overline{\boldsymbol{z}}^k(t)$, which we bound next.

8.3.5 Bounding the Network Error

What remains is to bound the term $\left\| \boldsymbol{z}_{\boldsymbol{i}}^k(t) - \overline{\boldsymbol{z}}^k(t) \right\|$ which describes an error induced by the network since the different nodes do not agree on the direction towards the optimum. By recalling that P is doubly stochastic and manipulating the

recursive expressions (8.18) and (8.17) for $\boldsymbol{z_i}(t)$ and $\overline{\boldsymbol{z}}^k(t)$ using arguments similar to those in [32, 90], we obtain the bound,

$$\begin{aligned}\left\|\boldsymbol{z}_{\boldsymbol{i}}^k(t)-\overline{\boldsymbol{z}}^k(t)\right\| \leq & a_k L \sum_{s=1}^{t-1}\sum_{j=1}^{n}\left|\frac{1}{n}\mathbf{1}^T-\left[P^{t-s-1}\right]_{ij}\right|_1 \\ & +2a_kL+\sum_{j=1}^{n}\left|\frac{1}{n}-[P^t]_{ij}\right|\left\|\boldsymbol{z}_{\boldsymbol{j}}^k(1)\right\| \\ = & a_k L \sum_{s=1}^{t-1}\left\|\frac{1}{n}\mathbf{1}^T-\left[P^{t-s-1}\right]_{i,:}\right\|_1 \\ & +2a_kL+\sum_{j=1}^{n}\left|\frac{1}{n}-[P^t]_{ij}\right|\left\|\boldsymbol{z}_{\boldsymbol{j}}^k(1)\right\|. \end{aligned} \tag{8.28}$$

We have already seen the bound for the ℓ_1 norm (see Appendix C for $h=1$), so recalling (7.21), (7.20) and using (8.19) we arrive at

$$\left\|\boldsymbol{z}_{\boldsymbol{i}}^k(t)-\overline{\boldsymbol{z}}^k(t)\right\| \leq 2a_kL\frac{\log\left(T_k\sqrt{n}\right)}{1-\sqrt{\lambda_2}}+3a_kL+\frac{L\sum_{s=1}^{k-1}a_sT_s}{T_k} \tag{8.29}$$

where λ_2 is the second largest eigenvalue of P. Using this bound in equation (8.27), along with the fact that $F(\boldsymbol{w})$ is convex, we conclude that

$$\begin{aligned} F(\overline{\boldsymbol{w}}_{\boldsymbol{i}}^{k+1})-F(\boldsymbol{w}^*) = & F\left(\frac{1}{T_k}\sum_{t=1}^{T_k}\boldsymbol{w}_{\boldsymbol{i}}^k(t)\right)-F(\boldsymbol{w}^*) \\ \leq & \frac{1}{T_k}\sum_{t=1}^{T_k}\left[F(\boldsymbol{w}_{\boldsymbol{i}}^k(t))-F(\boldsymbol{w}^*)\right] \\ \leq & \frac{\left\|\overline{\boldsymbol{y}}^k(1)-\boldsymbol{w}^*\right\|^2}{2a_kT_k}+\frac{a_kL^2}{2}+L^2a_k\left[6\frac{\log\left(T_k\sqrt{n}\right)}{1-\sqrt{\lambda_2}}+9\right] \\ & +\frac{3L^2\sum_{s=1}^{k-1}a_sT_s}{T_k}, \end{aligned} \tag{8.30}$$

where $\overline{\boldsymbol{y}}^k(1)=\Pi_{\mathcal{W}}\left[\frac{1}{n}\sum_{i=1}^{n}\boldsymbol{z}_{\boldsymbol{i}}^k(1)\right]$.

8.3.6 Analysis of DOGD over Multiple Rounds

As our last intermediate step, we must control the learning rate and update of T_k from round-to-round to ensure linear convergence of the error. From strong convexity of F we have

$$\left\|\overline{\boldsymbol{y}}^k(1)-\boldsymbol{w}^*\right\|^2 \leq 2\frac{F(\overline{\boldsymbol{y}}^k(1))-F(\boldsymbol{w}^*)}{\sigma} \tag{8.31}$$

and thus

$$F(\overline{\boldsymbol{w}}_{\boldsymbol{i}}^{k+1}) - F(\boldsymbol{w}^*) \leq \frac{F(\overline{\boldsymbol{y}}^k(1)) - F(\boldsymbol{w}^*)}{\sigma a_k T_k} + \frac{L^2 a_k}{2}\left[12\frac{\log(T_k\sqrt{n})}{1-\sqrt{\lambda_2}} + 19\right] + \frac{3L^2\sum_{s=1}^{k-1} a_s T_s}{T_k}. \tag{8.32}$$

Now, from Theorem 3 in [103] which is a direct consequence of Theorem 8.2 for the average sequence $\overline{\boldsymbol{y}}$ viewed as a single processor lazy projection algorithm, we have that after executing T_{k-1} gradient steps in round $k-1$,

$$F(\overline{\boldsymbol{y}}^k(1)) - F(\boldsymbol{w}^*) \leq \frac{\left\|\overline{\boldsymbol{y}}^{k-1}(1) - \boldsymbol{w}^*\right\|^2}{2a_{k-1}T_{k-1}} + \frac{a_{k-1}L^2}{2} \tag{8.33}$$

and by repeatedly using strong convexity and Theorem 8.2 we see that

$$\begin{aligned} F(\overline{\boldsymbol{y}}^k(1)) - F(\boldsymbol{w}^*) \leq & \frac{F(\overline{\boldsymbol{y}}^{k-1}(1)) - F(\boldsymbol{w}^*)}{\sigma a_{k-1}T_{k-1}} + \frac{a_{k-1}L^2}{2} \\ \leq \cdots \leq & \frac{F(\overline{\boldsymbol{y}}^1(1)) - F(\boldsymbol{w}^*)}{\prod_{j=0}^{k-1}(\sigma a_{k-j}T_{k-j})} + \sum_{j=1}^{k-1}\frac{a_{k-j}L^2}{2\prod_{s=1}^{j-1}(\sigma a_{k-s}T_{k-s})}. \end{aligned} \tag{8.34}$$

Now, let us fix positive integers b and c, and suppose we use the following rules to determine the step size and number of updates performed within each round:

$$a_k = \frac{a_{k-1}}{b} = \cdots = \frac{a_1}{b^{k-1}} \tag{8.35}$$

$$T_k = cT_{k-1} = \cdots = c^{k-1}T_1. \tag{8.36}$$

Combining (8.34) with (8.32) and invoking Lemma 8.1, we have

$$\begin{aligned} F(\overline{\boldsymbol{w}}_{\boldsymbol{i}}^{k+1}) - F(\boldsymbol{w}^*) \leq & \frac{2L^2}{\sigma\prod_{j=0}^{k-1}\left(\sigma a_1 T_1\left(\frac{c}{b}\right)^{k-j-1}\right)} \\ & + \sum_{j=1}^{k-1}\frac{a_1L^2}{2b^{k-j-1}\prod_{s=0}^{j-1}\left(\sigma a_1 T_1\left(\frac{c}{b}\right)^{k-s-1}\right)} \\ & + \frac{L^2 a_1}{2b^{k-1}}\left[12\frac{\log(T_1c^{k-1}\sqrt{n})}{1-\sqrt{\lambda_2}} + 19\right] \\ & + \frac{3L^2\sum_{s=1}^{k-1} a_1 T_1\left(\frac{c}{b}\right)^{s-1}}{T_1c^{k-1}}. \end{aligned} \tag{8.37}$$

