universität wien

DISSERTATION

Titel der Dissertation

Truly Distributed Approaches to Orthogonalization and Orthogonal Iteration on the Basis of Gossip Algorithms

Verfasserin

Hana Straková

angestrebter akademischer Grad

Doktorin der technischen Wissenschaften (Dr. techn.)

Wien, 2013

Studienkennzahl lt. Studienblatt:	A 786 880
Dissertationsgebiet lt. Studienblatt:	Informatik
Betreuer:	Assoz. Prof. Dr. Wilfried Gansterer, Privatdoz.

Acknowledgments

This thesis would not be possible without the help and support of many people. Foremost, I would like to gratefully thank my advisor Wilfried Gansterer for his guidance, support, valuable advises and his patience. In particular, I am grateful for every detailed comment he gave to me and for teaching me, how to regard a question from many perspectives. I would also like to thank for financial support for this research, which came from FWF under award S10608 within the National Research Network SISE.

I would also like to thanks all cooperation partners Markus Rupp, Ondrej Slučiak and Thomas Zemen for inspiring joint work and Le Song for introducing me to the topic of distributed linear regression.

Very special thanks go to Rich Vuduc for giving me the opportunity to spend three great months at Georgia Tech with his students and helping me with getting the most out of my stay.

Furthermore, I am grateful to my co-workers and friends Andi, Gerhard, Aparna, Veronica and Jee not only for inspiring debates about research but especially for helping me with settling down in new countries.

Last but not least, I thank my family for their support and encouragement during my entire studies.

Abstract

Gossip or epidemic algorithms are communication protocols based on randomized information exchange in nodes' neighborhoods. Thanks to the randomized communication they are very flexible with respect to the underlying hardware infrastructure, they can operate on arbitrary topologies and tolerate various types of failures. Moreover, they have the ability to trade invested cost for final accuracy, which may lead to significant cost reductions. The natural target systems are loosely coupled decentralized distributed systems, such as P2P networks or sensor networks. So far, gossip-based algorithms have been utilized mostly for simple operations, such as aggregation (summation or averaging), information spreading or estimating size of a network.

The main focus of this thesis is on design and investigation of truly distributed matrix algorithms based on gossip-style data aggregation algorithms. Due to the specific properties of highly distributed loosely coupled systems, traditional parallel algorithmic approaches (designed for tightly coupled systems like shared-memory systems or multicore architectures) have important drawbacks, such as strong assumptions on static and known topology, reliable components and full synchronization of nodes. Hence, there is a need to design fully distributed algorithms for which these assumptions can be relaxed.

In particular, we focus on distributed Gram-Schmidt orthogonalization for computing a QR factorization, and on distributed orthogonal iteration for computing the principal eigenpairs of a matrix. We also show how distributed QR factorization can be used for solving a linear regression problem over distributed data. Concrete motivating applications for distributed orthogonalization arise, for example, in telecommunications for various computations in mobile and wireless sensor networks. An example is *distributed sphere decoding*, a distributed algorithm allowing a group of receiving nodes to cooperate while decoding a signal sent by a group of transmitters. Distributed orthogonalization is also an important building block for more complex distributed algorithms, such as distributed least squares solvers, eigensolvers, etc.

The need for distributed computation of eigenvalues and eigenvectors may also

arise in network analysis. Some crucial properties of a graph, such as how well the graph is connected, can be determined by computing eigenvalues of the Laplacian matrix of the graph. Information about the connectivity of a graph can be found from the second smallest eigenvalue which is called the *algebraic connectivity* of a graph. In practice this can be applied, e.g., for detecting bottlenecks in computer networks or for analyzing relationships in social networks.

We first introduce a gossip-based distributed algorithm for Gram-Schmidt orthogonalization and we show experimentally as well as theoretically that the distributed algorithm preserves the numerical properties of the classical centralized algorithm. Moreover, we experimentally observe the influence of the level of synchronization between the nodes on the accuracy of the result. Although our algorithm requires some cooperation between nodes, very weak synchronization strategy based only on local information is sufficient for achieving accurate results.

Then, we investigate distributed orthogonal iteration as an illustrative example of using QR factorization as a building block. Apart from investigating numerical accuracy and convergence, we illustrate how accuracy-performance trade-offs can lead to significant reduction of communication cost. Moreover, using gossip-based building blocks substantially improves the robustness and fault tolerance compared to existing approaches.

Last but not least, we discuss first steps towards MPI implementations of the gossip-based algorithms for parallel computers in an effort to estimate runtime overhead of the distributed algorithms. We present first results of several runtime comparisons to standard parallel approaches, such as MPI_Allreduce or PBLAS routine `pdgemv` commonly used in ScaLAPACK.

Zussamenfassung

Gossip[1] bzw. epidemische Algorithmen sind Kommunikationsprotokolle in welchen Knoten ausschließlich mit zufällig bestimmten unmittelbaren Nachbarn Nachrichten austauschen. Aufgrund ihrer randomisierten Kommunikation sind diese Algorithmen äußerst flexibel bzgl. der zu Grunde liegenden Hardwareinfrastruktur, sowie der Netzwerktopologie. Darüber hinaus können sie vielerlei Fehler wie z. B. den Verlust von Nachrichten tolerieren und die erzielte Genauigkeit kann mit dem Gesamtaufwand abgewogen werden. Gossip-basierte Algorithmen wurden bisher hauptsächlich für elementare Operationen wie Aggregationen (Summation oder Durchschnittsbildung) bzw. allgemeiner, zur Verbreitung von Information in Netzwerken eingesetzt. Entsprechend stellen sich dezentrale verteilte Systeme wie P2P- oder Sensornetzwerke als natürliche Zielplatformen dar.

Das Hauptaugenmerk dieser Arbeit liegt auf der Entwicklung und Untersuchung von verteilten Matrixalgorithmen welche auf gossip-basierten Aggregationsalgorithmen beruhen. Insbesondere untersuchen wir einen verteilten Gram-Schmidt Orthogonalisierungsprozess zur Berechnung von QR Faktorisierungen und eine verteile Orthogonale Iteration zur Berechnung der dominierenden Eigenpaare einer Matrix. Darüber hinaus zeigen wir, wie mit Hilfe einer verteilten QR Faktorisierung lineare Regressionsprobleme über verteilten (Mess-)Daten gelöst werden können.

Aufgrund der Spezifika dezentraler verteilter Systeme ist es nicht bzw. nur sehr schwer möglich existierende Algorithmen aus der parallelen Datenverarbeitung direkt zu übernehmen, da diese an Regularitätsanforderungen gebunden sind, die verteilte Systeme nicht erfüllen. Daher ist es eine Notwendigkeit spezifische (verteilte) Algorithmen für derartige dezentrale Systeme zu entwickeln.

Nebst der Entwicklung verteilter Algorithmen zeigen wir auch einige potentielle Anwendungen dieser auf. Eine Anwendung der verteilten Orthogonalisierung stammend aus der Telekommunikation ist *distributed sphere decoding*. Dabei wird einer Gruppe von Empfängern, durch einen verteilten Algorithmus, die Kooperation während des Dekodierens von Signalen einer Gruppe von Sendern ermöglicht.

[1] engl. Klatsch, Plauderei, Gemunkel

Anwendungen der verteilten Orthogonalen Iteration finden sich unter anderem in der Netzwerkanalyse, wo z. B. mittels spezifischer Eigenwerte und Eigenvektoren Aussagen über die Konnektivität des Netzwerks getroffen werden können.

Zunächst führen wir in dieser Arbeit einen gossip-basierten verteilen Gram-Schmidt Orthogonalisierungsprozess ein und zeigen theoretisch sowie experimentell, dass dieser die numerischen Eigenschaften der klassischen Variante erhält. Darüber hinaus untersuchen wir experimentell in wie weit sich die Synchronisation zwischen den Knoten auf die Genauigkeit der berechneten Resultate auswirkt. Obwohl die Knoten klarerweise untereinander kooperieren müssen, sind nur einfachste lokale Synchronisationsmechanismen von Nöten, um genaue Ergebnisse zu liefern.

Darauf aufbauend untersuchen wir eine verteilte Orthogonale Iteration als Anwendungsbeispiel der verteilten QR Faktorisierung. Neben Untersuchungen zur Konvergenz und numerischen Genauigkeit, illustrieren wir auch wie Genauigkeit und Aufwand zur Reduzierung der Kommunikationskosten gegeneinander abgewogen werden können. Zusätzlich liefert der Einsatz von gossip-basierten Elementaroperationen auch substantiell robustere bzw. fehlertolerante Algorithmen (im Vergleich zu existierenden Methoden).

Abschließend präsentieren wir erste Schritte in Richtung einer MPI Implementierung der untersuchten gossip-basierten Algorithmen für Parallelrechner. Diese Untersuchungen dienen zur Quantifizierung des Mehraufwands den verteilte Algorithmen mit sich bringen. Konkret präsentieren wir erste Vergleiche mit etablierten parallelen Routinen wie MPI_Allreduce bzw. der PBLAS Routine `pdgemv` die in ScaLAPACK Verwendung findet.

Contents

List of Algorithms

List of Figures

List of Tables

Chapter 1

Introduction

Gossiping is a very natural way of communicating and information spreading in everyday life. If somebody gets a piece of information, he or she will very likely share it with a friend or a colleague, once there is an opportunity. The key aspect of gossiping is that in most cases the "communication partner" is not chosen intentionally, but the piece of information is transmitted to a first person for whom the information may be relevant. One could argue about the moral aspects of gossiping, but it has to be admitted that it is a very powerful method of information spreading.

The principle of gossiping has been utilized as the communication paradigm in various simple applications in signal processing in decentralized networks. But all kinds of mobile devices, such as mobile phones, laptops or netbooks, are becoming more and more computationally powerful and thus also networks built by connecting such devices have a potential for executing complex computations. Existing algorithms, such as various parallel algorithms or the centralized approach using a fusion center, have severe disadvantages in many scenarios for computations over such distributed systems, such as little flexibility with respect to the underlying hardware infrastructure or unbalanced load on the nodes. This causes increased interest in designing distributed algorithms which would be suited for decentralized distributed network and exploit their properties.

Question naturally arising in this context are: Can a simple tool like gossiping be used for more complex and organized computations over distributed systems? And if yes, does it bring any advantages over alternative approaches?

Distributed vs. Parallel Algorithms

The usage of the term "distributed" varies in the literature and it is often used interchangeably with the term "parallel". As we also explained in our publications (see Section 1.6), to avoid misunderstanding it is necessary to clarify right at the

beginning what exactly we mean by "distributed algorithm" and how we distinguish it from a "parallel algorithm".

We call *parallel algorithms* the algorithms developed primarily for "classical" parallel computers (comprising shared-memory systems, tightly coupled distributed computers, multicore architectures, etc.). Target systems for parallel algorithms are characterized by a *static* and usually regular topology (e. g., a processor grid) known in all nodes and reliable, relatively fast and cheap wired communication. The data distribution is usually chosen to bring the best parallel performance (e. g., 2D block-cyclic as in ScaLAPACK (Blackford et al., 1997)). Moreover, algorithms executed over parallel computers usually rely on synchronization of processes which can be guaranteed, either in terms of access to shared-memory or in terms of communication points. As a consequence, parallel algorithms can use fixed deterministic communication schedules in the way which optimized their performance.

In contrast, we use the term *distributed algorithm* for the algorithms which are designed for loosely coupled decentralized systems. The initial data of distributed algorithms, such as measurements, is usually generated directly in the nodes. Nodes in distributed algorithms usually do not have any global information about the topology and communicate only with their direct neighbors, which provides greater flexibility with respect to the hardware infrastructure compared to parallel algorithms.

Communication between nodes may be unreliable or noisy and therefore, resilience against link failures or bit-flip is for distributed algorithms more important property than for parallel algorithms. Furthermore, a desired property of a distributed algorithm is that it does not require global synchronization of the nodes, because synchronizing a loosely coupled system is usually more challenging than it is in parallel systems.

More generally speaking, distributed algorithms are useful in all computations over distributed computing systems where (*i*) the nodes do not have complete information about the system, but only local information about their neighborhood, and/or (*ii*) the system is not static, but may change dynamically (e. g., due to hardware failures, mobile nodes, etc.).

Main Goals of the Thesis

This thesis deals with the design and analysis of novel distributed matrix algorithms which are characterized by communication schedules where each node exchanges information only with *randomly* selected nearest neighbors. In particular, we address questions related to design of distributed QR factorization, and its applications in more complex numerical linear algebra algorithms, such as eigensolvers or distributed least-square solvers.

Above all, we want to investigate if distributed algorithms based on flexible communication schedule only with nearest neighbors brings any advantages in terms of higher fault tolerance and/or less synchronization of nodes than existing approaches and what is the additional cost for such properties.

Apart from that we investigate if the considered distributed matrix algorithms preserve known properties, such as numerical stability or convergence speed, of their sequential counterparts. We are particularly interested in the numerical accuracy of the distributed algorithms and the aspects influencing the accuracy of the delivered results. Moreover, the thesis aims to answer questions related to scalability with increasing number of nodes and problem size and to their communication cost. Last but not least, the comparison of the distributed algorithms with the alternative approaches, such as parallel algorithms, is of interest.

1.1 Motivation

The standard way of executing computations in decentralized systems, such as P2P networks, wireless sensor networks, mobile ad hoc networks, etc., is the *fusion center* approach. Locally stored values from all nodes in the distributed system are collected at one of the nodes, commonly called the fusion center, which computes the desired result locally, and distributes it back to the respective nodes. However, this approach may have significant drawbacks when applied in a distributed system. First of all, it may be infeasible to collect the data from all nodes at the fusion center due to technical constraints (e. g., when no single node can store the entire input data because of limited memory capacity) or due to security issues, if it is undesirable to collect all data into one storage location. Moreover, collecting the data at the fusion center may cause overhead because of potential congestion around the fusion center (Khan et al., 2013). In addition, such a centralized computation leads to highly unbalanced energy consumption across nodes and global knowledge of the network for routing information must be available in order to collect all data at a single node and to redistribute the result. Last but not least, the centralized approach introduces a highly vulnerable point to the system, because it collects the data from the entire network. If the fusion center fails, none of the nodes can get the result, unless all remaining nodes agree on a new fusion center and send their data there.

Also classical parallel algorithms designed for systems like multicore architectures or shared-memory systems have major drawbacks when executed over decentralized loosely coupled systems, because of substantial differences in the underlying hardware infrastructure. Among others, nodes in distributed systems have usually limited local resources in terms of storage, computational capabilities and energy

(stored in batteries). Due to potential limitations of local storage together with limited capacity of communication links, exchanging large messages between nodes does not have to be the best strategy, as it is a well established rule of thumb in parallel computing. Moreover, the initial data distribution for distributed matrix algorithms, such as measurements generated directly in the nodes, is mainly determined by the application and cannot be chosen to optimize the performance.

In addition, various outlooks on the development of large-scale parallel systems (e. g., Cappello et al. (2009); Varela et al. (2010); Dongarra (2012)) indicate that high performance computing (HPC) systems may significantly change in some aspects. With growing size of the high performance computer systems their overall reliability is decreasing, the systems may change dynamically over time due to frequent failures and also synchronization of the nodes may become very difficult. Consequently, the existing parallel approaches will probably not be sufficient for the anticipated extreme-scale HPC systems of the future and some of the attractive properties of distributed algorithms for loosely coupled systems may become relevant also for large-scale multicore systems or for cloud infrastructures.

1.2 Methodology

The basic idea we pursue in this thesis is to investigate the utilization a special class of simple network communication network protocols called *gossip* (or *epidemic*) *algorithms* for developing truly distributed algorithms for matrix computations. Gossip algorithms are decentralized iterative algorithms where in each round every node chooses (a) *random* partner(s) to exchange information with. Due to their randomized communication schedule they are much more flexible and robust with respect to the hardware infrastructure than classical parallel approaches. They do not make any assumptions about specific connections between nodes, thus they work on arbitrary (connected) topologies with potential changes during runtime. No global knowledge of the topology is needed, because the algorithms operate based on local knowledge in individual nodes. Moreover, because algorithms with randomized information exchange do not assume reliable communication channels, they have a big potential for delivering accurate results despite dynamic network changes and consequently achieve high fault tolerance. Due to their decentralized structure, gossiping-style algorithms do not introduce a single point of failure and they achieve good scalability with the number of nodes. Furthermore, they allow for trading off achieved numerical accuracy against communication cost by gradually adapting the intensity and regularity of interaction with other nodes. This is very attractive in situations where the application does not need result in double precision, but ap-

proximate intermediate results are of interest. Last but not least, simple gossip-style protocols do not assume synchronization across nodes.

In order to analyze the behavior of the distributed algorithms developed in this thesis we consider the following metrics. We investigate the numerical accuracy of the results influenced by several aspects, such as varying target accuracy of the building blocks or message losses. Furthermore, we evaluate the communication cost in terms of the number of messages sent as well as the total amount of data sent during the computation, and we study their scaling behavior with increasing problem and network size. We do not measure communication cost purely experimentally, but we quantify in based on analytical counts. We assume that the algorithms run across a physical topology, where each neighbor is reachable in one hop. Thus, sending an information to a neighbor we count as one message.

To evaluate quantitatively the important metrics for this dissertation project we use mainly analytical modeling and simulation methodology. For verifying the analytical bounds and theoretical results we simulate the distributed algorithms in MATLAB. Although MATLAB is a very suitable tool for observing numerical accuracy or convergence behavior of the algorithms even in the presence of message loss, certain aspects, such as the impact of the level of synchronization, cannot be investigated. Thus, we also implement several algorithms in the ns-3[1] network simulator. To compare our newly developed distributed approaches with state-of-the-art parallel algorithms we implement them in MPI and we provide runtime comparisons based on execution on parallel machines.

1.3 Scientific Challenges

Gossip-based matrix algorithms and their properties have hardly been investigated in the literature so far. Therefore, first of all we address questions related to the utilization of gossip algorithm as building blocks for distributed matrix algorithms, such as: Which of the attractive properties of simple gossip-based aggregation algorithms are inherited also by more complex distributed matrix algorithms and to what extent? How does the numerical accuracy of the classical sequential matrix algorithms change when the centralized summations are replaced by a randomized distributed gossip-based algorithm? What is the level of process synchronization needed in the gossip-based matrix algorithms in order to compute accurate results? What types of failures can be tolerated by gossip-based algorithms? Furthermore, we target one of the remarkable aspects of gossip-based algorithms, namely their ability to adjust invested cost to the final accuracy achieved: In which scenarios the newly developed

[1] http://www.nsnam.org/

distributed matrix algorithms benefit from the accuracy-performance trade-offs?

Additionally, it is to be expected that for the versatility and flexibility of the resulting distributed matrix algorithms additional cost in terms of number of messages sent or overall runtime has to be invested. Besides analytical comparison of the distributed algorithms to existing parallel approaches in terms of communication cost, we also focus on implementations of gossip-based algorithms in MPI in order to compare them to their parallel counterparts in terms of runtime. Because the two types of algorithms originate from substantially different contexts, it is not straightforward how to implement in MPI such algorithms with randomized information exchange. In particular, we target questions related to MPI implementations, such as: How to implement efficient gossip-based aggregation without calling global synchronization barriers? How to accomplish randomized information exchange between processes? How to terminate the computation with randomized communication schedule?

1.4 Application Scenarios

To illustrate some real world scenarios where distributed algorithms for matrix computations are beneficial, we discuss a couple of concrete applications for decentralized systems.

Distributed Sphere Decoding

As the motivating application for a distributed orthogonalization of a set of vectors we mention the *distributed sphere decoding* method (see Dumard and Riegler (2009)), which is used in cooperative transceiver design in telecommunication. It is a distributed algorithm allowing a group of receiving nodes to cooperate while decoding a received signal.

Centralized sphere decoding is used in multiple-input multiple-output (MIMO) communication systems that employ multiple antennas at the transmitter and receiver side for decoding a received signal. The communication system can be described as

$$y = Hx + z, \tag{1.1}$$

where the complex channel matrix $H \in \mathbb{C}^{n \times n}$, describing how good the quality of the channels from each transmitter to each receiver is, has elements drawn from complex standard normal distribution and $z(i) \sim \mathcal{CN}(0, \sigma_n^2)$ denotes additive complex white Gaussian noise with elements distributed according to a complex normal distribution. The transmit vector $x \in \mathcal{A}^n$ has elements drawn from a finite alphabet $\mathcal{A}$ with $|\mathcal{A}|$ elements and vector $y \in \mathbb{C}^n$ denotes the noisy received vector. Sphere

decoding allows for solving the maximum likelihood problem

$$\hat{x} = \underset{d \in \mathcal{A}^n}{\operatorname{argmin}} \|y - Hd\|^2$$

efficiently. For more details on the sphere decoding algorithm see Burg et al. (2005).

In order to use the advantages of MIMO communications for wireless sensor networks, the receiving node can use the help of nearby sensors (Ozgur et al., 2007) in order to jointly decode the signal sent by a group of transmitters. Dumard and Riegler (2009) formulated a *distributed* sphere decoding algorithm based on solving the maximum likelihood problem Equation (1.1) in a distributed way. The first part of their algorithm is a distributed QR factorization of the channel matrix H, which is initially distributed row-wise across the sensors.

Network Analysis, Localization

Motivating eigenvalue problems in a distributed setup arise in various types of network analyses, as important information about properties of the network structure can be obtained from the spectral analysis of the network (Fiedler, 1973) and from the Laplacian matrix of the graph (Chung, 1997). A network analysis can be used for example for finding potential performance-limiting bottlenecks when designing a computer network to detect connections and relationships between members in a social network (see Kempe and McSherry (2008) and the references therein). In the context of web search engines eigenvalues can be used for ranking webpages according to relevancy for a searched term (Page et al., 1998). An illustrative example showing how a distributed eigensolver can be used for network analysis is shown in Chapter 6.

Another application for a distributed eigensolver is a method called multidimensional scaling for computing accurate positioning of sensors in indoor environments described by Korada et al. (2011). This method uses the fact that the positions of the sensors can be obtained by computing three principal eigenvectors of a matrix containing measurements of pairwise distances between the sensors in the network.

Distributed Least-Squares Problems

Various applications based on computing a least-squares problem in a distributed way arise in sensor and computer networks. One example is tomography imaging applied to seismography, where is a need for a distributed least-squares method, as shown by Shi et al. (2013).

Another potential application context for a distributed gossip-based least-square

solver is data mining and machine learning. In many scenarios, the data used as input for data mining applications are often generated or collected directly in the participating nodes, but due to privacy reasons cannot be collected at a single node (e. g., Ormandi et al. (2013)). We focus on a distributed regression problem, which is suited for continuous output values. Each node computes a function fitting the best the currently available data in the whole network, and then it uses locally the calculated function for evaluating new incoming data. One of the methods for computing a linear function describing the dependencies in the input data is solving a linear least-squares problem in a distributed way. A concrete example is shown in Chapter 7.

1.5 Notation

Throughout this thesis we follow certain notational conventions. The number of nodes or processors in a system we denote as N or P. By d_l and $\bar{d}$ we understand the degree of node l and the average node degree of a network, respectively, and $\mathcal{N}_l$ stands for the neighborhood of node l. The neighbors of a node l are all nodes reachable from node l in one hop. Small letters n, k and m are used for number of rows or columns of a matrix or a vector.

For describing matrix algorithms or operations on matrices we usually use MATLAB notation, where $V(i,:)$ denotes ith row and $V(:,i)$ denotes ith column of a matrix V. If it is needed, we distinguish an (exact) matrix V from its numerical approximation $\hat{V}$ computed by an algorithm. By $\lambda_i(V)$ we denote the ith eigenvalue of a matrix V, where the eigenvalues are be sorted decreasingly in absolute value. Sometimes, the matrix V is omitted and we use only notation λ_i. For the condition number of a matrix V we use $\kappa(V)$.

We use the notation $i \in [a\!:\!s\!:\!b]$ for an integer i chosen from a set of numbers $i \in \{a, a+s, a+2s, \ldots, b\}$. The middle parameter s is omitted in the case where $s = 1$. The operation $i \oplus j$ stands for element-wise XOR of binary representations of i and j. Machine epsilon, also called machine precision or unit roundoff, we denote by ϵ_{mach}. We usually assume computations in double precision floating-point arithmetic, where $\epsilon_{mach} = 2^{-53}$, i. e., $\epsilon_{mach} \approx 2 \cdot 10^{-16}$.

1.6 Synopsis and Overview of Own Publications

In the first part of the thesis we review relevant theoretical background. In Chapter 2 the basics of prototypical sequential matrix algorithms are reviewed, such as QR factorization or orthogonal iteration. Chapter 3 gives an overview of the related

work including a summary of existing parallel and distributed approaches towards matrix computation. Moreover, we review gossip-based distributed data aggregation algorithms, which are important building blocks of the new distributed matrix computations. The first part of the thesis ends with Chapter 4, which summarizes the assumptions common for all introduced distributed algorithms and the simulation setup, which was used for analyzing the algorithms.

The new gossip-based distributed algorithms, *dmGS* for QR factorization and *dOI* for orthogonal iteration, are introduced and analyzed in Chapter 5 and in Chapter 6, respectively. A case study on utilizing the distributed QR factorization for solving a least-squares problem appearing in linear regression is presented in Chapter 7. Chapter 8 discusses the issues arising in implementation of gossip-based algorithms in MPI and shows first runtime comparisons to parallel algorithms. Chapter 9 concludes the thesis, summarizes the main results and outlines future work. Description of MPI terms and routines needed in Chapter 8 is summarized in Appendix A

The results presented in this thesis have appeared in several conference publications and journal papers. In Straková et al. (2011), we presented and analyzed the distributed QR factorization dmGS. A thorough comparison of the gossip-based orthogonalization to an alternative approach based on consensus is currently under revision in Sluciak et al. (2013). Preliminary results of the comparison of was provided in Sluciak et al. (2012). A new version of dmGS improved in terms of communication cost together with novel distributed eigensolver dOI based on orthogonal iteration including a distributed matrix-matrix multiplication were published in Straková and Gansterer (2013).

Besides an existing gossip-based data aggregation algorithm (push-sum, Kempe et al. (2003)), we also employ two recently developed fault tolerant distributed aggregation algorithms as a building block for distributed matrix computations. The push-flow algorithm has been analyzed in Gansterer et al. (2013), where we also discuss resilience of distributed QR factorization against node failures. First results introducing the behavior of the push-flow algorithm have been given in Gansterer et al. (2011). The push-cancel-flow algorithms, an improved version of push-flow, has been presented in Niederbrucker et al. (2012). The influence of the choice of the algorithm on the fault tolerant properties of dOI were discussed in Straková et al. (2013). Moreover, the overview of gossip-based distributed matrix algorithms was presented in Straková and Gansterer (2012).

Part I

Theoretical Background

Chapter 2

Prototypical Matrix Algorithms

All distributed matrix algorithms developed and analyzed in this thesis are based on well known sequential linear algebra algorithms. In this chapter we review the matrix algorithms relevant for the subsequent chapters.

2.1 QR Factorization

Computing QR factorization of a matrix $V \in \mathbb{R}^{n \times k}$, means finding a matrix Q with orthonormal columns and an upper-triangular matrix R, such that $V = QR$. We can compute either a *full* QR factorization with $Q \in \mathbb{R}^{n \times n}$ and $R \in \mathbb{R}^{n \times k}$ or a *thin QR factorization* (Golub and Van Loan, 1996), sometimes also called *reduced QR factorization* with rectangular $Q \in \mathbb{R}^{n \times k}$ and a square matrix $R \in \mathbb{R}^{k \times k}$. Throughout this thesis we focus on computing thin QR factorization by methods based on Gram-Schmidt orthogonalization. Another commonly used method for computing QR factorization is an approach based on Householder transformations (Golub and Van Loan, 1996).

The structure of *classical* Gram-Schmidt orthogonalization (cGS) for computing QR factorization is summarized in Algorithm 1 on page 34. One column of the matrix Q and one column of the matrix R are computed in each iteration. However, the process is numerically unstable and loss of orthogonality among the computed columns of Q may occur. Therefore, an improvement called *modified* Gram-Schmidt orthogonalization (mGS) has been introduced (see Golub and Van Loan (1996)), summarized in Algorithm 2 on page 35. In each iteration one column of the matrix Q and one row of the matrix R are computed. As the only difference in the structure of the two Gram-Schmidt orthogonalizations is the order of the computations, in exact arithmetic both would compute the same results. However, in finite precision the modified Gram-Schmidt is numerically more stable (Golub and Van Loan, 1996).

Algorithm 1 Classical Gram-Schmidt Orthogonalization

Input: $V \in \mathbb{R}^{n\times k}$, $n \geq k$

Output: $Q \in \mathbb{R}^{n\times k}$ with orthonormal columns, upper triangular $R \in \mathbb{R}^{k\times k}$

1: $R(1,1) \leftarrow \|V(:,1)\|_2$
2: $Q(:,1) \leftarrow V(:,1)/R(1,1)$
3: **for** $i = 2$ **to** n **do**
4: $\quad R(1\!:\!i-1,i) \leftarrow Q(:,1\!:\!i-1)^T V(:,i)$
5: $\quad z \leftarrow V(:,i) - Q(:,1\!:\!i-1)R(1\!:\!i-1,i)$
6: $\quad R(i,i) \leftarrow \|z\|_2$
7: $\quad Q(:,i) \leftarrow z/R(i,i)$
8: **end for**

Both Gram-Schmidt orthogonalization processes require $2nk^2$ flops for computing the QR factorization of an $n \times k$ matrix. The Householder-based method needs $2nk^2 - 2k^3/3$ flops to find Q in factored form and another $2nk^2 - 2k^3/3$ flops to explicitly compute the first k columns of Q (Golub and Van Loan, 1996; Trefethen and Bau, 1997). The matrix $\hat{Q}$ produced by the modified Gram-Schmidt satisfies (Golub and Van Loan, 1996, § 5.2.9)

$$\hat{Q}^H\hat{Q} = I + E_{MGS}, \quad \|E_{MGS}\|_2 \approx \epsilon_{\text{mach}}\kappa(V),$$

whereas for the Householder approach we have (Björck and Golub, 1967)

$$\hat{Q}^H\hat{Q} = I + E_H, \quad \|E_H\|_2 \approx \epsilon_{\text{mach}},$$

where ϵ_{mach} is the machine precision and $\kappa(V)$ is the condition number of the input matrix V. In terms of numerical properties, the Householder based orthogonalization tends to have a slight advantage over the modified Gram-Schmidt. However, in terms of operation count Gram-Schmidt orthogonalization is about twice as efficient as the Householder based approach.

2.2 Orthogonal Iteration

The orthogonal iteration (OI) algorithm (Trefethen and Bau, 1997; Golub and Van Loan, 1996), sometimes also called *subspace* or *simultaneous* iteration, is a method for computing k principal eigenvectors and eigenvalues of an input matrix $A \in \mathbb{R}^{n\times n}$. This iterative eigensolver, summarized in Algorithm 3 on page 35, starts with a given matrix $A \in \mathbb{R}^{n\times n}$ and an arbitrary matrix $Q_0 \in \mathbb{R}^{n\times k}$. Each *iteration* of the eigensolver consists of two steps – matrix-matrix multiplication $V_i = AQ_i$ (Line 3 in Algorithm 3) and a reduced QR factorization $V_i = Q_{i+1}R_{i+1}$ (Line 4 in

Algorithm 2 Modified Gram-Schmidt Orthogonalization

Input: $V \in \mathbb{R}^{n \times k}$, $n \geq k$
Output: $Q \in \mathbb{R}^{n \times k}$ with orthonormal columns, upper triangular $R \in \mathbb{R}^{k \times k}$

1: **for** $i = 1$ **to** n **do**
2: $\quad R(i,i) \leftarrow \|V(:,i)\|_2$
3: $\quad Q(:,i) \leftarrow V(:,i)/R(i,i)$
4: $\quad$ **for** $j = i+1$ **to** n **do**
5: $\quad\quad R(i,j) \leftarrow Q(:,i)^T V(:,j)$
6: $\quad\quad V(:,j) \leftarrow V(:,j) - Q(:,i)R(i,j)$
7: $\quad$ **end for**
8: **end for**

Algorithm 3), where $V_i \in \mathbb{R}^{n \times k}$, $n \geq k$, $Q_i \in \mathbb{R}^{n \times k}$ has orthogonal columns and $R_i \in \mathbb{R}^{k \times k}$ is upper triangular for each i.

It can be shown (e. g., Trefethen and Bau (1997)), that the columns of the matrices Q_i in Algorithm 3 converge to the k extreme eigenvectors of A and that the diagonal elements of the matrices R_i converge to the corresponding k eigenvalues of A under the assumption that the leading $k+1$ eigenvalues of the matrix A are distinct in absolute value, i. e., $|\lambda_1| > |\lambda_2| > ... > |\lambda_k| > |\lambda_{k+1}| \geq ... \geq |\lambda_n|$. The convergence speed of the eigenvectors Q_i corresponds to the maximum ratio between the computed eigenvalues $\max_{1 \leq i \leq k} |\lambda_{i+1}|/|\lambda_i|$.

Algorithm 3 Orthogonal Iteration (OI)

Input: $A \in \mathbb{R}^{n \times n}$, arbitrary $Q_0 \in \mathbb{R}^{n \times k}$, number of iterations t
Output: eigenvectors $Q_t \in \mathbb{R}^{n \times k}$, eigenvalues $\mathrm{diag}(R_t)$, $R_t \in \mathbb{R}^{k \times k}$

1: i = 0;
2: **repeat**
3: $\quad [V_i] = AQ_i$
4: $\quad [Q_{i+1}, R_{i+1}] = qr(V_i)$
5: $\quad i = i + 1$;
6: **until** $i == t$

2.3 Linear Regression using Least-Squares Solvers

The general purpose of linear regression is to model the relationship between several independent (*explanatory*) variables x_i, $1 \leq i \leq k$ and one dependent (*output*) variable y. When a *linear* regression is used, it is anticipated that the dependent variable can be well expressed as a linear combination of the explanatory variables. Having a set of n observations of the k independent variables $X \in \mathbb{R}^{n \times k}$ and for every observation the corresponding output variable $y \in \mathbb{R}^n$, we are interested in

coefficients w expressing the dependencies between X and y:

$$y = Xw + \epsilon,$$

where ϵ is "noise" capturing potential additional factors in the relationship between explanatory and output variables, e. g., measurement errors. When new observations of the independent variables are obtained, the linear model computed by linear regression can be used for predicting the output variables y corresponding to the new observations.

There are several methods how to estimate the parameters w in linear regression. In particular, we focus on computing linear models based on a linear least-squares solver, a method approximating a solution of an overdetermined system of linear equations. We consider an overdetermined system $Xw = y$ with $X \in \mathbb{R}^{n \times k}, k \ll n$ and $y \in \mathbb{R}^n$, where X consists in our case of n observations of the k independent variables and the corresponding output variables y. The goal is to compute a linear model $\hat{w}$, which fits "best" to the provided data X and y. More concretely, we search for a solution $\hat{w}$, where $\hat{w} = \text{argmin}_w(\|y - Xw\|^2)$.

In the following we focus on two possible solutions of a least-square problem, namely computing normal equations and utilizing QR factorization (Golub and Van Loan, 1996). Computing normal equations means to form a system of equations

$$(X^T X)w = X^T y,$$

which has only one solution, assuming that the columns of X are linearly independent. The unique solution $\hat{w}$ gives the optimal parameters of the linear model for the given least-squares problem. We refer to this algorithm as *LS-NE*.

Another approach utilizes QR factorization, which we call *LS-QR*. First it factorizes the input matrix $X = QR$ and subsequently, it solves a system of linear equations with upper-triangular matrix R

$$Rw = Q^T y,$$

yielding the desired parameters $\hat{w}$ as the unique solution of the linear system.

2.4 Graph Partitioning by Fiedler Vector

Many interesting properties of a network can be obtained by spectral analysis of the underlying graph structure. A widely used operation in network theory is graph partitioning, where the goal is to divide the underlying system into two (well-connected)

components so that the number of edges between the two components is small and both components are balanced in terms of the number of nodes. Finding such a cut in a graph can be used, for example, in social networks for finding communities or in the design of computer networks for finding a bottleneck in the communication over the network (Kempe and McSherry, 2008).

One possible method how to partition a graph is to analyze the spectrum of its Laplacian matrix. The Laplacian matrix $L \in \mathbb{R}^{N \times N}$ of a graph G with N nodes is defined as $L = D - A$, where D is the degree matrix and A is the adjacency matrix of the graph G. The degree matrix D is a diagonal matrix containing the degree of the nodes on the diagonal, i. e., $D(i, i) = d_i$ for $1 \leq i \leq N$. Consequently, the elements of the Laplacian matrix L are

$$L(i,j) = \begin{cases} d_i, & \text{if } i = j \\ -1, & \text{if } i \neq j \text{ and nodes } i \text{ and } j \text{ are neighbors} \\ 0, & \text{otherwise.} \end{cases}$$

To the second smallest eigenvalue of the Laplacian matrix it is usually referred as “Fiedler value” or “algebraic connectivity” and the corresponding eigenvector is commonly called “Fiedler vector”.

Fiedler (1973, 1975) presented a method for graph partitioning, which divides the nodes in two components based on the signs of the elements in the Fiedler vector of the Laplacian matrix. The partitioning obtained can be shown to be a “good” partitioning of a graph (cf. (Pothen et al., 1990, §2) and the references therein), i. e., the resulting graph components have similar number of nodes and the number of edges between them is small. A case study illustrating how to partition a graph in a distributed way we present in Section 6.4.1.

Chapter 3

Related Work and State-of-the-Art

Before introducing new distributed algorithms, we summarize what has been done in parallel and distributed computing so far. Besides reviewing parallel algorithms for matrix computation, we also discuss the state-of-the-art methods for tolerating failures in parallel systems.

In distributed computing we focus in particular on existing gossip-based approaches. We also describe in detail several distributed data aggregation algorithms, which are important building blocks for distributed matrix computations presented in this thesis. Furthermore, we also introduce distributed algorithms based on consensus and we explain the difference from gossip-based distributed algorithms.

3.1 Parallel Algorithms

Parallel algorithms developed for target systems such as high-performance parallel computers have been studied in the literature extensively. Parallel linear algebra algorithms relevant for this thesis, such as eigensolvers, have been implemented mainly as a part of state-of-the-art software libraries, such as ScaLapack (Blackford et al., 1997), a high-performance library for parallel distributed machines or a linear algebra library for scalable multi-core machines called Plasma (Agullo et al., 2009).

In particular, for the context of this thesis parallel algorithms for computing QR factorization (see Section 2.1) are of interest. The parallel algorithm in ScaLapack uses a block-partitioned approach, where the input matrix is distributed over processes in a 2-D block cyclic way and each block is processed separately. The algorithm for QR factorization implemented in Plasma called *tile QR factorization* (Buttari et al., 2008) provides good scalability for multicore architectures. The

input matrix is divided into small square tiles instead of rectangular panels, as in the block algorithms. Such approach achieves much finer granularity than the block approach and thus it is better suited for multicore architectures (Hadri et al., 2010).

To the latest developments certainly belong so called "communication avoiding" parallel algorithms. Demmel et al. (2012) introduced "Communication avoiding QR factorization" called *CAQR*, which is optimal (up to polylogarithmic factors) in terms of communication cost, and provided comparisons with parallel mGS (modified Gram-Schmidt orthogonalization reviewed in Section 2.1). The CAQR algorithm performs the factorization of a panel (block-columns) in parallel. The authors designed an algorithm called TSQR for computing the QR factorization of *tall and skinny matrices* (rectangular $n \times k$ matrices, where $n \gg k$) which is used for factorizing the panels. The panels themselves are divided into domains (block rows), that are factorized independently and merged using a binary tree strategy. An enhanced tile QR algorithm, combining the good scalability of the tile QR algorithm and performing orthogonalization of the individual tiles in parallel, as in CAQR, has been presented in Hadri et al. (2010). In Solomonik and Demmel (2011) 2.5D communication avoiding matrix-matrix multiplication has been introduced.

Fault Tolerant Methods in HPC

There are several approaches which are used nowadays on parallel computers for failure recovery. An entirely hardware-based approach is to execute the identical computation more times in parallel. The classical example is triple modular redundancy (Engelmann et al., 2009), where three identical systems are run in parallel and the results are determined by a two-out-of-three voting. Due to resource and energy constraints, this approach cannot cope with the challenges raised by future extreme scale systems.

A widely used fault tolerance paradigm for applications in HPC systems is checkpoint/restart (C/R), where the system state is regularly saved to a stable storage. When a failure happens and the computation cannot continue, it restarts the computation from the last saved checkpoint. Several variants of checkpointing have been introduced (Egwutuoha et al., 2013). They differ in aspects such as how the checkpointing interval (Daly, 2006) is determined, e. g., coordinated checkpointing (Buntinas et al., 2008) and uncoordinated checkpointing (Guermouche et al., 2011), or to what kind of storage is the data stored, e. g., diskless checkpointing (Plank et al., 1998). The major drawback of C/R approaches is the overhead introduced by the expensive disk I/O and consequently, for extreme-scale systems this approach will likely be unusable (Varela et al., 2010). Another property shared by C/R methods is that they require information about occurring failures, for two reasons. The

C/R mechanism has to determine the length of the interval between two checkpoints, which depends on the frequency of failures. Second, the system has to be informed about occurring failures such that the recovery procedures can be launched. Consequently, unreported soft errors like bit-flips are not tolerated by purely C/R-based solutions and require a more sophisticated logic, e. g., at the algorithmic level.

More recent approaches incorporate redundancy at the message passing level, i. e., in MPI libraries, such as RedMPI introduced in (Fiala et al., 2012). However, such approaches focus only on communication correctness, i. e., the correctness of the messages sent over MPI.

A different technique targeting the need of soft error resilience in the context of matrix operations is algorithm-based fault tolerance or ABFT (Huang and Abraham, 1984). In contrast to the previous mechanisms, ABFT focuses on failure recovery on the algorithmic level. Soft error resilience is achieved by extending the input matrices by checksums and by adapting the algorithms such that these checksums are updated correctly. Consequently, the checksums are used for detecting and recovering from soft errors. The original ABFT work (Huang and Abraham, 1984) was restricted to a single soft error and matrix-matrix multiplication. Later work extended the principal concept to handle more failures (Anfinson and Luk, 1988) and to integrate it with matrix factorizations (Luk and Park, 1988).

3.2 Distributed Algorithms

In contrast to parallel computing, much less work exists in the literature on distributed algorithms as introduced in Section 1. In fact, many publications about "distributed" algorithms make assumptions which are closer to what we call "parallel" algorithms. For example, several approaches towards "distributed" algorithms for wireless sensor networks have been introduces in Abdelhak et al. (2010, 2011). However, the assumptions used are all data initially centralized at a single node, specific roles assigned to individual nodes (distributing node, annihilating node, etc.) and a fixed communication schedule.

A different distributed approach called "local algorithms", introduced e. g., by Hasemann et al. (2012) or Suomela (2013), has several similar aspects in common with the algorithm designed in this thesis, namely proceeding in rounds and communication with nearest neighbors. However, the essential differences to distributed algorithms which we introduce are that the local algorithms require synchronized rounds and deterministic communication schedules.

3.2.1 Overview of Gossip-Based Algorithms

The most relevant literature for our context is such, which discusses utilizing randomized algorithms, especially algorithms based on gossiping (see Section 1.2). In classical gossip-based approaches nodes communicate with one of their closest neighbors, which is randomly chosen (Boyd et al., 2006). However, several protocols with modified communication schedules have been investigated, such as selecting the partners from the whole network regardless of the distance (Dimakis et al., 2008) or the nodes broadcast a message to all neighbors at once (Aysal et al., 2009).

Simple gossip-based algorithms are being due to their attractive properties used in many in-network computations, e. g., in distributed signal processing (Dimakis et al., 2010). Utilizing communication protocols based on gossiping has been investigated only for rather simple operations so far, such as information dissemination (rumor mongering), matching a query (Demers et al., 1988), summation and averaging (Kempe et al., 2003), finding rank of nodes (Montresor et al., 2008), estimating the size of a network (Kostoulas et al., 2005), clock synchronization (Iwanicki et al., 2006) or load balancing (Franceschelli et al., 2007).

The group of simple gossip-based algorithms essential for this thesis are simple protocols for computing an aggregate, in our case a sum or an average, of values distributed over a network. Throughout this thesis we denote an algorithm which computes an aggregate of numbers in a distributed and decentralized way, as *distributed data aggregation algorithm* (*DDAA*). The most important and most widely used gossip-based DDAA is push-sum (Kempe et al., 2003), an algorithm for computing the sum or average of scalars distributed over nodes. Recently developed fault-tolerant versions of the push-sum algorithm are called push-flow (Gansterer et al., 2013) and push-cancel-flow (Niederbrucker et al., 2012) and both have been extensively studied and compared in Niederbrucker and Gansterer (2013).

To the best of our knowledge, utilizing gossiping for constructing more complex algorithms, e. g., numerical linear algebra algorithms, has been investigated in the literature rarely. A few algorithms for data mining and machine learning have been introduced, such as data classification (Eyal et al., 2010) or creating a linear model over fully distributed data (Ormandi et al., 2013).

A distributed gossip-based QR factorization relevant for the context considered in this thesis has been introduced by Dumard and Riegler (2009). Their algorithm is based on *classical* Gram-Schmidt orthogonalization and they used it as the first part of a distributed sphere decoding algorithm reviewed in Section 1.4. However, Dumard and Riegler (2009) have investigated their algorithm under very restrictive assumptions. They considered only square input matrices with dimension equal to

the number of nodes in the network over a fully connected topology. Moreover, in the simulations presented in the paper they assumed that one node broadcasts its local estimate to the remaining nodes after each distributed summation. This makes the algorithm converge faster in terms of the factorization error, but a deterministic broadcasting after each push-sum would be expensive and impractical in real networks and the algorithm would lose its advantages gained from the randomized communication in push-sum.

