Scalable Optimization Algorithms for Recommender Systems

Faraz Makari Manshadi

Thesis for obtaining the title of
Doctor of Engineering
of the Faculties of Natural Sciences and Technology I
of Saarland University

Saarbrücken, Germany
2014

Abstract

Recommender systems have now gained significant popularity and been widely used in many e-commerce applications. Predicting user preferences is a key step to providing high quality recommendations. In practice, however, suggestions made to users must not only consider user preferences in isolation; a good recommendation engine also needs to account for certain constraints. For instance, an online video rental that suggests multimedia items (e.g., DVDs) to its customers should consider the availability of DVDs in stock to reduce customer waiting times for accepted recommendations. Moreover, every user should receive a small but sufficient number of suggestions that the user is likely to be interested in.

This thesis aims to develop and implement scalable optimization algorithms that can be used (but are not restricted) to generate recommendations satisfying certain objectives and constraints like the ones above. State-of-the-art approaches lack efficiency and/or scalability in coping with large real-world instances, which may involve millions of users and items. First, we study large-scale matrix completion in the context of collaborative filtering in recommender systems. For such problems, we propose a set of novel shared-nothing algorithms which are designed to run on a small cluster of commodity nodes and outperform alternative approaches in terms of efficiency, scalability, and memory footprint. Next, we view our recommendation task as a generalized matching problem, and propose the first distributed solution for solving such problems at scale. Our algorithm is designed to run on a small cluster of commodity nodes (or in a MapReduce environment) and has strong approximation guarantees. Our matching algorithm relies on linear programming. To this end, we present an efficient distributed approximation algorithm for mixed packing-covering linear programs, a simple but expressive subclass of linear programs. Our approximation algorithm requires a poly-logarithmic number of passes over the input, is simple, and well-suited for parallel processing on GPUs, in shared-memory architectures, as well as on a small cluster of commodity nodes.

Kurzfassung

Empfehlungssysteme haben eine beachtliche Popularität erreicht und werden in zahlreichen E-Commerce Anwendungen eingesetzt. Entscheidend für die Generierung hochqualitativer Empfehlungen ist die Vorhersage von Nutzerpräferenzen. Jedoch sollten in der Praxis nicht nur Vorschläge auf Basis von Nutzerpräferenzen gegeben werden, sondern gute Empfehlungssysteme müssen auch bestimmte Nebenbedingungen berücksichtigen. Zum Beispiel sollten online Videoverleihfirmen, welche ihren Kunden multimediale Produkte (z.B. DVDs) vorschlagen, die Verfügbarkeit von vorrätigen DVDs beachten, um die Wartezeit der Kunden für angenommene Empfehlungen zu reduzieren. Darüber hinaus sollte jeder Kunde eine kleine, aber ausreichende Anzahl an Vorschlägen erhalten, an denen er interessiert sein könnte.

Diese Arbeit strebt an skalierbare Optimierungsalgorithmen zu entwickeln und zu implementieren, die (unter anderem) eingesetzt werden können Empfehlungen zu generieren, welche weitere Zielvorgaben und Restriktionen einhalten. Derzeit existierenden Ansätzen mangelt es an Effizienz und/oder Skalierbarkeit im Umgang mit sehr großen, durchaus realen Datensätzen von, beispielsweise Millionen von Nutzern und Produkten. Zunächst analysieren wir die Vervollständigung großskalierter Matrizen im Kontext von kollaborativen Filtern in Empfehlungssystemen. Für diese Probleme schlagen wir verschiedene neue, verteilte Algorithmen vor, welche konzipiert sind auf einer kleinen Anzahl von gängigen Rechnern zu laufen. Zudem können sie alternative Ansätze hinsichtlich der Effizienz, Skalierbarkeit und benötigten Speicherkapazität überragen. Als Nächstes haben wir die Empfehlungsproblematik als ein generalisiertes Zuordnungsproblem betrachtet und schlagen daher die erste verteilte Lösung für großskalierte Zuordnungsprobleme vor. Unser Algorithmus funktioniert auf einer kleinen Gruppe von gängigen Rechnern (oder in einem MapReduce-Programmierungsmodel) und erzielt gute Approximationsgarantien. Unser Zuordnungsalgorithmus beruht auf linearer Programmierung. Daher präsentieren wir einen effizienten, verteilten Approximationsalgorithmus für vermischte lineare Packungs- und Überdeckungsprobleme, eine einfache aber expressive Unterklasse der linearen Programmierung. Unser Algorithmus benötigt eine polylogarithmische Anzahl an Scans der Eingabedaten. Zudem ist er einfach und sehr gut geeignet für eine parallele Verarbeitung mithilfe von Grafikprozessoren, unter einer gemeinsam genutzten Speicherarchitektur sowie auf einer kleinen Gruppe von gängigen Rechnern.

Acknowledgements

Foremost, I would like to express my deepest gratitude to my adviser Rainer Gemulla for his expertise, invaluable support, and encouragements. His excellent scientific advice and exemplary guidance made this work possible. I would like to express my sincere gratitude for this. I also would like to thank Mauro Sozio for his continuous support and for the stimulating discussions. I am sincerely grateful to Gerhard Weikum for giving me the opportunity to pursue doctoral studies and ensuring a great atmosphere in the research group. Furthermore, I would like to acknowledge and thank my co-authors, Julián Mestre, Rohit Khandekar, and Christina Teflioudi, without their support this work would have not been possible.

I owe many thanks to the International Max Planck Research School for my financial support, which allowed me to focus on my research. I am also thankful to all the friends and colleagues in the Database and Information Systems group. Last but not least, I am greatly indebted to my wife for her constant encouragement and support throughout these years.

Contents

1 Introduction **1**

2 Distributed Matrix Completion **7**

2.1 The Matrix Completion Problem 9

2.2 Matrix Completion via Stochastic Gradient Descent 13

2.2.1 Gradient Descent (GD) 13

2.2.2 Stochastic Gradient Descent (SGD) 13

2.3 Parallelizing SGD-based Methods 16

2.3.1 Shared-Nothing Setting 16

2.3.2 Shared-Memory Setting 25

2.4 Matrix Completion via Alternating Minimizations 27

2.4.1 Alternating Least Squares (ALS) 27

2.4.2 Cyclic Coordinate Descent (CCD++) 29

2.5 Alternating Minimizations Versus SGD 30

2.5.1 Complexity Analysis 30

2.5.2 Experimental Evaluation 34

2.6 Summary 42

3 Distributed Mixed-Packing-Covering Linear Programming **43**

3.1 The MPC Linear Programs 43

3.2 Solving MPC-LPs (Feasibility) 45

3.3 Solving MPC-LPs (Optimization) 59

3.4 Parallelizing MPCSolver 60

3.5 Implementing MPCSolver 62

3.5.1 Starting Point 63
3.5.2 Adaptive Error Bounds 64
3.5.3 Adaptive Step Size 64
3.5.4 Convergence Test 65
3.5.5 Multiple Updates 65
3.6 Experimental Study 66
3.7 Related Work 67
3.7.1 PLP Solvers 68
3.7.2 MPC-LP Solvers 69
3.7.3 General Solvers 70
3.8 Summary 71

4 Generalized Bipartite Matching **73**
4.1 Problem Definition 75
4.2 Algorithms 77
4.2.1 MPCSolver for GBM 77
4.2.2 Obtaining an Integral Solution 78
4.3 Related Work 83
4.4 Experimental Results 84
4.4.1 Experimental Setup 85
4.4.2 Results for GBM-LP (Feasibility) 87
4.4.3 Results for GBM-LP (Optimality) 93
4.4.4 Results for Distributed Rounding 94
4.4.5 Results for GBM 95
4.5 Summary 96

5 Conclusion and Outlook **99**

Bibliography **103**

A Basic Notations **115**

List of Figures **117**

List of Tables **119**

List of Algorithms **121**

1 Introduction

Recommender systems have now attained considerable interest both in industry and in the research community. They are being extensively used in many e-commerce applications that expose the users to a large collection of items. Such systems intend to provide the users with suggestions that they might appreciate and thus assist them with finding appropriate items in the collection. Predicting user preferences over unseen items lies at the heart of any recommender system and is the key step to providing personalized recommendations. To this end, various approaches have been proposed in the literature, among which matrix completion techniques have been especially popular and obtained impressive performance in the context of collaborative filtering in recommender systems (Chen et al. 2012; Hu et al. 2008; Koren et al. 2009; Zhou et al. 2008). They are currently considered as one of the best single approaches in collaborative filtering but often combined with other models.

Consider for example an online DVD rental that offers a large collection of DVDs to its customers. Online video services like Netflix[1] or Amazon's Prime Instant Video[2] give their customers the opportunity to submit ratings on movies they have watched; this dataset can be naturally represented in matrix form where the rows correspond to customers, the columns to movies, and the entries to ratings provided by the customers. Matrix completion methods can be used to predict missing ratings for unseen movies based on the past available feedback and produce personalized recommendations. Once the preferences of users over movies are inferred, for each user, top-k movies with highest predicted ratings are typically selected and recommended for viewing. In practice, however, recommendations must not only

[1] www.netflix.com
[2] www.amazon.de/Prime-Instant-Video

take account of user preferences in isolation; a good recommendation engine also needs to take various constraints into consideration. For instance, an online video store should consider the availability of DVDs to reduce customer waiting times for accepted recommendations. Moreover, each user should be supplied with a small but sufficient number of suggestions for DVDs that the user is likely to be interested in. This problem can be naturally modeled as a generalized (i.e., many-to-many) bipartite matching problem in which a set of users must be matched to DVDs subject to certain objectives and constraints on the overall matching. Predicted ratings obtained by matrix completion represent the preference of users over movies and reflect the quality of matching a given DVD to a user in “isolation”. The goal is to provide recommendations of DVDs to users such that (1) users are recommended items that they are likely to be interested in, (2) every user gets neither too few nor too many suggestions, and (3) only available items are recommended to users.

Large applications may involve millions of users and items and billions of ratings; for instance, Netflix offers tens of thousands of movies for rental or streaming to more than 20M customers. In order to cope with datasets of such massive scales, parallel approaches are essential to achieve reasonable performance.

This thesis deals with key aspects of recommender systems and provides scalable solutions. In particular, we propose scalable optimization algorithms for matrix completion as well as graph matching problems like the one mentioned above. Figure 1.1 illustrates a simple basic framework[3] utilizing the optimization algorithms developed in this thesis to generate recommendations “under constraints”. The basic framework comprises two steps: We first apply matrix completion techniques to predict missing entries of the partially observed rating matrix. Next, we treat the recovered ratings as a metric, which captures user preferences over movies and use graph optimization to determine recommendations satisfying certain objectives and constraints. Our matching algorithm is based on linear programming techniques. To this end, we propose a distributed algorithm to approximately solve a “simple” yet expressive subclass of linear programs (LP), the so-called mixed packing-covering LPs (MPC-LP). These problems arise as LP relaxation of certain important combinatorial problems including the generalized matching problems.

In order to cope with datasets at massive scales, we consider a general shared-nothing architecture which allows asynchronous communication between processors. The main challenge in such a shared-nothing environment is how to effectively manage the communication between the compute nodes. The goal is to distribute the data across the compute nodes such that (1) each node operates preferably on non-overlapping subsets of the data so to minimize the communication between the

[3]Note that real recommender systems are usually more involved; they might consist of several components and might combine various complex models.

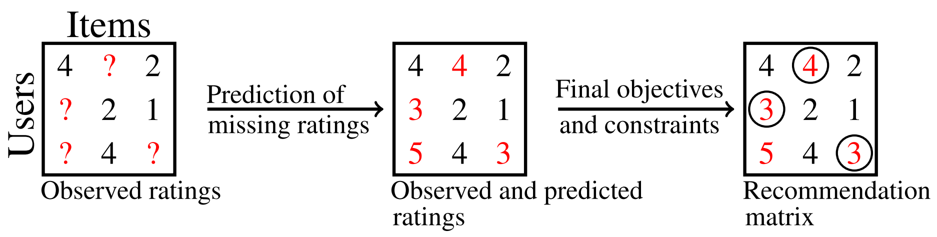


Figure 1.1: *A basic framework to generate recommendations under constraints. Observed ratings are shown in black, predicted in red, and selected for recommendation circled. Adapted from Charlin et al. (2012).*

compute nodes, and (2) the workload of each node is balanced as much as possible so to maximize the efficiency.

Contributions

This thesis provides efficient and scalable solutions to various optimization problems that arise at the heart of recommender systems. In particular, we deal with the task of predicting user preferences over unseen items, and generating recommendations satisfying certain objectives and constraints. We discuss each of these problems in Chapters 2, 3, and 4, respectively. Our contributions can be summarized as follows.

Matrix Completion

Chapter 2 concerns with matrix completion problems in the context of collaborative filtering in recommender systems. We review existing sequential, shared-memory, and shared-nothing approaches based on stochastic gradient descent as well as alternating-minimization ideas, and propose novel shared-nothing algorithms for large-scale matrix completion problems with millions of rows, millions of columns, and billions of entries. We focus on in-memory algorithms that run on a small cluster of commodity nodes, i.e., we assume that the input data fits into the aggregate memory of the cluster nodes. In contrast, most existing shared-nothing approaches for matrix completion are mainly designed for MapReduce (Das et al. 2010; Gemulla et al. 2011c; Liu et al. 2010; Zhou et al. 2008). In fact, it has been observed that MapReduce can be inefficient for the kind of iterative computations performed by matrix completion algorithms (Gemulla et al. 2011c). In the shared-nothing setting, there have been almost no studies of in-memory matrix completion algorithms based on programming models like MPI (2013) that allow asynchronous communication between processors and exploit multithreading.

Our algorithms are cache-friendly and exploit thread-level parallelism, in-memory processing, and asynchronous communication. Moreover, they are faster, more scalable, and less memory-intensive than existing MapReduce algorithms. We conduct a comprehensive comparison of the performance of both new and existing shared-nothing algorithms via a theoretical complexity analysis as well as an extensive experimental study. Our complexity analysis illuminates the various performance trade-offs and provides guidance in applying the algorithms to specific problems. We report results of an extensive set of experiments on both real-world and synthetic datasets of varying sizes.

Mixed Packing-Covering Linear Programming

In Chapter 3, we investigate scalable solutions for approximately solving MPC-LPs. MPC-LPs can be solved exactly using standard general-purpose solvers. However, these solvers fall short in coping with large practical problems. Parallel approaches, such as algorithms for shared-memory (Luby and Nisan 1993; Young 2001) or shared-nothing (Awerbuch and Khandekar 2009; Kuhn et al. 2006; Young 2001) architectures are essential for achieving reasonable performance at massive scales. Unfortunately, most of these algorithms can either handle only special cases, or suffer from high running times in practice.

We propose MPCSolver, a novel efficient distributed algorithm for approximately solving MPC-LPs and establish its convergence via a full theoretical analysis. MPCSolver requires a poly-logarithmic number of passes over the input, is simple, and easy to parallelize; it can be implemented in a few lines of code and is well-suited for parallel processing on GPUs, in shared-memory and shared-nothing architectures, as well as on MapReduce. We provide implementation issues that facilitate good performance in practice. In particular, we show how to distribute data effectively across the nodes in the cluster to minimize the communication costs and present a number of simple techniques to speed up MPCSolver in practice.

Generalized Bipartite Matching

Chapter 4 studies approximate solutions for large-scale generalized bipartite matching problems containing millions of vertices and billions of edges. We propose the first distributed algorithm for computing near-optimal solutions to such problems; in contrast, existing scalable solutions for bipartite matching problems (e.g., Huang and Jebara (2011); Morales et al. (2011)) can solely deal with special cases. Our approach rests on linear programming and randomized rounding. In particular, we utilize MPCSolver developed in Chapter 3 and present DDRounding, an efficient distributed randomized rounding algorithm. As a case study, we focus on an application in recommending multimedia items under certain constraints and conduct an extensive experimental study on both real and synthetic datasets of varying sizes. Our experiments indicate that both DDRounding and MPCSolver significantly outperform alternative approaches in terms of scalability and efficiency.

Finally, Chapter 5 concludes this thesis with a summary and a discussion of possible directions for future work. For a summary of notation used in this manuscript see Appendix A.

2 Distributed Matrix Completion

In this chapter,[1] we are concerned with low-rank matrix completion, an effective technique for statistical data analysis which recently has gained considerable attention in the data mining and machine learning community. At its heart, matrix completion is a variant of low-rank matrix factorization and involves recovering a partially observed and potentially noisy data matrix. Its most prominent application is perhaps in the context of collaborative filtering in recommender systems. Several other problems can be cast as a matrix completion problem, examples include, link prediction (Liben-Nowell and Kleinberg 2007), sensor localization (Drineas et al. 2006; Singer 2008), relation extraction (Riedel et al. 2013), etc.

In the setting of recommender systems, which we focus on in this chapter, matrix completion techniques are currently considered as one of the best single approaches and have attained remarkable performance in practice (Chen et al. 2012; Das et al. 2010; Gemulla et al. 2011c; Hu et al. 2008; Koren et al. 2009; Mackey et al. 2011; Recht and Ré 2013; Recht et al. 2011; Yu et al. 2012; Zhou et al. 2008). In such a setting, the rows in the data matrix correspond to users or customers, the columns to items such as movies or music pieces, and entries to feedback provided by users for items; the provided feedback is either explicit, e.g., in the form of numerical ratings,

[1]Parts of the material in this chapter have been jointly developed with Rainer Gemulla, Peter J. Haas, Yannis Sismanis, and Christina Teflioudi. The chapter is based on Gemulla et al. (2011b), Teflioudi et al. (2012), and Makari et al. (2014). The copyright of Gemulla et al. (2011b) is held by NIPS; the original publication is available at http://biglearn.org/2011/files/papers/biglearn2011_submission_15.pdf. The copyright of Teflioudi et al. (2012) is held by IEEE; the original publication is available at http://doi.ieeecomputersociety.org/10.1109/ICDM.2012.120. The copyright of Makari et al. (2014) is held by Springer; the original publication is available at www.springer.com and http://dx.doi.org/10.1007/s10115-013-0718-7.

which directly captures users' personal taste, or implicit, e.g., page reviews, which indirectly reflects users opinion about the item. Matrix completion is an effective tool for analyzing such dyadic data in that it discovers and quantifies the interactions between users and items.

The key principle in matrix completion is to fit a low-rank model,[2] which captures the important aspects of the data matrix from available past feedback. To this end, both users and items are mapped to a joint latent vector space of a small dimensionality $r \geq 1$ (usually less than 100). In particular, we associate r features to each user and to each item. For a particular item, the feature values reflect the extent to which the item possesses those feature. For a given user, the elements of the corresponding feature vector measure the relevance of those features for the user. These features are usually latent in that they do not have explicit semantic interpretations; feature values are learned from available entries. In the simplest case, which we focus on in this chapter, interactions between users and items are measured as the inner product of the feature vectors of the corresponding users and items in that latent vector space. In general, the estimations might depend on other additional data such as the time stamp of rating, user and item biases, implicit feedback, and so on.

Many real-world applications involve matrices with millions of rows, millions of columns, and billions of entries. Meanwhile, many online movie stores like Netflix have large volume of data available, which captures users' satisfaction about numerous movies. For example, Netflix collected over five billion ratings for more than 80k movies from its more than 20M customers (Amatriain and Basilico 2012; Bennett and Lanning 2007). Similarly, Yahoo! Music gathered billions of user ratings for musical pieces (Dror et al. 2011). Clearly, algorithms for matrix completion must be parallelized to be able to cope with such large instances efficiently.

Our goal is to develop efficient and scalable algorithms for large matrix completion problems. We focus on *in-memory* algorithms that run on a small cluster of commodity nodes, i.e., we assume that the data and feature vectors fit in the aggregate memory of the cluster nodes. A wide range of matrix completion tasks can be handled effectively in such a setup. Consider, for example, an extremely large hypothetical problem instance in which the input matrix contains 20M rows, 1M columns, and 1% of the entries are observed. (By comparison, the data in Bennett and Lanning (2007); Dror et al. (2011) imply that 0.3% of Netflix ratings and 0.4% Yahoo! Music entries are revealed, and the Netflix problem has fewer rows and columns than the hypothetical problem.) If a rank-100 factorization is used and assuming that each entry is a 64-bit number, then the total data and model size is

[2]Note that the user-movie-rating matrix may be low-rank since it is commonly believed that only a few features contribute to individual's preferences (Koren 2008).

approximately 1.5TB.[3] Small shared-nothing clusters can easily handle this amount of data. In such a setting, there have been almost no studies of in-memory matrix completion algorithms based on programming models, such as MPI (2013), that allow asynchronous communication between processors. Similarly, the possibilities for exploiting multithreading have not been considered. In a multithreaded shared-nothing architecture, different processing nodes do not share memory, but threads at the same node can share memory. However, most existing distributed shared-nothing algorithms for matrix factorization are designed for MapReduce (Das et al. 2010; Gemulla et al. 2011c; Liu et al. 2010; Zhou et al. 2008). Compared to our setting, MapReduce algorithms have multiple drawbacks: (1) they need to repeatedly read the input from disk into memory, (2) they are limited to the MapReduce programming model, and (3) they may suffer from runtime overheads of popular implementations such as Hadoop (which has been designed for much larger clusters and different workloads).

Popular algorithms for large-scale matrix completion are based on stochastic gradient descent (SGD) (Koren et al. 2009) or alternating minimization ideas (Yu et al. 2012; Zhou et al. 2008). We review parallel (shared-memory and shared-nothing) and MapReduce variants of these algorithms. We propose a set of shared-nothing algorithms that are faster, more scalable, and less memory-intensive than existing MapReduce algorithms. More specifically, our Asynchronous SGD (ASGD) and DSGD++ are novel variants of the SGD algorithm. ASGD is inspired by recent work on distributed LDA (Smola and Narayanamurthy 2010) and DSGD++ is based on the MapReduce algorithm of Gemulla et al. (2011c). Both algorithms are cache-friendly, and exploit thread-level parallelism and asynchronous communication. Our DALS is a scalable variant of the alternating least-squares (ALS) algorithm of Zhou et al. (2008) that exploits thread-level parallelism to speed up processing and reduce the memory footprint.

This chapter is organized as follows. In Section 2.1, we state the matrix completion problem formally. In Section 2.2, we focus on stochastic gradient descent. In Section 2.3, we first present our novel shared-nothing algorithms ASGD and DSGD++, and then proceed with a survey of prior shared-memory SGD algorithms. Algorithms based on alternating minimization approaches are reviewed afterwards in Section 2.4. Finally in Section 2.5, different algorithms are compared both via a theoretical complexity analysis as well as an extensive set of experiments.

2.1 The Matrix Completion Problem

As already mentioned, a special instance of matrix completion is the “Netflix problem” (Bennett and Lanning 2007) of recommending movies to customers.

[3]Here we assume that the input matrix is stored in sparse format.

2.1. The Matrix Completion Problem

Consider a vendor (e.g., Netflix) that offers movies for rental to its customers each of whom are given the opportunity to provide feedback about their preferences by rating movies (e.g., Netflix customers may rate movies with 1 to 5 stars). The feedback can be represented in matrix form, for example

$$\begin{array}{c} \\ \textit{Alice} \\ \textit{Bob} \\ \textit{Charlie} \end{array} \begin{array}{c} \begin{array}{ccc} \textit{Avatar} & \textit{Inception} & \textit{Shrek} \end{array} \\ \begin{pmatrix} ? & 4 & 2 \\ 3 & 2 & ? \\ 5 & ? & 3 \end{pmatrix} \end{array}.$$

In addition to the actual ratings, other forms of feedback might also be available such as time of rating and click history. The task is to estimate the missing entries (denoted by "?"), so the movies with high predicted values can potentially be recommended to users for watching. This approach has proven successful in practice; see Koren et al. (2009) for an excellent discussion of the underlying intuition.

In the following, we first introduce some notation and then proceed with the problem statement. Along with the notation from Appendix A, we use the notation summarized in Table 2.1 throughout this chapter. Let *training set* $\Omega = \{\,\omega_1, \ldots, \omega_N\,\}$ denote the set of observed entries in $m \times n$ input matrix $\boldsymbol{V}$, where $\omega_k = (i_k, j_k)$, $k \in [1, N]$, $i_k \in [1, m]$, and $j_k \in [1, n]$. In what follows, we assume without loss of generality that $m \geq n$.[4] Let N_{i*} and N_{*j} denote the number of revealed entries in row i and column j, respectively. Finally, let $r \ll \min(m, n)$ denote the rank parameter. The task is to find an $m \times r$ row-factor matrix $\boldsymbol{L}$ and an $r \times n$ column-factor matrix $\boldsymbol{R}$ such that $\boldsymbol{V} \approx \boldsymbol{LR}$, i.e., we aim to approximate $\boldsymbol{V}$ by the low-rank matrix $\boldsymbol{LR}$. The quality of the approximation is controlled by an application-dependent loss function $L(\boldsymbol{L}, \boldsymbol{R})$ that measures the difference between the observed entries in $\boldsymbol{V}$ and the corresponding entries in $\boldsymbol{LR}$ (we suppress the dependence on $\boldsymbol{V}$ for brevity). The matrix completion problem asks to find the loss-minimizing factor matrices, i.e.,

$$(\boldsymbol{L}^*, \boldsymbol{R}^*) = \operatorname*{argmin}_{\boldsymbol{L},\boldsymbol{R}} L(\boldsymbol{L}, \boldsymbol{R}). \tag{2.1}$$

The matrix $\boldsymbol{L}^*\boldsymbol{R}^*$ can be considered as a "completed version" of $\boldsymbol{V}$, and each unrevealed entry $\boldsymbol{V}_{ij}$ is predicted by $[\boldsymbol{L}^*\boldsymbol{R}^*]_{ij}$.

The loss function L may also encode additional information including user and item biases, implicit feedback, temporal aspects, confidence level, as well as regularization terms to avoid over-fitting. Table 2.2 summarizes some popular loss functions.

[4] Under this assumption, algorithms in the shared-nothing environment move mostly column-factor data between the nodes.

Table 2.1: *Notation for matrix completion algorithms*

Symbol	Description
$\boldsymbol{V}$	Data matrix
m, n	Number of rows & columns of $\boldsymbol{V}$
Ω	Set of revealed entries in $\boldsymbol{V}$
N	Number of revealed entries in $\boldsymbol{V}$
N_{i*}	Number of revealed entries in row i of $\boldsymbol{V}$
N_{*j}	Number of revealed entries in column j of $\boldsymbol{V}$
r	Rank of the factorization
$\boldsymbol{L}, \boldsymbol{R}$	Factor matrices
$\boldsymbol{E}$	Residual matrix
w	Number of compute nodes
t	Number of threads per compute node
p	Total number of threads
b	Number of row/column blocks (SSGD)
T	Repetition parameter (CCD++)
s	Number of shufflers (Jellyfish)

The most basic loss function is the squared loss

$$L_{\mathrm{Sl}}(\boldsymbol{L}, \boldsymbol{R}) = \sum_{(i,j)\in\Omega} (\boldsymbol{V}_{ij} - [\boldsymbol{LR}]_{ij})^2.$$

L_{L2} incorporates L2 regularization and is closely related to the problem of minimizing the nuclear norm of the reconstructed matrix (Recht and Ré 2013). L_{L2w} incorporates weighted L2 regularization (Zhou et al. 2008), in which the amount of regularization depends on the number of revealed entries. This particular loss function was a key ingredient in the best performing solutions of both the Netflix competition and the 2011 KDD-Cup (Chen et al. 2012; Koren et al. 2009; Zhou et al. 2008).

Our formulation of the matrix completion is motivated by its applications in data mining settings where a fixed set of training data and a loss function are provided, and the goal is to compute loss-minimizing factor matrices as efficiently as possible. There is a large body of work in the literature that assumes a “true” underlying $\boldsymbol{V}$ matrix together with a stochastic process that generates the training data from $\boldsymbol{V}$; both noiseless and noisy cases have been studied. The task is to statistically infer the true $\boldsymbol{V}$ matrix from the training data, where the inference algorithm may exploit knowledge about the stochastic process. In general, exact low-rank matrix completion is impossible without making any assumptions about the structure of the input matrix $\boldsymbol{V}$ and the training set Ω. A typical assumption (among other

Table 2.2: *Popular loss functions for matrix completion*

Loss	Definition
L_{Sl}	$\sum_{(i,j)\in\Omega}(\boldsymbol{V}_{ij} - [\boldsymbol{LR}]_{ij})^2$
L_{L2}	$L_{\text{Sl}} + \lambda\Big(\sum_{ik} \boldsymbol{L}_{ik}^2 + \sum_{kj} \boldsymbol{R}_{kj}^2\Big)$
L_{L2w}	$L_{\text{Sl}} + \lambda\Big(\sum_{ik} N_{i*}\boldsymbol{L}_{ik}^2 + \sum_{kj} N_{*j}\boldsymbol{R}_{kj}^2\Big)$

Loss	Local loss
L_{Sl}	$(\boldsymbol{V}_{ij} - [\boldsymbol{LR}]_{ij})^2$
L_{L2}	$(\boldsymbol{V}_{ij} - [\boldsymbol{LR}]_{ij})^2 + \lambda\sum_k(N_{i*}^{-1}\boldsymbol{L}_{ik}^2 + N_{*j}^{-1}\boldsymbol{R}_{kj}^2)$
L_{L2w}	$(\boldsymbol{V}_{ij} - [\boldsymbol{LR}]_{ij})^2 + \lambda\sum_k(\boldsymbol{L}_{ik}^2 + \boldsymbol{R}_{kj}^2)$

assumptions) in this setting is that the training set Ω is generated uniformly at random; see for example the results in Candes and Recht (2009); Krishnamurthy and Singh (2013); Recht (2011) for the noiseless setting and in Candès and Plan (2010); Negahban and Wainwright (2012) for the case in presence of Gaussian noise. Note that these settings differ from ours as well as many real applications where the training set is assumed to be fixed. In fact, most available datasets (e.g., the Netflix dataset) for recommendation tasks are highly non-uniform in the sense that the number of available ratings for each individual movie varies widely.

In this chapter, we focus on loss functions that admit a summation form; following Chu et al. (2006), we say that a loss function is in *summation form* if it is written as a sum of *local losses* L_{ij} that occur at only the revealed entries of $\boldsymbol{V}$, i.e.,

$$L(\boldsymbol{L}, \boldsymbol{R}) = \sum_{(i,j)\in\Omega} L_{ij}(\boldsymbol{L}_{i*}, \boldsymbol{R}_{*j}).$$

Table 2.2 shows examples of loss functions in summation form together with the corresponding local losses. We refer to the gradient of a local loss as a *local gradient*; by the linearity of the differentiation operator, the gradient of a loss function having summation form can be represented as a sum of local gradients:

$$L'(\boldsymbol{L}, \boldsymbol{R}) = \sum_{(i,j)\in\Omega} L'_{ij}(\boldsymbol{L}_{i*}, \boldsymbol{R}_{*j}).$$

In the following, we focus on popular algorithms based on stochastic gradient descent (Section 2.2) and alternating minimizations (Section 2.4), which have been shown to be effective in the collaborative filtering setting (Gemulla et al. 2011c; Recht and Ré 2013; Recht et al. 2011; Zhou et al. 2008).

2.2 Matrix Completion via Stochastic Gradient Descent

We first describe the basic SGD algorithm and then discuss various parallelization strategies. For brevity, we write $L(\theta)$ and $L'(\theta)$, where $\theta = (\boldsymbol{L}, \boldsymbol{R})$, to denote the loss function and its gradient. Denote by $\nabla_{\boldsymbol{L}} L$ (resp. $\nabla_{\boldsymbol{R}} L$) the $m \times r$ (resp. $r \times n$) matrix of the partial derivatives of L w.r.t. to the entries in $\boldsymbol{L}$ (resp. $\boldsymbol{R}$). Then $L' = (\nabla_{\boldsymbol{L}} L, \nabla_{\boldsymbol{R}} L)$. For example,

$$[\nabla_{\boldsymbol{L}} L_{\text{Sl}}]_{ik} = -2 \sum_{j:(i,j)\in\Omega_{i*}} \boldsymbol{R}_{kj}(\boldsymbol{V}_{ij} - [\boldsymbol{LR}]_{ij}),$$

where Ω_{i*} denotes the set of revealed entries in row $\boldsymbol{V}_{i*}$.

2.2.1 Gradient Descent (GD)

A number of gradient-based methods have been proposed in the literature for matrix completion. Perhaps, the most basic one is the gradient descent (GD). Starting from some initial point, GD iteratively takes small steps in the opposite direction of the gradient:

$$\theta_{n+1} = \theta_n - \epsilon_n L'(\theta_n),$$

where n denotes the step number and $\{ \epsilon_n \}$ is a sequence of decreasing step sizes. (Throughout, we assume that each ϵ_n is non-negative and finite.) Note that $-L'(\theta_n)$ corresponds to the direction of steepest descent. Under appropriate conditions, GD achieves a linear rate of convergence; faster convergence rates can be obtained by deploying second order methods, e.g., quasi-Newton methods such as L-BFGS-B (Byrd et al. 1995).

2.2.2 Stochastic Gradient Descent (SGD)

Stochastic gradient descent is based on GD, however, instead of using the function's gradient $L'(\theta)$, it utilizes a noisy estimate $\hat{L}'(\theta)$ of $L'(\theta)$. In order to find a minimizer θ^* of $L(\theta)$, SGD starts with some initial value θ_0, and refines the parameter value by iterating the stochastic difference equation

$$\theta_{n+1} = \theta_n - \epsilon_n \hat{L}'(\theta_n).$$

Thus, SGD can be seen as a noisy version of GD. The gradient estimate is obtained by scaling up just one of the local gradients, i.e., $\hat{L}'(\theta) = NL'_{ij}(\theta)$ for some $(i,j) \in \Omega$. At each SGD step a different training point (i,j) is selected according to a *training point schedule*; see below. Note that the local gradients at point (i,j) depend only on $\boldsymbol{V}_{ij}$, $\boldsymbol{L}_{i*}$ and $\boldsymbol{R}_{*j}$. Therefore, only a single row $\boldsymbol{L}_{i*}$ and a single column $\boldsymbol{R}_{*j}$ are updated during each SGD step; all other rows and columns remain unaffected.

The convergence properties of SGD can be established by using stochastic approximation theory. In particular, it can be shown that under certain conditions (Kushner and Yin 2003), SGD converges to a set of stationary points satisfying $L'(\theta) = 0$. These stationary points can be minima, maxima, or saddle points. However, the noisy gradient estimations reduce the likelihood of getting stuck in maxima or saddle points so that the SGD algorithm usually converges to a (local) minimum of L. To increase the likelihood of finding a global minimum, one could run SGD multiple times, starting from a set of randomly chosen initial points.

Note that SGD performs many quick-and-dirty steps in each pass over the training data, whereas GD (or a quasi-Newton method like L-BFGS-B) performs a single careful step. For large matrices, the increased number of SGD steps results in a faster convergence compared to GD (see, e.g., Gemulla et al. (2011c)). Moreover, the noisy estimation of the descent direction helps the algorithm escaping from local minima, especially during the early stages of the descent. The performance of SGD highly depends on the step size sequence and the training point schedule. In the following, we discuss these two issues in detail.

Step size sequence. A number of schemes for choosing the step size in each SGD step have been proposed in the literature. Often step size sequences of the form $\epsilon_n = \frac{1}{n^\alpha}$ with $\alpha \in (0.5, 1]$ are used; this choice of step sizes ensures asymptotic convergence (Kushner and Yin 2003).[5] In practice, however, one may follow other steps size choices to achieve faster convergence. A simple adaptive scheme for selecting the step size, termed the *bold driver* heuristic (Battiti 1989), has been shown to be very effective in practice (Gemulla et al. 2011c), even though without asymptotic convergence guarantees. In all of our SGD implementations, we applied this heuristic, which has worked extremely well in our experiments. The bold driver heuristic exploits the fact that current loss can be computed after every epoch and proceeds as follows. We refer to one GD step or a sequence of N SGD steps as an *epoch*; an epoch roughly corresponds to a single pass over the training data. Starting from an initial step size ϵ_0, the algorithm increases the step size by a small percentage (say by 5%) if the loss after every epoch has decreased, or drastically decreases the step size (say by 50%) if the loss has increased. Within each epoch, the step size remains unchanged. The initial step size ϵ_0 is obtained by trying different values on a small sample (say 0.1%) of the training points and selecting the best-performing one.

Training point schedule. We focus on three common training point schedules for processing the training data:

[5]"Convergence" refers to running the algorithm until some convergence criterion is met; "asymptotic convergence" means that the algorithm converges to the true solution as the runtime goes to $+\infty$.

- SEQ: Processing Ω sequentially in some fixed order,
- WR: Sampling from Ω with replacement, and
- WOR: Sampling from Ω without replacement.

All of these training point schedules guarantee convergence, however, they require different number of epochs until convergence. WR provides the best convergence rates. In fact, the provable rates for WOR are considerably slower (Nedic and Bertsekas 2000). In practice and on large datasets, however, WOR often outperforms WR in terms of the number of epochs to convergence (Bottou and Bousquet 2007). As discussed in Recht and Ré (2013), one reason is due to the *coupon collector phenomenon*. According to this phenomenon, for a large number N of data points, on average $N \log N$ samples are required in order to see each data point at least once. Clearly, in WOR, exactly N samples are required to touch all of N data points. As the size of the dataset increases, this discrepancy becomes more evident. SEQ requires significantly more epochs than both WR and WOR, and converges to an inferior solution. On the other hand, each individual SGD step is faster under SEQ. This is due to the fact that SEQ exhibits better memory locality compared to WR and WOR (see Section 2.5.2D).

SGD++

As discussed above, SEQ benefits from a sequential memory access pattern. In contrast, WR and WOR access data in memory discontinuously, which in turn leads to a high cache-miss rate and performance degradation. To reduce this performance gap, we will enhance WOR with latency-hiding techniques. In more details, we prefetch the required data into the CPU cache before it is accessed by the SGD algorithm (e.g., using gcc's `__builtin_prefetch` macro). In the beginning of each epoch, we precompute and store a permutation Π of $\{1, \ldots, N\}$ that indicates the order in which training points are to be processed. In the n-th step, the SGD algorithm accesses the values $\boldsymbol{V}_{i_n^\Pi j_n^\Pi}$, $\boldsymbol{L}_{i_n^\Pi *}$, and $\boldsymbol{R}_{* j_n^\Pi}$, whose common index value (i_n^Π, j_n^Π) is determined from the $\Pi(n)$-th entry of Ω. We access Π and then prefetch the index value (i_n^Π, j_n^Π) during SGD step $n-2$ (so that it is in the CPU cache at step $n-1$), and the values $\boldsymbol{V}_{i_n^\Pi j_n^\Pi}$, $\boldsymbol{L}_{i_n^\Pi *}$, and $\boldsymbol{R}_{* j_n^\Pi}$ in SGD step $n-1$ (so that they are in the CPU cache at step n). Note that Π itself is accessed sequentially so that no explicit prefetching is needed. We refer to SGD with prefetching as SGD++; see Algorithm 1. In our experiments, SGD++ was up to 13% faster than SGD (see Section 2.5.2C).