To ensure convergence to zero, we need $c \geq b$ and $\sigma a_1 T_1 > 1$ or $a_1 > \frac{1}{T_1 \sigma}$. Given these restrictions, let us make the choices

$$a_1 = 1, \quad T_1 = \left\lceil \frac{2}{\sigma} \right\rceil, \quad c = b = 2. \tag{8.38}$$

To simplify the exposition, let us assume that $T_1 = \frac{2}{\sigma}$ is an integer. Using the selected values, we obtain

$$\begin{aligned}
F(\overline{\boldsymbol{w}}_{\boldsymbol{i}}^{k+1}) - F(\boldsymbol{w}^*) \leq & \frac{2L^2}{\sigma \prod_{j=0}^{k-1} \left(2\left(\frac{2}{2}\right)^{k-j-1}\right)} \\
& + \sum_{j=1}^{k-1} \frac{L^2}{\cdot 2 \cdot 2^{k-j-1} \prod_{s=0}^{j-1} \left(2\left(\frac{2}{2}\right)^{k-s-1}\right)} \\
& + \frac{L^2}{2 \cdot 2^{k-1}} \left[12 \frac{\log\left(\frac{2}{\sigma} \cdot 2^{k-1}\sqrt{n}\right)}{1 - \sqrt{\lambda_2}} + 19 \right] \\
& + \frac{3L^2 \sum_{s=1}^{k-1} \left(\frac{2}{2}\right)^{s-1}}{2^{k-1}} \qquad (8.39) \\
\leq & \frac{2L^2}{\sigma 2^k} + \sum_{j=1}^{k-1} \frac{L^2}{2^{k-j} 2^j} \\
& + \frac{L^2}{2^k} \left[12 \frac{\log\left(\frac{2^k\sqrt{n}}{\sigma}\right)}{1 - \sqrt{\lambda_2}} + 19 \right] + \frac{3L^2(k-1)}{2^{k-1}} \qquad (8.40) \\
\leq & \frac{2L^2}{\sigma 2^k} + \frac{L^2(k-1)}{2^k} \\
& + \frac{L^2}{2^k} \left[12 \frac{\log\left(\frac{2^k\sqrt{n}}{\sigma}\right)}{1 - \sqrt{\lambda_2}} + 19 \right] + \frac{6L^2(k-1)}{2^k}. \qquad (8.41)
\end{aligned}$$

Finally, we have all we need to complete the analysis of Algorithm 2.

8.3.7 Proof of Theorem 8.1

Suppose we run Algorithm 2 for T total steps at each node. This allows for $\tilde{k}$ rounds, where $\tilde{k}$ is determined by solving

$$\sum_{i=1}^{\tilde{k}} T_i \leq T \iff \sum_{i=1}^{\tilde{k}} 2 \cdot 2^i \leq T \iff \tilde{k} \leq \log_2\left(\frac{T}{2} + 1\right). \tag{8.42}$$

Using this value for k we see that

$$\begin{aligned}
F(\overline{\boldsymbol{w}}_{\boldsymbol{i}}^{\tilde{k}+1}) - F(\boldsymbol{w}^*) \leq & \frac{L^2}{\sigma} 2^{\tilde{k}} + \frac{L^2(\tilde{k}-1)}{2^{\tilde{k}}} \\
& + \frac{L^2}{2^{\tilde{k}}} \left[12 \frac{\log\left(\frac{2^{\tilde{k}}\sqrt{n}}{\sigma}\right)}{1-\sqrt{\lambda_2}} + 19 \right] + \frac{6L^2(\tilde{k}-1)}{2^{\tilde{k}}} \\
\leq & \frac{L^2}{\sigma\left(\frac{T}{2}+1\right)} + \frac{L^2(\log_2\left(\frac{T}{2}+1\right)-1)}{\left(\frac{T}{2}+1\right)} \\
& + \frac{L^2}{\left(\frac{T}{2}+1\right)} \left[12 \frac{\log\left(\frac{\left(\frac{T}{2}+1\right)\sqrt{n}}{\sigma}\right)}{1-\sqrt{\lambda_2}} + 19 \right] \\
& + \frac{6L^2(\left(\frac{T}{2}+1\right)-1)}{\left(\frac{T}{2}+1\right)} \\
= & O\left(\frac{\log\left(\sqrt{n}T\right)}{T(1-\sqrt{\lambda_2})} \right) = O\left(\frac{\log(\sqrt{n}T)}{T} \right),
\end{aligned} \tag{8.43}$$

when λ_2 is constant and does not scale with n. This concludes the proof of Theorem 8.1.

8.4 Extension to Stochastic Optimization

The proof in the previous section can easily be extended to the stochastic case where the data and thus the cost functions $f(\boldsymbol{w}, \boldsymbol{x_i}(t))$ are generated i.i.d. from some unknown distribution. In that case, at each iteration the gradient $\hat{\boldsymbol{g}}(t)$ at each node is just an estimate of the true gradient in the sense that $\mathbb{E}[\hat{\boldsymbol{g}}(t)] = \boldsymbol{g}(t)$. We assume however that noisy gradients have bounded variance i.e., $\mathbb{E}[\|\hat{\boldsymbol{g}}_{\boldsymbol{i}}(t)\|^2] \leq L^2$. In this setting, instead of equation (8.22), we have

$$\begin{aligned}
A_2 = & \sum_{t=1}^{T_k} \frac{1}{n} \sum_{i=1}^{n} \langle \boldsymbol{g_i}(t), \overline{\boldsymbol{y}}^k(t) - \boldsymbol{w}^* \rangle \\
= & \sum_{t=1}^{T_k} \langle \frac{1}{n} \sum_{i=1}^{n} \hat{\boldsymbol{g}}_{\boldsymbol{i}}(t), \overline{\boldsymbol{y}}^k(t) - \boldsymbol{w}^* \rangle \\
& + \sum_{t=1}^{T_k} \frac{1}{n} \sum_{i=1}^{n} \langle \boldsymbol{g_i}(t) - \hat{\boldsymbol{g}}_{\boldsymbol{i}}(t), \overline{\boldsymbol{y}}^k(t) - \boldsymbol{w}^* \rangle.
\end{aligned} \tag{8.44}$$

Now notice that Theorem 8.2 holds for noisy gradients $\hat{\boldsymbol{g}}(t)$ as well. Moreover, we have $\mathbb{E}[\|\hat{\boldsymbol{g}}_{\boldsymbol{i}}(t)\|] \leq L$, and by Hölder's inequality $\mathbb{E}[\|\hat{\boldsymbol{g}}_{\boldsymbol{i}}(t)\| \, \|\hat{\boldsymbol{g}}_{\boldsymbol{j}}(t)\|] \leq L^2$. This yields $\mathbb{E}\left[\left\| \frac{1}{n} \sum_{i=1}^{n} \hat{\boldsymbol{g}}_{\boldsymbol{i}}(t) \right\|^2 \right] \leq L^2$. Thus, invoking Theorem 8.2, if the new data and thus

the subgradients are independent of the past, and since $\mathbb{E}[\hat{\boldsymbol{g}}_{\boldsymbol{i}}(t)] = \boldsymbol{g}_{\boldsymbol{i}}(t)$, we have

$$\begin{aligned}\mathbb{E}[A_2] \leq & \frac{\left\|\overline{\boldsymbol{y}}^k(1) - \boldsymbol{w}^*\right\|^2}{2a_k} + \frac{T_k a_k L^2}{2} \\ & + \mathbb{E}[\sum_{t=1}^{T_k} \frac{1}{n} \sum_{i=1}^{n} \langle \boldsymbol{g}_{\boldsymbol{i}}(t) - \hat{\boldsymbol{g}}_{\boldsymbol{i}}(t), \overline{\boldsymbol{y}}^k(t) - \boldsymbol{w}^* \rangle] \\ = & \frac{\left\|\overline{\boldsymbol{y}}^k(1) - \boldsymbol{w}^*\right\|^2}{2a_k} + \frac{T_k a_k L^2}{2}. \end{aligned} \tag{8.45}$$