Recently, several eigensolvers based on gossiping have been introduced. A distributed power iteration has been presented in Jelasity et al. (2007) and a distributed orthogonal iteration method in the sense defined in this paper was proposed by Kempe and McSherry (2008). We review the latter approach in detail in Section 3.2.4. Both gossip-based eigensolver are designed for the special case of computing the principal eigenvector(s) of the adjacency matrix of the underlying system. Moreover, they do *not* utilize *exclusively* randomized communication schedules, which may lead to several drawbacks, as shown in Section 6.3.

A different distributed eigensolver based on power iteration has been introduced by Korada et al. (2011). Their method is based on creating a sequence of different sparsifications of the input matrix, which means that several selected (non-zero) elements of the input matrix are set to zero. On the one hand, running the algorithm on a sparse matrix reduces significantly the communication cost per one round of the algorithm. On the other hand, it causes that the eigenvectors are not computed in full precision, but only approximated (Korada et al., 2011).

3.2.2 The Push-Sum Algorithm

The *push-sum* algorithm introduced by Kempe et al. (2003) is a simple gossip algorithm for computing an approximation of the average or the sum of values in a distributed way. Algorithm 4 on page 44 summarizes the push-sum algorithm for computing the average of values $x \in R^N$ distributed over N nodes. The input of every node l (Lines 2 and 3 in Algorithm 4) consists of the initial value $x(l)$ and a scalar weight $w(l)$, determining the type of aggregation. In every round, all nodes send their local data (the local value $v(l)$ and the weight $w(l)$) divided by two to a randomly chosen node (Lines 6 and 7 in Algorithm 4). Subsequently, all received data (Line 8 in Algorithm 4) are summed up with the local data of the receiver (Lines 9 and 10 in Algorithm 4). The final estimate in node l is computed as the local value $v(l)$ divided by the local weight $w(l)$ (Line 12 in Algorithm 4).

If each node l initializes $w(l) = 1$, as in Algorithm 4, push-sum estimates in every node the average of the initial values. A *sum* is computed, if the weights are initialized in one of the following ways. If the number of nodes N is known in each

Algorithm 4 The Push-Sum Algorithm

Input: number of nodes N and one value $x(l)$ in every node l
Output: estimate $s(l)$ of $\sum_{l=1}^{N} x(l)/N$ in every node l

1: **in each** node l **do**
2: $w(l) \leftarrow 1$
3: $v(l) \leftarrow x(l)$
4: **loop**
5: Choose randomly a *target node* t
6: Send $(v(l)/2, w(l)/2)$ to node t
7: Receive $\{(\hat{v}(r), \hat{w}(r))\}_r$
8: $v(l) \leftarrow \sum_r \hat{v}(r) + v(l)/2$
9: $w(l) \leftarrow \sum_r \hat{w}(r) + w(l)/2$
10: **end loop**
11: $s(l) = \frac{v(l)}{w(l)}$ the estimate of the average in node l
12: **end**

node, the initial weights are set to $1/N$. However, if the number of nodes N is large, the weight $1/N$ is small and this may cause numerical inaccuracies. A second possibility is to set the weight $w(l) = 1$ in one node l and in the rest of the nodes the weight is set to zero. Clearly, for the latter approach there is no need to know the size of the network in the nodes, but the nodes have to agree on which of them sets the local weight to 1.

Convergence behavior

Kempe et al. (2003) analyzed the behavior of the push-sum algorithm and proved the following theorem (under the assumption that each node can directly communicate with every other node in the network):

Theorem 3.2.1. *With probability of at least* $(1 - \delta)$ *there is a number of rounds* $i_0 = O(\log N + \log \frac{1}{\epsilon} + \log \frac{1}{\delta})$*, such that for all number of rounds* $i \geq i_0$ *in all nodes* l *the estimate of the average satisfies:*

$$\frac{1}{|\sum_j x(j)|} \left| \frac{v_i(l)}{w_i(l)} - \frac{1}{N} \sum_j x(j) \right| \leq \epsilon \frac{\sum_j |x(j)|}{|\sum_j x(j)|}, \tag{3.1}$$

where $v_i(l)$ and $w_i(l)$ is the local data of node l in round i. This theorem says that the estimates of the average computed by push-sum converge to the true average in every node and that the relative error is bounded by ϵ after certain number of rounds i_0. If the relative error of the estimate is bounded by ϵ in *all* nodes, we say that push-sum computed an ϵ-accurate estimate of the true aggregate. We refer to ϵ as to the "target accuracy" of the push-sum algorithm.

The drawback of push-sum is that its correctness relies on the so-called *mass conservation* (cf. Kempe et al. (2003)), which means that the global mass of the initial data has to be preserved during the whole computation. In other words, the equality $\sum_{l=1}^{N} x(l) = \sum_{l=1}^{N} v_i(l)$ and $\sum_{l=1}^{N} w(l) = N$ has to be preserved in each round i of the push-sum algorithm, assuming that initially $w(l) = 1,\ \forall l$. Obviously, any kind of failure, where data gets loss, such as a message loss or a node failure, violates mass conservation. Thus, the push-sum algorithm cannot guarantee correct results in the presence of such failures.

Influence of the Topology

The convergence proof by Kempe et al. (2003) states that the asymptotic number of rounds needed for computing an ϵ-accurate estimate of the true aggregate is $\mathcal{O}(\log N + \log \epsilon^{-1})$. This results holds for situations where each node can communicate directly with every other node in the network, which is equivalent to a fully connected network. Distributed systems may have implemented an overlay mechanism for emulating the fully connected network. Consequently, in such systems each node can communicate with every other node on a higher level, e. g., due to routing over the network, on an arbitrary underlying topology. However, we are interested in physical topologies of the systems, where each node communicates only with physical neighbors in the network, i. e., each node communicates only with nodes, which they can reach directly (in one hop). The only exception is executing the distributed algorithms on parallel machines (see Chapter 8), where we investigate the behavior of the computations on "virtual" topologies, where the communication partners are chosen regardless on the physical location in the system.

Clearly, the speed of spreading information across a network depends on its topology and thus, also the convergence speed of the distributed data aggregation algorithms is influenced by the underlying infrastructure. The hardware topology will be rarely a fully connected network, especially for large number of nodes. Boyd et al. (2006) provided a convergence analysis for simple distributed averaging algorithms, which communicate only with nearest neighbors. The analysis is focused on simple algorithms for computing an average (without using scalar weights), but it can be extended to more complex algorithms, such as push-sum. In the following we review the analysis as presented by Boyd et al. (2006).

In general, we can formally represent a round of a distributed averaging algorithm as a linear iteration of the form:

$$x_{i+1} = C_i x_i,$$

where x_i is the data in all nodes at round i and C_i is a doubly stochastic matrix describing the communication between the nodes in the round i. In particular, when $C_i(b,a) \neq 0$, node a sends in round i a message containing $C_i(a,b)x_i(a)$ to node b, where $x_i(a)$ is a local value in node a in round i. The matrices C_i can be viewed as random matrices drawn from a fixed distribution determined by the specific algorithm under investigation. Consequently, we can study the expected value $C_E = \mathbb{E}[C_i]$. The important property of the matrix C_E is that it reflects the underlying network topology, because $C_E(a,b) \neq 0$ if and only if nodes a and b are neighbors. Based on C_E, Boyd et al. (2006) derived an asymptotic bound of

$$\mathcal{O}\left(\frac{\log N + \log \epsilon^{-1}}{1-\lambda_2}\right)$$

on the number of rounds required to compute an ϵ-accurate estimate. This bound holds with high probability and λ_1 and λ_2 denote the two largest eigenvalues of C_E. Since the matrix C_E is doubly stochastic, its largest eigenvalue $\lambda_1 = 1$ (Boyd et al., 2006). Roughly speaking, the distance between the two eigenvalues expresses the connectivity of the underlying graph and thus, the larger the difference $1-\lambda_2$, the better connected the network. More precisely, the connectivity of a graph can be measured in several expansion properties and the value $(1-\lambda_2)$ is related to the spectral expansion (Hoory et al., 2006).

In this thesis, we are interested especially in *well-connected* graphs, where the distributed algorithms, such as push-sum, converge in $\mathcal{O}(\log N + \log \epsilon^{-1})$ rounds to an ϵ-accurate estimate. In such graphs the difference $(1-\lambda_2)$ is sufficiently large and it can be treated as constant. From Kempe et al. (2003) we know that this holds for a fully connected network. Similarly as Boyd et al. (2006) discussed the "good connectivity" of expander graphs, it can also be shown for hypercubes that $(1-\lambda_2)$ is sufficiently far away from zero. Thus, also hypercube can be treated as a well-connected topology where distributed data aggregation algorithms compute an ϵ-accurate estimate of the true aggregate in $\mathcal{O}(\log N + \log \epsilon^{-1})$ rounds. Note also that a prerequisite of a topology for fast information spreading is a short diameter.

Termination Criteria

Although Theorem 3.2.1 on page 44 proven by Kempe et al. (2003) says that the approximation in every node converges to the correct estimate, it does not give any instructions for detecting the convergence locally in individual nodes. In most of the algorithms utilizing push-sum (e. g., Dumard and Riegler (2009)) the gossip-based aggregation is terminated after a fixed number of rounds. Kempe and McSherry (2008) designed a local convergence criteria for stopping the DDAA in their dis-

tributed orthogonal iteration once the estimate is ϵ-accurate. However, the suggested termination procedure requires regular computation of a minimum and a maximum of values across all nodes. Consequently, this stopping criterion may be used for evaluation and testing purposes, but for practical use it is not suitable due to a big communication cost overhead. Although for distributed algorithms it is essential that the nodes can locally decide if their current estimate is accurate enough, local detection, which is usable in practice, remains an open research problem.

3.2.3 Fault Tolerant Gossip-Based DDAAs

As mentioned, the push-sum algorithm requires mass conservation for delivering correct results and thus, it does not tolerate any data loss (Section 3.2.2). Several new gossip-based aggregation algorithms improving the fault tolerant properties of push-sum have been developed (e. g., Eyal et al. (2012)). The most recent ones are the *push-flow* and the *push-cancel-flow* algorithms, introduced in Gansterer et al. (2013) and Niederbrucker et al. (2012), respectively. The main advantage of those two algorithms is that they preserve the good properties of the push-sum algorithm, such as logarithmic scaling of rounds with increasing number of nodes on well-connected networks (see Section 3.2.2). The three DDAAs, push-sum, push-flow and push-cancel-flow are equivalent in failure-free environments.

The basic idea behind the push-flow algorithm (see Algorithm 5 on page 48) is to utilize the graph theoretical flow concept in order to guarantee fault tolerant message exchange. Each node l keeps for every neighbor j a *flow variable* $f(l, j)$, which captures the data "flowing" from node l to its neighbor j. By introducing those flow variables, the global mass conservation necessary in push-sum translates into a local flow conservation $f(l, j) = -f(j, l)$, which is easier to test and preserve. Due to this modification, the push-flow tolerates certain kinds of permanent link and node failures as well as silent errors like message loss or bit-flips.

Several practical drawbacks of the push-flow algorithm, e. g., in terms of numerical accuracy, were observed and summarized in Niederbrucker and Gansterer (2013). As a result, a new improved version, the *push-cancel-flow algorithm* (Niederbrucker et al., 2012) has been introduced. It is an extension of the push-flow algorithm improving the numerical accuracy and computational efficiency (when system failures occur). The main modification compared to push-flow is that the flow variables converge also to the target aggregate instead to an arbitrary value. For all details about the push-cancel-flow algorithm see Niederbrucker et al. (2012).

Algorithm 5 The Push-Flow Algorithm

Input: Local initial data $x(l)$ and local weight $w(l)$ in each node l
Output: Estimate of $\sum_k x(k)/\sum_k w(k)$ in each node l

1: **in each** node l **do**
2: $v(l,:) \leftarrow (x(l), w(l))$
3: **for each** $j \in \mathcal{N}_l$
4: $f(l,j) \leftarrow (0,0)$
5: **end**
6: **loop**
7: **for each** received pair $f(j,l)$
8: $f(l,j) \leftarrow -f(j,l)$
9: **end**
10: $t \leftarrow$ choose a random neighbor $t \in \mathcal{N}_l$
11: $s(l,:) \leftarrow v(l,:) - \sum_{j \in \mathcal{N}_l} f(l,j)$
12: $f(l,t) \leftarrow f(l,t) + s(l,:)./2$
13: send $f(l,t)$ to node t
14: **end loop**
15: $s(l,1)/s(l,2)$ the estimate of the average in node l
16: **end**

3.2.4 Distributed Orthogonal Iteration by Kempe and McSherry

Kempe and McSherry (2008) introduced a distributed eigensolver based on orthogonal iteration (see Section 2.2). Throughout this thesis we denote this distributed algorithm as *KOI*. The KOI algorithm is designed for computing k principal eigenvectors of the (weighted) *adjacency matrix* $A \in \mathbb{R}^{n \times n}$ of the underlying network with $N = n$ nodes. KOI assumes both matrices $A \in \mathbb{R}^{n \times n}$ and $Q_i \in \mathbb{R}^{n \times k}$ initially distributed row-wise across the nodes in every iteration i. The KOI algorithm is summarized in Algorithm 6 on page 49, where steps highlighted in red are processed in a distributed way, the rest is computed locally in nodes.

The first step of KOI, a matrix-matrix multiplication (Line 4 in Algorithm 6), uses the assumption that the input matrix is the (weighted) adjacency matrix of the underlying network, i. e., $A(j,l) = 0$, whenever j and l are not adjacent. In order to compute the row $V_i(:,l) = A(:,l) \cdot Q_i$ in iteration i, each node l needs to obtain all rows $Q_i(:,j)$, where $A(j,l) \neq 0$. If the input matrix A is the (weighted) adjacency matrix of the underlying network, each node l needs to gather exactly the rows of Q_i, which are stored in its direct neighbors (Line 3 in Algorithm 6). Therefore, if only communication with nearest neighbors is allowed, KOI works only for matrices with a structure corresponding to the adjacency matrices of the underlying system. As a result of the deterministic communication schedule, the matrix product $V_i = AQ_i$ in

each iteration i is computed in full precision, but each node needs to communicate with all neighbors in order to complete the computation.

Algorithm 6 Distributed Orthogonal Iteration KOI

Input: $A \in \mathbb{R}^{n \times n}$, arbitrary $Q_0 \in \mathbb{R}^{n \times k}$, number of iterations t
Output: Eigenvectors $Q_t \in \mathbb{R}^{n \times k}$, eigenvalues $\text{diag}(R_t)$, $R_t \in \mathbb{R}^{k \times k}$
Data Distribution: A, all Q_i and V_i distributed row-wise for all i

1: i = 0;
2: **repeat (in each node l)**
3: receive $Q(j,:)$ from each $j \in \mathcal{N}(l)$
4: $[V_i(l,:)] = A(l,:) \cdot Q_i$
5: $S^{(l)} = V_i^T(l,:) \cdot V_i(l,:)$
6: $K_i^{(l)} = \text{DDAA-KOI}(S^{(j)})$ (consensus-based distributed sum)
7: $K_i^{(l)} = (R_i^{(l)})^T R_i^{(l)}$ (Cholesky factorization)
8: $Q_{i+1}(l,:) = V_i(l,:) \cdot (R_i^{(l)})^{-1}$ (linear solver)
9: $i = i + 1$;
10: **until** $i == t$

The next steps of the KOI algorithm are designed in order to sidestep the QR factorization algorithm, because a distributed QR factorization algorithm was not available to the authors. Instead, the QR factorization is replaced with a mathematically equivalent sequence of a local outer product of vectors (Line 5 in Algorithm 6), distributed summing of matrices (Line 6 in Algorithm 6), local Cholesky factorization (Line 7 in Algorithm 6) and local solving of a linear system (Line 8 in Algorithm 6). The distributed summing of matrices in KOI is also done by utilizing a DDAA, for which the authors also use the term "push-sum" algorithm. However, in our understanding their approach actually uses a form of *consensus* algorithm, because of the following key property. Nodes in the DDAA used in KOI (Line 6 in Algorithm 6) do *not* communicate only with a single randomly chosen neighbor, but with *all* neighbors at once, which besides availability of connections to all neighbors at all times also requires synchronization across nodes (Section 3.2.5). We denote this DDAA algorithm used in KOI as *DDAA-KOI*.

3.2.5 Consensus-Based Algorithms

A different group of distributed data aggregation algorithms are approaches based on *average consensus* (Olfati-Saber et al., 2007). Similarly to the push-sum algorithm, the goal of the consensus algorithm is to compute in each node l an approximation of the average $\sum_{l=1}^{N} x(l)/N$ of the initial data $x \in \mathbb{R}^N$ distributed across N nodes. Consensus-based algorithms also work in rounds and iteratively refine the local estimates. The main difference from the gossip-based approaches is that the nodes

communicate with *all* neighbors in each round and thus, this algorithm requires full synchronization of nodes in order to deliver correct results (Denantes et al., 2008).

In this thesis we distinguish between two versions of consensus, namely static and dynamic average consensus. In the *static* consensus, the initial data in the nodes $x_0 = x$ to be summed up stay for the whole computation constant. Consequently, also the true aggregate does not change during the computation. In every round i of the static consensus, each node l receive one message from every direct neighbors $b \in \mathcal{N}_l$ and updates its local status using the received data according to the following equation:

$$x_{i+1}(l) = W(l,l)x_i(b) + \sum_{b \in \mathcal{N}_l} W(l,b)x_i(b),$$

where $x_i(l)$ is the current estimate in node l in round i and $W \in \mathbb{R}^{N \times N}$ is a double stochastic weight matrix representing the connections in the network. The concrete selection of the weight matrix essentially influences the convergence of the algorithm (Olfati-Saber et al., 2007; Denantes et al., 2008). From a global point of view, the updating equation can be rewritten as:

$$x_{i+1} = W x_i.$$

It has been shown (Olfati-Saber et al., 2007) that the estimate in every node l converges to the average of the initial data, i. .e,

$$\lim_{i \to \infty} x_i(l) = 1/N \sum_{b=1}^{N} x_0(b)$$

in every node l.

In contrast to the static average consensus, the input of the *dynamic* average consensus is varying during the computation. The discrete dynamic consensus has been defined and analyzed in Sluciak et al. (2012). Analogously to static consensus, also in dynamic consensus each node follows an updating equation in every round i:

$$x_{i+1} = W[x_i + \Delta s_{i+1}],$$

where $s_{i+1} \in \mathbb{R}^N$ is a bounded input signal with $x_0 = s_0$ and time differences $\Delta s_{i+1} = s_{i+1} - s_i$. For the weight matrix W applies the same as in the case of the static average consensus. The estimates $x_i(l)$ in every node l converge to the average $\bar{s}_i$ of the signal s_i, i. .e.,

$$\lim_{i \to \infty} (x_i - \bar{s}_i) = 0.$$

Chapter 4

Basic Setup

First, we discuss the basic idea of constructing distributed matrix algorithms and the data distribution we consider in the thesis, because the initial data distribution is an essential part of constructing distributed and parallel algorithms. Furthermore, we describe simulation setup which we used for investigating the distributed algorithms in practice, for example the stopping criteria implemented in the simulations and the failure models used for studying the fault tolerance of the algorithms.

Another important aspect of distributed algorithms is the type of communication in the underlying hardware system: point-to-point communication between two nodes or broadcasting to all neighbors. Among loosely coupled distributed systems we easily find both types of communication. For us, the type of communication is important mainly for analyzing the algorithms in terms of number of messages sent.

4.1 Construction of Distributed Matrix Algorithms

All distributed matrix algorithms developed in this thesis are based on gossip-based distributed data aggregation algorithms (see Section 3.2.1). The important property of the matrix algorithms is that the nodes communicate *only* during the DDAAs and therefore, all communication between the nodes is flexible, which brings several advantages over existing approaches with (partially) deterministic information exchange (see Chapter 6). We explain how to compute the 2-norm of a vector and the dot product of two vectors in a distributed way using simple gossip-based data aggregation algorithms. This principle has been also utilized by Dumard and Riegler (2009) in distributed sphere decoding.

Distributed data aggregation algorithms can be seen as distributed versions of all-to-all reduction operations, in particular for summing or for averaging the elements of a long vector distributed over many nodes. Consequently, distributed versions

of basically all types of BLAS operations can potentially be constructed based on DDAAs. We describe the following principle for a general DDAA, but throughout this thesis we focus especially on gossip-based DDAAs, such as the push-sum algorithm (see Section 3.2.2).

The 2-norm of a vector $a \in \mathbb{R}^N$ is defined as follows:

$$\|a\|_2 = \sqrt{\sum_{l=1}^{N} a(l)^2}.$$

Assume that each element $a(l)$ of the vector a is stored in a different node l and that the goal is to compute the 2-norm of the vector a in a distributed way. First, each node l computes square of the element $a(l)$ locally (Line 2 in Algorithm 7). Then, the system computes an approximation $s(l)$ of the sum $\sum_{l=1}^{N} a(l)^2$ in each node l using a distributed aggregation algorithm (Line 3 in Algorithm 7). Finally, by locally computing a square root of the value $s(l)$, each node l obtains an estimate $dn(l)$ of the desired 2-norm (Line 4 in Algorithm 7).

Algorithm 7 Distributed Computing of the 2-Norm

Input: Number of nodes N, an element $a(l)$ in every node l
Output: Estimate $dn(l)$ of $\|a\|_2$ in every node l

1: **in each node** l **do**
2: $x(l) = a(l)^2$ ... local
3: $s(l) =$ **DDAA**(x) ... distributed
4: $dn(l) = \sqrt{s(l)}$ ... local
5: **end for**

The dot product of two vectors $a \in \mathbb{R}^N$ and $b \in \mathbb{R}^N$ is defined as follows:

$$a^T b = \sum_{l=1}^{N} a(l)b(l).$$

Analogously to the 2-norm, we assume that the vectors are distributed element-wise across the network, i. e., each node l contains the corresponding elements $a(l)$ and $b(l)$ of the two input vectors, and the network computes the dot product in a distributed way (see Algorithm 8 on page 53). In the first step, a scalar product $x(l) = a(l)b(l)$ is computed locally in nodes (Line 2 in Algorithm 8). Then an approximation $dp(l)$ of the dot product is obtained in every node l (Line 3 in Algorithm 8) by a distributed algorithm summing the values $x(l)$ over the network.

For the approach, where each node stores stores one element of the input vector $x(l)$ and the goal is to compute the sum of all elements of vector $x \in \mathbb{R}^N$ across

Algorithm 8 Distributed Computing of the Dot Product

Input: Number of nodes N, vector elements $a(l)$ and $b(l)$ in every node l
Output: Estimate $dp(l)$ of $a^T b$ in every node l

1: **in each node** l **do**
2: $x(l) = a(l)b(l)$ ... local
3: $dp(l) =$ **DDAA**(x) ... distributed
4: **end for**

N nodes, we use the term *scalar* distributed aggregation algorithms. For the push-sum algorithm, which is the main building block considered in this research project, it means that all exchanged messages contain one scalar value $x(l)$ and one scalar weight $w(l)$ (see Section 3.2.2). However, we also have to consider different scenarios.

First of all we look at the size of the messages exchanged. For constructing distributed matrix algorithms we also utilize a *vector* DDAA, where each node l stores a vector $v^{(l)} \in \mathbb{R}^k$ and the goal is to compute the element-wise sum $s = \sum_{l=1}^{N} v^{(l)},\ s \in \mathbb{R}^k$ of all vectors $v^{(l)}$ across the network. Note that vector DDAA computes the same result as scalar DDAAs run k-times on every element of the vectors separately. In the vector push-sum algorithm the messages exchanged contain a vector of length k and a scalar weight. Analogously, also a *matrix* DDAA can be constructed, where the DDAA sums matrices stored in nodes element-wise across a network.

Another situation we have to take into considerations is where the number of nodes N is smaller than the number of elements in the input vector. Let's have a closer look at a simple situation a vector $v \in \mathbb{R}^n$ is distributed over N $N < n$ nodes. Thus, a node in the network may store more then one element of the input vector $x \in \mathbb{R}^n$. The goal is to compute the sum $s = \sum_{i=1}^{n} x(i)$ by a single call of a DDAA. First, each node l locally computes the sum $\sum_{i=l_1}^{l_{r_l}} v(i)$ of all r_l locally stored values $x(l_1), ..., x(l_{r_l})$. Subsequently, a data aggregation algorithm computes a sum of the local partial sums. After the DDAA has finished, each node holds an approximation of the sum s of all elements of the vector $x \in \mathbb{R}^n$. Since each node sends one message with one value and one scalar weight in every round regardless if $N = n$ or $N < n$, the number of messages for a single call of a DDAA over a fixed topology is the same in both cases. However, more values per node may increase the amount of local computation and therefore the runtime, and bigger capacity of the local memory to store the input data is required.

4.2 Data Distribution

The initial distribution of the input data is a relevant aspect which determines the overall performance of a distributed algorithm, e. g., in terms of communication cost or scalability. Since this thesis deals with distributed matrix algorithms, we discuss data distribution of vectors and matrices. Parallel algorithms are usually implemented for a certain data distribution which optimizes the performance of the particular algorithm. The concrete data distribution depends on the implementation, but very common is a two-dimensional block-cyclic data distribution, used for instance in ScaLAPACK (Blackford et al. (1997)).

In contrast to parallel algorithms, the data in distributed algorithms are usually generated directly in nodes prior to the computation. The input data in decentralized loosely-coupled networks, such as sensor networks or P2P networks, often consists of knowledge about the nearest neighbors or of measurements from the environment. The data stored in nodes may be, for instance, the corresponding rows of a (weighted) adjacency matrix or of a channel matrix (see Dumard and Riegler (2009)), or measurements and observations used, e. g., in various data mining tasks. In most cases, the data stored locally in nodes represents blocks of entire rows or columns of a global matrix, which is the input of a distributed matrix algorithm.

As a consequence, all of the distributed algorithms developed and analyzed in this thesis assume row-wise or column-wise distribution of the input matrix $V \in \mathbb{R}^{n \times k}$ across N nodes, where $N \leq n$ if the matrix is distributed row-wise or $N \leq k$ in case of a column-wise distributed matrix. We do not explicitly discuss the situation when $N > n$ (or $N > k$), because in that case the number of nodes participating on the computation can be reduced to $N' = n < N$ (or $N' = k < N$). In the following, we discuss only situation where the matrix V is distributed row-wise. The same arguments as presented below can be applied also for a situation when the input matrix $V \in \mathbb{R}^{n \times k}$ is distributed column-wise.

Although the distributed approaches work for the general case $N \leq n$, in many concrete applications, e. g., working on the adjacency matrices of the network, it holds that every node stores one row or one column of V. As a consequence and also for the sake of clarity, we usually introduce a new distributed algorithm for the situation where $n = N$ and thus each node l contains exactly one row $V(l, :)$ of the input matrix $V \in \mathbb{R}^{n \times k}$.

Situations, where each node stores more rows or columns of the input matrix may appear, for instance, in various data mining applications, where each node contains several sets of observations, measurements or evaluations. Although the number of messages sent in a DDAA is the same regardless if $n = N$ or $n < N$, we showed in

Section 4.1, for a distributed matrix computation the situation may be slightly more complicated. If the number of DDAAs executed during a matrix algorithm depends on the size of the input problem, the number of messages is not the same for the cases where $n = N$ or $n < N$. And even if the number of messages odes not change, the size of the messages may increase with the problem size. The concrete communication cost and scalability is analyzed for each introduced distributed algorithm separately.

Although we usually present the newly developed distributed matrix algorithms for the case when $n = N$, it is important to point out that all of them for a general case when $N \leq n$. We implemented distributed algorithms for the case when $N < n$ and we show experimental results for such situations.

4.3 Simulation Setup

Although we present the distributed matrix algorithms with a general DDAA as a building block, for investigating them theoretically as well as experimentally we usually employ the push-sum algorithm (Section 3.2.2). If not stated otherwise, the elements of the input vectors and matrices are generated randomly with uniform distribution from the interval $[0, 1]$.

We utilize MATLAB for investigating the influence of various aspects, such as the target accuracy ϵ of the instances of the DDAA, various kinds of failures and the underlying topology of the network on the final accuracy. The simulations in MATLAB are implemented in synchronous rounds, although full synchronization of rounds in push-sum is not required (Bertsekas and Tsitsiklis, 1989).

To investigate the impact of asynchrony, we provide simulation results from a network-based simulator called ns-3. In order to compare distributed algorithms to existing parallel approaches in terms of runtime, we implement gossip-based algorithms also in MPI. Below we summarize the simulation setup for MATLAB simulations. For the details on MPI implementations and experiments, see Chapter 8.

4.3.1 Considered Topologies

In order to investigate the behavior of distributed algorithms over different network topologies, each distributed algorithm receives at input either the adjacency matrix corresponding to the investigated topology or the list of neighbors of every node. We consider the following network topologies:

(i) *Fully connected* topology, where each node is connected with every other node in the network. In other words, from each node is every other node in the network directly reachable by sending one message.

(*ii*) *Hypercube* topology with $N = 2^h$, $h \in \mathbb{N}$ nodes. In this topology each node l has exactly h neighbors which can be computed as $l \oplus 2^i$, with $1 \leq i \leq h = \log_2 N$, where $\oplus$ denotes the binary XOR.

(*iii*) *Random* or *random geometric* topology, which corresponds to a model of a wireless sensor network. Each node is randomly deployed on the unit square and it is connected to all nodes in a given radius ρ (*communication range*).

(*iv*) *Random regular* topology with the degree d, where each node is connected to randomly selected d nodes from the network.

4.3.2 Termination Conditions

Although the gossip-based DDAAs, such as push-sum, refine the approximations iteratively, efficient convergence criterion determining the accuracy of the current estimate locally in nodes remains an open research question (Section 3.2.2).

In our simulations of the distributed algorithms we assume for testing purposes that each node has locally available the true aggregate of the distributed values prior to the computation. Each DDAA is called with an input parameter ϵ determining the target accuracy of the approximation. In every round, all nodes check the current error of their local estimate relatively to the true result and when the estimate is ϵ-accurate in all nodes, the computation stops. To avoid infinite loops, if the algorithm does not converge or does not reach ϵ, we also use an additional input parameter `MAX_ROUNDS` determining the maximum number of rounds to be executed. We set the maximum number of rounds in the simulations high enough so that it does not influence the achieved accuracy of the instance of the DDAA. In other words, if a target accuracy is set to $\epsilon = 10^{-15}$ and the algorithm computes a result with error 10^{-10}, it is not caused by an insufficient number of rounds, but by inferior numerical properties of the simulated algorithm.

4.3.3 Types of Failures

As summarized in Section 3.1, failures can be handled at different levels. We are interested in recovering from any type of failures purely on the algorithmic level, e. g., investigating the utilization of self-healing algorithms, such as push-flow and push-cancel-flow (Section 3.2.3).

When investigating resilience of an algorithm, it is important to specify the types of failures, which may happen during the computation. In this thesis we follow the classification of failure types, which we established in Gansterer et al. (2013). The types of failures are ordered according to increasing difficulty for recovering from them at the algorithmic level.

1. Reported temporary unavailability of links/nodes
2. Unreported (silent) loss or corruption of a message
3. Reported permanent node or link failures
4. Unreported (silent) corruption of local data (e. g., bit flip)
5. Unreported (silent) permanent node failures

Under *reported* failures we mean such events, which are signaled to the affected nodes immediately, e. g., to all neighbors of a failed node. In contrast, no node is explicitly informed about a *unreported* or *silent* failure and thus it cannot be assumed that the nodes are aware of those failures. Failures which are reported but with a delay we treat as unreported, because in the meanwhile the status in the network has changed.

For investigating the resilience properties of distributed algorithms in the presence of failures, we introduce a failure rate p determining the probability that a communication link or a transmission of a message fails. Before sending every message it is decided, independently on the other transmissions, if a failure occurred with probability p or the message was successfully delivered and processed by the receiving node.

Permanent node failures, reported or unreported, are very difficult to handle even by a self-healing algorithm, because loss of data during a computation always changes the result. If there are no special mechanisms for dealing with the node failures, the original data stored locally in the failed node is irretrievably lost. However, we have to distinguish several scenarios. In distributed algorithms which exclude the lost data from the final result and which are expected to deliver an approximation corresponding only to the values in the remaining nodes, is the situation rather easy. Especially because no back-up of the local data is needed and we only have to assure that the distributed algorithm dynamically adjusts to the new values stored across the nodes and converges to the new result. A more difficult situation comes when the *original data* of *all* nodes has to be aggregated. In this case no input data must be lost and thus, redundancy of the initial data stored in the system has to be introduced, which is discussed in Section 5.5.2.

4.4 Point-to-point Communication vs. Broadcasting

There are essential differences between systems using point-to-point communication and those using broadcasting especially for analyzing the communication cost. In

systems with point-to-point communication, every node needs to send one message to every neighbor, whereas in systems with broadcasting, one message is enough to reach all neighbors at once.

Distributed algorithms, which were designed primarily for wireless networks using broadcasting, are usually based on consensus algorithms (see Section 3.2.5). In consensus-based algorithms the nodes do not choose one neighbor to send information to, but they broadcast their current status to all nodes within a certain communication range. It is assumed that the nodes know the number of its neighbors, because the weight matrix W used for updating the local statuses is built accordingly. The main limitation of consensus-based algorithms is that a full synchronization of the nodes is required in order to compute a correct result (see Olfati-Saber et al. (2007); Denantes et al. (2008)).

In contrast, gossip-based algorithms (Section 3.2.1) are better suited for applications in networks with point-to-point communication due to the choice of communication partners. However, in Aysal et al. (2009) an algorithm called "broadcast gossip" has been introduced. Instead of selecting one communication partner, the nodes in each round broadcast messages to all neighbors. In contrast to the consensus-based algorithms, the nodes do not assume any knowledge about the number of neighbors. More precisely, node l broadcasts in round i its current value $x_i(l)$ to all nodes p in a certain communication range. All receivers p update their status after receiving the message according the following formula:

$$x_{i+1}(p) = \gamma x_i(p) + (1-\gamma)x_i(p),$$

where $\gamma \in (0,1)$ is a so called "mixing parameter". According to Aysal et al. (2009), the best option is to choose $\gamma = 0.5$. Due to the missing knowledge of the number of neighbors, the estimates obtained through broadcast gossip converge to the true aggregate of the initial values in expectation, as shown in Aysal et al. (2009). In other words, the output of the distributed algorithm can be viewed as a random variable, where the expected value is the desired aggregate. However, the accuracy of the concrete output of the algorithm is not guaranteed, since this random variable has a non-zero variance (Aysal et al., 2009).

We implemented the broadcast gossip algorithm in MATLAB and investigated the accuracy of the estimates $\hat{s}$ of the true sum $s = \sum_{i=1}^{N} x(i)$, where a vector $x \in \mathbb{R}^{128}$ is distributed across a hypercube with $N = 2^7$ nodes. Although the local estimates computed in all nodes by the algorithm converge eventually to the same value, as illustrated in Figure 4.1 on page 59, the final accuracy of the estimates computed by the wireless gossip is too low for a practical use (see Figure 4.2 on page 59).

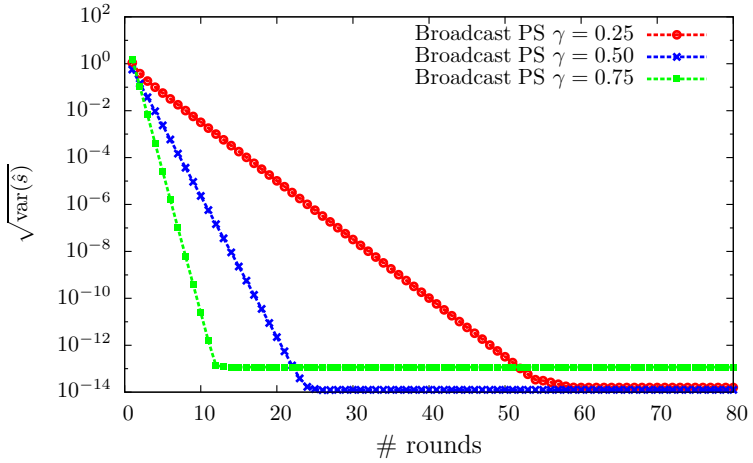

Figure 4.1: Standard deviation of estimates computed by the broadcast gossip algorithm with different mixing parameters γ. The simulations computed an estimate $\hat{s}$ of the sum s of values of a random vector $x \in R^{128}$ over a hypercube with $N = 2^7$ nodes. The figure shows results averaged over 50 different random vectors x.

To conclude, the broadcast gossip algorithm is intended for systems which allow for broadcasting, but it does not guarantee accurate results. Therefore, this concept is not suitable for our distributed algorithms, since we are interested in computations in high accuracy and thus we focus on developing gossip-based algorithms with point-to-point communication.

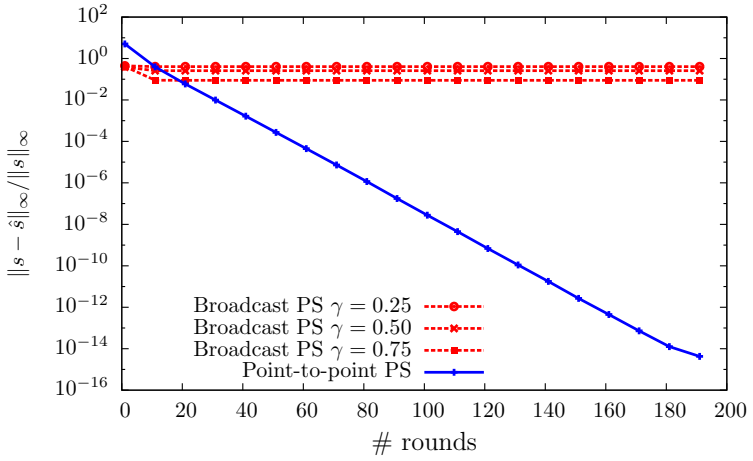

Figure 4.2: Relative error of results computed by the push-sum algorithm and by the broadcast gossip algorithm with different mixing parameters γ. The simulations computed an estimate $\hat{s}$ of the sum s of values of a random vector $x \in R^{128}$ over a hypercube with $N = 2^7$ nodes. The figure shows results averaged over 50 different random vectors x.

Part II

New Distributed Algorithms

Chapter 5

Distributed QR Factorization

In this chapter we discuss the potential of gossip-based algorithms for computing QR factorization in a distributed manner. We introduce a new distributed algorithm *dmGS* based on *modified* Gram-Schmidt orthogonalization and we provide a comprehensive analysis of various aspects of dmGS, such as numerical accuracy, scalability with increasing network and problem size and convergence behavior for varying degrees of synchronization. Furthermore, we also compare dmGS to parallel state-of-the-art algorithms and to distributed QR factorization based on consensus.

The results summarized in this chapter appeared in several publications. First version of the dmGS algorithm based on push-sum has been presented and analyzed Straková et al. (2011), where also an analytical comparison to parallel algorithms has been provided (see Section 5.4). An improved version of the dmGS algorithm in terms of scalability (see Section 5.3) was described in Straková and Gansterer (2013). In Gansterer et al. (2011) and Gansterer et al. (2013) we developed a robust version of dmGS tolerating node failures (see Section 5.5). In Section 5.7 we provide a comparison to a consensus-based distributed QR factorization, which we developed and analyzed in a joint publication Sluciak et al. (2012) with project partners. The related work relevant for this chapter can be found in Chapter 3.

5.1 Distributed QR Factorization *dmGS*

We consider the problem of computing QR factorization of a matrix $V \in \mathbb{R}^{n\times k}$, $n \geq k$, where the input matrix V is assumed to be initially distributed row-wise among N nodes of a decentralized distributed system (as discussed in Section 4.2). In particular, we are interested in modified Gram-Schmidt orthogonalization computing the *thin* QR factorization of V, i. e., finding a matrix $Q \in \mathbb{R}^{n\times k}$ with orthonormal columns and an upper-triangular matrix $R \in \mathbb{R}^{k\times k}$, such that $V = QR$. The goal is

to design and analyze QR factorization which proceeds in a distributed decentralized way and is based exclusively on randomized communication schedules.

The new distributed modified Gram-Schmidt orthogonalization algorithm *dmGS* is presented in the right part of Algorithm 9 on page 65. On the left side we can see sequential modified Gram-Schmidt orthogonalization (Section 2.1) to illustrate how the distributed algorithm was constructed. The sums marked in red color were replaced by a distributed data aggregation algorithm, once in the dot product (Lines 8-9 Algorithm 9) and once in the 2-norm (Lines 2-5 in Algorithm 9) (cf. Section 4.1). Note that nodes in dmGS communicate and exchange information only during the calls of the DDAA (Lines 4 and 11 in Algorithm 9). The rest of the algorithm contains only computation on local data in individual nodes.

The dmGS algorithm is described in Algorithm 9 for the special case when $n = N$ and thus each node stores exactly one row of the input matrix V. Note that this assumption is not required and that the algorithm works for a general situation $N \leq n$. If the input matrix has more rows than the number of nodes in the network, each node l stores $r_l \geq 1$ consecutive rows of the matrix V. Before every distributed summation in computing the 2-norm (Line 3 and 9 in Algorithm 9) and before computing the dot product (Line 9 in Algorithm 9), each node l locally computes a sum $x(l) = \sum_{t=1}^{r_l} V(l_t, i)^2$ and $x(l) = \sum_{t=1}^{r_l} Q(l_t, i)V(l_t, j)$, respectively. The rest of dmGS stays unchanged.

Since each node computes the rows of Q corresponding to the local rows of A (Line 5 in Algorithm 9), the output matrix Q is distributed also row-wise across the network. The second output matrix R is distributed column-wise over nodes 1 to k, if dmGS proceeds as described in Algorithm 9. Nevertheless, during dmGS each node automatically computes an estimate of every element of the matrix R (Lines 4 and 9 in Algorithm 9). Thus, if required by the application and if the local storage is large enough, each node l can keep an estimate $R^{(l)}$ of the entire matrix R by skipping Lines 6 and 11 in Algorithm 9.

Analogously, distributed *classical* Gram-Schmidt orthogonalization based exclusively on gossip-based DDAAs and using point-to-point communication can be built (Dumard and Riegler (2009)). Throughout this chapter we denote a fully distributed QR factorization based on classical Gram-Schmidt orthogonalization as *dcGS*.

5.2 Numerical Accuracy

In this section we address the question of preserving numerical properties of modified Gram-Schmidt orthogonalization in a distributed setup. Although analysis of numerical properties of the Gram-Schmidt orthogonalization method have been

Algorithm 9 Modified Gram-Schmidt Orthogonalization (mGS) and Distributed QR Factorization (dmGS)

Input: $V \in \mathbb{R}^{n \times k}$, $n \geq k$ (in dmGS matrix V distributed row-wise)

Output: $Q \in \mathbb{R}^{n \times k}$, $R \in \mathbb{R}^{k \times k}$ (in dmGS matrix Q distributed row-wise, matrix R distributed column-wise)

```
1:  for i ← 1 to k do
2:     x(l) ← V(l,i)^2
3:     s ← Σ_{l=1}^{n} x(l)
4:     R(i,i) ← √s
5:     Q(:,i) ← V(:,i)/R(i,i)
6:
7:     for j ← i+1 to k do
8:        x(l) ← Q(l,i)V(l,j)
9:        R(i,j) ← Σ_{l=1}^{n} x(l)
10:       V(:,j) ← V(:,j) − Q(:,i)R(i,j)
11:
12:    end for
13: end for
```

```
1:  for i ← 1 to k do (in node l)
2:     x(l) ← V(l,i)^2
3:     s(l) ← DDAA(x)
4:     R_l(i,i) ← √s(l)
5:     Q(l,i) ← V(l,i)/R_l(i,i)
6:     if l ≠ i delete R_l(i,i)
7:     for j ← i+1 to k do
8:        x(l) ← Q(l,i)V(l,j)
9:        R_l(i,j) ← DDAA(x)
10:       V(l,j) ← V(l,j) − Q(l,i)R_l(i,j)
11:       if l ≠ j delete R_l(i,j)
12:    end for
13: end for
```

provided extensively for the sequential case (Golub and Van Loan, 1996; Higham, 2002), we are interested in studying the impact of replacing several operations by their distributed counterparts.

For investigating the behavior of the algorithms we use two metrics: the relative factorization error $\|V - QR\|_2 / \|V\|_2$ and the orthogonality $\|I - Q^T Q\|_2$ of the matrix Q, where I denotes the identity matrix. If not stated otherwise, distributed matrix algorithms use push-sum as the basic building block. In the simulations, gossip-based data aggregation algorithms terminate after reaching the target accuracy ϵ, or executing the maximum number of rounds, as described in Section 4.3.2.

5.2.1 Numerical Stability of dmGS vs. dcGS

In theory, modified Gram-Schmidt orthogonalization has much better numerical properties than classical Gram-Schmidt orthogonalization (Higham, 2002, Section 19.8), both approaches are reviewed in Section 2.1. This fact motivated us to investigate distributed *modified* Gram-Schmidt orthogonalization in contrast to the existing distributed *classical* Gram-Schmidt orthogonalization introduced by Dumard and Riegler (2009). However, this effort is justified only if the properties of both Gram-Schmidt orthogonalization methods are also preserved in their distributed versions.