Algorithm 1 The SGD++ algorithm for matrix completion

```
Require: Incomplete matrix $\boldsymbol{V}$, initial values $\boldsymbol{L}$ and $\boldsymbol{R}$
1: while not converged do                                                  // epoch
2:     Create random permutation $\Pi$ of $\{1, \ldots, N\}$              // WOR schedule
3:     for $n = 1, 2, \ldots, N$ do                                         // step
4:         Prefetch indexes $(i^{\Pi}_{n+2}, j^{\Pi}_{n+2}) \in \Omega$ for next but one step
5:         Prefetch data $\boldsymbol{V}_{i^{\Pi}_{n+1} j^{\Pi}_{n+1}}$, $\boldsymbol{L}_{i^{\Pi}_{n+1} *}$, $\boldsymbol{R}_{* j^{\Pi}_{n+1}}$ for next step
6:         $\boldsymbol{L}'_{i^{\Pi}_n *} \leftarrow \boldsymbol{L}_{i^{\Pi}_n *} - \epsilon_n N \nabla_{\boldsymbol{L}_{i^{\Pi}_n *}} L_{i^{\Pi}_n j^{\Pi}_n}(\boldsymbol{L}, \boldsymbol{R})$
7:         $\boldsymbol{R}_{* j^{\Pi}_n} \leftarrow \boldsymbol{R}_{* j^{\Pi}_n} - \epsilon_n N \nabla_{\boldsymbol{R}_{* j^{\Pi}_n}} L_{i^{\Pi}_n j^{\Pi}_n}(\boldsymbol{L}, \boldsymbol{R})$
8:         $\boldsymbol{L}_{i^{\Pi}_n *} \leftarrow \boldsymbol{L}'_{i^{\Pi}_n *}$
```

2.3 Parallelizing SGD-based Methods

Sequential methods for matrix completion are effective only for problems at small scale. However, even on problems of moderate size they may require substantial amount of time until convergence and scalability becomes a bottleneck. Therefore, parallel (shared-memory or shared-nothing) versions of SGD have been developed.

The key challenge in parallelizing SGD is that SGD steps might depend on each other. In particular, if two SGD steps select the training points that lie in the same row (resp. column), both of these steps modify the same row (resp. column) and updates to the same row (resp. column) will be overwritten. In the following, we describe some approaches to overcome this issue for shared-nothing parallel processing environments. All shared-nothing approaches partition the data and factor matrices into a carefully chosen set of blocks; we refer to such a partitioning as a *blocking*. Denote by w the number of compute nodes and by t the number of threads per node; the total number of available threads is thus $p = wt$. We defer the discussion of parallel shared-memory algorithms to Section 2.3.2.

2.3.1 Shared-Nothing Setting

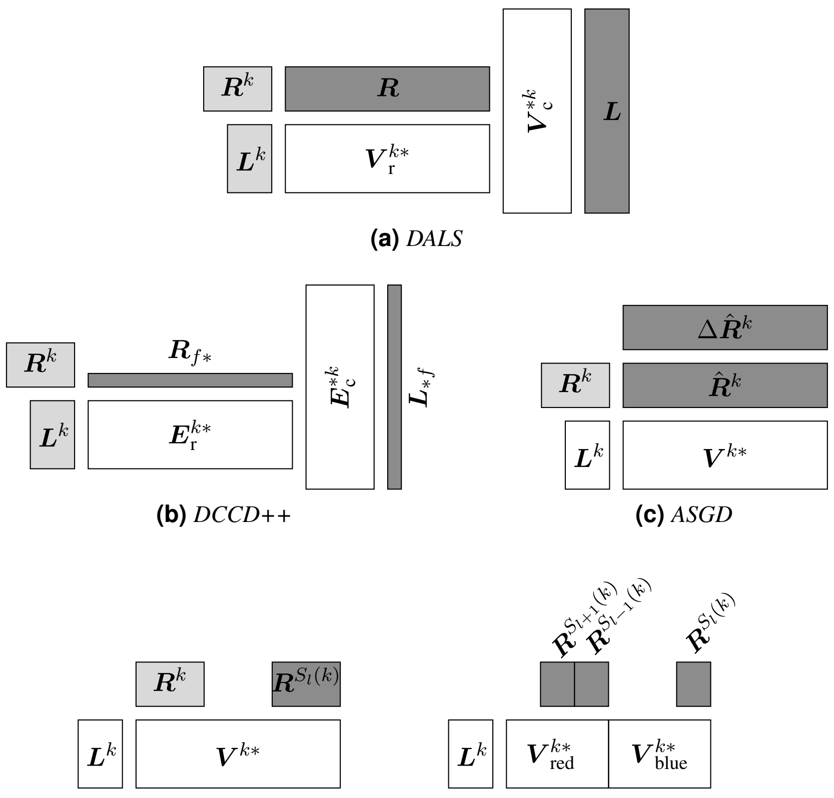
Distributed shared-nothing algorithms are designed for large-scale problems which exceed the main memory capacity of a single compute node. We now discuss why, in general, distributing SGD effectively is challenging. Throughout, we assume that t is no larger than the number of available (physical or virtual) threads at each compute node. The main challenge in a shared-nothing environment is to effectively manage the communication between the compute nodes. Ideally, our goal is to partition the input data and factors across the cluster such that (1) each node operates on a disjoint subset of the data and factors so that each partition can be processed independently, and (2) each node receives roughly the same amount of data so that the workload is balanced and distributed processing is effective. In general, however, these goals are not achievable at the same time. To see this, assume to the contrary that there exists such a partitioning of input matrix $\boldsymbol{V}$, and factor matrices $\boldsymbol{L}$ and $\boldsymbol{R}$ and that some node k is responsible for training points $\Omega_k \subseteq \Omega$. Moreover, suppose that training point $(i, j) \in \Omega_k$. Note that performing an SGD step on (i, j) updates $\boldsymbol{L}_{i*}$ and $\boldsymbol{R}_{*j}$ using the gradient estimate L'_{ij}. Since by assumption these parameters are not updated by any other node, all of the training points in row i and column j must also be in Ω_k, i.e., $(i, j) \in \Omega_k \implies \Omega_{i*} \cup \Omega_{*j} \subseteq \Omega_k$. Thus, Ω_k forms a submatrix of $\boldsymbol{V}$ that contains *all* revealed entries of any of its rows or columns. We can form w balanced partitions if and only if the rows and columns of $\boldsymbol{V}$ can be permuted to obtain a $w \times w$ block-diagonal matrix with a balanced number of revealed entries in each diagonal block. This is not possible in general; indeed, most (or even all) revealed entries usually concentrate in a single block.

In the sequel, we discuss two different approaches to circumvent this difficulty: stratified SGD (Gemulla et al. 2011c) and asynchronous SGD. With the exception of DSGD-MR discussed in Section 2.3.1C, all algorithms that we describe here assume that nodes can directly communicate with each other asynchronously, using a protocol such as MPI.

A. Asynchronous SGD (ASGD)

Assume that $\boldsymbol{V}$ and conformingly $\boldsymbol{L}$ are blocked row-wise $w \times 1$ while $\boldsymbol{R}$ is blocked column-wise $1 \times w$. At each node k, we store blocks $\boldsymbol{V}^{k*}$, $\boldsymbol{L}^k$, and $\boldsymbol{R}^k$. Note that under this blocking, updates to $\boldsymbol{L}$ are node-local and can be performed independently, whereas updates to $\boldsymbol{R}$ are either local or remote. We refer to $\boldsymbol{R}_{*j}$ as the *master copy* of the j-th column of $\boldsymbol{R}$ and to the node that stores $\boldsymbol{R}_{*j}$ as the *master node*. A naive way to parallelize SGD in a distributed shared-nothing setting is as follows: To process training point (i, j) at some node k, the master copy $\boldsymbol{R}_{*j}$ is first locked at its master node. Next, $\boldsymbol{R}_{*j}$ is fetched and $\boldsymbol{L}_{i*}$ and $\boldsymbol{R}_{*j}$ are updated locally. Finally, the new value of $\boldsymbol{R}_{*j}$ is sent back to its master node and the lock is released. This *asynchronous algorithm* is clearly impractical, since performing SGD steps is inexpensive so that most of the time is spent on communicating columns of $\boldsymbol{R}$.

Our ASGD algorithm avoids this problem by maintaining a *working copy* $\hat{\boldsymbol{R}}^k_{*j}$ of each column $\boldsymbol{R}_{*j}$ at each node k. Initially, all the working copies agree with their corresponding master copies. We now run SGD updates at each node as above, but update the working copy $\hat{\boldsymbol{R}}^k_{*j}$ instead of the master copy when processing (i, j). Note that up to this point there is no need for obtaining locks and synchronous communication. However, the working copies still need to be coordinated to ensure correctness. In the context of perceptron training, McDonald et al. (2010) proposed averaging the working copies once at the end of every epoch. In our case, however, nodes can communicate continuously, which allows us to perform averaging more


$\boldsymbol{R}^k$
$\boldsymbol{R}$
$\boldsymbol{V}_{\mathrm{c}}^{*k}$
$\boldsymbol{L}$
$\boldsymbol{L}^k$
$\boldsymbol{V}_{\mathrm{r}}^{k*}$
$\boldsymbol{R}_{f*}$
$\boldsymbol{E}_{\mathrm{c}}^{*k}$
$\boldsymbol{L}_{*f}$
$\boldsymbol{E}_{\mathrm{r}}^{k*}$
$\Delta\hat{\boldsymbol{R}}^k$
$\hat{\boldsymbol{R}}^k$
$\boldsymbol{V}^{k*}$
$\boldsymbol{R}^{S_l(k)}$
$\boldsymbol{R}^{S_{l+1}(k)}$
$\boldsymbol{R}^{S_{l-1}(k)}$
$\boldsymbol{V}_{\mathrm{red}}^{k*}$
$\boldsymbol{V}_{\mathrm{blue}}^{k*}$

(a) *DALS*

(b) *DCCD++* **(c)** *ASGD*

(d) *DSGD-MR (in-memory)* **(e)** *DSGD++*

Figure 2.1: *Memory layout used on node k by the shared-nothing algorithms ($t = 1$). Node-local data is shown in white, master copies in light gray, and temporary data in dark gray.*

frequently, namely also during each epoch. To this end, each node sends its *update vector* $\Delta\hat{\boldsymbol{R}}_{*j}^k$ to the master node from time to time, where $\Delta\hat{\boldsymbol{R}}_{*j}^k$ refers to the sum of updates to $\hat{\boldsymbol{R}}_{*j}^k$ since the last averaging. Once a master node receives all the update vectors $\Delta\hat{\boldsymbol{R}}_{*j}^1, \ldots, \Delta\hat{\boldsymbol{R}}_{*j}^w$, it adds their average to the master copy and broadcasts the result. Each node k then updates its working copy and integrates all local changes that have been accumulated meanwhile. The memory layout of ASGD is shown in Figure 2.1c. In the figure, $\hat{\boldsymbol{R}}^k = (\hat{\boldsymbol{R}}_{*1}^k \mid \hat{\boldsymbol{R}}_{*2}^k \mid \cdots \mid \hat{\boldsymbol{R}}_{*n}^k)$ and similarly for $\Delta\hat{\boldsymbol{R}}^k$.

In ASGD, each node operates on different working copies of $\boldsymbol{R}$ in parallel. Therefore, updates to a column of $\hat{\boldsymbol{R}}_{*j}$ at some node are not immediately visible to the other nodes. However, when the delay between updating a column and broadcasting the update is bounded, asynchronous SGD provably converges to a stationary point of L (Tsitsiklis et al. 1986). Note that under our blocking scheme, we only need to average a subset of parameters, i.e., $\boldsymbol{R}$ but not $\boldsymbol{L}$; this idea is motivated by the distributed LDA method for text mining (Smola and Narayanamurthy 2010). In our actual implementation, ASGD sends updates continuously during and once after every epoch. As a result, updates are communicated as often as possible and the local copies are consistent with the master copy after every epoch. The latter property also allows us to compute the loss across compute nodes after every epoch and apply the bold driver heuristic for step size selection. Furthermore, we make sure that averaging is non-blocking. Finally, rather than running ordinary sequential SGD on each node, we run a multithreaded version of SGD called PSGD, which is described in Section 2.3.2A. In addition to the threads performing PSGD, one single thread takes care of averaging. Note that this latter thread has low CPU utilization since the time to compute the update vectors is swamped by communication costs.

B. Stratified SGD (SSGD)

Gemulla et al. (2011c) presented an alternative approach to parallelizing SGD, which utilizes a stratification technique and avoids inconsistent updates. We refer to this method as stratified SGD (SSGD). SSGD serves as a basis for DSGD-MR and DSGD++ algorithms for the shared-nothing environment. We now discuss the main ideas behind SSGD in more details.

Assume that the input matrix is blocked $b \times b$, where b is chosen to be larger than or equal to the number available threads; the corresponding factor matrices are blocked conformingly:

$$\begin{array}{c} \\ \boldsymbol{L}^1 \\ \boldsymbol{L}^2 \\ \vdots \\ \boldsymbol{L}^b \end{array} \begin{array}{c} \begin{array}{cccc} \boldsymbol{R}^1 & \boldsymbol{R}^2 & \cdots & \boldsymbol{R}^b \end{array} \\ \begin{pmatrix} \boldsymbol{V}^{11} & \boldsymbol{V}^{12} & \cdots & \boldsymbol{V}^{1b} \\ \boldsymbol{V}^{21} & \boldsymbol{V}^{22} & \cdots & \boldsymbol{V}^{2b} \\ \vdots & \vdots & \ddots & \vdots \\ \boldsymbol{V}^{b1} & \boldsymbol{V}^{b2} & \cdots & \boldsymbol{V}^{bb} \end{pmatrix} \end{array}.$$

In order to ensure that all b^2 blocks contain N/b^2 training points in expectation, we randomly shuffle rows and columns of $\boldsymbol{V}$ before blocking. To run SGD on some block $\boldsymbol{V}^{ij}$, the algorithm requires access to matrices $\boldsymbol{L}^i$ and $\boldsymbol{R}^j$ only. It is easy to see that SGD can process each block on the main diagonal (i.e., $\boldsymbol{V}^{11}, \ldots, \boldsymbol{V}^{bb}$) independently in parallel. Following Gemulla et al. (2011c), we say that two different blocks $\boldsymbol{V}^{ij}$ and $\boldsymbol{V}^{i'j'}$ are *interchangeable* whenever $i \neq i'$ and $j \neq j'$, i.e., they share neither rows nor columns. Moreover, a set of b pairwise interchangeable blocks is called a *stratum*; the set of all strata is denoted by $\mathscr{S}$; see Figure 2.2 for an example. Thus, we can process all b blocks in $s \in \mathscr{S}$ in parallel. We can think of a stratum as a bijective map from a row-block index k to a column-block index

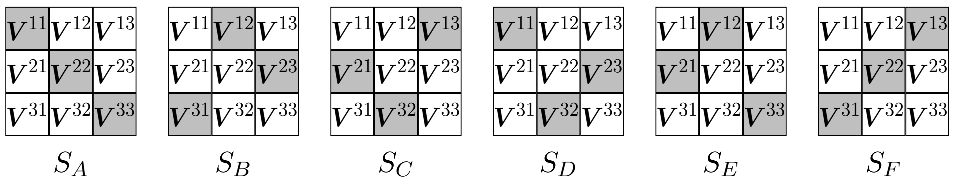

V^{11} V^{12} V^{13} V^{21} V^{22} V^{23} V^{31} V^{32} V^{33}
S_A
S_B
S_C
S_D
S_E
S_F

Figure 2.2: *Strata used by SSGD for a 3×3 blocking of $\boldsymbol{V}$*

Algorithm 2 The SSGD algorithm for matrix completion (Gemulla et al. 2011c)

```
Require: Incomplete matrix $V$, initial values $L$ and $R$, blocking parameter $b$
1: Block $V$ / $L$ / $R$ into $b \times b$ / $b \times 1$ / $1 \times b$ blocks
2: while not converged do                                        // epoch
3:   Pick step size $\epsilon$
4:   for $s = 1, \ldots, b$ do                                   // subepoch
5:     Pick $w$ blocks $\{V^{1j_1}, \ldots, V^{bj_b}\}$ to form a stratum
6:     for $k = 1, \ldots, b$ do                                 // in parallel
7:       Run SGD on the training points in $V^{kj_k}$ with step size $\epsilon$
```

$j = S(k)$; here k corresponds to a processing unit such as a nodes or thread. In our example, we have $S_B(1) = 2$, $S_B(2) = 3$, and $S_B(3) = 1$.

The key point in SSGD is how to select strata such that all training points are sampled correctly and the convergence is guaranteed. The SSGD algorithm is summarized in Algorithm 2. SSGD repeatedly selects and processes a stratum $s \in \mathscr{S}$; the selection is based on a *stratum schedule* (see below). All blocks in the selected stratum are processed in parallel: processing unit k processes block $\boldsymbol{V}^{kS(k)}$. For example, assume that stratum S_C has been selected during the execution of SSGD; in this case, blocks $\boldsymbol{V}^{13}$, $\boldsymbol{V}^{21}$, and $\boldsymbol{V}^{32}$ are processed in parallel by nodes 1, 2, and 3, respectively. In what follows, we refer to processing a single stratum as a *subepoch* and to a sequence of b subepochs as an *epoch*. Note that an epoch roughly corresponds to one single pass over the training data: each block contains N/b^2 training points in expectation, each epoch consists of b subepochs, and we process b blocks in each subepoch.

Stratum schedule. Stratum schedule determines which strata are chosen in each subepoch. More specifically, it consists of a (possibly random) sequence $S_1, S_2, \ldots$ from$\mathscr{S}$; S_l is the stratum visited in the l-th subepoch. The convergence properties of SSGD are established in Gemulla et al. (2011a). In particular, it has been shown that SSGD asymptotically converges to a stationary point of L under natural conditions on the stratum schedule. For instance, a stratum schedule must guarantee that every training point is processed equally often in a long run; for details see Gemulla et al. (2011a). Stratum schedule affects the convergence properties of SSGD in practice.

In Gemulla et al. (2011a) three possible strategies for stratum selection have been examined:

- Sequential selection (SEQ),
- With replacement selection (WR), and
- Without replacement selection (WOR).

The simplest correct schedule is to select exactly b strata such that they jointly cover all the training data, and process them in a sequential order (SEQ); this selection strategy ensures that all training points are chosen exactly once in every epoch. For example, we can use the stratum schedule (S_A, S_B, S_C) for every epoch. This strategy is similar to a *cyclic partitioning* of the training data which is used in the Jellyfish algorithm (see Section 2.3.2B). An alternative schedule is to uniformly sample d strata from $\mathscr{S}$ with replacement in every subepoch (WR); e.g., the schedule (S_A, S_C, S_A) might be selected in a given subepoch. Finally, we may select strata randomly but ensure that every block is processed exactly once in every epoch (WOR). This strategy can be seen as selecting a schedule according to SEQ uniformly at random in every epoch; in our example, all possible permutations of (S_A, S_B, S_C) and all possible permutations of (S_D, S_E, S_F) are valid schedules; in each epoch, one of these 12 schedules is selected at random. Our experiments show that WOR outperforms the other strategies in terms of the number epochs to convergence. The reason is that selecting strata according WOR randomizes the order of blocks as much as possible while ensuring that all the training points are processed in every epoch.

The generic SSGD forms the basis of various algorithms for different settings. In the sequel, we first describe DSGD-MR (Gemulla et al. 2011c) designed for the shared-nothing MapReduce framework and then our novel DSGD++, an in-memory algorithm for shared-nothing architectures in which nodes or threads can communicate directly.

C. Distributed SGD-MapReduce (DSGD-MR)

DSGD-MR implements the SSGD algorithm in the MapReduce framework. In MapReduce, the data is partitioned and stored in a distributed file system and is loaded into memory for processing. In this framework, the computation is divided into a sequence of map and reduce phases; after each phase the data is written back into disk to avoid data loss if a compute node fails. Note that in MapReduce, the compute nodes do not communicate directly, but rather indirectly through reading and writing of files.

Consider a stratification of the data based on a $w \times w$ blocking of $\boldsymbol{V}$; each node k stores blocks $\boldsymbol{V}^{k1}, \ldots, \boldsymbol{V}^{kw}$ of the input matrix together with blocks $\boldsymbol{L}^k$ and $\boldsymbol{R}^k$ of the factor matrices. This memory layout is depicted in Figure 2.1d (as before, $\boldsymbol{V}^{k*}$ refers to the k-th row of blocks of $\boldsymbol{V}$). Note that our implementation of DSGD-MR differs from the disk-based Hadoop implementation proposed in Gemulla et al. (2011c) in that it stores the input data and the factor matrices in main memory instead of in a distributed file system. This modification avoids the I/O costs incurred by the disk-based Hadoop implementation; moreover, it facilitates comparison with other algorithms. To process a stratum, a single map-only job consisting of w map tasks is executed. To this end, the k-th map task requires access to blocks $\boldsymbol{L}^k$, $\boldsymbol{V}^{kS_l(k)}$, and $\boldsymbol{R}^{S_l(k)}$. Note that blocks $\boldsymbol{L}^k$ and $\boldsymbol{V}^{kS_l(k)}$ are node-local, whereas $\boldsymbol{R}^{S_l(k)}$ needs to be fetched from node $S_l(k)$ and stored back afterwards. Therefore, only blocks of $\boldsymbol{R}$ need to be communicated while processing a stratum.

D. DSGD++

The MapReduce environment has a number of limitations; for example, nodes communicate via a distributed file systems but cannot communicate directly, and the data is stored on disk and loaded into memory when required. Our novel DSGD++ is an in-memory algorithm that can run on a small cluster of commodity nodes. DSGD++ utilizes a novel data partitioning and stratum schedule, and exploits asynchronous communication, as well as direct memory access and multithreading. In the sequel, we discuss in detail various features of DSGD++, which allow us to improve on DSGD-MR.

Direct fetches. Denote by S_l the stratum used in subepoch l and by $S_l^{-1}(j)$ the node that updates block $\boldsymbol{R}^j$ in the l-th subepoch. While running subepoch l, DSGD++ moves the blocks of $\boldsymbol{R}$ directly between the nodes, i.e., the algorithm avoids writing back the results to disk as in DSGD-MR. More specifically, node k fetches the block $\boldsymbol{R}^{S_l(k)}$ directly from node $S_{l-1}^{-1}(S_l(k))$, which processed this block in the previous subepoch. A similar approach, but in the context of the Spark cluster-computing framework, has been explored in the Sparkler system (Li et al. 2013); this system focuses on issues orthogonal to those of this chapter, such as cluster elasticity, fault-tolerance, and ease of programming.

Overlapping subepochs. In DSGD++, node k starts processing block $\boldsymbol{V}^{kS_l(k)}$ as soon as $\boldsymbol{R}^{S_l(k)}$ has been received. As a result, this strategy allows for overlapping of subepochs. For example, assume that all nodes are working on stratum S_A as in Figure 2.2. In the meanwhile, nodes 1 and 2 finish their jobs, but node 3, which is slower, still operates on block $\boldsymbol{V}^{33}$. Instead of forcing both nodes 1 and 2 to wait for node 3 to finish, node 1 can immediately proceed to S_B and start working on block $\boldsymbol{V}^{12}$ (because node 2 has finished and can send $\boldsymbol{R}^2$ to node 1). However, node 2 needs to wait until node 3 has finished processing $\boldsymbol{V}^{23}$ before moving to stratum S_B, since it cannot start updating $\boldsymbol{R}^3$ until it receives $\boldsymbol{R}^3$ from node 3. Note that this protocol prevents inconsistent updates, e.g., on $\boldsymbol{R}^3$. In this way, DSGD++ gracefully handles varying processing speeds across the compute nodes.

Asynchronous communication. Observe that the subepochs in DSGD-MR are separated into two phases: (1) a communication phase, in which next blocks of $\boldsymbol{R}$ are communicated, and (2) a computation phase, in which the blocks of $\boldsymbol{V}$ are processed. DSGD++, on the contrary, overlays communication and computation by using a $w \times 2w$ blocking of $\boldsymbol{V}$ instead of a $w \times w$ blocking. In each epoch, DSGD++ conceptually partitions $\boldsymbol{V}$ and conformingly $\boldsymbol{R}$ at random into two matrices $\boldsymbol{V}_{\text{red}}$ and $\boldsymbol{V}_{\text{blue}}$, each consisting of w of the $2w$ column blocks. The algorithm then alternates between running a subepoch on $\boldsymbol{V}_{\text{red}}$ and a subepoch on $\boldsymbol{V}_{\text{blue}}$. As a result, the red and blue subepochs work on disjoint blocks of $\boldsymbol{R}$ (cf. Figure 2.1e). This approach enables us to overlay communication and computation as follows: Suppose that some node k runs subepoch l (say, blue). Node k now simultaneously fetches the block of $\boldsymbol{R}$ required in the $(l+1)$-th subepoch (red) from the node that processed it in the $(l-1)$-th subepoch (also red). Thus, communication and computation are overlaid.

Multithreading. Given w compute nodes and t threads per node, DSGD++ exploits thread-level parallelism by using a more fine-grained $p \times 2p$ blocking (rather than $w \times 2w$), where $p = wt$. Each node then stores $2tp$ blocks of $\boldsymbol{V}$, t blocks of $\boldsymbol{L}$, and $2t$ blocks of $\boldsymbol{R}$. This blocking allows us to process t blocks of $\boldsymbol{V}$ in t parallel threads during a subepoch. In contrast to using multiple independent processes per node, multithreading enables us to share memory between the threads. In more detail, when the block required in subepoch l (say, blue) has been processed at the same node in subepoch $l-2$ (also blue), no communication cost is incurred (*local fetch*). Data only needs to be communicated if the required block is stored on some other node (*remote fetch*).

Locality-aware scheduling. A consequence of the distinction between local and remote fetches is that different stratum schedules have different communication costs, depending on the (expected) number of remote fetches of the required blocks of $\boldsymbol{R}$; thus, the more the (expected) number of remote fetches, the higher the communication costs. Note that with respect to the communication costs, SEQ is significantly more efficient than WR or WOR. The reason is that in every subepoch only a single remote fetch occurs per node and the other $t-1$ fetches are all local. Nevertheless, the increased randomization of WOR leads to superior convergence properties.

DSGD++ utilizes a locality-aware scheduling (LA-WOR) that strikes a compromise between the compute-efficiency of SEQ and the desirable randomness of WOR.

The goal is to use a scheduling to maximize the number of local fetches while preserving randomness. The key idea is to apply the stratification technique twice: once at the node-level and once at the thread-level. In more details, we proceed as follows: After $\boldsymbol{V}_{\text{red}}$ and $\boldsymbol{V}_{\text{blue}}$ have been determined at the beginning of an epoch, randomly group the wt column blocks of $\boldsymbol{V}_{\text{red}}$ (and independently $\boldsymbol{V}_{\text{blue}}$) into w groups. Similarly, group the wt row blocks of $\boldsymbol{V}_{\text{red}}$ by the node at which they are stored. We thus obtain a $w \times w$ *coarse-grained blocking* of $\boldsymbol{V}_{\text{red}}$. Each of the coarse-grained blocks is then broken up into $t \times t$ *fine-grained blocks*. A node-level (resp. thread-level) stratification of the input and factor matrices is based on this coarse-grained (resp. fine-grained) blocking. To illustrate the main idea, assume that $w = t = 2$ and that $\boldsymbol{V}_{\text{red}}$ consists of column blocks $\boldsymbol{V}^{*1}, \dots, \boldsymbol{V}^{*4}$. Moreover, suppose that row blocks $\boldsymbol{V}^{1*}$ and $\boldsymbol{V}^{2*}$ (resp. $\boldsymbol{V}^{3*}$ and $\boldsymbol{V}^{4*}$) are stored at node 1 (resp. 2). Then one possible blocking is:

$$\begin{array}{c} \text{Node 1} \\ \\ \text{Node 2} \end{array} \begin{pmatrix} \begin{pmatrix} \boldsymbol{V}^{11} & \boldsymbol{V}^{14} \\ \boldsymbol{V}^{21} & \boldsymbol{V}^{24} \end{pmatrix} & \begin{pmatrix} \boldsymbol{V}^{12} & \boldsymbol{V}^{13} \\ \boldsymbol{V}^{22} & \boldsymbol{V}^{23} \end{pmatrix} \\ \begin{pmatrix} \boldsymbol{V}^{31} & \boldsymbol{V}^{34} \\ \boldsymbol{V}^{41} & \boldsymbol{V}^{44} \end{pmatrix} & \begin{pmatrix} \boldsymbol{V}^{32} & \boldsymbol{V}^{33} \\ \boldsymbol{V}^{42} & \boldsymbol{V}^{43} \end{pmatrix} \end{pmatrix}.$$

Note that in this example the column blocks $\boldsymbol{V}^{*1}$ and $\boldsymbol{V}^{*4}$ (resp. $\boldsymbol{V}^{*2}$ and $\boldsymbol{V}^{*3}$) have been grouped together. To process a red stratum (a blue stratum is processed similarly), two WOR schedules are used: (1) a WOR schedule over the coarse-grained blocking to distribute across nodes, and (2) a WOR schedule, which is chosen independently for each coarse-grained block, to parallelize across threads. In this way, whenever a coarse-grained block is processed at a node, the corresponding fine-grained blocks of $\boldsymbol{R}$ have to be communicated only once. Continuing the example, we might obtain the following LA-WOR schedule for $\boldsymbol{V}_{\text{red}}$:

$$\begin{array}{r} \\ \text{Node 1, thread 1} \\ \text{Node 1, thread 2} \\ \text{Node 2, thread 1} \\ \text{Node 2, thread 2} \end{array} \begin{array}{c} \begin{array}{cccc} S_1 & S_3 & S_5 & S_7 \end{array} \\ \left(\begin{array}{cc|cc} \boldsymbol{V}^{11} & \boldsymbol{V}^{14} & \boldsymbol{V}^{13} & \boldsymbol{V}^{12} \\ \boldsymbol{V}^{24} & \boldsymbol{V}^{21} & \boldsymbol{V}^{22} & \boldsymbol{V}^{23} \\ \hline \boldsymbol{V}^{32} & \boldsymbol{V}^{33} & \boldsymbol{V}^{31} & \boldsymbol{V}^{34} \\ \boldsymbol{V}^{43} & \boldsymbol{V}^{42} & \boldsymbol{V}^{44} & \boldsymbol{V}^{41} \end{array}\right) \end{array}.$$

Here the column entries for a stratum S_i correspond to the blocks of $\boldsymbol{V}$ that comprise the stratum, and the strata $S_1, S_3, \dots, S_7$ are displayed in the order that they are processed during a given sequence of red subepochs (hence the odd-numbered stratum indexes). As before, we only need to communicate blocks of $\boldsymbol{R}$. Assume that in this example blocks $\boldsymbol{R}^1$ and $\boldsymbol{R}^2$ are initially stored on node 1, while blocks

$\boldsymbol{R}^3$ and $\boldsymbol{R}^4$ are at node 2. Denote by thread(i, j) the j-th thread at node i. Stratum S_1 is processed first. To process $\boldsymbol{V}^{11}$, thread$(1, 1)$ fetches $\boldsymbol{R}^1$, a local fetch, whereas thread$(1, 2)$ fetches $\boldsymbol{R}^4$ to process $\boldsymbol{V}^{24}$, a remote fetch. Similarly, thread$(2, 1)$ and thread$(2, 2)$ need to fetch $\boldsymbol{R}^2$ and $\boldsymbol{R}^3$, resulting in one remote and one local fetch. Next, S_3 is processed. In this stratum, $\boldsymbol{R}^4$ and $\boldsymbol{R}^1$ are fetched by thread$(1, 1)$ and thread$(1, 2)$, while $\boldsymbol{R}^3$ and $\boldsymbol{R}^2$ are retrieved by thread$(2, 1)$ and thread$(2, 2)$, respectively. Because these blocks of $\boldsymbol{R}$ were all fetched during the processing of S_1, they are now local, so that no remote fetches are required. Overall, the processing of S_1–S_7 incurs 10 local fetches and only 6 remote fetches.

2.3.2 Shared-Memory Setting

We now review some existing parallel shared-memory SGD-based methods designed to run on a single powerful compute node. Shared-memory algorithms are suited for the case when both the data matrix and the factor matrices can be loaded in the memory of a single compute node. As before, t denotes the number of available threads on a single node.

A. Parallel SGD (PSGD)

One natural approach to parallelize SGD across multiple threads is to partition the training point schedule evenly among available t threads; each thread then performs N/t SGD steps per epoch. To prevent the overwriting of updates, each thread locks row $\boldsymbol{L}_{i*}$ and column $\boldsymbol{R}_{*j}$ before processing training point (i, j). This approach is effective when the number of available threads t is small (say, $t \leq 8$). However, the overhead of locking and random memory access becomes substantial for larger number of threads. Recht et al. (2011) proposed a lock-free approach in which SGD steps are performed without locking and inconsistent updates are allowed. We refer to this lock-free variant of parallel SGD as PSGD. Since the number of rows and columns are usually significantly larger than the available threads, it is unlikely that two threads process training points that share a common row or column at the same time. For matrix completion tasks, Recht et al. found virtually no difference between the parallel SGD with and without locking in terms of running time and quality.

B. Jellyfish

Recht and Ré (2013) presented a parallel SGD algorithm, termed Jellyfish. Similar to SSGD, the main idea of Jellyfish is to partition the data into interchangeable blocks that can be processed independently in parallel. In particular, it makes use of a cyclic partitioning of the training data which enables performing SGD steps in parallel without obtaining locks. This partitioning scheme is precisely the same as the SEQ stratum schedule in SSGD; it ensures that multiple threads operate on data points (i.e., blocks) that do not share a common row or column. Jellyfish uses a $t \times t$ blocking of the input matrix; the factor matrices are blocked accordingly. However, in contrast to SSGD, Jellyfish changes the blocking of $\boldsymbol{V}$ in every epoch by reshuffling the entire data points. This reshuffling step randomizes the order of the data points to the extent possible for faster convergence. After this phase, the data is accessed and SGD steps are performed sequentially. For improved efficiency, Jellyfish overlaps the gradient computations in a given epoch with the reshuffling of the data points. To this end, the algorithm maintains s copies of the data, where $s \geq 2$ is a small number. While one copy is being processed, $s - 1$ parallel threads reorder the data in the remaining $s - 1$ copies. Similar to SSGD, Jellyfish operates on a copy of data using t parallel threads. Note that Jellyfish has high memory requirements, since multiple copies of the input matrix need to be stored. Moreover, parallel reorganizations of the data may also lead to memory bottlenecks.

C. Cache-Conscious Parallel SGD (CSGD)

Recall from Section 2.3.2A that PSGD conducts updates to the training points in a random order. Thus, a main drawback of PSGD is that its memory access pattern is highly discontinuous. This in turn results in a high cache-miss rate and performance degradation. This effect is even more emphasized when N is large. Makari et al. (2014) proposed a variant of SSGD, termed CSGD, designed for a shared-memory setting. The key principle in CSGD is to deploy a fine-grained blocking of the input matrix so that each block $\boldsymbol{V}^{ij}$ and the corresponding factor matrices $\boldsymbol{L}^{i}$ and $\boldsymbol{R}^{j}$ fit into the L2 cache of a single core. To enhance better memory locality, the training points within a block are laid out in consecutive memory locations. More specifically, CSGD partitions matrix $\boldsymbol{V}$ evenly across available threads such that each row block is assigned to exactly one single thread. Thus, no inconsistent updates will be performed on row-factor matrix $\boldsymbol{L}$. Note that overwriting of updates to column-factor matrix $\boldsymbol{R}$ may still occur. However, since the number of blocks is much larger than the available threads, i.e., $b \gg t$, we expect this to happen rarely.

D. Fast Parallel SGD (FPSGD)

Recently, Zhuang et al. (2013) has presented an SGD-based algorithm, called FPSGD, tailored to a shared-memory environment. To improve upon PSGD, the key ideas of FPSGD are (1) to improve the performance by keeping the available threads busy and (2) to alleviate the discontinuous memory-access pattern of PSGD. More specifically, FPSGD uses a $b \times b$ blocking of input matrix $\boldsymbol{V}$ where $b \geq t + 1$; the corresponding factor matrices are blocked accordingly. At a given point of time, a block is called *free* if it is interchangeable with all blocks being processed. Otherwise, it is a *non-free* block. In the course of the algorithm, a scheduler assigns a randomly chosen block which has been processed least frequently from the set of all free blocks. Similar to CSGD, FPSGD stores the training points within a block in consecutive memory locations such that the cache-miss rates are minimized. A comparison between FPSGD and CSGD in an experimental study remains for future work.

2.4 Matrix Completion via Alternating Minimizations

We now discuss some common algorithms for large-scale matrix completion based on the techniques of "alternating minimizations". For each method, we first focus on a sequential setting and then continue with parallel shared-memory and shared-nothing variants.

2.4.1 Alternating Least Squares (ALS)

Note that our formulation of matrix completion according to (2.1) is a non-convex problem for all loss functions from Table 2.2. However, when fixing one of the factor matrices $\boldsymbol{L}$ or $\boldsymbol{R}$, it becomes a least-squares problem with a globally optimal solution. ALS is a common approach to solve such quadratic problems. The basic idea is to repeatedly keep one of the unknown matrices ($\boldsymbol{L}$ or $\boldsymbol{R}$) fixed, so that the other one can be optimally recomputed. ALS then alternates between recomputing the rows of $\boldsymbol{L}$ in one step and the columns of $\boldsymbol{R}$ in the subsequent step. For the basic case of L_{Sl} this amounts to solving a set of least-squares subproblems: one for each row of $\boldsymbol{L}$ and one for each column of $\boldsymbol{R}$

$$\text{Compute } \boldsymbol{L}_{n+1}\text{: } (\forall i)\ \underline{\boldsymbol{L}_{i*}}\boldsymbol{R}_n^{(i)} = \boldsymbol{V}_{i*}, \tag{2.2}$$

$$\text{Compute } \boldsymbol{R}_{n+1}\text{: } (\forall j)\ \boldsymbol{L}_{n+1}^{(j)}\underline{\boldsymbol{R}_{*j}} = \boldsymbol{V}_{*j}, \tag{2.3}$$

where the unknown variable is underlined, $\boldsymbol{V}_{i*}$ (resp. $\boldsymbol{V}_{*j}$) denotes the revealed entries in row i (column j), and $\boldsymbol{R}_n^{(i)}$ (resp. $\boldsymbol{L}_{n+1}^{(j)}$) refers to the corresponding columns of $\boldsymbol{R}_n$ (rows of $\boldsymbol{L}_{n+1}$). These equations can be solved using a method of choice; for example, for the basic squared loss we obtain the closed from solutions

$$\boldsymbol{L}_{n+1,i*}^{\top} \leftarrow (\boldsymbol{R}_n^{(i)}[\boldsymbol{R}_n^{(i)}]^{\top})^{-1}\boldsymbol{R}_n\boldsymbol{V}_{i*}^{\top},$$

$$\boldsymbol{R}_{n+1,*j} \leftarrow ([\boldsymbol{L}_{n+1}^{(j)}]^{\top}\boldsymbol{L}_{n+1}^{(j)})^{-1}\boldsymbol{L}_{n+1}^{\top}\boldsymbol{V}_{*j},$$

where $\boldsymbol{L}_{n+1,i*}$ (resp. $\boldsymbol{R}_{n+1,*j}$) denotes the i-th row of $\boldsymbol{L}_{n+1}$ (resp. j-th column of $\boldsymbol{R}_{n+1}$). Note that during the computation of the least-squares solutions, matrix $\boldsymbol{V}$ needs to be accessed once by row when updating $\boldsymbol{L}$ (Equation (2.2)), and once by column when updating $\boldsymbol{R}$ (Equation (2.3)). Therefore, ALS implementations require two copies of the data matrix $\boldsymbol{V}$: one in row-major order (denoted by $\boldsymbol{V}_{\text{r}}$) and one in column-major order (denoted by $\boldsymbol{V}_{\text{c}}$). Loss functions L_{L2} and

L_{L2w} can also be handled as in Zhou et al. (2008). For solving each least-squares problem, ALS requires $O(Nr^2)$ time to form all the $r \times r$ matrices $\boldsymbol{R}_n^{(i)}[\boldsymbol{R}_n^{(i)}]^\top$ and $[\boldsymbol{L}_{n+1}^{(j)}]^\top \boldsymbol{L}_{n+1}^{(j)}$, and additional $O(r^3)$ time to solve the least-squares problem. Thus, under our running assumption that $m \geq n$, an ALS epoch takes $O(Nr^2 + mr^3)$.