Furthermore, the network error bound holds in expectation as well, i.e.,

$$\begin{aligned}\mathbb{E}\left[\left\|\overline{\boldsymbol{y}}^k(t) - \boldsymbol{w}_{\boldsymbol{i}}^k(t)\right\|\right] \leq & \mathbb{E}\left[\left\|\overline{\boldsymbol{z}}^k(t) - \boldsymbol{z}_{\boldsymbol{i}}^k(t)\right\|\right] \\ \leq & 2a_k L \frac{\log\left(T_k\sqrt{n}\right)}{1-\sqrt{\lambda_2}} + 3a_k L + \frac{L\sum_{s=1}^{k-1} a_s T_s}{T_k}. \end{aligned} \tag{8.46}$$

Collecting all these observations we have shown that, in expectation,

$$\begin{aligned}\mathbb{E}\left[F(\overline{\boldsymbol{w}}_{\boldsymbol{i}}^{k+1}) - F(\boldsymbol{w}^*)\right] \leq & \frac{\left\|\overline{\boldsymbol{y}}^k(1) - \boldsymbol{w}^*\right\|^2}{2a_k T_k} + \frac{a_k L^2}{2} \\ & + L^2 a_k \left[6\frac{\log\left(T_k\sqrt{n}\right)}{1-\sqrt{\lambda_2}} + 9\right] + \frac{3L^2\sum_{s=1}^{k-1} a_s T_s}{T_k}, \end{aligned} \tag{8.47}$$

which, after using the update rules for a_k and T_k, is exactly the same expression as (8.37) so the final convergence rate result is again $O\left(\frac{\log(\sqrt{n}T)}{T(1-\sqrt{\lambda_2})}\right)$. We note however that there may still be room for improvement in the distributed stochastic optimization setting since [40] describes a single-processor algorithm that converges at a rate $O\left(\frac{1}{T}\right)$.

8.5 Concluding Remarks

In this chapter we have proposed and analyzed a novel distributed optimization algorithm which we call Distributed Online Gradient Descent (DOGD). Our analysis shows that DOGD converges at a rate $O(\frac{\log(\sqrt{n}T)}{T})$ when solving online, stochastic or batch constrained convex optimization problems if the objective function is strongly convex. This rate is optimal in the number of iterations for the online and batch setting and slower than a serial algorithm only by a logarithmic factor in the stochastic optimization setting. The open question that remains is

whether a rate of $O\left(\frac{1}{nT}\right)$ for strongly convex functions is attainable by a consensus based distributed algorithm.

CHAPTER 9
Summary and Open Questions for Future Work

This work is motivated by the plethora of large scale optimization problems that arise more and more frequently in various domains from machine learning to computational biology and finance. Many of these problems present extreme computational challenges due to the sheer volume of data that needs to be processed. Furthermore there exist situations where the data itself is distributed and collecting it in a single location might be costly or violating data privacy. For all those reasons distributed computing is becoming the method of choice for solving such problems. Within the growing literature of different methods for parallel and distributed computation, we focus on the general class of consensus-based distributed optimization algorithms which are suitable for computer clusters and distributed computing over general ad-hoc networks.

The starting point for this work is the fundamental realization that the difference between a network of n slow computers and an n-times faster single processor is indeed the communication network. Any information exchange that takes place over the network is in general orders of magnitude slower than the local processing speed and especially in high-dimensional problems this communication cost can no longer be ignored. This simple observation brings up the important role of the network. In an attempt to understand this role, this thesis asks questions and contributes answers in four different directions:

1. How can the communication cost be modelled and how does it affect the scalability of different algorithms? Our results suggest that it is the tradeoff between communication and computation on a given system for a given problem that is the defining factor. It is also shown that this tradeoff can be exploited to achieve parallelization at the task level when solving multiple optimization problems on the same network.

2. What are the properties of practical distributed optimization algorithms that can work well in real network conditions? We identify three key properties – averaging, one-directional communication and asynchronism – which as we show are important to develop practical algorithms with guaranteed performance even in the presence of communication delays and slow nodes in the network.
3. What is the effect of communication delays? This work proposed models for both fixed and time varying-communication delays. Those models are used to analyze the effects of delays on convergence rates of different distributed optimization algorithms.
4. What are the best achievable convergence rates of consensus-based distributed algorithms? For this question the role of the network enters just through the network topology and the results are valid even if the actual cost of communication was free. Our work bridges some of the gaps between the performance of the best possible serial algorithms and their distributed counterparts.

For all of the work presented in the thesis, a conscious attempt is made to provide the necessary intuition, a theoretical analysis and explanation and experiments in simulation or real clusters that match, illustrate and validate the intuition and theoretical findings. However, this work in many ways only scratches the surface and in many ways opens the door for many more and exciting questions that need answers. Many of those questions have been mentioned at the end of each chapter. Here we summarize a full list of possible future directions:

1. There is a growing literature in both consensus-based distributed algorithms and also in parallel and hierarchical approaches that in many cases rely on communication tools such as MapReduce which may lead to simpler algorithms at the expense of restricting what an algorithm can do. The former class of algorithms provides full flexibility to the programmer and tends to lend itself to a more elegant theoretical analysis while the latter seem to currently exhibit the most promising results in practice. The experiments on a small scale (15 to 64 processors) in this thesis (see e.g., Section 5.6.3) seem to suggest

that asynchronous consensus-based algorithms could be a good alternative in practice and this is in agreement with the author's personal correspondence with practitioners. However, a large scale comparison of all state-of-the art methods on the same problem using hundreds of machines is still missing from the literature. Since this line of research is not purely theoretical but aims at solving large scale real life problems, a fair comparison under real conditions is becoming more and more necessary.

2. There is a secondary effect related to the network that has not been investigated in this thesis. We have focused on the optimization of separable problems which fortunately fit naturally in the framework of empirical loss minimization. No assumptions were made about the individual component functions $f_i(\boldsymbol{w})$ that reside at each node. However, the similarity of those functions should somehow be important. For example if the minima of the components are not located in the same place, without communication it is impossible to converge to the true solution. However, on the other extreme, in a stochastic optimization setting, if the data streams at the processors are i.i.d., then communication is not necessary as each node individually will eventually converge to the right solution. This naturally raises a question that merits some investigation in the future: Can we regulate the amount of communication based on the similarity or dissimilarity of the objective's components?
3. Related to the previous point, in stochastic optimization scenarios each node can individually reach the right solution given enough time. Thus, the purpose of communication is to reduce the time it takes to reach the solution. In Chapter 3 we saw that for a fixed dataset, using more nodes reduced the computation cost per node and this computational benefit can be trade off for the extra communication required when adding more processors to the network. It is interesting to ask if the same conclusions can be drawn for the stochastic setting. Notice that in that case, computation is the same per node (one point gradient) regardless of the network size. However, with more nodes, the network sees collectively more data in the same amount of

time. Previous work has exploited this observation in combination with mini-batches for variance reduction and perfect averaging to describe an algorithm that achieves the optimal distributed convergence rate $O\left(\frac{1}{\sqrt{nT}}\right)$, [28]. It is still an open question whether the same rate is achievable by a consensus-based algorithm.