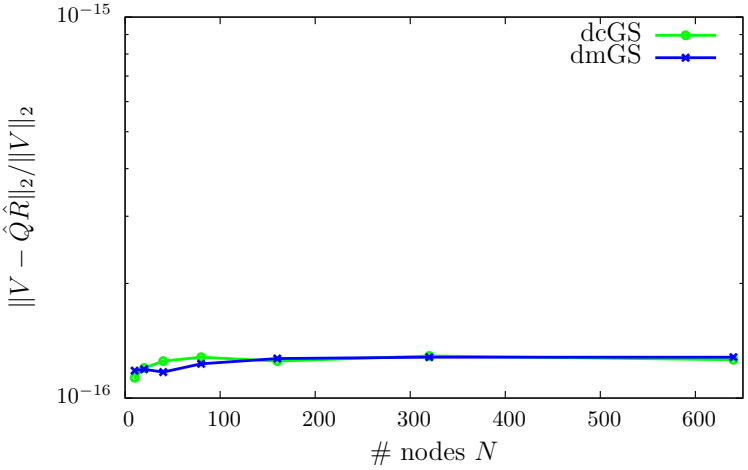

Figure 5.1: Factorization error of results computed by distributed classical (dcGS) and distributed modified (dmGS) Gram-Schmidt orthogonalization factorizing a matrix $V^{640\times640}$, $\kappa(V) = 2,43 \cdot 10^4$ over fully connected networks with N nodes.

We compared the dmGS and dcGS algorithms for varying number of nodes N factorizing a matrix $V \in \mathbb{R}^{640\times640}$ over a fully connected topology with the target accuracy $\epsilon = 10^{-15}$ of the push-sum algorithm. As we can see in Figure 5.1, factorization error of the matrices $\hat{Q}$ and $\hat{R}$ computed by dcGS and dmGS is almost identical for both approaches. In contrast, the final orthogonality of the matrix $\hat{Q}$ computed by dmGS is substantially better compared to the result delivered by dcGS. In Figure 5.2 we see that the achieved orthogonality differs by more than two orders of magnitude for different network sizes for a matrix with a condition number of order 10^4.

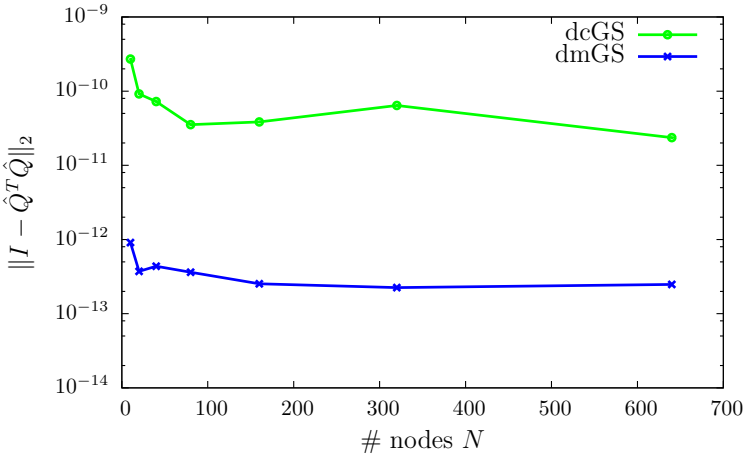

Figure 5.2: Orthogonality of $\hat{Q}$ computed by distributed classical (dcGS) and distributed modified (dmGS) Gram-Schmidt orthogonalization factorizing a matrix $V \in \mathbb{R}^{640\times640}$, $\kappa(V) = 2,43 \cdot 10^4$ over fully connected networks with N nodes.

More extensive testing of the influence of the condition number on the final results computed by dcGS and dmGS are presented at the end of this chapter in Section 5.7.1, where the methods are also compared to consensus-based approaches. All simulation results indicate that the distributed Gram-Schmidt orthogonalization algorithms behave in compliance with the theory for sequential algorithms..

5.2.2 Choice of the DDAA

So far, we studied only behavior of distributed algorithms based on the push-sum algorithm. Because later in this thesis we also employ fault-tolerant gossip-based DDAAs (see Section 3.2.3), in this section we address the questions related to impact of the concrete DDAA on the final accuracy of the distributed matrix algorithm built on top of it. We compare the numerical accuracy of the distributed QR factorization dmGS when employing the push-sum algorithm (PS), the push-flow algorithm (PF) and the push-cancel-flow algorithm (PCF). The respective distributed modified Gram-Schmidt orthogonalization algorithms are denoted by dmGS(PS), dmGS(PF) and dmGS(PCF).

We simulated dmGS when factorizing a random matrix $V \in \mathbb{R}^{2048\times 32}$ over a hypercube with $N = 2^h$, $h = 5, \ldots, 11$ nodes. Every DDAA runs as long as the local estimates do not reach the target accuracy $\epsilon = 10^{-15}$ or the maximum number of rounds has not been executed (Section 4.3.2). In Figure 5.3 on page 68 we notice that dmGS(PF) delivers a significantly worse result in terms of the factorization error $\|V - \hat{Q}\hat{R}\|_2/\|V\|_2$ than the other two variants, dmGS(PS) and dmGS(PCF). The influence of the choice of the DDAA on the orthogonality $\|I - \hat{Q}^T\hat{Q}\|_2$ is very similar, as plotted in Figure 5.4 on page 69.

This observations can be explained by the inferior numerical properties of push-flow compared to the other two versions, which has been discussed in Niederbrucker et al. (2012) and Niederbrucker and Gansterer (2013). The push-cancel-flow algorithm has been developed to preserve the resilient properties of the push-flow, but to improve numerical properties. Obviously, the numerical properties of the DDAA translates into the distributed matrix algorithms built on top of them, which is discussed in the next section with respect to theory.

5.2.3 Theoretical Analysis of Numerical Accuracy

The dmGS algorithm summarized on the right side of Algorithm 9 on page 65 originates from the modified Gram-Schmidt orthogonalization, where several summations were replaced by its distributed variants. The essential question is, whether such a change in the modified Gram-Schmidt orthogonalization method influences the

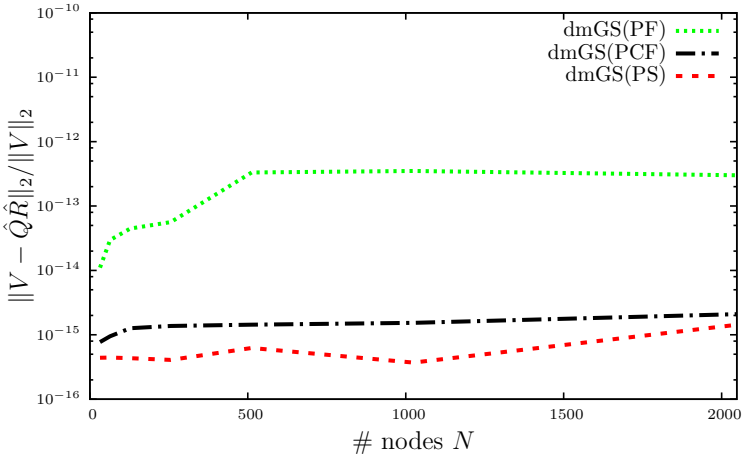

Figure 5.3: Factorization error of results computed by distributed QR factorization (dmGS) using push-sum (PS), push-flow (PF) and push-cancel-flow (PCF) factorizing a random matrix $V \in \mathbb{R}^{2048\times 32}$ over a hypercube with N nodes and with accuracy $\epsilon = 10^{-15}$ of DDAAs.

numerical stability of the distributed QR factorization. Our goal is to show that the final accuracy of the result delivered by dmGS is of the same order of magnitude as the inner accuracy ϵ set in the utilized distributed aggregation algorithm. The theoretical analysis of the error introduced by modified Gram-Schmidt orthogonalization in finite precision arithmetic can be found in the standard literature. The theorem essential for analyzing dmGS can be found in (Higham, 2002, Section 19.8):

Theorem 5.2.1. *Suppose the mGS method is applied to $V \in \mathbb{R}^{n\times k}$ of rank k, yielding computed matrices $\hat{Q} \in \mathbb{R}^{n\times k}$ and $\hat{R} \in \mathbb{R}^{k\times k}$. Then there are constants $c_i \equiv c_i(n,k)$ such that*

$$V + \Delta V_1 = \hat{Q}\hat{R}, \qquad \|\Delta V_1\|_2 \leq c_1\, \epsilon_{mach}\, \|V\|_2, \tag{5.1}$$

$$\|\hat{Q}^T\hat{Q} - I\|_2 \leq c_2\, \epsilon_{mach}\, \kappa_2(V) + \mathcal{O}((\epsilon_{mach}\, \kappa_2(V))^2), \tag{5.2}$$

where ϵ_{mach} is the machine precision.

Theorem 5.2.1 says that the modified Gram-Schmidt orthogonalization method is backward stable in terms of the relative factorization error and forward stable in terms of orthogonality of the matrix Q. In other words, the factorization error is of the same order as ϵ_{mach} and the computed solution $\hat{Q}\hat{R}$ is in fact an exact factorization of a slightly perturbed input matrix V. In contrast to the factorization error, orthogonality of Q also depends on the condition number $\kappa(V)$ of the input matrix V. The proof of this theorem given in Higham (2002) first establishes a connection between the Householder QR factorization (see Section 2.1) and the modified

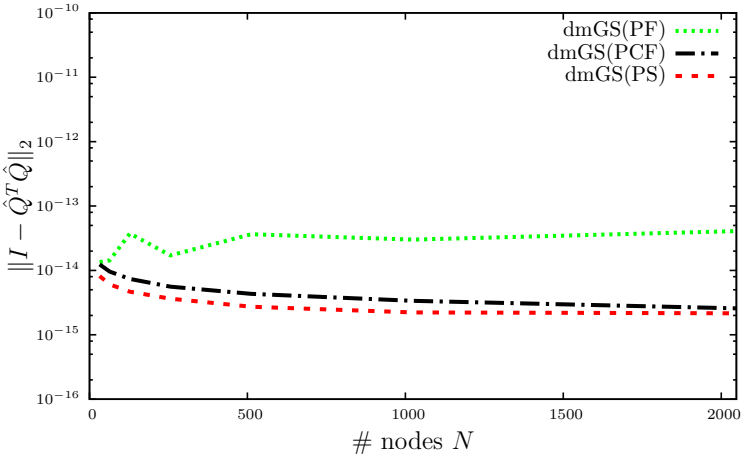

Figure 5.4: Orthogonality of $\hat{Q}$ computed by distributed QR factorization (dmGS) using push-sum (PS), push-flow (PF) and push-cancel-flow (PCF) factorizing a random matrix $V \in \mathbb{R}^{2048\times 32}$ over a hypercube with N nodes and with accuracy $\epsilon = 10^{-15}$ of DDAAs.

Gram-Schmidt orthogonalization method showing that both methods are mathematically and numerically equivalent. Then, using this key fact, the error bounds of the modified Gram-Schmidt orthogonalization are derived from the analysis of QR factorization based on Householder reflectors.

We are interested in the final accuracy of the results computed by dmGS, when all instances of the DDAA during the computation deliver an ϵ-accurate estimate with $\epsilon \geq \epsilon_{mach}$ and local computations in nodes are executed in accuracy ϵ_{mach}. Let's consider for a while weaker assumptions, when *all* computations in dmGS, distributed as well as local, are carried out in working precision $\epsilon \geq \epsilon_{mach}$. This algorithm we denote by dmGS$^{(\epsilon)}$. Using the theoretical bound Equation (5.1) on the factorization error we can conclude that dmGS$^{(\epsilon)}$ computes matrices $\hat{Q}$ and $\hat{R}$ with relative factorization error bounded by the working accuracy ϵ. Similarly, using the second bound Equation (5.2), we get an analogous statement for the orthogonality of the matrix $\hat{Q}$ computed by dmGS$^{(\epsilon)}$, which is also influenced by the condition number $\kappa(V)$ of the input matrix V.

We can conclude that the factorization error as well as the orthogonality of the matrix $\hat{Q}$ computed by dmGS$^{(\epsilon)}$ is bounded by ϵ if all DDAAs and the local computations are computed in accuracy $\epsilon \geq \epsilon_{mach}$. Obviously, the derived error bounds hold also for the situation we consider throughout this thesis, namely that only the DDAAs are computed in accuracy ϵ, but the local computations are carried out in full accuracy ϵ_{mach}. Not only the simulations, but also the theoretical analysis confirmed that if all instances of DDAA deliver ϵ-accurate results of the true aggregates, then the final accuracy of the result computed by dmGS is also of the order of ϵ.

5.3 Scalability

Many potential applications of distributed algorithms are on large-scale computers or social networks, or maybe even future large-scale high-performance computers. For such systems the key question to answer is how the algorithms scale with increasing problem or system size. Consequently, we analyze the scalability of dmGS with respect to the increasing number of nodes N, as well as the increasing size of the input matrix $V \in \mathbb{R}^{n \times k}$.

5.3.1 Scaling with System Size

Distributed aggregation algorithms are the only places in dmGS where the nodes communicate with each other (Lines 3 and 9 in Algorithm 9 on page 65), and thus dmGS scales with increasing number of nodes like the called DDAA.

Figure 5.5 illustrates the scaling behavior of push-sum on a fully connected network and a hypercube, as simulated by MATLAB . The plot shows the number of rounds executed by push-sum for summing N numbers over N nodes with target accuracy $\epsilon = 10^{-15}$. The nodes $N = 2^h$, $3 \leq h \leq 12$ are organized either in a fully connected network or in a hypercube of dimension h (Section 4.3.1).

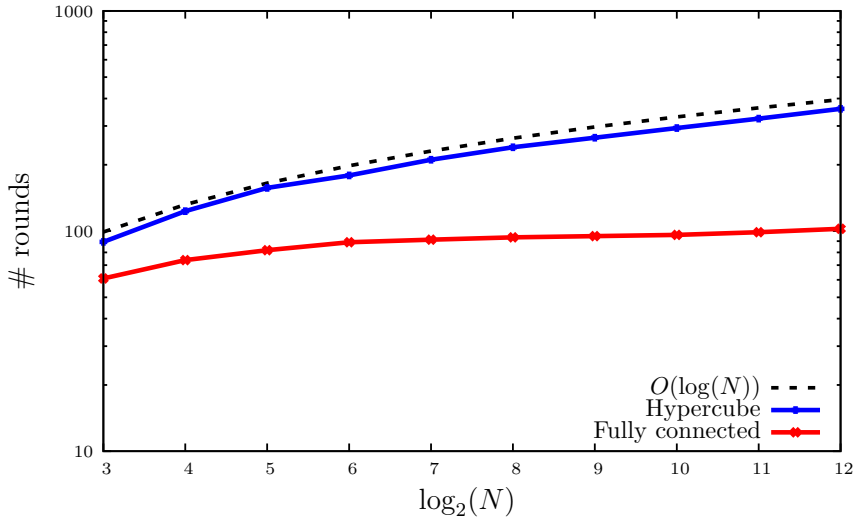

Figure 5.5: Scalability of the push-sum algorithm with increasing number of nodes N of a fully connected network and over a hypercube compared to the theoretical asymptotic bound.

In Section 3.2.2 we summarized the influence of the topology on the asymptotic number of rounds needed by push-sum for computing an ϵ-accurate estimate. The important result shows that for many important well-connected topologies the number of rounds scales logarithmically with the increasing number of nodes N for a fixed final accuracy ϵ.

From the figure we see that both topologies scale in compliance with the theoretical asymptotic bound (black dashed line). Due to the better connectivity of the fully connected network, the information spreads faster and the push-sum needs fewer rounds to converge than the distributed summation over a hypercube.

5.3.2 Scaling with Problem Size

Next, we investigate how the distributed modified Gram-Schmidt orthogonalization dmGS scales with increasing problem size for fixed number of nodes N. If $n > N$, nodes have to store more than one row of the input matrix $V \in \mathbb{R}^{n \times k}$. This influences the amount of local computation in each node, since before each DDAA the corresponding elements have to be summed up (see Section 4.2). Because the number of DDAA called in the dmGS algorithm is $k(k+1)/2$ (see right side of Algorithm 9 on page 65), the communication cost is not influenced by the number n of rows of the matrix V. I. e., number of messages sent as well as the amount of data stay the same for increasing n when the number of nodes N is fixed. Moreover, as shown in Figure 5.6, storing several rows in one node may even slightly increase the convergence speed of the factorization error. The more rows stored locally, the bigger parts of the distributed summations are computed locally in nodes in floating-point arithmetic, which may result in more accurate output than the distributed summation with target accuracy ϵ.

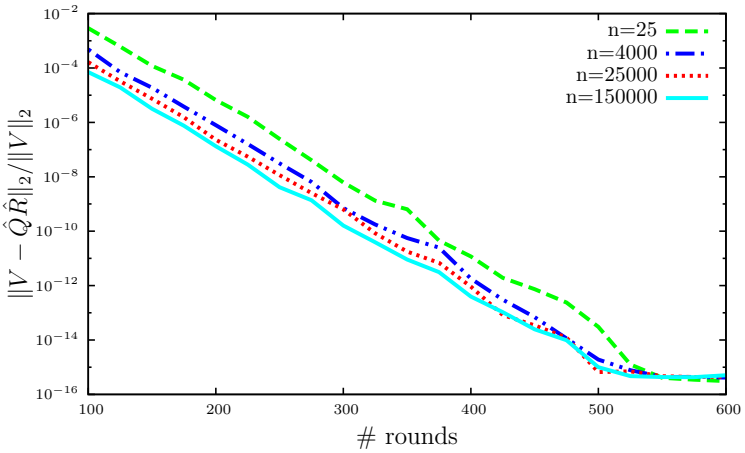

Figure 5.6: Scalability of dmGS with increasing number of rows n for matrix $V \in \mathbb{R}^{n \times 25}$ over a random network with $N = 25$ nodes and diameter 4.

Increasing the number of columns k has a much bigger influence on the performance of dmGS, because the number of calls of a DDAA $k(k+1)/2$ increases quadratically with increasing k. This translates to quadratically higher cost invested during DDAA, such as communication cost (a detailed analysis of communication

cost follows in Section 5.4). Clearly, high number of calls of DDAA is a limiting factor of dmGS for achieving good scalability with growing k.

Trading Bandwidth for Latency

We improve the scalability of dmGS with the number of columns k of the input matrix V by reducing the number of the distributed aggregation algorithms called. The new version of dmGS shown in Algorithm 10 on page 73, which we call *vdmGS*, is built by replacing several calls of a scalar DDAA by one call of a vector DDAA. We replace the original inner for-loop computing column-elements of the matrix Q (Lines 7-12 in Algorithm 9 on page 65) by *one* call of a vector DDAA exchanging messages of size $\mathcal{O}(k)$ (Line 8 in Algorithm 10).

This modification, which is well known, e. g., under the term "message coalescing" for MPI codes, does not have any influence on the numerical accuracy or convergence of the distributed algorithm and is suitable if bandwidth can be traded for latency. In vdmGS the total number of DDAAs executed (scalar and vector) is $2k - 1$, which leads to much better scalability with growing number of columns k of the matrix V compared to dmGS.

In order to illustrate the improved scalability of vdmGS, we implemented both algorithms in the ns-3 network simulator. Both algorithms executed the same number of rounds and achieved the same accuracy ϵ of the results. Figure 5.7 shows the comparison of dmGS and vdmGS in terms of the total number of messages sent and the results confirm the better scalability of vdmGS in terms of number of messages sent compared to dmGS with growing number of columns k.

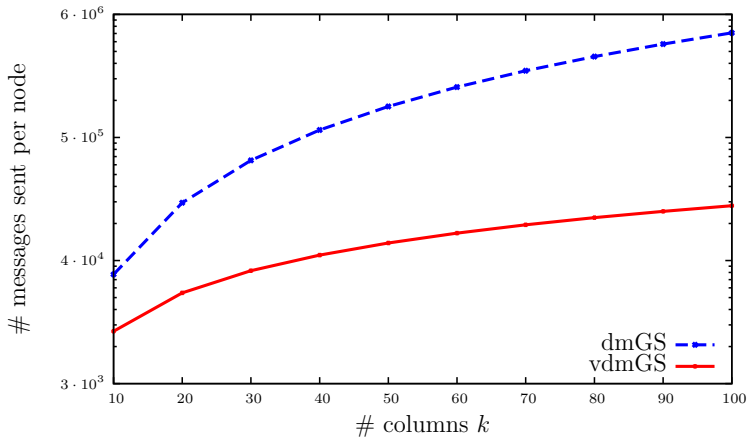

Figure 5.7: Number of messages sent per node by dmGS and vdmGS for growing number of columns k of a matrix $V^{100\times k}$ over a fully connected topology with $N = 100$ nodes.

Algorithm 10 Distributed QR Factorization with vector DDAA (vdmGS)

Input: matrix $V \in \mathbb{R}^{n\times k}$, $n \geq k$,
distributed row-wise over N nodes
Output: $Q \in \mathbb{R}^{n\times k}$, $R \in \mathbb{R}^{k\times k}$,
Q distributed row-wise, R distributed column-wise

1: **for** $i \leftarrow 1$ **to** k **do** (in each node l do)
2: $x(l) \leftarrow V(l,i)^2$
3: $s(l) \leftarrow$ **DDAA**(x)
4: $R_l(i,i) \leftarrow \sqrt{s(l)}$
5: $Q(l,i) \leftarrow V(l,i)/R_l(i,i)$
6: if $l \neq i$ delete $R_l(i,i)$
7: $X(l,1:k-i) \leftarrow Q(l,i).^*A(l,i+1:k)$
8: $R_l(i,i+1:k) \leftarrow$ **vector DDAA**(X)
9: $V(l,i+1:k) \leftarrow V(l,i+1:k) \; - \; Q(l,i)R_l(i,i+1:k)$
10: if $l \neq j \leftarrow i+1:k$ delete $R_l(i,j)$
11: **end for**

5.4 Comparison with Parallel Algorithms

Although distributed and parallel algorithms are designed for systems with different assumptions (Section 1.1), it is interesting to compare the invested cost of both approaches. We compare both distributed QR factorization algorithms, dmGS and vdmGS using push-sum, with state-of-the-art parallel algorithms for QR factorization (see Section 3.1), namely parallel mGS and with parallel CAQR, which is optimal in terms of communication cost (up to the polylogarithmic factors).

The comparisons are provided in terms of number of floating-point operations, local memory requirements and communication cost along the critical path for algorithms applied to a rectangular matrix $V \in \mathbb{R}^{n\times k}$ distributed over N nodes or P processors. The critical path is the longest path in a graph capturing the dependencies between individual steps in the computation. Since all of the nodes in dmGS and vdmGS proceed in parallel in rounds with the same structure, all paths in the graph have the same length. Consequently, the cost along the critical path in dmGS and vdmGS are equal to the cost per a single node.

For simplicity, we assume that the push-sum algorithms in dmGS and vdmGS proceeds in synchronized rounds and that in each round every node sends exactly one message which is *not* required in practice. We have to emphasize that this comparison has to be interpreted carefully, since these different types of algorithms are designed for very different target systems and are based on very different assumptions on properties of the target hardware and on the initial data distribution.

5.4.1 Operation Count

In both variants of the distributed QR factorization algorithm, each node l prepares a message before every push-sum by locally summing up $r_l = \mathcal{O}(n/N)$ elements, $N \leq n$, where dmGS sums up scalars and vdmGS works on vectors of length at most k. While dmGS executes push-sum in total $k(k+1)/2$-times, vdmGS calls only $(2k-1)$ push-sum algorithms. As a consequence, the total asymptotic flop count is the same for both variants, namely $\mathcal{O}(k^2 \log N + k^2 n/N)$ floating-point operations. According to Demmel et al. (2012), parallel mGS performs $2nk^2/P$ and parallel CAQR performs $2k^3/3 + 2k^2 n/P$ operations along the critical path. I. e., asymptotically, parallel mGS has the lowest flop count of the three methods and the count of dmGS and vdmGS is a little lower than the one of parallel CAQR (for fixed $P = N$).

5.4.2 Local Memory Requirements

For a general case $N \leq n$, in dmGS and vdmGS each node l needs to store $r_l = \mathcal{O}(n/N)$ rows of matrix V, r_l rows of matrix Q, and k of the nodes need to store one column of matrix R, each of them of length k. This leads to local memory requirements of $\Theta(r_l k) = \Theta(nk/N)$. According to Demmel et al. (2012), parallel CAQR needs $\Theta(nk/P)$ memory space per processor. This shows that asymptotically CAQR and the distributed algorithms have the same local memory requirements.

The situation is different, if dmGS or vdmGS keeps in each node an estimate of the entire matrix R. Then the local memory cost is higher, namely $\Theta(r_l k^2) = \Theta(nk^2/N)$. However, in many situations the input matrix V is a tall-and-skinny matrix, where $n \gg k$, and thus the increase in the local memory requirements is acceptable.

5.4.3 Communication Cost

For comparing the communication cost of the algorithms we use the size of messages, the number of messages and amount of data sent along the critical path. The number of messages C sent *per node* in both dmGS and vdmGS can be modeled as

$$C = P \cdot R \cdot M$$

where P is the number of executed push-sum algorithms, R is the number of rounds per push-sum and M is the number of messages sent during a round of push-sum. From the structure of push-sum (Algorithm 4 on page 44), we can see that in each round every node sends exactly one message. Therefore, in our case $M = 1$.

The number R of rounds per push-sum depends mostly on the underlying topology and on the accuracy parameter ϵ. As discussed in Section 3.2.2, for well-connected topologies, the push-sum algorithm converges in logarithmic time with respect to the size of the system for fixed accuracy ϵ. Consequently, in our communication cost analysis we consider $R = \mathcal{O}(\log N)$.

The number P of the executed aggregation algorithms during the distributed QR factorization depends on the concrete version used. Because in dmGS it is $P = k(k+1)/2$, the number of messages sent per node is:

$$C_{dmGS} = \frac{k(k+1)}{2} \cdot R \cdot 1 = \mathcal{O}(k^2 \cdot \log N)$$

In dmGS, each message has a constant size (two numbers), regardless of the size of the network or the size of the problem. Therefore, the amount of data transmitted is asymptotically the same as the number of messages sent.

In contrast, vdmGS calls push-sum only $P = 2k - 1$ times. Thus, the number of messages sent by each node in vdmGS is

$$C_{vdmGS} = (2k-1) \cdot R \cdot 1 = \mathcal{O}(k \cdot \log N),$$

which is asymptotically a factor $\mathcal{O}(k)$ less than indmGS. Size of the messages in vector push-sum varies from 2 to k, so it scales linearly with the increasing number of columns k. The amount of data sent per node in vdmGS is asymptotically the same as for dmGS, namely

$$D_{dmGS} = D_{vdmGS} = \mathcal{O}(k^2 \cdot \log N).$$

In Demmel et al. (2012), the exact leading terms of the communication cost for parallel mGS and for parallel CAQR are presented. For the sake of simplicity, we show only the corresponding asymptotic bounds on the number of messages sent and on the amount of data sent. Parallel mGS sends

$$C_{mGS} = \mathcal{O}(k \log(P))$$

messages along the critical path and

$$D_{mGS} = \mathcal{O}(k^2 \log(P))$$

words when factorizing a matrix $V \in \mathbb{R}^{n \times k}$ over P processors. Consequently, the average message size $D_{mGS}/C_{mGS} = k/4$ increases linearly with increasing k, similarly to vdmGS.

To complete the overview, parallel CAQR sends

$$C_{CAQR} = \mathcal{O}(\cdot\sqrt{kP/n} \cdot \log^2(nP/k) \cdot \log(P\sqrt{nP/k}))$$

messages and

$$D_{CAQR} = \mathcal{O}(\sqrt{nk^3/P} \cdot \log P - 1/4 \cdot \sqrt{k^5/(nP)} \cdot \log(kP/n))$$

words along the critical path.

Let's now compare the methods for a special case of a square matrix $V^{n\times n}$ over N nodes (or processors), which is also summarized in Table 5.1. We get that CAQR needs $\mathcal{O}(\sqrt{N}\log^3(N))$ messages and $\mathcal{O}(n^2\log(N)/(\sqrt{N}))$ words along the critical path for factorizing a square matrix over N processors. When comparing the number of *data* sent, we see that that dmGS as well as vdmGS sends asymptotically the same amount of *data* along the critical path as parallel mGS and by a factor $\sqrt{N}$ more than parallel CAQR.

	Operation count	# words	# messages
mGS	$\mathcal{O}(n^3/N)$	$\mathcal{O}(n^2\log N)$	$\mathcal{O}(n^2\log N)$
CAQR	$\mathcal{O}(n^3)$	$\mathcal{O}(n^2\log N/\sqrt{N})$	$\mathcal{O}(\sqrt{N}\log^3(N))$
dmGS	$\mathcal{O}(n^2(\log N + n/N))$	$\mathcal{O}(n^2\log(N))$	$\mathcal{O}(n^2\log(N))$
vdmGS	$\mathcal{O}(n^2(\log N + n/N))$	$\mathcal{O}(n^2\log(N))$	$\mathcal{O}(n\log(N))$

Table 5.1: Comparison of gossip-based distributed QR factorization with parallel algorithms for a square input matrix $V^{n\times n}$ over N nodes (or processors).

When comparing the number of *messages* sent, there is a bigger difference between the methods due to different size of messages in the methods. Asymptotically, parallel mGS sends by a factor n fewer messages along the critical path than dmGS, and the communication-optimal CAQR algorithm sends even by a factor $n^2/(\sqrt{N}\log^2(N))$ fewer messages than dmGS. This shows that dmGS would not be competitive in terms of communication cost when executed on standard target systems for parallel algorithms. However, distributed QR factorization vdmGS is, thanks to its improved scalability, comparable to parallel mGS in terms of the number of messages sent and only by a factor $n/(\sqrt{N}\log^2(N))$ worse than CAQR.

For a general case of a rectangular matrix $V \in \mathbb{R}^{n\times k}$, where $k < n$, it is not so easy to see directly from the formulas how CAQR and the methods based on modified Gram-Schmidt orthogonalization compare in terms of communication cost. In order to illustrate the principal dependency of the methods on the varying parameters n, k and P, we plot the ratios of the asymptotic cost for vdmGS and CAQR.

The following results show the ratio of the asymptotic cost C_{vdmGS}/C_{CAQR} for

the number of messages and D_{vdmGS}/D_{CAQR} for the amount of data. Because we compare only *asymptotic* cost, the results has to be interpreted carefully. The presented plots do not give any concrete numbers when the one method is better or worse than the other one. Instead, they illustrate the changes in the relation of the communication cost of the methods with increasing network size and problem size. The darker the color in the figure, the smaller is the ratio C_{vdmGS}/C_{CAQR} or D_{vdmGS}/D_{CAQR} and vice versa. We investigate the ratios for fixed number of rows $n = 10000$ and for n being a function of k.

The left part of Figure 5.8 shows the ratio of asymptotic bounds of the number of messages sent for $n = 10000$. We see a clear influence of the number of columns k on the result, since the more columns of the input matrix has, the more advantageous is parallel CAQR compared to vdmGS and mGS. Moreover, the figure indicates that the comparison of the methods depends linearly on the ratio k/P. In the right part of Figure 5.8, where $n = k \cdot P$, we can observe a similar influence of the increasing number of columns k on the comparison of the two methods as in the case where $n = 10000$. However, in this second scenario the relation of the asymptotic cost in terms of number of messages is not linearly dependent on the ratio k/P.

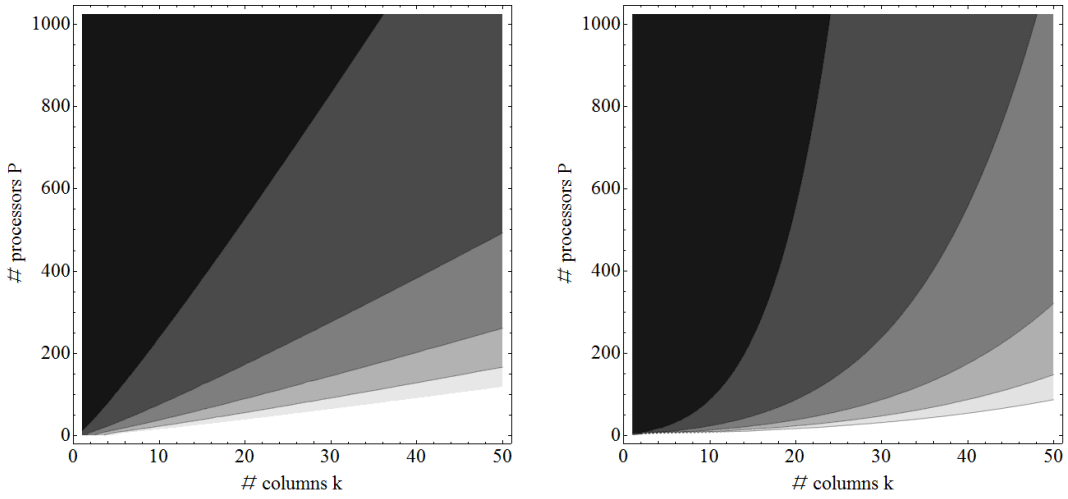

Figure 5.8: Ratios of the asymptotic number of messages sent C_{vdmGS}/C_{CAQR} showing the relation of vdmGS and CAQR depending on the increasing number of processors P and the number of columns k of the input matrix. On the left side, the number of rows $n = 10000$ is constant, on the right side $n = kP$. The darker the color, the closer the two methods are in terms of number of messages sent.

The comparison of the amount of data sent illustrates different dependencies on the parameters k and P. As shown in the left part Figure 5.9, for fixed $n = 10000$ vdmGS has the lowest communication cost compared to CAQR for a very small number of processors P and increasing the number of columns k, or for a tall-and-

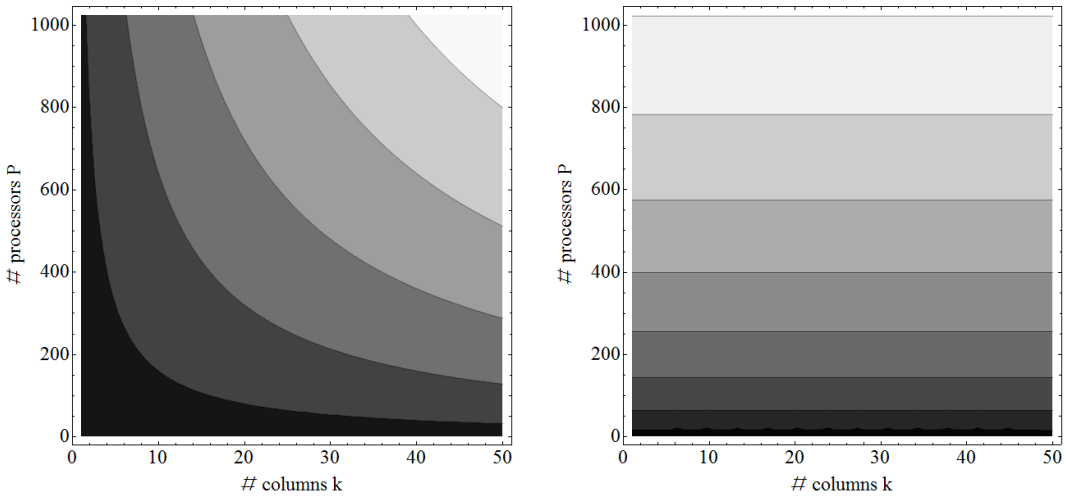

Figure 5.9: Ratios of the asymptotic amount of data sent D_{vdmGS}/D_{CAQR} showing the relation of vdmGS and CAQR depending on the increasing number of processors P and the number of columns k of the input matrix. On the left side, the number of rows $n = 10000$ is constant, on the right side $n = 100k$. The darker the color, the closer the two methods are in terms of amount of data sent.

skinny matrix factorized across increasing number of processors P.

If we set $n = k \cdot P$, the amount of data of CAQR are the asymptotically the same as for vdmGS, since

$$\mathcal{O}(\sqrt{kPk^3/P} \cdot \log P - \sqrt{k^5/(kP^2)} \cdot \log(\sqrt{kP/kP})) = \mathcal{O}(k^2 \cdot \log P).$$

I. e., for the situation $n = k \cdot P$ the ratio of the amount of data sent is constant for all k and P. Therefore, the right part of Figure 5.9 shows the comparison when the ratio between n and k stays constant, namely $n = 100k$. For this scenario, the more processors P, the better CAQR is in terms of amount of data exchanged compared to the methods based on mGS, regardless of the size of the matrix.

5.5 Fault Tolerance Properties

In the potential target systems of distributed algorithms, different types of failures may occur (see Section 4.3.3). Failures of communication links or of message losses are more likely to occur in loosely coupled distributed systems, such as sensor networks. In contrast, for high-performance parallel computers, resilience of algorithms against node failures is more crucial due to the increasing size and complexity of the machines.

In this chapter we investigate the potential of gossip-based QR factorization for handling different kinds of failures. First, we illustrate the limitations of dmGS

based on the push-sum algorithm in the presence of unreported message loss and outline possible solutions. Then, we discuss the impact of node failures on dmGS and we present a new more robust version *rdmGS* designed for systems where node failures occur.

5.5.1 Unreported Message Loss

Each unreported message loss, which happens during the push-sum algorithm, causes violation of the overall mass conservation in the system (see Section 3.2.2), which may lead to incorrect results. To illustrate the behavior of the dmGS algorithm in the presence of silent message loss, we investigate the factorization error and orthogonality of results returned by dmGS for factorizing a matrix $V \in \mathbb{R}^{32\times32}$ with $\kappa(V) = 279$ over a hypercube with $N = 2^5$ nodes. For every message sent it is independently decided with probability p, if the message transmission fails.

The impact of message losses on the relative factorization error of matrices computed by dmGS with push-sum is shown in Figure 5.10. We can notice that the increasing probability p of message loss causes slower convergence, but it does not influence the accuracy. This is due to the structure of dmGS originating from the modified Gram-Schmidt orthogonalization (see Algorithm 9 on page 65).

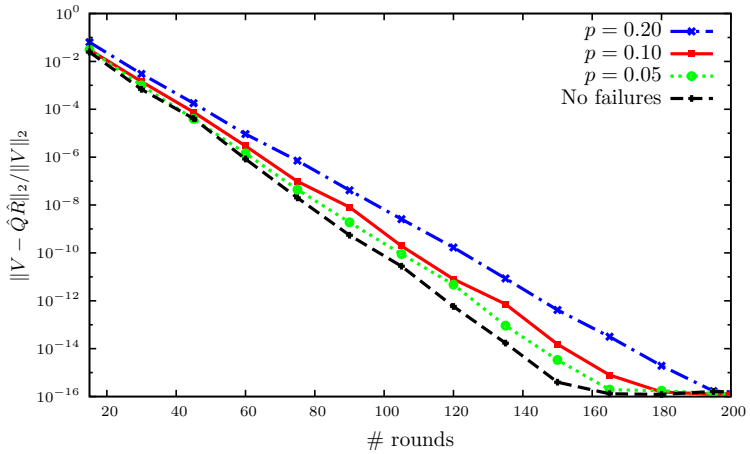

Figure 5.10: Relative factorization error of the results computed by dmGS factorizing a matrix $V \in \mathbb{R}^{32\times32}$ over a network with a hypercube topology and 2^5 nodes in the presence of unreported message loss with different probabilities p.

Matrix R computed by dmGS is always upper-triangular regardless of the reliability p of the transmissions, because every element of R is initialized to zero and the elements $R(i,j)$, $i < j$ are never changed. The elements $R(i,j)$, $i \geq j$, are directly computed by the DDAAs (Line 4 and 9 in Algorithm 9). The elements of the matrix Q are computed in such a way that the equality $V = QR$ holds, so matrices Q and R are always a factorization of V, regardless of the values in the matrix R.

Thus, losing messages may slow down the convergence, but the accuracy of the final factorization error is not influenced by wrong results computed by push-sum.

The fact that the push-sum algorithm does not deliver correct results in the presence of message loss has an impact on the orthonormality of the columns of Q, because the incorrect sum delivered by push-sum is used for computing the elements of the matrix Q (Line 5 in Algorithm 9). As a result, the columns of Q are *not* orthonormal to each other, as can be seen in Figure 5.11 (dotted lines). To prevent the loss of orthogonality of the matrix Q we have to either avoid losing messages or to recover from the mass loss after a failure.

Since the push-sum algorithm does not have the ability to naturally recover from a mass loss (see Section 3.2.1), the mass has to be preserved throughout the whole computation, in order to deliver correct results. Let's investigate one of the simplest strategies, where each message is sent several times to ensure that it will be received with a high probability. Figure 5.11 shows the orthogonality of Q delivered by dmGS, where each messages was sent exactly once (dotted lines), compared to dmGS transmitting each message several times (solid lines). The concrete number of transmissions of each message was 12 times for failure rate 20%, 10 times for failure rate 10% and 6 times for failure rate 5%. As we can see, sending each message several times leads to achiving good orthogonality of the computed matrix $\hat{Q}$ even for different failure rates.

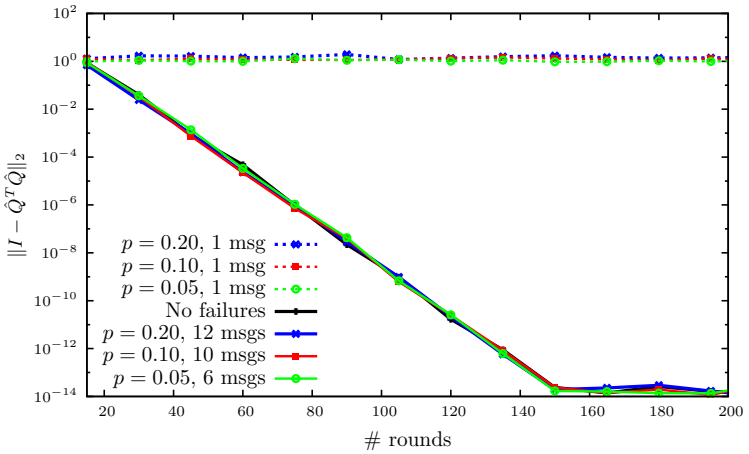

Figure 5.11: Orthogonality of $\hat{Q}$ computed by dmGS factorizing a matrix $V \in \mathbb{R}^{32\times32}$ over a network with a hypercube topology and 2^5 nodes in the presence of unreported message loss with different probabilities p.

Although the resending strategy for preventing mass loss is easy to carry out, it has several disadvantages, which makes it unattractive for a practical use. First of all, the concrete number of retransmissions of each message has to be determined. Since we assume each event of sending a message to be independent of the other

transmissions, we can calculate the concrete number of retransmissions r needed for a probability p of a message loss using the following inequality

$$p^r \leq m,$$

where m is the probability that a message is lost. For example, for a network with the rate of message loss $p = 10\%$ we want to reduce the probability that a message gets lost to $m = 10^{-10}$. Thus, from the expression $0.1^r \leq 10^{-10}$ we compute that each message has to be retransmitted at least 10 times. To be able to compute the concrete number r we have to know the failure rate p of each transmission in advance.

Obviously, in the situation where there is a temporary breakdown of a communication link in the system, no message can be sent over it, and retransmissions will not help to achieve a good orthogonality of Q. Last but not least, the presented strategy introduces a communication overhead regardless of whether a failure has occurred or not. It is usually preferable that additional cost is proportional to the number of failures occurred.

A better and more efficient solution is to base the dmGS algorithm on a DDAA which can naturally recover from a silent message loss, such as push-flow of push-cancel-flow. Using such self-healing distributed aggregation algorithm as the building block for distributed matrix computation and its influence on the performance in the presence of failures is discussed in Section 6.7.

5.5.2 Node Failures

Computing orthogonalization of the columns of a matrix V distributed over a network is a prototypical example where the loss of original data, e. g., resulting from a permanent node failure, *cannot* be tolerated. Distributed modified Gram-Schmidt orthogonalization assumes that the input matrix V and also the computed matrix Q are distributed row-wise over the N nodes. If a node permanently fails during the execution of dmGS, the locally stored part of the input matrix V together with the so far computed part of Q is irretrievably lost. As a consequence, it becomes impossible to orthogonalize all original columns of V.

We illustrate how resilience of dmGS against permanent node failures can be improved by introducing redundancy in storing the initial data. In this section, we present the *robust dmGS* algorithm (*rdmGS*) published in Gansterer et al. (2013), which produces a complete and accurate QR factorization of V in most of the scenarios where one or several nodes fail permanently during the computation.

The rdmGS Algorithm

The pivotal idea of the robust algorithm rdmGS is to maintain redundant copies of all nodes' relevant local data at more than one active node at all times. By construction, dmGS automatically computes an estimate of the *whole* matrix R at *all* nodes of the system (see Section 5.1). Thus, no specific measures are needed for backing up data of R. However, the matrices V and Q are distributed across the system and thus an explicit replication of the data across the system has to be introduced.

Every node y is responsible for a subset of entire rows of the initial matrix V and for the computed parts of the corresponding rows of Q, which we denote as y's *primary data*. Each node y may also act as a backup node for the primary data of one or more other nodes in the system. Such data, which are redundantly stored in node y, is called node y's *backup data*. A simple example of a starting data distribution of the rdmGS algorithm is drawn in Figure 5.12. Each of the three nodes x, y, z stores one row of the input matrix $V \in \mathbb{R}^{3\times 3}$ as its primary data, and another row of V as backup for another node from the network. We call node x, which backs up node y's data, y's *guardian*, and y in turn x's *protégé*.