Despite the wide practical applicability of ALS, its convergence properties is less understood. Recently, there has been some work on the analysis of the ALS method for matrix completion. For instance, Keshavan (2012) and Jain et al. (2013) studied theoretical guarantees of ALS on the global optimality. In particular, they proved geometric convergence of ALS when some natural conditions are imposed on the problems; for details see Keshavan (2012) and Jain et al. (2013).

A. Parallel ALS (PALS)

Parallelization of ALS underlies the observation that the involved least-squares subproblems can be solved independently (Zhou et al. 2008). In particular, an update to a row (resp. column) of $\boldsymbol{L}$ (resp. $\boldsymbol{R}$) does not affect other rows (resp. columns) of $\boldsymbol{L}$ (resp. $\boldsymbol{R}$). Thus, PALS partitions the rows of $\boldsymbol{L}$ uniformly among available threads; each partition is processed in parallel by its corresponding thread. The columns of $\boldsymbol{R}$ are processed in a similar way.

B. Distributed ALS (DALS)

Following Zhou et al. (2008), we extend PALS to a distributed shared-nothing setting. The basic assumption is that each node has enough memory to store $2/w$ of the entries of $\boldsymbol{V}$, together with a full copy of the factor matrices $\boldsymbol{L}$ and $\boldsymbol{R}$. DALS uses a $w \times 1$ blocking for $\boldsymbol{V}_\text{r}$ and a $1 \times w$ blocking for $\boldsymbol{V}_\text{c}$; the factor matrices are blocked accordingly. Each node k stores the following blocks: $\boldsymbol{V}_\text{r}^{k*}$, $\boldsymbol{V}_\text{c}^{*k}$, $\boldsymbol{L}^k$, and $\boldsymbol{R}^k$ ($\boldsymbol{V}_\text{r}^{k*}$ refers to the k-th row block of $\boldsymbol{V}_\text{r}$, $\boldsymbol{V}_\text{c}^{*k}$ refers to the k-th column block of $\boldsymbol{V}_\text{c}$). This memory layout is depicted in Figure 2.1a. In DALS, each node k updates its corresponding blocks $\boldsymbol{V}_\text{r}^{k*}$ and $\boldsymbol{V}_\text{c}^{*k}$ alternately. While updating $\boldsymbol{V}_\text{r}^{k*}$ at node k, block $\boldsymbol{V}_\text{r}^{k*}$ and the *entire* matrix $\boldsymbol{R}$ need to be accessed. Note that $\boldsymbol{V}_\text{r}^{k*}$ and $\boldsymbol{L}^k$ are stored locally at node k whereas $\boldsymbol{R}$ is not. Therefore, DALS broadcasts blocks $\boldsymbol{R}^1, \ldots, \boldsymbol{R}^k$ to create a local copy of the entire $\boldsymbol{R}$. The DALS algorithm proposed in Zhou et al. (2008) uses multiple processes (each with its own address space) on each node. In contrast, our DALS implementation utilizes the available threads (which share the same memory space and variables) on each node. This modification allows DALS to exploit shared-memory and reduce the memory footprint significantly.

2.4.2 Cyclic Coordinate Descent (CCD++)

Cyclic coordinate descent (CCD) is a well-known optimization technique (Bertsekas 1999; Section 2.7) which has been shown to be effective for numerous large-scale problems (Cichocki and Phan 2009; Hsieh et al. 2008; Hsieh and Dhillon 2011; Hsieh et al. 2011; Yu et al. 2011). It can be considered as an alternating minimization approach in that it optimizes a *single* entry of the factor matrices at a time while keeping all other entries fixed. Therefore, the optimization subproblems are significantly simpler than the least-squares problems of ALS and each single variable update can be performed more efficiently than in ALS. Practical variants of CCD adapt the approach of "hierarchical" ALS (Cichocki and Phan 2009): they do not operate on the original input matrix but instead maintain and process the residual matrix $\boldsymbol{E}$ with entries $\boldsymbol{E}_{ij} = \boldsymbol{V}_{ij} - [\boldsymbol{L}\boldsymbol{R}]_{ij}$ for $(i,j) \in \Omega$. This modification improves the time required for a single variable update.

Recently, Yu et al. (2012) has proposed a variant of CCD, termed CCD++, for matrix completion tasks. The memory requirement of CCD++ is similar to that of ALS. That is two copies of the residual matrix $\boldsymbol{E}$ need to be stored: one in row-major order, denoted $\boldsymbol{E}_{\mathrm{r}}$, and one in column-major order, denoted $\boldsymbol{E}_{\mathrm{c}}$. The key idea of CCD++ is to perform a feature-wise sequence of updates, i.e., to loop over all features $f \in [1, r]$. For each feature f, CCD++ performs T updates to the corresponding feature-vector of $\boldsymbol{L}$ (i.e., $\boldsymbol{L}_{*f}$), where T is an automatically tuned parameter of the algorithm which is independent of the data size. The algorithm then updates the corresponding feature-vector of $\boldsymbol{R}$ (i.e., $\boldsymbol{R}_{f*}$) T times. Next, both copies of the residual matrix are updated and the algorithm continues with feature $f+1$. Note that for each feature, the residual matrix is scanned $2T$ times during the updates to matrices $\boldsymbol{L}$ and $\boldsymbol{R}$, and twice to update $\boldsymbol{E}_{\mathrm{r}}$ and $\boldsymbol{E}_{\mathrm{c}}$. An iteration of CCD++, i.e., the processing of all features, thus consists of $2r(T+1)$ epochs and requires $O(TNr)$ time in total. Overall, our experiments as well as results in Yu et al. (2012) show that, CCD++ is computationally less expensive than ALS and can handle larger ranks more efficiently.

A. Parallel CCD++ (PCCD++)

Parallelizing CCD++ follows the same ideas as for ALS: similar partitioning scheme for the input matrix and the factor matrices is used; here again the involved subproblems are independent and can be optimized in parallel (Yu et al. 2012). We refer to the parallel shared-memory variant of CCD++ as PCCD++.

B. Distributed CCD++ (DCCD++)

Distributed shared-nothing variant of CCD++ (Yu et al. 2012), termed DCCD++, utilizes the same partitioning scheme and data distribution as for DALS. The memory layout of DCCD++ is illustrated in Figure 2.1b. The only difference between DALS and DCCD++ lies in the communication of data: DCCD++ broadcasts only a single feature-vector instead of the entire factor matrix as in DALS. However, the communication is performed every time a feature-vector is being updated, i.e., $2Tr$ times per iteration. Therefore, the synchronization costs between the nodes can potentially be substantial.

2.5 Alternating Minimizations Versus SGD

In this section, we provide a comprehensive comparison of the algorithms based on alternating minimizations with the SGD-based approaches first in terms of a complexity analysis and next through an extensive experimental evaluation. Table 2.3 summarizes various shared-nothing algorithms discussed in this chapter. Before continuing with the comparison, we make the following remarks. Since ALS involves solving a large number of least-squares problems, it is computationally much more expensive than SGD. When the rank of the factorization is sufficiently small (say, $r \leq 50$), this computational overhead is however acceptable. An advantage of both ALS and CCD++ over SGD is that the former methods are parameter-free, whereas SGD methods make use of a step size sequence. In fact, the convergence properties of SGD is significantly influenced by the choice of step size sequence. Our experiments suggest, however, that SGD is the method of choice when the step size sequence is chosen judiciously, e.g., using the bold driver method of Section 2.2.2. In terms of memory consumption, both ALS and CCD++ are inferior to SGD since they need to store two copies of the data matrix. Finally, SGD-based methods apply to a wide range of loss functions, whereas ALS and CCD++ target quadratic loss functions.

2.5.1 Complexity Analysis

In this section, we compare various shared-nothing matrix completion algorithms via a complexity analysis; we do not theoretically analyze shared-memory methods, as the results heavily rely on the assumptions about the memory architecture and cache behavior, which can vary widely. Afterwards in Section 2.5.2, empirical results are presented.

The main purpose of this section is to identify conditions under which distributed processing is effective. To simplify the analysis, we make the following assumptions:

Table 2.3: *Overview of shared-nothing methods (see Table 2.1 for notation)*

Method	Partitioning	Memory consumption/node	Epochs/it.	Time/it.	Communication/node/it.
DALS (Zhou et al. 2008)	$\boldsymbol{V}$ & $\boldsymbol{L}$ by rows $\boldsymbol{V}$ & $\boldsymbol{R}$ by columns	$2\boldsymbol{V}/w + \boldsymbol{L} + \boldsymbol{R}$	2	$O(p^{-1}[Nr^2 + (m+n)r^3])$	$O([m+n]r)$
DCCD++ (Yu et al. 2012)	$\boldsymbol{E}$ & $\boldsymbol{L}$ by rows $\boldsymbol{E}$ & $\boldsymbol{R}$ by columns	$(2\boldsymbol{E} + \boldsymbol{L} + \boldsymbol{R})/w$	$2r(T+1)$	$O(p^{-1}TNr)$	$O(T[m+n]r)$
ASGD (new)	$\boldsymbol{V}$ & $\boldsymbol{L}$ by rows	$(\boldsymbol{V} + \boldsymbol{L} + \boldsymbol{R})/w + 2\boldsymbol{R}$	1	$O(p^{-1}Nr + nr)^*$	$O(nr)^*$
DSGD-MR (Gemulla et al. 2011c)	$\boldsymbol{V}$ blocked, $\boldsymbol{L}$ by row, $\boldsymbol{R}$ by column	$(\boldsymbol{V} + \boldsymbol{L} + 2\boldsymbol{R})/w$	1	$O(p^{-1}Nr)$	$O(nr)$
DSGD++ (new)	$\boldsymbol{V}$ blocked rect., $\boldsymbol{L}$ by row, $\boldsymbol{R}$ by column	$(\boldsymbol{V} + \boldsymbol{L} + 1.5\boldsymbol{R})/w$	1	$O(p^{-1}Nr)$	$O(nr)$

*Assuming asynchronous averaging is performed a constant number of times per iteration.

1. computation and communication are not overlaid,
2. each revealed entry of $\boldsymbol{V}$ and each entry of $\boldsymbol{L}$ and $\boldsymbol{R}$ requires $O(1)$ words of memory and $O(1)$ time to communicate,
3. each compute node possesses a constant number t of threads,
4. $r, n \leq m$, and
5. $N = O(mr)$.

In what follows, we give bounds on the memory consumption per node as well as the time required for computation and communication per node and per epoch.

Memory consumption per node. In DALS, each node stores in memory two partitions of the input matrix $\boldsymbol{V}$ (one partition from $\boldsymbol{V}_r$ and one from $\boldsymbol{V}_c$) as well as both factor matrices $\boldsymbol{L}$ and $\boldsymbol{R}$; the overall memory requirement is therefore $O(N/w + mr)$ words. Note that DALS also maintains a row-partition of $\boldsymbol{L}$ and a column-partition of $\boldsymbol{R}$, however, these two partitions are negligible compared to the other data. DCCD++ stores the following data: two partitions of the residual matrix (i.e., $\boldsymbol{E}_{\mathrm{r}}$ and $\boldsymbol{E}_{\mathrm{c}}$), two feature vectors (i.e., a row of $\boldsymbol{R}$ and a column of $\boldsymbol{L}$), a row-partition of $\boldsymbol{L}$, and a column-partition of $\boldsymbol{R}$. Thus, the total memory consumption is $O((N + mr)/w + m)$ words. ASGD maintains a partition of the input matrix together with the corresponding partitions of $\boldsymbol{L}$ and $\boldsymbol{R}$ as well as the smaller factor matrix $\boldsymbol{R}$ entirely; its memory requirement is thus $O((N + mr)/w + nr)$ words. Finally, DSGD++ fully partitions the factor matrices and thus requires $O((N + mr)/w)$ words in total. We conclude that DSGD++ is most memory-efficient, followed by DCCD++ and ASGD, and then DALS.

Computation/communication trade-off. A crucial factor which affects the performance of shared-nothing methods is the relationship between computation and communication costs. We say that a distributed-processing algorithm is *effective* if the computation costs dominate the communication costs. The reason is that computation costs are linearly reduced by distributed processing, while communication costs are increased. Under our assumptions, we can derive the following computation and communication costs for the algorithms under consideration. DALS requires $O(mr^3/w)$ time for computation and $O(mr)$ time for communication per epoch. As $m, r, w \to \infty$, computation dominates communication if the rank of the factorization is sufficiently large: $r^2 = \omega(w)$. In practice, we often have $r > w$ so that we expect DALS to be effective. Analogously, DCCD++ has computation cost $O(mr/w)$ and communication cost $O(m)$ for updating and communicating a single factor. Thus, DCCD++ is effective when $r = \omega(w)$, i.e., under tighter conditions than DALS.

For DSGD-MR, the time required for computation and communication is $O(Nr/w)$ and $O(nr)$, respectively. The latter cost is due to the fact that DSGD-MR only communicates factor matrix $\boldsymbol{R}$. Denote by $\bar{N} = N/n$ the average number of revealed entries per column of $\boldsymbol{V}$; $\bar{N}$ measures the amount of work per column and reflects how well we can parallelize SGD. To see this, rewrite the computational cost as $O(\bar{N}nr/w)$ and observe that computation dominates communication only for large values of $\bar{N}$, i.e., $\bar{N} = \omega(w)$. Thus, we expect DSGD-MR to be effective when the data matrix is not too sparse or has few columns. The same conclusions hold for DSGD++ and ASGD, even though they do not satisfy Assumption 1 above. Indeed, the analysis for DSGD-MR carries over to DSGD++ directly and to ASGD under the additional assumption that working copies are averaged at least once per epoch.

Theoretical analysis and actual performance. The asymptotic analysis presented above may not fully conform to the actual performance of the algorithms. The reason is that our theoretical analysis is based on some assumptions which may not be true in practice.

The first cause of concern is that the data size and the number of available nodes are in fact finite. Our experiments confirm our theoretical analysis even in the case where the data is small enough to fit on a single machine. In particular, we observed that DALS is effective and considerably outperformed PALS for problems of a given size. Moreover, the relative time for a DSGD-MR or ASGD epoch compared to that of a PSGD epoch depends on $\bar{N}$. For DSGD++, however, our theoretical analysis diverged somewhat from our empirical results in that the algorithm proved to be effective even for small values of $\bar{N}$. The reason is that—unlike the assumption that we used for the complexity analysis—computation and communication are in fact overlaid.

Another potential inaccuracy in our complexity analysis is that distributed processing may affect the convergence behavior of the algorithms. This issue is not relevant for algorithms based on alternating minimizations: DALS and PALS, as well as DCCD++ and PCCD++, perform identically, as they solve exactly the same least-squares problems. On the contrary, DSGD-MR, DSGD++, and ASGD may require more epochs to converge compared to PSGD. The reasons are that the stratification used in DSGD-MR and DSGD++ reduces the randomness in training point selection, and asynchronous averaging used in ASGD introduces delay in broadcasting updates. As a consequence, the processing time per epoch itself is not an accurate indicator of the overall performance; convergence rates must also be taken into account. The following section contains an extensive set of experiments providing further insight into the performance of the algorithms.

2.5.2 Experimental Evaluation

We compared all shared-nothing algorithms in an extensive experimental study along the following dimensions: the time per epoch (excluding loss computations), the number of epochs until convergence, and the total time to convergence (including loss computations). Recall that an epoch corresponds roughly to one single pass over the input data. Thus, the number of epochs reflects the number of data scans. When comparing two algorithms A and B, we say that A is more *compute-efficient* than B if it requires less time per epoch, more *data-efficient* than B if it needs less epoch to converge (and thus less scans of the input data), and *faster* if it requires less total time.

A. Overview of Results

On our large-scale experiments and in all configurations, DSGD++ was the best-performing method. It was up to 12.8x faster than DALS, up to 5x faster than DSGD-MR, and up to 12.3x faster than ASGD. Indeed, DSGD++ even outperformed PSGD, by a large margin, in some of our experiments with data of moderate size for which both algorithms were applicable. DSGD++ was the only shared-nothing method that outperformed PSGD, which validates its high communication efficiency. ASGD was faster than DSGD++ in two experiments with few nodes and large $\bar{N}$, but its performance degraded as more nodes were added. Thus, ASGD and to a lesser extent DSGD-MR, were much more sensitive to communication overhead than DSGD++ and DALS. Finally, as with PALS, DALS was the most data-efficient but the least compute-efficient method. In terms of total time, it was competitive only when the rank r was small.

B. Experimental Setup

Implementation. We implemented SGD, SGD++, ALS, PSGD, PALS, DSGD-MR, DSGD++, ASGD, and DALS in C++. For communication, all shared-nothing algorithms used the MPICH2 implementation of MPI.[6] We used the GNU Scientific Library (2013) (GSL) for solving the least-squares problems of ALS; in our experiments, GSL was significantly faster than LAPACK (2012) and, in contrast to LAPACK, also supports multithreading.

Hardware. We ran our experiments on a 16-node compute cluster; each node had 48GB of main memory and was equipped with an Intel Xeon 2.40GHz processor with 8 cores.

[6] http://www.mcs.anl.gov/mpi/mpich/

Table 2.4: *Summary of datasets*

	m	n	N	$\bar{N}$	Size	L	λ
Netflix	480k	18k	99M	5.5k	2.2GB	L_{L2w}	0.05
KDD	1M	625k	253M	0.4k	5.6GB	L_{L2w}	1
Syn1B-rect	10M	1M	1B	1k	22.3GB	L_{Sl}	-
Syn1B-sq	3.4M	3M	1B	0.3k	22.3GB	L_{Sl}	-
Syn10B	10M	1M	10B	10k	223.5GB	L_{Sl}	-

Real-world datasets. We used two real-world datasets: Netflix and KDD. The Netflix dataset (Bennett and Lanning 2007) consists of roughly 99M ratings (ranging from 1 to 5) of 480k Netflix users for 18k movies; it occupies 2.2GB to store in main memory. The KDD dataset of Track 1 of KDD-Cup 2011 (Dror et al. 2011) consists of approximately 253M ratings of 1M Yahoo! Music users for 625k musical pieces; it occupies 5.5GB of main memory. Detailed statistics of these datasets, as well as for synthetic datasets described below, are summarized in Table 2.4. Netflix and KDD differ significantly in the value of $\bar{N}$ (large for Netflix, small for KDD). For both real-world datasets, we used the official validation sets and focused on L_{L2w} because it performs best in practice (Chen et al. 2012; Koren et al. 2009; Zhou et al. 2008). We did not tune the regularization parameter for varying choices of rank r but used the values given in Table 2.4 throughout.

Synthetic datasets. For our large-scale experiments, we generated three synthetic datasets that differ in the choice of m, n, and N. We generated each dataset by first creating two rank-50 matrices $\boldsymbol{L}^*_{m\times 50}$ and $\boldsymbol{R}^*_{50\times n}$ with entries sampled independently from the Normal$(0, 10)$ distribution. We then obtained the data matrix by sampling N random entries from $\boldsymbol{L}^*\boldsymbol{R}^*$ and adding Normal$(0, 1)$ noise. Note that the resulting datasets are very structured. We use them here to test the scalability of the various algorithms; the matrices can potentially be factored much more efficiently by exploiting their structure directly. To judge the impact of the shape of the data matrix, we generated two large datasets with 1B revealed entries and identical sparsity: Syn1B-rect is a tall rectangular matrix (high $\bar{N}$, easier to distribute), Syn1B-sq is a square matrix (low $\bar{N}$, harder to distribute). Note that we need to learn more parameters to complete Syn1B-rect (550M) than to complete Syn1B-sq (320M). We also generated a very large dataset with 10B entries (Syn10B) to explore the scalability of each method; Syn10B is significantly larger than the main memory of each individual machine.

Methodology. For all datasets, we centered the input matrix around its mean. To investigate the impact of the factorization rank, we experimented with ranks $r = 50$ and $r = 100$; in practice, values of up to $r = 1000$ can be beneficial (Zhou et al. 2008). The starting points $\boldsymbol{L}_0$ and $\boldsymbol{R}_0$ were chosen by taking i.i.d. samples from the Uniform$(-0.5, 0.5)$ distribution; the same starting point was used for each algorithm to ensure a fair comparison. For all SGD-based algorithms, we selected the initial step size based on a small sample of the data (1M entries): 0.0125 for Netflix ($r = 50$), 0.025 for Netflix ($r = 100$), 0.00125 for KDD ($r = 50$, $r = 100$) and 0.000625 for Syn1B and Syn10B. Throughout, we used the bold driver heuristic for step size selection, which was fully automatic. We used the WOR training point schedule and the WOR stratum schedule throughout our experiments unless stated otherwise, and ran a truncated version of SGD that clipped the entries in the factor matrices to $[-100, 100]$ after every SGD step. Also, unless stated otherwise, all SGD-based algorithms make use of prefetching as in the SGD++ algorithm of Section 2.2.2. For each algorithm, we declared convergence as soon as it reached a point within 2% of the overall best solution.

We mainly focus on experimental results for the shared-nothing setting and briefly present the results of sequential and shared-memory algorithms.

C. Sequential and Parallel Algorithms

For the sequential and shared-memory settings, we used the real (Netflix, KDD) datasets. We start with a brief discussion of sequential algorithms, which form a baseline for shared-memory and shared-nothing methods.

SGD step size sequence (Table 2.5). In this experiment, we compared the performance of different step size sequences for SGD on Netflix for $r = 50$. In Table 2.5, Standard(α) refers to s sequence of form $\frac{\epsilon_0}{n^\alpha}$, where n denotes the epoch and α is a parameter that controls the rate of decay; such sequences are commonly used in stochastic optimization. For this experiment only, we declared SGD as converged if the improvement in loss after one epoch falls below 0.1%. For SGD with the bold driver heuristic, we checked for convergence only in epochs following after a drop in the step size. We observed that the bold driver heuristic greatly outperformed other standard step size sequences, even though it does not guarantee asymptotic convergence. On Netflix, all step size sequences converged in roughly the same number of epochs, but the bold driver heuristic converged to a significantly better solution.

SGD, SGD++, ALS. We found that SGD++ is up to 13% more compute-efficient than the plain SGD (7.4 vs. 8.4min for KDD, $r = 100$); thus, prefetching is beneficial. Compared to ALS, SGD++ required up to 5x more epochs to converge (31 epochs for SGD++ vs. 6 epoch for ALS on Netflix, $r = 50$). However, SGD++ is more compute-efficient so that it was overall faster. This effect is strongest when r is high as SGD++ is less sensitive to the factorization rank; e.g., SGD++ was

Table 2.5: *SGD step size sequence (Netflix, $r = 50$)*

	Bold driver	Standard(1)	Standard(0.6)
Epochs	40	36	42
Loss (x10^7)	**7.936**	9.267	8.469

Table 2.6: *Impact of stratum schedules on DSGD++ (2x8)*

	Netflix, $r = 100$			KDD, $r = 100$		
	SEQ	WOR	LA-WOR	SEQ	WOR	LA-WOR
Time/ep. (s)	**10.47**	11.5	10.61	**32.13**	40.8	32.62
Epochs	200	**65**	106	88	**62**	69
Total time (s)	2426	**861**	1300	3750	3182	**2976**

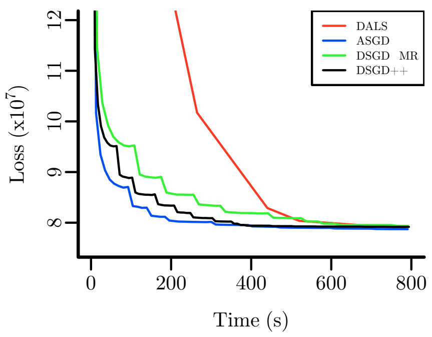
$\approx$5.5x faster than ALS on Netflix, $r = 100$ (52.8 vs. 290min), but only $\approx$1.8x faster for $r = 50$ (48 vs. 85min).

PSGD, PALS. We compared PSGD and PALS on real datasets using a single node with 8 threads (i.e., $w = 1$ and $t = 8$). On Netflix, PALS converged to a slightly better solution (1% lower loss) than PSGD. On KDD, however, both methods led to almost identical overall loss. On both Netflix and KDD, PALS was 7–8x faster than its sequential counterpart and equally data-efficient. Similarly, PSGD was 5x–6.8x faster than SGD; the speedups were less than for ALS since memory latency became a bottleneck. On the KDD data (where the larger number of rows and columns slowed down ALS), PSGD converged significantly faster (e.g., 21 vs. 162min, $r = 100$).

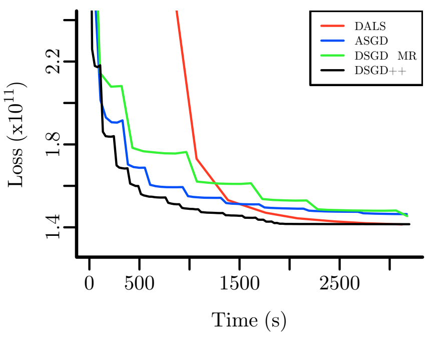
D. Shared-Nothing Algorithms

For the shared-nothing setting, we experimented with both real (Netflix, KDD) and synthetic datasets and used between 2 and 16 compute nodes with 8 threads each. We write $w \times t$ to refer to a setup with w compute nodes, each running t threads.

DSGD++ stratum schedule (Table 2.6). Recall that the DSGD++ stratum schedule affects both the time per epoch (governed by the relative number of local and remote fetches) and the number of epochs to convergence (governed by the degree of randomization). In Table 2.6, we compare the performance of DSGD++ with the SEQ, WOR, and LA-WOR schedules for the 2x8 setup. First, observe that WOR is more data-efficient than LA-WOR, which in turn is more data-efficient than SEQ. As with the SGD training point schedule, more randomness leads to better data-efficiency. Regarding compute-efficiency, we found that all three approaches



(a) *Netflix on 4x8*



(b) *KDD on 4x8*

Figure 2.3: *Performance of shared-nothing algorithms on real-world datasets,* $r = 100$

performed similarly on Netflix, because $\bar{N}$ is large and thus communication costs are relatively small. In such a setting, WOR is the method of choice. On KDD, where $\bar{N}$ is small so that communication becomes significant, LA-WOR outperformed both WOR and SEQ. These results are in accordance with the analysis in Section 2.5.1, and we therefore recommend the WOR schedule for datasets with large $\bar{N}$ and LA-WOR for those with small $\bar{N}$.

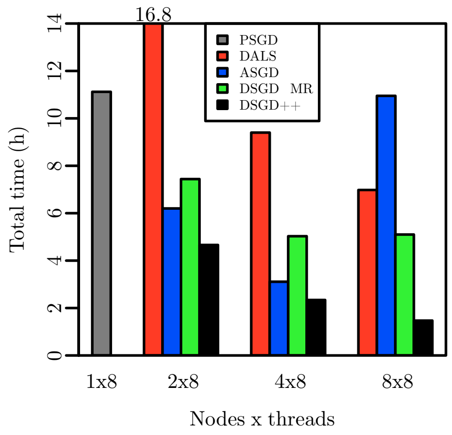
Netflix (Figure 2.3a). Even though the Netflix dataset is only moderately large, we found that distributed processing can speed up the factorization significantly; e.g., DALS achieved 22x–29x speedup on 4x8 (290 vs. 10min for $r = 100$). First, we observed that DSGD++ was up to 2.3x faster than DSGD-MR and ≈3x (4.6x–6.1x) more compute-efficient than PSGD on 2x8 (4x8). These superlinear speedups arise since DSGD++ exhibits better cache utilization. However, DSGD++ was less data-efficient than PSGD so that DSGD++ was only 1.7x–2.9x faster on 4x8 overall. Second, ASGD was less compute-efficient than DSGD++ (12.6 vs. 5.9s for $r = 50$); this is due to the overhead of averaging copies. The data-efficiency of ASGD dropped drastically as we increased the number of nodes (25 epochs on 2x8 vs. 60 epochs on 4x8, $r = 50$). The reason is that more working copies need to be averaged during an epoch when we use more nodes; as a result, the delay of propagating updates increases. For example, we performed on average 65 rounds of averaging per epoch on 2x8, but only 15 rounds on 4x8 (less time per epoch and at the same time more nodes to synchronize). Nevertheless, ASGD outperformed all other methods on 2x8 for $r = 100$. Finally, DALS was the fastest method for $r = 50$ but was outperformed by DSGD++ for higher ranks (5.8 vs. 10.7min on 4x8 for $r = 100$).

KDD (Figure 2.3b). For the KDD dataset, DALS provided similar speedups as for Netflix and was the overall fastest method for $r = 50$ on 4x8. In contrast, the compute-efficiency of DSGD-MR and ASGD was penalized since $\bar{N}$ is small for KDD. This effect was most pronounced for ASGD (51min on 2x8 vs. 125min on 4x8, $r = 100$). DSGD++ remained unaffected since communication and computation are overlapped. In all cases, shared-nothing SGD-based methods were less data-efficient than PSGD, which was faster overall. A reason for this behavior might be the specifics of the KDD dataset and our choice of loss. For example, when using L_{L2}, we observed a 3x overall speedup for DSGD++ on 4x8 and $r = 50$, when compared to PSGD.

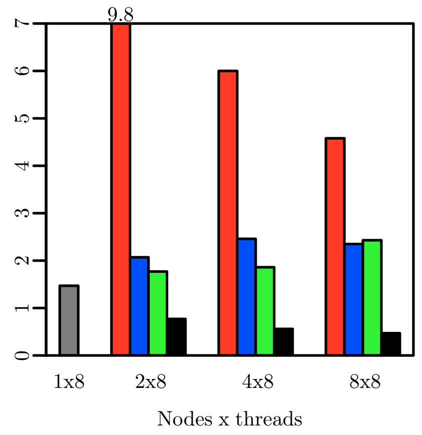
Large-scale experiments (Figure 2.4). For our large-scale experiments in the shared-nothing setting, we used the Syn1B-rect, Syn1B-sq, and Syn10B datasets, and varied the number of nodes (2–16) and threads per node (2–8). When running DSGD++, we used the LA-WOR schedule for Syn1B-sq and Syn1B-rect, and the WOR schedule for Syn10B. The results are summarized in Figure 2.4. The plots show the total time to convergence required by each of the shared-nothing methods and, if applicable, the corresponding shared-memory baselines. Note that the available memory was insufficient to run PALS for all datasets (since it requires two copies of the data in memory) and to run PSGD for Syn10B. In the remainder of this section, we describe the results in more detail for each of the three datasets.

Syn1B-rect (Figure 2.4a). We applied the shared-nothing algorithms to Syn1B-rect on the 1x8, 2x8, 4x8, and 8x8 setups. Syn1B-rect has a high value of $\bar{N}$, so that distributed processing should be relatively effective, at least for smaller numbers of nodes. DSGD-MR confirmed this expectation as it performed better than the PSGD baseline in all cases, though the communication and synchronization overheads of DSGD-MR caused its performance to deteriorate slightly when using beyond 4 nodes. ASGD also outperformed the baseline on 2 and 4 nodes but, in contrast to DSGD-MR, did not behave gracefully when using beyond 4 nodes. Specifically, on 8 nodes ASGD required significantly longer than PSGD. Even though ASGD was more compute-efficient on 8 nodes, the increased synchronization overhead led to increased delays between parameter updates and drastically reduced data efficiency.

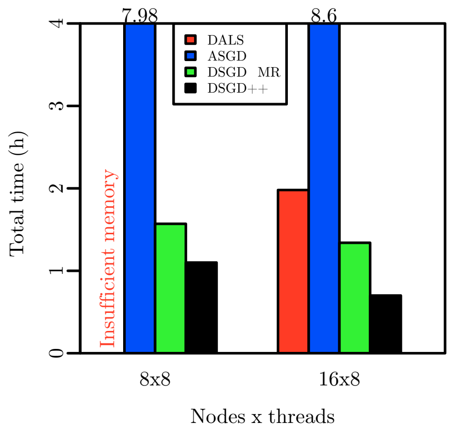
DSGD++ performed best in all setups. It was 2.4x faster than PSGD on 2 nodes and 4.7x faster on 4 nodes. It achieved superlinear speedup, presumably due to larger overall cache sizes. The communication overheads of DSGD++ did not adversely affect performance because of asynchronous communication and use of the LA-WOR schedule. On 8 nodes, the overhead of communication starts to become visible (7.5x speedup). Nevertheless, DSGD++ was able to factor Syn1B-rect in 88min on 8 nodes; its closest competitor was ASGD, which required 186min on 4 nodes.



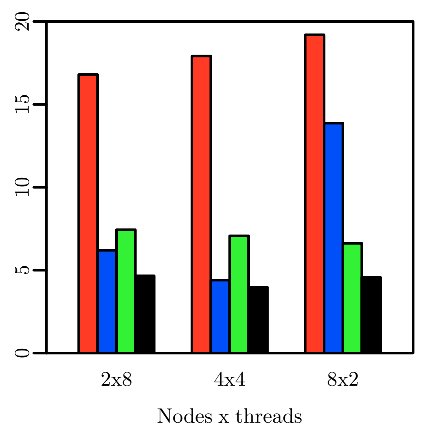
(a) *Rectangular matrix, 1B entries (Syn1B-rect)*



(b) *Square matrix, 1B entries (Syn1B-sq)*



(c) *Rectangular matrix, 10B entries (Syn10B)*



(d) *Overhead of distributed processing (Syn1B-rect)*

Figure 2.4: *Performance of shared-nothing algorithms on synthetic datasets*

Our final observation is that, even though Syn1B-rect is inherently amenable to distributed processing because of its high $\bar{N}$ value, DALS did not converge to an acceptable solution; its loss being four orders of magnitude larger than all other methods. We therefore report the running time of DALS until the change in its loss fell below 0.1% in two consecutive epochs. Such erratic behavior of DALS was not observed on any other dataset. Moreover, DALS exhibited sublinear speedup. This sublinearity indicates that the time required to broadcast the factor matrices becomes significant as the number of nodes increases.

Syn1B-sq (Figure 2.4b). On Syn1B-sq, all methods were faster than on Syn1B-rect since there were fewer factors to learn. Our other observations were as follows.

First, we found that DALS (now working correctly) was consistently slower than the PSGD baseline. As before, increasing the number of nodes led to a sublinear speedup due to increased communication overhead. Second, neither ASGD nor DSGD-MR were able to improve on the PSGD baseline. Indeed, Syn1B-sq is our "hard" dataset (low $\bar{N}$) so that communication overheads dominate potential gains due to use of multiple processing nodes. Note that ASGD behaves more gracefully than on Syn1B-rect for a large number of nodes. We conjecture that the low value of $\bar{N}$ decreases the effect of delayed parameter updates since fewer SGD updates are run per column and time unit (but, as before, the time per epoch decreased and the number of epochs increased). Our final observation was that DSGD++ was the only method that was able to improve upon PSGD. It achieved speedups of 1.6x (2 nodes), 2.3x (4 nodes), and 3.5x (8 nodes). As expected, the speedups are lower than the ones for Syn1B-rect but nevertheless significant. Overall, our results indicate that DSGD++ is the only SGD-based method that can handle matrices with low $\bar{N}$ gracefully.

Syn10B (Figure 2.4c). We could not run experiments on Syn10B using four or fewer nodes due to insufficient aggregate memory. We therefore show results only for 8x8 and 16x8. Even in this setting, we could not run DALS on 8 nodes because the available memory is insufficient to store the required two copies of the data matrix and a full copy of both factor matrices simultaneously (DALS requires at least 11 nodes to do this). On 16 nodes, DALS took 2h to converge; the increased density of the Syn10B matrix simplifies the completion problem so that only 11 epochs were needed. In contrast to DALS, all of the SGD-based methods were able to run on both 8x8 and 16x8. Since these methods store the data matrix only once and also fully partition the factor matrices, they are more memory efficient and can thus be used on smaller clusters. Strikingly, DSGD-MR was faster on 8 nodes than DALS on 16 nodes. ASGD did not converge to a satisfactory point in this experiment (2 orders of magnitude off the best loss) and was much slower than all other methods (as before, we declared convergence when the loss reduced by less than 0.1%). This is another indication that ASGD is not robust enough for larger clusters. Finally, DSGD++ required 1.1h on 8 nodes and 0.7h on 16 nodes. Thus, DSGD++ was faster on 8 nodes than any other method on 16 nodes, and was almost twice as fast on 16 nodes as its closest competitor (DSGD-MR).

Impact of distributed processing (Figure 2.4d). In our final experiment on Syn1B-rect, we investigated the behavior of the various algorithms as we increased the number of nodes, while keeping the overall number of threads constant (2x8, 4x4, 8x2). Since the number of threads is identical in each setup, this allowed us to directly measure the impact of distributed processing. We observed that all approaches except ASGD handle the increased cluster size gracefully. The runtime of DALS increases slightly (increased cost of broadcasting), while the runtimes of

DSGD++ and DSGD-MR decrease slightly (more cache per thread). Thus, even when less powerful compute nodes are available, these methods perform well. In contrast, the runtime of ASGD again increases sharply as we go beyond 4 nodes; see the discussion above.

2.6 Summary

The matrix completion problem, i.e., predicting missing entries of a partially revealed matrix, arises in various applications in data mining including collaborative filtering in recommender systems, latent semantic indexing, and link prediction in social networks.

In this chapter, we have studied parallel algorithms for large-scale matrix completion with millions of rows, millions of columns, and billions of revealed entries. Our shared-nothing algorithms are designed to run on a small cluster of commodity nodes, have less memory consumption, and offer better scalability than previous MapReduce alternatives. More specifically, our ASGD and DSGD++ algorithms are novel variants of the popular stochastic gradient descent algorithm. Both algorithms are cache-conscious and exploit thread-level parallelism, in-memory processing, and asynchronous communication. Furthermore, our in-memory DALS algorithm is a scalable variant of the alternating least-squares algorithm that utilizes thread-level parallelism to speed up processing and reduces the memory footprint; it is closely related to the shared-nothing algorithm of Zhou et al. (2008). Finally, we have provided the key metrics of the input data that play a crucial role in the performance of distributed matrix completion by both a theoretical complexity analysis as well as an experimental study. To this end, we have compared all the algorithms in an extensive set of experiments on both real-world and synthetic datasets of varying sizes. On large datasets, DSGD++ consistently outperformed alternative approaches with respect to speed, scalability, and memory consumption.

3 Distributed Mixed-Packing-Covering Linear Programming

3.1 The MPC Linear Programs

In this chapter,[1] we investigate linear programs that contain only *non-negative* coefficients and *non-negative* variables. Such LPs are referred to as *mixed packing-covering LPs* and have general form

$$\begin{aligned} \max \quad & \boldsymbol{w}^\top \boldsymbol{x} \\ \text{s.t.} \quad & \boldsymbol{P}\boldsymbol{x} \leq \boldsymbol{p} \\ & \boldsymbol{C}\boldsymbol{x} \geq \boldsymbol{c} \\ & \boldsymbol{x} \geq \boldsymbol{0}, \end{aligned} \tag{MPC-LP}$$

where $\boldsymbol{x} \in \Re_+^n$ denotes a vector of variables, $\boldsymbol{w} \in \Re_+^n$ a vector of weights, $\boldsymbol{P} \in \Re_+^{m \times n}$ a *packing-constraint matrix* with right-hand side $\boldsymbol{p} \in \Re_+^m$, and $\boldsymbol{C} \in \Re_+^{k \times n}$ a *covering-constraint matrix* with right-hand side $\boldsymbol{c} \in \Re_+^k$. Special cases of MPC-LPs include *pure packing* LPs, which only contain packing constraints, i.e., $\max\{\boldsymbol{w}^\top \boldsymbol{x} : \boldsymbol{P}\boldsymbol{x} \leq \boldsymbol{p}, \boldsymbol{x} \geq \boldsymbol{0}\}$, and *pure covering* LPs, which only contain covering constraints, i.e., $\min\{\boldsymbol{w}^\top \boldsymbol{x} : \boldsymbol{C}\boldsymbol{x} \geq \boldsymbol{c}, \boldsymbol{x} \geq \boldsymbol{0}\}$. Both pure packing and pure covering LPs are also referred to as *positive linear programs* (PLPs) in the literature.