4. Most of the research on distributed algorithms has focused on first-order methods that only use gradient information. This is justified by the fact that first order methods have a consistent performance even for high dimensional problems while they remain very simple. Simplicity is particularly important to have any hope of transitioning to a distributed algorithm which is amenable to theoretical analysis. However, despite the generality and minimal assumptions (convexity and Lipschitz continuity) required by first order methods, many problems of interest are also smooth and thus second order information is also available. In the serial algorithm literature, Newton methods [62] and limited memory LBFGS methods [19] are known to convergence in very few iterations. Furthermore, second order information can also be estimated and be helpful in the stochastic setting [18]. It would be important to see if any of those second order methods can be adapted to distributed optimization settings. That would require understanding what kind of information needs to be shared among nodes in an attempt to reach consensus. It is unclear if the second order Hessian or pseudo-Hessian should remain a locally computed quantity of if the extra communication of transmitting can yield computational speedups and reduction of number of iterations.
5. Finally, most of the literature so far has restricted to solving convex problems since there is no ambiguity about the location of the optimum. However, in practice there exist many real life problems in areas that range from from protein folding in bioinformatics to reservoir optimization in the oil and gas industry that are very computationally expensive and non-convex. Naturally, distributed non-convex optimization would be very desirable. However, non-convexity adds an extra degree of difficulty since it becomes hard to guarantee

convergence even to a local minimum due to oscillations. Non-convex distributed optimization is a highly open and unexplored field that merits further investigation.

Appendix A: Proof of equation (3.16)

We are looking to derive a recursive relationship.To unclutter notation, assume that $d = 1$ i.e., that all the DDA variables are scalars. Let us stack the local node variables in a vector $\boldsymbol{z} = [z_1 \cdots z_n]^T$ and $\boldsymbol{g} = [g_1 \cdots g_n]^T$. From (3.2) in matrix form we have after back-substituting in the recursion

$$\boldsymbol{z}(h+1) = P\boldsymbol{z}(h) + \boldsymbol{g}(h) = P\sum_{k=0}^{h-1} \boldsymbol{g}(k) + \boldsymbol{g}(h) \tag{9.1}$$

which can be generalized for s steps ahead to

$$\boldsymbol{z}(sh+1) = \sum_{w=1}^{s}\sum_{k=0}^{h-1} P^w \boldsymbol{g}\big((s-w)h+k\big) + \boldsymbol{g}(sh). \tag{9.2}$$

So in general

$$\begin{aligned}
\boldsymbol{z}(t) &= \sum_{w=1}^{H_t}\sum_{k=0}^{h-1} P^w \boldsymbol{g}\big((H_t-w)h+k\big) + \sum_{k=0}^{Q_t-1} \boldsymbol{g}(t-Q_t+k) && (9.3)\\
&= \sum_{w=1}^{H_t}\sum_{k=0}^{h-1} P^{H_t-w+1} \boldsymbol{g}\big((w-1)h+k\big) + \sum_{k=0}^{Q_t-1} \boldsymbol{g}(t-Q_t+k) && (9.4)\\
&= \sum_{w=0}^{H_t-1}\sum_{k=0}^{h-1} P^{H_t-w} \boldsymbol{g}\big(wh+k\big) + \sum_{k=0}^{Q_t-1} \boldsymbol{g}(t-Q_t+k) && (9.5)
\end{aligned}$$

where $H_t = \lfloor \frac{t-1}{h} \rfloor$ counts the number of communication steps in t iterations and $Q_t = \mathrm{mod}(t,h)$ if $\mathrm{mod}(t,h) > 0$ and $Q_t = h$ otherwise. From this last expression we take the i-th row to get the result.

Appendix B: Proof of equation (3.19)

If the consensus matrix P is doubly stochastic it is straightforward to show that $P^t \to \frac{1}{n}\mathbf{1}\mathbf{1}^T$ as $t \to \infty$. Moreover, from standard Perron-Frobenius is it easy to show (see e.g., [29])

$$\left\| \frac{1}{n}\mathbf{1}^T - \left[P^t\right]_{i,:} \right\|_1 = 2 \left\| \frac{1}{n}\mathbf{1}^T - \left[P^t\right]_{i,:} \right\|_{TV} \leq \sqrt{n}\left(\sqrt{\lambda_2}\right)^t \tag{9.6}$$

so in our case $\left\| \frac{1}{n}\mathbf{1}^T - \left[P^{H_t-w}\right]_{i,:} \right\|_1 \leq \sqrt{n}\left(\sqrt{\lambda_2}\right)^{H_t-w}$. Next, demand that the right hand side bound is less than $\sqrt{n}\delta$ with δ to be determined:

$$\sqrt{n}\left(\sqrt{\lambda_2}\right)^{H_t-w} \leq \sqrt{n}\delta \Rightarrow H_t - w \geq \frac{\log\left(\delta^{-1}\right)}{\log\left(\sqrt{\lambda_2}^{-1}\right)}. \tag{9.7}$$

So with the choice $\delta^{-1} = \sqrt{n}T$,

$$\left\| \frac{1}{n}\mathbf{1}^T - \left[P^{H_t-w}\right]_{i,:} \right\|_1 \leq \sqrt{n}\frac{1}{\sqrt{n}T} = \frac{1}{T} \tag{9.8}$$

if $H_t - w \geq \frac{\log\left(\delta^{-1}\right)}{\log\left(\sqrt{\lambda_2}^{-1}\right)} = \hat{t}$. When w is large and $H_t - w < \hat{t}$ we simply take $\left\| \frac{1}{n}\mathbf{1}^T - \left[P^{H_t-w}\right]_{i,:} \right\|_1 \leq 2$. The desired bound of (3.19) is not obtained as follows

$$\begin{aligned}
&\sum_{w=0}^{H_t-1} \left\| \frac{1}{n}\mathbf{1}^T - \left[P^{H_t-w}\right]_{i,:} \right\|_1 hL + 2hL \\
&\quad = \left(\sum_{w=0}^{H_t-\hat{t}-1} \left\| \frac{1}{n}\mathbf{1}^T - \left[P^{H_t-w}\right]_{i,:} \right\|_1 + \sum_{H_t-\hat{t}}^{H_t-1} \left\| \frac{1}{n}\mathbf{1}^T - \left[P^{H_t-w}\right]_{i,:} \right\|_1 \right) hL + 2hL \qquad (9.9) \\
&\quad \leq \left(\sum_{w=0}^{H_t-\hat{t}-1} \frac{1}{T} + \sum_{H_t-\hat{t}}^{H_t-1} 2 \right) hL + 2hL \qquad (9.10) \\
&\quad \leq \frac{H_t - \hat{t}}{T} hL + 2\hat{t}hL + 2hL. \qquad (9.11)
\end{aligned}$$

Since $t < T$ we know that $H_t - \hat{t} < T$. Moreover, $\log\left(\sqrt{\lambda_2}\right)^{-1} \geq 1 - \sqrt{\lambda_2}$. Using there two fact we arrive at the result.