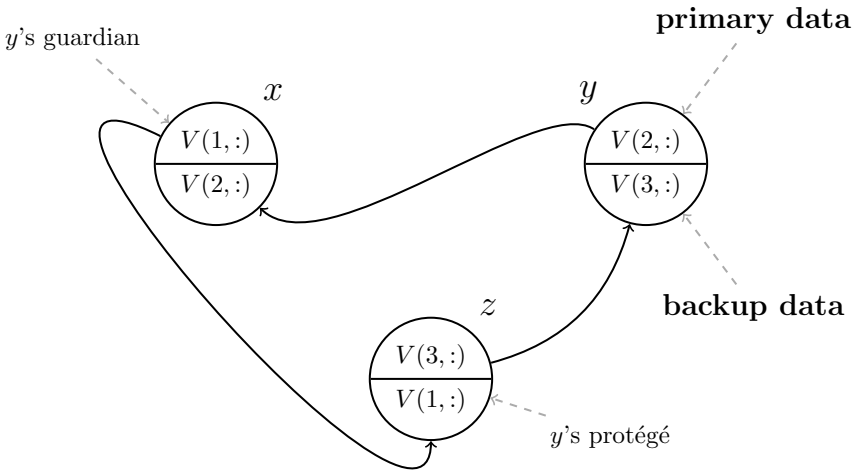

Figure 5.12: Scheme illustrating the principle of storing data across nodes in rdmGS.

At all times, $r-1$ backup copies of node l's primary data are stored in $r-1$ distinct other active nodes which act as backup nodes for node l. The parameter r is a measure of resilience as well as of overhead of the rdmGS algorithm. Larger r allows for tolerating more simultaneous node failures, albeit at higher cost. The structure of the algorithm rdmGS is outlined in Algorithm 11 on page 83, where P_l and B_l denote the set of global indices of rows of V and Q, which are primary data on node l and backup data on node l, respectively.

We discuss the case when $n = N$ and each node stores exactly one primary row

Algorithm 11 Robust Distributed QR Factorization (rdmGS)

Input: $V \in \mathbb{R}^{n \times k}$
Output: $Q \in \mathbb{R}^{n \times k}$, $R \in \mathbb{R}^{k \times k}$

1: **for** $i \leftarrow 1$ **to** k **do** (in node l)
2: ... check for node failures, update P_l and B_l ...
3: $x(l) \leftarrow \sum_{p \in P_l} V(p,i)^2$
4: $s(l) \leftarrow$ **DDAA**(x)
5: $R_l(i,i) \leftarrow \sqrt{s(l)}$
6: **for each** $p \in P_l$ **do**
7: $Q(p,i) \leftarrow V(p,i)/R_l(i,i)$
8: **end for**
9: **for each** $b \in B_l$ **do**
10: $Q(b,i) \leftarrow A(b,i)/R_l(i,i)$
11: **end for**
12: **for** $j \leftarrow i+1$ **to** k **do**
13: ... check for node failures, update P_l and B_l ...
14: $x(l) \leftarrow \sum_{p \in P_l} Q(p,i)V(p,j)$
15: $R_l(i,j) \leftarrow$ **DDAA**(x)
16: **for each** $p \in P_l$ **do**
17: $V(p,j) \leftarrow V(p,j) - Q(p,i)R_l(i,j)$
18: **end for**
19: **for each** $b \in B_l$ **do**
20: $V(b,j) \leftarrow V(b,j) - Q(b,i)R_l(i,j)$
21: **end for**
22: **end for**
23: **end for**

at the beginning, although the algorithm works for a general $n \geq N$ situation. We assume that each node l starts with one row $V(l,:)$, which is its primary data, and $r-1$ additional rows of V, which is the back-up data for other nodes in the network. However, those numbers may change over the time because of node failures. In the process of local computation, each node updates the primary data (Lines 7 and 17 in Algorithm 11), *as well as* all local backup data (Lines 10 and 20 in Algorithm 11).

Each execution of a DDAA in rdmGS is preceded by a fail-checking phase (Lines 2 and 13 in Algorithm 11), where every node l checks whether all of its protégés and its guardians are alive. If yes, node l proceeds with the algorithm. If no, the following steps are taken, depending on the concrete situation:

(*i*) A protégé p of node l has failed, which means that node l backs up the data for p. In this case, node l takes over the primary responsibility for p's primary data and selects another active node g to replace itself as guardian, which will from now on back up the primary data of the failed node p. As a result, again r copies of the primary data of the failed node l exist in the system.

(*ii*) A guardian g of node l has failed, which means that node l has lost one of

its $r-1$ backups. In this case, node l selects another active node, a new guardian g, and sends the primary data to it for backup. As a result, again r copies of the data exist in the system.

If more nodes have failed, the rdmGS takes one of the actions for each of the failed nodes. However, it may happen that the algorithm ends in a situation where it cannot continue or it cannot deliver correct results due to several failed nodes. These scenarios are described later in this section.

Upon termination, all remaining nodes in the system consider only its primary data as part of the final result, i. e., the final matrix Q is built from the primary data in all alive nodes at the end of the rdmGS algorithm.

Scope and Limitations of rdmGS

So far we discussed checking the node failures only *before* each call of the DDAA. However, in reality the rdmGS algorithm spends most of its time *in* the distributed summation. As explained in Section 3.2.2, the push-sum algorithm does not tolerate any data loss and if mass conservation is violated, it delivers incorrect results. Thus, if the push-sum algorithm is used, the nodes have to fail *neatly*. In practice it means that a node, before it fails (e. g., before it runs out of battery), sends its current estimate relevant for the ongoing push-sum to another node, usually a neighbor, to avoid violation of mass conservation. However, it does *not* have to send out all of its data and search for a new backup node, because this is done in the fail-checking phase.

To summarize, the rdmGS algorithm operates successfully under the following assumptions:

(*i*) The topology of the system stays connected despite all occurring node failures.

(*ii*) A reliable and efficient mechanism is available for determining whether a node (usually in the neighborhood) is alive (active) or not.

(*iii*) In case of permanent failures nodes fail *neatly*, if the DDAA does not tolerate node failures I. e., if a node fails *within* the execution of push-sum, this failure has to be reported immediately, and the failing node l has to send some of its local values to one of its neighbors in order to ensure mass conservation.

There are three scenarios of node failures which rdmGS cannot recover from:

(*i*) If the network gets disconnected, a correct result cannot be computed even if none of the nodes fails.

(*ii*) If a node l and all its $r-1$ guardians fail permanently before at least one of these failures is detected, then rows of V and Q are lost and cannot be recovered.

(*iii*) If a node l fails permanently after passing the fail-checks of all of its $r-1$ guardians, but before starting the next aggregation process. In this scenario mass

conservation is violated because l's primary data is not used within that aggregation process and l's guardians are not aware of it.

Note that the probability for these scenarios can be reduced by increasing the overhead in terms of fail-checking frequency. Moreover, the resilience properties of the distributed orthogonalization method depend also on the choice of the DDAA. Using the push-flow or push-cancel-flow (see Section 3.2.3) as the underlying aggregation algorithm for rdmGS allows for recovering from mass loss caused by link failures and thus increases the resilience of rdmGS to these types of failures. However, such modification helps *only* in the presence of link failures and message loss. Even when a self-healing DDAA is used, the rdmGS algorithm still needs to preserve the local values of a permanently failed node and thus, the nodes have to fail neatly. To improve the scalability of rdmGS, a version build on vdmGS (Section 5.3) can be constructed.

Node Management

So far, we have not discussed how the guardians and protégés are chosen, if a node fails. Many strategies can be suggested for managing the nodes and it may significantly influence the performance of the rdmGS algorithm. The following aspects should be taken into account, when designing the node management strategy in rdmGS.

First of all, the selection process should follow the objective of balancing the load across the active nodes. Since all nodes update the values of the primary as well as of the backup data, the more data they store, the more local computation they have to execute. Furthermore, the selection process has an impact on the distribution of the data among guardians and protégés and on the cost related to it. For instance, if primary data of node l is backed up always partially across several nodes, restoring the primary data when node l fails is naturally more complicated compared to the situation, when always the entire primary data of l are stored in one node. If a failed node has more guardians, a mechanism has to be established which determines, who takes the responsibility for the primary data of the failed node.

However, the choice of the most effective scheme essentially depends on the concrete topology, properties of the system and particular preferences. In some situations it is more efficient to choose one of the neighbors to store the back-up data in order to avoid complex routing across the network. In contrast, if there is a high probability that with a node also its neighborhood fails, the algorithm should back up the data at a more remote node. Consequently, we omit recommending any concrete suggestions in this thesis, because a lot of details play an important role in designing a concrete scenario.

Simulation Results

To illustrate the properties of rdmGS, we developed a simulation model based on push-sum in the ns-3 network simulator. Simulation results are shown for orthogonalizing a matrix $V \in \mathbb{R}^{512\times 32}$ on a wired network with a hypercube topology on $N = 2^9$ nodes and with $r = 2$ (i. e., each node has one guardian). The times until failure of a node are exponentially distributed with mean $\lambda = 15$ [s], and nodes fail neatly in the push-sum algorithm.

The termination criterion for the push-sum algorithm is in this case the prescribed number of rounds $t_{\max}$ to be executed, $t_{\max} \in [300:50:600]$. For every value of t_{max} we ran 100 simulations in total. In our simulations the initial setup evenly distributes the rows of V over all nodes in the system and cyclically sets up guardian nodes, i. e., the guardian of node l is node $(l-1) \mod N$, so that there is exactly one guardian and one protégé for every node. If the guardian $(l-1)$ of node l fails, node l searches for a next active node in the row, i. e., the first active from nodes $l-2$, $l-3$, ... as the new guardian.

Every simulation yields a relative factorization error and orthogonality, a count of failed nodes and two flags. One flag indicates whether a full result could be achieved and the other, whether the network stayed connected. The flag "no full result" refers to scenarios where a node *and* its backup node both failed between checks for node failures (within the call of push-sum), which causes the complete loss of information and thus no full result can be delivered.

Figure 5.13 on page 87 shows the relative factorization error $\|V - \hat{Q}\hat{R}\|_2/\|V\|_2$ of matrices $\hat{Q}$ and $\hat{R}$ computed by rdmGS. We observe that for small $t_{\max}$, low accuracy of push-sum leads to low accuracy of rdmGS and the more rounds per push-sum were executed, the more accurate the factorization is, as expected. The larger the number of rounds $t_{\max}$, the longer the rdmGS algorithm runs, and thus, the higher chance that such a node failure constellations occurs, which rdmGS cannot recover from. Moreover, the longer rdmGS runs, the more nodes fail and thus, the higher probability that the network gets disconnected. Consequently, the fraction of simulation runs where rdmGS does not produce the full result due to node failures or disconnected network tends to grow with $t_{\max}$. In Figure 5.14 on page 87, we observe analogous behavior of the orthogonality of the matrix $\hat{Q}$ computed by rdmGS.

From simulation results shown in Figure 5.13 and Figure 5.14 we can conclude that there is a certain range of $t_{\max}$ where factorization accuracy and orthogonality achieved by rdmGS are excellent and in most cases the rdmGS algorithm recovers from node failures (90% - 95%). For the simulation setup considered in this thesis this range is around $t_{\max} = 550$.

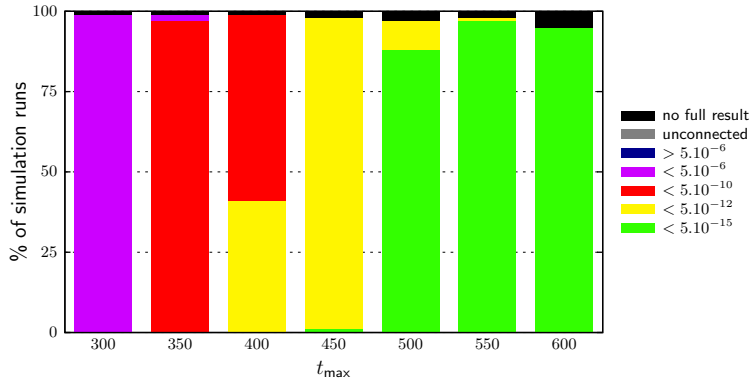

Figure 5.13: Relative factorization error of matrices computed by rdmGS for $\lambda = 15$ [s] over a hypercube topology with $N = 2^9$ nodes. For every value of t_{max} rounds, 100 simulations were run.

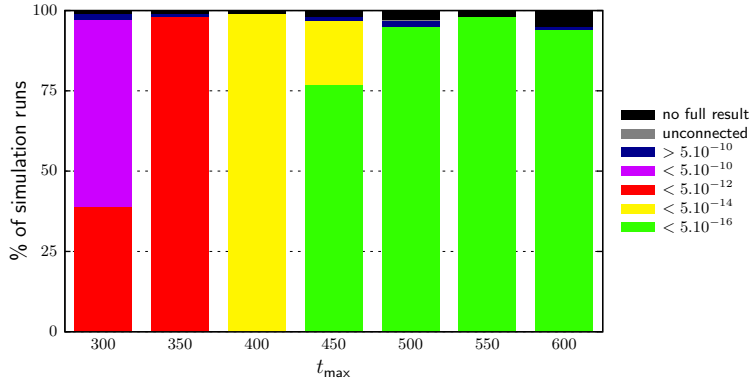

Figure 5.14: Orthogonality of matrix Q computed by rdmGS for $\lambda = 15$ [s] over a hypercube topology with $N = 2^9$ nodes. For every value of t_{max}, 100 simulations were run.

Figure 5.15 on page 88 depicts the average number of failed nodes for each value of $t_{\max}$. As expected, the more rounds per push-sum, the longer the algorithm runs and the more nodes fail. In our settings between 50 and 150 of the 512 nodes failed per simulation run and on average 94.95 nodes over all values of $t_{\max}$. Figure 5.16 on page 88 shows the distribution of the number of failed nodes depending on the maximum number of rounds $t_{\max}$ per push-sum.

5.6 Synchronization Issues

Distributed algorithms with low synchronization requirements are gaining attention in many application areas. In tightly coupled computers with massive parallelism, ensuring synchronization between nodes becomes more and more challenging with increasing size of the systems. In loosely-coupled distributed systems, such as (wireless) sensor networks or mobile ad-hoc networks, full global synchronization of com-

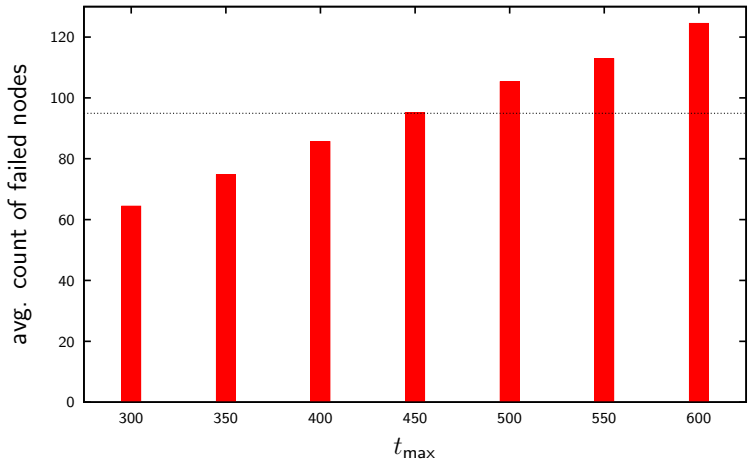

Figure 5.15: Average number of node failures per simulation for $\lambda = 15$ [s] and varying $t_{\max}$. The black line depicts the number 94.95, which is the number of failed nodes averaged over all simulation runs and all $t_{\max}$.

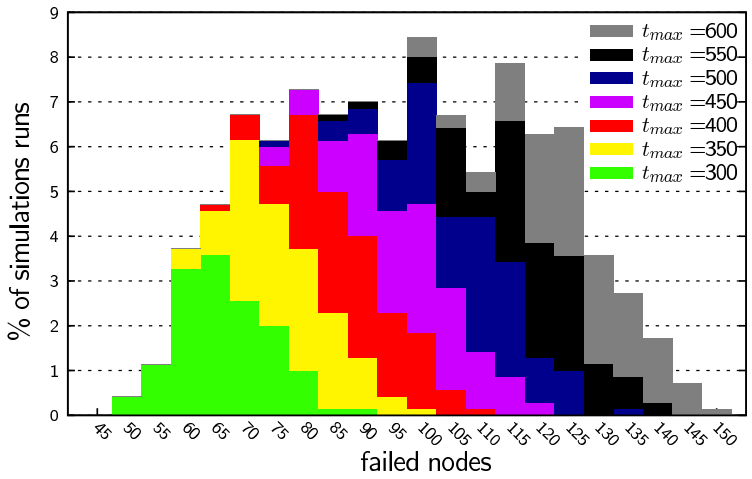

Figure 5.16: Distribution of node failures in rdmGS for $\lambda = 15$ [s] and varying $t_{\max}$.

putation across nodes can be very challenging in terms of good scalability and energy efficiency (Simeone et al., 2008). But it is important to point out that in the context of wireless sensor networks the main focus is on clock synchronization or synchronization to a physical media for transmission, which has been extensively address in the literature (e. g., Leng and Wu (2011); Yang et al. (2012); Awerbuch (1985); Peleg and Ullman (1987); van Greunen and Rabaey (2003)). In contrast, we focus exclusively on synchronization on the algorithmic level. The goal is to effectively control the progress of a distributed algorithm in individual nodes, in the ideal case without using global synchronization barriers, in such a way that a correct result is computed with possibly lowest communication cost.

In this section, we investigate how much the nodes need to coordinate their local computation during orthogonalizing a matrix using the dmGS algorithm, regardless

of the underlying hardware properties. Due to the flexibility their communication schedules, gossip-based DDAAs appear very attractive for systems, where ensuring synchronization of processes is difficult. However, in more complex algorithms such as dmGS, which comprise multiple calls of distributed data aggregation algorithms in sequence, the situation is different. Even if the DDAA is resilient to asynchrony, it does not necessarily hold for a sequence of such building blocks. We discuss potential problems, which may occur during dmGS if the nodes proceed independently of each other, and we introduce several synchronization strategies.

Our goal is to identify a strategy for delivering results in full accuracy with as few synchronization points in the algorithm as possible. We use the terms "synchronization point" or "global barrier" for a spot in the computation, where each node waits until every other node in the network has reached this spot before proceeding further. It is analogous for example to the MPI routine called `MPI_Barrier` (see Appendix A).

As explained in Section 5.1, in dmGS nodes exchange messages only during the distributed aggregation algorithm. If the DDAAs are globally synchronized, which means that all nodes start and finish every instance of DDAA jointly, the messages circulating in the network at one moment belong always to the *same* instance of the distributed aggregation algorithm. However, if all nodes call and finish the DDAA independently, messages exchanged in the network at a certain time may correspond to *different* instances of the DDAA. In this case, we extend each message by adding an identifier of the call of a DDAA denoted as `ddaaID`. Because dmGS executes always the same number of DDAAs in the same order, the identifier `ddaaID` guarantees that nodes can assign received messages to the right call of the DDAA. If node l receives a message, it compares the `ddaaID` contained in the received message ($\texttt{ddaaID}_{rec}$) with the `ddaaID` of its local computation ($\texttt{ddaaID}_{loc}$), and performs the following steps accordingly:

(*i*) If ($\texttt{ddaaID}_{rec} < \texttt{ddaaID}_{loc}$), node l cannot use the received data anymore, because it is already further in the computation than the neighbor who sent the message. Thus the received message is discarded.

(*ii*) If ($\texttt{ddaaID}_{rec} > \texttt{ddaaID}_{loc}$), the message comes from a neighbor who is ahead in the computation compared to node l. The message is stored and used later, when node l reaches the corresponding instance of the DDAA.

(*iii*) If ($\texttt{ddaaID}_{rec} = \texttt{ddaaID}_{loc}$), node l immediately uses the received value for updating its current local status.

5.6.1 Synchronization Strategies

The dmGS algorithm consists of a sequence of local computations in nodes and distributed data aggregation algorithms proceeding in rounds (see Algorithm 9 on page 65). Due to this granularity we have several possibilities where to place synchronization barriers into dmGS. We explore and compare the following synchronization strategies:

(S3) Fully synchronized (Algorithm 12): a global barrier is placed before *each round* in *every instance* of the DDAA. All nodes proceed to the next same round in the distributed summation only if all nodes in the network have finished the current round and are ready to continue.

Algorithm 12 (S3) Fully synchronized

Input: $V \in \mathbb{R}^{n\times k}$, $n \geq k$, V distributed row-wise over N nodes
Output: $Q \in \mathbb{R}^{n\times k}$, $R \in \mathbb{R}^{k\times k}$

for $i = 1$ **to** k **do**
  local computation in nodes
  global barrier
  DDAA with global barrier before every round
  local computation in nodes
  for $j = i + 1$ **to** k **do**
    local computation in nodes
    global barrier
    DDAA with global barrier before every round
    local computation in nodes
  end for
end for

(S2) DDAAs synchronized (Algorithm 13): a global barrier is placed before *each instance* of the DDAA, but the individual rounds within one DDAA are *not* synchronized across the nodes.

Algorithm 13 (S2) DDAAs synchronized

Input: $V \in \mathbb{R}^{n\times k}$, $n \geq k$, V distributed row-wise over N nodes
Output: $Q \in \mathbb{R}^{n\times k}$, $R \in \mathbb{R}^{k\times k}$

for $i = 1$ **to** k **do**
  local computation in nodes
  global barrier
  DDAA without synchronization of rounds
  local computation in nodes
  for $j = i + 1$ **to** k **do**
    local computation in nodes
    global barrier
    DDAA without synchronization of rounds
    local computation in nodes
  end for
end for

(S1) Wait for a neighbor (Algorithm 14): a node stops refining its current estimate in the DDAA as soon as it receives a message with a higher `ddaaID` than is its local value, which means that at least one of its neighbors is further ahead in the computation. If a node does not receive any message with a higher `ddaaID`, it continues with the current instance of DDAA until the stopping criterion is satisfied.

(S0) Fully asynchronous: nodes proceed completely independently of each other without coordinating the calls of a DDAA or the rounds withing a DDAA. They use only the `ddaaID` number to assign a received value to the correct computation.

Algorithm 14 (S1) Wait for a neighbor

Input: $V \in \mathbb{R}^{n \times k}$, $n \geq k$, V distributed row-wise over N nodes
Output: $Q \in \mathbb{R}^{n \times k}$, $R \in \mathbb{R}^{k \times k}$

1: **for** $i = 1$ **to** k **do**
2: *local computation in nodes*
3: **DDAA terminating based on progress of neighbors**
4: *local computation in nodes*
5: **for** $j = i + 1$ **to** k **do**
6: *local computation in nodes*
7: **DDAA terminating based on progress of neighbors**
8: *local computation in nodes*
9: **end for**
10: **end for**

5.6.2 Communication Cost of Synchronization Strategies

Strategies (S3) and (S2) require global synchronization barriers, which causes some communication overhead in practice. Especially (S3) with a global barrier before each round in every DDAA is very strongly synchronized, but may be also expensive in terms of communication cost. The strategy (S1) does not use any barriers, but only adjusts the computation according to the received messages, so it does not introduce *any* additional cost for synchronization. An important question is if some of the partially asynchronous strategies have lower overall communication cost than the completely synchronized (S3).

The cost of the synchronization strategies (S2) and (S3) depends crucially on the concrete implementation of the global barrier. There are several methods how to implement a synchronization barrier across a system with different communication cost (Hoefler et al., 2004). For simplicity, we assume that each node sends exactly one message in a synchronization barrier, which is a coarse lower bound for the communication cost. The total number of messages sent in dmGS can be calculated as:

$$C = P \cdot R \cdot M + S,$$

where P is the number of calls of the DDAA, R is the number of rounds per DDAA, M is the number of messages sent per round and S is the total number of messages spent on synchronization. In this thesis we assume that in every round of a DDAA, each node sends one message, therefore $M = N$.

Because each global barrier means at least N messages, the synchronization strategy (S3) using a global barrier before each round of the DDAA has the total communication cost

$$C_{S3} \geq P \cdot R_{S3} \cdot N + P \cdot R_{S3} \cdot N = 2 \cdot P \cdot R_{S3} \cdot N.$$

The synchronization strategy (S2) uses a global barrier only before each call of the DDAA. Thus, the total communication cost of (S2) is

$$C_{S2} \geq P \cdot R_{S2} \cdot N + P \cdot N = P \cdot (R_{S3} + 1) \cdot N.$$

The last synchronization strategy (S1) does not use any synchronization barrier, therefore $S = 0$ and the total number of messages sent is

$$C_{S1} \geq P \cdot R_{S1} \cdot N.$$

Clearly, if we want to compare the individual strategies in terms of communication cost and decide which of them is the least expensive, we have to focus especially on the concrete number of rounds executed in the DDAAs. From the communication cost summarized above we get that $C_{S3} \geq C_{S2} \geq C_{S1}$ if $2R_{S3} \geq R_{S3} + 1 \geq R_{S1}$. This is experimentally investigated in the next section.

5.6.3 Simulation Results

We compare dmGS based on push-sum with the three synchronization strategies (S1) - (S3) in terms of convergence speed, final accuracy achieved and communication cost. In order to investigate the influence of different levels of synchronization on dmGS, we implemented all variants in a simulator based on ns-3. To study the impact of asynchrony we use a simple model of different timesteps between rounds in each node. The timestep for each node is generated from an exponential distribution with mean $\beta = 1$. In the simulations, all nodes start executing dmGS at the same time.

We evaluate dmGS applied to a square matrix $V \in \mathbb{R}^{N \times N}$ with elements randomly generated from a uniform distribution. The matrix V is distributed row-wise over a random topology with $N = 25$ nodes and diameter 4. As termination criterion for push-sum we use the prescribed maximum number of rounds. Note that in

strategies (S2) and (S3), each DDAA executes always exactly the given number of rounds, whereas in the strategy (S1) some of the nodes terminate before executing the maximum number of rounds.

Simulations showed that the fully asynchronous strategy (S0) does not always deliver correct results. Even if the nodes can assign the received information to the correct instance of the push-sum algorithm, individual nodes drift apart in the computation and thus they receive less and less information they need from their neighbors. Either a node is too fast and all other nodes are far behind in the computation, or a node is too slow and cannot get any messages from other nodes corresponding to its instance of the push-sum.

Figure 5.17 shows that all strategies reach the same order of accuracy in terms of the orthogonality of the computed matrix Q. However, the executed number of rounds in the push-sum for reaching the highest accuracy differs. We see that the strategy (S3) (fully synchronized dmGS) needs about $R_{S3} = 320$ rounds per push-sum, whereas the other two strategies need around 100 rounds more to achieve the final accuracy. This concrete numbers of rounds obtained from the simulations we use for comparing the communication cost for this example.

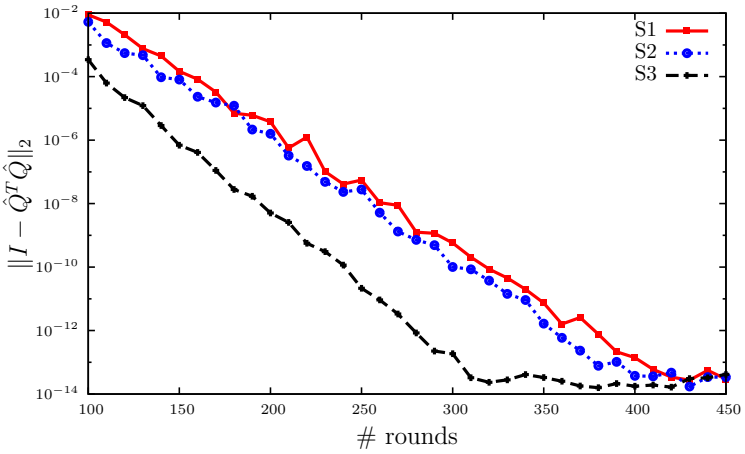

Figure 5.17: Orthogonality of $\hat{Q}$ computed by dmGS using push-sum with different synchronization strategies across a random network with $N = 25$ nodes and diameter 4.

Figure 5.18 on page 94 presents simulation results comparing all investigated synchronization strategies in terms of the number of messages sent per node for achieving a certain level of orthogonality of the computed matrix Q. First of all, we see that the fully synchronized strategy (S3) has much higher communication cost than the other strategies. In our case studies $R_{S3} = 320$ and $R_{S1} = 420$ (see Figure 5.17), therefore $C_{S1}/C_{S3} = R_{S1}/2R_{S3} < 1$. Second, we see in Figure 5.18 that the final communication cost of (S1) and (S2) in our case study is comparable,

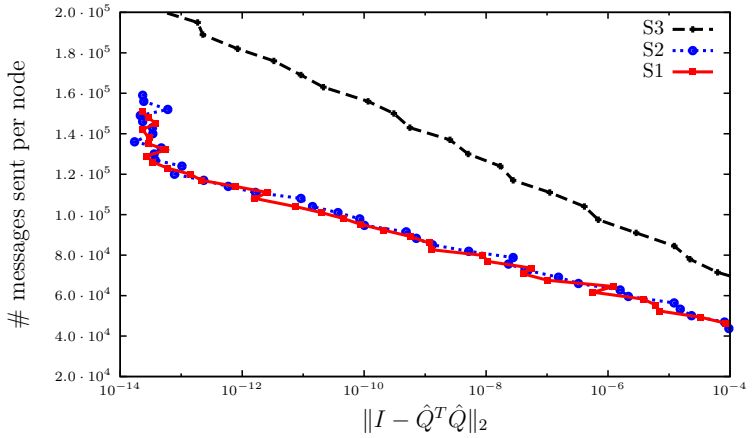

Figure 5.18: Number of messages per node in dmGS using push-sum for achieving a certain orthogonality level of $\hat{Q}$ with different synchronization strategies across a random network with $N = 25$ nodes and diameter 4.

because $C_{S1}/C_{S2} = R_{S1}/(R_{S2}+1) \approx 1$ and the strategy (S2) executed approximately the same number of rounds to converge as (S1) (see Figure 5.17).

In the strategy (S1) it is not guaranteed that all nodes execute the same instance of DDAA at every moment. Thus, some received messages may have to be discarded, if the receiver is already ahead in the computation. According to the simulation results, on average (over all calls of the DDAA) (S1) discarded 0,5% of all messages if the maximum number of rounds was set to 500. With the maximum number of rounds set to 100, (S1) discarded 2% of all messages.

Simulation showed that a very weakly synchronized variant (S1) of dmGS, which uses only local information from neighbors, delivers the same level of accuracy as the fully synchronized dmGS. The synchronization strategy (S1), where the nodes decide about their progress based only on their neighborhood, appears to be also advantageous in terms of communication cost. Consequently, it seems to be suited for decentralized systems, where global synchronization is very expensive or impossible.

Note that we used a coarse lower bound for estimating the cost for a synchronization barrier and in real scenarios, the communication cost a global barrier and consequently, for (S3) and (S2), may be higher. Although simplified, this coarse analysis serves for giving a general picture of the aspects influencing the communication cost of the different synchronization strategies. The same conclusions can be used also for vdmGS (Section 5.3). The improved orthogonalization vdmGS differs from dmGS in the number P of DDAAs executed, but the number P does not play a role in the final comparison in terms of communication cost.

5.7 Comparison with Consensus-Based Approaches

A potential drawback of the distributed gossip-based orthogonalization algorithm dmGS (Section 5.1) is its fixed algorithmic structure computing the matrix Q column by column (in sequence). Although the communication schedules in DDAAs are flexible, the orthogonalization algorithm itself is organized in fixed for-loops (see Lines 1 and 7 in Algorithm 9 on page 65) which compute one column of Q after another. This means that computation of the ith column of Q is not started until the $(i-1)$th column of Q is finished.

In a failure-free system, such fixed structure may not bring any disadvantages. But if the computation of dmGS gets interrupted, for instance due to a temporary disconnection of the network, the dmGS algorithm cannot provide an estimate of the entire matrices Q and R, but only of their parts (rows and columns) computed before the breakdown. Note that the same problems remain also in the vdmGS algorithm.

In Sluciak et al. (2012) we presented *dc-cGS*, a consensus-based distributed *classical* Gram-Schmidt orthogonalization. Like gossip-based algorithms, dc-cGS proceed in a decentralized way without a fusion center and assumes only local communication between neighboring nodes without global knowledge about the topology. In contrast to dmGS, the dc-cGS algorithm approximates all elements of both matrices Q and R *simultaneously* and as the algorithm proceeds, the values at *all* nodes are refined iteratively and approximate the exact values of the complete matrices Q and R. Consequently, dc-cGS can deliver an estimate of whole result, both matrices Q and R *at any moment* of the computation.

In order to estimate the full result at once, dc-cGS needs to be based on *classical* Gram-Schmidt orthogonalization, which is numerically less stable than the mGS. As shown in Section 2.1, in modified Gram-Schmidt orthogonalization the input matrix V gets updated after each computed column of Q. Because the input matrix V depends on the previously computed columns, it cannot be used for delivering the full result at once. Due to the fact that the classical Gram-Schmidt method works all the time with the same input matrix V, it could be used for designing the new algorithm estimating all elements of the new orthogonal basis simultaneously.

Another difference from dmGS is the considered distributed aggregation algorithm. The dc-cGS algorithm is based on a dynamic consensus algorithm (see Section 3.2.5), which is designed for processing dynamic time-varying signals. In our case, the time-varying input are the refined approximations of the 2-norm and the dot product needed in the Gram-Schmidt orthogonalization for computing the matrices Q and R.

The dc-cGS algorithm has the same input and output matrices as dmGS, namely matrix $V \in \mathbb{R}^{n \times k}$ as input and it returns matrices $Q \in \mathbb{R}^{n \times k}$ and $R \in \mathbb{R}^{k \times k}$. The matrices V and Q are distributed row-wise across the network, meaning that each node stores block of rows of the matrix V, corresponding rows of the matrix Q and an estimate of the entire matrix R. In case where $n > N$, more rows must be stored at the node and each node sums the data locally before broadcasting to its neighbors, analogously to gossip-based approaches (Section 4.2).

Although using dynamic consensus as the underlying algorithm makes it possible to deliver an approximation of the the QR factorization at any moment, dc-cGS suffers from different disadvantages than dmGS. First, drawback of dc-cGS is the requirement of synchronously working nodes, originating from the consensus algorithm, which communicates with all neighbors at once, instead of choosing one random communication partner. Second, the algorithm is based on the classical Gram-Schmidt orthogonalization, it is substantially less numerically stable compared to dmGS, as illustrated in the next section.

Simulation Setup

We compare the dmGS algorithm with push-sum, which is based on the modified Gram-Schmidt orthogonalization and several methods based on classical Gram-Schmidt orthogonalization: dcGS based on the push-sum algorithm, dc-cGS utilizing dynamic consensus and also the algorithm denoted as *sc-cGS* using static consensus (Section 3.2.5). For comparisons in terms of communication cost we used the vectorized versions of the gossip-based QR factorization — vdmGS.

We compare the distributed algorithms for QR factorization in terms of two aspects:

(i) the influence of the condition number $\kappa(V)$ of the input matrix V on the orthogonality of the computed matrix $\hat{Q}$.

(ii) the communication cost in terms of the number of messages as well as the amount of data sent.

5.7.1 Numerical Stability

As mentioned earlier, the classical Gram-Schmidt orthogonalization process has worse numerical properties than the modified Gram-Schmidt method (see e. g., Trefethen and Bau (1997) or Golub and Van Loan (1996)). In cases where the input matrix V has a high condition number $\kappa(V)$, the computed vectors $\hat{Q}$ can be far from orthogonal even when computed in a high precision.

Presented results showing the influence of the condition number $\kappa(V)$ of the

input matrix $V \in \mathbb{R}^{300\times100}$ on the final accuracy are based on MATLAB simulations. The four investigated algorithms factorized the input matrix over a fully connected network with $N = 30$ nodes, each of them storing 10 rows of the input matrix. The values shown are averages of results for ten different matrices for each condition number.

Figure 5.19 depicts the dependency of the orthogonality of the matrix Q on the condition number of the input matrix V. With growing condition number of the input matrix the orthogonality of the matrix Q is decreasing. Note that all methods based on classical Gram-Schmidt method achieve significantly worse results than dmGS based on modified Gram-Schmidt orthogonalization, which is in compliance with the theory.

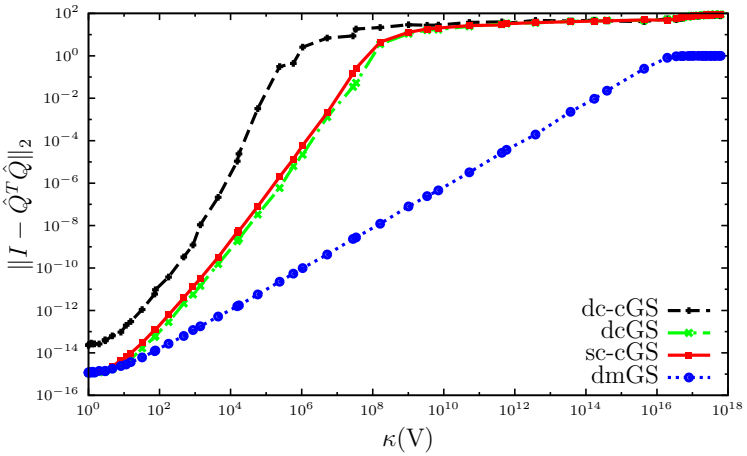

Figure 5.19: Influence of the condition number $\kappa(V)$ of the matrix $V \in \mathbb{R}^{300\times100}$ on the orthogonality of $\hat{Q}$ computed by different distributed QR factorization algorithms over a fully connected network with $N = 30$ nodes. Averaged over 10 matrices for each condition number.

5.7.2 Communication Cost

Analytically quantifying communication cost of a distributed algorithm essentially depends on the type of the communication considered, namely point-to-point communication or broadcasting (see Section 4.4). Distributed QR factorization based on gossiping communicates with one neighbor in round of the DDAA and thus its communication cost is always the same regardless of the type of the communication. In contrast, dc-cGS and sc-cGS communicate with all neighbors in each round. Consequently, if broadcasting is possible, only one message is required to reach all neighbors. In point-to-point communication, every node needs one message for each neighbor in every round, which makes the overall communication cost much higher compared to wireless systems. This is the only section in this thesis

where we provide communication analysis assuming broadcasting and *not* point-to-point communication, because dynamic consensus is developed mostly for wireless networks.

The presented figures compare the number of messages and amount of data sent for all four distributed QR factorization methods when factorizing a matrix $V \in \mathbb{R}^{300\times100}$ over a regular graph with $N = 30$ and node degrees $d = 5$. We can observe that the gossip-based approaches exchange, in general, less data (Figure 5.20), but since their message size is much smaller than in dc-cGS, they send more messages in wireless networks, as can be seen in Figure 5.21 on page 99. Simulation confirmed these conclusions also for other topologies, such as fully connected or random geometric topologies.

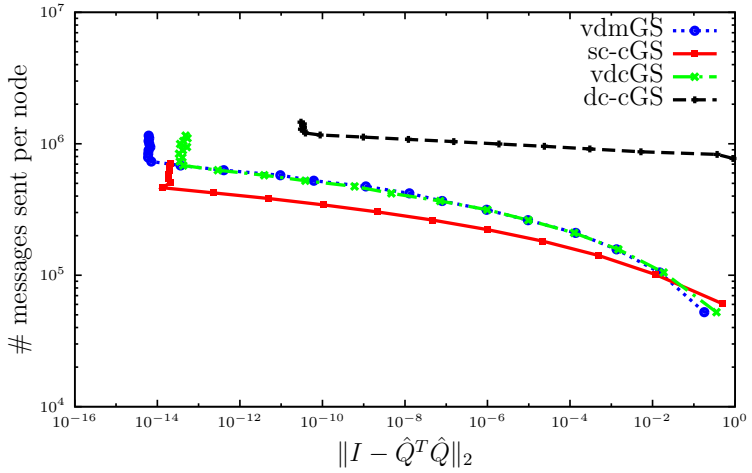

Figure 5.20: Amount of data sent per node in different distributed QR factorization algorithms factorizing a matrix $V \in \mathbb{R}^{300\times100}$ over a regular graph with $N = 30$ and node degree $d = 5$.

In wireless sensor networks it is in general more efficient to transmit as few messages as possible, because the energy consumption in a wireless sensor network is mostly influenced by the number of transmissions (Rost and Fettweis, 2010). Consequently, dc-cGS seems to be the most suitable method for wireless sensor networks in terms of energy consumption. The big drawback of dc-cGS is its pure numerical properties. In Figure 5.20 as well as in Figure 5.21 we can notice that the matrices computed by dc-cGS are significantly less accurate compared to the other methods. The reasons for such behavior have to be investigated in detail in the future.

The topology of the network has an impact on the communication cost of all the methods. In particular, it influences the number of iterations I_{sc} and I_{dc} needed for convergence in static and dynamic consensus, respectively, and the number of rounds R needed for convergence of push-sum in the gossip-based approaches. The

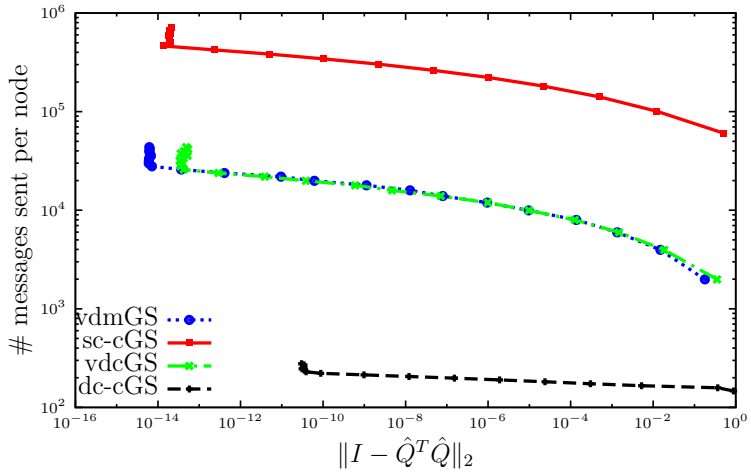

Figure 5.21: Number of messages sent per node in different distributed QR factorization algorithms factorizing a matrix $V \in \mathbb{R}^{300\times100}$ over a regular graph with $N = 30$ and node degree $d = 5$.

theoretical comparison of the communication cost of all distributed QR methods for a general rectangular matrix $V \in \mathbb{R}^{n\times k}$ is summarized in Table 5.2.

	# messages sent	# words sent	Local memory requirements
dc-cGS	$N \cdot I_{dc}$	$N \cdot I_{dc} \cdot \frac{k^2+5k}{2}$	$O(nk/N + k^2)$
sc-cGS	$N \cdot I_{sc} \cdot \frac{(k+1)k}{2}$	$N \cdot I_{sc} \cdot \frac{(k+1)k}{2}$	$O(nk/N+k^2)$
vdcGS	$N \cdot R \cdot (2k-1)$	$N \cdot R \cdot \frac{k^2+5k-2}{2}$	$O(nk/N)$
vdmGS	$N \cdot R \cdot (2k-1)$	$N \cdot R \cdot \frac{k^2+5k-2}{2}$	$O(nk/N)$

Table 5.2: Comparison of different distributed algorithms in a *wireless* system when factorizing a matrix $V \in \mathbb{R}^{n\times k}$. The number of iterations of consensus-based approaches and the number of rounds for gossip-based algorithm is denoted by I_{sc} or I_{dc} and R, respectively.

5.8 Summary of the Chapter

As the title indicates, this chapter focused on design and analysis of distributed algorithms for computing a QR factorization. We introduced *dmGS* in Section 5.1, a distributed QR factorization based on modified Gram-Schmidt orthogonalization utilizing gossip-based aggregation algorithms as building blocks. Any communication in dmGS is based exclusively on randomized information exchange and thus it offers many attractive properties.

First, we illustrated in Section 5.2 that the numerical properties and stability of the sequential algorithms are preserved also in the distributed versions, when

the sums are replaced by distributed aggregation algorithms. Consequently, dmGS has better numerical behavior in terms of orthogonality of the matrix Q than distributed classical Gram-Schmidt orthogonalization. We also showed, experimentally as well as theoretically that the final numerical accuracy of dmGS corresponds to the accuracy of the executed distributed summation. As a consequence, the numerical properties of the used DDAA directly translate to the higher level operations. Therefore, distributed matrix algorithms based on the push-flow algorithm, which has inferior numerical properties, deliver substantially less accurate results than distributed algorithms based on push-sum or on push-cancel-flow with much better numerical properties.

The push-sum algorithm does not tolerate any unreported failures, as illustrated in Section 5.5.1. We have shown that dmGS with push-sum experiences a significant loss in orthogonality of the matrix Q when messages are lost during the computation. Moreover, we discussed the behavior of dmGS in the presence of permanent node failures in Section 5.5.2 and we presented a modified version *rdmGS*, which can tolerate even several permanent node failures thanks to introducing redundancy of the original input data across the network.

The dmGS algorithm consists of a sequence of local computations and distributed summations accomplished by a distributed data aggregation algorithm. Because the nodes communicate only within the DDAA, the whole orthogonalization algorithm scales with the increasing number of nodes N like the used distributed data aggregation algorithm. The most limiting factor of dmGS in terms of scalability is the growing number of columns k. Thus, in Section 5.3 we suggested a modification of dmGS called *vdmGS*, where we replaced several calls of a scalar DDAA by one call of its vector version. This modification does not have any influence on the numerical accuracy or convergence behavior of the distributed orthogonalization, but it substantially improves the scalability of dmGS with increasing number of columns k in terms of the number of messages sent.

Although parallel and distributed algorithms work under different assumptions, we compared the communication requirements of dmGS with existing state-of-the-art parallel algorithms in Section 5.4. The communication-optimal CAQR parallel algorithm has the lowest communication cost requirements, by a factor of $n/(\sqrt{N}/\log^2(N))$ lower than vdmGS. But vdmGS has comparable communication cost to parallel mGS in terms of messages sent as well as amount of data sent.

Gossip-based DDAAs, such as push-sum, are known for working in asynchronous networks. However, in an algorithm comprising several calls of a gossip-based DDAA, such as the dmGS algorithm, the situation is different. We suggest and investigate several synchronization strategies, which differ in the number of synchro-

nization points required. Simulations presented in Section 5.6 showed that already a very weakly synchronized strategy delivered full results in high accuracy. Because it uses only information from neighbors, it is very suitable for decentralized systems where global synchronization would be too challenging. Nevertheless, it has to be investigated, how this strategy behaves with increasing number of nodes in the network.