[1] Parts of the material in this chapter have been jointly developed with Baruch Awerbuch, Rainer Gemulla, Rohit Khandekar, Julián Mestre, and Mauro Sozio. The chapter is based on Makari et al. (2013) and Makari and Gemulla (2013). The copyright of Makari et al. (2013) is held by VLDB Endowment; the original publication is available at `http://dl.acm.org/citation.cfm?id=2536362`. The copyright of Makari and Gemulla (2013) is held by NIPS; the original publication is available at `http://biglearn.org/2013/files/papers/biglearning2013_submission_14.pdf`.

MPC-LPs constitute a “simple” but still expressive subclass of LPs. These programs commonly arise as LP relaxations of a number of important combinatorial problems and therefore play a crucial role in designing approximation algorithms for such problems. Examples include various multicommodity flow problems (Garg and Könemann 2007; Karakostas 2002; Klein et al. 1994; Leighton et al. 1995; Shahrokhi and Matula 1990), min-max/max-min resource sharing problems (Grigoriadis and Khachiyan 1994; 1996; Grigoriadis et al. 2001; Jansen and Zhang 2002), generalized bipartite matching (Chapter 4), finding (approximate) solutions to non-negative system of linear equations, and so on.

MPC programs can be solved exactly using standard solvers. Today’s commercial LP solvers deploy highly tuned implementations of widely used Simplex (Danzig 1963) and interior-point methods (Karmarkar 1984); they can usually deal with problem instances containing up to millions of non-zero entries, even though, their running times can vary largely depending on the structure of the underlying problem. These solvers, however, may fall short in coping with practical problems of possibly extreme scale with billions of non-zero entries. Therefore, it has become of central importance to develop algorithms that are scalable enough to process large-scale datasets in a parallelized fashion and achieve reasonable performance at massive scales.

It is well-known that solving a general LP or even approximating its objective value is P-complete (Dobkin et al. 1979; Khachiyan 1979; Megiddo 1992; Serna 1991). Therefore, solving general LPs is inherently sequential, i.e., there is no fast parallel algorithm for solving or even approximating general LPs, unless $\mathsf{P} = \mathsf{NC}$, where NC refers to the set of decision problems solvable in *poly-logarithmic* time using a parallel computer with a *polynomial* number of processors. In fact, Trevisan and Xhafa (1998) showed that even solving the special class of PLPs exactly is P-complete, unless $\mathsf{P} = \mathsf{NC}$. Thus, this is also the case for the more general class of MPC-LPs (which we are interested in solving). Nevertheless, special classes of PLPs or MPC-LPs accept efficient parallel approximation algorithms; see the related work in Section 3.7.

In this chapter, we develop a parallel approximation algorithm for solving any *explicitly given* MPC-LP. We consider an approximate variant of MPC-LP in which we seek for a near-optimal solution and additionally allow for a small violation of packing and covering constraints. This “relaxation” allows us to develop a scalable distributed approximation algorithm for MPC-LPs. In particular, denote by $\epsilon > 0$ a small *error bound*. We say that $\boldsymbol{x} \in \Re^n_+$ is an ϵ*-feasible* solution to MPC-LP if packing and covering constraints are violated by at most a factor of $1 + \epsilon$ and $1 - \epsilon$, respectively, i.e., $\boldsymbol{Px} \leq (1+\epsilon)\boldsymbol{p}$ and $\boldsymbol{Cx} \geq (1-\epsilon)\boldsymbol{c}$. For $1 > \eta > 0$, an ϵ-feasible solution is called an (ϵ, η)*-approximation* if it obtains an objective value within a factor of $(1-\epsilon)(1-\eta)$ of the optimal solution to MPC-LP.

In the sequel, we first present our algorithm called MPCSolver that obtains an ϵ-feasible solution to any MPC-LP (Section 3.2). MPCSolver requires a poly-logarithmic number of passes over the data (in the input size and the width of the LP). Next, we show how MPCSolver can be used to obtain an (ϵ, η)-approximation to MPC-LPs (Section 3.3). MPCSolver is simple and efficient in practice, and can be readily implemented on GPUs, shared-memory and shared-nothing architectures, as well as MapReduce. Parallelizing MPCSolver in a shared-nothing environment is described afterwards (Section 3.4). Finally, we discuss some implementation details to improve the performance of our algorithm in practice (Section 3.5) and conclude with an overview of the results of a case study with instances of generalized bipartite matching problems (Section 3.6) and a review of related work (Section 3.7). Throughout this chapter, we use the notation from Appendix A.

3.2 Solving MPC-LPs (Feasibility)

In this section, we discuss how to solve MPC-LP feasibility problems, i.e., ignoring the objective function. MPCSolver (given as Algorithm 3) is inspired by, but more general than, the work of Awerbuch and Khandekar (2009), which can handle pure packing and covering constraints. To simplify our discussion, we first describe our algorithm in a centralized environment and then generalize to the distributed setting in Section 3.4.

Recall the definition of MPC-LP given above. Without loss of generality, we assume that $\boldsymbol{p} = \mathbb{1}$ and $\boldsymbol{c} = \mathbb{1}$, where $\mathbb{1}$ denotes an all-one vector of the appropriate dimensionality, and that each of the non-zero entries in $\boldsymbol{P}$ and $\boldsymbol{C}$ is equal to or larger than 1. Denote by M the largest entry in $\boldsymbol{P}$ and $\boldsymbol{C}$; this is referred to as the *width* of LP. Note that any MPC-LP can be transformed in this form: This can be achieved by first rescaling the rows, then the columns of the coefficient matrices. The former step ensures that all right-hand sides are equal to 1, while the latter step guarantees that all entries of the coefficient matrices are at least 1. Let $\boldsymbol{P}'$ and $\boldsymbol{C}'$, respectively, denote the packing- and covering-constraint matrices before this transformation. Notice that we can assume that $\boldsymbol{p}_i > 0$ for all i (since otherwise all variables $\boldsymbol{x}_{ij}$ with $\boldsymbol{P}'_{ij} > 0$ must be set to zero). Similarly, we can assume that $\boldsymbol{c}_i > 0$ for all i (since otherwise constraint i trivially holds and can be removed). We proceed as follows. Let $\pi_j = \min\{\pi_j^{\boldsymbol{P}}, \pi_j^{\boldsymbol{C}}\}$, where $\pi_j^{\boldsymbol{P}} = \min_{i,\boldsymbol{P}'_{ij} \neq 0} \frac{\boldsymbol{P}'_{ij}}{\boldsymbol{p}_i}$ and $\pi_j^{\boldsymbol{C}} = \min_{i,\boldsymbol{C}'_{ij} \neq 0} \frac{\boldsymbol{C}'_{ij}}{\boldsymbol{c}_i}$. We replace $\boldsymbol{P}'_{ij}$ by $\boldsymbol{P}_{ij} = \frac{\boldsymbol{P}'_{ij}}{\boldsymbol{p}_i \pi_j}$ and $\boldsymbol{C}'_{ij}$ by $\boldsymbol{C}_{ij} = \frac{\boldsymbol{C}'_{ij}}{\boldsymbol{c}_i \pi_j}$. Note that column scaling will change the solution; instead of working with the original variables $\boldsymbol{x}_j$, we work with the modified variables $\boldsymbol{x}'_j = \boldsymbol{x}_j \pi_j$. We can now recover the solution to the original problem by "inversely" scaling the result of the modified problem, i.e., $\boldsymbol{x}_j = \frac{\boldsymbol{x}'_j}{\pi_j}$.

Thus, we aim to find a vector $\boldsymbol{x}$ such that

$$\begin{aligned} \boldsymbol{Px} &\leq \mathbb{1} \\ \boldsymbol{Cx} &\geq \mathbb{1} \\ \boldsymbol{x} &\geq \mathbf{0}. \end{aligned} \tag{3.1}$$

For any value of $\boldsymbol{x} \in \Re^n_+$, let $\boldsymbol{y}(\boldsymbol{x}) = (\boldsymbol{y}_1, \ldots, \boldsymbol{y}_m) \in \Re^m_+$ and $\boldsymbol{z}(\boldsymbol{x}) = (\boldsymbol{z}_1, \ldots, \boldsymbol{z}_k) \in \Re^k_+$ be given as follows

$$\boldsymbol{y}_i(\boldsymbol{x}) = \exp\left[\mu(\boldsymbol{P}_{i*}\boldsymbol{x} - 1)\right] \tag{3.2}$$

$$\boldsymbol{z}_i(\boldsymbol{x}) = \exp\left[\mu(1 - \boldsymbol{C}_{i*}\boldsymbol{x})\right], \tag{3.3}$$

where μ is a scaling factor (defined in Algorithm 3). For brevity, we suppress the dependence of $\boldsymbol{y}(\boldsymbol{x})$ and $\boldsymbol{z}(\boldsymbol{x})$ on $\boldsymbol{x}$ when the value of $\boldsymbol{x}$ is clear from context. One may think of $\boldsymbol{y}_i$ as an exponential "penalty" function of packing constraint $\boldsymbol{P}_{i*}\boldsymbol{x} \leq 1$. Penalty $\boldsymbol{y}_i$ is small if the constraint is satisfied ($\boldsymbol{y}_i \leq 1$ if $\boldsymbol{P}_{i*}\boldsymbol{x} \leq 1$) and large otherwise ($\boldsymbol{y}_i > 1$ if $\boldsymbol{P}_{i*}\boldsymbol{x} > 1$). Similarly, $\boldsymbol{z}_i$ is a penalty function for covering constraint $\boldsymbol{C}_{i*}\boldsymbol{x} \geq 1$. In what follows, we refer to $\boldsymbol{y}_i$ and $\boldsymbol{z}_i$ as the *dual variable* of the corresponding constraint. Our algorithm tries to minimize the overall penalty, i.e., the *potential function*

$$\Phi(\boldsymbol{x}) = \sum_{i=1}^{m} \boldsymbol{y}_i(\boldsymbol{x}) + \sum_{i=1}^{k} \boldsymbol{z}_i(\boldsymbol{x}).$$

The scaling constant μ is chosen sufficiently large so that if Φ is (approximately) minimized, the corresponding solution is ϵ-feasible given that the MPC-LP is feasible (see the analysis of Theorem 1). Since Φ is differentiable and convex in $\boldsymbol{x}$, we use a version of gradient descent to find the optimal solution (i.e., the one with lowest penalty). Consider the partial derivative of Φ w.r.t. $\boldsymbol{x}_j$:

$$\frac{\partial \Phi}{\partial \boldsymbol{x}_j} = \mu \boldsymbol{P}_{*j}^{\top} \boldsymbol{y} - \mu \boldsymbol{C}_{*j}^{\top} \boldsymbol{z}.$$

At the minimum, all partial derivatives are zero. When the derivative is negative (positive), we will increase (decrease) $\boldsymbol{x}$ by a carefully chosen amount.

A description of our algorithm is given in Algorithm 3; we refer to this algorithm as MPCSolver. Here parameter ϵ' is an internal error bound, α acts as an *update threshold*, β as a multiplicative *step size*, and δ as an *additive increase*; each parameter is chosen carefully and depends on error bound ϵ. The algorithm starts with an arbitrary initial point $\boldsymbol{x}(0) \in \Re^+$. In each *round* (i.e., each iteration of the repeat-until loop), we first compute the values of the dual variables $\boldsymbol{y}$ and $\boldsymbol{z}$. We update variable $\boldsymbol{x}_j$ only if its partial derivative is sufficiently far away from zero. The algorithm terminates once all variables are left unmodified (or one

Algorithm 3 MPCSolver for mixed packing-covering LPs

```
Require: packing constraint $\boldsymbol{P}$, covering constraints $\boldsymbol{C}$, error bound $\epsilon$
 1: $\epsilon' = \epsilon/10$                                  // internal error bound
 2: $\mu \leftarrow \ln(mkM/\epsilon')/\epsilon'$                    // scaling constant
 3: $\alpha \leftarrow \epsilon'/4$                                 // update threshold
 4: $\beta \leftarrow \alpha/(20\mu)$                           // multiplicative step size
 5: $\delta \leftarrow \beta/nM$                                 // additive increase
 6: Start with any $\boldsymbol{x} \in \Re^n_+$
 7: repeat
 8:   Compute $\boldsymbol{y}_i(\boldsymbol{x}) = \exp\left[\mu(\boldsymbol{P}_{i*}\boldsymbol{x} - 1)\right]$ for $i = 1, \ldots, m$
 9:   Compute $\boldsymbol{z}_i(\boldsymbol{x}) = \exp\left[\mu(1 - \boldsymbol{C}_{i*}\boldsymbol{x})\right]$ for $i = 1, \ldots, k$
10:   for $j = 1, \ldots, n$ do
11:     if $\frac{\boldsymbol{P}_{*j}^\top \boldsymbol{y}(\boldsymbol{x})}{\boldsymbol{C}_{*j}^\top \boldsymbol{z}(\boldsymbol{x})} \leq 1 - \alpha$ then
12:       $\boldsymbol{x}_j \leftarrow \max\{\boldsymbol{x}_j(1+\beta), \delta\}$
13:     if $\frac{\boldsymbol{P}_{*j}^\top \boldsymbol{y}(\boldsymbol{x})}{\boldsymbol{C}_{*j}^\top \boldsymbol{z}(\boldsymbol{x})} \geq 1 + \alpha$ then
14:       $\boldsymbol{x}_j \leftarrow \boldsymbol{x}_j(1-\beta)$
15: until convergence (Section 3.5.4)
```

of the alternative convergence tests of Section 3.5.4 applies); we then obtain an approximate minimizer of Φ. In particular, we update $\boldsymbol{x}_j$ if and only if the ratio $r_j = \frac{\boldsymbol{P}_{*j}^\top \boldsymbol{y}(\boldsymbol{x})}{\boldsymbol{C}_{*j}^\top \boldsymbol{z}(\boldsymbol{x})}$ lies outside $(1-\alpha, 1+\alpha)$; thus α acts as an update condition. If the ratio r_j exceeds $1+\alpha$ (positive gradient), we decrease $\boldsymbol{x}_j$; if the ratio is below $1-\alpha$ (negative gradient), we increase $\boldsymbol{x}_j$. Step size parameter β determines how quickly we move through the parameter space. Updates are multiplicative: We add or subtract $\beta\boldsymbol{x}_j$ from $\boldsymbol{x}_j$; thus the larger $\boldsymbol{x}_j$, the more it is changed. Finally, we ensure that $\boldsymbol{x}_j \geq \delta$ after an increase so that we can quickly move away from 0 when $\boldsymbol{x}_j$ is very small. Note that our algorithm can be implemented in a few lines of code.

MPCSolver is designed such that it converges quickly to an ϵ-feasible solution. Our main theoretical result is as follows:

Theorem 1 (Main)**.** *For $0 < \epsilon \leq 0.5$, Algorithm 3 produces an ϵ-feasible solution after*

$$\tilde{O}\left(\frac{1}{\epsilon^5}\ln^3(kmMnx_{max})\right)$$

rounds, where $x_{max} = \max\{\delta, \boldsymbol{x}_1(0), \ldots, \boldsymbol{x}_n(0)\}$ and $\boldsymbol{x}_j(0)$ denotes the j-th element of starting point $\boldsymbol{x}(0)$ (note that $\delta \ll \epsilon$). Once the solution becomes ϵ-feasible, it will stay ϵ-feasible in subsequent rounds. For $\epsilon > 0.5$, all properties are retained w.r.t. $O(\epsilon)$.

The theorem asserts that the number of rounds required by MPCSolver is poly-logarithmic in the input, which ensures fast convergence. Here we used $\tilde{O}$-notation to hide lower-order terms; a more precise bound is given by

$$O\left(\frac{1}{\epsilon^5}\ln^3\frac{1}{\epsilon}\ln^2(kmM)\ln(k\ln(m)Mnx_{max})\right)$$

(see below for the derivation). Note that MPCSolver uses an internal error bound ϵ', which is set to $\frac{\epsilon}{10}$ in Algorithm 3. For sufficiently small $\epsilon \leq 0.5$, this ensures convergence to an ϵ-feasible solution. In Section 3.5.2, we show how to dynamically adapt ϵ' to improve performance in practice.

In the sequel, we give an outline of the proof; the full proof appears afterwards. The key ideas underlying our proof are due to Awerbuch and Khandekar (2009) (who considered packing LPs or covering LPs, but not MPC-LPs); here we adapt the proof to our setting. We make use of the internal step size parameter $\epsilon' = \frac{\epsilon}{10}$ throughout. We start by discussing our choice of parameters. First, parameters β and δ are chosen such that if $\boldsymbol{x}$ satisfies $\boldsymbol{Px} \leq 3 \cdot \mathbb{1}$ in the beginning of a round, the value of $\boldsymbol{P}_{*j}^\top \boldsymbol{y}(\boldsymbol{x})$ changes by at most a factor of $\frac{\alpha}{4}$ in that round, $1 \leq j \leq n$. We set

$$\beta = \frac{\alpha}{20\mu} = \Theta\left(\frac{\epsilon}{\mu}\right), \text{ and } \delta = \frac{\alpha}{20\mu nM} = \Theta\left(\frac{\epsilon}{\mu nM}\right).$$

With these values, we ensure that if the solution $\boldsymbol{x}$ in the beginning of a round satisfies $\boldsymbol{P}_{i*}\boldsymbol{x} \leq 3$ for some i, then the maximum change (increase or decrease) in $\boldsymbol{P}_{i*}\boldsymbol{x}$ (upper bounded by $3\beta + nM\delta$) is at most $\frac{\alpha}{5\mu}$ in that round. Thus, $\boldsymbol{y}_i(\boldsymbol{x})$ changes by a factor of at most $\exp(\frac{\alpha}{5}) \leq 1 + \frac{\alpha}{4}$. Similarly, if $\boldsymbol{x}$ in the beginning of a round satisfies $\boldsymbol{C}_{i*}\boldsymbol{x} \leq 3$, then the value of $\boldsymbol{C}_{*j}^\top \boldsymbol{z}(\boldsymbol{x})$ for any j changes by at most $\frac{\alpha}{4}$ in that round.

Analysis

Proof sketch. Assume for now that the MPC-LP is feasible. We start by partitioning the rounds of MPCSolver into a set of disjoint *intervals*, each consisting of a sequence of consecutive rounds. The first interval is a "warm-up" interval consisting of a poly-logarithmic number τ_0 of rounds. We show that after τ_0 rounds, the potential is sufficiently small, i.e., at most $\Phi_{\text{init}} = m\exp(\mu(1+2\epsilon')) + k\exp(\mu)$ (implied by Lemma 1). More specifically, using the values of β and δ as specified above, we have

$$\begin{aligned}\tau_0 &= O\left(\frac{1}{\beta}\ln(\frac{kMnx_{max}}{\delta})\right)\\ &= O\left(\frac{1}{\epsilon^2}\ln(\frac{kmM}{\epsilon})\ln(\frac{1}{\epsilon^2}kM^2n^2x_{max}\ln(\frac{kmM}{\epsilon}))\right)\\ &= O\left(\frac{1}{\epsilon^2}\ln(\frac{kmM}{\epsilon})\ln(\frac{1}{\epsilon}k\ln(m)Mnx_{max})\right). \qquad (3.4)\end{aligned}$$

Next, we prove that after the warm-up interval, the potential Φ is monotonically non-increasing, and that its decrease per round can be bounded from below (Lemma 3). We then divide the remaining intervals into *stationary* and *unstationary* intervals of length τ_1, where

$$\begin{aligned}\tau_1 &= O\left(\frac{1}{\beta}\ln\frac{1}{\delta}\right)\\ &= O\left(\frac{1}{\epsilon^2}\ln(\frac{kmM}{\epsilon})\ln(\frac{Mn}{\epsilon^2}\ln(\frac{kmM}{\epsilon}))\right)\\ &= O\left(\frac{1}{\epsilon^2}\ln(\frac{kmM}{\epsilon})\ln(\frac{Mn}{\epsilon}\ln(km))\right).\end{aligned} \tag{3.5}$$

An interval is stationary if the potential does not change significantly (specified in Definition 1); as soon as we see a stationary interval, we show that the solution is $\theta(\epsilon)$-feasible (ϵ-feasible for $\epsilon \le 0.5$, Lemma 5). All other intervals are unstationary; in these intervals, the potential decreases by a factor of at least $\Omega(\epsilon^2)$ (Lemma 4). Note that with our choice of μ, the solution becomes feasible and the algorithm terminates when the potential drops below $\Phi_{\text{fin}} = m+k$. We conclude that there are at most $O(\frac{1}{\epsilon^2}\log\frac{\Phi_{\text{init}}}{\Phi_{\text{fin}}})$ unstationary intervals. Putting all pieces together, x becomes $\theta(\epsilon)$-feasible after $\tau = \tau_0 + O(\frac{\tau_1}{\epsilon^2}\log\frac{\Phi_{\text{init}}}{\Phi_{\text{fin}}})$ rounds, which matches the bound of Theorem 1 as follows

$$\begin{aligned}\tau &= \tau_0 + O\left(\frac{\tau_1}{\epsilon^2}\log\frac{\Phi_{\text{init}}}{\Phi_{\text{fin}}}\right)\\ &= O\left(\frac{1}{\epsilon^2}\ln(\frac{kmM}{\epsilon})\ln(\frac{1}{\epsilon}k\ln(m)Mnx_{max}) + \frac{1}{\epsilon^5}\ln^2(\frac{kmM}{\epsilon})\ln(\frac{Mn}{\epsilon}\ln(km))\right)\\ &= O\left(\frac{1}{\epsilon^5}\ln^3\frac{1}{\epsilon}\ln^2(kmM)\ln(k\ln(m)Mnx_{max})\right)\\ &= \tilde{O}\left(\frac{1}{\epsilon^5}\ln^3(kmMnx_{max})\right).\end{aligned}$$

Here we used (3.4), (3.5), and $\frac{\Phi_{\text{init}}}{\Phi_{\text{fin}}} = O(\exp(\mu))$. Since after the warm-up interval the potential Φ is monotonically non-increasing, the solution stays $\theta(\epsilon)$-feasible in subsequent rounds. Finally, to handle infeasible problem instances, we always terminate Algorithm 3 after τ rounds, whether or not a solution has been found. ■

Denote by $\boldsymbol{x}(t), \boldsymbol{y}(t)$, and $\boldsymbol{z}(t)$ the values of $\boldsymbol{x}$, $\boldsymbol{y}(\boldsymbol{x})$, and $\boldsymbol{z}(\boldsymbol{x})$ in round t. The potential in round t is then given by

$$\Phi(t) = \mathbb{1}\cdot\boldsymbol{y}(t) + \mathbb{1}\cdot\boldsymbol{z}(t).$$

Note that $\boldsymbol{y}$ and $\boldsymbol{z}$ are fast-growing functions of $\boldsymbol{x}$: Any "significant" change in the value of $\boldsymbol{x}$ leads to a "significant" change in the value of Φ. We assume throughout that the problem instance is feasible; otherwise, our analysis implies that if Algorithm 3 does not find a solution after the poly-logarithmic number of rounds asserted by Theorem 1, the MPC-LP is infeasible.

E. Warm-Up Interval

We show in Lemma 1 that after a warm-up interval of poly-logarithmic length, we do not violate the packing and covering constraints by too much. This allows us to bound the value of function Φ so that we can bound the number of stationary and unstationary intervals later on.

Lemma 1. *Let the algorithm start from a point* $\boldsymbol{x}(0) \in \Re^n_+$ *and let* $x_{\max} = \max_j \boldsymbol{x}_j(0)$. *After* $\tau_0 = O\left(\frac{1}{\beta}\log(\frac{nkM}{\delta}(x_{\max}+\delta))\right)$ *rounds, as long as* $\boldsymbol{x}(t)$ *does* not *form a feasible solution to* (3.1)*, we have*

- $1 - 2\epsilon' \leq \max_i \boldsymbol{P}_{i*}\boldsymbol{x}(t) < 2 + 2\epsilon'$, *and* (3.6)
- $\min_i \boldsymbol{C}_{i*}\boldsymbol{x}(t) \leq 1 + 2\epsilon'$. (3.7)

Proof. We first prove $\max_i \boldsymbol{P}_{i*}\boldsymbol{x}(t) < 2 + 2\epsilon'$. Assume that $\max_i \boldsymbol{P}_{i*}\boldsymbol{x} > 2 + \epsilon'$ holds initially. Note that for all packing constraints i with $\boldsymbol{P}_{i*}\boldsymbol{x} > 2 + \epsilon'$, we have

$$\boldsymbol{y}_i > \exp(\mu(1+\epsilon')) = \frac{mkM}{\epsilon'}\exp(\mu).$$

Note also that $\boldsymbol{z}_i \leq \exp(\mu)$ for all covering constraints i. Thus, for all variables j such that $\boldsymbol{P}_{ij} \neq 0$ for some i with $\boldsymbol{P}_{i*}\boldsymbol{x} > 2 + \epsilon'$, it holds that

$$\frac{\boldsymbol{P}_{*j}^\top \boldsymbol{y}}{\boldsymbol{C}_{*j}^\top \boldsymbol{z}} > \frac{m}{\epsilon'} > 1 + \alpha.$$

Thus, all these variables decrease by a factor of β. Hence after $O(\frac{1}{\beta}\log(nMx_{\max}))$ rounds, it holds that $\max_i \boldsymbol{P}_{i*}\boldsymbol{x} \leq 2 + \epsilon'$. Moreover, since $\boldsymbol{P}_{i*}\boldsymbol{x}$ in a single round increases to at most $(1+\beta)\boldsymbol{P}_{i*}\boldsymbol{x} + nM\delta$, we conclude that $\max_i \boldsymbol{P}_{i*}\boldsymbol{x}$ can increase to at most

$$(2+\epsilon')(1+\beta) + nM\delta < 2 + 2\epsilon'$$

in any subsequent round.

We now show $\max_i \boldsymbol{P}_{i*}\boldsymbol{x}(t) \geq 1 - 2\epsilon'$. Consider the duration in which $\boldsymbol{x}$ does not form an $O(\epsilon')$-feasible solution to (3.1). Also assume that $\max_i \boldsymbol{P}_{i*}\boldsymbol{x} \leq 1 - \epsilon'$. Thus, $\boldsymbol{y}_i \leq \frac{\epsilon'}{mkM}$ for all packing constraints i. Note that since $\boldsymbol{x}$ is not a feasible solution, we have $\min_i \boldsymbol{C}_{i*}\boldsymbol{x} < 1$. Since $\boldsymbol{C}_{i*}\boldsymbol{x} < 1$ implies $\boldsymbol{z}_i > 1$, we have that all variables j with $\boldsymbol{C}_{ij} \neq 0$ for some i with $\boldsymbol{C}_i\boldsymbol{x} \leq 1$ satisfy

$$\frac{\boldsymbol{P}_{*j}^\top \boldsymbol{y}}{\boldsymbol{C}_{*j}^\top \boldsymbol{z}} \leq \frac{\epsilon'}{k} < 1 - \alpha.$$

Thus, all these variables increase by factor $(1+\beta)$ and therefore after $O(\frac{1}{\beta}\log\frac{1}{\delta})$ rounds, we have $\max_i \boldsymbol{P}_{i*}\boldsymbol{x} > 1-\epsilon'$. Since any $\boldsymbol{x}_j$ decreases by a factor of at most $(1-\beta)$ in any single round, we always have

$$\max_i \boldsymbol{P}_{i*}\boldsymbol{x} > (1-\epsilon')(1-\beta) \geq 1-2\epsilon'$$

in any subsequent round.

Next, we show $\min_i \boldsymbol{C}_{i*}\boldsymbol{x}(t) \leq 1+2\epsilon'$. Assume that $\min_i \boldsymbol{C}_{i*}\boldsymbol{x} \geq 1+\epsilon'$. Thus, $\boldsymbol{z}_i \leq \frac{\epsilon'}{mkM}$ for all covering constraints i. Note that since $\boldsymbol{x}$ is not a feasible solution, we have $\max_i \boldsymbol{P}_{i*}\boldsymbol{x} > 1$. Since $\boldsymbol{P}_{i*}\boldsymbol{x} > 1$ implies $\boldsymbol{y}_i > 1$, we have that all variables j with $\boldsymbol{P}_{ij} \neq 0$ for some i with $\boldsymbol{P}_{i*}\boldsymbol{x} > 1$ satisfy

$$\frac{\boldsymbol{P}_{*j}\boldsymbol{y}}{\boldsymbol{C}_{*j}\boldsymbol{z}} \geq \frac{m}{\epsilon'} > 1+\alpha.$$

Thus, all these variables decrease by factor $(1-\beta)$. Hence after $O(\frac{1}{\beta}\log(kMx_{\max}))$ rounds, we have $\min_i \boldsymbol{C}_{i*}\boldsymbol{x} < 1+\epsilon'$. Since any $\boldsymbol{x}_j$ increases by a factor of at most $(1+\beta)$ in any single round, we always have

$$\min_i \boldsymbol{C}_{i*}\boldsymbol{x} < (1+\epsilon')(1+\beta) \leq 1+2\epsilon'$$

in any subsequent round. ■

Note that Lemma 1 implies that after τ_0 rounds the potential is at most

$$\Phi_{\text{init}} = m\exp(\mu(1+2\epsilon')) + k\exp(\mu).$$

The following rather technical lemma will be needed in the proof of Lemma 5.

Lemma 2. *After τ_0 rounds, we have*

- $(\max_i \boldsymbol{P}_{i*}\boldsymbol{x} - \epsilon')(\mathbb{1}\cdot\boldsymbol{y}) \leq (1+\epsilon')\,\boldsymbol{y}^\top\boldsymbol{P}\boldsymbol{x}$, *and* (3.8)
- $(\min_i \boldsymbol{C}_{i*}\boldsymbol{x} + \epsilon')(\mathbb{1}\cdot\boldsymbol{z}) \geq \boldsymbol{z}^\top\boldsymbol{C}\boldsymbol{x}.$ (3.9)

Proof. We first show (3.8). Let $1 \leq i_0 \leq m$ be such that $\boldsymbol{P}_{i_0*}\boldsymbol{x} = \max_i \boldsymbol{P}_{i*}\boldsymbol{x}$ and define $S_p = \{i \mid \boldsymbol{P}_{i*}\boldsymbol{x} < \boldsymbol{P}_{i_0*}\boldsymbol{x} - \epsilon'\}$. Thus, for all $i \in S_p$ it holds that

$$\boldsymbol{y}_i < \exp(\mu(\boldsymbol{P}_{i_0*}\boldsymbol{x} - \epsilon' - 1)) < \boldsymbol{y}_{i_0}\exp(-\epsilon'\mu) < \frac{\epsilon'\boldsymbol{y}_{i_0}}{m}.$$

Hence $\sum_{i\in S_p}\boldsymbol{y}_i \leq \epsilon'\boldsymbol{y}_{i_0}$ and we get

$$\mathbb{1}\cdot\boldsymbol{y} = \sum_{i\in S_p}\boldsymbol{y}_i + \sum_{i\notin S_p}\boldsymbol{y}_i < \epsilon'\boldsymbol{y}_{i_0} + \sum_{i\notin S_p}\boldsymbol{y}_i < (1+\epsilon')\sum_{i\notin S_p}\boldsymbol{y}_i. \tag{3.10}$$

Therefore, we get the desired assertion as follows

$$\begin{aligned}(\max_i \boldsymbol{P}_{i*}\boldsymbol{x} - \epsilon')(\mathbb{1}\cdot\boldsymbol{y}) &< (\max_i \boldsymbol{P}_{i*}\boldsymbol{x} - \epsilon')(1+\epsilon')\sum_{i\notin S_p}\boldsymbol{y}_i && (3.11)\\ &\le (1+\epsilon')\sum_{i\notin S_p}\boldsymbol{y}_i\cdot\boldsymbol{P}_{i*}\boldsymbol{x} && (3.12)\\ &\le (1+\epsilon')\,\boldsymbol{y}^\top\boldsymbol{P}\boldsymbol{x},\end{aligned}$$

where (3.11) follows from (3.10) and (3.12) holds since $\boldsymbol{P}_{i*}\boldsymbol{x} \ge \boldsymbol{P}_{i_0*}\boldsymbol{x} - \epsilon'$ for all $i \notin S_p$.

Next, we show (3.9). Let $1 \le i_0 \le k$ be such that $\boldsymbol{C}_{i_0*}\boldsymbol{x} = \min_i \boldsymbol{C}_{i*}\boldsymbol{x}$ and define $S_c = \{i \mid \boldsymbol{C}_{i*}\boldsymbol{x} > \boldsymbol{C}_{i_0*}\boldsymbol{x} + \epsilon'\}$. From Lemma 1, we get that $\boldsymbol{C}_{i_0*}\boldsymbol{x} \le 1 + 2\epsilon'$. It is easy to check that

$$\epsilon'\boldsymbol{z}_{i_0} = \epsilon'\exp(\mu(1-\boldsymbol{C}_{i_0*}\boldsymbol{x})) \ge k(\boldsymbol{C}_{i_0*}\boldsymbol{x}+\epsilon')\exp(\mu(1-\boldsymbol{C}_{i_0*}\boldsymbol{x}-\epsilon')).$$

Now fix $i \in S_c$ and let $\eta = \boldsymbol{C}_{i*}\boldsymbol{x}$. Note that $\eta\exp(\mu(1-\eta))$ is a decreasing function of η for $\eta \ge \frac{1}{\mu}$. Since $\boldsymbol{C}_{i_0*}\boldsymbol{x} + \epsilon' \ge \frac{1}{\mu}$, we obtain

$$k(\boldsymbol{C}_{i_0*}\boldsymbol{x}+\epsilon')\exp(\mu(1-\boldsymbol{C}_{i_0*}\boldsymbol{x}-\epsilon')) \ge \sum_{i\in S_c}\boldsymbol{z}_i\cdot\boldsymbol{C}_{i*}\boldsymbol{x},$$

and therefore $\epsilon'\boldsymbol{z}_{i_0} \ge \sum_{i\in S_c}\boldsymbol{z}_i\cdot\boldsymbol{C}_{i*}\boldsymbol{x}$. This implies

$$\epsilon'\boldsymbol{z}_{i_0} + \sum_{i\notin S_c}\boldsymbol{z}_i\cdot\boldsymbol{C}_{i*}\boldsymbol{x} \ge \sum_{i\in S_c}\boldsymbol{z}_i\cdot\boldsymbol{C}_{i*}\boldsymbol{x} + \sum_{i\notin S_c}\boldsymbol{z}_i\cdot\boldsymbol{C}_{i*}\boldsymbol{x} = \boldsymbol{z}^\top\boldsymbol{C}\boldsymbol{x}.$$

Finally from the definition of S_c, we get the desired assertion as follows

$$(\min_i \boldsymbol{C}_{i*}\boldsymbol{x} + \epsilon')(\mathbb{1}\cdot\boldsymbol{z}) \ge \sum_{i\notin S_c}\boldsymbol{z}_i\cdot\boldsymbol{C}_{i*}\boldsymbol{x} + \epsilon'\boldsymbol{z}_{i_0} \ge \boldsymbol{z}^\top\boldsymbol{C}\boldsymbol{x}.$$

■

F. Non-Increasing Potential

The following lemma shows that the potential is monotonically non-increasing after τ_0 rounds, and that its decrease can be bounded from below (w.r.t. the current values of $\boldsymbol{x}$, $\boldsymbol{y}$, and $\boldsymbol{z}$). Before proceeding with Lemma 3, we need the following elementary property of differentiable convex functions.

Fact 1. *Let $f : \Re^n \to \Re$ be a differentiable convex function. For any $\boldsymbol{x}^0, \boldsymbol{x}^1 \in \Re^n$ we have*

$$f'(\boldsymbol{x}^0)(\boldsymbol{x}^1 - \boldsymbol{x}^0) \le f(\boldsymbol{x}^1) - f(\boldsymbol{x}^0) \le f'(\boldsymbol{x}^1)(\boldsymbol{x}^1 - \boldsymbol{x}^0),$$

where $f'(\boldsymbol{x}) = (\frac{\partial f}{\partial \boldsymbol{x}_1}, \ldots, \frac{\partial f}{\partial \boldsymbol{x}_n})^\top$ denotes the gradient of f evaluated at $\boldsymbol{x}$.

Proof. Let $\eta \in [0,1]$. Since f is a convex function, we have

$$f(\boldsymbol{x}^0 + \eta(\boldsymbol{x}^1 - \boldsymbol{x}^0)) = f((1-\eta)\boldsymbol{x}^0 + \eta\boldsymbol{x}^1) \leq (1-\eta)f(\boldsymbol{x}^0) + \eta f(\boldsymbol{x}^1).$$

The inequality on the left is obtained by first subtracting $f(\boldsymbol{x}^0)$ from both sides, then dividing by η on both sides, and finally taking limit as $\eta \to 0$. The inequality on the right follows from exchanging the role of $\boldsymbol{x}^0$ and $\boldsymbol{x}^1$. ■

Lemma 3. *Let $\Delta\Phi(t) = \Phi(t+1) - \Phi(t)$ denote the increase in Φ in round t. Similarly let $\Delta\boldsymbol{x}_j(t) = \boldsymbol{x}_j(t+1) - \boldsymbol{x}_j(t)$, $\Delta\boldsymbol{y}_i(t) = \boldsymbol{y}_i(t+1) - \boldsymbol{y}_i(t)$, and $\Delta\boldsymbol{z}_i(t) = \boldsymbol{z}_i(t+1) - \boldsymbol{z}_i(t)$. After τ_0 rounds, we have*

- $$-\Delta\Phi(t) \geq \Omega(\alpha)\sum_i |\Delta\boldsymbol{y}_i(t)|, \tag{3.13}$$
- $$-\Delta\Phi(t) \geq \Omega(\alpha)\sum_i |\Delta\boldsymbol{z}_i(t)|, \tag{3.14}$$
- $$-\Delta\Phi(t) \geq \Omega(\beta\mu)\big[\boldsymbol{z}(t)^\top\boldsymbol{C}\boldsymbol{x}(t) - (1+\alpha)\boldsymbol{y}(t)^\top\boldsymbol{P}\boldsymbol{x}(t)\big], \tag{3.15}$$
- $$-\Delta\Phi(t) \geq \Omega(\beta\mu)\big[(1-\alpha)\boldsymbol{y}(t)^\top\boldsymbol{P}\boldsymbol{x}(t) - \boldsymbol{z}(t)^\top\boldsymbol{C}\boldsymbol{x}(t)\big]. \tag{3.16}$$

Proof. Note that the potential Φ is a convex differentiable function of $\boldsymbol{x}$. According to Fact 1, for any two vectors $\boldsymbol{x}^0$ and $\boldsymbol{x}^1$, we have

$$\Phi'(\boldsymbol{x}^0)(\boldsymbol{x}^1 - \boldsymbol{x}^0) \leq \Phi(\boldsymbol{x}^1) - \Phi(\boldsymbol{x}^0) \leq \Phi'(\boldsymbol{x}^1)(\boldsymbol{x}^1 - \boldsymbol{x}^0),$$

where $\Phi'(\boldsymbol{x})$ is the gradient of Φ evaluated at $\boldsymbol{x}$. Note that

$$\Phi'_j(\boldsymbol{x}) = \mu(\boldsymbol{P}_{*j}^\top\boldsymbol{y}(\boldsymbol{x}) - \boldsymbol{C}_{*j}^\top\boldsymbol{z}(\boldsymbol{x}))$$

for all j ($\Phi'_j(\boldsymbol{x})$ denotes the partial derivative of Φ w.r.t. $\boldsymbol{x}_j$, i.e., $\Phi'_j(\boldsymbol{x}) = \frac{\partial\Phi}{\partial x_j}$). Thus,

$$\Delta\Phi(t) \leq \mu\sum_j \Delta\boldsymbol{x}_j(t)(\boldsymbol{P}_{*j}^\top\boldsymbol{y}(t+1) - \boldsymbol{C}_{*j}^\top\boldsymbol{z}(t+1)).$$

We first prove (3.13). Define

$$S^+ = \{j \mid \frac{\boldsymbol{P}_{*j}^\top\boldsymbol{y}(t)}{\boldsymbol{C}_{*j}^\top\boldsymbol{z}(t)} \leq 1-\alpha\} \text{ and } S^- = \{j \mid \frac{\boldsymbol{P}_{*j}^\top\boldsymbol{y}(t)}{\boldsymbol{C}_{*j}^\top\boldsymbol{z}(t)} \geq 1+\alpha\}.$$

Note that $\Delta\boldsymbol{x}_j(t) > 0$ for all $j \in S^+$ and $\Delta\boldsymbol{x}_j(t) < 0$ for all $j \in S^-$. Moreover, if $j \notin S^+ \cup S^-$, then $\Delta\boldsymbol{x}_j(t) = 0$.