Appendix C: Proof of Theorem 6.1

Consider a graph G with a consensus protocol P. Given a set of canonical paths $\Gamma = \{\gamma_{xy}\}$ on G we can compute the Poincaré constant K. If each link of G delivers messages with some arbitrary fixed delay of no more than B, we will show that the Poincaré constant $\widehat{K}$ of $\widehat{G}$ using the lazy additive reversibilization U of $\widehat{P}$ is bounded like $\widehat{K} \leq ZK$ where $Z = \Theta(B^2)$ We start with the definition of the Poincaré constant for $\widehat{K}$ and use the path associations discussed already to break the sum over all paths into nine summands. Assume that there are N_{vw} canonical paths in G that go through the bottleneck edge $e = (v, w)$ of G and let the bottleneck edge of $\widehat{G}$ be $\widehat{e} = (u, z)$ where u is in the set v^+ and z is in the the set w^-. Let x, y denote the starting and ending node of a path $\widehat{\gamma}_i$. We have

$$\begin{aligned}\widehat{K} =& \frac{1}{\widehat{\pi}_{v^+}[U]_{uz}}\big(T_1[x \to y] + T_2[x \to y^-] + T_3[x \to y^+] \\ &+ T_4[x^- \to y^-] + T_5[x^- \to y] + T_6[x^- \to y^+] \\ &+ T_7[x^+ \to y^-] + T_8[x^+ \to y] + T_9[x^+ \to y^+]\big) \qquad (9.12)\end{aligned}$$

with

$$T_1 = \sum_{i=1,\widehat{e}\in\widehat{\gamma}_i}^{N^{vw}} |\widehat{\gamma}_i|\,\widehat{\pi}_x\widehat{\pi}_y \tag{9.13}$$

$$T_2 = \sum_{i=1,\widehat{e}\in\widehat{\gamma}_i}^{N^{vw}} \sum_{k=-\frac{B^-}{2}}^{-1} \big(|\widehat{\gamma}_i| + k\big)\widehat{\pi}_x\widehat{\pi}_{y^-} \tag{9.14}$$

$$T_3 = \sum_{i=1,\widehat{e}\in\widehat{\gamma}_i}^{N^{vw}} \sum_{r=1}^{deg(y)} \sum_{k=1}^{\frac{B_r}{2}} \big(|\widehat{\gamma}_i| + k\big)\widehat{\pi}_x\widehat{\pi}_{y_r^+} \tag{9.15}$$

$$T_4 = \sum_{i=1,\widehat{e}\in\widehat{\gamma}_i}^{N^{vw}} \sum_{h=1}^{deg(x)} \sum_{j=-\frac{B_h}{2}}^{-1} \sum_{k=-\frac{B^-}{2}}^{-1} \big(|\widehat{\gamma}_i| + j + k\big)\widehat{\pi}_{x_h^-}\widehat{\pi}_{y^-} \tag{9.16}$$

$$T_5 = \sum_{i=1,\widehat{e}\in\widehat{\gamma}_i}^{N^{vw}} \sum_{h=1}^{deg(x)} \sum_{j=-\frac{B_h}{2}}^{-1} \big(|\widehat{\gamma}_i| + j\big)\widehat{\pi}_{x_h^-}\widehat{\pi}_y \tag{9.17}$$

$$T_6 = \sum_{i=1,\widehat{e}\in\widehat{\gamma}_i}^{N^{vw}} \sum_{h=1}^{deg(x)} \sum_{r=1}^{deg(y)} \sum_{j=-\frac{B_h}{2}}^{-1} \sum_{k=1}^{\frac{B_r}{2}} \big(|\widehat{\gamma}_i| + j + k\big)\widehat{\pi}_{x_h^-}\widehat{\pi}_{y_r^+} \tag{9.18}$$

$$T_7 = \sum_{i=1,\widehat{e}\in\widehat{\gamma}_i}^{N^{vw}} \sum_{j=-\frac{B^+}{2}}^{-1} \sum_{k=-\frac{B^-}{2}}^{-1} \big(|\widehat{\gamma}_i| + j + k\big)\widehat{\pi}_{x^+}\widehat{\pi}_{y^-} \tag{9.19}$$

$$T_8 = \sum_{i=1,\widehat{e}\in\widehat{\gamma}_i}^{N^{vw}} \sum_{j=-\frac{B^+}{2}}^{-1} \big(|\widehat{\gamma}_i| + j\big)\widehat{\pi}_{x^+}\widehat{\pi}_y \tag{9.20}$$

$$T_9 = \sum_{i=1,\widehat{e}\in\widehat{\gamma}_i}^{N^{vw}} \sum_{r=1}^{deg(y)} \sum_{j=-\frac{B^+}{2}}^{-1} \sum_{k=1}^{\frac{B_r}{2}} \big(|\widehat{\gamma}_i| + j + k\big)\widehat{\pi}_{x^+}\widehat{\pi}_{y_r^+} \tag{9.21}$$

To obtain a cleaner bound for $\widehat{K}$ we assume that P is doubly stochastic, recalling that the stationary distribution of delay nodes is $\widehat{\pi}_{x^*} \leq p\widehat{\pi}_x = \frac{p\pi_x}{c}$ for $p = \max_{i\neq j}(P_{ij})$ and replacing $*$ with either $+, -$. Recall also that each path in $\widehat{G}$ corresponds to exactly one path in G. Below we show how to bound the term T_6; bounds for all of the other terms defined above are obtained using similar arguments. Observe that for every path γ_{xy} between compute nodes x and y, if γ_{xy} goes through a bottleneck edge e in G, then all the delay paths $\widehat{\gamma}$ that are associated with γ_{xy} will go through $\widehat{e}$ in the middle of the delay chain that replaces e. So, for

term T_6 we have

$$\begin{aligned} T_6 \leq & \sum_{i=1,e\in\gamma_i}^{N^{vw}} \sum_{h=1}^{\deg(x)} \sum_{r=1}^{\deg(y)} \sum_{j=-\frac{B_h}{2}}^{-1} \sum_{k=1}^{\frac{B_r}{2}} \big((B+1)\,|\gamma_i| + j + k\big) \\ & \times \frac{p\pi_x}{c}\frac{p\pi_y}{c} \qquad (9.22) \\ \leq & \frac{p^2}{c^2} \sum_{i=1,e\in\gamma_i}^{N^{vw}} \deg(x)\deg(y) \\ & \times \sum_{j=-\frac{B}{2}}^{-1} \sum_{k=1}^{\frac{B}{2}} \big((B+1)\,|\gamma_i| + j + k\big)\pi_x\pi_y \qquad (9.23) \end{aligned}$$

Now since all paths γ_i are at least one edge long, bounding the node degrees by the maximum degree d_{max} in G gives

$$\begin{aligned} T_6 \leq & \frac{p^2}{c^2} d_{max}^2 \sum_{j=-\frac{B}{2}}^{-1} \sum_{k=1}^{\frac{B}{2}} \big((B+1) + j + k\big) \\ & \times \sum_{i=1,e\in\gamma_i}^{N^{vw}} |\gamma_i|\,\pi_x\pi_y \qquad (9.24) \\ = & \frac{p^2 d_{max}^2}{c^2} \frac{B^3+B^2}{4} \sum_{i=1,e\in\gamma_i}^{N^{vw}} |\gamma_i|\,\pi_x\pi_y \qquad (9.25) \end{aligned}$$

