Parallel Implementation of Interior-Point Methods for Semidefinite Optimization

Ivan D. Ivanov

Stellingen

behorende bij het proefschrift

Parallelle Implementatie of Inwendige Punt Methoden voor Semidefiniete Optimalisering

Ivan D. Ivanov

1. Inwendige punt methoden zijn niet in het bijzonder geschikt voor parallel rekenen.

   [*Dit proefschrift, Hoofdstuk 2, 4*]

2. Bij parallel rekenen hoeft een lokaal voordelige actie niet noodzakelijk te leiden tot een verbetering in globaal opzicht.

   [*Dit proefschrift, Hoofdstuk 5*]

3. De ironie bij het gebruik van clusters is dat een significante reductie van de rekentijd kan worden verkregen, ondanks dat er N maal zoveel sequentiele berekeningen nodig zijn (waarbij N het aantal processoren is).

   [*Dit proefschrift*]

4. Geparalleliseerde inwendige punt methoden voor semidefiniete optimalisering zullen in de toekomst meer revolutie dan evolutie nodig hebben.

   [*B. Borchers and J. G. Young. "Implementation of a primal-dual method for SDP on a shared memory parallel architecture." Computational Optimization and Applications, 37(3):355-369, 2007*]

5. Als de grens van een bepaalde processor technologie is bereikt dan kunnen computers met meerdere processoren uitkomst bieden.

6. 'High performance computing' is meer een kunst dan een wetenschap.

7. Balanceren tussen praktische toepassingen en wetenschappelijk belang is een van de meest onbevredigende opgaven, maar geeft de meest bruikbare resultaten.

8. Het ontbreken van de juiste middelen is geen excuus voor het niet doen van de juiste dingen.

9. In de eenentwintigste eeuw lijkt het een slechte zaak om privacy in te ruilen voor publieke veiligheid.

10. De Engelse taal mag dan wel helpen om te overleven in Nederland, maar het opent alleen de deur naar de 'logeerkamer'.

Deze stellingen worden opponeerbaar en verdedigbaar geacht en zijn als zodanig goedgekeurd door de promotor, Prof. dr. ir. C. Roos.

Propositions

accompanying the thesis

Parallel Implementation of Interior-Point Methods for Semidefinite Optimization

Ivan D. Ivanov

1. Interior-point algorithms are not particularly suitable for parallel computations.

   [*This thesis, Chapter 2, 4*]

2. In parallel computing, doing what is best on local level does not always result in overall improvement.

   [*This thesis, Chapter 5*]

3. The irony when using cluster computing is that a significant time reduction can be achieved, despite the N times more sequential computations (where N is the number of processors).

   [*This thesis*]

4. Parallel IPM's for large-scale SDO will in future require revolution rather than evolution.

5. When the limits of certain processor technology is reached, multiprocessor computers save the day.

6. High performance computing is more of an art than a science.

7. Balancing between the practical applications and the scientific value of your work is among the most unsatisfying jobs, but gives the most useful results.

8. Missing the right tools should not be an excuse for not doing the right things.

9. In 21st century, it seems like a bad deal to trade privacy for public security.

10. The English language may help you to survive in the Netherlands, but it only opens the door to the 'guest room'.

These propositions are considered opposable and defendable and as such have been approved by the supervisor, Prof. dr. ir. C. Roos.

Parallel Implementation of Interior-Point Methods for Semidefinite Optimization

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus Prof. dr. ir. J.T. Fokkema,
voorzitter van het College voor Promoties,
in het openbaar te verdedigen
op donderdag 5 juni 2008 om 10.00 uur

door

Ivan Dimitrov IVANOV

Automatics and Control Engineer
Technical University of Sofia
geboren te Silistra, Bulgarije.

Dit proefschrift is goedgekeurd door de promotor:

Prof. dr. C. Roos

Toegevoegd promotor: Dr. E. de Klerk

Samenstelling promotiecommissie:

Rector Magnificus	voorzitter
Prof. dr. C. Roos	Technische Universiteit Delft, promotor
Dr. E. de Klerk	Universiteit van Tilburg, toegevoegd promotor
Prof. dr. H.J. Sips	Technische Universiteit Delft
Prof. dr. C.W. Oosterlee	Technische Universiteit Delft
Prof. dr. B. Borchers	New Mexico Tech, USA
Prof. dr. F. Jarre	Heinrich Heine Universität, Germany
Prof. dr. T. Terlaky	McMaster University, Canada
Prof. dr. G.J. Olsder	Technische Universiteit Delft, reservelid

Dit proefschrift kwam tot stand onder auspicien van:

Thomas Stieltjes Institute
for Mathematics

ISBN: 978-90-9023109-9

Typeset by the author with the LaTeX Documentation System.

Author email: id.ivanov@dir.bg

To Desie

Acknowledgements

Having reached the level called a PhD degree, I realize that many people contributed to my success story and I would like to acknowledge them here.

First and foremost, I would like to thank my scientific advisor and promotor Prof. Cornelis Roos. He accepted me in 2003 as a member of the Optimization group in TU Delft and provided me with the necessary guidance and support during all these years. His advice, drive for perfection, and attention to detail brought the quality of my work to a higher level.

My deep appreciations and thanks to my co-promotor and advisor Dr. Etienne de Klerk. Despite the fact that he only joined me in the last two years of my work, he contributed a lot with his ideas and comments for the final results, presented in this thesis. Etienne, I enjoyed very much working with you. Thank you for everything you did for me.

I would also like to thank to all members of the Optimization group in Delft, especially to my office mates Manuel, Mohamed and Hossein. Guys, I enjoyed all discussions we had and our work together.

Furthermore, I wish to thank to the department of Software Technology and Prof. Henk Sips for proving me with the necessary financial grant to finish this manuscript. Special thanks to Dr. Dick Epema for proofreading Chapter 3 of this thesis. I also appreciate the help provided by the technical stuff of our department, especially Paulo Anita.

Moving to more personal acknowledgements, I would like to thank all my Bulgarian friends in Delft: Blagoy and Nadia, Ivo, Alex, Kuzi, Petar, Ventzi, Plamen, Yana and all others who shared my lunchtimes, travels and parties. All those years, you made my life easier in the Netherlands.

I greatly appreciate the true friendship of the Hoogstraat 33 (Veldhoven) people: David, Paulo, Antanas, Martin, Cristiano, Fernando, PoYang, Madeline, and Cecile. They were my first house mates when I arrived to the Netherlands. The international atmosphere surrounding them was the experience of a lifetime for me.

My deepest gratitude to my parents for their care, trust, advices, moral and financial support. I'm thankful to my brother for the joyful moments and helpful discussions. But most of all, I would like to apologize to my family and friends in Bulgaria for being far away from them the last 5 years.

Last, but certainly not least, I want to thank my wife Desie, to whom I dedicate this thesis, for her love and great support, sharing good and worse with me.

Delft, June 2008
Ivan Ivanov

Contents

List of Figures

List of Tables

List of Notation

$\mathbb{R}^n$	:	the n-dimensional Euclidean vector space;
$\mathbb{R}^{n\times n}$	:	the space of square n-by-n matrices;
(P)	:	primal problem in standard form;
(D)	:	Lagrangian dual problem of (P);
$\mathcal{P}$	:	feasible set of problem (P);
$\mathcal{D}$	:	feasible set of problem (D);
$\mathbb{S}^n$	:	the space of symmetric n-by-n matrices;
$\mathbb{S}^n_+$	:	the space of symmetric positive semidefinite matrices;
$\mathbb{S}^n_{++}$	:	the space of symmetric positive definite matrices;
A^T	:	transpose of $A \in \mathbb{R}^{m\times n}$;
A_{ij}	:	ijth entry of $A \in \mathbb{R}^{m\times n}$;
$\mathbf{Tr}(A)$	=	$\sum_i A_{ii}$ (trace of $A \in \mathbb{R}^{n\times n}$);
$\|A\|_F$	=	$\sqrt{\sum_i \sum_j A_{ij}^2}$ (Frobenius norm);
$A \succeq 0$	:	A is symmetric positive semidefinite;

$A \succ 0$: A is symmetric positive definite;

$A^{\frac{1}{2}}$ = unique symmetric square root factor of $A \succeq 0$;

$\text{vec}(A)$ = $[A_{11}, A_{21}, ..., A_{n1}, A_{22}, ..., A_{nn}]^T$ for $A \in \mathbb{R}^{n \times n}$;

$\text{diag}(x)$ = diagonal matrix with components of $x \in \mathbb{R}^n$ on diagonal;

$\text{diag}(X)$ = vector obtained by extracting diagonal of $X \in \mathbb{R}^{n \times n}$;

$A \otimes B$: Kronecker product on two matrices A and B of arbitrary size;

I : identity matrix of size depending on the context;

e : vector of all-ones of size depending on the context;

0 : zero vector/matrix of size depending on the context;

Chapter 1

Introduction

Optimization is often used to make decisions about the best way to use the available resources. The resources may be raw materials, machine time or man hours, money, or anything else in limited supply. The best or optimal solution may mean minimizing costs, maximizing profits, or achieving the best quality.