Fix j and assume $j \in S^+$. Note that Lemma 1 and our choices of the parameters ensure that, $\boldsymbol{P}_{*j}^\top\boldsymbol{y}(t)$ does not increase by a factor of more than $\frac{\alpha}{4}$ in round $t+1$, i.e.,

$$\boldsymbol{P}_{*j}^\top\boldsymbol{y}(t+1) \leq (1+\frac{\alpha}{4})\boldsymbol{P}_{*j}^\top\boldsymbol{y}(t).$$

Similarly, $\boldsymbol{C}_{*j}^{\top}\boldsymbol{z}(t)$ does not decrease by a factor of more than $\frac{\alpha}{4}$ in round $t+1$, i.e.,

$$\boldsymbol{P}_{*j}^{\top}\boldsymbol{y}(t+1) \leq (1+\frac{\alpha}{4})\boldsymbol{P}_{*j}^{\top}\boldsymbol{y}(t).$$

Thus,

$$\begin{aligned}\boldsymbol{P}_{*j}^{\top}\boldsymbol{y}(t+1) - \boldsymbol{C}_{*j}^{\top}\boldsymbol{z}(t+1) &\leq (1+\frac{\alpha}{4})\boldsymbol{P}_{*j}^{\top}\boldsymbol{y}(t) - (1-\frac{\alpha}{4})\boldsymbol{C}_{*j}^{\top}\boldsymbol{z}(t) \\ &\leq (1+\frac{\alpha}{4})\boldsymbol{P}_{*j}^{\top}\boldsymbol{y}(t) - \frac{(1-\frac{\alpha}{4})}{(1-\alpha)}\boldsymbol{P}_{*j}^{\top}\boldsymbol{y}(t) \qquad (3.17) \\ &\leq (-\frac{\alpha}{2})\boldsymbol{P}_{*j}^{\top}\boldsymbol{y}(t) \\ &\leq -\Omega(\alpha)\boldsymbol{P}_{*j}^{\top}\boldsymbol{y}(t),\end{aligned}$$

where inequality (3.17) follows from the fact that $j \in S^{+}$.

Now fix j and assume $j \in S^{-}$. Following similar arguments as above we have

$$\begin{aligned}\boldsymbol{P}_{*j}^{\top}\boldsymbol{y}(t+1) - \boldsymbol{C}_{*j}^{\top}\boldsymbol{z}(t+1) &\geq (1-\frac{\alpha}{4})\boldsymbol{P}_{*j}^{\top}\boldsymbol{y}(t) - (1+\frac{\alpha}{4})\boldsymbol{C}_{*j}^{\top}\boldsymbol{z}(t) \\ &\geq (1-\frac{\alpha}{4})\boldsymbol{P}_{*j}^{\top}\boldsymbol{y}(t) - \frac{(1+\frac{\alpha}{4})}{1+\alpha}\boldsymbol{P}_{*j}^{\top}\boldsymbol{y}(t) \\ &\geq \Omega(\alpha)\boldsymbol{P}_{*j}^{\top}\boldsymbol{y}(t).\end{aligned}$$

From the above analysis, we have

$$\Delta\Phi(t) \leq -\Omega(\alpha)\mu\sum_{j}|\Delta\boldsymbol{x}_j(t)|\,\boldsymbol{P}_{*j}^{\top}\boldsymbol{y}(t). \qquad (3.18)$$

Now note that for any i, we have

$$\Delta\boldsymbol{y}_i(t) \leq \sum_{j}\mu\boldsymbol{P}_{ij}\boldsymbol{y}_i(t+1)\Delta\boldsymbol{x}_j(t) \leq (1+\frac{\alpha}{4})\sum_{j}\mu\boldsymbol{P}_{ij}\boldsymbol{y}_i(t)\Delta\boldsymbol{x}_j(t).$$

The first inequality follows from Fact 1 (since $\boldsymbol{y}_i(\boldsymbol{x})$ is a convex differentiable function of $\boldsymbol{x}$) and the second inequality is due to the fact that $\boldsymbol{y}_i$ does not change by a factor of more than $\frac{\alpha}{4}$ in any single round.

Therefore,

$$|\Delta\boldsymbol{y}_i(t)| \leq (1+\frac{\alpha}{4})\sum_{j}\mu\boldsymbol{P}_{ij}\boldsymbol{y}_i(t)\,|\Delta\boldsymbol{x}_j(t)|\,.$$

Summing up over all i, we get

$$\sum_{i}|\Delta\boldsymbol{y}_i(t)| \leq \mu(1+\frac{\alpha}{4})\sum_{j}\boldsymbol{P}_{*j}^{\top}\boldsymbol{y}(t)\,|\Delta\boldsymbol{x}_j(t)|\,. \qquad (3.19)$$

The desired assertion (3.13) follows from the inequalities (3.19) and (3.18). The proof of (3.14) is similar to the above and is omitted.

We now prove (3.15):

$$
\begin{aligned}
\Delta\Phi(t) &\leq \mu\sum_{j}\Delta\boldsymbol{x}_j(t)\left[\boldsymbol{P}_{*j}^\top\boldsymbol{y}(t+1)-\boldsymbol{C}_{*j}^\top\boldsymbol{z}(t+1)\right] \\
&= \mu\sum_{j\in S^+}\Delta\boldsymbol{x}_j(t)\left[\boldsymbol{P}_{*j}^\top\boldsymbol{y}(t+1)-\boldsymbol{C}_{*j}^\top\boldsymbol{z}(t+1)\right]+ \\
&\ \mu\sum_{j\in S^-}\Delta\boldsymbol{x}_j(t)\left[\boldsymbol{P}_{*j}^\top\boldsymbol{y}(t+1)-\boldsymbol{C}_{*j}^\top\boldsymbol{z}(t+1)\right] \\
&\leq \mu\sum_{j\in S^+}\Delta\boldsymbol{x}_j(t)\left[\boldsymbol{P}_{*j}^\top\boldsymbol{y}(t+1)-\boldsymbol{C}_{*j}^\top\boldsymbol{z}(t+1)\right] \qquad (3.20)\\
&\leq \beta\mu\sum_{j\in S^+}\boldsymbol{x}_j(t)\left[\boldsymbol{P}_{*j}^\top\boldsymbol{y}(t+1)-\boldsymbol{C}_{*j}^\top\boldsymbol{z}(t+1)\right] \qquad (3.21)\\
&\leq \beta\mu\sum_{j\in S^+}\boldsymbol{x}_j(t)\left[(1+\frac{\alpha}{4})\boldsymbol{P}_{*j}^\top\boldsymbol{y}(t)-(1-\frac{\alpha}{4})\boldsymbol{C}_{*j}^\top\boldsymbol{z}(t)\right] \\
&\leq \Omega(\beta\mu)\sum_{j\in S^+}\boldsymbol{x}_j(t)\left[(1+\alpha)\boldsymbol{P}_{*j}^\top\boldsymbol{y}(t)-\boldsymbol{C}_{*j}^\top\boldsymbol{z}(t)\right] \\
&\leq \Omega(\beta\mu)\sum_{j}\boldsymbol{x}_j(t)\left[(1+\alpha)\boldsymbol{P}_{*j}^\top\boldsymbol{y}(t)-\boldsymbol{C}_{*j}^\top\boldsymbol{z}(t)\right] \qquad (3.22)\\
&= \Omega(\beta\mu)\left[(1+\alpha)\boldsymbol{y}(t)^\top\boldsymbol{P}\boldsymbol{x}(t)-\boldsymbol{z}(t)^\top\boldsymbol{C}\boldsymbol{x}(t)\right].
\end{aligned}
$$

The inequality (3.20) holds since

$$\boldsymbol{P}_{*j}^\top\boldsymbol{y}(t+1)-\boldsymbol{C}_{*j}^\top\boldsymbol{z}(t+1)\geq 0$$

and $\Delta\boldsymbol{x}_j\leq 0$ for any $j\in S^-$. The inequality (3.21) follows from the fact that if $j\in S^+$, then we indeed have $\Delta\boldsymbol{x}_j(t)\geq\beta\boldsymbol{x}_j(t)$. The inequality (3.22) holds since

$$(1+\alpha)\boldsymbol{P}_{*j}^\top\boldsymbol{y}(t)-\boldsymbol{C}_{*j}^\top\boldsymbol{z}(t)>0$$

for any $j\in S^-$. The proof of (3.16) is similar to the above and is omitted. ■

G. Stationary and Unstationary Intervals

We proceed by defining *stationary* intervals in which the potential function does not change significantly.

Definition 1 (Stationary interval). *An interval $\tau=[t_0,t_1]$ of rounds is called* stationary *if all of the following conditions hold:*

- $\sum_{t\in\tau}\sum_i|\Delta\boldsymbol{y}_i(t)|\leq\kappa_1\Phi(t_0)$,
- $\sum_{t\in\tau}\sum_i|\Delta\boldsymbol{z}_i(t)|\leq\kappa_2\Phi(t_0)$,
- $(1-\alpha)\boldsymbol{y}(t)^\top\boldsymbol{P}\boldsymbol{x}(t)-\boldsymbol{z}(t)^\top\boldsymbol{C}\boldsymbol{x}(t)\leq\kappa_3\Phi(t_0)$ *for all $t\in\tau$, and*

- $\boldsymbol{z}(t)^\top \boldsymbol{C}\boldsymbol{x}(t) - (1+\alpha)\boldsymbol{y}(t)^\top \boldsymbol{P}\boldsymbol{x}(t) \le \kappa_4 \Phi(t_0)$ *for all* $t \in \tau$.

Here $\kappa_1, \kappa_2, \kappa_3, \kappa_4 = \theta(\epsilon')$ *are small constants. An interval that is not stationary is called* unstationary.

Lemma 4 shows that the potential function decreases by a multiplicative factor in any unstationary interval, which in turn bounds the total number of unstationary intervals by a poly-logarithmic function.

Lemma 4. *In any unstationary interval, the potential* Φ *decreases by a factor of* $\Omega(\epsilon' \min\{\alpha, \beta\mu\}) = \Omega(\epsilon'^2)$.

Proof. Let $\tau = [t_0, t_1]$ be any unstationary interval. According to Lemma 3, after τ_0 rounds, we can bound the potential decrease from below in any two consecutive rounds. Using inequality (3.13) together with Definition 1, we can bound the overall potential decrease in interval τ as follows

$$\Phi(t_0) - \Phi(t_1) \ge \Omega(\alpha) \sum_{t\in\tau} \sum_i |\Delta \boldsymbol{y}_i(t)| > \Omega(\epsilon'\alpha)\Phi(t_0).$$

Similarly, we can apply inequality 3.16 and Definition 1 to get

$$\Phi(t_0) - \Phi(t_1) \ge \Omega(\beta\mu) \left[(1-\alpha)\boldsymbol{y}(t)^\top \boldsymbol{P}\boldsymbol{x}(t) - \boldsymbol{z}(t)^\top \boldsymbol{C}\boldsymbol{x}(t)\right] > \Omega(\epsilon'\beta\mu)\Phi(t_0).$$

Thus, the claim follows. ■

Lemma 5 completes the proof. It states that all solutions computed in a "sufficiently" long stationary interval are $\theta(\epsilon')$-feasible. The proof is included below.

Lemma 5. *Consider a stationary interval* $\tau = [t_0, t_1]$ *where* $t_0 \ge \tau_0$ *and* $t_1 - t_0 \ge \tau_1$ *where* $\tau_1 = O(\frac{1}{\beta}\log\frac{1}{\delta})$. *Let* $\boldsymbol{x}^0, \boldsymbol{y}^0, \boldsymbol{z}^0$ *denote the values of* $\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z}$ *at round* t_0. *Then* $\boldsymbol{x}^0$ *forms a* $\theta(\epsilon')$*-feasible solution and, in particular, if* $0 < \epsilon' \le 0.05$, $\boldsymbol{x}^0$ *is a* $10\epsilon'$*-feasible solution.*

Since $\epsilon' = \frac{\epsilon}{10}$, we obtain ϵ-feasibility for $\epsilon \le 0.5$.

Proof. Assume to the contrary that the solution $\boldsymbol{x}^0$ is not $\theta(\epsilon')$-feasible, e.g., that $\frac{\min_i \boldsymbol{C}_{i*}\boldsymbol{x}^0}{\max_i \boldsymbol{P}_{i*}\boldsymbol{x}^0} = \lambda \le 1 - 5\epsilon'$. We have

$$\begin{aligned}
&(\max_i \boldsymbol{P}_{i*}\boldsymbol{x}^0 - \epsilon')(\mathbb{1}\cdot \boldsymbol{y}^0) \\
&\le (1+\epsilon')(\boldsymbol{y}^0)^\top \boldsymbol{P}\boldsymbol{x}^0 && (3.23)\\
&\le \frac{1+\epsilon'}{1-\alpha}\left[(\boldsymbol{z}^0)^\top \boldsymbol{C}\boldsymbol{x}^0 + \kappa_3\Phi(t_0)\right] && (3.24)\\
&\le \frac{1+\epsilon'}{1-\alpha}\left[(\min_i \boldsymbol{C}_{i*}\boldsymbol{x}^0 + \epsilon')(\mathbb{1}\cdot \boldsymbol{z}^0) + \kappa_3\Phi(t_0)\right] && (3.25)\\
&\le (1+\frac{5}{3}\epsilon')\left[(\min_i \boldsymbol{C}_{i*}\boldsymbol{x}^0 + \epsilon')(\mathbb{1}\cdot \boldsymbol{z}^0) + \kappa_3\Phi(t_0)\right],
\end{aligned}$$

where the inequalities (3.23) and (3.25) follow from (3.8) and (3.9), respectively, and the inequality (3.24) follows from Definition 1. Now from Lemma 1, after the warm-up interval it always holds that $1 - 2\epsilon' \leq \max_i \boldsymbol{P}_{i*}\boldsymbol{x}^0 \leq 2 + 2\epsilon'$. Therefore,

$$\begin{aligned}\mathbb{1} \cdot \boldsymbol{y}^0 &\leq \left(1 + \frac{5}{3}\epsilon'\right)\left[\frac{\lambda \max_i \boldsymbol{P}_{i*}\boldsymbol{x}^0 + \epsilon'}{\max_i \boldsymbol{P}_{i*}\boldsymbol{x}^0 - \epsilon'}(\mathbb{1} \cdot \boldsymbol{z}^0) + \frac{\kappa_3 \Phi(t_0)}{\max_i \boldsymbol{P}_{i*}\boldsymbol{x}^0 - \epsilon'}\right] \\ &\leq (1 - \epsilon')(\mathbb{1} \cdot \boldsymbol{z}^0) + O(\kappa_3)\Phi(t_0).\end{aligned}$$

Note that $\boldsymbol{y}$ and $\boldsymbol{z}$ have low mileage in the interval τ. According to Definition 1 we have

$$\begin{aligned}&\mathbb{1} \cdot \boldsymbol{y}^0 + \sum_{t\in\tau}\sum_i |\Delta \boldsymbol{y}_i(t)| \\ &\leq (1 - \epsilon')(\mathbb{1} \cdot \boldsymbol{z}^0) + O(\kappa_3)\Phi(t_0) + \sum_{t\in\tau}\sum_i |\Delta \boldsymbol{y}_i(t)| \\ &\leq (1 - \epsilon')(\mathbb{1} \cdot \boldsymbol{z}^0) - \sum_{t\in\tau}\sum_i |\Delta \boldsymbol{z}_i(t)| + (\kappa_1 + \kappa_2)\Phi(t_0) + O(\kappa_3)\Phi(t_0) \\ &\leq (1 - \epsilon')(\mathbb{1} \cdot \boldsymbol{z}^0 - \sum_{t\in\tau}\sum_i |\Delta \boldsymbol{z}_i(t)|) + O(\kappa_1 + \kappa_2 + \kappa_3)\Phi(t_0),\end{aligned}$$

where as before $\kappa_1, \kappa_2, \kappa_3, \kappa_4 = \theta(\epsilon')$ are small constants. Let $\kappa_{123} = \kappa_1 + \kappa_2 + \kappa_3$. Since $\Phi(t_0) = \mathbb{1} \cdot \boldsymbol{y}^0 + \mathbb{1} \cdot \boldsymbol{z}^0$, we have

$$\begin{aligned}&\mathbb{1} \cdot \boldsymbol{y}^0 + \sum_{t\in\tau}\sum_i |\Delta \boldsymbol{y}_i(t)| \\ &\leq (1 - \epsilon')(\mathbb{1} \cdot \boldsymbol{z}^0 - \sum_{t\in\tau}\sum_i |\Delta \boldsymbol{z}_i(t)|) + O(\kappa_{123})(\mathbb{1} \cdot \boldsymbol{y}^0 + \mathbb{1} \cdot \boldsymbol{z}^0)\end{aligned}$$

and

$$\begin{aligned}&(1 - O(\kappa_{123}))(\mathbb{1} \cdot \boldsymbol{y}^0 + \sum_{t\in\tau}\sum_i |\Delta \boldsymbol{y}_i(t)|) \\ &\leq (1 - \epsilon' + O(\kappa_{123}))(\mathbb{1} \cdot \boldsymbol{z}^0 - \sum_{t\in\tau}\sum_i |\Delta \boldsymbol{z}_i(t)|).\end{aligned}$$

Thus,

$$\frac{\mathbb{1} \cdot \boldsymbol{y}^0 + \sum_{t\in\tau}\sum_i |\Delta \boldsymbol{y}_i(t)|}{\mathbb{1} \cdot \boldsymbol{z}^0 - \sum_{t\in\tau}\sum_i |\Delta \boldsymbol{z}_i(t)|} \leq 1 - \epsilon' + O(\kappa_{123}) \leq 1 - \alpha.$$

Since the given mixed LP is feasible, there exists $\boldsymbol{x}^* \geq \boldsymbol{0}$ such that $\boldsymbol{P}\boldsymbol{x}^* \leq \mathbb{1}$ and $\boldsymbol{C}\boldsymbol{x}^* \geq \mathbb{1}$. Thus we have

$$\frac{(\boldsymbol{y}^0)^\top \boldsymbol{P}\boldsymbol{x}^* + \sum_{t\in\tau}\sum_i |\Delta \boldsymbol{y}_i(t)|\, \boldsymbol{P}\boldsymbol{x}^*}{(\boldsymbol{z}^0)^\top \boldsymbol{C}\boldsymbol{x}^* - \sum_{t\in\tau}\sum_i |\Delta \boldsymbol{z}_i(t)|\, \boldsymbol{C}\boldsymbol{x}^*} \leq 1 - \alpha.$$

Now divide both the numerator and the denominator of the left-hand side of the above inequality by $\sum_j \boldsymbol{x}_j^*$. After rearranging the terms we have

$$\frac{\frac{1}{\sum_j \boldsymbol{x}_j^*} \sum_j \boldsymbol{x}_j^* \Big[\boldsymbol{P}_{*j}^\top \boldsymbol{y}^0 + \sum_{t\in\tau} \sum_i |\Delta \boldsymbol{y}_i(t)|\, \boldsymbol{P}_{*j}^\top\Big]}{\frac{1}{\sum_j \boldsymbol{x}_j^*} \sum_j \boldsymbol{x}_j^* \Big[\boldsymbol{C}_{*j}^\top \boldsymbol{z}^0 - \sum_{t\in\tau} \sum_i |\Delta \boldsymbol{z}_i(t)|\, \boldsymbol{C}_{*j}^\top\Big]} \quad \le \quad 1-\alpha.$$

Now think of the numerator (resp. denominator) on the left-hand side above as the (weighted) average of the terms in square brackets in the numerator (resp. denominator). Since the average is less than $1-\alpha$, there exists j such that

$$\frac{\boldsymbol{P}_{*j}^\top \boldsymbol{y}^0 + \sum_{t\in\tau} \sum_i \Big|\boldsymbol{P}_{*j}^\top \boldsymbol{y}(t+1) - \boldsymbol{P}_{*j}^\top \boldsymbol{y}(t)\Big|}{\boldsymbol{C}_{*j}^\top \boldsymbol{z}^0 - \sum_{t\in\tau} \sum_i \Big|\boldsymbol{C}_{*j}^\top \boldsymbol{z}(t+1) - \boldsymbol{C}_{*j}^\top \boldsymbol{z}(t)\Big|} \quad \le \quad 1-\alpha.$$

Note that the first term in the numerator (resp. denominator) is the value of $\boldsymbol{P}_{*j}^\top \boldsymbol{y}$ (resp. $\boldsymbol{C}_{*j}^\top \boldsymbol{z}$) at round t_0 and the second term corresponds to the absolute change in $\boldsymbol{P}_{*j}^\top \boldsymbol{y}^0$ (resp. $\boldsymbol{C}_{*j}^\top \boldsymbol{z}^0$) throughout interval τ. Hence for this j and for all $t \in \tau$, we can conclude that

$$\frac{\boldsymbol{P}_{*j}^\top \boldsymbol{y}(t)}{\boldsymbol{C}_{*j}^\top \boldsymbol{z}(t)} \quad \le \quad 1-\alpha.$$

Our algorithm will increase this variable by a factor of β in each of the $\tau_1 = O(\frac{1}{\beta}\log\frac{1}{\delta})$ rounds in τ. Note that $\boldsymbol{x}_j$ increases to at least δ in each single round and therefore would become larger than 4 in τ_1 rounds. Considering the fact that each non-zero entry in $\boldsymbol{P}$ is at least 1, the potential Φ becomes at least $\exp(3\mu)$ and thus larger than $(m+k)\exp(2\mu)$ after τ_1 rounds. However, this contradicts the fact that the value of the potential always remains less than

$$\Phi_{\text{init}} = m\exp(\mu(1+2\epsilon')) + k\exp(\mu)$$

after τ_0 rounds (implied by lemma 1). Therefore, we can conclude that the solution $\boldsymbol{x}^0$ is indeed $\theta(\epsilon')$-feasible.

To establish the final assertion, observe that

$$\frac{\min_i \boldsymbol{C}_{i*}\boldsymbol{x}^0}{\max_i \boldsymbol{P}_{i*}\boldsymbol{x}^0} > 1 - 5\epsilon', \tag{3.26}$$

as we proved above. Let $\epsilon' \le 0.05$. From (3.26) and (3.6), we obtain

$$\min_i \boldsymbol{C}_{i*}\boldsymbol{x}^0 > 1 - 7\epsilon'.$$

Moreover, (3.26) together with (3.7) imply that

$$\max_i \boldsymbol{P}_{i*}\boldsymbol{x}^0 < 1 + 10\epsilon'.$$

Therefore, the solution is $10\epsilon'$-feasible. ■

3.3 Solving MPC-LPs (Optimization)

Up to this point, we have described how to use MPCSolver to obtain an ϵ-feasible solution to MPC-LPs. In this section, we consider the optimization version of MPC-LP and show, for given $\epsilon > 0$ and $1 > \eta > 0$, how to derive an (ϵ, η)-approximation, i.e., we also optimize the objective function while maintaining ϵ-feasibility. Our techniques are an adaptation of the techniques of Young (2001) to our setting. The key idea of Young (2001) is to push the objective into the constraints. In particular, we consider the following optimization problem:

$$\text{find } \lambda^* = \max\{\lambda : (\exists \boldsymbol{x}) \boldsymbol{P}\boldsymbol{x} \leq \boldsymbol{b}, \boldsymbol{C}\boldsymbol{x} \geq \boldsymbol{d}, \boldsymbol{w}^\top \boldsymbol{x} \geq \lambda\}.$$

Given the value of λ^*, we can determine $\boldsymbol{x}$ via solving a feasibility problem with the additional covering constraint $\boldsymbol{w}^\top \boldsymbol{x} \geq \lambda^*$. To obtain λ^*, we run a sequence of feasibility problems of form $\{\boldsymbol{P}\boldsymbol{x} \leq \boldsymbol{p}, \boldsymbol{C}\boldsymbol{x} \geq \boldsymbol{c}, \boldsymbol{w}^\top \boldsymbol{x} \geq \lambda\}$, where λ is determined via binary search. Our search strategy is close to the one of Young (2001), but we use a fixed error bound and directly compute an (ϵ, η)-approximation. The following lemma allows us to select an initial value for λ.

Lemma 6. *Let* $\lambda_{min} = \min\{\boldsymbol{w}^\top \boldsymbol{x} : \boldsymbol{C}\boldsymbol{x} \geq \boldsymbol{c}, \boldsymbol{x} \geq \boldsymbol{0}\}$ *and* $\lambda_{max} = \max\{\boldsymbol{w}^\top \boldsymbol{x} : \boldsymbol{P}\boldsymbol{x} \leq \boldsymbol{p}, \boldsymbol{x} \geq \boldsymbol{0}\}$. *Moreover, let* $\bar{\boldsymbol{x}}$ *be any feasible solution to MPC-LP, and set* $\lambda = \boldsymbol{w}^\top \bar{\boldsymbol{x}}$. *Then* $\frac{\lambda_{min}}{\lambda_{max}} \lambda^* \leq \lambda \leq \lambda^*$.

Proof. The right inequality holds since by definition $\lambda \leq \lambda^*$. Consider the minimization version of MPC-LP that includes only the covering constraints; we have $\boldsymbol{w}^\top \boldsymbol{x} \geq \lambda_{\text{min}}$. Since we ignore packing constraints, λ_{min} is a lower bound on the objective value of any feasible solution to MPC-LP, i.e., $\lambda \geq \lambda_{\text{min}}$. Similarly, consider the maximization version of MPC-LP that contains only the packing constraints. The optimal value of this problem is an upper bound on the objective value of any feasible solution including the optimal one, i.e., $\lambda^* \leq \lambda_{\text{max}}$. Thus, $\frac{\lambda_{\text{min}}}{\lambda_{\text{max}}} \lambda^* \leq \lambda_{\text{min}} \leq \lambda$ and the assertion follows. ■

In order to obtain an estimate of λ_{min} and λ_{max}, we use the distributed algorithm of Awerbuch and Khandekar (2009), which obtains a $(1+\epsilon)$-factor approximation for covering problems ($\hat{\lambda}_{\text{min}}$) and a $(1-\epsilon)$-factor approximation for packing problems ($\hat{\lambda}_{\text{max}}$). Since MPCSolver is a generalization of the algorithm of Awerbuch and Khandekar to MPC, we use our existing implementation and data partitioning; only parameters, update condition, and update rules need to be changed.

We assume that MPC-LP is feasible so that MPCSolver is able to obtain an ϵ-feasible solution. Denote by $\hat{\lambda}$ the objective realized by any such ϵ-feasible solution $\hat{\boldsymbol{x}}$. We distinguish between two cases:

Case $\hat{\lambda} > \lambda^*$.[2] This implies that $\hat{x}$ is already an (ϵ, η)-approximation.

Case $\hat{\lambda} \leq \lambda^*$. Let $\rho = \frac{(1-\epsilon)\hat{\lambda}_{\min}}{(1+\epsilon)\hat{\lambda}_{\max}}$. According to Lemma 6, we have $\rho\lambda^* \leq \hat{\lambda} \leq \lambda^*$. Our binary search algorithm proceeds as follows: We start with $\lambda_0 = \frac{\hat{\lambda}}{\rho}$, then $\rho \leq \frac{\lambda^*}{\lambda_0} \leq 1$. We then aim to find the integer l^* such that

$$(1-\eta)^{l^*+1} < \frac{\lambda^*}{\lambda_0} \leq (1-\eta)^{l^*}$$

via binary search (on l). Given any l, we run MPCSolver with $\lambda = (1-\eta)^l \lambda_0$ as lower bound on the objective. If we obtain an ϵ-feasible solution, then $l^* \leq l$. Otherwise, the problem is infeasible so $\lambda^* < \lambda = (1-\eta)^l \lambda_0$ and hence $l^* > l$. Note that if the problem instance is not exactly feasible (i.e., $\lambda > \lambda^*$) but ϵ-feasible, MPCSolver still might return an ϵ-feasible solution. The range used for binary search is given by $0 \leq l \leq \log_{1-\eta} \rho$ (since l^* must fall into this range). The solution obtained at $l = l^*$ is now an (ϵ, η)-approximation to the MPC-LP.

Lemma 7. *Given $\epsilon > 0$ and $1 > \eta > 0$, our binary search algorithm computes an (ϵ, η)-approximation to any feasible MPC-LP by solving at most*

$$O\left(\log\log\frac{\lambda_{max}}{\lambda_{min}} - \log\log\frac{1}{1-\eta}\right)$$

ϵ-feasibility problems.

In practice, we need to solve only few ϵ-feasibility problems (up to 7 in our experiments).

3.4 Parallelizing MPCSolver

Note that each iteration of MPCSolver essentially involves a set of matrix-vector products. Massively parallel computation of such products can be carried out efficiently using modern GPUs. Moreover, MPCSolver is straightforward to parallelize on a shared-memory architecture since both primal and dual variables are updated independently of each other. In what follows, we focus on in-memory processing in a shared-nothing setting and exploit shared-memory parallel processing when multiple threads are available on each compute node. Denote by s the number of compute nodes and by t the number of threads per node; the total number of threads in the cluster is thus given by $T = st$.

Recall from Algorithm 3 that, in each round, MPCSolver first computes the dual variables $\boldsymbol{y}$ and $\boldsymbol{z}$ associated with each packing and covering constraint, respectively

[2]This is possible since $\hat{\boldsymbol{x}}$ is ϵ-feasible.

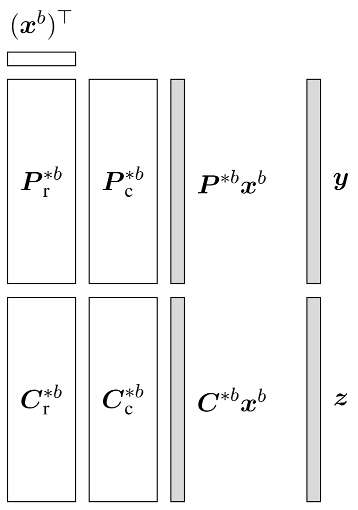

$(\boldsymbol{x}^b)^\top$
$\boldsymbol{P}^{*b}_{\mathrm{r}}$
$\boldsymbol{P}^{*b}_{\mathrm{c}}$
$\boldsymbol{P}^{*b}\boldsymbol{x}^b$
$\boldsymbol{y}$
$\boldsymbol{C}^{*b}_{\mathrm{r}}$
$\boldsymbol{C}^{*b}_{\mathrm{c}}$
$\boldsymbol{C}^{*b}\boldsymbol{x}^b$
$\boldsymbol{z}$

Figure 3.1: *Memory layout used on node b. Node-local data is shown in white and temporary data in gray.*

(lines 8 and 9). To distribute the computation, we conformingly partition $\boldsymbol{x}$ and matrices $\boldsymbol{P}$ and $\boldsymbol{C}$ column-wise across the compute nodes. Denote by $\boldsymbol{P}^{*b}$, $\boldsymbol{C}^{*b}$, and $\boldsymbol{x}^b$ the respective partitions stored at some node b ($1 \leq b \leq s$). Each node computes the products $\boldsymbol{P}^{*b}\boldsymbol{x}^b$ and $\boldsymbol{C}^{*b}\boldsymbol{x}^b$ independently and in parallel. The partial results are then aggregated (i.e., summed up) and subsequently broadcast across the compute cluster to obtain $\boldsymbol{Px}$ and $\boldsymbol{Cx}$, i.e., $\boldsymbol{Px} = \sum_{b=1}^{s} \boldsymbol{P}^{*b}\boldsymbol{x}^b$ and similarly $\boldsymbol{Cx} = \sum_{b=1}^{s} \boldsymbol{C}^{*b}\boldsymbol{x}^b$; see below for details on how this step can be implemented.

At this point, each node is able to compute the value of the duals and update the primal variables in partition $\boldsymbol{x}^b$ independently and in parallel (lines 10- 14). Since matrices $\boldsymbol{P}$ and $\boldsymbol{C}$ are accessed once row-wise (to compute the partial products) and once column-wise (to update the primal variables), we store at each node b two copies of $\boldsymbol{P}^{*b}$ and $\boldsymbol{C}^{*b}$ in memory: one in row-major (denoted by $\boldsymbol{P}^{*b}_{\mathrm{r}}$ and $\boldsymbol{C}^{*b}_{\mathrm{r}}$, respectively) and one in column-major order (denoted by $\boldsymbol{P}^{*b}_{\mathrm{c}}$ and $\boldsymbol{C}^{*b}_{\mathrm{c}}$, respectively). This memory layout is depicted in Figure 3.1. Note that the constraint matrices are partitioned only once before the algorithm starts.

Notice that computing $\boldsymbol{Px}$ and $\boldsymbol{Cx}$ involves similar operations as in Allreduce, which in general, consists of the following three steps: (1) every compute node provides a value, (2) values of all compute nodes are combined applying specified operations into one single value, and (3) the resulting value is distributed to all the compute nodes. Such an operation can be realized in multiple ways each with different performance efficiency.

In the most basic implementation, each node sends its partial products to a single node to which we refer as the *master node*; the actual aggregation and computation is then performed at the master node. Note that this simple implementation requires a small number of messages ($2s - 2$ in total) to be transmitted. However, this approach has a number of drawbacks when the number of constraints is large, i.e., when partial products contain a large number of elements. First, large messages need to be exchanged resulting in a high memory-bandwidth usage, which in turn, has a negative impact on the performance of the algorithm. Second, this realization might introduce network contention especially at the master node. Finally, since all computations are shifted to the master node, this single node implies a bottleneck.

An improved implementation, uses a *reduction tree*, i.e., a spanning tree over the nodes for reducing (i.e., accumulating) s partial products. Compared to the previous approach, the computational load is more balanced, while requiring the same number of message to be exchanged. However, it still suffers from high memory-bandwidth usage when the number of constraints is large. We can improve on this approach by further splitting the partial results $\boldsymbol{P}^{*b}\boldsymbol{x}^b$ and $\boldsymbol{C}^{*b}\boldsymbol{x}^b$ into s partitions and deploying s *different* reduction trees: one at each node and for each partition to obtain $\boldsymbol{P}\boldsymbol{x}$ and $\boldsymbol{C}\boldsymbol{x}$. Figure 3.2 illustrates an example for $s = 4$ and $t = 1$ ($\boldsymbol{M}^{*b}_{(S)}\boldsymbol{x}^b$ refers to partition $S \in \{A, B, C, D\}$ of the partial products $\boldsymbol{M}^{*b}\boldsymbol{x}^b$, where $\boldsymbol{M} \in \{\boldsymbol{P}, \boldsymbol{C}\}$). Compared to the previous implementations discussed above, the number of messages to be communicated is larger ($s \log s - s$ in total) but the size of each message is smaller ($\frac{1}{s}$ times). Thus, this realization distributes the computation evenly across cluster nodes while ensuring low latency.

Note that the performance efficiency of the implementations described above (based on a column-wise partitioning of the data matrices) depends on the problem structure; for instance, this realization can suffer from high communication costs when there are many more constraints than variables. In this case, a row-wise partitioning of the data matrices might be more beneficial. In general, however, the communication costs can be reduced significantly by exploiting the sparsity of the constraint matrices. Moreover, we can exploit shared-memory parallel processing when multiple threads are available on each compute node as follows. On each compute node b, we further partition matrices $\boldsymbol{P}^{*b}$ and $\boldsymbol{C}^{*b}$ column-wise and conformingly $\boldsymbol{x}^b$ across the available t threads, and apply the same aggregation as described above to obtain $\boldsymbol{P}^{*b}\boldsymbol{x}^b$ and $\boldsymbol{C}^{*b}\boldsymbol{x}^b$. Note that the aggregation step on each node can be performed with practically no communication overhead as the required data is node-local.

3.5 Implementing MPCSolver

We provide some optimizations that can be applied to speed up the performance of MPCSolver in practice. These practical techniques apply to general MPC problems. In Section 4.4, we consider instances of generalized bipartite matching problems and empirically show the effectiveness of these optimization techniques for such problems.

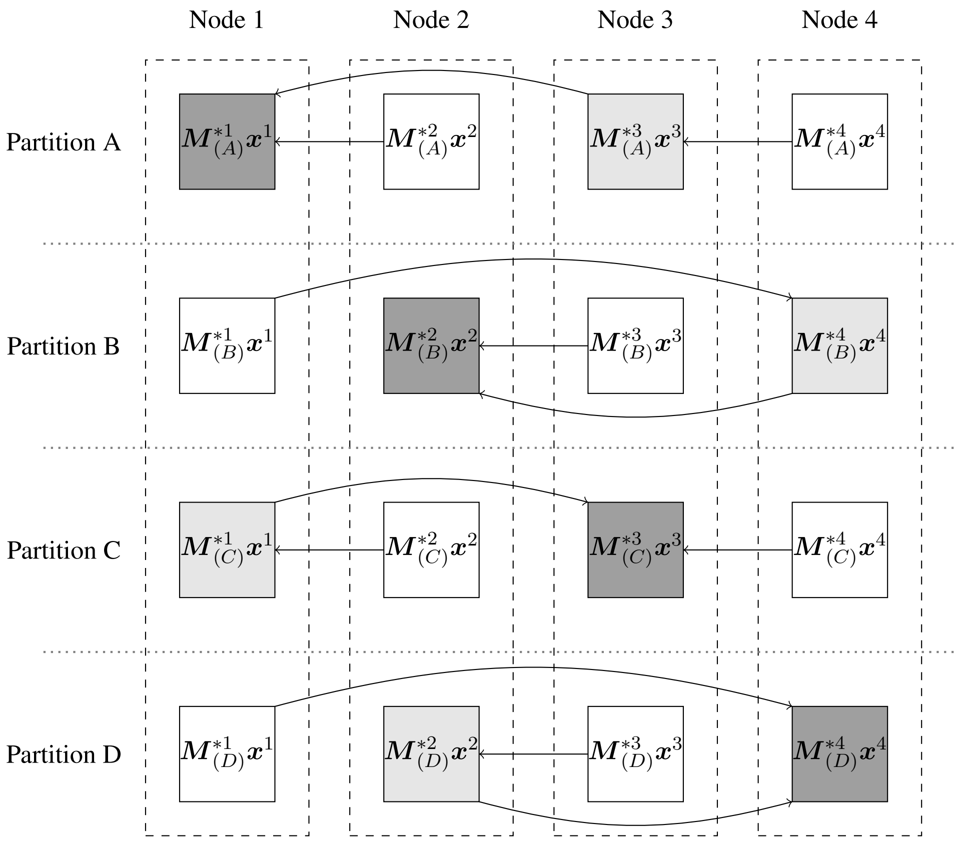


Figure 3.2: *Reduction trees and data flow for partitions A–D, $M = \{P, C\}$, $s = 4$. In reduction trees, leaf nodes are colored white, internal nodes light gray, and root nodes dark gray. Final results are aggregated at root nodes and then distributed to all the compute nodes (not shown in the figure).*

3.5.1 Starting Point

MPCSolver can start from any arbitrary initial point $x \in \Re^n_+$. However, in order to reduce the number of iterations to convergence and improve the efficiency of MPCSolver, we make use of the following special initializations: cold-start and warm-start. For a single feasibility problem, we apply the cold-start initialization, while the warm-start initialization is used when solving the optimization problems.