Through a similar derivation, all nine terms can be bound by a constant times $\sum_{i=1,e\in\gamma_i}^{N^{vw}} |\gamma_i|\,\pi_x\pi_y$ which appears in the expression for the Poincaré constant K without delays (see (6.20)). To make the exact expression for K appear, we focus on the leading term in (9.12) to see that

$$\begin{aligned} \frac{1}{\widehat{\pi}_{v^+}[U]_{uz}c^2} = & \frac{c}{\pi_v \frac{[U]_{uz}+[\tilde{U}]_{uz}}{2} c^2} = \frac{2}{\pi_v([U]_{uz}+0)c} \qquad (9.26) \\ = & \frac{2}{\pi_v c} = \frac{2p_{vw}}{c}\frac{1}{\pi_v p_{vw}}. \qquad (9.27) \end{aligned}$$

Next, remembering that $e=(v,w)$ is the bottleneck edge, after computing the exact constants in all terms, we write $\widehat{K} \leq ZK$ where Z is a function of the node degrees,

edge delays and consensus matrix P. Specifically,

$$\begin{aligned}\widehat{K} \leq & \frac{2p_{vw}}{c}\Big[(B+1)+p\frac{3B^2+2B}{8}+p\ d_{max}\frac{5B^2+6B}{8} \\ & + p^2 d_{max}\frac{B^3}{8}+pd_{max}\frac{3B^2+2B}{8}+p^2d_{max}^2\frac{B^3+B^2}{4} \\ & + p^2\frac{B^3}{8}+p\frac{3B^2+2B}{8}+p^2d_{max}\frac{B^3+B^2}{4}\Big] \\ & \times \underbrace{\frac{1}{\pi_v p_{vw}}\sum_{i=1,e\in\gamma_i}^{N^{vw}}|\gamma_i|\,\pi_x\pi_y}_{K} = ZK. \end{aligned} \tag{9.28}$$

Finally, focusing on the expression for Z, after some algebra, we see that

$$\begin{aligned} Z = \frac{p_{vw}}{4c}\Big[& p^2(2d_{max}^2+3d_{max}+1)B^3 \\ & + p(2pd_{max}^2+2pd_{max}+8d_{max}+6)B^2 \\ & + (8pd_{max}+p+8)B+8\Big] \end{aligned} \tag{9.29}$$

which completes the proof.

References

[1] Alekh Agarwal, Olivier Chapelle, Miroslav Dudik, and John Langford. A reliable effective terascale linear learning system. *arXiv:1110.4198v2*, 2012.

[2] Alekh Agarwal and John C. Duchi. Distributed delayed stochastic optimization. In *Neural Information Processing Systems*, 2011.

[3] Omer Angel, Alexander E. Holroyd, James Martin, David B. Wilson, and Peter Winkler. Avoidance coupling. In *arXiv:1112.3304v1*, 2011.

[4] Tuncer C. Aysal, Mehmet E. Yildiz, Anand D. Sarwate, and Anna Scaglione. Broadcast gossip algorithms for consensus. *IEEE Transactions on Signal Processing*, 57(7):2748 – 2761, July 2009.

[5] Peter L. Bartlett, Elad Hazan, and Alexander Rakhlin. Adaptive online gradient descent. In J C Platt, D Koller, Y Singer, and SEditors Roweis, editors, *Advances in Neural Information Processing Systems 20*. MIT Press, 2007.

[6] Ron Bekkerman, Mikhail Bilenko, and John Langford. *Scaling up Machine Learning, Parallel and Distributed Approaches*. Cambridge University Press, 2011.

[7] Florence Benezit, Vincent Blondel, Patrick Thiran, John Tsitsiklis, and Martin Vetterli. Weighted gossip: Distributed averaging using non-doubly stochastic matrices. In *IEEE International Symposium on Information Theory Proceedings (ISIT)*, pages 1753 – 1757, 2010.

[8] Dimitri P. Bertsekas. *Nonlinear Programming*. Athena Scientific, 1999.

[9] Dimitri P. Bertsekas. Incremental gradient, subgradient, and proximal methods for convex optimization: A survey. Technical Report 2848, MIT-LIDS, 2010.

[10] Dimitri P. Bertsekas and John N. Tsitsiklis. *Parallel and distributed computation: numerical methods*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1st edition, 1989.

[11] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[12] Pierre-Alexandre Bliman and Giancarlo Ferrari-Trecate. Average consensus problems in networks of agents with delayed communications. *Automatica*, 44, 2008.

[13] Vincent D. Blondel, Julien M. Hendrickx, Alex Olshevsky, and John N. Tsitsiklis. Convergence in multiagent coordination, consensus, and flocking. In *IEEE Conference on Decision and Control*, pages 2996 – 3000, 2006.

[14] Leon Bottou. Large-scale machine learning with stochastic gradient descent. In Yves Lechevallier and Gilbert Saporta, editors, *Proceedings of the 19th International Conference on Computational Statistics*, pages 177–187, Paris, France, August 2010.

[15] Stephen Boyd, Arpita Ghosh, Balaji Prabhakar, and Devavrat Shah. Randomized gossip algorithms. *IEEE Transactions on Information Theory*, 52:2508–2530, 2006.

[16] Stephen Boyd, Neal Parikh, Eric Chu, Borja Peleato, and Jonathan Eckstein. Distributed optimization and statistical learning via the alternating direction method of multipliers. *Foundations and Trends in Machine Learning*, 3(1):1–122, 2010.

[17] Stephen Boyd and Lieven Vandenberghe. *Convex Optimization*. 2004.

[18] Richard H. Byrd, Gillian M. Chiny, Will Neveitt, and Jorge Nocedal. On the use of stochastic hessian information in optimization methods for machine learning. *SIAM Journal on Optimization*, 21(3):977–995, 2011.

[19] Richard H Byrd, Peihuang Lu, and Jorge Nocedal. A limited memory algorithm for bound constrained optimization. *SIAM Journal on Scientific and Statistical Computing*, 16(5):1190–1208, 1995.

[20] Ming Cao, Stephen A. Morse, and Brian D. O. Anderson. Reaching a consensus in a dynamically changing environment: A graphical approach. *SIAM Journal on Control and Optimization*, 47(2):575–600, February 2008.

[21] Ming Cao, Stephen A. Morse, and Brian D. O. Anderson. Reaching a consensus in a dynamically changing environment: Convergence rates, measurement delays, and asynchronous events. *SIAM Journal on Control and Optimization*, 47:601–623, 2008.

[22] Yair Censor and Stavros Andrea Zenios. *Parallel Optimization: Theory, Algorithms, and Applications*. Oxford University Press, 1997.

[23] Nicolo Cesa-Bianchi and Gabor Lugosi. *Prediction, Learning, and Games*. Cambridge University Press, 2006.

[24] Annie I. Chen and Asuman Ozdaglar. A fast distributed proximal-gradient method. In *50th Allerton Conference on Communication, Control, and Computing*, 2012.

[25] Jianshu Chen and Ali H. Sayed. Diffusion adaptation strategies for distributed optimization and learning over networks. *IEEE Transactions on Signal Processing*, 60(8):4289–4305, August 2012.

[26] Fan Chung. *Spectral Graph Theory*. AMS, 1998.

[27] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM Magazine - 50th anniversary issue*, 51(1):107–113, 2008.

[28] Ofer Dekel, Ran Gilad-Bachrach, Ohad Shamir, and Lin Xiao. Optimal distributed online prediction using mini-batches. *Journal of Machine Learning Research*, 13:165–202, 2012.