The best known and the most widely used form of optimization is linear optimization, where the objective and all of the constraints are linear functions of the variables. Many practical problems can be formulated with linear objectives (say for profits) and linear constraints (for budgets, capacities, etc.). These problems involve thousands to hundreds of thousands of variables and constraints, and linear optimization problems of this size can be solved nowadays quickly and reliably with modern optimization software. What stands behind such software are the efficient algorithms for linear optimization (LO).

Interior-point methods (IPM) are among the most efficient algorithms for solving linear optimization problems. They were pioneered by Karmakar [66] in 1984. Karmakar's algorithm was the first polynomial-time algorithm for LO performing well in practice. His paper attracted a lot of attention in the optimization community and many researchers developed algorithms inspired by that algorithm and its analysis. As a result interior-point methods became a very active area of research. Some variants of Karmakar's *polynomial-time projective* algorithm are related to classical methods like the logarithmic barrier method of Frisch [44, 45] and Fiacco and McCormick [39], see Gill et al. [51]. The primal-dual framework for following the central path was introduced by Meggido [77]. Path-following primal-dual IPM's for linear optimization were proposed by Kojima et al. [70] and by Monteiro and Adler [82].

In contrast, nonlinear optimization problems – where the objective and the constraints may not be linear functions of the variables – are in principal more difficult to solve than the linear case.

> "In fact, the great watershed in optimization isn't between linearity and nonlinearity, but convexity and nonconvexity."
>
> — R. Tyrrell Rockafellar, in [93], 1993

This statement should be qualified a bit, because not every *convex* nonlinear optimization problem can be solved in polynomial time. Many NP-hard problems have a natural convex reformulation, for example computing the stability number of a graph, see e.g. DeKlerk and Pasechnik [31]. Therefore, convexity done is not enough to solve an optimization problem efficiently. From a complexity point of view, we also require a polynomial time computable *self-concordant* barrier function for the convex set of the problem, see Nesterov and Nemirovsky [87]. Hence, when we refer to convex optimization problems further on, we mean problems for which there exist such barrier functions.

Primal-dual interior-point methods for LO, were extended to the field of convex nonlinear optimization, particulary to semidefinite optimization. This was done by Nesterov and Nemirovsky [87] and by Alizadeh [2], independently. In *semidefinite optimization* (SDO) one minimizes a linear objective function subject to the constraint that an affine combination of symmetric matrices is positive semidefinite. This type of constraint is convex despite the fact that it is nonlinear and nonsmooth, see e.g. Vandenberghe and Boyd [110]. Hence, semidefinite optimization problems are indeed a subclass of convex optimization problems.

The interest in SDO during the last two decades was due to the ever growing list of applications in engineering, combinatorial optimization, and control theory. For a survey of engineering applications of SDO including control theory, structural and combinatorial optimization problems we refer to the article of Vandenberghe and Boyd [110]. Recently, SDO was also used for electronic structure calculations in quantum chemistry [119]. A number of hard combinatorial optimization problems such as the max-cut [52], the traveling salesman problem [26], and the quadratic assignment problem [118] have SDO relaxations that provide an approximation of the original problem. In some problems, like max-cut, the approximation is provably good. Thus, semidefinite optimization proved itself as a powerful modeling technique of practical importance.

1.1 Interior-point methods for Semidefinite optimization

Here we introduce in more details the notion of a semidefinite optimization problem. The goal of SDO is to minimize the linear objective

$$p := \mathbf{Tr}(CX),$$

which consists of trace (denoted by '$\mathbf{Tr}$') of the product of two symmetric $n \times n$ matrices, a given data matrix C and a variable matrix X, subject to a set of linear and nonlinear constraints. The linear constraints are:

$$\mathbf{Tr}(A_i X) = b_i, \quad i = 1, ..., m,$$

where the A_i's are given $n \times n$ symmetric matrices, and the b_i's are known scalars. The nonlinear constraint is that the matrix variable X must be symmetric positive semidefinite[1], denoted by $X \succeq 0$.

Hence, the semidefinite optimization problem has the following form, also called *primal*:

$$\begin{aligned} &\inf_X \ \mathbf{Tr}(CX) \\ &\text{s.t. } \mathbf{Tr}(A_i X) = b_i, \quad i = 1, ..., m, \\ &\quad X \succeq 0. \end{aligned}$$

Here the infimum is used because the minimum may not be attained. Without loss of generality we can assume that all constraint matrices A_i are linearly independent. It is well known that the Lagrangian *dual* of this problem is given by

$$\begin{aligned} &\sup_{y,Z} \ b^T y \\ &\text{s.t. } \textstyle\sum_{i=1}^m y_i A_i + Z = C, \\ &\quad Z \succeq 0, \end{aligned}$$

where $y_i \in \mathbb{R}^m$ and $Z \in \mathbb{S}_+^n$. With $\mathbb{S}_+^n$ we denote the space of $n \times n$ symmetric positive semidefinite matrices.

The primal and dual feasible sets, denoted by $\mathcal{P}$ and $\mathcal{D}$ respectively are

$$\mathcal{P} := \{X : \mathbf{Tr}(A_i X) = b_i, i = 1, ..., m, X \succeq 0\},$$

and

$$\mathcal{D} := \{(y, Z) : \textstyle\sum_{i=1}^m y_i A_i + Z = C, y \in \mathbb{R}^m, Z \succeq 0\}.$$

We will refer to (X, y, Z) as a feasible solution when $X \in \mathcal{P}$ and $(y, Z) \in \mathcal{D}$, i.e., they satisfy the primal and the dual constraints, respectively.

The duality gap for semidefinite optimization is defined as

$$\mathbf{Tr}(CX) - b^T y = \mathbf{Tr}((Z - \textstyle\sum_{i=1}^m y_i A_i)X) - \sum_{i=1}^m y_i \mathbf{Tr}(A_i X) = \mathbf{Tr}(ZX).$$

For a feasible solution (X, y, Z) the *weak duality* property holds in SDO, namely

$$\mathbf{Tr}(ZX) \geq 0.$$

This follows from the conditions $X \succeq 0$ and $Z \succeq 0$ for the primal and the dual problems, respectively. In other words the duality gap is nonnegative for feasible solutions.

A standard assumption on the primal-dual problems is the so-called "strict feasibility" assumption.

[1] A symmetric matrix X is positive semidefinite if all of its eigenvalues are nonnegative, or equivalently, if $s^T X s \geq 0$ for all $s \in \mathbb{R}^n$.

Definition 1.1.1. (Strict feasibility) *A primal problem (resp. dual) is called strictly feasible if there exists $X \in \mathcal{P}$ with $X \succ 0$ (resp. $(y, Z) \in \mathcal{D}$ with $Z \succ 0$).*

Let $(X \succ 0, Z \succ 0)$ (i.e. positive definite) satisfies both the linear constraints for the primal and dual SDO problems. This ensures the existence of an optimal primal-dual pair (X^*, Z^*) with zero duality gap (the *strong duality* property), see Vandenberghe and Boyd [110, Theorem 3.1], and optimal solutions of SDO are characterized by

$$\begin{aligned} b_i - \mathbf{Tr}(A_i X) = 0, \quad i = 1, ..., m, \quad X \succeq 0, \quad & \text{(Primal feasibility)} \\ C - Z - \textstyle\sum_{i=1}^m y_i A_i = 0, \quad Z \succeq 0, \quad & \text{(Dual feasibility)} \\ XZ = 0. \quad & \text{(Complementary slackness)} \end{aligned}$$

The idea of the interior-point methods is to replace the complementary slackness condition $XZ = 0$ by the perturbed condition $XZ = \mu I$, where I denotes the identity matrix and $\mu > 0$ is given. As a result, we get the system

$$\begin{aligned} b_i - \mathbf{Tr}(A_i X) = 0, \quad & i = 1, ..., m, \\ C - Z - \textstyle\sum_{i=1}^m y_i A_i = 0, \quad & \\ XZ = \mu I, \quad & \\ X, Z \succ 0. \quad & \end{aligned}$$

It is well known this system has an unique solution $(X(\mu), y(\mu), Z(\mu))$ for each $\mu > 0$, see e.g. [27, chap. 3]. The set of solutions, denoted by

$$\mathcal{C} = \left\{ (X(\mu), y(\mu), Z(\mu)) \in \mathbb{S}_+^n \times R^m \times \mathbb{S}_+^n : \mu > 0 \right\},$$

defines an analytic curve inside the feasible region called the primal-dual central path. Note that the strict feasibility is necessary in order to exist $\mathcal{C}$ and thus to solve the primal and the dual problem. The idea of the interior-point methods is to follow the central path to optimality for a decreasing sequence of values of the parameter μ. These are the so-called path-following IPM.

There are different types of interior-point methods for SDO, for example primal-dual path-following methods, primal-dual potential reduction methods, dual scaling methods, etc. This thesis deals with primal-dual path-following IPM's which we discuss in detail in Chapter 2. These methods are well known for their efficiency, accuracy and robustness on many problem classes. There exist several software packages for SDO such as CSDP5.0 [18], SeDuMi [99, 100], and SDPA [113] based on primal-dual IPM methods. For a review of the other interior-point methods for semidefinite optimization we refer to the book of De Klerk [27].