Cold-start. Denote by $\mathrm{nnz}(j)$ the largest number of non-zero entries in any row i of P in which $P_{ij} \neq 0$. We then set

$$x_j = \frac{1}{\mathrm{nnz}(j) \cdot \max_i P_{ij}} \text{ for } j \in [1, n]. \tag{3.27}$$

Note that this special initial point ensures that no packing constraint exceeds the right-hand side (i.e., 1) and thus all packing constraints are satisfied. Alternatively, one could select a random initial point or choose an initial point that satisfies all covering constraints. In such cases, however, some packing constraints might be violated by a large amount resulting in a high value of the dual variables $\boldsymbol{y}$. For this reason, we only consider the initialization as in (3.27) when solving a single feasibility problem.

Warm-start. Recall from Section 3.3 that we reduce the optimization problem into a small sequence of feasibility problems; solving each feasibility problem serves the purpose of computing a bound on the objective value. To facilitate our binary search, we first need to obtain λ by computing $\lambda_{\text{min}} = \min\{\boldsymbol{w}^\top \boldsymbol{x} : \boldsymbol{C}\boldsymbol{x} \geq \boldsymbol{c}, \boldsymbol{x} \geq \boldsymbol{0}\}$ and $\lambda_{\text{max}} = \max\{\boldsymbol{w}^\top \boldsymbol{x} : \boldsymbol{P}\boldsymbol{x} \leq \boldsymbol{p}, \boldsymbol{x} \geq \boldsymbol{0}\}$. We can maintain either of the solutions to the respective pure packing or covering problems and use it as the starting point for the first feasibility problem in the binary search procedure. In order to accelerate MPCSolver while solving each subsequent problem, we “warm-start” each feasibility problem using the solution obtained from solving the previous feasibility problem.

3.5.2 Adaptive Error Bounds

The internal error bound parameter ϵ' controls the quality of the final solution, but it also dramatically affects the number of rounds to convergence. Our experiments, however, suggest that setting ϵ' to values larger than $\frac{\epsilon}{10}$ has only a mild impact on the quality of the final solution but leads to faster convergence; in fact, our analysis of MPCSolver is somewhat loose so that our choice of $\epsilon' = \frac{\epsilon}{10}$ is conservative. We devise a simple adaptive method for choosing ϵ', which greatly improves the running time while ensuring ϵ-feasibility. We exploit the fact that the potential function can be efficiently evaluated at the end of each iteration and proceed as follows. Let $\epsilon \leq 0.5$. We first set ϵ' to some value larger than $\frac{\epsilon}{10}$ (e.g., 2). We keep running with value ϵ' as long as the potential improves significantly (e.g., by 0.001%). Whenever the potential stagnates, we decrease ϵ' (e.g., by 1%)—but not below $\frac{\epsilon}{10}$—and update parameters μ, α, β, and δ accordingly. This simple adaptive scheme worked extremely well in our experiments (see Section 4.4).

3.5.3 Adaptive Step Size

Recall from Section 3.2 that the step size parameter β controls how fast we explore the parameter space. From a theoretical standpoint, the preassigned value of the step size β (as defined in Algorithm 3) guarantees convergence, however, this choice may be quite conservative. In practice, we could choose β as to yield a maximum potential decrease (so that we can make more progress in each single round of

MPCSolver). To this end, we deploy a backtracking line search to select among a few values of β, e.g., $10^k\beta$ for $k = 0, 1, 2, 3$. In particular, we maintain k temporary variables $\boldsymbol{x}'_1, \dots \boldsymbol{x}'_k$, which we update separately in parallel using a different value of β and evaluate the resulting potential function for the corresponding value of β. The choice of β which yields the maximum potential decrease is selected for the actual updates to the primal variables $\boldsymbol{x}$. Note that, in this case, the cost of each round increases by factor k; the increased cost per round, however, might or might not pay off as it affects the number of round to convergence (see Section 4.4.2C).

3.5.4 Convergence Test

MPCSolver has converged as soon as one of the following criteria is met: (1) the solution is ϵ-feasible, (2) all variables remain unmodified, or (3) the number of rounds exceeds the poly-logarithmic bound of Theorem 1. If (2) or (3) hold and the solution is not ϵ-feasible, the MPC-LP is infeasible. In practice, the poly-logarithmic bound is usually too large to be useful (in particular when ϵ is small). A heuristic convergence test is to stop MPCSolver when the decrease in potential stagnates; e.g., when it falls below some threshold (say, 0.001%) in two consecutive rounds. For the adaptive scheme, we apply the heuristic convergence test only when ϵ' has been reduced to $\frac{\epsilon}{10}$. Even though the guarantees of Theorem 1 do not hold when this heuristic is used, our experiments suggest that the test is effective in practice.

3.5.5 Multiple Updates

Recall Algorithm 3. To update the primal variables $\boldsymbol{x}$ in any round t, MPCSolver performs two passes over the data: one pass to compute the products $\boldsymbol{Px}(t)$ and $\boldsymbol{Cx}(t)$ and to specify the values of the dual variables $\boldsymbol{y}(t)$ and $\boldsymbol{z}(t)$, and another pass to calculate the ratio $r_j(t) = \frac{\boldsymbol{P}_{*j}^\top \boldsymbol{y}(t)}{\boldsymbol{C}_{*j}^\top \boldsymbol{z}(t)}$ for all j. Depending whether the ratio $r_j(t)$ is smaller than $1-\alpha$ or larger than $1+\alpha$, the primal variable $\boldsymbol{x}_j$ is increased or decreased. If the ratio $r_j(t)$ lies in $(1-\alpha, 1+\alpha)$, then $\boldsymbol{x}_j$ remains unmodified.

First, note that the ratio r_j for some primal variable $\boldsymbol{x}_j$ might lie far away from $1-\alpha$ or $1+\alpha$, particularly during the initial rounds of MPCSolver. Next, note that our choice of the parameters of MPCSolver ensures an increase (resp. decrease) in the value of $\boldsymbol{P}_{*j}^\top \boldsymbol{y}$ (resp. $\boldsymbol{C}_{*j}^\top \boldsymbol{z}$) by a factor of at most $\frac{\alpha}{4}$ in any single round,[3] i.e.,

$$(1-\frac{\alpha}{4})\boldsymbol{P}_{*j}^\top \boldsymbol{y}(t) \leq \boldsymbol{P}_{*j}^\top \boldsymbol{y}(t+1) \leq (1+\frac{\alpha}{4})\boldsymbol{P}_{*j}^\top \boldsymbol{y}(t),$$
$$(1-\frac{\alpha}{4})\boldsymbol{C}_{*j}^\top \boldsymbol{z}(t) \leq \boldsymbol{C}_{*j}^\top \boldsymbol{z}(t+1) \leq (1+\frac{\alpha}{4})\boldsymbol{C}_{*j}^\top \boldsymbol{z}(t).$$

[3]These bounds on the maximum change of each packing and covering constraint hold under the assumption that $\boldsymbol{Px} \leq 3 \cdot \mathbb{1}$ and $\boldsymbol{Cx} \leq 3 \cdot \mathbb{1}$. In particular, this assumption is valid after the warm-up interval (see Lemma 1).

Let $f = (1 + \frac{\alpha}{4})/(1 - \frac{\alpha}{4})$. This implies that

$$\frac{r_j(t)}{f} \leq r_j(t+1) \leq f r_j(t),$$

and thus we can bound the ratio r_j in round $t+1$. In fact, due to our conservative choice of the parameters, it is unlikely that $r_j(t+1)$ decreases (increases) to $\frac{r_j(t)}{f}$ $(f r_j(t))$. Consequently, all variables x_j for which the ratio $r_j(t)$ is too small (resp. too large) compared to $1-\alpha$ (resp. $1+\alpha$) will be increased (resp. decreased) multiple times in several consecutive rounds. We can exploit this information to update those primal variables multiple times in each single round of MPCSolver in the hope to achieve faster convergence.

Based on the observation above, we devise a heuristic to update the primal variables multiple times in each round as follows. In any round t, we first obtain the ratio $r_j(t)$ for each individual primal variable $\boldsymbol{x}_j$. Next, we estimate the number of rounds we can update $\boldsymbol{x}_j$ and still make sure that r_j remains outside $(1-\alpha, 1+\alpha)$. To this end, we increase $\boldsymbol{x}_j(t)$, $\lceil \log_f \frac{1-\alpha}{r_j(t)} \rceil$ times if $r_j(t) \leq 1-\alpha$ and decrease $\boldsymbol{x}_j(t)$, $\lceil \log_f \frac{r_j(t)}{1+\alpha} \rceil$ times if $r_j(t) \geq 1+\alpha$.

We have not yet fully analyzed how this simple technique affects the convergence properties of MPCSolver; in our experiments, however, we observed a considerable reduction in the number of rounds and the running time until convergence; e.g., up to 70% reduction in the number of rounds and the running time (see Section 4.4.2C).

3.6 Experimental Study

We conducted a case study with instances of the generalized bipartite matching problems. In such problems, we aim to find a maximum-weight matching (subset of edges) of an undirected bipartite graph such that the degree (i.e., number of incident edges) of each node in the matching lies between a specified lower and upper bound. The LP relaxation of generalized bipartite matching is a mixed packing-covering LP. Here, we give an overview of the results; generalized bipartite matching problems are described in the next chapter and a detailed set of experiments are given in Section 4.4. We compared MPCSolver with Young's algorithm from Young (2001) as well as with a start-of-the-art general LP solver on real and synthetic datasets of varying sizes in terms of efficiency and scalability. We found that MPCSolver scales to very large datasets and is significantly (multiple orders of magnitude) faster than Young's algorithm. Moreover, MPCSolver is competitive to state-of-the-art LP solvers on moderately-sized datasets, but can handle much larger problem instances. Finally, MPCSolver can be readily implemented and well-suited for parallel processing on GPUs; it obtained up to 38.6x speedup compared to a sequential execution on small-scale problem instances.

3.7 Related Work

Lagrangian relaxation was one of the first methods proposed for solving (general) linear programs. Efficient approximation algorithms for MPC-LPs (and their special cases of PLPs) are mostly based on Lagrangian relaxation with LP decomposition (a.k.a. Lagrangian decomposition, multiplicative weights) methods which can serve as a useful alternative to interior-point or Simplex methods; see, e.g. Arora et al. (2012); Bienstock (2002); Todd (2002), for an overview. In contrast to exact LP algorithms, Lagrangian-relxation based techniques obtain an approximate solution: for any $\epsilon > 0$, they return solutions which are either approximately feasible (i.e., constraints are satisfied within a multiplicative factor of $1 \pm \epsilon$), or approximately optimal (i.e., whose costs are within a factor of $1 \pm \epsilon$ of the optimal cost OPT). However, compared to standard approaches, Lagrangian relaxation-based methods can be faster, are simpler, and more amenable to parallelization.

At a high level, the key principle in such algorithms is to select some of the constraints and replace them by a sum of smooth “penalties”, one for each selected constraint. The algorithm then iteratively constructs a solution while trying to maintain the remaining constraints and minimizing the increase in the sum of the penalties.

In some algorithms, the penalties have exponential dependency in the constraint violations as in Plotkin et al. (1995); Young (2001), whereas in the others the penalties have logarithmic form as in Diedrich and Jansen (2007b); Grigoriadis and Khachiyan (1994). Furthermore, some proposed methods rely on the first-order approximations of the change in the sum of penalties (e.g., Plotkin et al. (1995) and Young (2001)), while the others use second-order approximations (e.g., Bienstock (2002) and Bienstock and Iyengar (2004)).

MPCSolver shares some common features with Lagrangian-relaxation based methods; in particular, it utilizes exponential functions to penalize violations in the constraints and iteratively tries to minimize the sum of the penalties. This is achieved by minimizing a first-order approximation of the sum of the penalties.

In general, the running times of such iterative approximation algorithms increase as the error parameter ϵ gets small. For the algorithms that use a linear approximation of the changes in the sum of penalties, the running times grow at least quadratically in $\frac{1}{\epsilon}$ (times a polynomial in the input), while the algorithms that rely on a second-order approximation of the changes in the sum of penalties have a reduced dependency on $\frac{1}{\epsilon}$ from quadratic to linear, but at the expense of an increased dependency on the other parameters; compared to the algorithms of the first type, the algorithms of the second type require fewer iterations but more time per iteration.

Unfortunately, most existing Lagrangian-relaxation based algorithms for packing

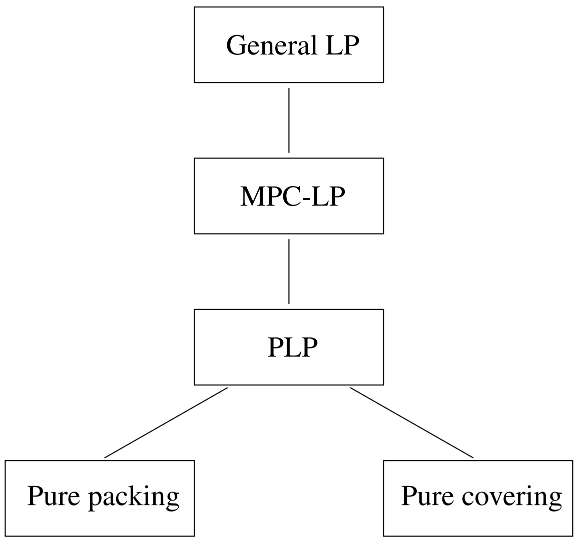


Figure 3.3: *Various types of LPs considered in Chapter 3*

and covering problems deal with sequential settings (e.g., Diedrich and Jansen (2007a;b); Khandekar (2004); Plotkin et al. (1995)) and lack efficiency when applied to real instances at massive scales. In the following, we focus on a parallel (shared-memory and shared-nothing) setting and review some existing approaches based on Lagrangian relaxation techniques for PLPs (pure packing or covering LPs), MPC-LPs, as well as general LPs. Figure 3.3 illustrates how these classes of LPs relate to each other. The problems are arranged in form of a tree. A problem defined in a node is a special case of the problem defined in the parent node. Unless other specified, m refers to the number of constraints, n refers to the number of variables, and N denotes the number of non-zero entries in the constraint matrix.

3.7.1 PLP Solvers

The first parallel approximation NC-algorithm for PLPs was presented in the seminal work of Luby and Nisan (1993). The complexity class NC includes the set of decision problems solvable in *poly-logarithmic* time using a parallel computer with a *polynomial* number of processors. Their algorithm obtains a $(1+\epsilon)$-factor approximation in time polynomial in $\frac{1}{\epsilon}$ and $\log N$, using $O(N)$ processors.

Some previous research deals with solving PLPs in a distributed setting in which multiple agents operate in a cooperative but uncoordinated manner. In this setting, an agent is associated with each of n variables and controls the value of the corresponding variable. Moreover, each agent has access to current values of only those constraints in which it has non-zero coefficients. The goal is to generate feasible solutions to PLPs using only information about local constraints (Linial 1992).

This study was initiated by Papadimitriou and Yannakakis (1993) who considered approximately solving pure packing LPs by distributed decision-makers based only on information from local constraints without performing any communication. Bartal et al. (2004) extended this model to allow local communication and presented a distributed approximation algorithm for PLPs, which trades off the quality of approximation achieved against the amount of communication performed. More specifically, their proposed algorithm obtains a logarithmic-factor approximation after a constant number of rounds and a $(1 + \epsilon)$-factor approximation after a logarithmic number of rounds. These results were further improved and generalized by Kuhn et al. (2006). For pure covering LPs, they showed a constant-factor approximation with high probability in $O(\log m)$ rounds. For pure packing LPs, they gave a $(1 + \epsilon)$-factor approximation algorithm requiring logarithmic message size, but the number of rounds depend on the input coefficients. Assuming that agents are allowed to send and receive messages of unbounded size, they achieved a constant-factor approximation in $O(\log m)$ rounds with high probability.

Koufogiannakis and Young (2009; 2011; 2013) presented efficient distributed approximation algorithms for a number of packing and covering combinatorial problems including PLPs. In particular, they gave a distributed Δ-factor approximation algorithm for a more general class of covering problems that includes pure covering LPs (with variable upper bounds) as a special case. Here Δ denotes the maximum number of variables in any constraint. Moreover, they proposed a distributed δ-factor approximation algorithm for pure packing LPs, where δ refers to the maximum number of constraints any variable appears in. Both algorithms are randomized and return solutions in $O(\log^2 m)$ rounds in expectation and with high probability.

Awerbuch and Khandekar (2009) proposed a distributed stateless approximation algorithm for PLPs. We refer to this algorithm as AK. AK is very simple and achieves a $(1 + \epsilon)$-factor approximation to any packing LP in time proportional to $\log(MN)$ and ϵ^{-5}. Here M denotes the width of the LP.[4] A similar algorithm works for covering LPs. MPCSolver is based on AK and shares with AK features like the use of an exponential potential function and gradient descent with multiplicative updates; key differences are that MPCSolver can handle mixed packing-covering problems and can start from an arbitrary initial point. Awerbuch et al. (2009) further generalized AK to the case of any *implicitly given* packing LP addressing problem instances with exponentially (or even infinitely) many variables.

3.7.2 MPC-LP Solvers

Young (2001) developed a first parallel (1+ϵ)-factor approximation algorithm for MPC-LPs, which obtains a solution that satisfies all constraints within a (1$\pm\epsilon$) factor

[4] In Awerbuch and Khandekar (2009), for pure packing LPs, the width is defined as $M = \min_{i,j} \frac{\boldsymbol{P}_{ij}}{\boldsymbol{p}_i \boldsymbol{w}_j}$; similarly for pure covering LPs.

in a poly-logarithmic number of rounds (in the input). Young's algorithm has a better theoretical bound than MPCSolver; in particular, its runtime is proportional to ϵ^{-4} (and poly-logarithmic in the size of the input), and is independent of the width of the problem (M). Young's algorithm initially sets all variables to sufficiently small values (all packing constraints satisfied), and then gradually increases but never decreases variables until either approximate feasibility is achieved or infeasibility is detected. In contrast to MPCSolver, variable increments cannot be undone. For this reason, Young's algorithm cannot start from arbitrary starting point and it does not support adaptive changes of the error bound, which greatly affect performance in practice (see Section 4.4). Both MPCSolver and Young's algorithm make use of exponential dual variables to penalize constraint violations.

3.7.3 General Solvers

The *alternating direction method of multipliers* (ADMM) is a general powerful framework for distributed convex optimization including linear programming. It was first introduced in the 1970s (Gabay and Mercier 1976; Glowinski 1975), with its origin dating back to the 1950s. ADMM combines ideas from *dual decomposition* and *augmented Lagrangian methods*. At a high level, it coordinates the solutions to small local subproblems to derive a solution to a large global problem; see Boyd et al. (2011) for an extensive literature survey and a review of the method.

Some previous research has studied the convergence properties of ADMM for the case of linear programming (see,e.g., Boley (2013); Boyd et al. (2011); Eckstein and Bertsekas (1990)); in particular, (Eckstein and Bertsekas 1990) showed that for general LPs, ADMM converges at a global linear rate, and more recently Boley (2013) established bounds on the local behavior of ADMM during the course of iterations for LPs containing equality constraints. The latter work proves that under normal conditions ADMM should pass through a number of stages each with different convergence rates. More specifically, ADMM takes constant steps in some stages but eventually achieves linear convergence in the vicinity of the optimum. The author also showed via examples that the linear convergence can still be very slow in practice. We postpone exploring the effectiveness of ADMM for MPC-LPs for future work. As a final remark note that MPCSolver is substantially simpler compared to ADMM. Nevertheless, ADMM can be applied to more general optimization problems.

Very recently, Sridhar et al. (2013) presented a parallel algorithm, termed Thetis, for approximately solving general LPs. The key idea of Thetis is to transform the given LP into a convex quadratic formulation and apply stochastic coordinate descent (SCD) to approximately solve the original LP. To facilitate fast convergence, Thetis benefits from an asynchronous parallel SCD implementation given by Liu et al. (2013) for solving the convex quadratic relaxation of the underlying LP. Note that Thetis applies to general LPs whereas MPCSolver is designed for MPC-LPs. Moreover, compared to MPCSolver, the runtime of Thetis grows as ϵ^{-2} and is thus less sensitive to approximation error ϵ, however, the overall worst-case complexity of Thetis depends polynomially in m and n. Overall, MPCSolver is significantly easier to implement.

3.8 Summary

MPC-LPs capture a simple yet expressive subclass of linear programs. They commonly arise as linear programming relaxations of a number of important combinatorial problems including various network design and generalized matching problems.

In this chapter, we have presented MPCSolver, a novel efficient distributed algorithm for approximately solving mixed packing-covering linear programs and proved its convergence via a full theoretical analysis. MPCSolver is inspired by, but more general than the work of Awerbuch and Khandekar (2009), which can handle either packing or covering constraints, but not both. MPCSolver requires a polylogarithmic number of passes over the input and is easy to implement and parallelize. In particular, it is well-suited for parallel processing on GPUs, in shared-memory architectures, or on small clusters of commodity nodes. We have first discussed how to solve mixed packing-covering feasibility problems and then showed how to solve the optimization version via a small sequence of feasibility problems. We have described implementation issues that are key to good performance in practice. In particular, we have discussed how to distribute data effectively across nodes so that communication costs are minimized. Moreover, we have provided several techniques to further improve the performance of MPCSolver. Finally, we have reviewed some existing approaches for solving MPC-LPs and related problems and have given an overview of the results of a case study with instances of generalized bipartite matching problems whose LP relaxations are MPC-LPs.

4 Generalized Bipartite Matching

In this chapter,[1] we are concerned with generalized bipartite matching problems. Such matching problems are ubiquitous; they appear in a wide variety of applications, including trade markets (Penn and Tennenholtz 2000), computational advertising (Bhalgat et al. 2012; Charles et al. 2010), and semi-supervised learning (Jebara et al. 2009).

Consider, for example, the problem of assigning DVDs to customers in online DVD rentals. Online video vendors such as Netflix or Amazon's Prime Instant Video allow customers to specify which DVDs they would like to rent. Due to the limited number of physical available DVDs, customers are encouraged to provide a large ranked list of their favorite movies; the online video vendor then automatically selects which DVDs to ship to customers based on both customers' preferences and availability. Moreover, customers are recommended movies that they might be interested in; a good recommender engine should consider availability to reduce customer waiting times for accepted recommendations.

The problem of assigning DVDs to customers is non-trivial: For shipping, we want to make sure that users are sent movies ranked as high on their lists as possible, while at the same time maintaining fairness, i.e., every user should be provided with a sufficient number of DVDs without exceeding the user's DVD budget. Similarly, for recommendation, we want to recommend a few DVDs to each user such that the

[1]Parts of the material in this chapter have been jointly developed with Rainer Gemulla and Mauro Sozio. The chapter is based on Makari et al. (2013) and Makari and Gemulla (2013). The copyright of Makari et al. (2013) is held by VLDB Endowment; the original publication is available at http://dl.acm.org/citation.cfm?id=2536362. The copyright of Makari and Gemulla (2013) is held by NIPS; the original publication is available at http://biglearn.org/2013/files/papers/biglearning2013_submission_14.pdf.

user is likely to be interested in his recommendations and, in case of acceptance, the recommended DVD is (likely to be) physically available.

As another example, consider the problem of recommending partners for relationship to members of an online matchmaking service. For instance, eHarmony [2]—an online matchmaking company, which connects compatible members likely to enjoy long-term relationships—predicts the compatibility between singles based on a detailed questionnaire filled out by its users about their personality, values, attitudes, beliefs, etc. The goal is to deliver high quality recommendations for partner relationships to individual singles (so to increase the likelihood that the suggestions lead to marriage), while preserving fairness, i.e., each single should receive sufficient suggestions without overloading the user with recommendations.

We can naturally model problems like the ones above as *generalized bipartite matching* problems. In such a problem, we are given a set of users and a set of items. There is an edge between a user u and an item v if u is interested in v; each edge is associated with a positive weight that captures the degree of interest of u in v. Furthermore, each user u and each item v is associated with a lower and upper bound on the number of edges adjacent to u and v that can participate in the matching (the matching problem is "generalized" due to the presence of these bounds). Our goal is to find a matching—i.e., subset of the edges—such that all lower- and upper-bound constraints are satisfied and the total weight of the edges participating in the matching is maximized. This problem is also known in the literature as the *maximum weight degree-constrained subgraph* problem, while the special case with only upper bounds is known as *maximum weight b-matching*.

The problems stated above can be solved in polynomial time via traditional max-flow techniques given by Ahuja et al. (1993), linear programming solvers such as Gurobi Optimization (2013), or using the combinatorial algorithm developed in Gabow and Tarjan (1989). Unfortunately, these approaches cannot cope with the massive scale of real-world problem instances which may involve millions of users and items and billions of edges; for instance, Netflix offers tens of thousands of movies for rental to its more than 20M customers. Similarly, eHarmony has currently more than 33M members to whom it provides recommendation for long-term partnership.

In this chapter, we propose a scalable distributed approximation algorithm for large-scale generalized bipartite matching. Though several scalable algorithms for b-matching have been proposed in the literature, our algorithm is the first distributed (i.e., shared-nothing) algorithm for generalized bipartite matching problems, and it can cope with massive real-world instances. Our algorithm can readily be adapted to capture more complex matching problems with additional constraints; e.g., in

[2] `www.eHarmony.com`

the case of DVD recommendation, one may enforce that users are recommended at most one movie per genre to encourage diversity.

Our algorithm is randomized and produces an approximate solution to the matching problem with strong approximation guarantees. Given small error bounds ϵ and η, we compute a solution that is within a factor of $(1-\epsilon)(1-\eta)$ of the optimal solution in expectation, and satisfies the lower- and upper-bound constraints within a factor of $1-\epsilon$ and $1+\epsilon$, respectively. Our method is based on linear programming and consists of two phases: (1) Compute an approximate fractional solution to the LP relaxation of the integer linear program formulation of the matching problem, and (2) round the fractional solution to obtain an integral solution. In phase 1, we utilize MPCSolver from Chapter 3 to compute an approximate fractional solution in a distributed setting. Next, we combine the sequential rounding scheme of Gandhi et al. (2006) with the recent work from Lattanzi et al. (2011) on “filtering” in the MapReduce setting to obtain an efficient distributed rounding algorithm called DDRounding in phase 2.

Finally, we experimentally compare the efficiency and scalability of our algorithms to existing alternatives on both real-world and synthetic datasets of varying sizes. Our experiments indicate that our algorithms can handle significantly larger problem instances than LP solvers, and are orders of magnitude faster than existing alternative distributed algorithms.

The remainder of this chapter is organized as follows: We formally define the generalized bipartite matching problem in Section 4.1. In Section 4.2, we first show how to implement MPCSolver to obtain an approximate fractional solution for GBM in a distributed setting. We then provide details on how to efficiently round the fractional solution to an integral solution in a distributed environment. We summarize related work in Section 4.3 and finally give results of our extensive experimental study in Section 4.4. Throughout this chapter, we use the notation from Appendix A.

4.1 Problem Definition

We are given an undirected bipartite graph $G=(U,V,E)$, where U represents a set of users, V represents a set of items, and there is an edge $(u,v)\in E$ if user u is interested in item v. Each edge (u,v) is associated with a *positive weight* $w(u,v)$ measuring the degree of interest of u in v. For each user and item, we are additionally given a *lower bound* $l(v)$ and an *upper bound* $b(v)$, where $v\in U\cup V$. The bounds constrain the degree of vertex v in the solution; e.g., bounds on the number of recommendations for a user or the availability of a movie. More formally, denote by $\bar{E}\subseteq E$ a subset of the edges in G, and by $\bar{E}_v$ the set of edges incident to vertex v in subgraph $(U,V,\bar{E})$. We say that $\bar{E}$ is *feasible* if $l(v)\leq|\bar{E}_v|\leq b(v)$ for all

$v \in U \cup V$, i.e., all lower- and upper-bound constraints are met. An instance of our problem is feasible if there exists a feasible $\bar{E}$. The *generalized bipartite matching* (GBM) problem is to determine the best feasible matching between users and movies, i.e., we seek to maximize the *objective function* $f(\bar{E}) = \sum_{(u,v)\in\bar{E}} w(u,v)$ over all feasible $\bar{E} \subseteq E$.

Although GBM can be solved in polynomial time via maximum-flow techniques or integer linear programs, available solvers do not scale to the large problem instances that occur in practice, i.e., instances with millions of users and movies, and potentially billions of edges. In this chapter, we consider an approximate variant of GBM in which we seek for a "good" (but not necessarily optimal) solution and additionally allow for a small violation of lower- and upper-bound constraints. This "relaxation" allows us to develop a scalable distributed approximation algorithm for GBM. In particular, denote by $\epsilon > 0$ a small *error bound*. We say that $\bar{E}$ is *ϵ-feasible* if lower- and upper-bound constraints are violated by at most a factor of $1-\epsilon$ and $1+\epsilon$ (up to rounding), respectively. For $1 > \eta > 0$, an ϵ-feasible solution $\bar{E}$ is called an *(ϵ,η)-approximation* of the GBM if its objective value is within a factor of $(1-\epsilon)(1-\eta)$ of the optimal solution OPT to GBM. The relaxed problem is stated formally as follows.

Problem 1 (GBM$_{\epsilon,\eta}$)**.** *We are given $\epsilon > 0$, $1 > \eta > 0$, and a GBM in terms of an undirected bipartite graph $G = (U, V, E)$, a weight function $w : E \to \Re^+$, and lower- and upper-bound functions $l : U \cup V \to \mathbb{N}$ and $b : U \cup V \to \mathbb{N}$, respectively. If the GBM is feasible with optimum objective* OPT*, find a subset $\bar{E} \subseteq E$ such that*

$$\lfloor (1-\epsilon)l(v) \rfloor \le |\bar{E}_v| \le \lceil (1+\epsilon)b(v) \rceil$$

for each $v \in U \cup V$, and

$$\sum_{(u,v)\in\bar{E}} w(u,v) \ge (1-\epsilon)(1-\eta)\mathrm{OPT}$$

in expectation. If the GBM is infeasible, return any edge set $\bar{E} \subseteq E$.

In Section 4.2, we develop a randomized algorithm to efficiently solve large problem instances of GBM$_{\epsilon,\eta}$ in a distributed environment. Our algorithm produces a solution which has an objective value at least $(1-\epsilon)(1-\eta)\mathrm{OPT}$ in expectation and is guaranteed to be ϵ-feasible. Although we do not make any formal claims here, our experiment suggests that the objective is also concentrated around the expected value, i.e., we obtained an objective that was close to or better than $(1-\epsilon)(1-\eta)\mathrm{OPT}$ in every single run of our algorithm.

ILP Formulation and LP Relaxation

The GBM problem can be formulated as an integer linear program (ILP) as follows. For each edge $e \in E$, we introduce a binary variable $x_e \in \{0, 1\}$; $x_e = 1$ if e is included in the solution, otherwise $x_e = 0$. The ILP is then given by

$$\begin{array}{lrll} \max & \sum_{e \in E} w(e) x_e & & \\ \text{s.t.} & \sum_{e \in E_v} x_e \le b(v) & \forall v \in U \cup V, & \\ & \sum_{e \in E_v} x_e \ge l(v) & \forall v \in U \cup V, & \\ & x_e \in \{0,1\} & \forall e \in E, & \end{array} \tag{GBM-ILP}$$

where E_v denotes the set of edges adjacent to vertex v in G. Here the first two constraints express the upper- and lower-bound constraints, respectively. Since the constraint matrix of GBM-ILP (see Section 4.2.1) is totally unimodular and the right-hand sides are all integer-valued, an optimum solution can be computed in polynomial time, e.g., using the algorithm of Ahuja et al. (1993). However, our experiments in Section 4.4 suggest that state-of-the-art LP solvers cannot handle very large problem instances.

We obtain the so-called *LP relaxation*, denoted GBM-LP, by allowing x_e to be fractional, i.e., to take any value in $[0, 1]$. As described in Section 4.2, we make use of the LP relaxation in our algorithms in that we first compute an (ϵ, η)-approximation of GBM-LP, and then round the solution (sensibly) to obtain an integral one. Note that GBM-LP belongs to the class of mixed packing-covering LPs (see Section 3.1). The mapping of GBM-LP to MPC-LP is described in Section 4.2.1.

4.2 Algorithms

As mentioned in Section 4.1, we solve GBM$_{\epsilon,\eta}$ by computing a solution to the corresponding LP relaxation, which is subsequently rounded to an integral solution. To this end, we first utilize MPCSolver to compute an ϵ-feasible solution (see Sections 3.2 and 3.3). The special structure of GBM-LP makes MPCSolver very attractive; this can be exploited to reduce communication costs (see Section 4.2.1). We then combine ideas from randomized rounding presented in Gandhi et al. (2006) with “filtering” techniques for MapReduce given by Lattanzi et al. (2011) to obtain an integral solution (Section 4.2.2).

4.2.1 MPCSolver for GBM

Recall the definition of an MPC-LP from Section 3.1 and consider the LP relaxation GBM-LP of a given GBM instance. Then $n = |E|$ denotes the number of edges, $\boldsymbol{x} = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n)$ the edge variables, and $\boldsymbol{w}$ the corresponding edge weights. Set $r = |U \cup V|$ and denote by $\boldsymbol{M}$ the $r \times n$ incidence matrix of bipartite graph G. Then

$\boldsymbol{C}_{r\times n} = \boldsymbol{M}$ (i.e., $k = r$), $\boldsymbol{c}$ is the vector of lower bounds, $\boldsymbol{P}_{(r+n)\times n} = (\boldsymbol{M}^\top\ \boldsymbol{I}_n)^\top$ (i.e., $m = r + n$), where $\boldsymbol{I}_n$ is the $n \times n$ identity matrix, and $\boldsymbol{p}$ a vector of r upper bounds followed by n ones. Thus, $\boldsymbol{P}$ handles both the upper-bound constraints and the constraint that $\boldsymbol{x}_j \leq 1$ for all j. Note that matrices $\boldsymbol{C}$ and $\boldsymbol{P}$ are usually sparse.

Recall Section 3.4 where we discussed how to parallelize MPCSolver in general. Denote by $\boldsymbol{y_M}$ and $\boldsymbol{y_I}$ the subset of the dual variables corresponding to upper-bound constraints and at-most-one constraints, respectively. For the computation of both $\boldsymbol{y_M}$ and $\boldsymbol{z}$ only product $\boldsymbol{Mx}$ is needed (according to (3.2) and (3.3), $(\boldsymbol{y_I})_i = \exp\left[\mu\left((\boldsymbol{Mx})_i - 1\right)\right]$ and $\boldsymbol{z}_i = \exp\left[\mu\left(1 - (\boldsymbol{Mx})_i\right)\right]$). To obtain $\boldsymbol{Mx}$, we communicate partial products $\boldsymbol{M}^{*b}\boldsymbol{x}^b$ at node b in every round. However, for the computation of $\boldsymbol{y_I}$ no communication is needed; these variables can be computed locally at each node. This is advantageous since the size of $\boldsymbol{y_I}$ is linear in n (large) but the size of $\boldsymbol{Mx}$ (as well as $\boldsymbol{y_M}$ and $\boldsymbol{z}$) is linear in r (small). Note that all communicated values are vectors of length r, which makes MPCSolver especially attractive for GBM-LP.

4.2.2 Obtaining an Integral Solution

In the following, we propose a distributed algorithm that takes as input a fractional solution $\boldsymbol{x}$, which is an (ϵ, η)-approximation of the GBM-LP and generates an ϵ-feasible integral solution X such that $\mathrm{E}\left[\,\boldsymbol{w}^\top X\,\right] = \boldsymbol{w}^\top\boldsymbol{x}$. In particular, our algorithm ensures that:

1. $\mathrm{E}\left[\,\boldsymbol{w}^\top X\,\right] = \boldsymbol{w}^\top\boldsymbol{x}$,
2. all edge variables $X_e \in \{\,0, 1\,\}$,
3. when $\boldsymbol{P}_{i*}\boldsymbol{x} \leq (1+\epsilon)\boldsymbol{p}_i$, then $\boldsymbol{P}_{i*}X \leq \lceil(1+\epsilon)\boldsymbol{p}_i\rceil$, and
4. when $\boldsymbol{C}_{i*}\boldsymbol{x} \geq (1-\epsilon)\boldsymbol{c}_i$, then $\boldsymbol{C}_{i*}X \geq \lfloor(1-\epsilon)\boldsymbol{c}_i\rfloor$.

A naïve approach is to use an independent rounding scheme, which independently rounds every edge variable such that $P(X_e = 1) = x_e$. Such a scheme, however, may not (and often does not) lead to an ϵ-feasible solution. To see this, consider the (packing) constraint $X_1 + X_2 + X_3 \leq \lceil 1+\epsilon\rceil$ and the fractional solution $x_1 = 0.2$, $x_2 = 0.3$, $x_3 = 0.5 + \epsilon$ for $1 > \epsilon > 0$; independent rounding sets $X_1 = X_2 = X_3 = 1$ with non-zero probability but this solution is not ϵ-feasible. We thus need some form of dependent rounding, in which variables are rounded dependent on the rounding of other variables.

Some methods have been proposed in the literature to generate randomized roundings for fractional variables defined on the edges of bipartite graphs that preserve certain dependencies between the edge variables, e.g., so to satisfy cardinality constraints on the degree of each individual vertex. Call an edge *integral* if $x_e \in \{0,1\}$ and *fractional* if $x_e \in (0,1)$. We refer to the sum of fractional edge variables incident to vertex $v \in U \cup V$, i.e., $\sum_{e \in E_v \wedge x_e \in (0,1)} x_e$, as the *fractional degree* of v. Similarly, we define the *integral degree* of v to be $\sum_{e \in E_v \wedge x_e \in \{0,1\}} x_e$. The *bipartite edge variable rounding* asks for a rounding of x to X that guarantees marginal preservation—i.e., $\mathrm{E}\,[\,X_e\,] = x_e$, addressing (1)—and degree preservation, i.e. $\lfloor \sum_{e \in E_v} x_e \rfloor \leq \sum_{e \in E_v} X_e \leq \lceil \sum_{e \in E_v} x_e \rceil$ for all $v \in U \cup V$, addressing (3)+(4).

Gandhi et al. (2006) proposed a randomized sequential rounding scheme based on the deterministic *pipage rounding* given by Ageev and Sviridenko (2004) to solve the bipartite edge variable rounding problem. We refer to the algorithm due to Gandhi et al. as DRounding. In the sequel, we first discuss the key ideas of both the pipage rounding (Section 4.2.2A) and DRounding (Section 4.2.2B). Next, we show how to adapt DRounding to a distributed environment and present our distributed rounding algorithm, termed DDRounding.