[29] Persi Diaconis and Daniel Stroock. Geometric bounds for eigenvalues of markov chains. *The Annals of Applied Probability*, 1(1):36–61, 1991.

[30] Alexandros G. Dimakis, Soummya Kar, Jose M.F. Moura, Michael G. Rabbat, and Anna Scaglione. Gossip algorithms for distributed signal processing. *Proceedings of the IEEE*, 98(11):1847 – 1864, November 2010.

[31] Alejandro D. Domnguez-Garca and Christoforos N. Hadjicostis. Distributed matrix scaling and application to average consensus in directed graphs. *IEEE Transactions on Automatic Control*, 2013.

[32] John Duchi, Alekh Agarwal, and Martin Wainwright. Dual averaging for distributed optimization: Convergence analysis and network scaling. *IEEE Transactions on Automatic Control*, 57(3):592–606, 2011.

[33] Pául Erdős and Alfred Rényi. On the evolution of random graphs. *Publ. Math. Inst. Hungary. Acad. Sci.*, 5:17–61, 1960.

[34] James Allen Fill. Eigenvalue bounds on convergence to stationarity for non reversible markov chains, with an application to the exclusion process. *The Annals of Applied Probability*, 1(1):62–87, 1991.

[35] Bahman Gharesifard and Jorge Cortes. When does a digraph admit a doubly stochastic adjacency matrix? In *Proceedings of the American Control Conference*, pages 2440–2445, Baltimore, Maryland, 2010.

[36] Ananth Grama, George Karypis, Vipin Kumar, and Anshul Gupta. *Introduction to Parallel Computing*. Addison-Wesley, 2003.

[37] William Gropp, Steven Huss-Lederman, Andrew Lumsdaine, Ewing Lusk, Bill Nitzberg, William Saphir, and Marc Snir. *MPI—The Complete Reference - Volumes 1,2*. MIT Press, Cambridge, MA, 1998.

[38] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer, 2009.

[39] Elad Hazan, Adam Kalai, Satyen Kale, and Amit Agarwal. Logarithmic regret algorithms for online convex optimization. In *19'th COLT*, pages 499–513, 2006.

[40] Elad Hazan and Satyen Kale. Beyond the regret minimization barrier: an optimal algorithm for stochastic strongly-convex optimization. In *24th Annual Conference on Learning Theory (COLT)*, 2010.

[41] Mingyi Hong and Zhi-Quan Luo. On the linear convergence of the alternating direction method of multipliers. *arXiv:1208.3922v3*, 2013.

[42] Hal Daume III, Jeff M. Phillips, Avishek Saha, and Suresh Venkatasubramanian. Efficient protocols for distributed classification and optimization. In *Proceedings of the 23rd international conference on Algorithmic Learning Theory*, pages 154–168, 2012.

[43] Ilse C. F. Ipsen and Teresa M. Selee. Ergodicity coefficients defined by vector norms. *SIAM Journal on Matrix Analysis and Applications*, 32(1):153–200, 2011.

[44] Franck Iutzeler, Pascal Bianchi, Philippe Ciblat, and Walid Hachem. Asynchronous distributed optimization using a randomized alternating direction method of multipliers. In *IEEE Conference on Decision and Control*, 2013.

[45] Dusan Jakovetic, Joao Xavier, and Jose M.F. Moura. Fast distributed gradient methods. *arXiv:1112.2972v3*, 2013.

[46] Rong Jin, Shijun Wang, and Yang Zhou. Regularized distance metric learning: Theory and algorithm. In *Neural Information Processing Systems*, 2012.

[47] Bjorn Johansson, Maben Rabi, and Mikael Johansson. A randomized incremental subgradient method for distributed optimization in networked systems. *SIAM Journal on Control and Optimization*, 20(3), 2009.

[48] Sham M. Kakade and Shai Shalev-Shwartz. Mind the duality gap: Logarithmic regret algorithms for online optimization. In *Neural Information Processing Systems*, 2009.

[49] David Kempe, Alin Dobra, and Johannes Gehrke. Gossip-based computation of aggregate information. In *FOCS, vol. 44. IEEE Computer Society Press, pp. 482–491*, 2003.

[50] John Langford, Alexander J. Smola, and Martin Zinkevich. Slow learners are fast. In *Neural Information Processing Systems*, 2009.

[51] Gideon Mann, Ryan McDonald, Mehryar Mohri, Nathan Silberman, and Daniel D. Walker. Efficient large-scale distributed training of conditional maximum entropy models. In *Neural Information Processing Systems*, pages 1231–1239, 2009.

[52] Ryan McDonald, Keith Hall, and Gideon Mann. Distributed training strategies for the structured perceptron. In *Annual Conference of the North American Chapter of the Association for Computational Linguistics*, pages 456–464, 2012.

[53] Ravi Montenegro and Prasad Tetali. *Mathematical Aspects of Mixing Times in Markov Chains*. Foundations and Trends in Theoretical Computer Science (Vol 1, No 3), 2006.

[54] Angelia Nedic, Dimitri P. Bertsekas, and Vivek S. Borkar. Distributed asynchronous incremental subgradient methods. In *Inherently parallel algorithms in feasibility and optimization and their applications*, 2000.

[55] Angelia Nedic and Asuman Ozdaglar. Distributed subgradient methods for multi-agent optimization. *IEEE Transactions on Automatic Control*, 54(1), January 2009.

[56] Angelia Nedic and Asuman Ozdaglar. Convergence rate for consensus with delays. *Journal of Global Optimization*, 47(3):437–456, 2010.

[57] Angelia Nedic and Asuman Ozdaglar. Cooperative distributed multi-agent optimization. *Convex Optimization in Signal Processing and Communications*, 2010.

[58] Angelia Nedic, Asuman Ozdaglar, and Pablo A. Parrilo. Constrained consensus and optimization in multi-agent networks. *IEEE Transactions on Automatic Control*, 55(4):922–938, 2010.

[59] Arkadi Nemirovski, Anatoli Juditsky, Guanghui Lan, and Alex Shapiro. Robust stochastic approximation approach to stochastic programming. *SIAM Journal on Optimization*, 19(4):1574–1609, 2009.

[60] Yuri Nesterov. *Introductory Lectures on Convex Optimization: A Basic Course*. Kluwer, Boston, 2004.

[61] Yuri Nesterov. Primal-dual subgradient methods for convex problems. *Mathematical Programming Series B*, 120:221–259, 2009.

[62] Jorge Nocedal and Steve J. Wright. *Numerical Optimization*. Springer, 1989.

[63] Reza Olfati-Saber, J. Alex Fax, and Richard M. Murray. Consensus and cooperation in networked multi-agent systems. In *Proceedings of the IEEE*, volume 95:1, pages 215 – 233, 2007.

[64] Reza Olfati-Saber and Richard M. Murray. Consensus problems in networks of agents with switching topology and time-delays. *IEEE Transactions on Automatic Control*, 49(9):1520–1533, September 2004.

[65] Alex Olshevsky and John N. Tsitsiklis. Convergence speed in distributed consensus and averaging. *SIAM Journal on Control and Optimization*, 48, No 1:33–55, 2009.

[66] Victor M. Preciado, Alireza Tahbaz-Salehi, and Ali Jadbabaie. On asymptotic consensus value in directed random networks. In *49th IEEE Conference on Decision and Control*, Atlanta, GA, USA, December 2010.