1.2 State-of-the-art in problem sizes

A natural question to ask is: What is the computational effort required to solve a semidefinite optimization problem using the interior-point methods described above? More specifically, how does the effort grow with problem size? In terms of iterations required, all IPM's have similar polynomial worst-case complexity. The worst-case bound on the number of iterations necessary to solve a semidefinite optimization problem to a given accuracy grows with problem size as $\mathcal{O}(n^{1/2})$, see e.g. [110].

In practice the IPM algorithms behave much better than predicted by the worst-case complexity analysis. It has been observed by many researchers that the number of IPM iterations is essentially uncorrelated with traditional measures of problem size such as the number of equality constraints m or the dimension n of the variables in the primal SDO in standard form. For a wide variety of SDO problems and a large range of problem sizes, IPM's typically require between 5 and 50 iterations, see Nesterov and Nemirovsky [87, §6.4.4] for comments on the average behavior.

Space complexity is another important aspect when one has to solve ever larger SDO problems. By this term we mean the amount memory, or space required by the IPM algorithms. Storage requirements are as important as the computational complexity. Often in practice, the size of the largest SDO problems that can be solved depends more on available memory than on available CPU time, see e.g. [20].

Primal-dual interior-point algorithms typically involve the so-called Schur complement equation (SCE) with a symmetric and positive definite Schur complement matrix, say M, of size $m \times m$ where m is the number of linear constraints of (P), see e.g. Helmberg et al. [60]. In most practical SDO problems, this matrix is dense, even if the data matrices are sparse. In addition both primal and dual variables X and Z, respectively, need n^2 storage.

Until recently, the desktop PC's have been using 32bit addressing space only which limits the physical memory to 2^{32} unique cells, i.e., only 4 gigabytes (GB) of RAM. On the other hand 4Gb of memory would be enough to accommodate a fully dense square matrix of dimension 23,170 that contains double-precision[2] floating-point numbers. Such storage is already inadequate for SDO problems arising from practice nowadays. For example, the hamming_11_2 problem that we solve later in this thesis has number of constraints $m = 56,321$. The resulting Schur complement matrix in this case will have $56,321$ rows and columns and it would require 25GB of RAM alone if it is fully dense.

There are two general approaches in case one deals with such large-scale SDO problems. The first one is to use IPM's that avoid explicit calculation of M, see e.g. Kočvara and Stingl [72]. Such interior-point methods make use of an

[2]The IEEE Standard for Binary Floating-Point Arithmetic (IEEE 754) is the most widely-used standard for floating-point computation. It specifies that one double-precision floating-point number is represented by 64-bits, i.e., 8-bytes.

iterative solver such as the conjugate gradient (CG) method instead of direct factorization. The main difficulties that arise with these approaches is that the Schur complement equation becomes ill-conditioned when approaching the optimal value. In order for an iterative method to be efficient, a good preconditioner is required. However, solving a general SDO problem would require a general case preconditioner. It is well-known that there is no preconditioner that is both general and efficient. Consequently, the use of iterative methods in IPM's for solving large-scale SDO problems is limited to cases where a (very) low accuracy is necessary.

The second and widely used in practice approach is to use the latest generation computers capable of 64bit addressing, and larger than 4GB available memory. The theoretical memory limit there is 10^{10}GB, while most of the servers in practice have 128GB or more RAM nowadays. Such computers allow solution of semidefinite optimization problems with up to hundred thousand constraints and dimension of the primal and the dual variables in the range of tens of thousands. An example of an implementation of interior-point algorithm for SDO developed for such computers is the CSDP solver [20]. Unfortunately, such supercomputers are purpose-built and their cost puts them beyond the reach of many users.

An alternative solution to overcome the memory limitations is to distribute the Schur complement matrix over computers organized as a cluster. This is much more economical solution since it employs PCs and ethernet network, and a free open source software. As a result, clusters are cheaper, often by orders of magnitude, than single supercomputer with 128GB of memory. Implementations of interior-point methods for SDO that use a distributed computation of the Schur complement matrix are already available: SDPARA [114], SDPARA-C [85] and PDSDP[10]. Unfortunately, they are designed for clusters with high speed network interconnects (much more expensive than ethernet). In this thesis we focus on developing a distributed IPM solver for large-scale SDO problems better suited for an ethernet network.

1.3 Large-scale SDO problems

During the last 15 years the major driving force behind the development of IPM software for semidefinite optimization was the size of the problems arising in practice. Over the years researchers collected a number of instances of semidefinite problems that were challenging for the solvers and organized them in the late 1990s as test sets such as SDPLIB [19]. Since then, these problems became a benchmark for interior-point codes. Initially, most of the problems in SDPLIB were considered truly large scale and appeared challenging in terms of memory use and processor time for a single CPU computer. The largest instances from this set are maxG60 and theta51 both with around 7000 constraints, and size of matrix variables 7000 and 1001, respectively. These instances gave a boost to the area of parallel and high-performance computations for interior-point methods.

The Seventh DIMACS Implementation Challenge[3] in 2002 lifted the bar even higher for the semidefinite optimization software. The DIMACS library[80] that consists of mixed semi-definite-quadratic-linear programs was established to provide up-to-date benchmarks. It contains SDO problems from different applications that represent all levels of difficulty. For example instances fap09 and hamming_11_2 have 15,255 and 56,321 constraints, respectively. The data matrices and the Schur complement matrix of the hamming_11_2 instance require more than 20GB of memory and PC's rarely have this amount of RAM.

Recently, a number of new applications of SDO resulted also in larger problem instances. One of these applications is calculating electronic structures in quantum chemistry [86]. One problem that we solve later from this application is CH4.1A1.STO6G. It has 24,503 constraints and order of the matrix variables $n = 630$.

These large-scale benchmark problems play an important role in the research and development of improved IPM's software for semidefinite optimization. Next, we will describe in detail a number of applications in combinatorial optimization that still challenge the modern day SDO solvers.

1.3.1 Bounding the crossing number of complete bipartite graphs

The *crossing number* of a graph G, denoted by $\mathrm{cr}(G)$, is the minimum number of intersections of edges (at a point other than a vertex) in a drawing of G in the plane.

A complete bipartite graph $G := (V_1 \cup V_2, E)$ is a bipartite graph such that for any two vertices $v_1 \in V_1$ and $v_2 \in V_2$, v_1v_2 is an edge in G. The complete bipartite graph with partitions of size $|V_1| = m$ and $|V_2| = n$, is denoted $K_{m,n}$.

In the earliest known instance of a crossing number question, Paul Turán raised the problem of calculating the crossing number of $K_{m,n}$.

Zarankiewicz published a paper [116] in 1954, in which he claimed that $\mathrm{cr}(K_{m,n}) = Z(m,n)$ for all positive integers m and n, where

$$Z(m,n) = \left\lfloor \frac{m-1}{2} \right\rfloor \left\lfloor \frac{m}{2} \right\rfloor \left\lfloor \frac{n-1}{2} \right\rfloor \left\lfloor \frac{n}{2} \right\rfloor. \tag{1.1}$$

However, several years later Ringel and Kainen independently found a gap in Zarankiewicz's argument. A comprehensive account of the history of the problem, including a discussion of the gap in Zarankiewicz's argument, is given by Guy [57].

Figure 1.1 shows a drawing of $K_{4,5}$ with 8 crossings. As Zarankiewicz observed, this kind of a drawing strategy can be naturally generalized to construct, for any positive integers m and n, drawings of $K_{m,n}$ with exactly $Z(m,n)$ crossings. This implies the following well–known upper bound for $\mathrm{cr}(K_{m,n})$:

$$\mathrm{cr}(K_{m,n}) \leq Z(m,n).$$

No one has yet succeeded in drawing any $K_{m,n}$ with fewer than $Z(m,n)$ crossings.

[3] http://dimacs.rutgers.edu/Challenges/Seventh/Instances/

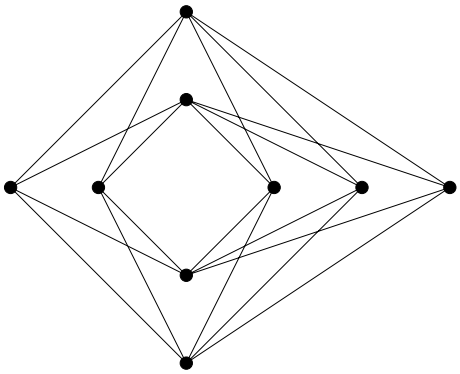

Figure 1.1: *A drawing of $K_{4,5}$ with 8 crossings.*

In allusion to Zarankiewicz's failed attempt to prove that this is the crossing number of $K_{m,n}$, the following is commonly known as *Zarankiewicz's Crossing–Number Conjecture*:

$$\mathrm{cr}(K_{m,n}) \stackrel{?}{=} Z(m,n), \qquad \text{for all positive integers } m,n.$$

A lower bound on cr$(K_{m,n})$ via SDO

De Klerk et al. [30] showed that one may obtain a lower bound on $\mathrm{cr}(K_{m,n})$ via the optimal value of the following SDO problem

$$\begin{aligned}\mathrm{cr}(K_{m,n}) &\geq \frac{n}{2}\left(n\min\{x^TQx \;\mid\; x\in\mathbb{R}_+^{(m-1)!}, e^Tx=1\} - \left\lfloor\frac{m}{2}\right\rfloor\left\lfloor\frac{m-1}{2}\right\rfloor\right)\\ &\geq \frac{n}{2}\left(n\min_{X\succeq 0,\, X\geq 0}\{\mathbf{Tr}(QX) \;\mid\; \mathbf{Tr}(JX)=1\} - \left\lfloor\frac{m}{2}\right\rfloor\left\lfloor\frac{m-1}{2}\right\rfloor\right),\end{aligned}$$

where Q is a certain (given) matrix of order $(m-1)!$ (i.e. $m-1$ factorial), J is the all-ones matrix of the same size, and e is the all ones vector.