A. Pipage Rounding

Denote by F the set of fractional edges and let G' be the graph spanned by F, i.e., $G' = (U \cup V, F)$. The pipage rounding is based on the following key observation. Suppose that G' contains a simple cycle[3] $C = (e_1, \ldots, e_{2k})$. Then, there exists an $\epsilon \in (0,1)$ such that alternately adding and subtracting ϵ to/from the edge values in C modifies at least one edge value in C to 0 or 1 while preserving the fractional degree of each individual vertex and keeping the remaining edge variables in $[0,1]$. Similarly, we can modify the edge values of any maximal path P. Note that each vertex is an end-vertex of a maximal path at most once. In this case, the fractional degree of the two end vertices of P may change, but never exceed their floors and ceilings; the fractional degrees of the remaining vertices of P remain unchanged. Thus, performing this procedure iteratively up to m times moves all fractional edges to integral values.

B. Dependent Rounding (DRounding)

DRounding can be seen as a randomized version of the deterministic pipage rounding algorithm. The algorithm additionally satisfies some negative correlation properties, which imply the usual Chernoff bounds; for details see Gandhi et al. (2006). DRounding is summarized as Algorithm 4. In the following, we briefly describe the algorithm. Similar to the pipage rounding, DRounding iteratively rounds fractional edges until all edges become integral. Each iteration of DRounding finds either

[3]A simple cycle is a cycle with no repeated vertices or edges other than the starting and ending vertices.

Algorithm 4 DRounding (Gandhi et al. 2006)

```
Require: $U$, $V$, $E$, solution $x$ of GBM$_\epsilon$-LP
1: $I_0 \leftarrow \{ (e, x_e) : e \in E, x_e \in \{0,1\} \}$        // integral edges
2: $F_0 \leftarrow \{ (e, x_e) : e \in E, 0 < x_e < 1 \}$        // fractional edges
3: for $i \in 1, 2, \ldots$ do
4:    if $F_{i-1} = \emptyset$ then        // all edges integral?
5:       return $\bigcup_{j=0}^{i} I_j$        // output integral solution
6:    else
7:       Find a simple cycle or maximal path $P = (e_1, \ldots, e_k)$ and partition $P$
         into two matchings $M_1$ and $M_2$
8:       Let $\alpha = \min\{\gamma > 0 : (\exists e \in M_1 : x_e + \gamma = 1) \bigvee (\exists e \in M_2 : x_e - \gamma = 0)\}$
         and $\beta = \min\{\gamma > 0 : (\exists e \in M_1 : x_e - \gamma = 0) \bigvee (\exists e \in M_2 : x_e + \gamma = 1)\}$

9:       Set $x_e = x_e + \alpha, \forall e \in M_1$ and $x_e = x_e - \alpha, \forall e \in M_2$ with prob. $\frac{\beta}{\alpha+\beta}$
10:      Set $x_e = x_e - \beta, \forall e \in M_1$ and $x_e = x_e + \beta, \forall e \in M_2$ with prob. $\frac{\alpha}{\alpha+\beta}$
11:      $I_i \leftarrow \{ (e, X_e) \in E : X_e \in \{0,1\} \}$
12:      $F_i \leftarrow F_i \setminus I_i$
```

a cycle or a maximal path in the subgraph spanned by the remaining fractional edges. Once such a cycle or path is found, typically one (but possibly more) of its edges are rounded. In particular, an iteration of the algorithm proceed as follows: As before, denote by F the current set of fractional edges. If $F = \emptyset$, we are done and the algorithm terminates. Otherwise, find a simple cycle or maximal path P in the subgraph spanned by F, and partition P into two matchings M_1 and M_2. Denote by α (resp. β) the minimum amount we can add to/subtract from the edges variables in M_1 (resp. M_2) such that at least one edge in M_1 (resp. M_2) becomes integral (line 8). Now, select α with probability $\frac{\beta}{\alpha+\beta}$ and β with the complementary probability $\frac{\alpha}{\alpha+\beta}$ to modify the edge variables in M_1 and M_2 (lines 9 and 10). Note that the rounding of a cycle or maximal path does not require global information, but only the values of the edge variables on the cycle or maximal path, respectively. Thus, it suffices to maintain the set F of the fractional edges, which shrinks after every rounding step. Since we can detect and process a cycle in $O(r)$ time (via depth-first-search), and since every iteration rounds at least one fractional edge, the total running time of DRounding is $O(rn)$, where as before $r = |U \cup V|$ and $n = |E|$.

C. Distributed Dependent Rounding (DDRounding)

In what follows, we show how to adapt DRounding to a distributed environment. Our distributed algorithm, called DDRounding, is inspired by recent work of Lattanzi et al. (2011) on "filtering" in the MapReduce framework and, in particular, the filtering algorithm for computing minimum spanning trees. We exploit the fact that the approximation guarantees of DRounding do not depend on the order in which cycles or maximal paths are processed; we are thus free to choose an order that facilitates distributed processing. DDRounding is summarized as Algorithm 5. In each iteration i, the algorithm checks whether the set F_{i-1} of remaining fractional edges is small. If so, we run DRounding on F_{i-1} and output the solution. If not, we evenly distribute F_{i-1} across s_i compute nodes, where s_i is chosen carefully (see below). In parallel, each node then runs on its local partition a version of DRounding that rounds cycles (can be detected locally) but not maximal paths (cannot be detected locally) (line 11). The reason why we only round cycles but not maximal paths from local partitions is that any cycle in a local partition of some node is also a valid cycle in the full graph, whereas a maximal path in a local partition is not necessarily a maximal path in the global graph, as it could potentially be extended by adding edges that are present in the full graph but not in the local partition, to its end vertices. After a local partition is rounded, it contains at most $r-1$ remaining fractional edges (since cycles have been rounded so that the remaining fractional edges form a forest). We then (conceptually) merge the remaining fractional edges across partitions to construct the set F_i for the next iteration.

Runtime and memory. The properties of DDRounding are similar to those of the filtering techniques of Lattanzi et al. (2011); we describe them briefly here. Assume without loss of generality that $|E_v| \geq 1$ for all $v \in U \cup V$, i.e., every user and every item has at least one incident edge. Also assume that there are more edges than vertices, i.e., $n = r^{1+c}$ for some $0 < c \leq 1$. Assume that each compute node has insufficient memory to store the whole graph; in particular, each node can store $\nu = O(r^{1+\gamma})$ edges for $0 < \gamma < 1$ and $\gamma < c$. Further assume that the input fits in the aggregate memory of all nodes, i.e., there are $\Theta(\frac{n}{\nu}) = \Theta(r^{c-\gamma})$ nodes available. Note that these assumptions imply that the set of vertices can be stored on a single node, while the set of edges cannot. We thus model the situation where there are many more edges than nodes, as it is often the case in practice. Set $n_{i-1} = |F_{i-1}|$. In iteration i, we use $s_i = \Theta(\frac{n_{i-1}}{\nu})$ nodes so that each node stores $O(\nu)$ edges; memory constraints are thus preserved. Now observe that by the arguments above $n_i \leq s_i(r-1) = O(\frac{n_{i-1}}{r^\gamma})$. Since $n_0 \leq n = O(r^{1+c})$, we conclude that $n_i = O(\frac{r^{1+c}}{r^{i\gamma}})$. The algorithm terminates as soon as $n_i = O(\nu)$, i.e., after $O(\lceil \frac{c}{\gamma} \rceil)$ iterations. Since every node runs a (less expensive variant of) DRounding independently and in parallel, each iteration has time complexity $O(\nu r) = O(r^{2+\gamma})$. The overall time complexity is thus $O(r^{2+\gamma}\lceil \frac{c}{\gamma} \rceil)$.

Quality of approximation. ϵ-feasibility of X follows directly from the degree preservation property of DRounding. The marginal preservation property implies

Algorithm 5 DDRounding

```
Require: $U$, $V$, $E$, solution $x$ of GBM$_\epsilon$-LP
 1: $I_0 \leftarrow \{ (e, x_e) : e \in E, x_e \in \{ 0, 1 \} \}$                 // integral edges
 2: $F_0 \leftarrow \{ (e, x_e) : e \in E, 0 < x_e < 1 \}$                      // fractional edges
 3: for $i \in 1, 2, \dots$ do
 4:   if $|F_{i-1}| < \nu$ then                                               // few fractional edges?
 5:     $I_i \leftarrow$ DRounding$(U, V, F_{i-1})$                             // sequential rounding
 6:     return $\bigcup_{j=0}^{i} I_j$                                       // output integral solution
 7:   else
 8:     $s_i \leftarrow \Theta(|F_{i-1}|/\nu)$                                // number of compute nodes
 9:     Partition $F_{i-1}$ evenly across $s$ compute nodes to obtain partitions
        $F_{i-1,1}, \dots, F_{i-1,s_i}$
10:     for $j \in \{ 1, \dots, s_i \}$ do                                    // in parallel
11:       $T \leftarrow$ RemoveCyclesWithDRounding$(U, V, F_{i-1,j})$
12:       $I_{i,j} \leftarrow \{ (e, X_e) \in T : X_e \in \{ 0, 1 \} \}$
13:       $F_{i,j} \leftarrow T \setminus I_{i,j}$
14:     $I_i = \bigcup_{j=1}^{s_i} I_{i,j}$                                    // newly obtained integral edges
15:     $F_i = \bigcup_{j=1}^{s_i} F_{i,j}$                                    // remaining fractional edges
```

that the objective function is within $(1-\epsilon)(1-\eta)$ of the optimum in expectation, since $\mathrm{E}[\,\boldsymbol{w}^\top X\,] = \boldsymbol{w}^\top \boldsymbol{x}$. Note that $S = \boldsymbol{w}^\top X$ is a bounded random variable (both from below and above). Standard arguments then show that if we round sufficiently often, then $\boldsymbol{w}^\top X$ is close to $\boldsymbol{w}^\top \boldsymbol{x}$ with high probability. The number of required rounding steps depend on the problem though (i.e., on the value of the optimal solution). In our experiments, we found that even a single run of DDRounding produces results close to or even above $\boldsymbol{w}^\top \boldsymbol{x}$.

Implementing DDRounding. Recall that, after MPCSolver has finished, each node j already stores locally a subset $F_{0,j} = \boldsymbol{x}^j$ of the primal variables, where $|F_{0,j}| \leq \lceil \frac{n}{s} \rceil$. We thus set $s_1 = s$ so that there is no need for any data redistribution in the first iteration of DDRounding ($i = 1$). When available, we use multiple threads on each node to remove cycles with DRounding. In each subsequent iteration $i > 1$, we halve the number of available nodes so that $s_i = \lceil \frac{s_{i-1}}{2} \rceil$. We thus need to communicate only half of the remaining fractional edges , i.e., the remaining fractional edges of every second node. This procedure balances communication cost evenly. Also note that iterations $i > 1$ are faster because every node processes at most $2r + 2$ edges; the bulk of the work is performed in the first iteration.

Implementing DRounding. Cycle detection using depth-first-search (DFS) can be implemented efficiently as follows. We start by selecting an arbitrary root node $v_0 \in U \cup V$ and perform DFS starting from v_0. Whenever we find a cycle $C = (vv_1 \cdots v_i v_{i+1} \cdots v_l v)$, some of its corresponding edges are rounded. Suppose that by doing so, only edge (v_i, v_{i+1}) becomes integral.[4] This removes an edge of the current path of DFS, i.e., we are in an invalid state. To avoid restarting DFS from scratch, we decompose C into two paths $C_1 = (vv_1 \cdots v_i)$ and, in reverse order, $C_2 = (vv_l \cdots v_{i+1})$. We replace the portion of the DFS stack that corresponds to C by C_1 or C_2, whichever is longer. To the extent possible, this allows us to avoid reprocessing the same paths over and over again. Moreover, once a node is fully processed (i.e., there are no more cycles involving this node), we mark it so that we never need to visit this node again. Both optimizations significantly improved the efficiency of our implementation of DRounding.

4.3 Related Work

The classical optimization task of finding an assignment of entities to users under a given set of constraints has been extensively studied in various domains of computer science. Entities could be items in an auction (Penn and Tennenholtz 2000), advertisements (Charles et al. 2010), scientific papers (Garg et al. 2010), social content (Morales et al. 2011), or multimedia items as in our case. Additionally, matching problems (in particular weighted b-matching) have been shown to be a useful tool in a wide variety of machine learning tasks, including semi-supervised learning (Jebara et al. 2009), spectral clustering (Jebara and Shchogolev 2006), graph embedding (Shaw and Jebara 2007), and manifold learning (Shaw and Jebara 2009). In some of the applications mentioned above, the input data is not immediately available and decisions need to be made as new data arrives; this is referred to as the online version of the problem. The focus of our work is however on offline algorithms, which can also help solving the corresponding online versions (Alon et al. 2006).

The problem we studied in this chapter is also known in the literature as the maximum weight degree-constrained subgraph problem. The special case with upper bounds only is known as maximum weight b-matching. In a centralized environment, both problems can be solved in polynomial time via linear programming solvers, maximum flow techniques as in Ahuja et al. (1993), or using the combinatorial algorithm developed in Gabow and Tarjan (1989). Unfortunately, these algorithms do not cope well with massive datasets.

In what follows, let n denote the number of vertices and m refer to the number of edges. For b-matching in general graphs, a simple greedy algorithm can achieve an approximation guarantee of $\frac{1}{2}$ running in $O(m \log n)$, and there is a randomized $(\frac{2}{3} - \epsilon)$-approximation algorithm with expected running time $O(bm \log \frac{1}{\epsilon})$ as shown

[4]We proceed similarly when more than one edge become integral.

by Mestre (2006), where $b = \max_{v \in V} b(v)$. To the best of our knowledge, the algorithm developed in Gabow and Tarjan (1989) with running time $\Omega(mn^{\frac{1}{2}})$ is the most efficient algorithm developed for exact b-matching to date.

Scalable algorithms for the weighted b-matching have been proposed in the literature. Using the message passing model of distributed computation, Panconesi and Sozio (2010) presented a distributed randomized algorithm for the bipartite case that yields a $\frac{1}{6+\epsilon}$-approximate solution requiring a poly-logarithmic number of rounds. Furthermore, Koufogiannakis and Young (2011) gave a distributed $\frac{1}{2}$-approximation algorithm for general graphs running in $O(\log m)$ rounds in expectation and with high probability. Later, Morales et al. (2011) adapted the algorithm of Panconesi and Sozio (2010) as well as a greedy algorithm to the MapReduce environment. More specifically, they presented an algorithm called GreedyMR, which implements the simple greedy algorithm in a MapReduce setting and achieves a $\frac{1}{2}$ approximation requiring a linear number of MapReduce steps, and an algorithm termed StackMR, which allows to violate upper-bound constraints by a factor of $1 + \epsilon$ and yields an approximation guarantee of $\frac{1}{6+\epsilon}$, but requires a poly-logarithmic number of MapReduce steps. Unfortunately, none of the above algorithms can handle lower- and upper-bound constraints simultaneously.

For ordinary b-matching as well as the more difficult case of perfect b-matching in bipartite graphs (a special case of generalized bipartite matching in which the degree of each node in the matching is constrained to be *equal to* some prespecified integral value) a belief propagation algorithm was first introduced by Bayati et al. (2011) and further improved by Huang and Jebara (2007; 2011). In particular, Bayati et al. (2011) showed that the belief propagation algorithm converges to the correct solution if and only if the LP relaxation of the integer program formulation of the b-matching has no fractional solution; in case that the LP relaxation has fractional solutions this algorithm can be used to solve the corresponding LP. The proposed algorithms also apply to general graphs and have worst-case complexity $O(mn)$. Parallel versions of the belief propagation algorithm seem be theoretically straightforward (Huang and Jebara 2011), however, the performance of parallel implementations on large-scale datasets has not been experimentally studied so far.

4.4 Experimental Results

In this section, we investigate the performance of MPCSolver (presented in Sections 3.2 and 3.3), DDRounding, and alternative algorithms for solving GBM in an extensive experimental study on both (semi-)synthetic and real-world datasets. We found that our algorithms are competitive with state-of-the-art LP solvers on small to moderately large problem instances, and multiple orders of magnitude faster than alternative methods on large instances.

4.4.1 Experimental Setup

A. Computational Environment

We implemented MPCSolver, Young's algorithm from Young (2001), and DDRounding in C++. Additionally, we made use of a CUDA implementation of MPCSolver for GPUs. To ensure a fair comparison, we used the data distribution techniques of Section 3.5 when implementing Young's algorithm, which greatly reduced its communication cost. All C++ implementations employed MPICH2 for communication.[5] We used three different setups to run our experiments: (1) a single high-memory server, (2) a compute cluster consisting of 16 nodes (with significantly less main memory), and (3) an NVIDIA GeForce GTX TITAN GPU with 6GB device memory. The high-memory server had 512GB of main memory and was equipped with 4 Intel Xeon 2.40GHz processors with 10 cores each (40 cores in total). Each node in the compute cluster had 48GB of main memory and an Intel Xeon 2.40GHz processor with 8 cores.

B. Real-World Datasets

Table 4.1 gives a brief overview of the datasets used in our experiments. We used the following real-world datasets: the Movielens10M dataset consisting of 10M movie ratings of 72k users over 10k items, the Netflix dataset (Bennett and Lanning 2007), which consists of roughly 99M ratings (1–5) of 456k Netflix users for 18k movies, the NetflixTop50 containing 24M movie ratings of 480k users over 18k items, and the KDD dataset of Track 1 of KDD-Cup 2011 (Dror et al. 2011), which consists of approximately 253M ratings of 1M Yahoo! Music users for 625k musical pieces. We excluded users with less than 10 ratings from the Netflix dataset (this ensures feasibility and is more realistic).

The real-world datasets above (Netflix, KDD, and Movielens10M) are somewhat unrealistic because we recommend items to users that the users have already rated. A more realistic setup is covered by NetflixTop50 (and our large-scale semi-synthetic dataset, denoted Semi-syn; see below). To create the NetflixTop50 dataset, we first applied the DSGD++ algorithm from Section 2.3.1D to predict unknown ratings in the Netflix matrix, and then selected the set of items with the top-50 largest predicted ratings for each user. In all datasets above, except for NetflixTop50, the number of ratings per user is unbalanced, i.e., there are users with very few ratings but also users with a large number of ratings. To construct a GBM instance, we converted each dataset to a bipartite graph (users and items form vertices; ratings correspond to weighted edges); our goal was to match (i.e., recommend) items to users. We used a lower bound of 3 and an upper bound of 5 on the number of

[5] http://www.mcs.anl.gov/mpi/mpich/

Table 4.1: *Summary of datasets*

Dataset	$\|U\|$	$\|V\|$	$\|E\|$
Movielens10M	72k	10k	10M
NetflixTop50	480k	18k	24M
Netflix	456K	18k	99M
KDD	1M	625k	253M
Syn	10M	1M	1B
Semi-Syn	480k	18k	3.2B

recommendations given to each user. We did not enforce a lower bound for items, but required each item to be recommended at most 200 times (Movielens10M, Netflix, and NetflixTop50) and 2000 times (KDD). These choices of bounds ensured that the resulting instances were feasible.

C. Synthetic and Semi-Synthetic Datasets

In order to investigate the scalability of our algorithms, we created two large-scale datasets: a semi-synthetic dataset, denoted Semi-Syn, and a synthetic dataset, denoted Syn. Semi-Syn is generated as follows: Similar to NetflixTop50, we first predicted the missing ratings of the Netflix matrix with DSGD++. Next, we sampled 3.2B entries uniformly from this matrix; each sample corresponds to an edge in the bipartite graph, weighted by the predicted rating. Note that this dataset is balanced, i.e., each user has the same number of ratings in expectation. The Semi-Syn dataset has a large number of edges but only a moderate number of vertices. In order to investigate the performance of our algorithms with a large number of vertices, we additionally generated Syn which consists of 10M users, 1M items, and 1B edges. The edges and their corresponding weights (between 1 and 5) are sampled uniformly at random. We generally use the same lower and upper bounds as for Netflix, except for Syn, where we modify the upper bound on the number of recommendations for each item to 50.

D. Optimal Solution

In order to compute the value of the optimum, we solved the LP relaxations using Gurobi Optimization (2013; 5.0), a state-of-the-art commercial solver for linear programs (and other problems); Gurobi takes advantage of multiple cores if available. Gurobi computed the optimal solution of small real-world datasets Movielens10M and NetflixTop50 on a cluster node. Gurobi was not able to solve problem instances of Netflix and KDD on one of our cluster nodes due to insufficient memory, even though these datasets were moderately large. On the high-memory server, however, Gurobi did produce an optimal solution for the Netflix and KDD datasets. For Syn and Semi-Syn, Gurobi ran out of memory even on the high-memory server; we thus were not able to compute the value of the optimum. Moreover, in each experiment with Gurobi, we tried both available Simplex and barrier LP solvers; the reported running time is the time required by the fastest LP solver of Gurobi.

E. Convergence Test

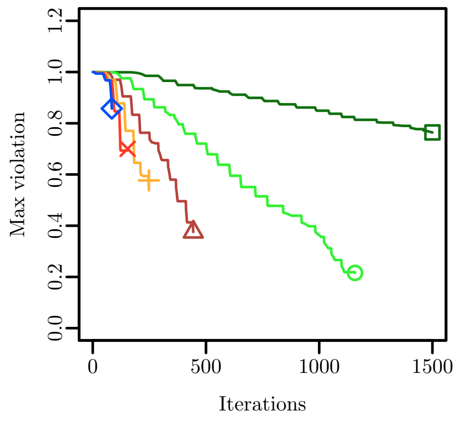
When running MPCSolver and Young's algorithm, we need to detect whether the algorithms have "practically" converged. Say that a solution has *maximum violation* λ if it is barely λ-feasible (i.e., not λ'-feasible for any $\lambda' < \lambda$). For MPCSolver we used the heuristic convergence test as described in Section 3.5.4. Young's algorithm was declared converged as soon as the maximum violation of only the packing constraints and only the covering constraints became equal (or the solution had maximum violation of ϵ). This modified convergence bound for Young's algorithm ensures a fair comparison: If we ran the algorithm any further, the maximum violation would increase (i.e., covering violation gets smaller, packing violation gets larger).

4.4.2 Results for GBM-LP (Feasibility)

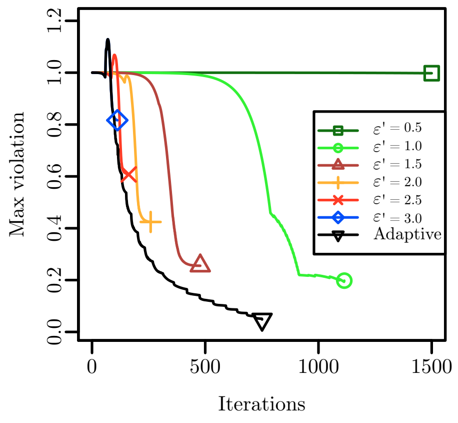
A. Experiments on Compute Cluster

In our first set of experiments, we compared MPCSolver and Young's algorithm for GBM-LP feasibility problems with respect to efficiency (in terms of both number of iterations and total time to convergence) and (strong) scalability. Our goal was to produce a 0.05-feasible solution, i.e., $\epsilon = 0.05$. As discussed below, running any of the two algorithms directly with such a small error bound leads to poor performance in practice. Instead, we ran the algorithms with some value $\epsilon' \geq \epsilon$ in the hope to still get an ϵ-feasible solution. For MPCSolver, we also considered the adaptive scheme of Section 3.5 for selecting the error bounds (with the parameters given in that section). The time for rescaling the input problem and computing the starting point was negligible (a few seconds) and is not included in our plots.

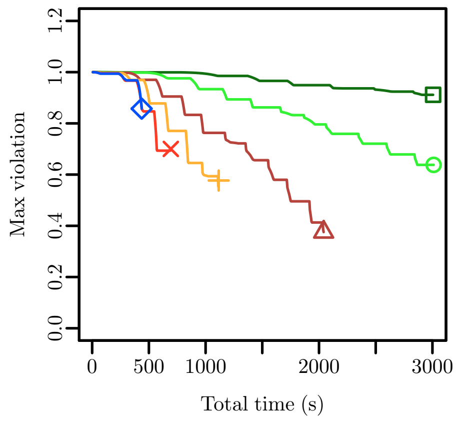
Efficiency (Figures 4.1 and 4.2). We studied the effect of error bound parameter ϵ' for both MPCSolver and Young's algorithm. For MPCSolver, ϵ' refers to the internal error bound, while for Young's algorithm ϵ' refers to the desired error bound given to the algorithm. Note that the choice of ϵ' affects both running time and the feasibility of the final solution. Figures 4.1 and 4.2 plot the maximum violation after every iteration until convergence for various choices of ϵ'. Here we only used a single cluster node with 8 parallel threads. All algorithms were run from the same initial point.



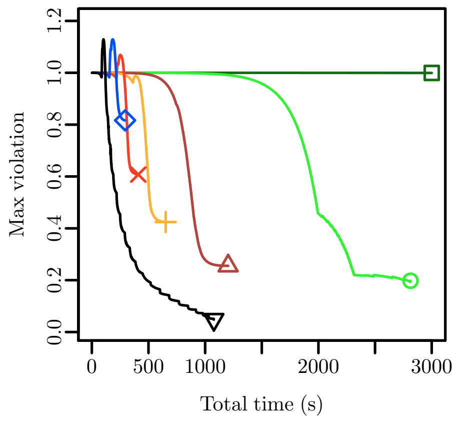
(a) *Young's algorithm on Netflix*



(b) *MPCSolver on Netflix*



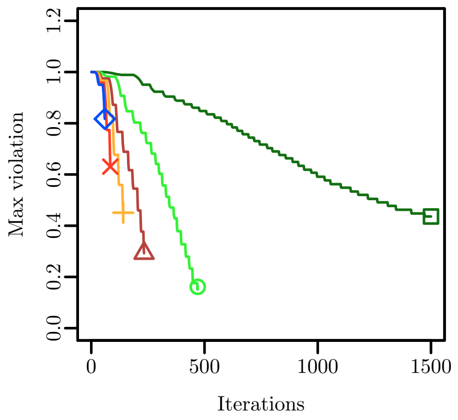
(c) *Young's algorithm on Netflix*



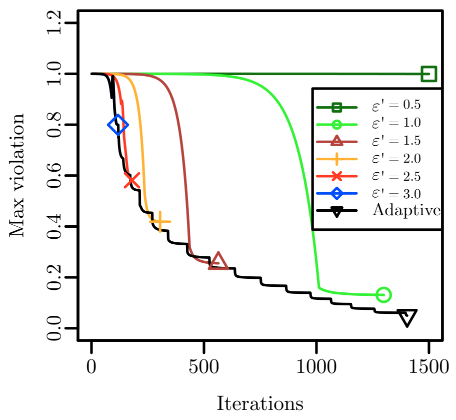
(d) *MPCSolver on Netflix*

Figure 4.1: *Efficiency of Young's algorithm and MPCSolver for* 0.05*-feasibility on Netflix (1x8)*

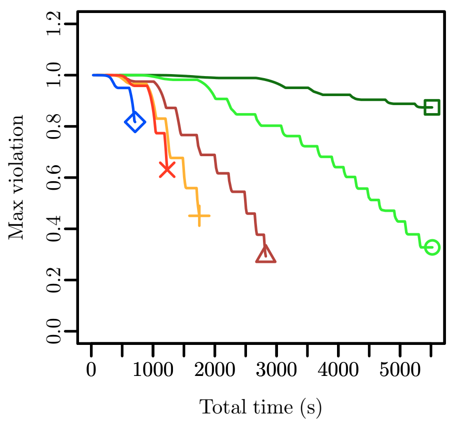
First note that ϵ' indeed affects time to convergence for both algorithms, which is in accordance with theory (MPCSolver is linear in $(\epsilon')^{-5}$, Young's algorithm linear in $(\epsilon')^{-4}$). Nevertheless, both algorithms produce solutions with maximum violation far less than ϵ' (but above $\epsilon = 0.05$); this effect was more pronounced for MPCSolver. For all choices of $\epsilon' \geq 1$, MPCSolver converged faster and achieved a higher precision (i.e., less maximum violation). Moreover, for both algorithms and on both datasets, a choice of $\epsilon' = 1$ worked best within 1500 iterations; the final maximum violation achieved by MPCSolver was slightly better than the one obtained by Young's algorithm (0.19 vs. 0.21 on Netflix, 0.13 vs. 0.16 on KDD). For $\epsilon' = 0.5$, neither of the algorithms converged after 1500 iterations: The iterate moved very slowly towards approximate feasibility. On Netflix (KDD)



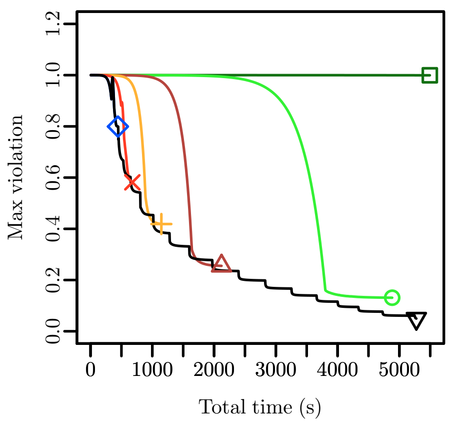
(a) *Young's algorithm on KDD*



(b) *MPCSolver on KDD*



(c) *Young's algorithm on KDD*



(d) *MPCSolver on KDD*

Figure 4.2: *Efficiency of Young's algorithm and MPCSolver for* 0.05*-feasibility on KDD (1x8)*

the achieved error was 0.98 (0.99) for MPCSolver and 0.76 (0.43) for Young's algorithm, respectively.

For high values of ϵ', the maximum violation dropped quickly in the beginning, but did not improve significantly in further iterations (in fact, both algorithms converged quickly for large ϵ'). For small values of ϵ', the maximum violation improved more slowly but eventually reached a lower value. This behavior motivated our adaptive method for selecting ϵ' (denoted "adaptive") discussed in Section 3.5. Recall that Young's algorithm cannot be run with adaptive error bound selection; see Section 3.7. In contrast, MPCSolver is well suited to adaptive error bound selection because it can start from an arbitrary starting point and is able to "undo" bad steps. As can be seen in Figures 4.1 and 4.2, MPCSolver with adaptive ϵ' outperforms all fixed choices of ϵ'. As a consequence, MPCSolver with adaptive error bound selection was the only method that achieved 0.05-feasibility in 1500 or less iterations. In particular, the maximum violation fell below 0.05 within 1077s (Netflix) and 5317s (KDD). In what follows, we focus on MPCSolver with adaptive ϵ'.

Regarding running time, we found that an iteration of MPCSolver was faster than an iteration of Young's algorithm (1.8x faster on average). Communication costs were similar, but Young's algorithm is more computationally intensive (since it needs to compute a global parameter from time to time). However, MPCSolver required significantly less iterations to converge, mainly due to its flexibility in terms of selecting ϵ'. Overall, MPCSolver was multiple orders of magnitude faster than Young's algorithm.

As a final remark, Gurobi's LP solver required about 2h (Netflix) and 3.5h (KDD) to find a feasible solution on our high-memory server. To have a fair comparison to MPCSolver, we also used Gurobi to find 0.05-feasible solution: the running times decreased to 1.9h (Netflix) and 3.3h (KDD). Thus, MPCSolver was competitive in terms of overall runtime to state-of-the-art parallel solvers for feasibility problems.

Strong scalability (Table 4.2). We investigated the runtime performance (measured as the average time per iteration and the total time until convergence) of MPCSolver and Young's algorithm as we increase the number of compute nodes from 1 to 16, where each node runs 8 threads. We refer to these setups using a "node x threads" abbreviation, i.e., 1x8, 2x8, . . ., 16x8. Our results are summarized in Table 4.2.

We first discuss the results on the moderately-sized real-world datasets Netflix and KDD (which do fit into the memory of a single cluster node). Compared to the per-iteration execution time of Netflix (KDD) on 1x8, MPCSolver provided 1.6x (1.7x) speedup on 2x8, and a 1.9x (2.4x) speedup on 4x8. Here communication overhead becomes significant so that speedup is sublinear. Young's algorithm has slightly higher speedup than MPCSolver, but starts at a higher cost initially. In fact, Young's algorithm on 8 nodes takes more time per iteration as MPCSolver on 2 nodes. Similar to MPCSolver, the benefit of moving from 4 to 8 or more nodes was marginal. Turning to overall time to convergence, we found that Young's algorithm did not converge after 24h. Note that we ran Young's algorithm with $\epsilon' = 0.05$ to ensure that we can actually find an ϵ-feasible solution, which caused it to move very slowly. In contrast, MPCSolver converged after 660 (1050) iterations on Netflix (KDD); the total running time was less than 1h on 4x8 and above.

We also investigated the performance of MPCSolver and Young's algorithm on the large Syn and Semi-Syn datasets. For Syn (Semi-Syn), we give results for 4x8, 8x8, and 16x8 (8x8 and 16x8) only; a smaller number of nodes had insufficient

Table 4.2: *Performance of Young's algorithm (Young) and MPCSolver (MPC) for feasibility problems ($\epsilon = 0.05$)*

	1x8		2x8		4x8		8x8		16x8	
	Young	MPC	Young	MPC	Young	MPC	Young	MPC	Young	MPC
Avg. time/iteration (s) on Netflix	5	**2.8**	3.5	**1.8**	2.7	**1.5**	2.3	**1.3**	2	**1.1**
Avg. time/iteration (s) on KDD	14	**7.8**	8.7	**4.6**	6.2	**3.3**	5.2	**3**	4.4	**2.7**
Avg. time/iteration (s) on Syn					29.2	**15.7**	19.7	**10.5**	14.6	**8**
Avg. time/iteration (s) on Semi-Syn							21	**11**	13.1	**6.9**
Time for feasibility (h) on Netflix	>24	**0.51**	>24	**0.33**	>24	**0.28**	>24	**0.24**	>24	**0.2**
Time for feasibility (h) on KDD	>24	**2.3**	>24	**1.3**	>24	**0.96**	>24	**0.87**	>24	**0.78**
Time for feasibility (h) on Syn					>24	**2.4**	>24	**1.6**	>24	**1.2**
Time for feasibility (h) on Semi-Syn							>24	**3.4**	>24	**2.1**

aggregate memory to store the data. On Syn, MPCSolver achieved 1.5x speedup when moving from 4x8 to 8x8. However, the algorithm ran solely 1.3x faster when using 16x8. This sublinear speedup is caused by increased communication cost (see below). Young's algorithm exhibited similar speedups. Each MPCSolver iteration took 8s on average using 16x8; MPCSolver converged to a 0.05-feasible solution after 551 iterations (1.2h). In contrast, iterations of Young's algorithm took longer (14.6s) and the algorithm did not converge within 24 hours. On Semi-Syn, though, we observed a better scalability for both MPCSolver and Young's algorithm; both algorithms provided 1.6x speedup on 16x8.

To understand this behavior, recall that the communication cost of each MPCSolver iteration is governed by the number of vertices, whereas the computation cost depends on the number of edges. Semi-Syn has less vertices than Syn (less communication) and many more edges (more computation) so that Semi-Syn is easier to parallelize. On Semi-Syn, each MPCSolver iteration required 6.9s on average using 16x8, and MPCSolver required 1120 iterations (2.1h) to compute a 0.05-feasible solution. Similar to Syn, iterations of Young's algorithm took longer (13.1s) and the algorithm did not converge within 24 hours.

We conclude that MPCSolver scales to very large datasets and is significantly faster than Young's algorithm. Moreover, MPCSolver is competitive to state-of-the-art LP solvers, but can handle much larger problem instances.

B. Experiments on GPU

An attractive feature of MPCSolver is its simplicity; it can be implemented in a few lines of code. Note that at each iteration, MPCSolver essentially computes a set of sparse matrix-vector products. Modern GPU architectures allow for massively parallel computation of such products, and GPU implementations can outperform CPU implementations by orders of magnitude. In our next experiment, we made use of a CUDA implementation for GPUs using CUSPARSE (Bell and Garland 2009) for matrix-vector products.

In Table 4.3, we report results of the performance of MPCSolver on the small-scale real datasets Movielens10M and NetflixTop50 (which do fit in the device memory of our GPU). Compared to a sequential CPU implementation, MPCSolver obtained a 38.6x and 26x speedup on NetflixTop50 and Movielens10M, respectively, when running on GPU. We also compared MPCSolver to Young's algorithm. On both datasets and when running with 8 threads, Young's algorithm required a few hours to obtain a 0.05-feasible solution. In the same setup, MPCSolver achieved the desired precision within 9.8 and 3.7 minutes, respectively.

Table 4.3: *Experimental results on Movielens10M and NetflixTop50*

Dataset	Average time/iteration (ms)			Time for 0.05-feasibility (s)		
	1x1	1x8	GPU	1x1	1x8	GPU
Movielens10M	727	250	**28**	652	223	**25**
NetflixTop50	2513	750	**65**	1971	588	**51**

C. Accelerating MPCSolver

In the next set of experiments, we explored the impact of the adaptive step size selection (Section 3.5.3) as well as the multiple updates heuristic (Section 3.5.5) on real-world datasets.To this end, we implemented the above techniques and measured the change in the number of rounds and the overall running time to convergence when compared to the plain MPCSolver. As before, our goal was to produce a 0.05-feasible solution. We report the results when using the adaptive step size selection (AS), using the multiple updates heuristic (MU), and using both the adaptive step size selection and the multiple updates heuristic combined (AS+MU), respectively, for various datasets on 1x8 in Table. 4.4. For AS, we examined two values of the step size: β and 10β; we selected the one which led to a lower potential value. First, observe that using all techniques substantially reduced the number of rounds to convergence on all datasets, however, each technique affected the overall running time of MPCSolver differently. Using AS only, increased the execution time; even though MPCSolver required less rounds until convergence, each round of the algorithm required twice more (since the cost of each round increases by factor 2 when trying two different values for the step size) compared to a round of the plain MPCSolver and the algorithm was overall slower (between 2% for NetflixTop50 and 60% for KDD). The MU heuristic, however, was very effective; it dramatically reduced the running time of MPCSolver on all datasets (between 30% for KDD and 70% for NetflixTop50). In fact, the most performance gain was obtained for all datasets when using MU, except for NetflixTop50 where using both AS and MU combined was most effective and led to 80% reduction in the overall running time.

Table 4.4: *Effect of adaptive step size selection (AS) and multiple updates heuristic (MU) for* 0.05*-feasibility on real datasets (1x8)*

Dataset	Increase (+) or decrease (-) in # rounds (left) and running time (right) to convergence (%)					
	AS		MU		AS+MU	
Movielens10M	-40	+20	-45	**-45**	-62	-24
NetflixTop50	-49	+2	-70	-70	-90	**-80**
Netflix	-26	+48	-32	**-32**	-49	+2
KDD	-20	+60	-30	**-30**	-45	+10

4.4.3 Results for GBM-LP (Optimality)

Recall that we need to run a sequence of feasibility instances to approximately solve the optimization version of the GBM problem. We first present results for Netflix and KDD with $\epsilon = 0.05$. We compared the solution of MPCSolver with the optimal solution computed by Gurobi and found that the desired approximation ratio was obtained (i.e., the value of the objective was at least 95% of the optimum). On 16x8, MPCSolver required 2.9h (Netflix) and 7.4h (KDD) in total. On the high-memory server, Gurobi required 2.2h (Netflix) and 3.8h (KDD).

For both Syn and Semi-Syn, Gurobi ran out of memory. In contrast, MPCSolver required 9.5h (13.6h) for Syn and 14.7h (24h) for Semi-Syn on 16x8 (8x8), respectively. Note that the high-memory server had sufficient memory for MPCSolver to process both Syn and Semi-Syn; our algorithm required 9.7h (Syn) and 18h (Semi-Syn) using all 40 cores. Thus, MPCSolver is faster on the high-memory server (with

40 cores) than on the 8x8 cluster setup (with 64 cores). Although the hardware in both setups is not identical, a key reason for the performance difference is that communication between workers is fast in shared-memory systems (high-memory server) but significantly slower in shared-nothing systems (compute cluster).