[67] Alexander Rakhlin, Ohad Shamir, and Karthik Sridaran. Making gradient descent optimal for strongly convex optimization. In *arXiv:1109.5647v6*, 2012.

[68] S. Sundhar Ram, Angelia Nedic, and Venugopal V. Veeravalli. A new class of distributed optimization algorithms: Application to a new class of distributed

optimization algorithms: Application to regression of distributed data. *Optimization Methods and Software*, 27(1):71–88, 2009.

[69] S. Sundhar Ram, Angelia Nedic, and Venugopal V. Veeravalli. Distributed stochastic subgradient projection algorithms for convex optimization. *Journal of Optimization Theory and Applications*, 147(3):516–545, 2011.

[70] Omer Reingold, Salil Vadhan, and Avi Wigderson. Entropy waves, the zig-zag graph product, and new constant-degree expanders. *Annals of Mathematics*, 155(2):157–187, 2002.

[71] Jeal-Pierre Richard. Time-delay systems: an overview of some recent advances and open problems. *Automatica*, 39:1667–1694, 2003.

[72] Mark Schmidt, Nicolas Le Roux, and Francis Bach. Convergence rates of inexact proximal-gradient methods for convex optimization. In *25th Annual Conference on Neural Information Processing Systems (NIPS)*, 2011.

[73] Eugene Seneta. *Non-negative Matrices and Markov Chains.* Springer, 1973.

[74] Alexandre Seuret, Dimos V. Dimarogonas, and Karl H. Johansson. Consensus under communication delays. In *Proceedings of the 47th IEEE Conference on Decision and Control*, 2008.

[75] Shai Shalev-Shwartz. *Online learning and online convex optimization.* Foundations and Trends in Machine Learning, 4, (2):107-194, 2012.

[76] Shai Shalev-Shwartz, Ohad Shamir, Nathan Srebro, and Karthik Sridaran. Stochastic convex optimization. In *Proceedings of the Conference on Learning Theory (COLT)*, 2009.

[77] Shai Shalev-Shwartz and Yoram Singer. Logarithmic regret algorithms for strongly convex repeated games. Technical report, The Hebrew University, 2007.

[78] Shai Shalev-Shwartz, Yoram Singer, and Andrew Y. Ng. Online and batch learning of pseudo-metrics. In *International Conference in Machine Learning (ICML)*, pages 743–750, 2004.

[79] Suvrit Sra, Sebastian Nowozin, and Stephen J. Wright. *Optimization for Machine Learning.* MIT Press, 2011.

[80] Kunal Srivastava and Angelia Nedic. Distributed asynchronous constrained stochastic optimization. *IEEE Journal of Selected Topics in Signal Processing*, 5:4:772–790, 2011.

[81] W. Richard Stevens, Bill Fenner, and Andrew M. Rudoff. *Unix Network Programming, Volume 1: The Sockets Networking API.* Addison-Wesley Professional Computing Series, 2003.

[82] Alireza Tahbaz-Salehi and Ali Jadbabaie. Necessary and sufficient conditions for consensus over random independent and identically distributed switching

graphs. In *Proceedings of the 46th IEEE Conference on Decision and Control*, 2007.

[83] Hakan Terelius, Ufuk Topcu, and Richard M. Murray. Decentralized multi-agent optimization via dual decomposition. In *IFAC World Congress*, 2011.

[84] Behrouz Touri. *Product of Random Stochastic Matrices and Distributed Averaging.* PhD thesis, University of Illinois at Urbana-Champaign, 2011.

[85] Paul Tseng. On accelerated proximal gradient methods for convex-concave optimization. *SIAM Journal on Optimization*, 2008.

[86] Konstantinos I. Tsianos, Sean Lawlor, and Michael G. Rabbat. Communication/computation tradeoffs in consensus-based distributed optimization. In *Neural Information Processing Systems*, pages 1952–1960, 2012.

[87] Konstantinos I. Tsianos, Sean Lawlor, and Michael G. Rabbat. Consensus-based distributed optimization: Practical issues and applications in large-scale machine learning. In *50th Allerton Conference on Communication, Control, and Computing*, pages 1543 – 1550, 2012.

[88] Konstantinos I. Tsianos, Sean Lawlor, and Michael G. Rabbat. Push-sum distributed dual averaging for convex optimization. In *51st IEEE Conference on Decision and Control*, pages 5453 – 5458, 2012.

[89] Konstantinos I. Tsianos and Michael G. Rabbat. Distributed consensus and optimization under communication delays. In *49th Allerton Conference on Communication, Control, and Computing*, pages 974 – 982, 2011.

[90] Konstantinos I. Tsianos and Michael G. Rabbat. Distributed dual averaging for convex optimization under communication delays. In *American Control Conference (ACC)*, pages 1067 – 1072, 2012.

[91] Konstantinos I. Tsianos and Michael G. Rabbat. Distributed strongly convex optimization. In *50th Allerton Conference on Communication, Control, and Computing*, pages 593 – 600, 2012.

[92] Konstantinos I. Tsianos and Michael G. Rabbat. The impact of communication delays on distributed consensus algorithms. *Submitted to Transactions on Automatic Control (http://arxiv.org/abs/1207.5839)*, 2012.

[93] Konstantinos I. Tsianos and Michael G. Rabbat. Simple iteration-optimal distributed optimization. In *European Signal Processing Conference*, 2013.

[94] John N. Tsitsiklis, Dimitri P. Bertsekas, and Michael Athans. Distributed asynchronous deterministic and stochastic gradient optimization algorithms. *IEEE Transactions on Automatic Control*, 31(9):803–812, 1986.

[95] Nitin H. Vaidya, Christoforos N. Hadjicostis, and Alejandro D. Dominguez-Garcia. Distributed algorithms for consensus and coordination in the presence of packet-dropping communication links - part ii: Coefficients of ergodicity analysis approach. Technical Report UILU-ENG-11-2208 (CRHC-11-06), UIUC, 2011.

[96] Ermin Wei and Asuman Ozdaglar. Distributed alternating direction method of multipliers. In *IEEE Conference on Decision and Control*, pages 5445–5450, 2012.

[97] Kilian Q. Weinberger and Lawrence K. Saul. Distance metric learning for large margin nearest neighbor classification. *Journal of Optimization Theory and Applications*, 10:207–244, 2009.

[98] Kilian Q. Weinberger, Fei Sha, and Lawrence K. Saul. Convex optimizations for distance metric learning and pattern classification. *IEEE Signal Processing Magazine*, 2010.

[99] Chai Wah Wu. On some properties of contracting matrices. *Linear Algebra and its Applications*, 428:2509–2523, 2008.

[100] Lin Xiao. Dual averaging methods for regularized stochastic learning and online optimization. *Journal of Machine Learning Research*, 11:2543–2596, 2010.

[101] Eric P. Xing, Andrew Y. Ng, Michael I. Jordan, and Stuart Russell. Distance metric learning, with application to clustering with side-information. In *Neural Information Processing Systems*, 2003.

[102] Minghui Zhu and Sonia Martínez. An approximate dual subgradient algorithm for distributed non-convex constrained optimization. In *49th IEEE Conference on Decision and Control (CDC)*, pages 7487 – 7492, 2010.

[103] Martin A. Zinkevich. Online convex programming and generalized infinitesimal gradient ascent. In *20th International Conference on Machine Learning (ICML)*, 2003.

[104] Martin A. Zinkevich, Markus Weimer, Alexander Smola, and Lihong Li. Parallelized stochastic gradient descent. In *Neural Information Processing Systems*, 2010.