De Klerk and al. [30] could solve the SDO problem for $m=7$ by exploiting the algebraic structure of the matrix Q, to obtain the bound:

$$\mathrm{cr}(K_{7,n}) > 2.1796n^2 - 4.5n.$$

Using an averaging argument, the bound for $\mathrm{cr}(K_{7,n})$ implies the following asymptotic bound

$$\lim_{n\to\infty}\frac{\mathrm{cr}(K_{m,n})}{Z(m,n)} \geq 0.83\frac{m}{m-1}.$$

Hence, asymptotically, $Z(m,n)$ and $\mathrm{cr}(K_{m,n})$ do not differ by more than 17% loosely speaking.

In subsequent, related work, De Klerk, Schrijver and Pasechnik [32] improved the constant 0.83 to 0.859 by solving the SDO for $m = 9$. This was possible by using a more sophisticated way when exploiting the algebraic structure of Q. Nevertheless, the final SDO formulation had more than 4×10^7 nonzero data entries.

The (reduced) SDO has not yet been solved for $m = 10$. Thus these SDO problems form an interesting and challenging test set of SDO instances.

1.3.2 SDO relaxation of the quadratic assignment problem

The quadratic assignment problem (QAP) may be stated in the following form:

$$\min_{X \in \Pi_n} \mathbf{Tr}(AXBX^T) \tag{1.2}$$

where A and B are certain $n \times n$ symmetric matrices, and Π_n is the set of $n \times n$ permutation matrices.

The QAP has many applications in facility location, circuit design, graph isomorphism and other problems, but is NP-hard in the strong sense, and hard to solve in practice for $n \geq 30$; for a review, see Anstreicher [6].

An SDO relaxation of (QAP) from [118] and [35] takes the form:

$$\begin{aligned} \min_{Y} \ & \mathbf{Tr}(B \otimes A)Y \\ \text{s.t.} \ & \mathbf{Tr}((I \otimes (J-I))Y + ((J-I) \otimes I)Y) = 0, \\ & \mathbf{Tr}(Y) - 2e^T y = -n, \\ & \begin{pmatrix} 1 & y^T \\ y & Y \end{pmatrix} \succeq 0, \\ & Y \geq 0, \end{aligned} \tag{1.3}$$

where I and J denote the identity matrix and the all-ones matrix respectively, and e is the all ones vector of size n. *The Kronecker* product, or tensor product, of two matrices is denoted by $B \otimes A$. Let $vec(A)$ denote the vector formed from the columns of the matrix A and $diag(B)$ is the main diagonal of matrix B.

It is easy to verify that this is indeed a relaxation of QAP, by setting $y = \mathrm{diag}\,(Y)$ and $Y = \mathrm{vec}(X)\mathrm{vec}(X)^T$. Then formulation (1.3) has a feasible solution if $X \in \Pi_n$.

These SDO problems form challenging test instances, since the matrix variable Y is nonnegative and of order n^2.

Special case: the traveling salesman problem

It is well-known that the QAP contains the symmetric traveling salesman problem (TSP) as a special case. To show this, we denote the complete graph on n vertices

with edge lengths (weights) $D_{ij} = D_{ji} > 0$ for $(i \neq j)$, by $K_n(D)$, where D is called the matrix of edge lengths (weights). The TSP is to find a Hamiltonian circuit of minimum length in $K_n(D)$. A Hamiltonian circuit (or Hamiltonian cycle) is a circuit in an undirected graph which visits each vertex exactly once and also returns to the starting vertex.

The n vertices are often called *cities*, and the Hamiltonian circuit of minimum length the *optimal tour*.

To see that TSP is a special case of QAP, let S_1 denote the adjacency matrix of the standard circuit on n vertices:

$$S_1 := \begin{bmatrix} 0 & 1 & 0 & \cdots & 0 & 1 \\ 1 & 0 & 1 & 0 & \cdots & 0 \\ 0 & 1 & 0 & 1 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \\ 0 & & & & 0 & 1 \\ 1 & 0 & \cdots & 0 & 1 & 0 \end{bmatrix}.$$

Now the TSP problem is obtained from the QAP problem (1.2) by setting $A = \frac{1}{2}D$ and $B = S_1$. To see this, note that every Hamiltonian circuit in a complete graph has adjacency matrix XS_1X^T for some $X \in \Pi_n$. Thus we may formulate the TSP as

$$\min_{X \in \Pi_n} \mathbf{Tr}\left(\frac{1}{2}DXS_1X^T\right).$$

It was shown by De Klerk et al. [33] that the SDO relaxation (1.3) reduces to the following problem for the special case of TSP.

$$\begin{aligned} \min_{X_{(1)}} \ & \frac{1}{2}\mathbf{Tr}\left(DX_{(1)}\right) \\ \text{s.t.} \quad & X_{(k)} \geq 0, \qquad k = 1, \ldots, d, \\ & \sum_{k=1}^{d} X_{(k)} = J - I, \\ I + \sum_{k=1}^{d} \cos\left(\frac{2\pi ik}{n}\right) X_{(k)} & \succeq 0, \qquad i = 1, \ldots, d, \\ & X_{(k)} \in \mathcal{S}^n, \qquad k = 1, \ldots, d, \end{aligned}$$

where $d = \lfloor \frac{1}{2}n \rfloor$.

Note that this problem only involves matrix variables $X_{(1)}, \ldots, X_{(d)}$ of order n as opposed to the matrix variable of order n^2 in (1.3), i.e. the problem size is reduced by a factor n in this sense. Nevertheless, these SDO problems are of a challenging size if $n \geq 30$.

1.4 The scope of this thesis

This thesis deals with solving large-scale semidefinite optimization problems by using interior-point methods and parallel computations. Next we give a summary of the content of each chapter.

In Chapter 2 we revisit the topic of the computational complexity of the primal-dual interior-point method for SDO. Our main focus is on the time per iteration complexity for the three most popular search directions in semidefinite optimization solvers: the Alizadeh–Haeberly–Overton [3], the Helmberg–Kojima–Monteiro [60, 71, 81] and the Nesterov–Todd [88] directions. We also outline the most computationally intensive operations involved in the primal-dual interior-point algorithm.

Chapter 3 contains a review of the different parallel computer architectures from the programming point of view. We also discuss the suitability of the parallel systems for practical implementation of IPM's. Finally, we motivate our choice of a computer architecture for development of a new parallel primal-dual IPM solver for semidefinite optimization.

Chapter 4 describes related work and problems in parallelization of interior-point methods for linear and semidefinite optimization. We focus on the different strategies to identify and use parallel computation to speed-up interior-point algorithms, in particular primal-dual methods for SDO. We give a discussion on the computational overhead caused by performing tasks in parallel.

Chapter 5 contains the algorithmic framework behind our newly developed primal-dual interior-point method solver for SDO using the Helmberg–Kojima–Monteiro search direction. We introduce there a different parallel computational approach than the existing IPM solvers. We also describe the new feature in our software that allows to deal explicitly with rank-one constraint matrices in SDO problems.

Chapter 6 presents the computational results from the numerical tests of our software. We compare also the scalability between our IPM solver and other parallel solvers for SDO. Results from the test of the rank-one feature in our solver are presented too.

In Chapter 7 we give concluding remarks, open problems and suggestions for further research.

Chapter 2

Computational complexity of IPM's

In this chapter we review the computational complexity of primal-dual interior-point methods (IPMs) for semidefinite optimization (SDO). The main focus is on the time per iteration complexity of three popular search directions for SDO, namely the search directions Alizadeh–Haeberly–Overton (AHO) [3], the Helmberg-Kojima–Monteiro (HKM) [60, 71, 81], and the Nesterov–Todd (NT) [88]. Special attention is paid to the most computationally intensive operations involved for each search direction. At the end some additional overhead computations in primal-dual IPM's for SDO are discussed.

2.1 Complexity

The goal of complexity theory is to provide an insight on how efficiently a certain type or class of problems can be solved. There are two main models of computation used to determine the complexity of a problem class.

1) The classical approach was introduced in the 1930s and is called Turing machine model, see for details Garey and Johnson [48]. It uses only integer numbers. The size of the problem data in Turing machines is measured in terms of bits. Therefore, complexity results for the Turing model are sometimes referred as a bit complexity. In terms of this model one arithmetic operation on two numbers, say multiplication of x and y, takes time that depends on the bit length of both x and y. As a result, complexity results in the Turing model depend on L, where L is defined as the total length of the data string, in bits, that represents the problem instance. Turing machine do not correspond well to the actual computers because they use only integer numbers. Most modern computers carry out numerical computations using floating-point numbers, that is numbers with a finite number of digits.