In Section 5.7, we reviewed a different approach to distributed QR factorization, namely the distributed algorithm *dc-cGS* utilizing dynamic consensus, and we compared it to gossip-based distributed algorithms. The main advantage of this approach is its ability to estimate the entire matrix Q at once, in contrast to dmGS, which computes the matrix Q column by column. Although dc-cGS sends a low number of messages compared to other distributed QR factorization algorithms, it has very weak numerical properties. The exact reasons for the inaccurate results have to be studied in the future.

Chapter 6

Distributed Orthogonal Iteration

We introduce and analyze a distributed eigensolver *dOI* based on orthogonal iteration using the gossip-based dmGS algorithm introduced in Chapter 5 as a building block. Orthogonal iteration (see Section 2.2) is a simple algorithm for computing the k principal eigenpairs of a matrix $A \in \mathbb{R}^{n \times n}$ and it is a good prototypical example of a matrix algorithm building on QR factorization. Thus, we can nicely illustrate the properties and the challenges of designing a more complex matrix distributed algorithm based on gossiping. In particular, we illustrate how accuracy-performance trade-offs in gossip-based data aggregation algorithms can lead to a substantial reduction in communication cost of the dOI algorithm.

This chapter mostly builds on two publications, namely Straková and Gansterer (2013) introducing the dOI algorithm and analyzing it in terms of numerical accuracy and communication cost, and Straková et al. (2013), where the fault tolerance properties and the convergence behavior of dOI have been investigated. The fault-tolerant DDAAs push-flow and push-cancel-flow were introduced in Gansterer et al. (2013) and Niederbrucker et al. (2012), respectively.

First of all, we discuss how to design a distributed matrix-matrix multiplication for dOI, an important (missing) building block for constructing a flexible and fault tolerant distributed orthogonal iteration dOI. Then, we analyze the new dOI algorithm in terms of numerical accuracy and communication cost and we discuss its convergence properties. Furthermore, we illustrate the important advantages of dOI over the existing distributed orthogonal iteration by Kempe and McSherry (2008), which we call *KOI*. The last part of this chapter is dedicated to discussing properties of dOI in the presence of failures.

6.1 Towards Highly Flexible and Resilient dOI

The orthogonal iteration algorithm consists of two parts, as described in Algorithm 3 on page 35: matrix-matrix multiplication and QR factorization. Kempe and McSherry (2008) replaced QR factorization in their distributed orthogonal iteration by a mathematically equivalent sequence of a distributed summation and locally computed matrix operations, such as Cholesky factorization and solving a linear system (see Algorithm 6 on page 49). Consequently, a preliminary version towards our new distributed orthogonal iteration dOI is established by replacing this sequence of operations in KOI (Lines 5-8 in Algorithm 6) by the gossip-based distributed QR factorization dmGS (Section 5.1). This can be done directly, as the data distribution in KOI and the one expected by dmGS fits perfectly; both matrices V and Q are distributed row-wise across the network. The main advantage of using dmGS is avoiding complex computations in nodes. Especially in simple sensor networks it cannot be assumed that the nodes can (efficiently) compute Cholesky factorization or solve a linear systems on matrices.

However, the matrix-matrix multiplication with exclusively deterministic communication schedules, which is the same as in KOI, prevents the whole algorithm from benefiting from the properties of gossip-based algorithms, such as flexibility and potential for high resilience. Moreover, due to this particular matrix-matrix multiplication the resulting orthogonal iteration works only for input matrices which have the structure of the adjacency matrix of the underlying network, if communication only with nearest neighbors is allowed (see Section 3.2.4).

As a consequence, if we want to achieve high flexibility and fault tolerance of the distributed orthogonal iteration algorithm, we have to develop a flexible distributed algorithm for matrix-matrix multiplication.

6.2 Distributed Matrix-Matrix Multiplication

It is important to point out that in this thesis we focus on developing a distributed algorithm for matrix-matrix multiplication in the specific context of the gossip-based distributed orthogonal iteration using dmGS for distributed QR factorization. This context determines especially the data distribution of the input and output matrices. Any distributed matrix-matrix multiplication algorithm in the context of dOI needs a row-wise or column-wise distributed matrix A and a row-wise distributed matrix Q as input. It produces a matrix $V = AQ$ also distributed row-wise, which is then used as input for the distributed QR factorization dmGS.

We describe three gossip-based approaches for a special case of multiplying two

matrices $A \in \mathbb{R}^{n\times n}$ and $Q \in \mathbb{R}^{n\times k}$ across N nodes. All three approaches work for multiplying general rectangular matrices $A \in \mathbb{R}^{n\times m}$ and $Q \in \mathbb{R}^{m\times k}$, but the special case with square matrix A is relevant for the distributed orthogonal iteration presented in the second part of this chapter.

Similarly to reducing the communication cost of dmGS by employing the vector push-sum in Section 5.3, we apply the principle of trading bandwidth for latency also to the distributed matrix-matrix multiplication. As a result, all three variants have the same asymptotic communication cost in terms of the total amount of data sent, but they differ in the size and number of messages sent. The concrete communication cost is provided for distributed algorithms using push-sum.

6.2.1 The dmmm Algorithm

The first version of distributed matrix-matrix multiplication called *dmmm* was introduced in Straková et al. (2013). It is based on local computations of an outer product followed by a distributed summation of matrices. The starting data distribution is column-wise for the matrix $A \in \mathbb{R}^{n\times n}$ and row-wise for the matrix $Q \in \mathbb{R}^{n\times k}$ across N nodes.

Algorithm 15 shows the dmmm algorithm for the case where $n = N$. Each node l locally computes an outer product of the column $A(:, l)$ and the corresponding row $Q(l, :)$, resulting in a local matrix $X^{(l)} \in \mathbb{R}^{N\times k}$, where

$$X^{(l)}(i, j) = A(i, l)Q(l, j),\ 1 \le i \le N,\ 1 \le j \le k,$$

as written in Line 2 in Algorithm 15. Subsequently, a data aggregation algorithm for summing matrices is called (Line 3 in Algorithm 15) to sum up all local outer products $X^{(l)}$, $1 \le l \le N$ in a distributed way to produce an approximation of the result $V^{(l)} \in \mathbb{R}^{N\times k}$,

$$V^{(l)}(i, j) = X^{(1)}(i, j) + ... + X^{(N)}(i, j) = A(i, 1)Q(1, j) + ... + A(i, N)Q(N, j),$$

in every node l for $1 \le i \le N$ and $1 \le j \le k$.

Algorithm 15 Distributed Matrix-Matrix Multiplication (dmmm)

Input: $A \in \mathbb{R}^{N\times N}$, $Q \in \mathbb{R}^{N\times k}$, A distributed column-wise, Q distributed row-wise
Output: $V^{(l)} \in \mathbb{R}^{N\times k}$ in each node l

1: in each node l **do**:
2: $\quad X^{(l)} \leftarrow A(:, i) \cdot Q(i, :)$
3: $\quad V^{(l)} \leftarrow$ matrix DDAA(X)

If $n > N$, nodes may store several columns of $A \in \mathbb{R}^{n \times n}$ and the corresponding rows of $Q \in \mathbb{R}^{n \times k}$. Before calling the DDAA, every node computes the outer product (Line 2 in Algorithm 15) of every locally stored column of A and the corresponding row of Q, and sums all resulting matrices. More precisely, assume that a node l stores n_l columns $A(:, l_1), ..., A(:, l_{n_l})$ and n_l rows $Q(l_1, :), ..., Q(l_{n_l}, :)$. Then the input matrix $X^{(l)}$ for the distributed summation (Line 3 in Algorithm 15) is locally computed as

$$X^{(l)} = \sum_{j=1}^{n_l} A(:, l_j) Q(l_j, :).$$

Note that dmmm computes in each node l an estimate $V^{(l)}$ of the entire matrix V, although only the lth row is needed later in the orthogonal iteration process as input for the dmGS algorithm (or several rows, if $N < n$). It introduces an additional overhead, but this redundancy improves the robustness of the algorithm since the data is naturally replicated across nodes. Thus losing data from nodes would not be critical for proceeding in the computation. Due to its structure with only a single call of the DDAA, dmmm is able to deliver an estimate of the *entire* matrix V in every round and in every node. If the dmmm algorithm gets (temporarily) interrupted, every node can continue with the currently computed local estimate of the whole matrix V, if reduced accuracy is acceptable.

Communication Cost of dmmm

The dmmm algorithm using push-sum sends the same number of messages regardless of whether $N = n$ or $N < n$, because every node passes one matrix to the data aggregation algorithm in both situation. In contrast, the total amount of data differs, because the size of the matrix $X^{(l)} \in \mathbb{R}^{n \times k}$, sent by each node l, depends on the size of the input matrix $Q \in \mathbb{R}^{n \times k}$. The push-sum algorithm is called only once during dmmm and it executes R rounds in total. In every round, each node l sends one message of size $nk+1$, which contains a matrix $X^{(l)} \in \mathbb{R}^{n \times k}$ and a scalar weight. Obviously, the size of the messages exchanged in dmmm grows with increasing size n of the matrix A and the number k of computed eigenvectors. For large input matrices A and many computed eigenpairs, the matrices $X \in \mathbb{R}^{n \times k}$ may become too big to be stored locally. Furthermore, also exchanging such large messages may cause technical problems due to their sheer size. On the other hand, dmmm has low communication cost in terms of numbers of messages sent compared to the other gossip-based approaches, which is summarized in Table 6.1 on page 111.

To summarize, the dmmm algorithm is designed for situations, where matrices of size $n \times k$ can be stored locally and also exchanged efficiently over the network.

In particular, the dmmm algorithm is well suited for matrix-vector multiplication, where the size of messages is reduced to $n \times 1$.

Numerical Accuracy of dmmm

Analogously to the error analysis of distributed QR factorization provided in Section 5.2.3, we analyze the numerical accuracy of the distributed matrix-matrix multiplication dmmm. We use an existing error analysis of an algorithm multiplying two matrices $A \in \mathbb{R}^{n \times m}$ and $B \in \mathbb{R}^{m \times k}$, when computing the product $C = AB$ with machine epsilon ϵ_{mach}. The normwise forward error bound on the error of matrix $C \in \mathbb{R}^{n \times k}$ is given in (Higham, 2002, Section 3.5):

$$\|C - \hat{C}\|_p \leq \gamma_k \|A\|_p \|B\|_p, \quad p = 1, \infty, F \qquad \gamma_k = k\epsilon_{mach}/(1 - k\epsilon_{mach}), \tag{6.1}$$

where $\hat{C}(:, j) = (A + \Delta A_j)B(:, j),\ |\Delta A_j| \leq \gamma_k |A|$.

Obviously, if we set the working accuracy ϵ_w of the DDAA equal to ϵ_{mach} and we assume that also other computations are carried out in precision $\epsilon_w = \epsilon_{mach}$, the bound on the error in the computed matrix $\hat{C}$ in Equation (6.1) applies also to the dmmm algorithm.

We study the numerical properties also experimentally. In particular, the impact of the choice of the DDAA on the final relative accuracy of dmmm. Figure 6.1 shows the accuracy $\|\hat{V} - AQ\|_\infty / \|AQ\|_\infty$ for computing the distributed matrix product of two randomly generated matrices $A \in \mathbb{R}^{1024 \times 1024}$ and $Q \in \mathbb{R}^{1024 \times 32}$ over a hypercube with $N = 2^h, h = 5, .., 8$ nodes and target accuracy $\epsilon = 10^{-15}$ of the DDAAs.

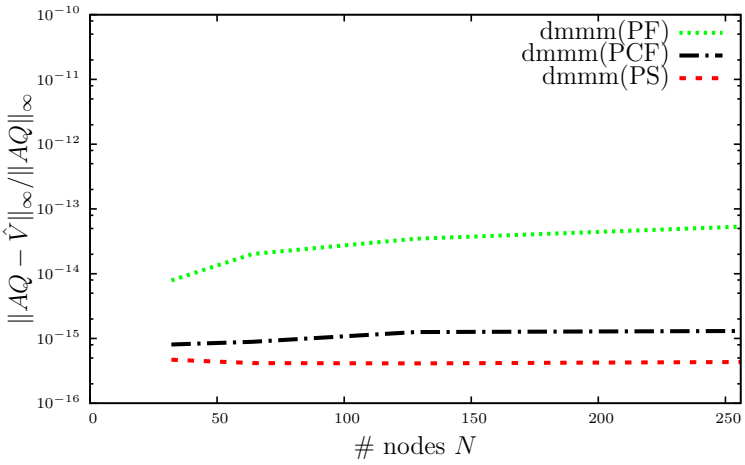

Figure 6.1: Numerical accuracy of results computed by distributed matrix-matrix multiplications (dmmm) using push-sum (PS), push-flow (PF) and push-cancel-flow (PCF) multiplying random matrices $A \in \mathbb{R}^{1024 \times 1024}$ and $Q \in \mathbb{R}^{1024 \times 32}$ over a hypercube with N nodes and with accuracy $\epsilon = 10^{-15}$ of DDAAs.

Similarly to dmGS in Section 5.2.2, dmmm using push-cancel-flow is basically as accurate as dmmm using push-sum, whereas the result computed by dmmm with push-flow is more than one order of magnitude worse and the quality even slightly decreases with increasing N. Thus, we further omit detailed investigation of matrix algorithms based on the push-flow algorithm.

6.2.2 The Scalar dmmm Algorithm (sdmmm)

In contrast to dmmm, which exchanges as large messages as possible, the second approach called *scalar* dmmm (*sdmmm*) sends the smallest possible messages. Whereas in dmmm each node l computes the whole matrix $V^{(l)}$ at once by a single call of a matrix DDAA, sdmmm computes the output matrix $V = AQ$ element by element in two nested for-loops, where a scalar DDAA is called nk times in total. The data distribution is the same as for dmmm, namely the matrix A distributed column-wise and the matrix Q distributed row-wise.

Algorithm 16 summarizes the sdmmm algorithm for a scenario where every node stores one column of A and the corresponding row of Q, i. e., $n = N$. Before each call of the DDAA, every node l computes a scalar product $x_l(i,j)$ of the corresponding elements $A(i,l)$ and $Q(l,j)$ (Line 4 in Algorithm 16), which are then summed up across all nodes, resulting in one element $V^{(l)}(i,j)$ of the final matrix $V^{(l)}$, where:

$$V^{(l)}(i,j) = x_1(i,j) + ... + x_N(i,j) = A(i,1)Q(1,j) + ... + A(i,N)Q(N,j).$$

Algorithm 16 Scalar Distributed Matrix-Matrix Multiplication (sdmmm)

Input: $A \in \mathbb{R}^{N\times N}$, $Q \in \mathbb{R}^{N\times k}$, A distributed column-wise, Q distributed row-wise
Output: $V^{(l)} \in \mathbb{R}^{N\times k}$ in each node l or
$V \in \mathbb{R}^{N\times k}$ distributed row-wise

1: in each node l **do**:
2: **for** $i = 1, .., N$
3: **for** $j \leftarrow 1, .., k$
4: $x(l) = A(i,l) \cdot Q(l,j)$
5: $V^{(l)}(i,j) \leftarrow$ DDAA(x)
6: **endfor**
7: **endfor**

The distributed data aggregation algorithm estimates each element of V in all nodes, and therefore, each node computes an approximation of the whole matrix V, as in dmmm. However, because the elements of V are computed one at a time, each node l can potentially store only those elements which are part of the lth row of V and immediately delete every $V^{(l)}(i,j)$, where $i \neq l$. Thus, in contrast to dmmm,

the nodes do not necessarily need a local storage of size $n \times k$.

If $N < n$, every node computes and sums the scalar product (Line 4 in Algorithm 16) for all locally stored elements of a column of A and of a row of Q. If a node l stores n_l columns $A(:, l_1), ..., A(:, l_{n_l})$ and n_l rows $Q(l_1, :), ..., Q(l_{n_l}, :)$, the input $x(l)$ for the DDAA (Line 5 in Algorithm 16) is computed locally for each $1 \leq i \leq N,\ 1 \leq j \leq k$ as

$$x(l) = \sum_{b=1}^{n_l} A(i, b)Q(b, j).$$

Communication Cost of sdmmm

As discussed above, the scalar DDAA in sdmmm is called nk times in order to sum $x(l)$ for all $1 \leq i \leq n$ and for $1 \leq j \leq k$ across all nodes l. Note that all instances of the DDAA are independent and could be called simultaneously.

For multiplying the matrix $A \in \mathbb{R}^{n \times n}$ with the matrix $Q \in \mathbb{R}^{n \times k}$ by sdmmm using push-sum, each node sends nkR messages, where R is the number of rounds per push-sum. The messages have a constant size, because every message from node l contains a scalar $x(l)$ and a scalar weight for the push-sum algorithm. Therefore, the number of messages sent by sdmmm is asymptotically the same as the amount of data, namely $\mathcal{O}(nkR)$ per node.

On the one hand, the large number of messages sent in sdmmm may be inefficient in wireless sensor networks, since the energy consumption usually corresponds to the number of messages sent. On the other hand, for nodes with low storage capacities it may be infeasible to store and send large messages, which may be required in dmmm. For such situations, sdmmm offers a good solution.

6.2.3 The Vector dmmm Algorithm (vdmmm)

The *vdmmm* algorithm (shown in Algorithm 17 on page 110) represents a compromise between sdmmm and dmmm in terms of size and number of messages sent. In contrast to the previous two approaches, it assumes the input matrix A to be distributed row-wise across the nodes. Moreover, vdmmm does not exchange intermediate products of A and Q (Line 2 in Algorithm 15 on page 105 and Line 4 in Algorithm 16 on page 108), but it simply replicates the matrix Q in its original form across the network and nodes compute the products with the elements of A locally.

The goal of the vdmmm algorithm is to compute in each node l a product of the locally stored row $A(l, :)$ and the whole matrix Q, in order to compute the lth row of the result $V(l, :) = A(l, :) \cdot Q$. In vdmmm, the vector DDAA emulates a broadcasting algorithm (see Line 5 in Algorithm 17), which spreads the rows of the matrix Q across

the network. When a node b wants to broadcast its row $Q(i,:)$ to the rest of the nodes by calling a distributed summation algorithm, it sets the input vector $X \in \mathbb{R}^k$ for the DDAA to the row of the matrix Q to be broadcasted (Line 3 in Algorithm 17). The rest of the nodes in the network initialize the vector X to a vector of zeros (Line 4 in Algorithm 17). Then, by calling a distributed aggregation algorithm, each node l computes an estimate $\tilde{Q}^{(l)}(i,:) = Q(i,:) + (0,...,0) + ... + (0,...,0)$ of the row $Q(i,:)$.

The computed estimate $\tilde{Q}^{(l)}(i,:)$ is used for locally updating the row $V(l,:)$ of the matrix V in each node l:

$$V(l,:) = V(l,:) + A(l,i)\tilde{Q}^{(l)}(i,:), \text{ for } l = 1,...,N.$$

As a result, each node l holds an estimate of the lth row of $V \in \mathbb{R}^{n\times k}$ after n steps. Note that in contrast to dmmm and sdmmm, in vdmmm each node computes only those elements which are required in the node as input for dmGS. However, every node receives during vdmmm enough information to compute locally an approximation of the complete matrix V, if needed.

Algorithm 17
Vector Distributed Matrix-Matrix Multiplication (vdmmm)

Input: $A \in \mathbb{R}^{N\times N}$, $Q \in \mathbb{R}^{N\times k}$, both A and Q distributed row-wise
Output: $V \in \mathbb{R}^{N\times k}$, V distributed row-wise

1: in each node l **do**:
2: **for** $i = 1,..,N$
3: **if** $(i = l)$ $X(i,:) = Q(i,:)$
4: **else** $X(i,:) \leftarrow (0,...,0)$
5: $\tilde{Q}^{(l)}(i,:) \leftarrow$ vector DDAA(X)
6: $V(l,:) \leftarrow V(l,:) + A(l,i)\tilde{Q}^{(l)}(i,:)$
7: **endfor**

Communication Cost of vdmmm

One vector DDAA is called for broadcasting each row of $Q \in \mathbb{R}^{n\times k}$, so vdmmm calls the distributed aggregation algorithm n times in total. In vdmmm with push-sum, each of the DDAAs executes R rounds with exchanging messages of size $k+1$. Consequently, the number of messages per node is nR and the amount of data sent per node is $nR(k+1)$, which is asymptotically the same as in the previous two approaches.

6.2.4 Comparison of Communication Cost

Table 6.1 shows the trade-offs between the number and size of messages per node in the distributed approaches based on push-sum multiplying two matrices $A \in \mathbb{R}^{n \times n}$ and $Q \in \mathbb{R}^{n \times k}$ distributed over N nodes. Note that the concrete number of rounds R of the push-sum algorithm crucially depends on the underlying system's size and topology, and on the required accuracy level (cf. Section 3.2.2).

	# calls of push-sum	# messages sent per node	size of message (in real numbers)	total amount of data sent per node
dmmm	1	R	$nk+1$	$(nk+1)R$
sdmmm	nk	nkR	2	$2nkR$
vdmmm	n	nR	$k+1$	$(k+1)nR$

Table 6.1: Communication cost of algorithms for distributed matrix-matrix multiplication using push-sum multiplying matrices $A \in \mathbb{R}^{n \times n}$ and $Q \in \mathbb{R}^{n \times k}$.

Communication Cost of 2.5D Matrix-Matrix Multiplication

Communication avoiding 2.5D matrix-matrix multiplication algorithm, which we denote as *CAMM*, has been introduced by Solomonik and Demmel (2011). The algorithm assumes the input matrices $A \in \mathbb{R}^{n \times n}$ and $B \in \mathbb{R}^{n \times n}$ distributed on a processor grid of shape $(P/c)^{1/2} \times (P/c)^{1/2} \times c$. The authors presented how the algorithm takes advantage of any amount of extra memory. In particular, if $c \in \{1, 2, \ldots, \lfloor P^{1/3} \rfloor\}$ copies of the input matrices A and B are distributed over P processors, the number of messages sent is $C_{CAMM} = \mathcal{O}(\sqrt{P/c^3} + \log c)$ and the total amount of data transfered is $D_{CAMM} = \mathcal{O}(n^2/\sqrt{cP})$. Both values express communication cost along the critical path. Consequently, the more copies of the input data can be stored in the system, the fewer messages and data has to be communicated.

For comparison with the distributed algorithms we assume $c = 1$, i. e., only one copy of the data is distributed across the system, which is also known as Cannon's algorithm. In this special case, $C_{CAMM} = \mathcal{O}(\sqrt{P})$ and $D_{CAMM} = \mathcal{O}(n^2/\sqrt{P})$. We know that the dmmm algorithm sends $C_{dmmm} = R$ messages and $D_{dmmm} = (n^2+1)R$ amount of data for multiplying two square matrices of size $n \times n$ across P nodes. Number of rounds R per push-sum depends essentially on the underlying topology, but in every scenario the dmmm algorithm exchanges $R\sqrt{P}$ times more data than CAMM with $c = 1$.

6.3 The dOI Algorithm

The new *dOI* algorithm for computing k principal eigenpairs of a matrix $A \in \mathbb{R}^{n \times n}$ was introduced in Straková et al. (2013). Its basic idea to replace the original matrix computations by their distributed fully randomized versions. The structure of dOI is shown in the right part of Algorithm 18, next to sequential orthogonal iteration. The parts highlighted in red (Lines 3 and 4 in Algorithm 18) denote distributed gossip-based matrix algorithms, namely distributed matrix-matrix multiplication and distributed QR factorization. The dOI algorithm can use any of the designed variants of the distributed matrix algorithms, namely dmmm, sdmmm and vdmmm for distributed matrix-matrix multiplication and dmGS or vdmGS for distributed QR factorization. If not stated otherwise, we assume dOI based on dmmm and vdmGS. The input matrix A is distributed row-wise or column-wise, depending on the version of matrix-matrix multiplication (Section 6.2). The matrices Q_i and V_i are distributed row-wise in each iteration i.

Because our dOI algorithm uses only randomized communication, it differs from the existing distributed orthogonal iteration KOI designed by Kempe and McSherry (2008) in several aspects. KOI is specifically designed for computing eigenpairs of adjacency matrices, so it does not achieve the same flexibility with respect to the underlying hardware as dOI, it cannot tolerate any kinds of failures, as shown in Section 6.7 at the end of this chapter and it cannot exploit the accuracy-performance trade-offs offered by gossiping.

Algorithm 18
Orthogonal Iteration and Distributed Orthogonal Iteration (dOI)

Input: $A \in \mathbb{R}^{n \times n}$, arbitrary $Q_0 \in \mathbb{R}^{n \times k}$, number of iterations t
Output: eigenvectors $Q_t \in \mathbb{R}^{n \times k}$, eigenvalues $\mathrm{diag}(R_t)$, $R_t \in \mathbb{R}^{k \times k}$

1: i = 0;	1: i = 0;
2: **repeat**	2: **repeat**
3: $[V_i] \leftarrow A \cdot Q_i$	3: $[V_i] \leftarrow dmmm(A, Q_i)$
4: $[Q_{i+1}, R_{i+1}] \leftarrow qr(V_i)$	4: $[Q_{i+1}, R_{i+1}] \leftarrow vdmGS(V_i)$
5: $i \leftarrow i + 1$	5: $i \leftarrow i + 1$
6: **until** $i = t$	6: **until** $i = t$

We illustrate many properties of the dOI algorithm as well as the comparison to the existing approach KOI by simulation results obtained from MATLAB. For comparisons to KOI, we use adjacency matrices of various underlying topologies (see Section 4.3.1). For investigating the behavior of dOI, we used random symmetric matrices $A \in \mathbb{R}^{n \times n}$ with elements from the interval $[-100, 100]$, if not

stated otherwise. The first six eigenvalues of each input matrix of dOI satisfy $|\lambda_1| > |\lambda_2| > |\lambda_3| > |\lambda_4| > |\lambda_5| > |\lambda_6|$ to guarantee that the orthogonal iteration converges, since in our experiments we do not compute more than $k = 5$ eigenpairs. The matrix Q_0 is initialized with the first k unit vectors of size n and dOI always runs for a prescribed number t of iterations. The data aggregation algorithms are terminated as soon as the estimates in all nodes reach the target accuracy $\epsilon = 10^{-15}$ or the maximum number of rounds (cf. Section 4.3.2). In the following, "DP-OI" denotes classical sequential orthogonal iteration in double precision floating-point arithmetic.

6.4 Numerical Accuracy and Flexibility

We evaluate several metrics describing convergence behavior of dOI with increasing number of iterations. We measure the relative error of the computed eigenvectors $\|Q - \hat{Q}\|_\infty / \|Q\|_\infty$, and the ith residual $\|A\hat{Q}(:,i) - \hat{\lambda}_i \hat{Q}(:,i)\|_\infty / \|A\|_\infty$, where $\hat{\lambda}_i$ is the eigenvalue corresponding to the computed eigenvector $\hat{Q}(:,i)$.

Random Matrix over a Hypercube

First we study convergence behavior of dOI using push-sum, when the five principal eigenpairs of a random symmetric matrix $A \in \mathbb{R}^{64\times 64}$ are computed over a hypercube with $N = 2^6$ nodes. The ratios among the first six eigenvalues are similar, between $|\lambda_6|/|\lambda_5| = 0.828$ and $|\lambda_2|/|\lambda_1| = 0.898$. Figure 6.2 on page 114 plots the relative error of the five principal eigenvectors together with the expected convergence rate, the maximal ratio $(|\lambda_2|/|\lambda_1|)^i$ (cf. Section 2.2) with increasing number i of iterations. As can be seen, all of the eigenvectors converge with the same speed as the maximal ratio or even faster, which corresponds to the theoretical analysis of sequential orthogonal iteration. Figure 6.3 on page 114 compares the first and the fifth eigenpair in terms of the accuracy of the residual computed by dOI and by sequential orthogonalization. We observe that the residual decreases identically for both methods with increasing number of iterations.

The Adjacency Matrix of a Geometric Topology

Next, we compute by dOI the five principal eigenpairs of the adjacency matrix A_{25} of a random geometric graph G_{25}, which is depicted in Figure 6.4 on page 115. The graph G_{25} has $N = 25$ nodes, the average node degree $\bar{d} = 7.12$, the maximum node degree $d_{max} = 10$ and the minimum node degree $d_{min} = 4$. It potentially models a small mobile or sensor network, where nodes can reach neighbors in a

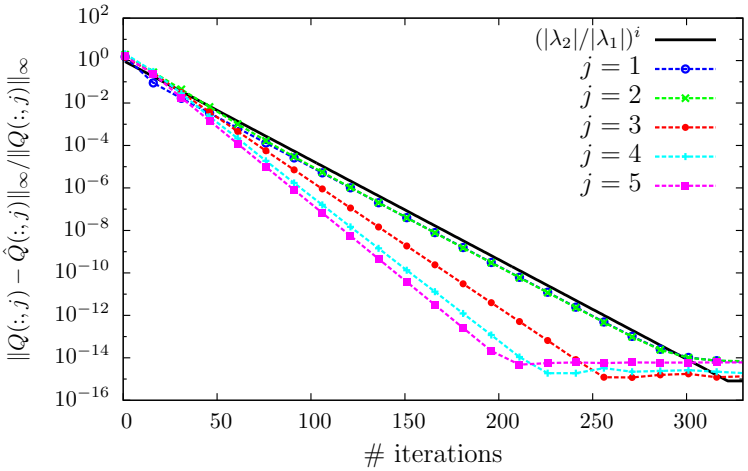

Figure 6.2: Convergence speed of the first five eigenvectors computed by dOI and the expected convergence speed of the eigenvectors of a matrix $A \in \mathbb{R}^{64\times64}$ over a hypercube with $N = 2^9$ nodes with $\epsilon = 10^{-15}$.

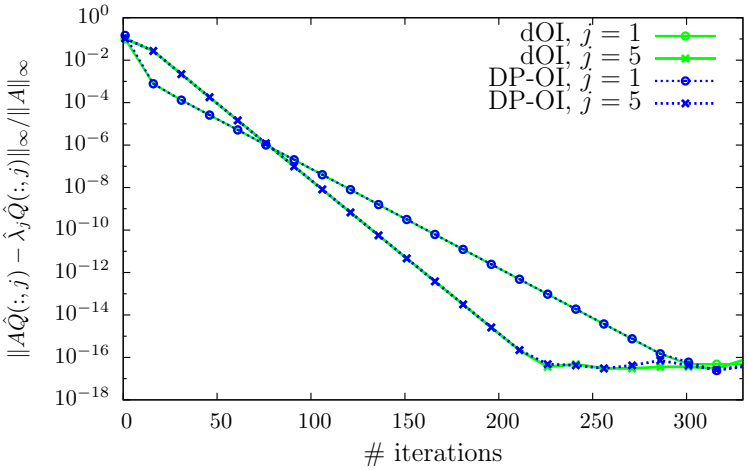

Figure 6.3: Convergence speed of the residuals of the first and fifth eigenpair computed by dOI with $\epsilon = 10^{-15}$ of DDAAs and orthogonal iteration in double precision (DP-OI).

communication range $\rho = 0.4$, and the topology is rather irregular, compared ,e. g., to hypercubes. The first six eigenvalues of A_{25} are $\lambda_1 = 7.73$, $\lambda_2 = 6.85$, $\lambda_3 = 4.09$, $\lambda_4 = -2.4$, $\lambda_5 = -2.63$ and $\lambda_6 = -2.82$, which leads to the ratios $|\lambda_2|/|\lambda_1| = 0.887$, $|\lambda_3|/|\lambda_2| = 0.5965$, $|\lambda_4|/|\lambda_3| = 0.689$, $|\lambda_5|/|\lambda_4| = 0.9352$ and $|\lambda_6|/|\lambda_5| = 0.9095$.

Figure 6.5 on page 115 shows the first five ratios $(|\lambda_{j+1}|/|\lambda_j|)^i$, $1 \leq j \leq 5$ with increasing iterations i and convergence of the fourth eigenvector corresponding to the maximal ratio of eigenvalues $|\lambda_5|/|\lambda_4| = 0.9352$. As can be seen, the convergence speed of the fourth eigenvector corresponds to the rate of the maximal ratio. The other four eigenvectors, which are not shown in the figure, converge with the same speed or faster.

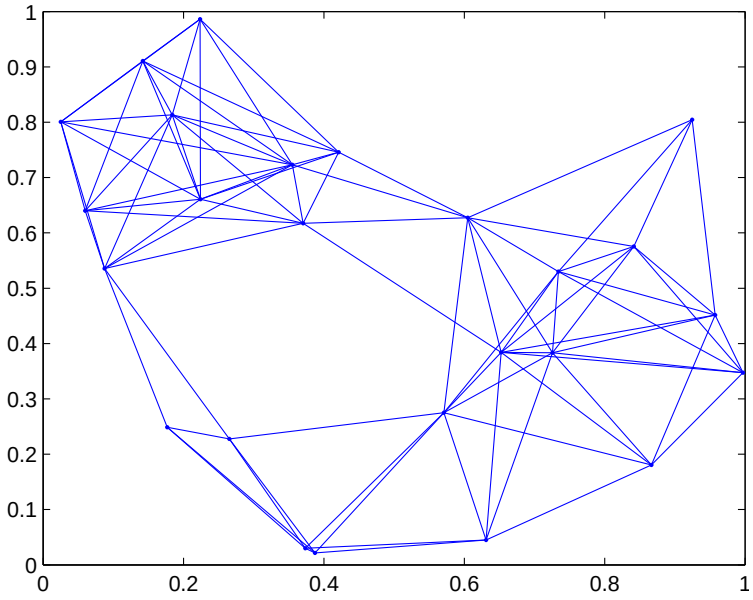

Figure 6.4: Topology of the random geometric network G_{25} with $N = 25$ and $\rho = 0.4$.

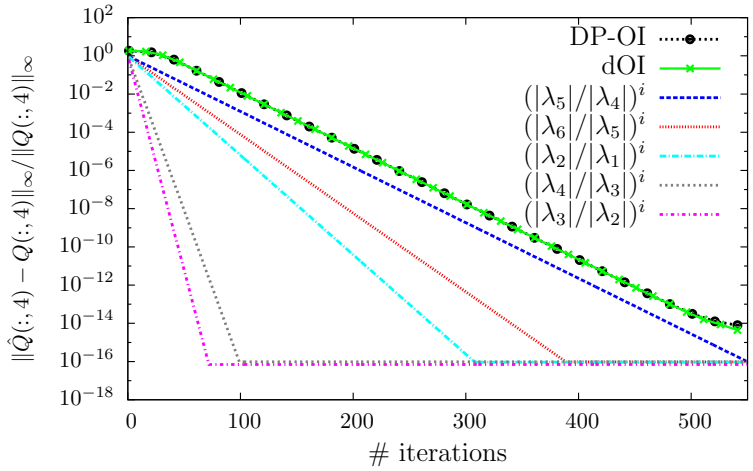

Figure 6.5: Convergence speed of the fourth eigenvector computed by dOI with $\epsilon = 10^{-15}$ and by centralized orthogonal iteration plotted next to all ratios of the eigenvalues for an adjacency matrix $A_{25} \in \mathbb{R}^{25\times 25}$ of the random geometric graph G_{25} with $N = 25$ nodes.

To summarize, all simulations (e. g., Figure 6.3 or Figure 6.5) confirmed that if the target accuracy in the push-sum algorithm is set to $\epsilon = 10^{-15}$, dOI behaves identically to the centralized orthogonal iteration computed in double precision in terms of the convergence speed of eigenvectors and residual, and that it preserves the convergence properties known from theory (Section 2.2).

6.4.1 Case Study: Graph Partitioning

Distributed orthogonal iteration can be used for example for graph partitioning (Kempe and McSherry, 2008) in various network analyses. We show how the dOI algorithm can be used in practice for finding a cut in a graph according to the signs of elements of the Fiedler vector, as described in Section 2.4. Concretely, by using dOI with push-sum we partition a random graph G_{12} with $N = 12$ nodes and random geometric topology with $\rho = 0.37$. In Figure 6.6 showing G_{12} we see that the maximal node degree is $d_7 = d_9 = 6$ and the minimal degree is $d_{12} = 2$. We assume that each node knows its neighbors, the number of nodes $N = 12$ and the maximal node degree $d_{max} = 6$ in the network.

First of all, each node l locally computes the corresponding row of the Laplacian matrix of the underlying graph $L(l,:) = D(l,:) - A(l,:)$ locally without any additional communication. Every node l knows a row $A(l,:)$ of the adjacency matrix A and as a consequence, it can construct locally the row $D(l,:) = (0,\ldots,0,d_l,0,\ldots,0))$, with the node degree d_l on the lth position in the vector.

The orthogonal iteration method computes eigenvectors corresponding to eigenvalues which are the *largest* in absolute value, and dOI preserves this property. However, we need to compute the eigenvector corresponding to the second *smallest* eigenvalue of L. To compute the eigenpairs corresponding to the smallest eigenvalues by dOI, we have to slightly modify the matrix L.

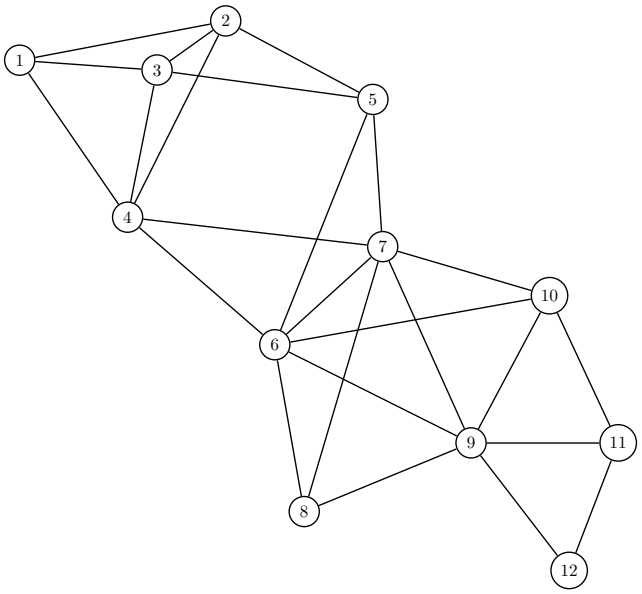

Figure 6.6: Topology of the random geometric network G_{12} with $N = 12$ and $\rho = 0.37$.

We use that the Laplacian matrix is positive-semidefinite, i. e., $\lambda_i(L) \geq 0$ for $1 \leq i \leq N$. Thus, shifting the original Laplacian matrix $L_s = L - \mu I$ leads to

a matrix L_s with eigenpairs $\lambda_i(L_s) = (\lambda_i(L) - \mu, Q(:,l))$ for $1 \leq i \leq N$ (Golub and Van Loan (1996)). If the shift μ is large enough, concretely if $\mu \geq \lambda_1(L)$, where $\lambda_1(L)$ is the largest eigenvalue of L, we reverse the order of the eigenvalues in absolute value. Therefore, dOI with $k = 2$ and input matrix L_s with $\mu \geq \lambda_1(L)$ computes the eigenvectors corresponding to the two largest eigenvalues of L_s in absolute value, which are in fact the two smallest eigevalues of the matrix L. Since the eigenvectors of L_s are identical to the eigenvectors of L, there is no need to modify the computed eigenvectors. If the original eigenvalues are needed, every node computes $\lambda_i(L) = \lambda_i(L_s) + \mu,\ 1 \leq i \leq k$.

One detail to be clarified is how to set μ so that $\mu \geq \lambda_1(L)$. Because the considered graph is undirected, the adjacency matrix A and consequently also L are symmetric. For a symmetric matrix L we know that $\|L\|_2 = \lambda_1(L)$ and $\|L\|_1 = \|L\|_\infty$ (Golub and Van Loan, 1996). From computing the Laplacian matrix $L = D - A$, we get the following bound:

$$\|L\|_1 = \|L\|_\infty = \|A - D\|_\infty \leq \|A\|_\infty + \|D\|_\infty \leq 2d_{max},$$

where d_{max} is the maximal degree in the network.

Using $\|L\|_2 \leq \sqrt{\|L\|_1 \|L\|_\infty}$ (Golub and Van Loan, 1996, Corollary 2.3.2), we get

$$\lambda_1(L) = \|L\|_2 \leq \sqrt{\|L\|_1 \|L\|_\infty} = \sqrt{2d_{max} 2d_{max}} = 2d_{max}$$

as the upper bound on the largest eigenvalue of L and consequently, an estimate for the shift μ of the matrix L. If the maximal node degree is *not* known in all nodes locally, we can use $2d_{max} \leq 2(N-1)$ to estimate μ.

Since we assume that in our case study d_{max} is known in all nodes, we shift L by $\mu = 2d_{max} = 12$ to compute the smallest eigenvalues of L. Each node l computes locally $L_s(l,:) = L(l,:) - 12e_l$, where e_l is a vector of zeros with 1 at the lth place. Then, dOI with $k = 2$ returns $\lambda_1(L_s) = -12, \lambda_2(L_s) = -11.3421$, which are the smallest eigenvalues of L shifted by $\mu = 12$. We are especially interested in the eigenvector corresponding to $\lambda_2(L_s) + 12 = \lambda_{11}(L)$, which has the elements

$$\hat{Q}(:,2) = (-0.40, -0.35, -0.35, -0.24, -0.19, 0.04, 0.04, 0.13, 0.24, 0.21, 0.39, 0.47)^T.$$

Clearly, the first five elements $Q(1{:}5, 2)$ of the eigenvector have an opposite sign compared to the elements $Q(6{:}12, 2)$ in the rest of the eigenvector. As a result, the first component of the graph consists of nodes $1, 2, 3, 4, 5$ and the second consists of nodes $6, 7, 8, 9, 10, 11, 12$. The cut between the two components computed by dOI is led between nodes 4,5 and 6,7, as shown in Figure 6.7.

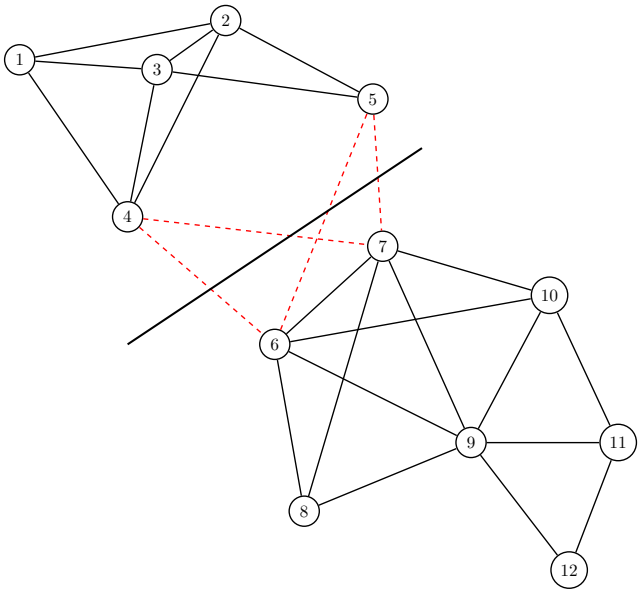

Figure 6.7: Cut in the graph G_{12} computed by dOI using the Fiedler vector.

Note that eigenvectors Q computed by dOI are distributed row-wise across the nodes, which means that every node contains its element of the eigenvector corresponding to the Fiedler value. Consequently, every node can decide immediately after dOI, to which of the components it belongs and also, comparing its value with the values of its neighbors, it can find out which connection links to the neighbors are in the cut.

6.4.2 Adaptive Algorithm adOI

Also in the case of dOI we are interested in the influence of the accuracy ϵ of the instances of a DDAA on the accuracy of the computed eigenpairs. Figure 6.8 on page 119 shows the convergence behavior of dOI based on push-sum for computing five eigenpairs of a random matrix $A \in \mathbb{R}^{512\times512}$ over a hypercube with $N = 2^9$ nodes, where the target accuracy ϵ of the distributed summation varies. First of all, the final accuracy of the residuals is about the same order of magnitude as the target accuracy ϵ of push-sum. Second aspect, which can be observed, is the that convergence speed of dOI remains the same despite different values of ϵ.

These observations motivate for starting the computation in low accuracy and making it more precise in later iterations. This seems especially attractive for gossip-based algorithms due to their ability to adjust invested cost to final accuracy of the result. The lower target accuracy ϵ in the DDAA, the fewer rounds are executed and thus, the fewer messages are exchanged.Note that KOI cannot exploit such trade-offs because of the deterministic matrix-matrix multiplication.

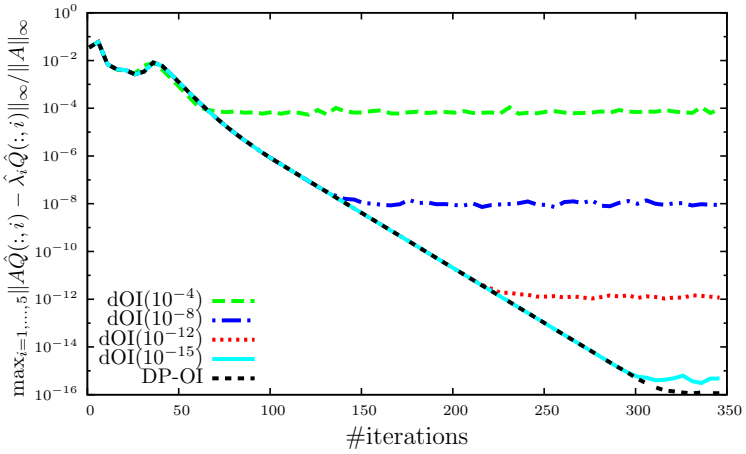

Figure 6.8: Numerical accuracy of dOI with various accuracy levels ϵ of DDAAs and DP-OI over a hypercube with $N = 2^9$ nodes with input matrix $A \in \mathbb{R}^{512\times512}$.