Overall, we found that MPCSolver is competitive in terms of overall runtime. It's key advantage is that it can scale over a compute cluster and thus to much larger problem instances; it's key disadvantage is that results are approximate up to ϵ.

4.4.4 Results for Distributed Rounding

Recall that the approximate solution of GBM-LP is generally not integral. In our next set of experiments, we evaluated the performance of DDRounding to produce an integral solution. As before, we measured performance with respect to quality (i.e., value of objective function after rounding), efficiency, and strong scalability. We used the 1x8 to 16x8 setups described previously.

Quality. For all datasets, we took the fractional solution obtained by MPCSolver with $\epsilon = 0.05$ as input to DDRounding. For each real-world dataset, we executed 10 independent runs of DDRounding, thereby obtaining 10 integral solutions. We found that the value of the objective differed only marginally (within 0.5%) from the value of the fractional solution across all 10 runs. Moreover, for Netflix (KDD) the best run produced solutions that were slightly better than the fractional solution (increase of 0.5% for Netflix, 0.1% for KDD); this increase is possible because the fractional solution is ϵ-feasible but not feasible. For Syn and Semi-Syn, we only executed a single run. As before, the objective was close to the fractional solution ($\approx 0.01\%$ off) for both datasets. Thus, the integral solutions obtained by DDRounding rounding were of essentially the same quality as the fractional solutions.

Efficiency and strong scalability (Figure 4.3). We studied the performance of DDRounding with varying number of cluster nodes. As can be seen in Figure 4.3, DDRounding clearly benefits from distributed processing, i.e., performance improved significantly when adding more nodes: The local subgraph processed in the DRounding step on each node becomes smaller so that cycles can be removed more efficiently. We expect, however, that using too many nodes is not beneficial because then the number of cycles per subgraph may become too small (so that there is little work to do). Nevertheless, we found that the performance of DDRounding improves even if we go beyond 8x8, even on our moderately-sized real-world datasets. On 16x8, we achieved an integral solution for all the real-world datasets within 1.6h. For Syn, we obtained an integral solution after 9.4h, 5.8h, and 3.6h using 4x8, 8x8, and 16x8, respectively; on smaller setups, the data did not fit into the main memory. We observed similar speedups for Semi-Syn; the time to obtain an integral solution

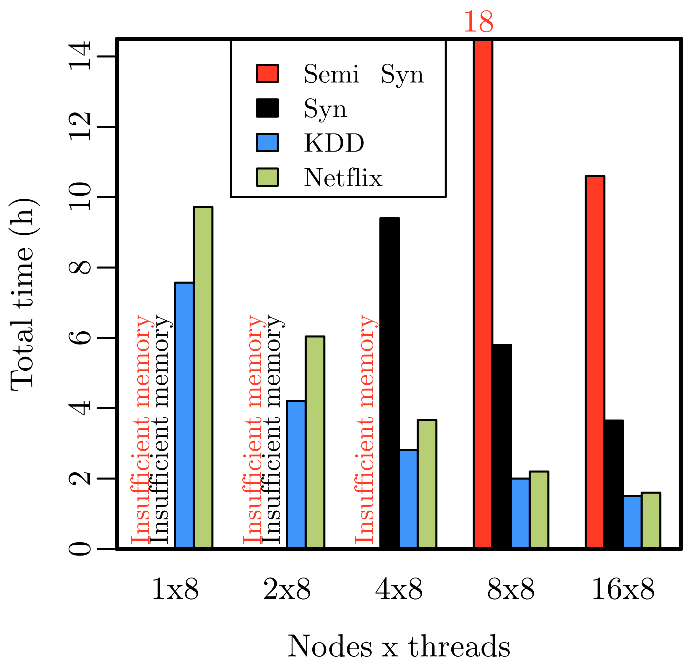


Figure 4.3: *Scalability of DDRounding on different datasets*

was reduced from 18h using 8x8 to 10.5h on 16x8; as before, the data exceeded the main memory available when using less than 8 compute nodes.

Constraint violations (Table 4.5). Recall that DDRounding preserves the lower- and upper-bound constraints (up to rounding). In our next experiment, we investigated the actual constraint violations obtained by MPCSolver, both before and after applying DDRounding. Table 4.5 summarizes our results. First note that on all datasets, both the fractional and the integral solution violated only very few constraints (i.e., is almost feasible). As mentioned previously, our error analysis is somewhat loose so that MPCSolver performs better than guaranteed by theory. For Netflix and Semi-Syn, DDRounding further decreased the violations in the constraints (slightly).

4.4.5 Results for GBM

In our final experiments, we put everything together and investigated how well we can solve the GBM problem. Recall that computation of an optimal integral solution with Gurobi on the high-memory server took 2.2h (3.8h) for Netflix (KDD). With MPCSolver and DDRounding on 16x8 and for our choice of $\epsilon = 0.05$, we obtained a 0.05-feasible solution in 4.5h (8.9h) that was 95.5% (95.1%) of the optimal solution. Thus, the desired approximation ratio was indeed realized. Approximately solving GBM-LP took 2.9h (7.4h), rounding to an integral solution took 1.6h (1.5h).

Table 4.5: *Constraint violations before (fractional solution) and after (integral solution) rounding*

Dataset	# satisfied constraints		# ϵ-feasible (but not feasible)	
	Fractional	Integral	Fractional	Integral
Netflix	930 085	930 086	45	44
KDD	2 626 509	2 626 509	432	432
Syn	20 998 614	20 998 623	1386	1377
Semi-Syn	978 090	978 105	58	43

Dataset	Avg. rel. violation (infeasible)		Max rel. violation (infeasible)	
	Fractional	Integral	Fractional	Integral
Netflix	0.0473	0.0145	0.0496	0.015
KDD	0.0407	0.0409	0.0499	0.05
Syn	0.0485	0.0491	0.0499	0.05
Semi-Syn	0.0492	0.0163	0.0498	0.02

Observe that MPCSolver with rounding was slower than Gurobi. However, the individual cluster nodes used by our algorithms were less powerful, both in terms of memory and in terms of number of cores. Moreover, our approach can handle much larger problem instances. For Syn and Semi-Syn, which cannot be handled by Gurobi on our high-memory server, a 0.05-feasible solution to GBM was obtained after 13.1h (9.5h for GBM-LP, 3.6h for rounding) and 25.2h (14.7h for GBM-LP, 10.5h for rounding), respectively, on 16x8. We expect that running time can be further reduced by adding more compute nodes.

4.5 Summary

Graph matching problems continuously arise in a wide variety of applications, e.g., in computational advertising, recommender systems, and trade markets. In this chapter, we have studied the generalized bipartite matching problem and its application in recommending multimedia items (e.g., DVDs) under constraints. Our goal is to provide high quality recommendations to customers such that some desired properties are satisfied.

We have proposed the first distributed approximation algorithm for computing near-optimal solutions to large-scale generalized bipartite matching problems. Unlike traditional methods that suffer from scalability issues, our algorithm is designed to run on a small cluster of commodity nodes and scales to realistic-sized problems which may involve millions of users and millions of items. Our method is based on linear programming and randomized rounding. In particular, we have first applied MPCSolver presented in Chapter 3 to compute an approximate fractional solution to the LP relaxation of the integer linear program corresponding to the matching problem. Next, we have developed DDRounding, a distributed randomized rounding algorithm to transform the fractional solution to an integral one of high quality. Our experiments on both real-world and synthetic datasets suggest that our algorithms scale to very large problem sizes and can be orders of magnitude faster than alternative approaches.

5 Conclusion and Outlook

The overall goal of this thesis is to obtain scalable solutions for a number of optimization problems that can be applied in (but not restricted to) recommender systems to generate personalized recommendations satisfying certain objectives and constraints.

Matrix Completion

We addressed large-scale matrix completion problems, i.e., recovering a large partially observed matrix from a small subset of its entries, with millions of rows, millions of columns, and billions of revealed entries. We considered an application in the context of collaborative filtering in recommender systems; in this setting matrix completion techniques have proven very effective. The goal is to predict missing entries of a large user-movie-rating matrix efficiently. We proposed novel shared-nothing ASGD, and DSGD++ algorithms, which are designed to run on a cluster of commodity nodes, have less memory consumption, and offer better scalability than previous MapReduce approaches. Our algorithms are cache-friendly and utilize thread-level parallelism, in-memory processing, and asynchronous communication. We highlighted the key metrics that affect the performance of different algorithms in practice and analyzed the performance of our novel algorithms with existing approaches through a theoretical complexity analysis as well as an extensive empirical study on real and synthetic datasets. On large datasets, DSGD++ consistently outperformed alternative algorithms in terms of efficiency, scalability, and memory consumption.

Future work. The SGD-based algorithms presented in this thesis are effective for matrix completion tasks. One further step would be to examine and extend these distributed SGD-based techniques to more general matrix factorization tasks such as matrix reconstruction or non-negative matrix factorization. Note that matrix reconstruction differs from matrix completion in that, unlike matrix completion, zero entries in the input matrix encode actual information and need to be taken into consideration while learning the factors. Therefore, loss functions used for matrix reconstruction measure the error between *all* entries of the original matrix and the reconstructed matrix. Consequently, the algorithm needs to iterate over all matrix entries making the factorization more challenging. Moreover, our experimental evaluation can be extended to include more algorithms, different loss functions, and regularization terms.

Mixed Packing-Covering Linear Programming

We investigated distributed solutions for MPC-LPs, a simple yet expressive subclass of LPs with an abundance of applications in combinatorial optimization. We developed MPCSolver, a novel approximation algorithm for solving such LPs in a shared-nothing setting and proved its convergence via a full theoretical analysis. The key properties of MPCSolver are its simplicity and efficiency. It is well-suited for parallel processing on GPUs, in shared-memory architectures, and on a small cluster of commodity nodes. Moreover, we discussed how to distribute data across nodes to minimize the communication costs and provided implementation details to improve its performance in practice. A case study with instances of large-scale GBM problems indicate that MPCSolver offers better scalability and efficiency than alternative approaches.

Future work. We list some interesting questions related to MPCSolver worth investigating.

- Our current implementation of MPCSolver deploys a heuristic to determine whether the algorithm has converged, namely, if the potential function remains almost unchanged in two consecutive rounds. Is it possible to perform a feasibility check (after every round or from time to time) and detect the infeasibility earlier without running the algorithm for several rounds?

- Recall that for the optimization version of MPC-LP, MPCSolver requires solving a sequence of feasibility problems. To improve efficiency, it is worth investigating how to optimize the objective function directly and avoid the binary search procedure completely.

- In our experiments, we observed that the multiple updates heuristic is very effective. It would be worth analyzing how this technique affects the convergence of MPCSolver.

- A comparison between MPCSolver, Thetis (Sridhar et al. 2013), and ADMM in an experimental evaluation in terms of efficiency, accuracy (i.e., violation in the constraints), and quality (i.e., the actual approximation guarantee) would be insightful.

- It would be also interesting to investigate whether similar techniques used in MPCSolver would generalize to broader classes of mathematical programs, e.g., more general LPs and convex problems including semidefinite programs.

Generalized Bipartite Matching

We studied generalized bipartite matching problems, which continuously appear in a wide range of applications in computational advertising, recommender systems, trade markets, etc. We considered an application in recommending multimedia items (e.g., DVDs) to customers under a set of constraints. We presented the first shared-nothing approximation algorithm for computing near-optimal solutions for large-scale generalized bipartite matching problems involving millions or users and millions of items. Our approach is based on LP relaxation and randomized rounding: We first obtained an approximate fractional solution to the LP relaxation of the integer linear programming formulation of the matching problem and then rounded the fractional solution to an integral one. For the first step, we applied MPCSolver and for the second step, we developed DDRounding, a randomized rounding algorithm for obtaining high quality solutions in a shared-nothing environment. Experiments on real and synthetic datasets on varying sizes indicate better scalability and efficiency compared to alternative approaches.

Future work. The algorithms presented in Chapter 4 can be generalized to contexts different from providing constrained recommendations. Exploring new applications and finding the limitations of the algorithms in practice is an interesting line of research. Another line of research worth notice is to solve generalized matching problems on other models of computations on massive datasets like streaming models (Alon et al. 1999; Henzinger et al. 1999) or semi-streaming models (Feigenbaum et al. 2005), and finally to investigate solving such matching problems on more general graphs of large-scale.

The contributions of this thesis can be seen as a step towards efficient large-scale optimization algorithms for matrix completion, mixed packing-covering linear programs, and generalized bipartite matching problems. We believe that the techniques presented in this thesis have the potential of great practical applicability. We hope that the contributions of this thesis will be valuable for future research.

Bibliography

Ageev, A. A. and Sviridenko, M. (2004). Pipage rounding: A new method of constructing algorithms with proven performance guarantee. *Journal of Combinatorial Optimization*, 8(3):307–328.

Ahuja, R. K., Magnanti, T. L., and Orlin, J. B. (1993). *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall, Englewood Cliffs, NJ.

Alon, N., Awerbuch, B., Azar, Y., Buchbinder, N., and Naor, J. (2006). A general approach to online network optimization problems. *ACM Transactions on Algorithms*, 2(4):640–660.

Alon, N., Matias, Y., and Szegedy, M. (1999). The space complexity of approximating the frequency moments. *Journal of Computer and System Sciences*, 58(1):137–147.

Amatriain, X. and Basilico, J. (2012). Netflix recommendations: Beyond the 5 stars (part 1). http://techblog.netflix.com/2012/04/netflix-recommendations-beyond-5-stars.html.

Arora, S., Hazan, E., and Kale, S. (2012). The multiplicative weights update method: a meta-algorithm and applications. *Theory of Computing*, 8(1):121–164.

Awerbuch, B., Fu, Z., and Khandekar, R. (2009). Brief announcement: Stateless distributed algorithms for generalized packing linear programs. In *Proceedings of the ACM annual Symposium on Principles of Distributed Computing (PODC)*, pages 270–271.

Awerbuch, B. and Khandekar, R. (2009). Stateless distributed gradient descent for positive linear programs. *SIAM Journal on Computing*, 38(6):2468–2486.

Bartal, Y., Byers, J. W., and Raz, D. (2004). Fast, distributed approximation algorithms for positive linear programming with applications to flow control. *SIAM Journal on Computing*, 33(6):1261–1279.

Battiti, R. (1989). Accelerated backpropagation learning: Two optimization methods. *Complex Systems*, 3:331–342.

Bayati, M., Borgs, C., Chayes, J. T., and Zecchina, R. (2011). Belief propagation for weighted b-matchings on arbitrary graphs and its relation to linear programs with integer solutions. *SIAM Journal on Discrete Mathematics*, 25(2):989–1011.

Bell, N. and Garland, M. (2009). Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–11.

Bennett, J. and Lanning, S. (2007). The Netflix prize. In *Proceedings of the KDD Cup and Workshop*, pages 3–6.

Bertsekas, D. (1999). *Nonlinear Programming*. Athena Scientific.

Bhalgat, A., Feldman, J., and Mirrokni, V. S. (2012). Online allocation of display ads with smooth delivery. In *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1213–1221.

Bienstock, D. (2002). *Potential function methods for approximately solving linear programming problems : Theory and practice*. International Series in Operations Research and Management Science. Kluwer academic publishers.

Bienstock, D. and Iyengar, G. (2004). Solving fractional packing problems in $O^*(1/\epsilon)$ iterations. In *Proceedings of the ACM Annual Symposium on Theory of Computing (STOC)*, pages 146–155.

Boley, D. (2013). Local linear convergence of the alternating direction method of multipliers on quadratic or linear programs. *SIAM Journal on Optimization*, 23(4):2183–2207.

Bottou, L. and Bousquet, O. (2007). The tradeoffs of large scale learning. In *Proceedings of the Conference on Advances in Neural Information Processing Systems (NIPS)*, pages 161–168.

Boyd, S. P., Parikh, N., Chu, E., Peleato, B., and Eckstein, J. (2011). Distributed optimization and statistical learning via the alternating direction method of multipliers. *Foundations and Trends in Machine Learning*, 3(1):1–122.

Byrd, R. H., Lu, P., Nocedal, J., and Zhu, C. (1995). A limited memory algorithm for bound constrained optimization. *SIAM Journal on Scientific Computing*, 16(5):1190–1208.

Candès, E. J. and Plan, Y. (2010). Matrix completion with noise. *Proceedings of the IEEE*, 98(6):925–936.

Candes, E. J. and Recht, B. (2009). Exact matrix completion via convex optimization. *Foundations of Computational Mathematics*, 9(6):717–772.

Charles, D. X., Chickering, M., Devanur, N. R., Jain, K., and Sanghi, M. (2010). Fast algorithms for finding matchings in lopsided bipartite graphs with applications to display ads. In *Proceedings of the ACM Conference on Electronic Commerce (EC)*, pages 121–128.

Charlin, L., Zemel, R., and Boutilier, C. (2012). Active learning for matching problems. In *Proceedings of the International Conference on Machine Learning (ICML)*, pages 337–344.

Chen, P.-L., Tsai, C.-T., Chen, Y.-N., Chou, K.-C., Li, C.-L., Tsai, C.-H., Wu, K.-W., Chou, Y.-C., Li, C.-Y., Lin, W.-S., Yu, S.-H., Chiu, R.-B., Lin, C.-Y., Wang, C.-C., Wang, P.-W., Su, W.-L., Wu, C.-H., Kuo, T.-T., McKenzie, T., Chang, Y.-H., Ferng, C.-S., Ni, C.-M., Lin, H.-T., Lin, C.-J., and Lin, S.-D. (2012). A linear ensemble of individual and blended models for music rating prediction. *Journal of Machine Learning Research: Proceedings Track*, 18:21–60.

Chu, C.-T., Kim, S. K., Lin, Y.-A., Yu, Y., Bradski, G. R., Ng, A. Y., and Olukotun, K. (2006). Map-reduce for machine learning on multicore. In *Proceedings of the Conference on Advances in Neural Information Processing Systems (NIPS)*, pages 281–288.

Cichocki, A. and Phan, A. H. (2009). Fast local algorithms for large scale nonnegative matrix and tensor factorizations. *IEICE Transactions*, 92-A(3):708–721.

Danzig, G. B. (1963). *Linear Programming and Extensions*. Princeton University Press, Princeton, N.J.

Das, S., Sismanis, Y., Beyer, K. S., Gemulla, R., Haas, P. J., and McPherson, J. (2010). Ricardo: integrating r and hadoop. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 987–998.

Diedrich, F. and Jansen, K. (2007a). An approximation algorithm for the general mixed packing and covering problem. In *Proceedings of the International Symposium on Combinatorics, Algorithms, Probabilistic and Experimental Methodologies (ESCAPE)*, pages 128–139.

Diedrich, F. and Jansen, K. (2007b). Faster and simpler approximation algorithms for mixed packing and covering problems. *Theoretical Computer Science*, 377(1-3):181–204.

Dobkin, D. P., Lipton, R. J., and Reiss, S. P. (1979). Linear programming is log-space hard for p. *Information Processing Letters*, 8(2):96–97.

Drineas, P., Magdon-Ismail, M., Pandurangant, G., Virrankoski, R., and Savvides, A. (2006). Distance matrix reconstruction from incomplete distance information for sensor network localization. In *Proceedings of the Annual IEEE Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks (SECON)*, pages 536–544.

Dror, G., Koenigstein, N., Koren, Y., and Weimer, M. (2011). The Yahoo! Music Dataset and KDD-Cup'11. In *KDDCup 2011 Workshop*.

Eckstein, J. and Bertsekas, D. P. (1990). An alternating direction method for linear programming. In *LIDS Technical Reports*. Laboratory for Information and Decision Systems, Massachusetts Institute of Technology.

Feigenbaum, J., Kannan, S., McGregor, A., Suri, S., and Zhang, J. (2005). On graph problems in a semi-streaming model. *Theoretical Computer Science*, 348(2-3):207–216.

Gabay, D. and Mercier, B. (1976). A dual algorithm for the solution of nonlinear variational problems via finite element approximation. *Computers and Mathematics with Applications*, 2(1):17–40.

Gabow, H. N. and Tarjan, R. E. (1989). Faster scaling algorithms for network problems. *SIAM Journal on Computing*, 18(5):1013–1036.

Gandhi, R., Khuller, S., Parthasarathy, S., and Srinivasan, A. (2006). Dependent rounding and its applications to approximation algorithms. *Journal of the ACM*, 53(3):324–360.

Garg, N., Kavitha, T., Kumar, A., Mehlhorn, K., and Mestre, J. (2010). Assigning papers to referees. *Algorithmica*, 58(1):119–136.

Garg, N. and Könemann, J. (2007). Faster and simpler algorithms for multicommodity flow and other fractional packing problems. *SIAM Journal on Computing*, 37(2):630–652.

Gemulla, R., Haas, P. J., Nijkamp, E., and Sismanis, Y. (2011a). Large-scale matrix factorization with distributed stochastic gradient descent. Technical Report RJ10481, IBM Almaden Research Center, San Jose, CA. http://researcher.watson.ibm.com/researcher/files/us-phaas/rj10482Updated.pdf.

Gemulla, R., Haas, P. J., Sismanis, Y., Teflioudi, C., and Makari, F. (2011b). Large-scale matrix factorization with distributed stochastic gradient descent. In *Proceedings of the NIPS Workshop on Big Learning: Algorithms, Systems, and Tools for Learning at Scale*. http://biglearn.org/2011/files/papers/biglearn2011_submission_15.pdf.

Gemulla, R., Nijkamp, E., Haas, P. J., and Sismanis, Y. (2011c). Large-scale matrix factorization with distributed stochastic gradient descent. In *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 69–77.

Glowinski, R., M. A. (1975). Sur l'approximation, par éléments finis d'ordre un, et la résolution, par pénalisation-dualité d'une classe de problémes de dirichlet non linéaires. *ESAIM: Mathematical Modelling and Numerical Analysis - Modélisation Mathématique et Analyse Numérique*, 9(R2):41–76.

GNU Scientific Library (2013). GNU Scientific Library. http://www.gnu.org/software/gsl/.

Grigoriadis, M. D. and Khachiyan, L. G. (1994). Fast approximation schemes for convex programs with many blocks and coupling constraints. *SIAM Journal on Computing*, 4(1):86–107.

Grigoriadis, M. D. and Khachiyan, L. G. (1996). Coordination complexity of parallel price-directive decomposition. *Mathematics of Operations Research*, 21(2):321–340.

Grigoriadis, M. D., Khachiyan, L. G., Porkolab, L., and Villavicencio, J. (2001). Approximate max-min resource sharing for structured concave optimization. *SIAM Journal on Optimization*, 11(4):1081–1091.

Gurobi Optimization, I. (2013). Gurobi optimizer reference manual.

Henzinger, M. R., Raghavan, P., and Rajagopalan, S. (1999). Computing on data streams. *Dimacs Series In Discrete Mathematics And Theoretical Computer Science*, pages 107–118.

Hsieh, C.-J., Chang, K.-W., Lin, C.-J., Keerthi, S. S., and Sundararajan, S. (2008). A dual coordinate descent method for large-scale linear svm. In *Proceedings of the International Conferenceon Machine Learning (ICML)*, pages 408–415.

Hsieh, C.-J. and Dhillon, I. S. (2011). Fast coordinate descent methods with variable selection for non-negative matrix factorization. In *Proceedings of the ACM*

SIGKDD International Conference on Knowledge Discovery and Data Mining, pages 1064–1072.

Hsieh, C.-J., Sustik, M. A., Dhillon, I. S., and Ravikumar, P. D. (2011). Sparse inverse covariance matrix estimation using quadratic approximation. In *Proceedings of the Conference on Advances in Neural Information Processing Systems (NIPS)*, pages 2330–2338.

Hu, Y., Koren, Y., and Volinsky, C. (2008). Collaborative filtering for implicit feedback datasets. In *Proceedings of the IEEE International Conference on Data Mining (ICDM)*, pages 263–272.

Huang, B. C. and Jebara, T. (2007). Loopy belief propagation for bipartite maximum weight b-matching. In *Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS)*, pages 195–202.

Huang, B. C. and Jebara, T. (2011). Fast b-matching via sufficient selection belief propagation. *Journal of Machine Learning Research - Proceedings Track*, 15:361–369.

Jain, P., Netrapalli, P., and Sanghavi, S. (2013). Low-rank matrix completion using alternating minimization. In *Proceedings of the ACM Annual Symposium on Theory of Computing (STOC)*, pages 665–674.

Jansen, K. and Zhang, H. (2002). Approximation algorithms for general packing problems with modified logarithmic potential function. In *Proceedings of the Foundations of Information Technology in the Era of Networking and Mobile Computing (IFIP) International Conference on Theoretical Computer Science (TCS)*, pages 255–266.

Jebara, T. and Shchogolev, V. (2006). B-matching for spectral clustering. In *Proceeding of the European Conference on Machine Learning (ECML)*, pages 679–686.

Jebara, T., Wang, J., and Chang, S.-F. (2009). Graph construction and *b*-matching for semi-supervised learning. In *Proceedings of the International Conferenceon Machine Learning (ICML)*, page 56.

Karakostas, G. (2002). Faster approximation schemes for fractional multicommodity flow problems. In *Proceedings of the ACM-SIAM Annual Symposium on Discrete Algorithms (SODA)*, pages 166–173.

Karmarkar, N. (1984). A new polynomial-time algorithm for linear programming. In *Proceedings of the ACM Annual Symposium on Theory of Computing (STOC)*, pages 302–311.

Keshavan, R. H. (2012). *Efficient Algorithms for Collaborative Filtering*. PhD thesis, Stanford University.

Khachiyan, L. G. (1979). A polynomial time algorithm in linear programming. *soviet mathematics doklady*, 20:191–194.

Khandekar, R. (2004). Lagrangian relaxation based algorithms for convex programming problems.

Klein, P. N., Plotkin, S. A., Stein, C., and Tardos, É. (1994). Faster approximation algorithms for the unit capacity concurrent flow problem with applications to routing and finding sparse cuts. *SIAM Journal on Computing*, 23(3):466–487.

Koren, Y. (2008). Factorization meets the neighborhood: a multifaceted collaborative filtering model. In *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 426–434.

Koren, Y., Bell, R. M., and Volinsky, C. (2009). Matrix factorization techniques for recommender systems. *IEEE Computer*, 42(8):30–37.

Koufogiannakis, C. and Young, N. E. (2009). Distributed and parallel algorithms for weighted vertex cover and other covering problems. In *Proceedings of the ACM annual Symposium on Principles of Distributed Computing (PODC)*, pages 171–179.

Koufogiannakis, C. and Young, N. E. (2011). Distributed algorithms for covering, packing and maximum weighted matching. *Distributed Computing*, 24(1):45–63.

Koufogiannakis, C. and Young, N. E. (2013). Greedy δ-approximation algorithm for covering with arbitrary constraints and submodular cost. *Algorithmica*, 66(1):113–152.

Krishnamurthy, A. and Singh, A. (2013). Low-rank matrix and tensor completion via adaptive sampling. In *Proceedings of the Conference on Advances in Neural Information Processing Systems (NIPS)*, pages 836–844.

Kuhn, F., Moscibroda, T., and Wattenhofer, R. (2006). The price of being near-sighted. In *Proceedings of the ACM-SIAM Annual Symposium on Discrete Algorithms (SODA)*, pages 980–989.

Kushner, H. and Yin, G. (2003). *Stochastic Approximation and Recursive Algorithms and Applications*. Applications of mathematics. Springer.

LAPACK (2012). Linear Algebra PACKage. http://www.netlib.org/lapack.

Bibliography

Lattanzi, S., Moseley, B., Suri, S., and Vassilvitskii, S. (2011). Filtering: a method for solving graph problems in mapreduce. In *Proceedings of the ACM Annual Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 85–94.

Leighton, F. T., Makedon, F., Plotkin, S. A., Stein, C., Stein, É., and Tragoudas, S. (1995). Fast approximation algorithms for multicommodity flow problems. *Journal of Computer and System Sciences*, 50(2):228–243.

Li, B., Tata, S., and Sismanis, Y. (2013). Sparkler: Supporting large-scale matrix factorization. In *Proceedings of the ACM International Conference on Extending Database Technology (EDBT)*, pages 625–636.

Liben-Nowell, D. and Kleinberg, J. M. (2007). The link-prediction problem for social networks. *Journal of the Association for Information Science and Technology (JASIST)*, 58(7):1019–1031.

Linial, N. (1992). Locality in distributed graph algorithms. *SIAM Journal on Computing*, 21(1):193–201.

Liu, C., chih Yang, H., Fan, J., He, L.-W., and Wang, Y.-M. (2010). Distributed nonnegative matrix factorization for web-scale dyadic data analysis on mapreduce. In *Proceedings of the International World Wide Web Conference (WWW)*, pages 681–690.

Liu, J., Wright, S. J., Ré, C., and Bittorf, V. (2013). An asynchronous parallel stochastic coordinate descent algorithm.

Luby, M. and Nisan, N. (1993). A parallel approximation algorithm for positive linear programming. In *Proceedings of the ACM Annual Symposium on Theory of Computing (STOC)*, pages 448–457.

Mackey, L. W., Talwalkar, A., and Jordan, M. I. (2011). Divide-and-conquer matrix factorization. In *Proceedings of the Conference on Advances in Neural Information Processing Systems (NIPS)*, pages 1134–1142.

Makari, F., Awerbuch, B., Gemula, R., Khandekar, R., Mestre, J., and Sozio, M. (2013). A distributed algorithm for large-scale generalized matching. *Proceedings of the VLDB Endowment*, 6(9):613–624.

Makari, F. and Gemulla, R. (2013). A distributed approximation algorithm for mixed packing-covering linear programs. In *Proceedings of the NIPS Workshop on Big Learning: Advances in Algorithms and Data Management*. http://biglearn.org/2013/files/papers/biglearning2013_submission_14.pdf.

Makari, F., Teflioudi, C., Gemulla, R., Haas, P. J., and Sismanis, Y. (2014). Shared-memory and shared-nothing stochastic gradient descent algorithms for matrix completion. *Knowledge and Information Systems*.

McDonald, R., Hall, K., and Mann, G. (2010). Distributed training strategies for the structured perceptron. In *Proceedings of the Annual Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (HLT-NAACL)*, pages 456–464.

Megiddo, N. (1992). A note on approximate linear programming. *Information Processing Letters*, 42(1):53.

Mestre, J. (2006). Greedy in approximation algorithms. In *Proceedings of the Annual European Symposium on Algorithms (ESA)*, pages 528–539.

Morales, G. D. F., Gionis, A., and Sozio, M. (2011). Social content matching in mapreduce. *Proceedings of the VLDB Endowment*, 4(7):460–469.

MPI (2013). Message Passing Interface Forum. http://www.mpi-forum.org.

Nedic, A. and Bertsekas, D. (2000). *Convergence Rate of Incremental Subgradient Algorithms*. Kluwer.

Negahban, S. and Wainwright, M. J. (2012). Restricted strong convexity and weighted matrix completion: Optimal bounds with noise. *Journal of Machine Learning Research*, 13:1665–1697.

Panconesi, A. and Sozio, M. (2010). Fast primal-dual distributed algorithms for scheduling and matching problems. *Distributed Computing*, 22(4):269–283.

Papadimitriou, C. H. and Yannakakis, M. (1993). Linear programming without the matrix. In *Proceedings of the ACM Annual Symposium on Theory of Computing (STOC)*, pages 121–129.

Penn, M. and Tennenholtz, M. (2000). Constrained multi-object auctions and b-matching. *Information Processing Letters*, 75(1-2):29–34.

Plotkin, S. A., Shmoys, D. B., and Tardos, É. (1995). Fast approximation algorithms for fractional packing and covering problems. *Mathematics of Operations Research*, 20:257–301.

Recht, B. (2011). A simpler approach to matrix completion. *Journal of Machine Learning Research*, 12:3413–3430.

Recht, B. and Ré, C. (2013). Parallel stochastic gradient algorithms for large-scale matrix completion. *Mathematical Programming Computation*, 5:201–226.

Recht, B., Re, C., Wright, S. J., and Niu, F. (2011). Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Proceedings of the Conference on Advances in Neural Information Processing Systems (NIPS)*, pages 693–701.

Riedel, S., Yao, L., McCallum, A., and Marlin, B. M. (2013). Relation extraction with matrix factorization and universal schemas. In *Proceedings of the Annual Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (HLT-NAACL)*, pages 74–84.

Serna, M. J. (1991). Approximating linear programming is log-space complete for p. *Information Processing Letters*, 37(4):233–236.

Shahrokhi, F. and Matula, D. W. (1990). The maximum concurrent flow problem. *Journal of the ACM*, 37(2):318–334.

Shaw, B. and Jebara, T. (2007). Minimum volume embedding. *Journal of Machine Learning Research - Proceedings Track*, 2:460–467.

Shaw, B. and Jebara, T. (2009). Structure preserving embedding. In *Proceedings of the International Conferenceon Machine Learning (ICML)*, pages 937–944.

Singer, A. (2008). A remark on global positioning from local distances. *Proceedings of the National Academy of Sciences*, 105(28):9507–9511.

Smola, A. and Narayanamurthy, S. (2010). An architecture for parallel topic models. *Proceedings of the VLDB Endowment*, 3(1–2):703–710.

Sridhar, S., Wright, S. J., Re, C., Liu, J., Bittorf, V., and Zhang, C. (2013). An approximate, efficient lp solver for lp rounding. In *Proceedings of the Conference on Advances in Neural Information Processing Systems (NIPS)*, pages 2895–2903.

Teflioudi, C., Makari, F., and Gemulla, R. (2012). Distributed matrix completion. In *Proceedings of the IEEE International Conference on Data Mining (ICDM)*, pages 655–664.

Todd, M. J. (2002). The many facets of linear programming. *Mathematical Programming*, 91(3):417–436.

Trevisan, L. and Xhafa, F. (1998). The parallel complexity of positive linear programming. *Parallel Processing Letters*, 8(4):527–533.

Tsitsiklis, J., Bertsekas, D., and Athans, M. (1986). Distributed asynchronous deterministic and stochastic gradient optimization algorithms. *IEEE Transactions on Automatic Control*, 31(9):803–812.

Young, N. E. (2001). Sequential and parallel algorithms for mixed packing and covering. In *Proceedings of the Annual Symposium on Foundations of Computer Science (FOCS)*, pages 538–546.

Yu, H.-F., Hsieh, C.-J., Si, S., and Dhillon, I. S. (2012). Scalable coordinate descent approaches to parallel matrix factorization for recommender systems. In *Proceedings of the IEEE International Conference on Data Mining (ICDM)*, pages 765–774.

Yu, H.-F., Huang, F.-L., and Lin, C.-J. (2011). Dual coordinate descent methods for logistic regression and maximum entropy models. *Machine Learning*, 85(1-2):41–75.

Zhou, Y., Wilkinson, D. M., Schreiber, R., and Pan, R. (2008). Large-scale parallel collaborative filtering for the netflix prize. In *Proceedings of the International Confernece on Algorithmic Aspects in Information and Management (AAIM)*, pages 337–348.

Zhuang, Y., Chin, W.-S., Juan, Y.-C., and Lin, C.-J. (2013). A fast parallel sgd for matrix factorization in shared memory systems. In *Proceedings of the ACM Conference on Recommender Systems (RecSys)*, pages 249–256.

$\mathcal{A}$ Basic Notations

We use the following notation throughout this manuscript. For a positive number n, let $[1, n] = \{1, \ldots, n\}$. Every vector $\boldsymbol{v}$ will be understood as a column-vector and its transpose (a row-vector) will be $\boldsymbol{v}^\top$. We write $\boldsymbol{x}_i$ to refer to the i-th entry of any vector $\boldsymbol{x}$. For any $m \times n$ matrix $\boldsymbol{A}$, denote by $\boldsymbol{A}_{i*}$ the i-th row of $\boldsymbol{A}$ (i.e., a $1 \times n$ matrix), by $\boldsymbol{A}_{*j}$ the j-th column of $\boldsymbol{A}$ (i.e., an $m \times 1$ matrix), and by $\boldsymbol{A}_{ij}$ the (i, j)-entry of $\boldsymbol{A}$. Denote by $\mathbb{1}$ a vector of all ones. The dimension of $\mathbb{1}$ is left unspecified and will be clear from the context. Denote by $\Re$ the set of real numbers and by $\Re_+$ the set of non-negative real numbers. Thus, $\Re_+^n$ refers to the set of all n-dimensional vectors of non-negative real numbers. We use $\exp(x) = e^x$ to denote the exponential function. The expression $\ln x$ and $\log x$ refer to the logarithm of $x > 0$ to the base $e = 2.71828...$ and 2, respectively. Along with the standard O-notation, we also use $\tilde{O}$-notation to hide poly-logarithmic factors, i.e., $\tilde{O}(f)$ refers to a function in $O(f \log^c f)$ for a constant $c \geq 0$. For a minimization (resp. maximization) problem $\mathcal{P}$, an algorithm $\mathcal{A}$ is an α-factor approximation for $\mathcal{P}$, for some $\alpha > 1$ (resp. $\alpha < 1$) if the objective value of any solution produced by $\mathcal{A}$ is at most (resp. at least) α times the value of the optimal solution to $\mathcal{P}$.

List of Figures

1.1 A basic framework to generate recommendations under constraints. Observed ratings are shown in black, predicted in red, and selected for recommendation circled. Adapted from Charlin et al. (2012). . 3

2.1 Memory layout used on node k by the shared-nothing algorithms ($t = 1$). Node-local data is shown in white, master copies in light gray, and temporary data in dark gray. 18

2.2 Strata used by SSGD for a 3×3 blocking of $\boldsymbol{V}$ 20

2.3 Performance of shared-nothing algorithms on real-world datasets, $r = 100$ 38

2.4 Performance of shared-nothing algorithms on synthetic datasets . 40

3.1 Memory layout used on node b. Node-local data is shown in white and temporary data in gray. 61

3.2 Reduction trees and data flow for partitions A–D, $\boldsymbol{M} = \{\boldsymbol{P}, \boldsymbol{C}\}$, $s = 4$. In reduction trees, leaf nodes are colored white, internal nodes light gray, and root nodes dark gray. Final results are aggregated at root nodes and then distributed to all the compute nodes (not shown in the figure). 63

3.3 Various types of LPs considered in Chapter 3 68

4.1 Efficiency of Young's algorithm and MPCSolver for 0.05-feasibility on Netflix (1x8) 88

4.2 Efficiency of Young's algorithm and MPCSolver for 0.05-feasibility on KDD (1x8) 89

4.3 Scalability of DDRounding on different datasets 95

List of Tables

2.1 Notation for matrix completion algorithms 11

2.2 Popular loss functions for matrix completion 12

2.3 Overview of shared-nothing methods (see Table 2.1 for notation) 31

2.4 Summary of datasets 35

2.5 SGD step size sequence (Netflix, $r = 50$) 37

2.6 Impact of stratum schedules on DSGD++ (2x8) 37

4.1 Summary of datasets 86

4.2 Performance of Young's algorithm (Young) and MPCSolver (MPC) for feasibility problems ($\epsilon = 0.05$) 91

4.3 Experimental results on Movielens10M and NetflixTop50 92

4.4 Effect of adaptive step size selection (AS) and multiple updates heuristic (MU) for 0.05-feasibility on real datasets (1x8) 93

4.5 Constraint violations before (fractional solution) and after (integral solution) rounding 96

List of Tables

List of Algorithms

1 The SGD++ algorithm for matrix completion 16
2 The SSGD algorithm for matrix completion (Gemulla et al. 2011c) 20
3 MPCSolver for mixed packing-covering LPs 47
4 DRounding (Gandhi et al. 2006) 80
5 DDRounding 82