Despite this deficiency, the Turing model has been a convenient framework for analyzing interior-point algorithms, see e.g. [90].

2) Another computational model was proposed by Blum, Shub and Smale [17] in the 1980s. Their model for computation is based on a real number data representation and is often referred as the real-number model or Blum-Shub-Smale (BSS) model. It is assumed that the algorithm in this model performs exact real arithmetic. One operation, such as multiplication or addition of two numbers x and y, takes the same time independently of the values of x and y. Compared with Turing machines this approach seems closer to the way modern computers work, see Blum et. al [16]. Unfortunately, the real-number model has two shortcomings in computational practice. The first one is that on actual computers all real numbers are truncated or rounded to floating-point numbers. This process of rounding introduces a round-off error that could affect the numerical computations significantly. The second problem is that all arithmetic operations on floating-point numbers in a computer are not exact. As a result, an additional round-off error is introduced on almost every operation. Hence, the real-number model has its own drawbacks. For example, the complexity of the linear optimization (LO) problem is known in terms of the Turing model, but still not in terms of the real-number model.

Worst-case complexity of algorithms

Independent of the choice of computational model, complexity theory makes a fundamental distinction between two kind of algorithms: polynomial and exponential algorithms. Polynomial-time algorithms are those with time complexity function bounded from above by $p(n)$ for some polynomial function p, where $n \geq 0$ is used to denote the problem size, see Garey and Johnson [48]. Here, time complexity function for an algorithm is defined as the largest amount of time needed by the algorithm to solve a problem instance of certain predefined size. Note that p above is independent of the problem instance. Any algorithm whose time complexity function cannot be bounded above by a polynomial function is of the second type, and is called an exponential algorithm.

A polynomial-time algorithm is said to be strongly polynomial time, if the number of arithmetic operations of the algorithm is bounded in the dimension of the matrices and vectors of an instance. Dimension in this case means the size, not the bit length of the vectors and matrices. For example, checking that a matrix with dimension n is positive semidefinite requires $\mathcal{O}(n^3)$ arithmetic operations, so it is strongly polynomial.

Complexity results based on a time complexity function defined as above are worst-case results. A well-known example for this is the simplex algorithm, proven to have exponential complexity by Klee and Minty [69]. It was observed in practice that often instances of LO require far less solution time than expected. Due to

such results an average-case analysis was developed. Using average-case analysis on the simplex algorithm resulted in expected polynomial running time for a large class of problems, see Anstreicher at al. [7].

In what follows we restrict our discussion of complexity to worst-case behavior. Broader treatment of general complexity definitions, models and results can be found in the books of Garey and Johnson [48], Blum et. al [16] and Vavasis [111].

One has to distinguish also between complexity of a given algorithm for a problem class and the complexity of the problem class itself. Some basic information about the complexity of problem classes is given in the next section.

Complexity of problem classes

The main focus of the complexity theory framework are decision problems, see e.g. [48]. Such problems have only two possible solutions or answers, 'yes' or 'no'. Since a decision problem requires a verification of a statement, the term 'decided' is used instead of 'solved' for such problems. Next we give information about the three basic complexity classes relevant for IPM's. A complexity class is a class of problems that satisfy a certain computational bound.

The class P consists of problems that can be decided in polynomial time, i.e., there exist an algorithm that requires only a polynomial number of operations. The class NP consists of problems that given a certificate, or guess solution, a 'yes' answer can be verified in polynomial time. Notice that P $\subseteq$ NP. It is not known if P = NP, but is widely believed that this is not the case, see e.g. [90]. The class co-NP includes the problems that given a certificate, a 'no' answer can be verified in polynomial time. As a result P $\subseteq$ co-NP. We also refer to a problem as intractable if there is no polynomial-time algorithm that can solve it.

Assigning an optimization problem to a specific complexity class requires the problem to be translated into a decision problem first. Certain types of linear optimization and quadratic optimization problems can be transformed into decision problems, see e.g. Papadimitriou et al. [90] and Vavasis [111]. For example, consider the linear optimization problem in primal form

$$\min_x \left\{ c^T x : Ax = b,\ x \geq 0 \right\}.$$

Its corresponding dual problem is defined by

$$\max_{y,s} \left\{ b^T y : A^T y + s = c,\ s \geq 0 \right\}.$$

The linear optimization problem translates into a decision problem by asking: are there x, y and s such that

$$Ax = b,\ A^T y + s = c,\ x^T s = 0,\ x \geq 0,\ s \geq 0?$$

2.1.1 Complexity of primal-dual IPM's

Complexity had an important role for establishing interior-point methods. The search for a method with a better complexity than the ellipsoid method for LO of Khaciyan [68] has lead to a polynomial-time interior-point method, proposed by Karmakar [66]. There are many papers that prove results about the complexity of IPMs, see e.g. Renegar [92], Nesterov and Nemirovskii [87], Vavasis [111]. Most of the complexity results for IPM's were given in terms of the Turing model, see [112]. We outline here just the aspects that are relevant to primal-dual IPM algorithms for SDO since they are the primary objective of this thesis.

Primal-dual IPMs for LO are known [87, p. 246] to be polynomial time in the bit complexity framework and hence are computationally tractable. Existence of a polynomial algorithm for LP in the framework of the real-number model is still an open question. Traub and Woźniakowski [108] conjectured that there is no polynomial-time algorithm in terms of the real number-model.

Interior-point algorithms for semidefinite optimization are in class NP$\cap$co-NP in the Turing model as well as in the real-number model. In practice, instead of the exact solution it is enough to obtain an SDO solution with certain predefined accuracy $\epsilon \in (0, 1)$. For such so-called ϵ-approximate SDO solutions there exist polynomial-time IPM algorithms, see Nesterov and Nemirovskii [87].

In the analysis that follows we focus on the practical numerical aspects of primal-dual IPMs for SDO. Therefore, when we refer to the complexity of different search directions for SDO, we mean the computational complexity per iteration and not the worst-case iteration complexity.

2.1.2 Practice vs Theory

On actual computers that typically use 32-bit or 64-bit data representations and operations, complexity of algorithms is expressed in terms of flops. According to Golub [53], a *flop* denotes one floating-point operation. Results in terms of flops are different from both the bit and the real-number model. When using flops, any arithmetic operation between two real numbers may be done at unit cost, but the computations are inexact. On the other hand flops use data representation as in the bit model, but with finite and fixed precision.

In spite of this difference we implement algorithms on practical computers, but still often refer to complexity results that are really only valid for the 'idealized' models of computation (e.g. Turing machines or the real-number model). Thus, a complexity result that states"an algorithm solves the problem in $\mathcal{O}(n^3)$" typically refers to the Turing model, and not to the practical computation of our everyday experience.

There have been attempts to bring 'practice' and 'theory' closer by studying computational models that use flops with a fixed (but large) numbers of bits, or using arbitrary-precision arithmetic in software like Mathematica[1], but these

[1] http://www.wolfram.com/

paradigms are still somewhat different from daily practice.

With these considerations in mind, we describe later on in this chapter the complexity (in terms of the real-number model) of a primal-dual interior-point method for SDO using the Mehrotra predictor-corrector strategy.

2.2 Primal-dual IPM for SDO

Recall from Section 1.1 that the semidefinite optimization problem in the primal standard form can be written as

$$\begin{aligned} (P) \qquad & \min_X \mathbf{Tr}(CX) \\ & \text{s.t. } \mathbf{Tr}(A_i X) = b_i, \quad i = 1, ..., m, \\ & \qquad X \succeq 0, \end{aligned} \tag{2.1}$$

where $X \in \mathbb{S}^n_+$ is a symmetric positive semidefinite (psd) matrix, $C \in \mathbb{S}^n$, $A_i \in \mathbb{S}^n$ and $b_i \in \mathbb{R}^m$ for $i = 1, ..., m$.
The (Lagrangean) dual of (P) in standard form is given by

$$\begin{aligned} (D) \qquad & \max_{y,Z} b^T y \\ & \text{s.t.} \sum_{i=1}^m y_i A_i + Z = C, \\ & \qquad Z \succeq 0, \end{aligned} \tag{2.2}$$

where $y_i \in \mathbb{R}^m$ and $Z \in \mathbb{S}^n_+$. Recall from Chapter 1, that the relaxed optimality conditions used in interior-point methods for SDO are:

$$\begin{aligned} b_i - \mathbf{Tr}(A_i X) = 0, \quad & i = 1, ..., m, \\ C - Z - \sum_{i=1}^m y_i A_i = 0, & \\ XZ = \mu I, & \\ X, Z \succ 0. & \end{aligned} \tag{2.3}$$

Primal-dual path-following IPMs use Newton's method to find approximates for the solution of (2.3) for a decreasing sequence of values of the positive parameter μ. There are primal-dual interior-point methods that require a feasible starting point, as well as some that do not require such. They are known as feasible and infeasible start methods, respectively.