This trade-off is utilized in an *adaptive* dOI (*adOI*), which starts with a working accuracy $\epsilon_w = 10^{-1}$ of the DDAA and then gradually decreases ϵ_w to achieve eventually a high accuracy of the DDAA and therefore, also a high accuracy of the computed eigenpairs. We suggest two ways of deciding when to make the distributed computations more accurate.

The first strategy, denoted as *adOI-IT*, starts with $\epsilon_w = 10^{-1}$ and in regular intervals after every s iterations changes the working accuracy to $\epsilon_w = \epsilon_w \cdot 10^{-1}$ until $\epsilon_w = 10^{-16}$. The interval s between adjusting the accuracy influences the convergence speed and as a result also the communication cost of adOI-IT. In Figure 6.9 on page 120 we compare the convergence speed of adOI-IT with $s = 10, 20$, and 30, when computing the first five eigenpairs of an input matrix $A \in \mathbb{R}^{64\times64}$ over a hypercube with $N = 2^6$ nodes. We see that adOI-IT achieves the fastest convergence speed for $s = 10$ and $s = 20$ iterations. If we decrease ϵ_w every 30 iterations, adOI-IT also eventually achieves the same high accuracy of the final result as the other approaches, but it needs more iterations to converge.

The simulations indicate that if $s > t/15$, the convergence of adOI-IT is delayed. Thus, for determining an interval s where adOI-IT converges fast, the number of iterations t to converge has to be known. To overcome this drawback, we investigate a second approach *adOI-RES*, which adapts ϵ_w according to the accuracy of the computed data. It computes in each iteration i the current error of the residual

$$e_{res} = \max_{j=1,\ldots,k} \|A\hat{Q}_i(:,j) - \hat{\lambda}_j\hat{Q}_i(:,j)\|_\infty / \|A\|_\infty,$$

and if $\log_{10}(e_{res}) \leq \log_{10}(\epsilon_w)$, adOI-RES changes $\epsilon_w = \epsilon_w \cdot 10^{-1}$. Otherwise, it continues the computation without change. The idea behind adOI-RES is that it

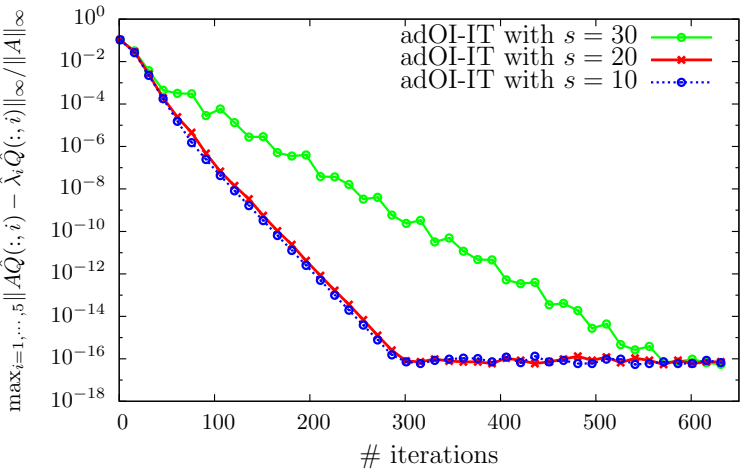

Figure 6.9: Convergence speed of adOI-IT with different intervals s for increasing the working accuracy when computing $k = 5$ eigenpairs of a random matrix $A \in \mathbb{R}^{64\times64}$ over a hypercube with $N = 2^6$ nodes.

always changes ϵ_w in the moment where it is needed, i. e., when the residual reached the working accuracy.

Obviously, adOI-RES relies on the fact that the residuals always reach the accuracy ϵ_w set in the distributed aggregation algorithm. Although the simulations indicate that this holds, it is not trivial to prove this theoretically, as discussed in Section 6.6. To guarantee that the adaptive strategy computes highly accurate results, the two adaptive approaches can be also combined. Apart from checking the accuracy of the residuals, the algorithm can set a maximal number of iterations after which the working accuracy will be increased regardless of the progress in the residuals.

Simulation results illustrating the convergence behavior of adOI-IT with $s = 22$ and adOI-RES are shown in Figure 6.10 on page 121 and compared to dOI with $\epsilon = 10^{-15}$ in every iteration. We can see that all approaches achieve the same accuracy with similar convergence speed. However, the figure does not reflect the communication cost, which is discussed in detail in Section 6.5.

6.5 Communication Cost and Scalability

We presented several variants of gossip-based orthogonal iteration and we compared them in terms of convergence speed and numerical accuracy of the delivered result. The simulations in the previous sections showed that all approaches calculate eigenpairs with similar (small) error, but an interesting and important aspect is communication cost of the individual variants needed for achieving the accuracy. We investigate dOI and adaptive dOI in terms of the number of messages sent as well as

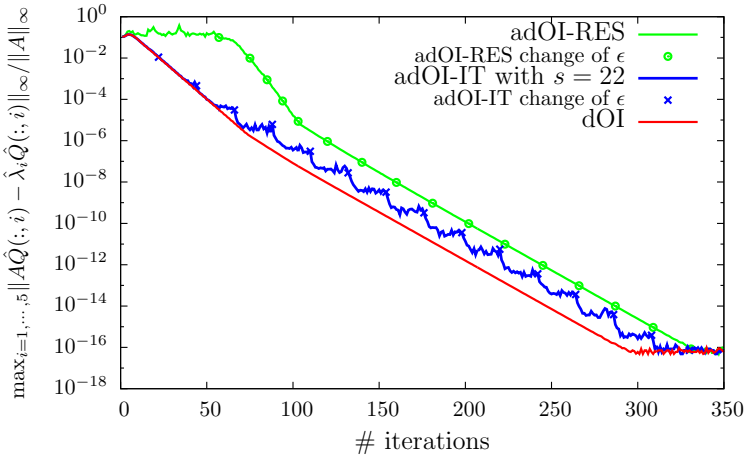

Figure 6.10: Convergence speed of adOI-IT and adOI-RES gradually adjusting the inner accuracy ϵ compared to dOI with $\epsilon = 10^{-15}$ computing $k = 5$ eigenpairs of a random matrix $A \in \mathbb{R}^{64\times 64}$ over a hypercube with $N = 2^6$ nodes.

the total amount of data (real numbers) sent for computing k principal eigenpairs of an input matrix $A \in \mathbb{R}^{n\times n}$. Moreover, we compare the gossip-based approaches to KOI.

6.5.1 Comparison of KOI and dOI

During the matrix-matrix multiplication in KOI (Line 3 in Algorithm 6 on page 49), each node sends one row of the matrix Q to all neighbors, which leads to d_l messages of size k per node l, where d_l is the degree of node l. In the second part of KOI the nodes communicate only within a consensus-based distributed summation of $k \times k$ matrices (Line 6 in Algorithm 6). DDAA-KOI proceeds in R_{CN} rounds and every node communicates with all neighbors in every round. The total number M_K of messages sent per iteration of KOI can be expressed as $M_K = 2|E| + R_{CN} \cdot 2|E|$, where $|E|$ is the number of edges (undirected links) in the network and the number of messages sent per node l is $M_K^l = d_l + R_{CN} \cdot d_l$. The total amount of data per iteration of KOI is $(2|E| \cdot k + N \cdot R_{CN} \cdot 2|E| \cdot (k^2 + 1))$. The complete overview of the communication cost of KOI is given in Table 6.2.

	Communication cost of KOI
Messages sent per node l M_K^l	$d_l + R_{CN} \cdot d_l$
Messages sent in total M_K	$2\|E\| + R_{CN} \cdot N \cdot 2\|E\|$
Data sent per node l D_K^l	$d_l \cdot k + R_{CN} \cdot d_l \cdot (k^2 + 1)$
Data sent in total D_K	$2\|E\| \cdot k + N \cdot R_{CN} \cdot 2\|E\| \cdot (k^2 + 1)$

Table 6.2: Communication cost per one iteration of KOI.

For the communication cost comparison we assume that dOI uses the most efficient building blocks in terms of the number of messages sent, namely the dmmm (see Algorithm 15 on page 105) and the vdmGS (see Algorithm 10 on page 73) algorithm, both based on push-sum. The communication cost of dmmm is summarized in Table 6.1 on page 111 and of vdmGS in Table 5.1 on page 76. Adding the cost of the individual building blocks together, we get the total communication cost of dOI and communication cost per node in dOI, all provided in Table 6.3, where R_{PS} denotes the number of rounds per push-sum.

The communication cost of both methods depends on several parameters, but the most important is the underlying topology, which has an impact on the number of nodes N, number of edges $|E|$, the degree d_l of node l, and also on the number of rounds R_{CN} and R_{PS} of the DDAAs to converge. The better connected the network is, the fewer rounds a DDAA needs to converge but the more connections $|E|$ usually are in the network.

	Communication cost of dOI
Messages sent per node M_D^l	$R_{PS} + R_{PS} \cdot (2k-1)$
Messages sent in total M_D	$R_{PS} \cdot N + R_{PS} \cdot N \cdot (2k-1)$
Data sent per node D_D^l	$R_{PS} \cdot (nk+1) + R_{PS} \cdot \frac{(k^2+5k-2)}{2}$
Data sent in total D_D	$R_{PS} \cdot (nk+1) \cdot N + N \cdot R_{DA} \cdot \frac{(k^2+5k-2)}{2}$

Table 6.3: Communication cost per one iteration of dOI with dmmm and vdmGS.

For the special case of a fully connected network with N nodes Wwe have $M_K = N(N-1) + 1 \cdot N(N-1)$, because $|E| = N(N-1)$ and $R_{CN} = 1$, since in one round of DDAA-KOI each node exchanges information with every other node. For dOI the communication cost per iteration is $M_D = N \cdot \mathcal{O}(\log N + \log 1/\epsilon) + N \cdot \mathcal{O}(\log N + \log 1/\epsilon) \cdot (2k-1)$, because $R_{PS} = \mathcal{O}(\log N + \log 1/\epsilon)$ for achieving accuracy level ϵ (see Section 3.2.2). The concrete number of messages sent depends on the size of the network, but we can see that with increasing number of nodes N for fixed k, dOI scales better than KOI, which was also confirmed by simulations (see Figure 6.11).

If the underlying topology is a hypercube with $N = 2^h$ nodes and $|E| = 2^{h-1} \cdot h$ edges, we know that $R_{CN} = R_{PS} = \mathcal{O}(\log N + \log 1/\epsilon)$ (see Section 3.2.2). Consequently, we get the total number of messages $M_K = \mathcal{O}(\log N) \cdot 2^h \cdot h$ of KOI and for dOI we have $M_D = \mathcal{O}(\log N) \cdot 2^h \cdot 2k$. If we compute a fixed (small) number of eigenvectors k, then KOI scales worse with the increasing number of nodes compared to dOI. However, for computing all $k = N = 2^h$ eigenvectors, KOI has scales better in terms of the total number of messages sent than dOI.

In many scenarios for different topologies the simulations show that dOI sends

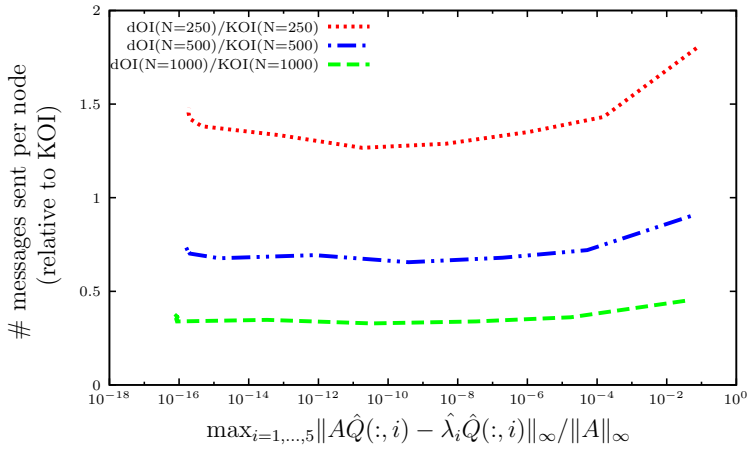

Figure 6.11: Number of messages per node in dOI(10^{-15}) relatively to KOI for computing five eigenpairs of the adjacency matrix $A \in \mathbb{R}^{N\times N}$ of fully connected networks with N nodes.

fewer messages than KOI. For example on a random graph with $N = 20$ nodes or on a regular graph with $N = 500$ nodes (Figure 6.12 on page 125). However, due to the different impact of the concrete topology on the cost of KOI and dOI, it is difficult to make a general statement. Moreover, we see that also the number of computed eigenvectors k influences the comparison. In KOI, k determines only the size of the messages in DDAA-KOI, but not the actual number of messages. In contrast, in dOI the number of messages grows linearly with increasing number of eigenvectors k, because the cost of dmGS depends on the number of columns of the input matrix. Therefore, it is to be expected that for computing a large number of eigenvectors k close to n, KOI will to scale better than dOI with increasing size of the network. However, orthogonal iteration is usually used for finding only the first few principal $k \ll n$ eigenvectors and it is not expected that it will be used for computing all n eigenvectors of the input matrix.

6.5.2 Communication Cost of adOI

Next, we show that the adaptive strategy adOI may significantly reduce the overall communication cost compared to the dOI algorithm. We know that the number of rounds executed during a distributed data aggregation algorithm is proportional to the final accuracy ϵ. Thus, every DDAA in lower accuracy requires less communication than the DDAA in double precision on the same network. Naturally, the more underlying DDAAs need to produce the result only in low accuracy, the less communication we have to invest in total.

Suppose that the goal of distributed orthogonal iteration is to compute the eigenpairs in target accuracy $\epsilon = 10^{-e},\ e \in \mathbb{N}$. The number of messages sent per node of

dOI running for t iterations is

$$C_{dOI} = t \cdot P \cdot C(\epsilon),$$

where P is the number of push-sum algorithms and $C(\epsilon)$ is the number of messages needed per node in one push-sum for computing ϵ-accurate estimates. The adOI calls the same number P of push-sum algorithms per iteration as dOI, but the working accuracy ϵ_w in push-sum is changing. We focus on analyzing adOI-IT, where the accuracy ϵ_w is changed in regular intervals $s = t/e$, where t is the number of iterations needed by dOI to deliver ϵ-accurate residuals with $\epsilon = 10^{-e}$. The number of messages sent per node in t iterations of adOI-IT with $s = t/e$ is

$$C_{adOI-IT} = t/e \cdot P \cdot C(10^{-1}) + \cdots + t/e \cdot P \cdot C(10^{-e}) = t \cdot P \cdot \sum_{i=1}^{e} C(10^{-i}).$$

Consequently, we get

$$\frac{C_{adOI-IT}}{C_{dOI(10^{-e})}} = \frac{\sum_{i=1}^{e} C(10^{-i})}{e \cdot C(10^{-e})}.$$

For well-connected networks, such as a fully connected topology or a hypercube, each node needs $C(\epsilon) = \mathcal{O}(\log N + \log 1/\epsilon)$ rounds and the same number of messages sent per node in order to achieve the accuracy level ϵ. It follows that

$$\frac{C_{adOI-IT}}{C_{dOI(10^{-e})}} = \frac{\sum_{i=1}^{e} C_1(\log_{10} N + \log_{10} 1/10^{-i} + C_2)}{e \cdot C_1(\log_{10} N + \log_{10} 1/10^{-e} + C_2)} = \frac{1 + \cdots + e}{e \cdot e} = \frac{1+e}{2e},$$

where $C_1, C_2 \in \mathbb{R}$ are the same constants for both approaches, since the underlying topology is fixed and the same DDAA is used in dOI and in adOI-IT. We see that due to the ability of push-sum to adjust the invested cost proportionally to the final accuracy, adOI-IT needs fewer messages than dOI(ϵ) if $\epsilon \leq 10^{-2}$. Moreover, the higher the final accuracy, the bigger is the advantage of adOI-IT. Suppose that the target accuracy of dOI(ϵ) is $\epsilon = 10^{-7}$. Then, the adaptive strategy adOI-IT sends $8/14 = 0.571$ of the messages required by dOI(10^{-7}). Similarly, if $\epsilon = 10^{-15}$, the ratio of the number of messages sent by adOI-IT to number of messages sent by dOI(10^{-15}) is $16/30 = 0.533$.

So far we compared the communication cost of adOI to dOI(ϵ), where the DDAAs compute ϵ-accurate estimates and ϵ is also the target accuracy of dOI. An interesting comparison is also to dOI(10^{-15}), which is also shown in simulations. In this case we have

$$\frac{C_{adOI-IT}}{C_{dOI}} = \frac{\sum_{i=1}^{e} C(10^{-i})}{e \cdot C(10^{-15})}.$$

Consequently, compared to dOI in full accuracy, adaptive strategy computing the

result in until $\epsilon = 10^{-7}$ reduces the number of messages to $\frac{C_{adOI-IT}}{C_{dOI}} = 28/105 = 0.26$.

In Figure 6.12 the number of messages of dOI and adOI-IT are plotted relatively to KOI for computing two eigenpairs of an adjacency matrix considering two different scenarios. First, we investigate a randomly generated network with $N = 20$ nodes and the average node degree $\bar{d} = 7,2$, which corresponds to a small sensor network. From the relative comparison we can see that dOI needs about 3/4 of the messages sent by KOI and adaptive dOI at most 2/5. The second comparison shows the communication cost over a random regular graph with $N = 500$ nodes, where each node has a degree $d = 40$. For this scenario dOI needs about 3/10 of the messages sent by KOI and adaptive dOI only slightly more than 1/10.

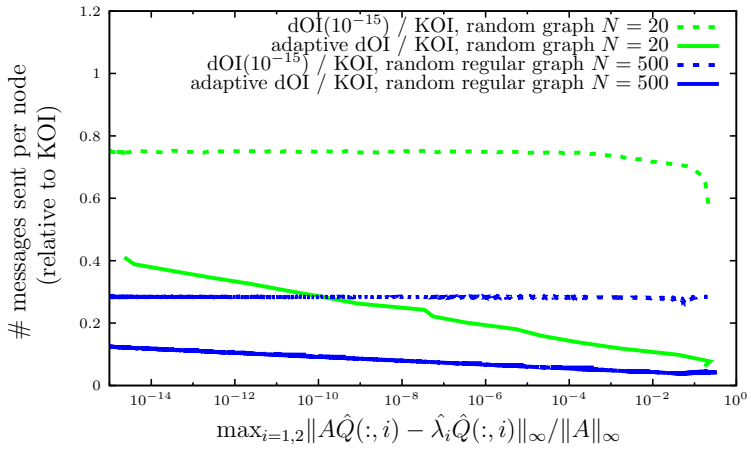

Figure 6.12: Number of messages per node in dOI(10^{-15}) relatively to KOI computing two eigenpairs of the adjacency matrix of a random regular graph with $N = 500$ and $d = 40$ and of a random graph with $N = 15$ and $\bar{d} = 7.2$.

Figure 6.13 on page 126 depicts the relative number of corresponding to the simulations shown in Figure 6.10 on page 121 comparing the convergence behavior of the two adaptive strategies adOI-IT and adOI-RES. For both adaptive algorithms the number of messages is plotted relatively to dOI computing five eigenpairs of a matrix $A \in \mathbb{R}^{64\times64}$ on a hypercube with $N = 2^6$ nodes. Clearly, the adaptive strategies send substantially fewer messages than dOI with full accuracy. For computing a result in double precision both versions of adOI require about half of the messages compared to the dOI algorithm.

For adOI-IT, we illustrated in Figure 6.9 on page 120 that the interval s has an impact on the convergence speed. However, it also influences the number of messages sent. Simulation results indicate that the most efficient in terms of communication cost is to choose $s = t/e$, where t is the number of iterations needed by DP-OI to converge and $\epsilon = 10^{-e}$ is the target accuracy. Figure 6.14 illustrates that the lowest number of messages is sent by adOI-IT with $s = 300/15 = 20$, where $t = 300$ is the

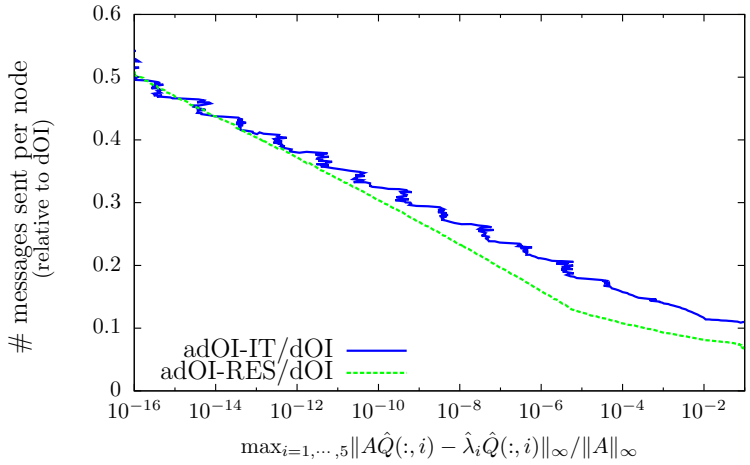

Figure 6.13: Number of messages per node in adOI-IT and adOI-RES relatively to dOI($\epsilon = 10^{-15}$) computing $k = 5$ eigenpairs of an input matrix $A \in \mathbb{R}^{64\times64}$ over a hypercube with $N = 2^6$ nodes.

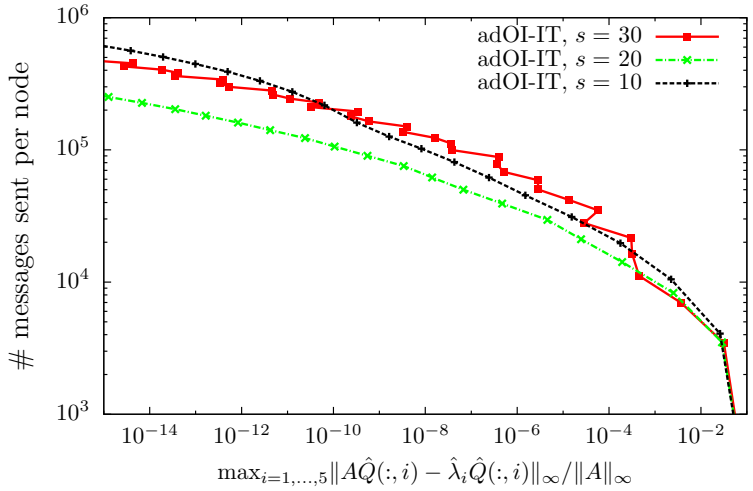

Figure 6.14: Number of messages sent per node in adOI-IT with different interval s for increasing the working accuracy level when computing $k = 5$ eigenpairs of a random matrix $A \in \mathbb{R}^{64\times64}$ over a hypercube with $N = 2^6$ nodes.

number of iteration needed by DP-OI to compute results in accuracy $\epsilon = 10^{-15}$. If $s < t/e$, adOI-IT does not exploit fully its adaptivity and sends more messages and intervals longer than t/e delay the convergence of adOI-IT (see Figure 6.9) and do not reduce the communication cost significantly.

Communication Cost of Graph Partitioning

Next, we discuss communication cost of the case study presented in Section 6.4.1, where we partitioned the graph G_{12} (Figure 6.6 on page 116) by computing the Fiedler vector, and compare it to the communication cost of KOI. In this situation

the input matrix L_s has the structure of an adjacency matrix, so the KOI algorithm can also be applied. Figure 6.15 shows the communication cost relatively to the number of messages sent by KOI. For solving this task even the dOI algorithm itself has slightly lower communication cost than KOI, but when any of the adOI algorithms is applied, the number of messages sent is substantially reduced. Note that particularly in this application only the sign of the elements in the Fiedler vector are of interest and thus, only one correct decimal place of the elements of the eigenvector is enough to be calculated by the distributed algorithm. Consequently, the adaptive strategies can set $\epsilon = 10^{-1}$ and thus reduce substantially the communication cost.

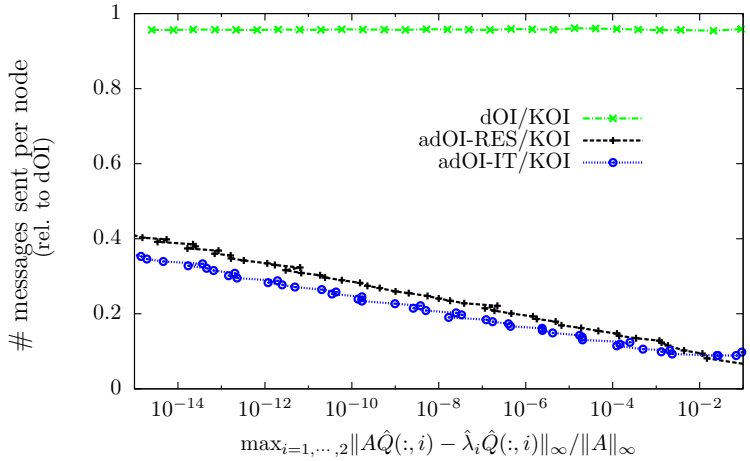

Figure 6.15: Number of messages in adOI-IT, adOI-RES and of dOI with $\epsilon = 10^{-15}$ relatively to KOI computing a graph partitioning of a random geometric graph with $N = 12$ nodes using Fiedler vector.

6.6 Convergence Behavior of dOI

Convergence analysis of the classical sequential orthogonal iteration procedure in *exact arithmetic* can be found in the standard literature, e. g., in Golub and Van Loan (1996) or Trefethen and Bau (1997). However, we are interested in the behavior of dOI in floating-point arithmetic with finite precision and in understanding the influence of the accuracy ϵ of the DDAA on the final error. Analogously to how we discussed the accuracy of distributed QR factorization in Section 5.2.3.

Kempe and McSherry (2008) provided a convergence analysis of their distributed orthogonal iteration KOI (Algorithm 6 on page 49). Note that the KOI algorithm uses a distributed aggregation algorithm DDAA-KOI for summing matrices (Line 6 in Algorithm 6), where each node communicates with all neighbors in each round. In the following summary of the proof by Kempe and McSherry (2008) we change notation of several variables so that it corresponds to the notation used in this thesis.

6.6.1 Convergence Proof by Kempe and McSherry

Before proving the convergence of KOI, Kempe and McSherry (2008) provided an analysis of DDAA-KOI, which says that after executing $\mathcal{O}(\tau_{mix} \log \epsilon^{-1})$ rounds, DDAA-KOI computes an ϵ-accurate estimate of the true aggregate, where τ_{mix} is the mixing time of the underlying Markov chain. Roughly speaking, the mixing time is the time needed for an arbitrary starting vector to come is close to the stationary distribution π of the Markov chain (Levin et al., 2006). One can see the process of spreading information over a network by a distributed algorithm as a stochastic process, which can be modeled by a Markov chain. Thus, the mixing time of the underlying Markov chain is one way how to express and study the convergence speed of a distributed aggregation algorithm.

The key theorem of the convergence proof from Kempe and McSherry (2008) states the following: Let $\lambda_1, \lambda_2, \ldots$ be the eigenvalues of a matrix $A \in \mathbb{R}^{n \times n}$, such that $|\lambda_1| \geq |\lambda_2| \geq \ldots$. Assuming that DDAA-KOI executes $\Omega(t\tau_{mix} \cdot \log(\|A\|_2 kc/\delta))$ rounds, where k is the number of eigenvectors to be computed, $\|R^{-1}\|_2$ is consistently less than c, $\delta = 2kc\|A\|_2(\epsilon^{1/4t})$ and ϵ is the target accuracy of DDAA-KOI. Then after t iterations of KOI it holds that

$$\|P_Q - P_{\hat{Q}}\|_2 \leq \mathcal{O}(|\lambda_{k+1}/\lambda_k|^t \cdot n) + 3\delta^{4t}, \tag{6.2}$$

where $P_Q = QQ^T$ denotes the projection onto the space spanned by the top k eigenvectors of A and $P_{\hat{Q}} = \hat{Q}\hat{Q}^T$ denotes the projection onto the space spanned by the top k eigenvectors of A computed by KOI after t iterations. The matrix Q is an orthonormal matrix and $\hat{Q}$ is a slightly perturbed matrix computed by KOI.

Kempe and McSherry (2008) structured the proof of this theorem in several steps. First, they showed that if Q is orthonormal and $\hat{Q}$ is a slightly perturbed matrix which fulfills $\|Q - \hat{Q}\|_F + \delta k \leq (2\|A\|_2\|R^{-1}\|_2)^{-3}$, the error of the eigenvectors introduced by one iteration of KOI is bounded by:

$$\|Q' - \hat{Q}'\|_F \leq \sqrt{k}(2\|A\|_2\|R^{-1}\|_2)^4(\|Q - \hat{Q}\|_F + \delta k), \tag{6.3}$$

where Q' and $\hat{Q}'$ are the results of one iteration of DP-OI applied on Q and one iteration of KOI applied on $\hat{Q}$, respectively.

Then, using the error introduced by one iteration (Equation (6.3)) and substituting $\epsilon = (\delta/(2kc\|A\|_2))^{4t}$ they derived that the total error introduced by t iterations of KOI is:

$$\epsilon k \cdot \sum_{i=0}^{t-1}(\sqrt{k}(2\|A\|_2\|R^{-1}\|_2)^4)^i = \mathcal{O}(\epsilon k \cdot (\sqrt{k}(2\|A\|_2\|R^{-1}\|_2)^4)^t).$$

Using the assumption that $\|R^{-1}\|_2 \leq c$ in each iteration, Kempe and McSherry (2008) showed that $\|Q - \hat{Q}\|_F$ is bounded by δ^{4t} after t iterations. Consequently, for the bound on the projection it holds that

$$\|P_Q - P_{\hat{Q}}\|_F = \|QQ^T - \hat{Q}\hat{Q}^T\|_F \leq (\|Q\|_2 + \|\hat{Q}\|_2)\|Q - \hat{Q}\|_F.$$

Because $(\|Q\|_2 + \|\hat{Q}\|_2) \leq 3$, as shown in Kempe and McSherry (2008), the bound $\|Q - \hat{Q}\|_F \leq \delta^{4t}$ and orthogonal iteration itself introduces an error bounded by $\mathcal{O}(|\lambda_{k+1}/\lambda_k|^t \cdot n)$ (Kempe and McSherry, 2008; Golub and Van Loan, 1996), the final bound in Equation (6.2) is proven.

The estimates computed by DDAA-KOI with $\Omega(t\tau_{mix} \cdot \log(\|A\|_2 kc/\epsilon))$ rounds reach an accuracy $\epsilon = (\delta/(8k\|A\|_2 c))^t$, where t is the number of the current iteration of KOI. For $\delta < 1/3$ the right hand side of Equation (6.2) goes to zero with $t \to \infty$. Moreover, from $\|A\|_2 \approx \|R\|_2$ and $\kappa(R) = \|R\|_2\|R^{-1}\|_2 \geq 1$ we get that $1 \leq \|A\|_2\|R^{-1}\|_2 \leq \|A\|_2 c$. As a consequence, the target accuracy $\epsilon = (\delta/(8k\|A\|_2 c))^t$ of the DDAA-KOI goes to zero with $t \to \infty$.

Although it is not stated explicitly, the convergence proof by Kempe and McSherry (2008) applies only in exact arithmetic. The issue in the convergence analysis is that the iteration number t of KOI is at the same time a parameter of the number of rounds required for DDAA-KOI. With growing number of iterations t, the required accuracy of the DDAA-KOI increases. As a result, DDAA-KOI has to return the aggregates in higher than double precision for all t larger than some t_0.

For a concrete example, when $k = 5$ eigenvectors of a matrix are computed, $\|A\|_2 = c = 10$ and $\delta = 1/4$, we see that in the first iteration $t = 1$ of KOI the target accuracy of DDAA-KOI is $\epsilon = 1/(4 \cdot 8 \cdot 10 \cdot 10) = 6.25 \cdot 10^{-5}$, and already in the fifth iteration $t = 5$ the accuracy of DDAA-KOI is of the order of 10^{-25}. Note that for larger norms of A and R^{-1} it gets rapidly worse. Therefore, the convergence analysis by Kempe and McSherry (2008) doest not give any answer for standard floating-point arithmetic with the computations carried out only in finite precision.

6.6.2 An Alternative Approach

Although a convergence proof along the lines of the proof by Kempe and McSherry (2008) would confirm that the dOI algorithm also converges for exact arithmetic, we are interested in the behavior for standard floating-point arithmetic. To the best of our knowledge, there is no convergence analysis of orthogonal iteration in floating-point arithmetic in the literature, which would describe the influence of the limited precision of the computations on the final error. From the error analyses summarized in Section 5.2.3 and Section 6.2 we know that the results computed

by the gossip-based building blocks dmmm and vdmGS are bounded by the target accuracy ϵ of the DDAA. However, analyzing the error of orthogonal iteration when the matrix $\hat{Q}_i$ is perturbed slightly at each iteration i is not a trivial problem and it is out of the scope of this thesis.

The desired convergence analysis for orthogonal iteration in floating-point arithmetic would show the following: If the first k eigenvalues of matrix $A \in \mathbb{R}^{n \times n}$ fulfill $|\lambda_1| > ... > |\lambda_k| > |\lambda_{k+1}| \geq |\lambda_{k+2}| ... \geq |\lambda_n|$, the principal k eigenvectors of A are computed by orthogonal iteration (Algorithm 3 on page 35) in t iterations and all computations are carried in precision ϵ_w, then the final error of the computed eigenvectors $\hat{Q}$ is also bounded by ϵ_w times a mildly growing function $p(n)$ of the problem dimension n, i. e.,

$$\|\hat{Q}_t(:,i) - Q(:,i)\| = \mathcal{O}(p(n)\epsilon_w). \tag{6.4}$$

Assuming that a proof of the bound in Equation (6.4) exists, we could conclude that if all computations are carried in accuracy ϵ_w, in particular, if all DDAAs compute ϵ_w-accurate estimates, dOI computes its results also in accuracy ϵ_w. Furthermore, it would justify the idea behind the adaptive strategy adOI-RES, where ϵ_w is decreased once the residual is ϵ_w-accurate. Such an error analysis would prove that the accuracy of the residuals always reaches the working accuracy ϵ_w. Although we do not have a theoretical proof, all experimental observations indicate that the final accuracy of the result computed by dOI corresponds to the accuracy level ϵ of the distributed summation.

6.7 Fault Tolerance Properties

Beyond the advantages of dOI discussed so far, it is potentially also more *fault tolerant* than the existing orthogonal iteration KOI. Both gossip-based building blocks of the dOI algorithm use only DDAAs with randomized communication schedules, which is the key property for achieving high flexibility and resilience against failures. The key aspect of fault tolerance properties of a gossip-based matrix algorithm is the resilience of the used DDAA. We present simulation results comparing the fault tolerance of dOI based on the push-sum algorithm (*dOI(PS)*), dOI based on the push-cancel-flow algorithm (*dOI(PCF)*) and the KOI algorithm designed by Kempe and McSherry (2008). All distributed algorithms are investigated in the presence of the following two types of failure (see Section 4.3.3):
(*i*) *reported* temporary unavailability of links or nodes
(*ii*) *silent* (unreported) loss of messages.

6.7.1 Reported Unavailability of Links/Nodes

With respect to reported failures, dOI benefits from the absence of any deterministic communication schedule between nodes, regardless of the concrete gossip-based DDAA used. If a neighbor or a communication link to a neighbor is *temporary* unavailable and this fact is known to the nodes, any of the remaining available communication partners of the node can be chosen, since the selection of the communication partner is not fixed. If a reported *permanent* link failure occurs, the algorithm communicates with the remaining neighbors. The final accuracy achieved is not influenced in any of the cases, under the assumption that the network stays connected. However, permanent failures of connections in the network change the topology of the network, which may influence the convergence speed of the DDAA and consequently of the whole dOI algorithm.

In this section, we investigate the convergence behavior of KOI and dOI(PS) in the presence of *reported* temporary link failures when computing five eigenpairs of the adjacency matrix $A \in \mathbb{R}^{150\times150}$ of a random geometric graph with $N = 150$ nodes and an average node degree $\bar{d} = 20$. For each message transmission it is independently decided, if the communication link failed (with probability p) or not.

As shown in the upper part of Figure 6.16 on page 132, for KOI even reported message losses lead to a severe loss of accuracy. Due to the fixed communication schedule of the matrix-matrix multiplication in KOI each node has to exchange information with all neighbors. Thus, it cannot adapt even to a known temporary change in the network. If a message cannot be delivered, the matrix-matrix multiplication delivers an incorrect result which substantially influences the final result of the distributed eigensolver.

In contrast to KOI, dOI has fully randomized communication schedules, which makes it possible to react to changes in the network during computation. The lower part of Figure 6.16 shows the convergence behavior of dOI with push-sum under the same conditions as for KOI, i. e., the same underlying graph as well as the input matrix and transmission failure probability p. As we can see, dOI(PS) achieves full accuracy also in the presence of reported link failures.

Nevertheless, the concrete behavior in the presence of failures also depends on the frequency of the failures. In Figure 6.16, in every iteration at least one failure occurs and thus, the KOI algorithm never converges. If only a few link failures happen during the computation, even KOI may compute results in full accuracy, as shown in Figure 6.17 on page 133. In both algorithms, in dOI with push-sum and in KOI in this case, only two reported failures happened - in the 100^{th} and in the 300^{th} iteration, always during the distributed matrix-matrix multiplication.

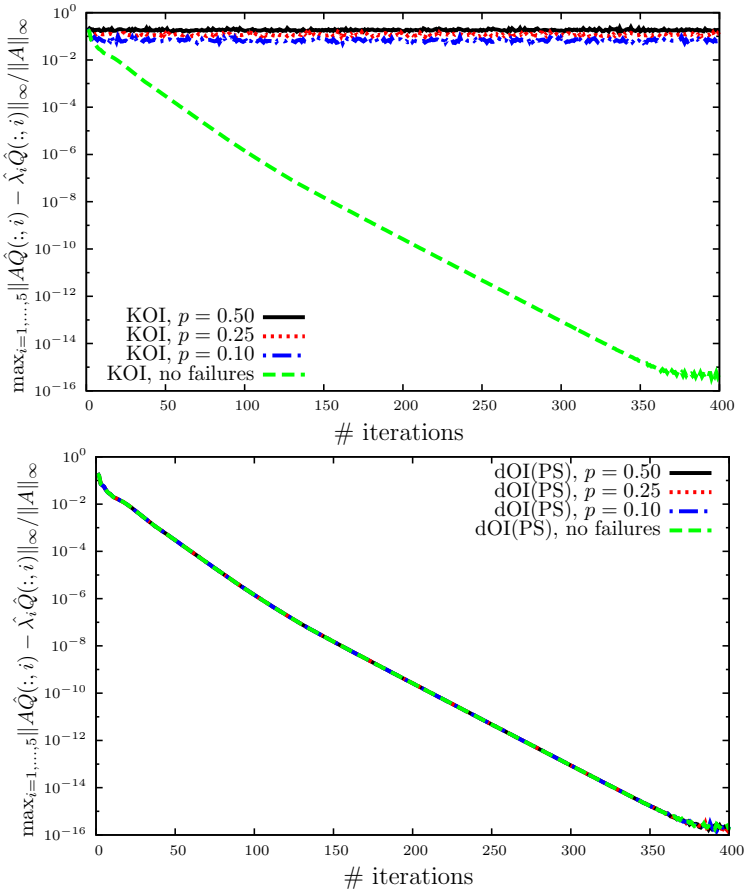

Figure 6.16: Numerical accuracy of results computed by KOI (top) and by dOI with push-sum (bottom) computing $k = 5$ eigenpairs of an adjacency matrix of a random geometric graph with $N = 150$ nodes, an average node degree $\bar{d} = 20$ in the presence of reported links failures with failure probability p and with accuracy $\epsilon = 10^{-15}$ for DDAA.

Obviously, a failure in matrix-matrix multiplication in KOI causes a big loss of accuracy in the computed result and the computation has to be almost restarted from the beginning. Because the input matrix Q for orthogonal iteration can be arbitrary, KOI again starts refining the estimate and converges eventually to full accuracy if failures do not happen frequently. However, its convergence is substantially delayed. As expected, the reported unavailability of communication links does not influence the convergence speed of dOI with push-sum.

6.7.2 Silent Message Loss

Although distributed algorithms based on push-sum can tolerate known link failures thanks to their flexible communication schedule, in practice we cannot expect that

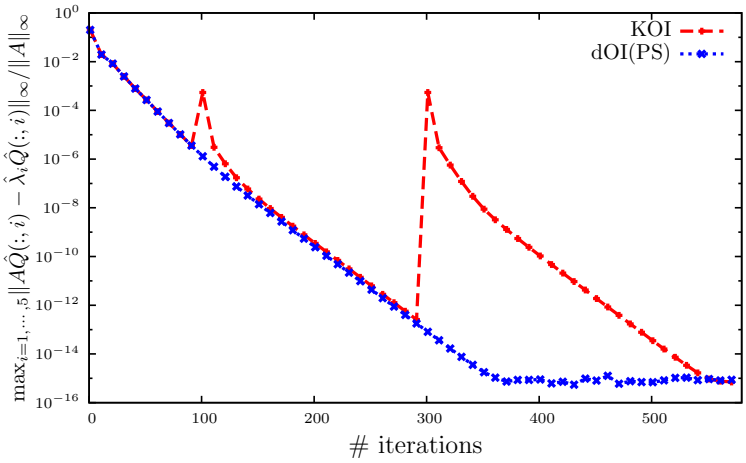

Figure 6.17: Convergence behavior of KOI and dOI in the presence of two reported link failures in the 100^{th} and 300^{th} iteration computing $k = 5$ eigenpairs of the adjacency matrix $A \in \mathbb{R}^{N \times N}$ of a random graph with $N = 150$ and an average node degree $\bar{d} = 20$.

all changes in decentralized distributed systems can be monitored and (immediately) reported. An unreported link failure and a subsequent message loss is more difficult to deal with. It usually introduces a (temporary) error from which the system has to recover without actual knowledge that the failure has happened. As shown in Section 5.5.1, push-sum and distributed algorithms built on top of it are not able to handle such unreported failures. In order to make the dOI algorithm resilient against silent message loss, we use the push-cancel-flow (Section 3.2.1) as the underlying data aggregation algorithm.

Figure 6.18 on page 134 shows the comparison of dOI(PS) and dOI(PCF) in the presence of silent message loss. The figure depicts results for computing $k = 5$ eigenpairs of an input matrix $A \in \mathbb{R}^{1024 \times 1024}$ over a hypercube with $N = 2^6$ nodes, where each message transmission failed with probability p. We clearly see that in contrast to dOI with push-sum, dOI using push-cancel-flow can deliver full accuracy despite unreported message loss. The price to be paid for these superior resilience properties is higher communication cost (see Figure 6.19 on page 135). The more failures occur, the more rounds are needed in each push-cancel-flow to recover and to converge to the target accuracy. Thus also the overall communication cost of dOI increases with higher failure probability p. Nevertheless, dOI with push-cancel-flow achieves the best fault tolerance properties of all fully distributed orthogonal iteration algorithms.

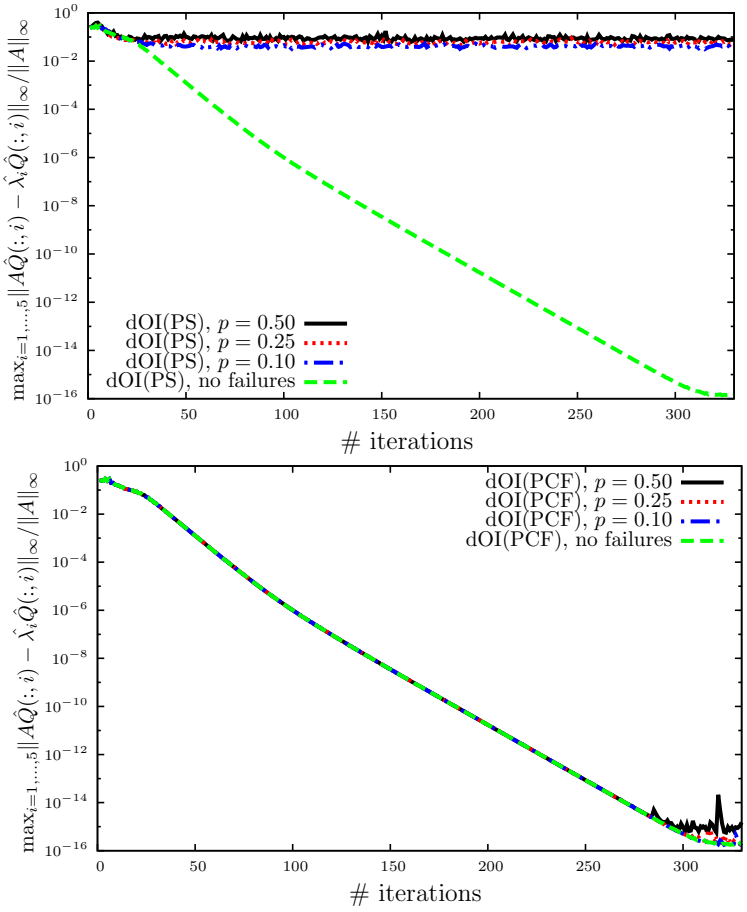

Figure 6.18: Numerical accuracy of results computed by dOI with push-sum (top) and dOI with push-cancel-flow (bottom) computing $k = 5$ eigenpairs of a random matrix $A \in \mathbb{R}^{1024\times1024}$ over a hypercube with $N = 64$ nodes in the presence of *silent* message loss with the failure probability p of each message and with accuracy $\epsilon = 10^{-15}$ in the DDAAs.

6.8 Summary of the Chapter

This chapter focused especially on illustrating how gossip-based *matrix* algorithms can be used as building blocks for a distributed eigensolver. In order to construct a distributed orthogonal iteration based exclusively on gossiping, in Section 6.2 we designed three different approaches to computing a multiplication of matrices $A \in \mathbb{R}^{n\times n}$ and $Q \in \mathbb{R}^{n\times k}$ distributed row-wise or column-wise, which differ in the number and size of messages sent. Distributed matrix multiplication based exclusively on randomized communication schedules is a necessary building block for designing flexible and highly fault tolerant distributed orthogonal iteration.

Then, we introduced dOI in Section 6.3, a distributed eigensolver originating from orthogonal iteration and utilizing gossip-based matrix-matrix multiplication

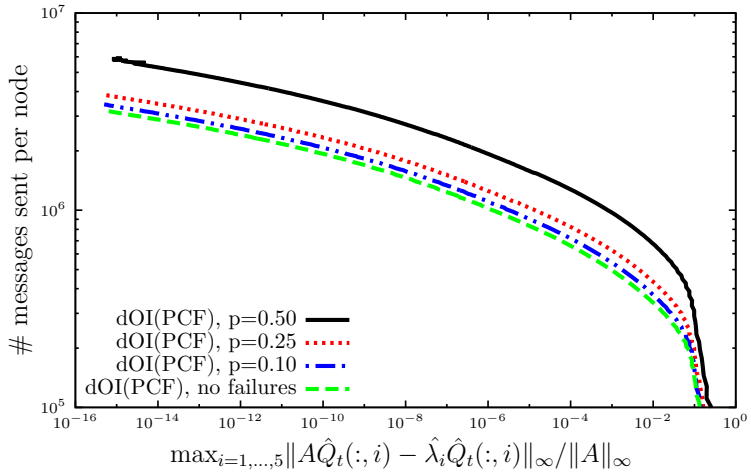

Figure 6.19: Number of messages per node in dOI with push-cancel-flow (PCF) computing $k = 5$ eigenpairs of a random matrix $A \in \mathbb{R}^{1024\times1024}$ over a hypercube with $N = 64$ nodes in the presence of *silent* message loss with the failure probability p of each message and with accuracy $\epsilon = 10^{-15}$ for DDAA.

and distributed QR factorization. The main advantage of this concept is the preservation of all attractive properties of gossip-based algorithms in the resulting algorithm, such as potential for accuracy-performance trade-offs or high robustness.

We studied the numerical accuracy and convergence behavior of the distributed eigensolver, experimentally in Section 6.4 as well as theoretically in Section 6.6. We observed that dOI preserves the numerical properties of the sequential orthogonal iteration computed in double precision and also that the behavior of dOI in terms of convergence speed of the computed eigenpairs corresponds to the theoretical analysis. Moreover, we illustrated in a case study, how dOI can be used in practice for constructing graph partitioning by computing the Fiedler vector of the Laplacian matrix of the graph. The dOI algorithm is very suitable for such computations, because the nodes have locally all information they need and only low accuracy of the result is sufficient, which may lead to substantial reductions of communication cost.

Investigating the influence of the accuracy ϵ of the DDAA on the convergence of dOI led to the conclusion that lower accuracy of the distributed summation causes lower accuracy of the overall result, but does not have any impact on the convergence speed. In Section 6.4.2 we introduced an adaptive version adOI, which starts with low accuracy and makes the algorithm more accurate with the increasing number of executed iterations. Such an approach may achieve the same convergence speed as dOI with full accuracy, but it reduces the communication cost of the whole distributed eigensolver at least twice, as discussed in Section 6.5. However, it has to be investigated what is the best strategy for locally adjusting the accuracy of DDAAs.

Although it is not trivial to compare the communication cost of dOI and KOI in general due to the complex influence of the underlying topology and number of computed eigenvectors, simulations confirmed that dOI requires in many situations fewer messages to be sent than KOI (Section 6.5). Moreover, dOI has much better resilience properties than KOI due to the absence of deterministic communication schedules. Although the push-sum algorithm does not provide explicit fault tolerance mechanisms, it can tolerate known failures during the computation thanks to its randomized communication schedules. As a result, the dOI eigensolver based on push-sum delivers results in full accuracy even if reported link occur frequently in contrast to KOI, where every message loss means a severe loss in accuracy in the output of the iteration. However, if the communication links are unavailable only rarely, KOI may also compute the results in full accuracy, but its convergence is delayed.

The push-sum algorithm and consequently also the dOI(PS) algorithm are not able to recover from unreported message loss, because of the violation of mass conservation. By replacing all push-sum algorithms by push-cancel-flow algorithms in dOI, we achieve superior fault tolerance properties of the eigensolver and we illustrated that the dOI(PCF) is "self-healing" even in the presence of many silent message losses, as illustrated in Section 6.7.

Chapter 7

Illustrative Case Study: Distributed Linear Regression

Linear regression is a particular form of regression analysis, a statistical technique for calculating a function which describes the relationship in input data (Section 2.3). We focus on investigating a distributed method for linear regression based on the distributed solution of a linear least-squares problem. Note that the purpose of this chapter is *not* to give a comprehensive overview and comparison of truly distributed methods used in data mining. The following case study on distributed gossip-based linear regression rather sketches an application scenario of the gossip-based distributed building blocks developed in this thesis.

7.1 Distributed Linear Least-Squares Solver

We consider a situation, where d independent variables $x_1, \ldots, x_d$ are observed and for each observed d-tuple there is a corresponding output variable y depending on the observations. Each d-tuple of observations together with its corresponding output value we denote as "instance". For the following computation we assume in total n sets of observations $X \in \mathbb{R}^{n \times d}$ distributed row-wise over a system consisting of $N \leq n$ nodes. In this context, a node can be a sensor node storing local measurements or a computer storing a part of a database related to persons or to products. Collecting the entire data matrix at a single node for centrally solving the least squares problem may be often impossible, due to the insufficient memory capacity in one node to store all possible data or due to privacy reasons, when all the data must not be collected at a central location. We introduce a gossip-based algorithm *dLS* based on a sequential algorithm for solving least-squares problems by using QR factorization (see Section 2.3).

The input data for the *dLS* algorithm is a matrix $X \in \mathbb{R}^{n \times d}$, $n > d$ containing n observations of $(x_1, \ldots, x_d)$ and a vector $y \in \mathbb{R}^n$ containing the output values corresponding to the n observations. The input for a linear least-squares solver is in most scenarios an overdetermined system X with $n \gg d$.

The dLS algorithm, presented in Algorithm 19, approximates a solution $w \in \mathbb{R}^d$ of the overdetermined system $Xw = y$ in a distributed way. The matrix X is distributed row-wise across the participating nodes, since each d-tuple is stored in one node together with the output value. The algorithm does not require any knowledge about the global topology of the network and it does not assume any specific connections between the nodes. The only two assumptions are that each node knows its neighbors and the total number of nodes N.

As the first step, nodes orthogonalize the columns of the matrix X by executing distributed QR factorization algorithm (Line 2 in Algorithm 19). We utilize the distributed QR factorization vdmGS introduced in Section 5.3. As described in Section 5.3, vdmGS assumes the input matrix distributed row-wise and delivers a row-wise distributed matrix $Q \in \mathbb{R}^{n \times d}$ and an estimate of the entire matrix $R \in \mathbb{R}^{d \times d}$ in every node.

The next step of the dLS algorithm is to multiply y by the transposed matrix Q^T in order to get a linear system $Rw = Q^T y$, using equations $X = QR$ and $Q^T Q = I$. In our distributed setup it simply means that every row $Q(l,:)$ stored in node l becomes a column $Q^T(:,l)$ of the transposed matrix (Line 3 in Algorithm 19). As a result, we get a matrix Q^T distributed column-wise across the nodes without executing any additional steps or introducing communication cost overhead.

Algorithm 19 Distributed Least-Squares Solver (dLS)

Input: $X \in \mathbb{R}^{n \times d}$ and $y \in \mathbb{R}^n$ distributed row-wise over N nodes
Output: $w^{(l)} \in \mathbb{R}^d$ in every node l

1: in each node l do:
2: $[Q(l,:), R^{(l)}] \leftarrow$ vdmGS(X)
3: $Q^T(:,l) \leftarrow Q(l,:)$
4: $z^{(l)} \leftarrow$ dmmm(Q^T, y)
5: $w^{(l)} \leftarrow$ solve $R^{(l)} w = z^{(l)}$ by backward substitution

Next, the distributed matrix-matrix multiplication dmmm (see Section 6.2) is called to multiply the column-wise distributed matrix Q^T with vector y in order to get $z = Q^T y$, where $z \in \mathbb{R}^d$ (see Line 4 in Algorithm 19). The distributed matrix-matrix multiplication dmmm can be applied directly in our situation, because the matrix Q^T as well as the vector y are distributed as dmmm requires.

After dmmm has finished, every node l contains locally an approximation of

the product $z^{(l)} \in \mathbb{R}^d$, because the dmmm algorithm computes an estimate of the entire product in all nodes. As shown in Line 5 in Algorithm 19, the last step is to locally solve in each node l a linear system $R^{(l)}w = z^{(l)}$ by a backward substitution, since $R^{(l)}$ is upper-triangular. As a result, every node l computes its own local approximation $w^{(l)}$ of the solution w of the original least-squares problem.

A big advantage of this particular application of gossip-based methods is the reasonable size of messages exchanged between nodes. During the QR factorization of the matrix $X \in \mathbb{R}^{n\times d}$, the largest message exchanged has a size d, if we use vdmGS. The dmmm algorithm may exchange very large messages in some scenarios, as explained in Section 6.2. However, it is well-suited for computing matrix-vector products. When multiplying a matrix $Q^T \in \mathbb{R}^{d\times n}$ with a vector $y \in \mathbb{R}^n$ using the distributed algorithm dmmm, all messages exchanged between nodes are of size $d+1$. Since the input for a linear least-squares solver is most likely an overdetermined system where $n \gg d$, it is very unlikely that sending messages of size d causes any technical problems.

7.2 Simulation Results

In order to study the accuracy of the dLS algorithm when creating a linear regression model, we first present results of dLS performing on artificially generated input data, which fulfill given requirements. Afterwards, we illustrate the conclusions on a data set downloaded from UCI[1].

Analogously to the other distributed gossip-based algorithms, the target accuracy ϵ of the called DDAA has to be specified. We compare the results computed by the dLS algorithm using the push-sum algorithm with $\epsilon = 10^{-1}, 10^{-2}, \ldots, 10^{-16}$ to the solution computed by a *centralized* approach using the QR factorization (LS-QR) and to approach computing normal equations (LS-NE) (see Section 2.3). Note that the centralized approaches do not depend on the accuracy ϵ.

For simplicity, let us assume that $N = n$. Thus, every node l contains one d-tuple $X(l,:)$ of the observed variables and the corresponding output variable $y(l)$. Moreover, the simulations assume a fully connected topology as the underlying network. However, these assumptions are not required and the dLS algorithm works for arbitrary topologies with more than one row per node.

For testing purposes we always divide the original data set $X \in \mathbb{R}^{n\times d}$ and $y \in \mathbb{R}^n$ into two parts. Randomly selected $n/2$ observations $X_m \in \mathbb{R}^{(n/2)\times d}$ and the corresponding output values $y_m \in \mathbb{R}^{(n/2)}$ are used for creating the model by calling a linear least-squares solver. The second part of the data, $X_e \in \mathbb{R}^{(n/2)\times d}$ and

[1] see http://archive.ics.uci.edu/ml/datasets.html

$y_e \in \mathbb{R}^{(n/2)}$ serves for evaluating the model created on the first part of the original data set. In order to test and compare the computed linear models we quantify the relative error $\|\hat{y}_e - y_e\|_\infty / \|y_e\|_\infty$, where $\hat{y}_e = X_e \hat{w}$ and $\hat{w} \in \mathbb{R}^d$ is a solution calculated by one of the least-squares solvers. Concretely,

$$e_{dLS} = \|\hat{y}_e - y_e\|_\infty / \|y_e\|_\infty = \|X_e w_{dLS} - y_e\|_\infty / \|y_e\|_\infty$$

is the relative prediction error of the linear model computed by dLS when applied on the input data X_e and y_e. The dLS algorithm computes in *every* node l a local estimate of the solution $w_{dLS}^{(l)}$, but for our evaluations we use the approximation of the model computed in node $l = 1$, i. e., $w_{dLS} = w_{dLS}^{(1)}$. Analogously, e_{NE} denotes the relative error of the model calculated by centralized normal equations. When evaluating the calculated models, we also quantify the difference $|e_{NE} - e_{dLS}|$ between the relative error computed by the normal equations e_{NE} and the relative error computed by the dLS algorithm e_{dLS} .

Synthetic Test Case

First, we start with two simple synthetic examples with one variable x and output variable y, where the relation between them is described by $x = y + \tau$. In the first scenario we randomly generated $|\tau| \leq 10^{-11}$, so the data is very well described by $x = y$. The second data set was generated with the parameter $|\tau| \leq 10^{-1}$. As illustrated in Figure 7.1, for the first data set with $|\tau| \leq 10^{-1}$ is the function $x = y$ much better fit than for the latter one.

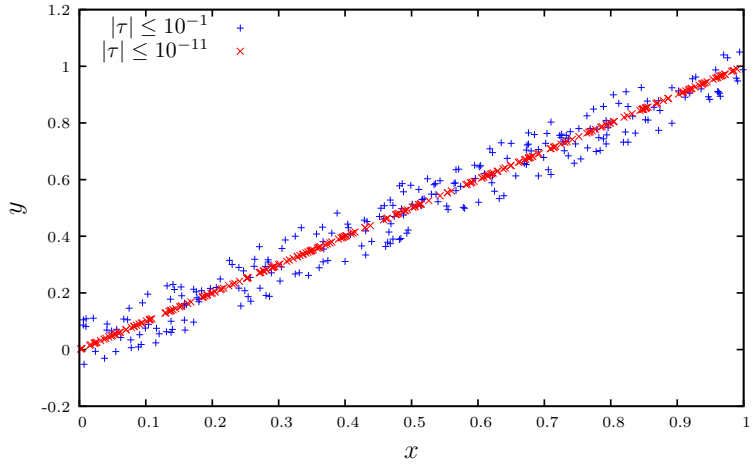

Figure 7.1: Two data sets generated to fulfill $x = y + \tau$, where $|\tau| \leq 10^{-1}$ and $|\tau| \leq 10^{-11}$.

For each of the two situations ($|\tau| \leq 10^{-11}$ and $|\tau| \leq 10^{-1}$) we generated 500 couples $(x, y + \tau)$, from which 250 randomly chosen instances were used for creating

the linear model and the remaining half was used for evaluating the model. In the first situation with $|\tau| \leq 10^{-11}$, the linear regression model $x = y$ describes the data very well and consequently, both centralized linear least-squares methods calculate a model which predicts the values y_e very precisely, as can be observed in Figure 7.2. The quality of the model computed by dLS decreases with increasing value of the parameter ϵ determining the target accuracy of the DDAAs. Because the models calculated by the centralized methods are very precise, the difference between the accuracy of the predictions computed by centralized methods and dLS are noticeable already for $\epsilon \geq 10^{-10}$. As a consequence, dLS has to compute the distributed summation with rather high accuracy to return as precise result as the centralized methods. The black line in Figure 7.2 depicts the difference $|e_{NE} - e_{dLS}|$, which nicely illustrates how the difference in the quality of the predictions calculated by the two methods corresponds to the target accuracy ϵ set in dLS.

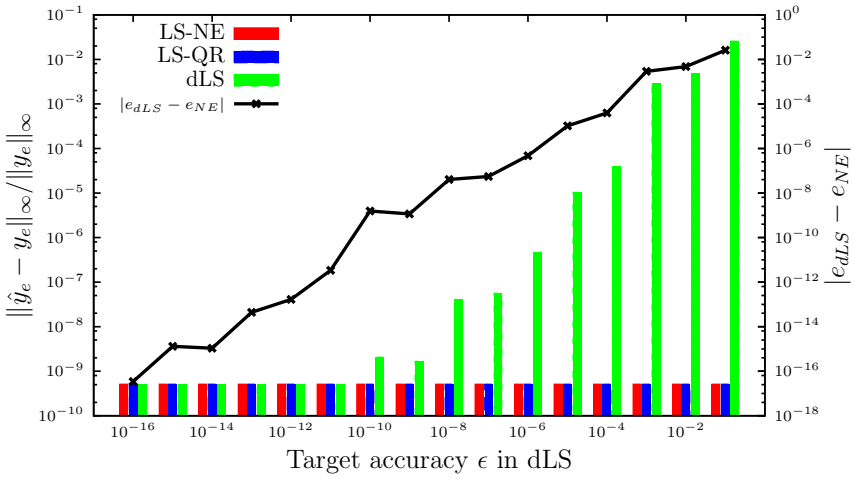

Figure 7.2: Prediction errors of estimates computed by dLS and by centralized approaches based on QR factorization (LS-QR) and normal equations (LS-NE) applied on a data set with $|\tau| \leq 10^{-11}$. The black line depicts the difference between result computed by dLS and by centralized normal equations.

In the second scenario with $|\tau| \leq 10^{-1}$ the situation is different, because a linear function does not describe the dependency between x and y as precisely as in the first case. As a consequence, linear models computed by centralized methods are not as precise as in the previous case, as can be observed in Figure 7.3 on page 142. Although the difference $|e_{NE} - e_{dLS}|$ between the relative errors of predictions by both computed models behaves analogously as in Figure 7.2, the difference in evaluating new data by the two models is not noticeable until $\epsilon = 10^{-3}$ in the dLS algorithm. As a result, the dLS algorithm does not have to compute everything up to high accuracy but in such scenario it can reduce the overall communication cost by computing every distributed summation only to lower accuracy, e. g., $\epsilon = 10^{-4}$.

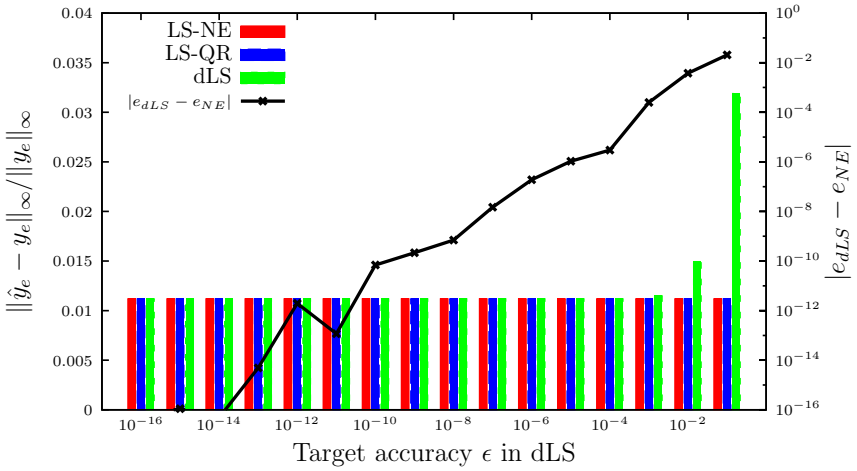

Figure 7.3: Prediction errors of estimates computed by dLS and by centralized approaches based on QR factorization (LS-QR) and normal equations (LS-NE) applied on a data set with $|\tau| \leq 10^{-1}$. The black line depicts the difference between result computed by dLS and by centralized normal equations.

Real World Case

As a second example we present the performance of the dLS algorithm on the data set "Concrete Compressive Strength"[2] from UCI storing values related to concrete compressive strength. The data set contains $d = 8$ independent variables, the amount of seven ingredients such as water or cement in the concrete mixture and its age in days. The output value is the compressive strength of the mixture depending on the independent variables. In total, the data set contains $n = 1030$ instances, from which we use 515 for creating a model and the rest for evaluating the model.

Analogously to the previous simulations, in Figure 7.4 on page 143 we plotted the quality of the model calculated by dLS compared to a model delivered by centralized approaches. We can see that due to the relatively high prediction error of the centralized approaches, the difference between the predictions computed by dLS and centralized methods starts to be significant when $\epsilon \geq 10^{-1}$. Consequently, the dLS algorithm can compute all distributed summations in very low accuracy without changing the quality of the resulting model compared to the centralized approaches. Such accuracy-performance trade-offs bring several advantages compared to dLS with high target accuracy ϵ such as significantly reducing communication cost.

7.3 Summary of the Chapter

We illustrated on a short case study how distributed gossip-based algorithms can be used for distributed solving of a linear least-squares problem arising in the context of

[2] http://archive.ics.uci.edu/ml/datasets/Concrete+Compressive+Strength

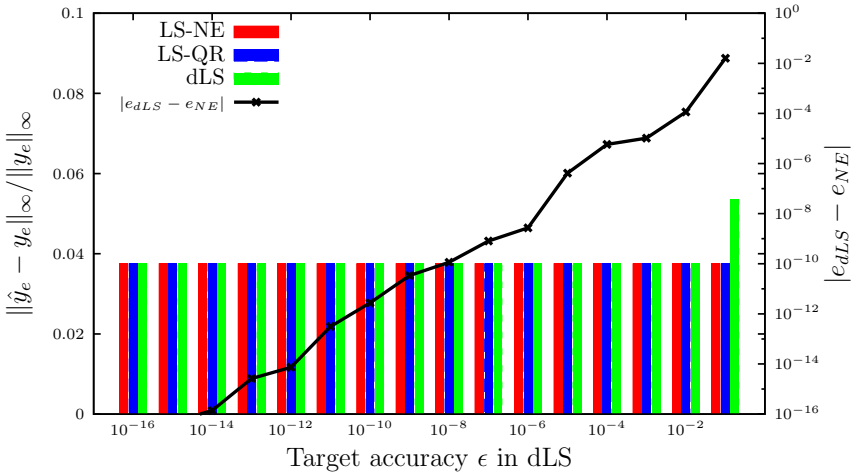

Figure 7.4: Prediction errors of estimates computed by dLS and by centralized approaches based on QR factorization (LS-QR) and normal equations (LS-NE) applied on a real data set containing variables describing types of concrete and its compressive strength. The black line depicts the difference between result computed by dLS and by centralized normal equations.

linear regression. We considered a situation where the input data for the algorithm is distributed row-wise across nodes in a network. Every participating node stores local knowledge of several independent observations and on them depending output values. The main goal of the participating nodes is to calculate a linear regression model using all data in the network without collecting the data at a single node.

We introduced the distributed algorithm *dLS* built on previously developed gossip-based QR factorization vdmGS and matrix-matrix multiplication dmmm. Besides properties common for gossip-based matrix algorithms, such as no need of knowledge of the global topology or no assumptions on particular connections, we also identified several reasons why the distributed gossip-based algorithm is particularly suitable for this scenario.

First of all, in a data mining context one expects the original data to be distributed across nodes and usually the individual nodes do not want to centralize the data, e. g., because of technical or privacy reasons. Furthermore, the data is naturally distributed row-wise across nodes which is the desired data distribution for the gossip-based building blocks developed in this thesis. In this context it is also very likely that every node stores more than one observation and thus more rows of the input matrix which is also easy to handle by the introduced gossip-based algorithms. Another positive aspect of the dLS algorithm is that it computes an approximation of the regression model in all nodes. As a result, after the algorithm is finished, every node can evaluate new incoming observations independently of the other nodes with the locally computed model.

We simulated the dLS algorithm for three different scenarios and compared the calculated model to the result computed by two centralized methods. The simulation results confirmed that the better the input data can be expressed by a linear function, the more precise are the predictions of the calculated model, and thus the higher accuracy is needed by the DDAAs in dLS in order to return results similar to the centralized computation. However, in real world scenarios it is *not* to be expected that a linear function fits the input data perfectly. Consequently, target accuracy ϵ lower than double precision will often be sufficient for the data aggregation algorithms called during the dLS algorithms which significantly reduces communication cost and thus runtime compared to computing a solution in double precision. The disadvantage of the dLS algorithm is that it is not known in advance which accuracy ϵ of the dLS algorithm will be sufficient, and thus it is difficult to exploit the property of trading final accuracy of the result for invested cost without any prior knowledge or experience.

Nevertheless, the illustrative case study served for a giving a brief insight into an application utilizing the attractive properties of the gossip-based algorithms. Comprehensive and detailed analyses of different approaches to distributed least squares as well as comparisons to existing distributed approaches for linear regression is a scope of future work.

Chapter 8

Towards MPI Implementation of Gossip-Based Algorithms

In order to investigate the performance of gossip-based algorithms in terms of runtime on a classical parallel machine, we provide first insights into how they can be implemented in MPI. We chose the message passing interface (MPI), because it is a standard environment for developing parallel programs for high-performance computers providing the interface for exchanging messages between processes. The MPI background needed for this chapter, such as the description of routines, is summarized in Appendix A.

First, we discuss difficulties when implementing gossip-based algorithms in MPI and we present several variants of possible implementation strategies of the push-sum algorithm (see Section 3.2.2). Then, we present first runtime comparisons of the push-sum algorithm with a standard parallel approach `MPI_Allreduce` and we investigate scaling behavior in MPI with growing number of processes and increasing size of messages. The collective communication operation `MPI_Allreduce` is a frequently used routine for computing an aggregation (e. g., sum or average) of values distributed over several processes and it delivers the result to all participating processes. Obviously, the push-sum algorithm can be considered a distributed counterpart of the parallel reduction `MPI_Allreduce`. Moreover, according to theory, the push-sum algorithm and parallel reduction algorithms scale similarly with the increasing size of the system (Thakur et al., 2005). We also compare a gossip-based matrix-vector multiplication to a PBLAS routine called `pdgemv` used commonly in ScaLAPACK (see Blackford et al. (1997)).

Note that in the previous chapters we considered the distributed data for a gossip-based algorithms to be distributed across nodes, because the primary target systems are loosely-coupled systems, such as sensor networks. However, in the context of

parallel computers the term "node" has a different meaning. Thus, in this chapter we use the terminology common for HPC computing. The important entity for a distributed algorithm on a parallel machine is a process, since the initial data for the routine is distributed across N processes, regardless of the number of the physical nodes in the machine.

8.1 Motivation

The properties of future extreme-scale computers are predicted to be different from parallel computers used nowadays. It is anticipated that due to the extreme number of components, the mean time between failures will be much shorter than on today's systems, and system failures will occur several times a day (see Cappello et al. (2009)). As a consequence, handling different kinds of failures is expected to become one of the essential aspects of algorithms designed for future parallel systems.

In today's parallel computers, failures in the systems or in the computations are handled mainly by mechanisms which do not change the underlying algorithm, such as checkpoint/restart (C/R) or redundant executions of the computation (see Section 3.1). However, current mechanisms very likely become insufficient for future systems because of their poor scalability with the growing number of nodes (see Riesen et al. (2012), Varela et al. (2010)). In addition, the extreme-scale systems will be difficult to synchronize globally across nodes and processes. As a result, there is a need for either essentially modifying the current mechanisms and improve their scalability, or investigating brand new approaches.

Based on the research presented in the previous chapters we see that gossip-based algorithms offer several properties which may be attractive also for future large-scale systems. They have a big potential for high fault tolerance and also they do not need full synchronization across processes. However, it has to be investigated first, how gossip-based distributed algorithms behave on high-performance machines in terms of runtime and scalability.

8.2 Experimental Setup

We used two parallel machines for running experiments to get the first insights into the performance of the gossip-based algorithms. For experiments on a small number of processes we used a cluster at Georgia Institute of Technology called Jinx[1], where OpenMPI 1.4.3 is installed. The second machine we used for experiments

[1] http://support.cc.gatech.edu/facilities/instructional-labs/jinx-cluster

is the NERSC'S Cray XE6 called Hopper[2], a high performance parallel computer consisting of in total 6384 nodes, each with 24 cores. The MPI version used on this computer is Cray MPICH2. For each experiment we reserved enough resources so that the number of processes was smaller than the number of cores reserved. At Jinx we ran jobs with at most 96 processes, because the maximum number of nodes reserved is eight, each with two six-core processors. At Hopper each node has 24 cores and thus, for N processes always the smallest number d of nodes was reserved, where $N \leq 24d$.

All MPI experiments are run on N processes. Each process gets a vector `local_data` $\in \mathbb{R}^{k+1}$ as the input, where the first element `local_data[0]` is initialized by the scalar weight (Section 3.2.2) and the remaining k elements contain the values of every process to be summed up by push-sum. If we run a scalar push-sum (Section 4.1), i. e., $k = 1$, the input vector `local_data` stores two numbers: the scalar weight and the scalar value, which contributes to the final sum across the processes. In the case of a vector or matrix push-sum, k is initialized to the number of elements in the vector or matrix to be summed up. Apart from the initial data, each process gets a target accuracy ϵ, the true aggregate of the initial data `true_result`, and auxiliary parameters `MAX_ROUNDS` and `MAX_SENT_MSG` for terminating the processes, which is explained in detail in Section 8.4. The output in every process is a vector `local_sum` $\in \mathbb{R}^k$, where every element `local_sum[i]`, $1 \leq i \leq k$ of the output is an approximation of the sum $\sum_{p=1}^{N}$ `local_data`$^{(p)}$`[i+1]` across all processes p.

We do not consider any special mapping of the processes on particular cores in the system. However, to investigate the influence of connectivity of the processes on the runtime of the algorithms, we implement two virtual topologies. In the first scenario, the experiments are run over a virtual fully connected network, where every process can communicate with every other participating process. The processes are perfectly connected and the information is spread very fast among the participating processes. As an alternative, we execute the push-sum algorithm over a virtual hypercube with $N = 2^h$ nodes. The h neighbors of each node p are easily computed by taking the binary representation of the number p and changing exactly one bit, e. g., by computing binary XOR of p and any power of two smaller than 2^h. I. e., every process p communicates exclusively with processes $p \oplus 2^i$, with $0 \leq i < h = \log_2 N$, where $\oplus$ is the binary XOR. As a consequence, the hypercube topology substantially decreases the size of the neighborhood of each process compared to the fully connected network, while preserving a good connectivity between the processes for fast spreading of information. Note that in both scenarios the topology is given by

[2] http://www.nersc.gov/users/computational-systems/hopper/

the selection of communication partners in the algorithm and it is not connected to the underlying hardware infrastructure. Throughout this chapter, by "neighbors" of process p we understand the processes which send messages to and receive messages from process p regardless of the physical layout.

In our implementations the processes exchange messages serving different purposes. In order to distinguish between them, every message contains one of the three following tags stating the purpose of the message:

1. `TAG_DATA` — messages containing data for updating a process' `local data`
2. `TAG_ACK` — acknowledgment messages
3. `TAG_TERM` — messages signaling that the sender is ready to terminate

8.3 Implementation of Push-Sum in MPI

The structure of the push-sum algorithm for computing a sum or average of distributed values has been reviewed in Algorithm 4 on page 44. Because the principle of gossiping does not target high performance systems, implementing distributed gossip-based algorithms for a parallel machine is a complex topic and it is not straightforward how to accomplish the communication and data exchange between the processes in an MPI implementation of push-sum. Due to the randomized and asynchronous communication schedules, the processes do not know when and from whom they receive a message. This causes, among others, difficulties in deciding when and how to terminate the processes so that no unprocessed message remains in the system. To solve this issue, each process p proceeds in two phases in our MPI implementations of the gossip-based algorithms.

In the first *computation* phase, process p executes the push-sum algorithm, which means sending and receiving messages and updating its local data. When convergence criteria are met, process p enters the second *termination* phase, where it is ready to terminate and does not modify its local data anymore. When all processes which may exchange data with process p also have entered the second phase, process p terminates. In this section we present several variants of MPI implementations of the computational phase of the push-sum algorithm. The solutions for termination of the processes are discussed in Section 8.4.

Although the push-sum algorithm does not require full synchronization of all processes in order to achieve the target accuracy, if several processes send out their information and terminate before the other processes even start sending, nobody in the network can compute a correct result. Therefore, the implementations in MPI require a certain coordination of the processes. Also in terms of runtime, the desired

property of the push-sum implementations in MPI is that all participating processes make progress in the computation with similar speed.

The ultimate solution is to implement push-sum with fully synchronized rounds by using, e. g., `MPI_Barrier`. This means that every node sends a message, processes receives messages and waits for all other nodes before proceeding further. The advantage of this approach is that all processes progress in the computation with the same speed and thus, it is also easy to terminate all processes at the same time after a given number of rounds executed. However, such implementation introduces large additional communication overhead, because every barrier is of similar complexity as one call of `MPI_Allreduce` (see Hoefler et al. (2004) for barriers and Thakur et al. (2005) for parallel reductions) and it does not exploit one of the strengths and attractive properties of the push-sum algorithm, namely that it does not require the rounds to be synchronized across all processes. As a consequence, we do not consider this implementation in this thesis. We introduce several implementation variants which avoid the usage of global barriers.

Version 1 - "PS-WIN"

The PS-WIN implementation (shown in Algorithm 20 on page 150) of push-sum in MPI uses one-sided communication between the processes. Before computing the sum, all participating processes call `MPI_Win_create` to establish a "window", a part of memory which is accessible to other participating processes (Line 1 in Algorithm 20). The value in this window is initialized with the `local_data` of the process. Sending a message during the computation is accomplished in the following way: First, process p chooses randomly a target process t (Line 3 in Algorithm 20) and locks exclusively the window of process t by `MPI_Win_lock` (Line 4 in Algorithm 20). Then, process p adds half of its `local_data` to the values stored in the window of the target process t by `MPI_Accumulate` (Lines 5 and 6 in Algorithm 20) and unlocks the window by `MPI_Win_unlock` (Line 7 in Algorithm 20). This sequence corresponds to one round of the push-sum algorithm in process p. For receiving a message there is no action required from the processes, because the information is directly stored to their memory window.

The main advantage of this implementation is one-sided communication, because after destroying the window of a process by `MPI_Win_free` (Line 9 in Algorithm 20), no other process can access the window. Consequently, the remaining processes cannot send information to nodes which have already finished and no pending messages are left after all processes have terminated. On the other hand, it does not correspond to the original design of the push-sum algorithm, because of the shared-memory "window" to which the access is synchronized by using locks.

Algorithm 20 PS-WIN

Input: `local_data` $\in \mathbb{R}^{k+1}$, convergence parameters ϵ and `MAX_ROUNDS` number of processes N
Output: `local_sum` $\in \mathbb{R}^k$

```
1: initialize memory window win by MPI_Win_create()
2: while (not converged)
3:    choose target t
4:    lock win of t by MPI_Win_lock()
5:    local_data ← local_data/2
6:    add local_data to t's window by MPI_Accumulate()
7:    unlock win of t by MPI_Win_unlock()
8:    if ((ERROR ≤ ε) || (rounds > MAX_ROUNDS))
9:       destroy win by MPI_Win_free()
10:      converged
11:   end if
12: end while
13: for i = 1 : k
14:   local_sum[i-1] ← N· local_data[i]/local_data[0]
15: end for
16: ... termination ...
```

Version 2 - "PS-ACK"

In the second implementation of the push-sum algorithm in MPI described in Algorithm 21 on page 151, each process has to actively receive a sent message. This communication by the means of sending and receiving messages we call *two-sided* communication. Thus, one round in process p consists of two steps. First, process p sends a message with `TAG_DATA` to a randomly chosen process t using the blocking routine `MPI_Send()` (Lines 2 and 3 in Algorithm 21). The message contains data for updating the local status of process t. Then, process p keeps receiving messages. If a message with `TAG_DATA` is received, process p updates its `local_data` and sends an acknowledgment `TAG_ACK` to the sender of the message (Lines 8 and 9 in Algorithm 21). Once process p receives an acknowledgment for a message it sent earlier, it continues to the next round.

In contrast to PS-WIN, this implementation of the push-sum algorithm does not need any initialization phase of shared-memory windows. However, due to acknowledgment messages, which have to be delivered to the target processes, this approach works only on systems with reliable communication. Moreover, the acknowledgment messages introduce additional overhead in terms of communication cost, since the PS-ACK implementation sends twice as many messages than necessary for achieving the target accuracy.

Algorithm 21 PS-ACK

Input: `local_data` $\in \mathbb{R}^{k+1}$, convergence parameters ϵ and `MAX_ROUNDS`
number of processes N
Output: `local_sum` $\in \mathbb{R}^k$

```
1:  while (not converged)
2:    local_data ← local_data/2
3:    send local_data to randomly chosen process t
4:    while (not next_round)
5:      wait for receiving a message msg
6:      case msg.tag
7:      TAG_DATA:
8:        local_data ← local_data + msg.data
9:        send TAG_ACK to msg.source
10:       if (ERROR ≤ ε) converged
11:     TAG_ACK:
12:       next_round
13:       if (rounds > MAX_ROUNDS) converged
14:     TAG_TERM:
15:       term_count++
16:       if (term_count == neighbors) converged
17:   end while
18: end while
19: for i = 1 : k
20:   local_sum[i-1] ← N· local_data[i]/local_data[0]
21: end for
22: ... termination ...
```

Version 3 - "PS-PUP"

The third version of implementation of push-sum in MPI called PS-PUP (shown in Algorithm 22 on page 152) improves PS-ACK by utilizing acknowledgments for transmitting local data which can be used in the receiving process for making progress in the computation. Whenever a process p sends an acknowledgment message, it includes its `local_data` in the same way as when sending a data message (Lines 8 and 9 in Algorithm 22). Consequently, in PS-PUP processes always mutually exchange their local data (so called "push-pull" approach). Although this implementation needs less rounds to converge then the version PS-ACK with empty acknowledgments, PS-PUP is rather an implementation of a push-pull algorithm with mutual information exchange, instead of push-sum.

Algorithm 22 PS-PUP

Input: `local_data` $\in \mathbb{R}^{k+1}$, convergence parameters ϵ and `MAX_ROUNDS` number of processes N
Output: `local_sum`$\in \mathbb{R}^k$

```
1:  while (not converged)
2:    local_data ← local_data/2
3:    send local_data to randomly chosen process t
4:    while (not next_round)
5:      wait for receiving a message msg
6:      case msg.tag
7:      TAG_DATA:
8:        local_data ← local_data + msg.data
9:        send TAG_ACK with local_data to msg.source
10:       if (ERROR ≤ ε) converged
11:     TAG_ACK:
12:       local_data ← local_data + msg.data
13:       next_round
14:       if (rounds > MAX_ROUNDS) converged
15:     TAG_TERM:
16:       term_count++
17:       if (term_count == neighbors) converged
18:   end while
19: end while
20: for i = 1 : k
21:   local_sum[i-1] ← N· local_data[i]/local_data[0]
22: end for
23: ... termination ...
```

Version 4 - "PS-ASY"

The motivation for the last version PS-ASY shown in Algorithm 23 on page 153 is to overcome the drawbacks of the previous implementations, namely initializing the memory window and sending acknowledgment messages, but to preserve the asynchronicity of the push-sum algorithm. The basic idea pursued in this implementation is to keep always N messages in the computation, where N is the number of participating processes. In theory, this condition would be preserved in synchronized rounds, because every process sends at the beginning of each round exactly one message, regardless of how many messages it has received. However, since we want to design an asynchronous implementation, we have to preserve the number of messages in the system by other means. In PS-ASY, every process begins the computation by sending one message to a randomly chosen neighbor (Line 1 in Algorithm 23) and waits for incoming messages from other processes. Note that N

Algorithm 23 PS-ASY

Input: local_data $\in \mathbb{R}^{k+1}$, convergence parameters ϵ and `MAX_SEND_MSG`
number of processes N
Output: local_sum $\in \mathbb{R}^k$

```
1: send local_data to randomly chosen process p
2: while (not converged)
3:    recv_msg ← 0
4:    wait for receiving a message msg
5:       do
6:          recv_msg++
7:          local_data ← local_data + msg.data
8:       while (a message msg to receive)
9:       local_data ← local_data/(recv_msg + 1)
10:      for (i = 1 : recv_msg)
11:         choose target t
12:         send local_data to t
13:      end for
14:   if ((ERROR ≤ ε) || (sent_messsages > MAX_SENT_MSG)) converged
15: end while
16: for i = 1 : k
17:    local_sum[i-1] ← N· local_data[i]/local_data[0]
18: end for
19: ... termination ...
```

messages in total are sent at the beginning of the computation. As soon as a process receives a message, it updates its `local_data` (Line 7 in Algorithm 23) and checks for another received message. After processing all currently received messages, the process chooses a random neighbor for every message received in this round (Lines 10-11 in Algorithm 23), sends a message to the selected neighbors (Line 12 in Algorithm 23) and waits for next messages. Since the computation was started with N messages and for each received message, one message is sent, the number of messages in the computation is always equal to N.

8.4 Termination of Gossip-Based Algorithms in MPI

An important part of push-sum implementations is termination of all processes. The following objectives should be met. First, we require that the approximation `local_sum` in every node reaches the target accuracy ϵ. Second, no pending messages should be left after all processes terminated. Obviously, the second aspect is not relevant for PS-WIN due to one-sided communication. In the three implementations

using two-sided communication the nodes do not know when the last message will be sent to them due to the randomized and asynchronous communication. Consequently, if processes terminate independently of the others, it may happen that they finish before receiving and processing all messages, which is undesirable in MPI codes. Furthermore, the situation when a process terminates and stops contributing to the computation without sending its local status to another process can be compared to a process failure. Such a situation causes the remaining active nodes to converge to an incorrect value. Thus, it is needed that all processes terminate at the same time.

As mentioned, in all presented MPI implementations, processes work in two phases, computation and termination. If process p has finished the first phase and it is ready to terminate, it sends a message with the tag `TAG_TERM` to all of its neighbors and stops refining its local estimate. Process p stops refining its local estimate and enters the termination phase when at least one of the following conditions holds:

1. The local estimate is accurate enough:
   (ERROR $\leq \epsilon$)

2. The maximum number of messages/rounds has been sent/executed:
   (`send_msg` $>$ MAX_SENT_MSG) or (`rounds` $>$ MAX_ROUNDS)

3. One of its neighbors entered the termination phase:
   (`msg.tag` $==$ `TAG_TERM`)

After entering the termination phase, process p keeps receiving messages. Once it receives a message with `TAG_TERM` from all of its neighbors, it terminates. Since a message with `TAG_TERM` is always the last message sent by a process and only nearest neighbor communication happens, a node can terminate once it has received the termination message from every neighbor and no pending messages are left in the computation.

In the MPI experiments we use the same convergence criterion as in our MATLAB simulations (Section 4.3.2), namely evaluating the relative error of the local estimate in every process compared to the correct result combined with the maximum number of rounds. We provide every process with the `true_result` at the beginning of the computation. Once the local relative error compared to the correct result is less or equal to ϵ, the process stops refining the output and enters the termination phase. Every process also counts the number of rounds or equivalently the number of sent messages containing data. Once these numbers exceed the maximal value given as an input parameter (`MAX_ROUNDS` or `MAX_MSG_SENT`) in a process, the process enters the termination phase.

Obviously, terminating a process by comparing the estimate to a true result is a theoretical solution and it serves for the experiments and investigation of the algorithms. In practice, it is not to be expected that the true result is available in any of the processes. A possibility how to solve the situation, when we cannot provide the processes with the correct result, is to use the idea of the convergence proof of push-sum presented in Kempe et al. (2003). Before starting the push-sum algorithm, every process p appends to its local data an additional *contribution vector* of length equal to the number of processes N. This vector has 1 on the pth position and zeros otherwise. Clearly, the sum of contribution vectors across all processes is equal to the vector of all ones. Thus, the correct result of the sum of all contribution vectors is a priori known in all processes. Since those vectors are sent and updated together with the local data, each process can compute the relative error of the current estimate of the local data compared to the desired sum by computing the relative error of the contribution vectors compared to the vector of all ones. Although this solution does not require any global knowledge, it introduces an overhead in terms of the amount of data exchanged. Because the overhead is proportional to the number of processes N, it is more suitable for situations where the size of the data k is (much) larger than the number of processes.

Apart from checking the relative error ϵ, a process also stops the computation of the estimate as soon as it receives the first message with `TAG_TERM` signaling that its neighbor has entered the termination phase. When several processes carry on refining the local estimate whereas other processes have entered the termination phase and do not contribute to the computation anymore, the network behaves as if it is disconnected and the active processes converge to a wrong value.

Clearly, it is possible that a process receives a message with `TAG_TERM` and stops the computation before its local estimate has reached the target accuracy ϵ. To guarantee that all processes have ϵ-accurate estimate at the end, we can utilize the `TAG_TERM` messages for spreading out the accurate results. A node, which has entered the termination phase because its `local_sum` is ϵ-accurate, includes the final estimate in the `TAG_TERM` messages sent to its neighbors. Once a neighbor receives the first `TAG_TERM` message, it uses the value included in the message as the local estimate of the true aggregate. We assume that the maximal number of rounds is set high enough so that at least one process can compute an accurate local estimate. As a result, at the end of the execution of the algorithm all nodes contain a local estimate which is ϵ-accurate. Note that this solution introduces overhead in terms of amount of data exchanged, since every termination messages contains the whole local estimate, which is of size k.

8.5 Runtime Comparisons

In this section we provide runtime comparisons of the individual implementations of push-sum in MPI (Section 8.3) to each other and also to the standard routine `MPI_Allreduce`. If not stated otherwise, the accuracy of the push-sum algorithm was set to $\epsilon = 10^{-15}$. We focus on investigating the scaling behavior of the implementations with increasing number of processes N and message size k. We also explore how the scaling behavior changes with different virtual connectivity of the processes, namely fully connected and hypercube. Last but not least, we compute a matrix-vector product by distributed algorithm dmmm (see Algorithm 15 on page 105) and we compare it to PBLAS routines `pdgemm` and `pdgemv`. For all processes we measure the time needed for both phases, the computation as well as the termination. The presented figures show the maximal time over all N processes.