In the analysis that follows next we restrict ourselves to infeasible start primal-dual interior-point methods for SDO, since they are widely used in practical implementations. These methods need a strictly feasible initial point (X, y, Z) to obtain primal and dual directions $\triangle X$ and $\triangle y, \triangle Z$, respectively, that satisfy

$$\begin{aligned} \mathbf{Tr}(A_i \triangle X) = b_i - \mathbf{Tr}(A_i X), \quad & i = 1, ..., m, \\ \textstyle\sum_{i=1}^m \triangle y_i A_i + \triangle Z = C - Z - \sum_{i=1}^m y_i A_i, & \\ (X + \triangle X)(Z + \triangle Z) = \mu I, & \end{aligned} \tag{2.4}$$

as well as $X + \triangle X \succeq 0$ and $Z + \triangle Z \succeq 0$. It is easy to see that the last equation in (2.4) is nonlinear. A widely used approach to make it linear is to drop out the nonlinear term $\triangle X \triangle Z$. Hence, the above system becomes

$$\begin{aligned} \mathbf{Tr}(A_i \triangle X) &= b_i - \mathbf{Tr}(A_i X), \quad i = 1, ..., m, \\ \textstyle\sum_{i=1}^m \triangle y_i A_i + \triangle Z &= C - Z - \textstyle\sum_{i=1}^m y_i A_i, \\ \triangle X Z + X \triangle Z &= \mu I - XZ. \end{aligned} \tag{2.5}$$

Here $\triangle X$ and $\triangle Z$ are required to be symmetric. This is always true for $\triangle Z$ by the second equation of the (2.5), but for $\triangle X$ this may not be the case. As a result a symmetrization is needed for the last equation.

Zhang [117] suggested to replace the equation $\triangle X Z + X \triangle Z = \mu I - XZ$ by

$$H_P(\triangle X Z + X \triangle Z) = \mu I - H_P(XZ),$$

where H_P is a linear transformation given by

$$H_P(M) = \frac{1}{2}\left(PMP^{-1} + (PMP^{-1})^T\right), \tag{2.6}$$

for any $M \in \mathbb{R}^{n \times n}$ and a nonsingular scaling matrix P. The symmetrization strategies depend on P and some popular choices are listed in Table 2.1.

P	**Reference**
I	Alizadeh et al. [3]
$Z^{\frac{1}{2}}$	Monteiro [81], Helmberg et al. [60], Kojima et al. [71]
$\left[X^{-\frac{1}{2}}(X^{\frac{1}{2}} Z X^{\frac{1}{2}})^{\frac{1}{2}} X^{-\frac{1}{2}}\right]^{\frac{1}{2}}$	Nesterov and Todd [88]

Table 2.1: *Choices for the scaling matrix P.*

There are more than twenty search directions for SDO, see Todd [103]. Our main focus is on three of them, namely the AHO, the HKM and the NT. The reason to choose HKM and NT is that they have various desirable properties that make them very efficient in practice, see [83, 103]. The AHO was chosen since it pioneered the field of IPM's for semidefinite optimization and has good theoretical properties.

If the current point for system (2.5) is (X, y, Z) then we can expand the third equation of the system as

$$\mu I - H_P(XZ) = H_P(\triangle X Z + X \triangle Z) = H_P(\triangle X Z) + H_P(X \triangle Z).$$

Hence, the search direction $(\triangle X, \triangle y, \triangle Z)$ at (X, y, Z) satisfies

$$\begin{aligned} \mathbf{Tr}(A_i \triangle X) &= r_i, \ i = 1, ..., m, \\ \textstyle\sum_{i=1}^m \triangle y_i A_i + \triangle Z &= R, \\ H_P(\triangle X Z) + H_P(X \triangle Z) &= \mu I - H_P(XZ), \end{aligned}$$

where the vector r_i is defined as

$$r_i := b_i - \mathbf{Tr}(A_i X), \quad i = 1, ..., m, \tag{2.7}$$

and the matrix R is

$$R := C - Z - \textstyle\sum_{i=1}^{m} y_i A_i. \tag{2.8}$$

A unified framework for the analysis of different search directions in terms of linear operators is given by Monteiro and Zanjácomo [83]. Using a similar framework, we will show how to compute the search direction in terms of the linear operators $\mathcal{E} : \mathbb{S}^n \to \mathbb{S}^n$ defined as

$$\mathcal{E}(E) := H_P(EZ), \quad \text{for all } E \in \mathbb{S}^n, \tag{2.9}$$

and $\mathcal{F} : \mathbb{S}^n \to \mathbb{S}^n$ is

$$\mathcal{F}(F) := H_P(XF), \quad \text{for all } F \in \mathbb{S}^n. \tag{2.10}$$

Using $\mathcal{E}$ and $\mathcal{F}$, we can write the system of equations (2.5) defining the search direction as

$$\begin{array}{rcl} \mathbf{Tr}(A_i \triangle X) & = & r_i, \ i = 1, ..., m, \\ \sum_{i=1}^{m} \triangle y_i A_i + \triangle Z & = & R, \\ \mathcal{E}(\triangle X) + \mathcal{F}(\triangle Z) & = & H, \end{array} \tag{2.11}$$

where H is defined as

$$H := \mu I - H_P(XZ). \tag{2.12}$$

The vectors r_i and matrix R are defined as in (2.7) and (2.8), respectively. The following result from [83] provides a scheme for computing the search direction $(\triangle X, \triangle y, \triangle Z)$ in terms of the operators $\mathcal{E}$ and $\mathcal{F}$.

Lemma 2.2.1. *Let $\mathcal{E} : \mathbb{S}^n \to \mathbb{S}^n$ and $\mathcal{F} : \mathbb{S}^n \to \mathbb{S}^n$ be linear operators, defined by (2.9) and (2.10), and such that $\mathcal{E}$ has an inverse and let $(r, R, H) \in \mathbb{R}^m \times \mathbb{S}^n \times \mathbb{S}^n$ and $A_i \in \mathbb{S}^n, i = 1, ..., m$ be given. Let*

$$\begin{array}{rcll} V_j & := & \mathcal{E}^{-1}(\mathcal{F}(A_j)), \ j = 1, ..., m, & (2.13) \\ V & := & \mathcal{E}^{-1}(\mathcal{F}(R) - H), & (2.14) \\ M_{ij} & := & \mathbf{Tr}(A_i V_j), \ i, j = 1, ..., m, & (2.15) \\ h_i & := & r_i + \mathbf{Tr}(A_i V), & (2.16) \end{array}$$

where M_{ij} and h_i for $i, j = 1, ..., m$ are the elements of matrix $M \in \mathbb{R}^{m \times m}$ and vector $h \in \mathbb{R}^m$, respectively.
Then $(\triangle X, \triangle y, \triangle Z)$ satisfies system (2.11) if and only if $\triangle y$ satisfies the system

$$M \triangle y = h, \tag{2.17}$$

and $(\triangle X, \triangle Z)$ *is given by*

$$\begin{aligned} \triangle X &= \textstyle\sum_{j=1}^{m} \triangle y_j V_j - V, & (2.18)\\ \triangle Z &= R - \textstyle\sum_{i=1}^{m} \triangle y_i A_i. & (2.19) \end{aligned}$$

In particular, if the matrix M *is invertible, then system (2.11) has exactly one solution.*

Proof: Suppose that $(\triangle X, \triangle y, \triangle Z)$ satisfies (2.11). From the second equation in (2.11) we obtain (2.19). If we apply the operator $\mathcal{E}^{-1}$ to both sides of the third equation in (2.11) we get

$$\triangle X = \mathcal{E}^{-1}\left(H - \mathcal{F}(\triangle Z)\right). \tag{2.20}$$

Next we substitute the expression for $\triangle Z$ into (2.20) and using the definitions of V and V_j as in (2.14) and (2.13), respectively, we obtain

$$\begin{aligned} \triangle X &= \mathcal{E}^{-1}\left(H - \mathcal{F}(R - \textstyle\sum_{i=1}^{m} \triangle y_i A_i)\right),\\ &= \mathcal{E}^{-1}\left(H - \mathcal{F}(R) + \textstyle\sum_{i=1}^{m} \triangle y_i \mathcal{F}(A_i)\right),\\ &= \textstyle\sum_{i=1}^{m} \triangle y_i \mathcal{E}^{-1}\left(\mathcal{F}(A_i)\right) - \mathcal{E}^{-1}\left(\mathcal{F}(R) - H\right),\\ &= \textstyle\sum_{i=1}^{m} \triangle y_i V_i - V, \end{aligned}$$

which is exactly (2.18). Taking the inner product of both sides of

$$\triangle X = \textstyle\sum_{j=1}^{m} \triangle y_j V_j - V,$$

with A_i and using the first equation from (2.11) results to

$$\sum_{j=1}^{m} \mathbf{Tr}(A_i V_j) \triangle y_j - \mathbf{Tr}(A_i V) = \mathbf{Tr}(A_i \triangle X) = r_i, \quad i = 1, ..., m. \tag{2.21}$$

Rearranging the left-hand side and the right-hand of (2.21) we get

$$\sum_{j=1}^{m} \mathbf{Tr}(A_i V_j) \triangle y_j = r_i + \mathbf{Tr}(A_i V), \quad i = 1, ..., m. \tag{2.22}$$

It is easy to see that (2.22) can be written as (2.17), namely $M \triangle y = h$, for M_{ij} and h_i defined as in (2.15) and (2.16), respectively.