8.5.1 Distributed All-to-All Reduction

First of all, we investigate the runtime of all four push-sum implementations in MPI on the Jinx cluster for summing numbers distributed across processes. Figure 8.1 on page 157 shows the runtime of gossip-based approaches averaged over 20 runs. We see that PS-WIN using one-sided communication is the slowest of all the methods. As expected, PS-ACK sending empty acknowledgment messages is slower than PS-PUP including local data into the acknowledgments. Note that all implementations show similar scaling behavior with the number of processes as the MPI routine. Figure 8.2 on page 157 shows the slow-down factors of the gossip-based algorithms compared to `MPI_Allreduce`. The slowest variant PS-WIN is on average about 20 times slower, the fastest implementations PS-PUP and PS-ASY are about 5-10 times slower than `MPI_Allreduce`.

Scaling behavior with number of processes

Let's observe the influence of the virtual connectivity of processes on the scalability behavior of PS-WIN and PS-ASY with increasing number of processes N. We investigate PS-WIN only for the fully connected case, because `MPI_Win_create()` establishes a window accessible to all remaining processes. For PS-ASY we study the situation where every process can communicate with every other process (PS-ASY(FC)) as well as the situation where the processes are organized into a virtual hypercube (PS-ASY(HC)). We omit investigating the versions PS-ACK and PS-PUP, because due to sending acknowledgments, their design does not correspond to the original push-sum algorithm and they did not outperform PS-ASY in terms of runtime.

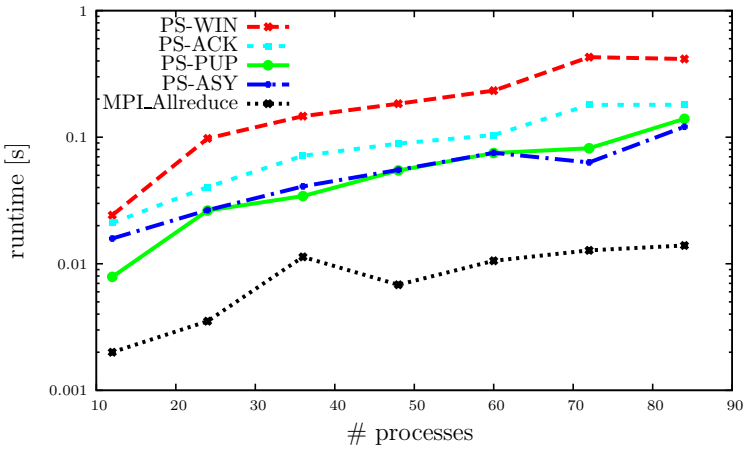

Figure 8.1: Runtime of all four implementations of push-sum in MPI and of `MPI_Allreduce` summing elements of a random vector $x \in \mathbb{R}^N$ across N processes on Jinx. Averaged over 20 runs.

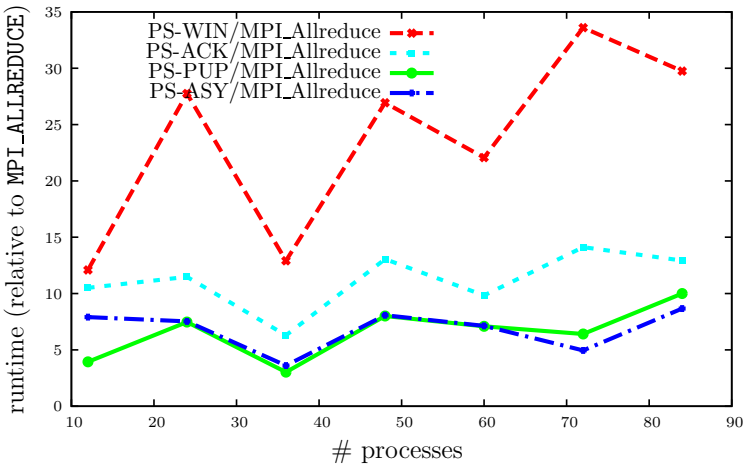

Figure 8.2: Slow-down factors of all four implementations of push-sum in MPI compared to `MPI_Allreduce` summing elements of a random vector $x \in \mathbb{R}^N$ across N processes on Jinx. Averaged over 20 runs.

From theory we know that the push-sum algorithm computes ϵ-accurate estimates over hypercube as over a fully connected network in asymptotically the same number of rounds (Section 3.2.2). The MATLAB simulations presented in Figure 5.5 on page 70 indicated that the more connected network, the faster convergence of the push-sum algorithm, since it converges slightly slower over hypercube compared to a fully connected network.

In MPI the situation is different, because different aspects have an impact on the overall runtime. In the first phase of PS-ASY, every process is waiting for receiving a message from any neighbor. Consequently, for a fully connected network, MPI has to detect incoming messages from all $(N-1)$ remaining processes, which is likely to be slower than listening to only $h = \log_2 N$ neighbors, as it is in the case of the

virtual hypercube. Moreover, during the termination phase, every process sends a message to every other process, which means all-to-all communication in the fully connected network.

The resulting runtimes obtained from Hopper are plotted in Figure 8.3. Although PS-WIN seemed to be the slowest implementation when running on a small number of processes on Jinx, for larger numbers of processes it outperforms the variant PS-ASY(FC). Moreover, the simulations confirmed the expectations that PS-ASY(FC) scales much worse than PS-ASY(HC) with communication corresponding to a hypercube topology, which is the opposite observation compared to the MATLAB simulations. Reducing the number of neighbors but keeping good connectivity of the graph makes the distributed summation based on gossiping scale very well with increasing number of processes. Further in this section we concentrate only on the PS-ASY(HC) on hypercube.

The slow-down factors compared to `MPI_Allreduce` are plotted in Figure 8.4 on page 159. In contrast to the comparisons on Jinx, on Hopper are all gossip-based methods much slower than the MPI routine.

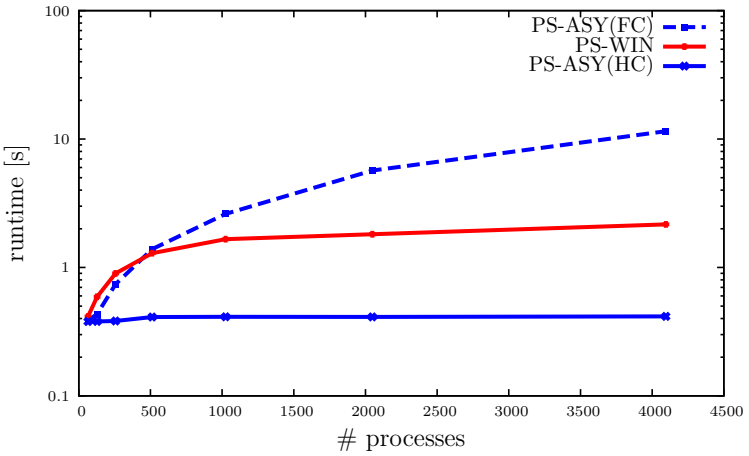

Figure 8.3: Runtime comparison of PS-WIN to PS-ASY(FC) over a fully connected network and PS-ASY(HC) on hypercube summing elements of a random vector $x \in \mathbb{R}^N$ across N processes on Hopper.

Scaling behavior with increasing message size

Another aspect we investigate for push-sum implementations in MPI is their scaling behavior with increasing message size k. This is relevant, for instance, because of the potential utilization of push-sum in more complex algorithms, where not only its scalar version, but also vector or matrix versions (Section 4.1) are of interest. As can be observed in Figure 8.5 on page 159, the runtime of both explored push-sum implementations PS-WIN and PS-ASY(HC) hardly changes with increasing

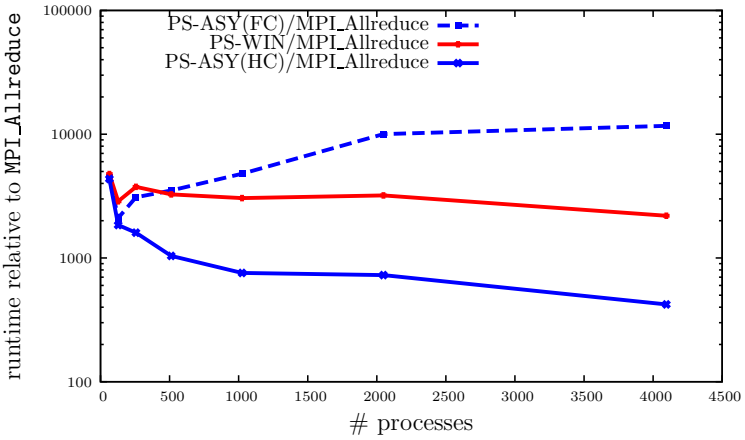

Figure 8.4: Slow-down of PS-WIN, PS-ASY(FC) over a fully connected network and PS-ASY(HC) over hypercube compared to `MPI_Allreduce` summing elements of a random vector $x \in \mathbb{R}^N$ across N processes on Hopper.

message size k. Nevertheless, it is to be expected that with further increasing size of messages, the methods will eventually slow down due to internal dividing of the MPI messages into several packets. The exact message size depends on the concrete system and on the communication protocol used.

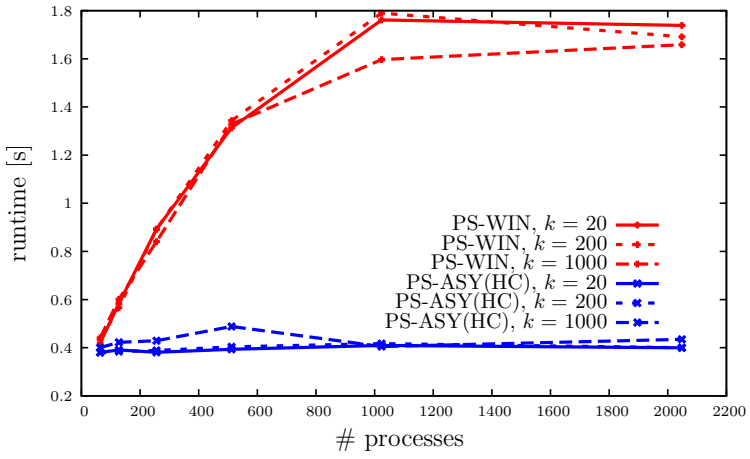

Figure 8.5: Runtime of PS-WIN and PS-ASY(HC) over hypercube summing elements of a random vector $x \in \mathbb{R}^N$ across N processes on Hopper with increasing message size k.

8.5.2 Matrix-Vector Multiplication

In the last part we investigate the gossip-based algorithm dmmm introduced in Section 6.2 compared to PBLAS routines `pdgemv` and `pdgemm`. We present results for multiplying a matrix $A \in \mathbb{R}^{n \times n}$ and a vector $V \in \mathbb{R}^n$. In dmmm we used two implementations of the push-sum algorithm, PS-WIN and PS-ASY(HC).

First, we compare the methods in terms of runtime when computing a matrix-vector multiplication with input data $A \in \mathbb{R}^{N\times N}$ and $V \in \mathbb{R}^N$ over N processes. The target accuracy ϵ of the push-sum called in the dmmm was set to 10^{-14}. The results plotted in Figure 8.6 indicate that the gossip-based method based on PS-ASY(HC) scales better with the increasing number of processes than the dmmm based on PS-WIN. The corresponding slow-down factors compared to `pdgemv` are shown in Figure 8.7 on page 161. As can be observed, dmmm using the PS-ASY(HC) implementation is about 2-3 times slower than `pdgemv`. Moreover, runtime of `pdgemm` designed for computing matrix-matrix multiplications is comparable to the parallel routine `pdgemv` for matrix-vector multiplication.

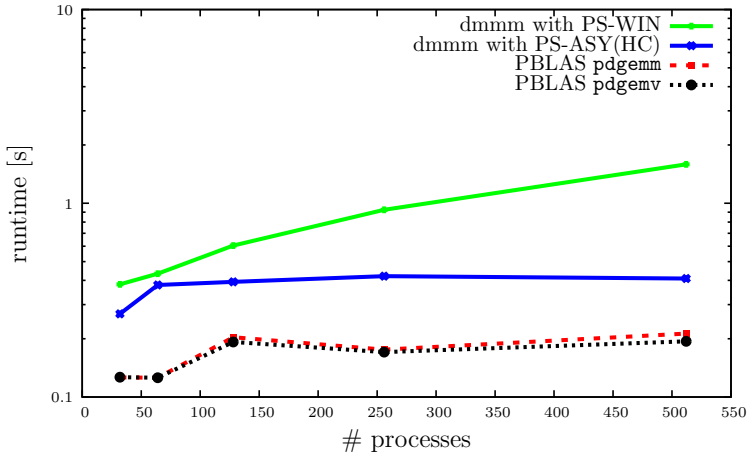

Figure 8.6: Runtime of distributed matrix-matrix multiplication dmmm based on PS-WIN and PS-ASY(HC) and of `pdgemm` and `pdgemv` multiplying a square matrix $A \in \mathbb{R}^{N\times N}$ by a vector $V \in \mathbb{R}^N$ over N processes on Hopper.

The comparison presented in Figure 8.8 on page 161 depicts the scalability of both gossip-based matrix-vector multiplications with increasing problem size n when the number of processes N stays fixed. The runtime is plotted relatively to the runtime of the routine `pdgemv`. We ran all three routines for $N = 64$ processes for multiplying $A \in \mathbb{R}^{n\times n}$ and $V \in \mathbb{R}^n$ with $n = 64, 128, 256, 512, 960$ being multiples of 64. Similarly to the previous experiments with push-sum, also for this matrix-vector multiplication the gossip-based methods scale well with increasing messages size.

8.6 Summary of the Chapter

Although gossip-based algorithms are originally intended for distributed systems, they might be useful also in future parallel computing. The goal of this chapter was to answer first questions related to implementation of gossip-based algorithms in MPI and to the behavior of the MPI implementations on parallel machines in terms

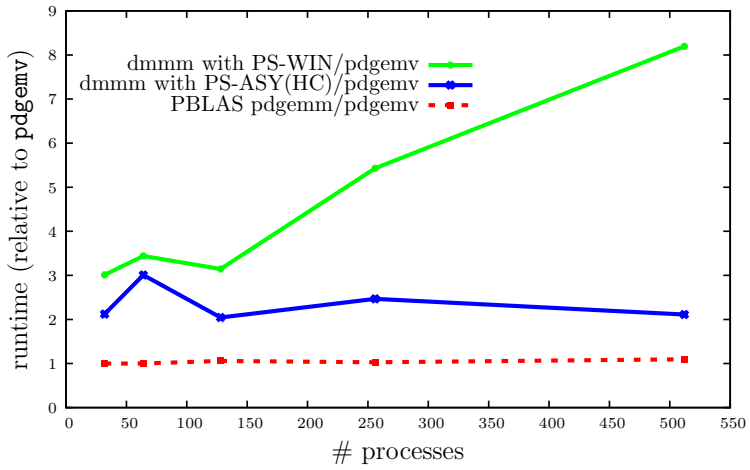

Figure 8.7: Slow-down factors of distributed matrix-matrix multiplication dmmm based on PS-WIN and PS-ASY(HC), and of `pdgemm` compared to parallel matrix-vector multiplication `pdgemv` multiplying a square matrix $A \in \mathbb{R}^{N \times N}$ by a vector $V \in \mathbb{R}^N$ over N processes on Hopper.

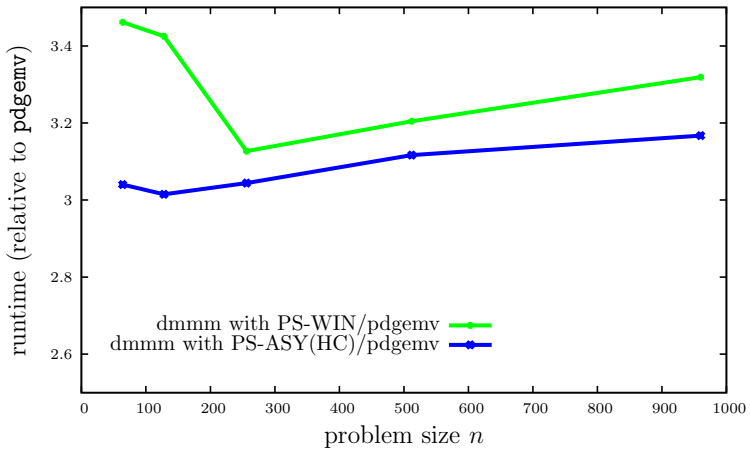

Figure 8.8: Slow-down factors of distributed matrix-matrix multiplication dmmm based on PS-WIN and PS-ASY(HC) relatively to `pdgemv` with increasing problem size n across $N = 64$ processes on Hopper.

of runtime and scalability.

Since the target systems for the push-sum algorithm are different from parallel computers, it is not straightforward how to implement it in MPI in order to preserve its asynchronicity and randomized communication schedule without leaving pending messages after the termination. We discussed several implementations of the push-sum algorithm in MPI, with different pros and cons.

An advantage of the implementation using one-sided communication is that processes can finish their computation without any danger of leaving unprocessed messages. However, the processes have to synchronize their access to shared-memory windows, which does not completely correspond to the idea of the push-sum algo-

rithm, where the processes send messages independently of each other.

Implementations using message passing communication do not need shared-memory windows, but they have to terminate all processes so that no pending messages between the processes are left. We introduced an implementation solution of terminating the processes, where each processes waits for all its neighbors to reach a termination phase in order to finish.

We considered two virtual topologies of the MPI processes and we studied their influence on scalability. In theory, we know that the more connected the graph, the better the convergence speed of push-sum. Therefore, if the underlying topology is fully connected so that every process can communicate directly to every other, push-sum converges faster than on any other topology. However, on parallel machines the situation is different. In a fully connected topology every process constantly listens to all other processes in order to receive a message. Therefore, the resulting implementation scales worse with increasing number of processes compared to the experiments run over a virtual hypercube, where the neighborhood of each node is substantially smaller.

First experiments showed that gossip-based algorithms may have the potential to run on parallel machines with preserving their attractive properties, such as no global synchronization requirements. We presented several experimental results showing the behavior of different push-sum implementations and of distributed matrix-vector multiplication based on push-sum. The concrete slow-down factors of the distributed algorithms compared to `MPI_Allreduce` and `pdgemv` depend on the machine, on the MPI implementations of push-sum used and on the virtual topology of the processes. Although first results on Jinx indicated that PS-ASY is the fastest method being about 5-10 times slower than `MPI_Allreduce`, the comparison to `MPI_Allreduce` on Hopper showed slow-down factors of the fastest implementation of about 1000. Experiments on Hopper also showed that PS-ASY on fully connected topology scales worse than PS-WIN and that PS-ASY ran on virtual hypercube has much better scaling behavior both of the methods, which is also in contrast to observations made in MATLAB . The runtime comparisons of matrix-vector multiplication indicated that the gossip-based algorithms on parallel machines may be only 2-3 times slower than `pdgemv`.

Several fundamental questions still remain open, especially how to efficiently terminate the participating processes without leaving unprocessed messages and how to locally detect convergence of the estimates without knowing the true aggregate. Once those issues are solved theoretically as well as for practical usage, the next steps are to implement an effective push-sum for parallel computers and subsequently distributed matrix algorithms based on the new effective implementation.

Chapter 9

Summary and Conclusions

Gossip-based algorithms have been so far utilized for rather simple computations, but their attractive properties, such as flexibility and good scalability, motivate a deeper investigation of their behavior in practice as underlying principle for future flexible, asynchronous and resilient numerical matrix algorithms. In this thesis we provided an extensive discussion and analysis of prototypical distributed matrix approaches constructed on top of gossip-based distributed data aggregation algorithms (DDAAs), which are algorithms allowing for a decentralized computation an aggregate, such as sum or average, across a network. A well known gossip-based algorithm of this kind is the *push-sum algorithm*.

The major contribution of this thesis is the development and analysis of prototypical distributed matrix algorithms based on gossiping. We presented the *dmGS* algorithm, a gossip-based modified Gram-Schmidt orthogonalization for computing a QR factorization of a general rectangular matrix distributed over a loosely coupled decentralized system. The second algorithm we investigated in detail is a gossip-based eigensolver dOI based on orthogonal iteration, which comprises a distributed matrix-matrix multiplication and the dmGS algorithm. Based on the analysis and presented simulation results we can conclude that the presented distributed gossip-based matrix algorithms preserve properties of existing sequential numerical linear algorithms, such as numerical accuracy and convergence speed, if the DDAAs compute the results in double precision.

The key aspect of all distributed matrix algorithms developed in this thesis is that all communication between nodes is randomized and flexible. The absence of any operation with a deterministic communication schedule in all building blocks leads to many attractive properties. An existing distributed orthogonal iteration designed by Kempe and McSherry (2008) is partially based on a DDAA, but it also contains a distributed computation with deterministic schedules. Consequently, it

cannot take advantage of any of the attractive properties, such as flexibility, fault tolerance or communication cost reduction through accuracy-performance trade-offs.

Thanks to the ability of gossip-based algorithms for accuracy-performance trade-offs, the overall invested cost can be significantly reduced, which we illustrated on dOI. The lower the target accuracy ϵ of the DDAA, the lower also the final accuracy of the results computed by distributed matrix algorithms, but also the lower the invested cost. Because dOI works in iterations and calls a DDAA working in rounds, the resulting inner-outer structure allows for a perfect utilization of accuracy-performance trade-offs. By gradually adjusting the accuracy of the DDAA to the progress in dOI, the overall communication cost can be reduced by at least a factor of two.

Another advantage of the push-sum algorithm is its good scalability with the number of nodes on well-connected networks, which is also inherited by the distributed algorithms built on top of it. With increasing problem size is the situation slightly more difficult and depends on the concrete algorithm. We illustrated how distributed gossip-based algorithms can trade off the message size for the number of the messages, and thus they can adapt to the needs of the concrete underlying system. If the priority is to send as few messages as possible, a vector version or a matrix version of a DDAA can be employed to increase the size of the message and decrease the overall count. The concrete examples are three versions of distributed matrix-matrix multiplication or a version of dmGS called *vdmGS*, which exchanges fewer (but larger) messages than dmGS. Thanks to the modification, the vdmGS algorithm is comparable to parallel mGS in terms of communication cost. Nevertheless, the communication-optimal CAQR algorithm outperforms all methods based on modified Gram-Schmidt in terms of number of messages as well as the amount data sent. For the special case of a square matrix $V \in \mathbb{R}^{n \times n}$ distributed over N nodes, vdmGS sends by a factor $\sqrt{N}$ more real numbers and by a factor of $n/(\sqrt{N}log^2N)$ more messages than CAQR.

In general, distributed matrix algorithm based on push-sum tolerate reported link failures thanks to their flexible communication schedules. However, since push-sum requires preservation of the global mass in the system during the whole computation, it does not tolerate any silent failures, such as message losses. In contrast, distributed algorithms using a self-healing gossip-based DDAA, such as push-cancel-flow, deliver an accurate result even in the presence of numerous messages losses. Delivering an accurate result in the presence of node failures is in general more challenging than tolerating link failures due to the loss of initial data. If all of the input data has to be included in the final result, distributed algorithms have to back-up the initial data due to the whole process in order to deliver a correct result, as we

illustrated by the rdmGS algorithm.

Apart from comparing the distributed gossip-based algorithms to their parallel counterparts, we also focus on distributed alternatives, such as consensus-based approaches. A main difference of gossip-based DDAAs compared to consensus-based algorithms is that they do not require global synchronization of every round across the network, as we illustrated by a dmGS algorithm with a very weak synchronization strategy (at the algorithmic level) achieving full accuracy. However, a distributed QR factorization based on dynamic consensus has several advantages over gossip-based DDAA, especially the ability to compute an estimate of entire result in every round.

The introduced distributed gossip-based algorithms can be used thanks to their versatility also for constructing other distributed algorithms, e. g., distributed least-squares solvers. As we illustrated in a short case study, a distributed linear least-squares solver based on dmGS can be utilized for computing a linear regression model. Such an approach is attractive for various data mining tasks over P2P or social networks with initially distributed data.

Although gossip-based algorithms are originally targeted for systems such as P2P networks or sensor networks, due to anticipated changes in the large-scale parallel high-performance computers, they may offer an interesting solution for future HPC systems. We presented several possible implementation variants of the push-sum algorithm in MPI and we compared in terms of runtime. Comparisons to the standard parallel reduction operation `MPI_Allreduce` showed a slow-down of the factor 10 on a small machine and around a factor 1000 on a large machine. In contrast, a gossip-based matrix-matrix algorithm compared to the PBLAS routines `pdgemm` and `pdgemv` was only about 2-3 times slower. An important aspect revealed by the experiments on high-performance computers is that the runtime of the push-sum algorithm in MPI is influenced by the connectivity of the processes in the opposite way than in theory. In theory, the push-sum algorithm converges faster on a fully connected topology than on a hypercube. In MPI implementations, computations where each process can communicate with every other process scale much worse with increasing number of processes than executions over a virtual hypercube.

Future Work

To the best of our knowledge, this is the first comprehensive overview and analysis of distributed matrix algorithms based on the principle of gossiping. Consequently, many aspects outlined in this thesis have to be addressed in greater detail. These include, for example, a complex investigation of synchronization strategies of gossip-based algorithms or designing an adaptive dOI with adjustments based on local

criteria. In terms of theoretical analysis, comparisons to centralized approaches or a complete convergence proof of dOI in floating-point arithmetic are missing. Moreover, the inferior numerical properties observed in distributed QR factorization based on dynamic consensus have not been explained yet.

The research presented in this thesis can be also further extended by changing the communication pattern of the underlying DDAA and studying its influence. For example, one can study the changes when each node communicates with $d > 1$ communication partners in every round. Furthermore, a concept preserving the attractive properties of gossiping together with exploiting the posibility of broadcasting will be needed.

Last but not least, further investigation of gossip-based algorithms on HPC machines is needed, such as comparison to parallel approaches in terms of flop count or peak performance, comparison to state-of-the-art fault tolerant approaches or investigating implementations based on lower level alternatives to MPI, e. g., plain socket communication.

Appendix A

Background on MPI

The following descriptions and definitions of routines and terms used in Chapter 8 can be found in the MPI-2.2 standard[1]. We focus on a high level overview of the functionality of the routines. The full details including descriptions of all input and output parameters are given in the MPI standard.

Terminology

Collective communication — It is defined as communication that involves a group or groups of processes.

Nonblocking — Nonblocking procedure may return before the operation completes and before the user is allowed to reuse resources (such as buffers) specified in the call.

Blocking — Return from a blocking procedure indicates that the user is allowed to reuse resources specified in the call.

One-sided communications — Defines communication routines that can be completed by a single process. These include shared-memory operations (put/get) and remote accumulate operations.

MPI Routines

`MPI_ALLREDUCE` — Global reduction operations such as sum, maximum, minimum, etc., where the result is returned to all members of a group. MPI requires that all processes participating in these operations receive identical results.

[1] http://www.mpi-forum.org/docs/

`MPI_BARRIER` — Barrier synchronization across all members of a group blocks the caller until all group members have called it. The call returns at any process only after all group members have entered the call.

`MPI_WIN_CREATE` — Creates an MPI Window object for one-sided communication. The initialization operation allows each process in a group to specify a "window" in its memory that is made accessible to accesses by remote processes.

`MPI_SEND` — Performs a blocking send.

`MPI_ISEND` — Performs a nonblocking send.

`MPI_RECV` — Performs a blocking receive.

`MPI_IRECV` — Performs a nonblocking receive.

`MPI_WIN_LOCK(rank, win,...)` — Starts an remote-memory-access (RMA) epoch. Only the window at the process with rank `rank` can be accessed by RMA operations on window `win` during that epoch. Locks are used to protect accesses to the locked target window effected by RMA calls issued between the lock and unlock call, and to protect local load/store accesses to a locked local window executed between the lock and unlock call.

`MPI_WIN_UNLOCK(rank, win)` — Completes an RMA (remote-memory-access) access epoch started by a call to `MPI_WIN_LOCK(win,...)`. RMA operations issued during this period will have completed both at the origin and at the target when the call returns.

`MPI_WIN_FREE(win,...)` — Frees the window object `win` and returns a null handle (equal to MPI_WIN_NULL).

`MPI_ACCUMULATE(target, win, op, ...)` — Accumulates the contents of the origin buffer to the buffer in the target window specified by `target` and `win`, using the operation `op`. For example, if `op` is `MPI_SUM`, each element of the origin buffer is added to the corresponding element in the target, replacing the former value in the target.

Bibliography

Abdelhak, S., Abdelgawad, A., Ghosh, S., and Bayoumi, M. (2010). A complete scheme for distributed lu decomposition on wireless sensor networks. In *Circuits and Systems (MWSCAS), 2010 53rd IEEE International Midwest Symposium on*, pages 347–350.

Abdelhak, S., Chaudhuri, R. S., Gurram, C. S., Ghosh, S., and Bayoumi, M. (2011). Energy-aware distributed qr decomposition on wireless sensor nodes. *Comput. J.*, 54(3):373–391.

Agullo, E., Demmel, J., Dongarra, J., Hadri, B., Kurzak, J., Langou, J., Ltaief, H., Luszczek, P., and Tomov, S. (2009). Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects. *Journal of Physics: Conference Series*, 180(1):012037.

Anfinson, C. and Luk, F. (1988). A Linear Algebraic Model of Algorithmic-Based Fault Tolerance. In *Proceedings of the International Conference on Systolic Arrays*, pages 483–493.

Awerbuch, B. (1985). Complexity of network synchronization. *J. ACM*, 32(4):804–823.

Aysal, T., Yildiz, M., Sarwate, A., and Scaglione, A. (2009). Broadcast gossip algorithms for consensus. *IEEE Trans. Signal Processing*, 57(7):2748–2761.

Bertsekas, D. P. and Tsitsiklis, J. N. (1989). *Parallel and distributed computation: numerical methods*. Prentice-Hall, Inc.

Björck, Å. and Golub, G. H. (1967). Iterative refinement of linear least squares solution by Householder transformation. *BIT*, 7:322–337.

Blackford, L. S., Choi, J., Cleary, A., D'Azevedo, E., Demmel, J. W., Dhillon, I., Dongarra, J. J., Hammarling, S., Henry, G., Petitet, A., Stanley, K., Walker, D., and Whaley, R. C. (1997). ScaLapack *Users' Guide*. SIAM Press.

Boyd, S., Ghosh, A., Prabhakar, B., and Shah, D. (2006). Randomized gossip algorithms. *IEEE Trans. Information Theory*, 52(6):2508–2530.

Buntinas, D., Coti, C., Herault, T., Lemarinier, P., Pilard, L., Rezmerita, A., Rodriguez, E., and Cappello, F. (2008). Blocking vs. non-blocking coordinated checkpointing for large-scale fault tolerant {MPI} protocols. *Future Generation Computer Systems*, 24(1):73–84.

Burg, A., Borgmann, M., Wenk, M., Zellwegera, M., Fichtner, W., and Bölcskei, H. (2005). VLSI implementation of MIMO detection using the sphere decoding algorithm. *IEEE J. Solid-State Circuits*, 40(7):1566–1577.

Buttari, A., Langou, J., Kurzak, J., and Dongarra, J. (2008). Parallel tiled QR factorization for multicore architectures. In *Proceedings of the 7th International Conference on Parallel Processing and Applied Mathematics*, pages 639–648. Springer-Verlag.

Cappello, F., Geist, A., Gropp, B., Kale, L., Kramer, B., and Snir, M. (2009). Toward exascale resilience. *Int. J. High Perform. Comput. Appl.*, 23(4):374–388.

Chung, F. R. K. (1997). *Spectral Graph Theory*. American Mathematical Society.

Daly, J. T. (2006). A higher order estimate of the optimum checkpoint interval for restart dumps. *Future Gener. Comput. Syst.*, 22(3):303–312.

Demers, A., Greene, D., Houser, C., Irish, W., Larson, J., Shenker, S., Sturgis, H., Swinehart, D., and Terry, D. (1988). Epidemic algorithms for replicated database maintenance. *SIGOPS Oper. Syst. Rev.*, 22(1):8–32.

Demmel, J., Grigori, L., Hoemmen, M., and Langou, J. (2012). Communication-optimal parallel and sequential qr and lu factorizations. *SIAM Journal on Scientific Computing*, 34(1):A206–A239.

Denantes, P., Benezit, F., Thiran, P., and Vetterli, M. (2008). Which distributed averaging algorithm should i choose for my sensor network? In *INFOCOM 2008. The 27th Conference on Computer Communications. IEEE*, pages 986–994.

Dimakis, A., Kar, S., Moura, J., Rabbat, M., and Scaglione, A. (2010). Gossip algorithms for distributed signal processing. *Proceedings of the IEEE*, 98(11):1847–1864.

Dimakis, A., Sarwate, A., and Wainwright, M. (2008). Geographic gossip: Efficient averaging for sensor networks. *IEEE Trans. Signal Processing*, 56(3):1205–1216.

Dongarra, J. (2012). Algorithmic and software challenges when moving towards exascale. In *Proceedings of the ATIP/A*CRC Workshop on Accelerator Technologies for High-Performance Computing: Does Asia Lead the Way?*, ATIP '12, pages 17:1–17:35. A*STAR Computational Resource Centre.

Dumard, C. and Riegler, E. (2009). Distributed sphere decoding. In *International Conference on Telecommunications ICT '09*, pages 172–177.

Egwutuoha, I. P., Levy, D., Selic, B., and Chen, S. (2013). A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems. *The Journal of Supercomputing*, pages 1–25.

Engelmann, C., Ong, H. H., and Scott, S. L. (2009). The Case for Modular Redundancy in Large-Scale High Performance Computing Systems. In *Proceedings of the 27th IASTED International Conference on Parallel and Distributed Computing and Networks (PDCN) 2009*, pages 189–194. ACTA Press.

Eyal, I., Keidar, I., and Rom, R. (2010). Distributed data classification in sensor networks. In *PODC '10: Proceeding of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, pages 151–160. ACM.

Eyal, I., Keidar, I., and Rom, R. (2012). Limosense — live monitoring in dynamic sensor networks. In *Proceedings of the 7th international conference on Algorithms for Sensor Systems, Wireless Ad Hoc Networks and Autonomous Mobile Entities*, ALGOSENSORS'11, pages 72–85. Springer-Verlag.

Fiala, D., Mueller, F., Engelmann, C., Riesen, R., Ferreira, K., and Brightwell, R. (2012). Detection and correction of silent data corruption for large-scale high-performance computing. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 78:1–78:12. IEEE Computer Society Press.

Fiedler, M. (1973). Algebraic connectivity of graphs. *Czechoslovak Mathematical Journal*, 23(98):298–305.

Fiedler, M. (1975). A property of eigenvectors of nonnegative symmetric matrices and its application to graph theory. *Czechoslovak Mathematical Journal*, 25(4):619–633.

Franceschelli, M., Giua, A., and Seatzu, C. (2007). Load balancing on networks with gossip-based distributed algorithms. In *Decision and Control, 2007 46th IEEE Conference on*, pages 500–505.

Gansterer, W. N., Niederbrucker, G., Straková, H., and Schulze Grotthoff, S. (2011). Robust distributed orthogonalization based on randomized aggregation. In *Proceedings of the Second Workshop on Scalable algorithms for Large-Scale Systems*, ScalA '11, pages 7–10. ACM.

Gansterer, W. N., Niederbrucker, G., Straková, H., and Schulze Grotthoff, S. (2013). Scalable and Fault Tolerant Orthogonalization Based on Randomized Distributed Data Aggregation. *Journal of Computational Science*. http://dx.doi.org/10.1016/j.jocs.2013.01.006 (in press).

Golub, G. H. and Van Loan, C. F. (1996). *Matrix Computations*. The Johns Hopkins University Press, third edition.

Guermouche, A., Ropars, T., Brunet, E., Snir, M., and Cappello, F. (2011). Uncoordinated checkpointing without domino effect for send-deterministic mpi applications. In *IPDPS*, pages 989–1000.

Hadri, B., Ltaief, H., Agullo, E., and Dongarra, J. (2010). Tile qr factorization with parallel panel processing for multicore architectures. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–10.

Hasemann, H., Hirvonen, J., Rybicki, J., and Suomela, J. (2012). Deterministic local algorithms, unique identifiers, and fractional graph colouring. In *Proceedings of the 19th international conference on Structural Information and Communication Complexity*, SIROCCO'12, pages 48–60. Springer-Verlag.

Higham, N. J. (2002). *Accuracy and Stability of Numerical Algorithms*. Society for Industrial and Applied Mathematics, second edition.

Hoefler, T., Mehlan, T., Mietke, F., and Rehm, W. (2004). A Survey of Barrier Algorithms for Coarse Grained Supercomputers. *Chemnitzer Informatik Berichte*, 04(03).

Hoory, S., Linial, N., and Wigderson, A. (2006). Expander graphs and their applications. *Bull. Amer. Math. Soc.*, 43:439–561.

Huang, K.-H. and Abraham, J. (1984). Algorithm-Based Fault Tolerance for Matrix Operations. *IEEE Trans Comput*, C-33(6):518–528.

Iwanicki, K., van Steen, M., and Voulgaris, S. (2006). Gossip-based clock synchronization for large decentralized systems. In *Proceedings of the Second IEEE international conference on Self-Managed Networks, Systems, and Services*, SelfMan'06, pages 28–42.

Jelasity, M., Canright, G., and Engø-monsen, K. (2007). Asynchronous distributed power iteration with gossip-based normalization. In *Euro-Par 2007, volume 4641 of Lecture Notes in Computer Science*, pages 514–525. Springer-Verlag.

Kempe, D., Dobra, A., and Gehrke, J. (2003). Gossip-Based Computation of Aggregate Information. In *Proceedings of the 44th Annual IEEE Symposium on Foundations of Computer Science*, pages 482–491.

Kempe, D. and McSherry, F. (2008). A decentralized algorithm for spectral analysis. *J Comput Syst Sci*, 74(1):70–83.

Khan, M. I., Gansterer, W. N., and Haring, G. (2013). Static vs. mobile sink: The influence of basic parameters on energy efficiency in wireless sensor networks. *Computer Communications*, 36(9):965–978.

Korada, S. B., Montanari, A., and Oh, S. (2011). Gossip pca. In *Proceedings of the ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, SIGMETRICS '11, pages 209–220. ACM.

Kostoulas, D., Psaltoulis, D., Gupta, I., Birman, K., and Demers, A. (2005). Decentralized schemes for size estimation in large and dynamic groups. In *Proceedings of the Fourth IEEE International Symposium on Network Computing and Applications*, NCA '05, pages 41–48.

Leng, M. and Wu, Y.-C. (2011). Distributed clock synchronization for wireless sensor networks using belief propagation. *Signal Processing, IEEE Transactions on*, 59(11):5404–5414.

Levin, D. A., Peres, Y., and Wilmer, E. L. (2006). *Markov chains and mixing times*. American Mathematical Society.

Luk, F. and Park, H. (1988). Fault-tolerant matrix triangularizations on systolic arrays. *IEEE Trans Comput*, 37(11):1434–1438.

Montresor, A., Jelasity, M., and Babaoglu, O. (2008). Decentralized ranking in large-scale overlay networks. In *Self-Adaptive and Self-Organizing Systems Workshops, 2008. SASOW 2008. Second IEEE International Conference on*, pages 208–213.

Niederbrucker, G. and Gansterer, W. N. (2013). Robust Gossip-Based Aggregation: A Practical Point of View. In Sanders, P. and Zeh, N., editors, *Proceedings of the 2013 Meeting on Algorithm Engineering & Experiments*, ALENEX 2013, pages 133–147. SIAM / Omnipress.

Niederbrucker, G., Strakova, H., and Gansterer, W. (2012). Improving fault tolerance and accuracy of a distributed reduction algorithm. In *Proceedings of the third Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA)*, pages 643–651, Salt Lake City, UT, USA.

Olfati-Saber, R., Fax, J. A., and Murray, R. M. (2007). Consensus and Cooperation in Networked Multi-Agent Systems. *Proc IEEE*, 95(1):215–233.

Ormandi, R., Hegedus, I., and Jelasity, M. (2013). Gossip learning with linear models on fully distributed data. *Concurrency and Computation: Practice and Experience*, 25(4):556–571.

Ozgur, A., Leveque, O., and Tse, D. (2007). Hierarchical cooperation achieves optimal capacity scaling in ad hoc networks. *IEEE Transactions on information theory*, 53(10):3549–3572.

Page, L., Brin, S., Motwani, R., and Winograd, T. (1998). The PageRank Citation Ranking: Bringing Order to the Web. Technical Report 1999-66, Stanford Digital Library Technologies Project.

Peleg, D. and Ullman, J. D. (1987). An optimal synchronizer for the hypercube. In *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, PODC '87, pages 77–85. ACM.

Plank, J., Li, K., and Puening, M. (1998). Diskless checkpointing. *IEEE Trans. Parallel and Dist. Syst.*, 9(10):972–986.

Pothen, A., Simon, H. D., and Liou, K.-P. (1990). Partitioning sparse matrices with eigenvectors of graphs. *SIAM J. Matrix Anal. Appl.*, 11(3):430–452.

Riesen, R., Ferreira, K., Da Silva, D., Lemarinier, P., Arnold, D., and Bridges, P. G. (2012). Alleviating scalability issues of checkpointing protocols. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 18:1–18:11. IEEE Computer Society Press.

Rost, P. and Fettweis, G. (2010). On the transmission-computation-energy tradeoff in wireless and fixed networks. In *GLOBECOM Workshops (GC Wkshps), 2010 IEEE*, pages 1394–1399.

Shi, L., Song, W.-Z., Lees, J. M., and Xing, G. (2013). Imaging seismic tomography in sensor networks. *IEEE SECON 2013 TPC*. (to appear).

Simeone, O., Spagnolini, U., Bar-Ness, Y., and Strogatz, S. H. (2008). Distributed synchronization in wireless networks. *Signal Processing Magazine, IEEE*, 25(5):81–97.

Sluciak, O., Straková, H., Rupp, M., and Gansterer, W. (2012). Distributed gram-schmidt orthogonalization based on dynamic consensus. In *Signals, Systems and Computers (ASILOMAR), 2012 Conference Record of the Forty Sixth Asilomar Conference on*, pages 1207–1211.

Sluciak, O., Straková, H., Rupp, M., and Gansterer, W. (2013). Dynamic Average Consensus and Distributed Orthogonalization. *Signal Processing, IEEE Transactions on*. (under revision).

Solomonik, E. and Demmel, J. (2011). Communication-optimal parallel 2.5d matrix multiplication and lu factorization algorithms. In *Proceedings of the 17th international conference on Parallel processing - Volume Part II*, Euro-Par'11, pages 90–109.

Straková, H. and Gansterer, W. (2013). A distributed eigensolver for loosely coupled networks. In *Parallel, Distributed and Network-Based Processing (PDP), 2013 21st Euromicro International Conference on*, pages 51–57.

Straková, H. and Gansterer, W. N. (2012). Poster: Gossip-based distributed matrix computations. In *SC Companion*, page 1407. IEEE Computer Society.

Straková, H., Gansterer, W. N., and Zemen, T. (2011). Distributed QR Factorization Based on Randomized Algorithms. In Wyrzykowski, R., Dongarra, J., Karczewski, K., and Wasniewski, J., editors, *PPAM (1)*, volume 7203 of *Lecture Notes in Computer Science*, pages 235–244. Springer.

Straková, H., Niederbrucker, G., and Gansterer, W. N. (2013). Fault tolerance properties of gossip-based distributed orthogonal iteration methods. *Procedia Computer Science*, 18(0):189–198. Proceedings of the International Conference on Computational Science, ICCS 2013.

Suomela, J. (2013). Survey of local algorithms. *ACM Comput. Surv.*, 45(2):24:1–24:40.

Thakur, R., Rabenseifner, R., and Gropp, W. (2005). Optimization of Collective Communication Operations in MPICH. *IJHPCA*, 19(1):49–66.

Trefethen, L. N. and Bau, D. (1997). *Numerical Linear Algebra*. SIAM: Society for Industrial and Applied Mathematics.

van Greunen, J. and Rabaey, J. (2003). Lightweight time synchronization for sensor networks. In *Proceedings of the 2nd ACM international conference on Wireless sensor networks and applications*, WSNA '03, pages 11–19. ACM.

Varela, M., Ferreira, K., and Riesen, R. (2010). Fault-tolerance for exascale systems. In *Cluster Computing Workshops and Posters (CLUSTER WORKSHOPS), 2010 IEEE International Conference on*, pages 1–4.

Yang, Z., Cai, L., Liu, Y., and Pan, J. (2012). Environment-aware clock skew estimation and synchronization for wireless sensor networks. In *INFOCOM, 2012 Proceedings IEEE*, pages 1017–1025.

Biography

Personal

Name: Hana Straková

Born: 4th March 1984, Prague, Czech Republic

Nationality: Czech

Affiliation

Research Group Theory and Applications of Algorithms
Faculty of Computer Science, University of Vienna
Währinger Straße 29, A-1090 Vienna, Austria

Phone.: +43 1 4277 78349

Email: hana.strakova@univie.ac.at

Academic Education

9/2009 - present	Ph.D. Program in Computer Science University of Vienna anticipated graduation: September 2013
9/2002 - 6/2008	M. Sc. - Master Studies in Computer Science Faculty of Mathematics and Physics Charles University of Prague

Professional Experience

9/2009-present	Research Associate Research Group Theory and Applications of Algorithms Faculty of Computer Science, University of Vienna FWF project NFN SISE
10/2008-9/2009	Technical Consultant SPSS CR

Exchange Programs, Internships

10/2012-12/2012	Georgia Tech, Atlanta, USA College of Computing School of Computational Science and Engineering Research Internship
09/2005-06/2006	University of Oulu, Finland Department of Information Processing Science and Department of Mathematics Erasmus Program

June 28, 2013