To prove the other implication, suppose $(\triangle X, \triangle y, \triangle Z)$ satisfies (2.17), (2.18) and (2.19). As a result the second equation in system (2.11) holds automatically, due to (2.19). Applying operator $\mathcal{E}$ to both sides of (2.18) and taking into account (2.15), (2.16) and (2.19) we get the first equality of (2.11). Next using equations (2.17), (2.15) and (2.16), we obtain

$$\begin{aligned} \mathcal{E}(\triangle X) &= \textstyle\sum_{i=1}^{m} \triangle y_i \mathcal{F}(A_j) - H + \mathcal{F}(R),\\ &= H - \mathcal{F}(R - \textstyle\sum_{i=1}^{m} \triangle y_i A_i),\\ &= H - \mathcal{F}(\triangle Z), \end{aligned}$$

which is exactly the third equation of system (2.11). We can use the same approach as before and take the inner product of both sides of (2.18) with A_i. This operation will result in (2.21) due to (2.15),(2.16) and (2.17). Hence, we obtain the first equation of (2.11) and the lemma is proved. □

There are different types of interior-point algorithms for SDO, for a survey on this topic, see Todd [104]. The most popular class of primal-dual algorithms are the path-following algorithms that use predictor-corrector strategy. They are the most efficient in practice and are widely used in SDO software. Since our aim is the best practical performance, we give more details about the primal-dual algorithm using the predictor-corrector strategy next.

The Mehrotra [78] predictor-corrector algorithm computes two directions per iteration, the predictor direction, called also affine-scaling direction, and the corrector direction. The affine-scaling direction $(\triangle X^a, \triangle y^a, \triangle Z^a)$ is the solution of the system (2.11) with $\mu = 0$, i.e., (2.12) is now $H = -H_P(XZ)$, and r_i and R are defined as in (2.7) and (2.8), respectively. Next, the triple $(\triangle X^a, \triangle y^a, \triangle Z^a)$ is used to compute the centering parameter $\sigma \in [0, 1)$ from

$$\sigma := \left[\frac{\mathbf{Tr}\left((X + \alpha_P^a \triangle X^a)(Z + \alpha_D^a \triangle Z^a)\right)}{\mathbf{Tr}(XZ)}\right]^2.$$

The step sizes α_P^a and α_D^a are computed from

$$\alpha_P^a = \min(1, \bar{\alpha}_P),\ \alpha_D^a = \min(1, \bar{\alpha}_D), \tag{2.23}$$

where

$$\begin{aligned} \bar{\alpha}_P &:= \max\{\alpha \geq 0 : X + \alpha \triangle X^a \succeq 0\}, \\ \bar{\alpha}_D &:= \max\{\alpha \geq 0 : Z + \alpha \triangle Z^a \succeq 0\}. \end{aligned}$$

Computation of the corrector direction $(\triangle X^c, \triangle y^c, \triangle Z^c)$ is done, see Monteiro [83], by solving the system

$$\begin{aligned} \mathbf{Tr}(A_i \triangle X^c) &= 0,\ i = 1, ..., m, \\ \textstyle\sum_{i=1}^m \triangle y_i^c A_i + \triangle Z^c &= 0, \\ \mathcal{E}(\triangle X^c) + \mathcal{F}(\triangle Z^c) &= \sigma\mu I - H_P(\triangle X^a \triangle Z^a), \end{aligned} \tag{2.24}$$

where

$$\mu := \frac{\mathbf{Tr}(XZ)}{n}. \tag{2.25}$$

Using (2.14), from Lemma 2.2.1, and (2.24), the matrix V^c for the corrector direction is defined by

$$V^c := \mathcal{E}^{-1}(H_P(\triangle X^a \triangle Z^a) - \sigma\mu I). \tag{2.26}$$

Generic Primal-Dual Algorithm for SDO with Predictor-Corrector Strategy

Input:
An accuracy parameter $\varepsilon > 0$;
Data matrices C, A_i, b_i where $i = 1, ..., m$;

begin
Construct initial solution $y_0 = 0$ and $X_0, Z_0 \succ 0$;
while $\mathbf{Tr}(XZ) > \varepsilon$ **do**
begin
Construct the affine-scaling direction ($\mu = 0$):
- Compute M and h by (2.15) and (2.16);
- Solve $M \triangle y^a = h$;
- Compute $\triangle X^a$ and $\triangle Z^a$ by (2.18) and (2.19);

Compute σ and the step sizes α_P^a and α_D^a from (2.23);
Fix target μ, (2.25);
Get the corrector direction;
- Compute h^c by (2.27);
- Solve $M \triangle y^c = h^c$;
- Compute $\triangle X^c$ and $\triangle Z^c$ by (2.29) and (2.19);

Compute a step sizes α_P and α_D by (2.31) and (2.32);
Add affine-scaling and corrector directions (2.30);
Update the current point:
- $\widehat{X} = X + \alpha_P \triangle X$; $\widehat{Z} = Z + \alpha_D \triangle Z$; $\widehat{y} = y + \alpha_D \triangle y$;

end
end

Figure 2.1: *The algorithm.*

The elements of the vector h, defined by (2.16), are computed for the corrector direction from

$$h_i^c := r_i + \mathbf{Tr}(A_i V^c), \quad i = 1, ..., m. \tag{2.27}$$

The process of solving (2.24) includes solving $M \triangle y^c = h^c$, where the elements of h^c are defined by (2.27). Note that for the corrector direction M is the same as for the affine-scaling direction. Using (2.24) and (2.26), $\triangle X^c$ and $\triangle Z^c$ are computed from

$$\triangle Z^c \quad := \quad -\textstyle\sum_{i=1}^m \triangle y_i^c A_i, \tag{2.28}$$

and

$$\triangle X^c := \mathcal{E}^{-1}(\sigma\mu I - H_P(\triangle X^a \triangle Z^a) - \mathcal{F}(\triangle Z^c)) = -V^c - \mathcal{E}^{-1}(\mathcal{F}(\triangle Z^c)). \quad (2.29)$$

The final search direction $(\triangle X, \triangle y, \triangle Z)$ is obtained by adding the affine-scaling and corrector directions

$$(\triangle X, \triangle y, \triangle Z) := (\triangle X^a, \triangle y^a, \triangle Z^a) + (\triangle X^c, \triangle y^c, \triangle Z^c). \quad (2.30)$$

The next iterate $(\widehat{X}, \widehat{y}, \widehat{Z})$ is obtained as

$$\begin{aligned} \widehat{X} &= X + \alpha_P \triangle X, \\ \widehat{y} &= y + \alpha_D \triangle y, \\ \widehat{Z} &= Z + \alpha_D \triangle Z, \end{aligned}$$

where α_P is the step size for (P) obtained by

$$\alpha_P = \min(1, \max\{\alpha \geq 0 : X + \alpha \triangle X \succeq 0\}), \quad (2.31)$$

and α_D is the step size for (D) computed as

$$\alpha_D = \min(1, \max\{\alpha \geq 0 : Z + \alpha \triangle Z \succeq 0\}). \quad (2.32)$$

A generic primal-dual algorithm using the predictor-corrector strategy is shown in Figure 2.1.

We will require the following result on several occasions in the next section. In particular, it gives a sufficient condition for the so-called Lyapunov equation to have an unique solution.

Lemma 2.2.2. *For every $A \in \mathbb{S}^n_{++}$ and $H \in \mathbb{S}^n$, the equation*

$$AU + UA = H \quad (2.33)$$

has a unique solution $U \in \mathbb{S}^n$.

Proof: See e.g. Graham [55]. □

As a matter of convenience, the unique solution U of (2.33) will also be denoted as $\ll H \gg_A$.

Since $Z \in \mathbb{S}^n_+$, it follows that there exist a unique $Z^{\frac{1}{2}} \in \mathbb{S}^n_+$ such that

$$Z^{\frac{1}{2}} Z^{\frac{1}{2}} = Z. \quad (2.34)$$

The matrix $Z^{\frac{1}{2}}$ is called the square root of Z. Due to similarity, the matrices $XZ, ZX, Z^{\frac{1}{2}} X Z^{\frac{1}{2}}$ have the same spectrum, which we denote and arrange as

$$\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_n > 0,$$

where $\lambda_i, i = 1, ..., n$ are the eigenvalues of XZ.

Making use of the general framework above, we next analyze the iteration complexity of the chosen three search directions for SDO, see Table 2.1. We show the steps involved in the computation of each of the directions.

2.3 Forming the matrix M

This section is dedicated to the complexity of computing the matrix M and the vector h in (2.17) for the search directions for SDO: AHO, HKM, and NT. The analysis presented in this section assumes that all matrices are in the dense format. When the A_i's are sparse, there are alternative methods that compute the matrix M more efficiently than the approaches shown in this section. More specifically for SDO problems with sparse A_i's having a random sparsity structures, Fujisawa et al. [46] have developed an efficient procedure to compute the matrix M in (2.15) for the HKM and NT search directions. More results on exploiting structure on a sparse SDO problems via matrix completion were introduced by Fukuda et al. [47].

For SDO with special structure of the matrices A_i, such as the max-cut SDO relaxation (see e.g. [60]), it is possible to obtain simplified expressions for the matrix M associated with the HKM and the NT directions, which speeds up their computations, e.g. see [60] and [11].

We show two approaches for computing the matrix M and the vector h in (2.17), defined by (2.15) and (2.16), respectively. The first approach is widely used in the literature and is using the original matrices A_i. The second approach was introduced by Monteiro and Zanjácomo [83] and it makes use of the scaled matrices $\tilde{A}_i$ for $i = 1, ..., m$, obtained from the original A_i's. This way of computing M has better complexity, but it requires the $\tilde{A}_i$'s to be explicitly available (in addition to the A_i's). Therefore, the second approach has extra storage requirements compared to the first approach.

2.3.1 AHO

This search direction introduced in [3] by Alizadeh, Haebery, and Overton, uses the symmetrization scheme that can be obtained from (2.6) by using $P = I$, where I denotes the identity matrix. As a result we get the direction specific expressions in (2.11) in terms of operators $\mathcal{E}$ and $\mathcal{F}$ as

$$\mathcal{E}(\triangle X) \quad := \quad H_I(\triangle X Z) = \frac{1}{2}(\triangle X Z + Z \triangle X), \tag{2.35}$$

and

$$\mathcal{F}(\triangle Z) \quad := \quad H_I(X \triangle Z) = \frac{1}{2}(X \triangle Z + \triangle Z X), \tag{2.36}$$

respectively, as well as for

$$H \quad := \quad \mu I - H_I(XZ). \tag{2.37}$$

From (2.35) follows that the matrices $V_j, i = 1, ..., m$ defined by (2.13) can be obtained as the unique solutions (by Lemma 2.2.2) of the following Lyapunov

equation

$$V_j Z + Z V_j \quad = \quad X A_j + A_j X. \tag{2.38}$$

Let

$$Z = Q \Lambda Q^T, \tag{2.39}$$

denote the eigenvalue decomposition of Z, where Λ is a diagonal matrix and Q is an orthogonal matrix, i.e. $Q^T Q = I$. Also, define

$$\tilde{V}_j := Q^T V_j Q, \ \tilde{X} := Q^T X Q, \ \tilde{A}_j := Q^T A_j Q, \ j = 1, ..., m. \tag{2.40}$$

First, we substitute Z in (2.38). Second, we multiply on the left by Q^T and by Q on the right. As a result, the Lyapunov equation (2.38) can be written in terms of the eigenvalues of Z as

$$\tilde{V}_j \Lambda + \Lambda \tilde{V}_j = \tilde{X} \tilde{A}_j + \tilde{A}_j \tilde{X}, \quad j = 1, ..., m. \tag{2.41}$$

A similar approach can be used for computing V, defined by (2.14), for the AHO direction. It can be obtained also as an unique solution of the following Lyapunov equation

$$V Z + Z V \quad = \quad X R + R X - 2H. \tag{2.42}$$

Let

$$\tilde{V} := Q^T V Q, \ \tilde{R} := Q^T R Q. \tag{2.43}$$

If we substitute Z in (2.42) and multiply on the left by Q^T and by Q on the right we get

$$\tilde{V} \Lambda + \Lambda \tilde{V} \quad = \quad \tilde{X} \tilde{R} + \tilde{R} \tilde{X} + \tilde{X} \Lambda + \Lambda \tilde{X} - 2\mu I. \tag{2.44}$$

In the first computational approach, using (2.40), M is computed as

$$M_{ij} = \mathbf{Tr}(A_i V_j) = \mathbf{Tr}\left(A_i (Q \tilde{V}_j Q^T) \right), \quad i, j = 1, ..., m, \tag{2.45}$$

and the elements of the vector h are obtained from

$$h_i = r_i + \mathbf{Tr}(A_i V) = r_i + \mathbf{Tr}\left(A_i (Q \tilde{V} Q^T) \right), \quad i = 1, ..., m. \tag{2.46}$$

Computation of the matrix products XQ and RQ requires $\mathcal{O}(n^3)$ flops. By using them, the product $\tilde{X}\tilde{R} = (XQ)^T RQ$ in (2.44) also requires $\mathcal{O}(n^3)$ flops. We also can pre-compute the product $A_j Q$ for $j = 1, ..., m$ in $2mn^3 + \mathcal{O}(mn^2)$ flops and use it along with XQ for obtaining $\tilde{X}\tilde{A}_j = (XQ)^T A_j Q$ in (2.44). In this way $\tilde{X}\tilde{A}_j$ is computed also in $2mn^3 + \mathcal{O}(mn^2)$ flops.

Computation of the matrices $\tilde{V}$ and $\tilde{V}_j, j = 1, ..., m$, involves solution of (2.41) and (2.44), i.e., $(m + 1)$ Lyapunov systems. These solutions are obtained in $\mathcal{O}(mn^2)$ flops, because they involve only the diagonal matrix Λ. The additional cost of the eigenvalue decomposition of Z to obtain Λ is negligible in comparison to the other dense matrix operations involved. Therefore, the computational complexity of this operation is not taken into account.

Matrices $V_j = Q\tilde{V}_j Q^T, i = 1, ..., m$ and $V = Q\tilde{V}Q^T$ take $2mn^3 + \mathcal{O}(mn^2 + n^3)$ flops. The $m(m+1)$ trace products $\mathbf{Tr}(A_iV)$ and $\mathbf{Tr}(A_iV_j), i, j = 1, ..., m$ in (2.45) and (2.46), respectively can be obtained in $m^2n^2 + \mathcal{O}(mn^2 + m^2n)$ flops.

Notice that using (2.40), (2.43) and $Q^TQ = I$, M and h can be written as

$$M_{ij} = \mathbf{Tr}(A_iV_j) = \mathbf{Tr}(Q^{-T}\tilde{A}_i\tilde{V}_jQ^{-1}) = \mathbf{Tr}(Q^{-1}Q^{-T}\tilde{A}_i\tilde{V}_j) = \mathbf{Tr}(\tilde{A}_i\tilde{V}_j), \quad (2.47)$$

and

$$h_i = r_i + \mathbf{Tr}(A_iV) = r_i + \mathbf{Tr}(\tilde{A}_i\tilde{V}), \quad (2.48)$$

for $i, j = 1, ..., m$, respectively.

The second approach to compute the matrix M and the vector h for the AHO direction is based on the results (2.47) and (2.48). In this case, computation of $\tilde{A}_j, j = 1, ..., m$ in (2.40) takes $\frac{5}{3}mn^3 + \mathcal{O}(mn^2)$ flops. This result is due to the way Monteiro and Zanjácomo [83] compute the matrix product $\tilde{A}_i := Q^TA_iQ$ for some i, where A_i is a symmetric matrix. Computing the matrix product $\tilde{X}\tilde{A}_j, j = 1, ..., m$ in (2.41) can be done in $2mn^3 + \mathcal{O}(mn^2)$ flops.

Finally, if we sum the flop counts for both methods for computing M, we get $6mn^3+m^2n^2+\mathcal{O}(mn^2+m^2n+n^3)$ flops and $\frac{11}{3}mn^3+m^2n^2+\mathcal{O}(mn^2+m^2n+n^3)$ flops by the first and the second computational approach, respectively. Note that these results include also computation of h. Since the vector h requires the solution of one Lyapunov equation and a trace of a matrix product the computational effort for h is significantly less than for the matrix M. Indeed, even if we exclude the flop count for h it is easy to see that summarized results we gave above will not change.

2.3.2 HKM

In this subsection we analyze the computational complexity to obtain M for the search direction corresponding to $P = Z^{\frac{1}{2}}$ in (2.6). This search direction is widely referred to as the HKM direction and is studied in several papers including Helmberg et. al. [60], Kojima et al. [71] and Monteiro [81]. We assume that all symmetric matrices are stored in triangular form.

Now using (2.6) with $P = Z^{\frac{1}{2}}$ and the definition of $\mathcal{E}$ and $\mathcal{F}$, the HKM direction is the solution of the system (2.11) with $\mathcal{E}(\triangle X)$ and $\mathcal{F}(\triangle Z)$ defined as

$$\mathcal{E}(\triangle X) \quad := \quad H_{Z^{\frac{1}{2}}}(\triangle XZ) = Z^{\frac{1}{2}}\triangle XZ^{\frac{1}{2}}, \quad (2.49)$$

and

$$\mathcal{F}(\triangle Z) \;:=\; H_{Z^{\frac{1}{2}}}(X\triangle Z) = \frac{1}{2}(Z^{\frac{1}{2}}X\triangle Z Z^{-\frac{1}{2}} + Z^{-\frac{1}{2}}\triangle Z X Z^{\frac{1}{2}}), \quad (2.50)$$

respectively, and

$$H \;:=\; H_{Z^{\frac{1}{2}}}(\mu I - XZ) = \mu I - Z^{\frac{1}{2}}XZ^{\frac{1}{2}}. \quad (2.51)$$

The matrices V_j defined by (2.13) can be obtained from the expression,

$$V_j \;=\; \frac{1}{2}(XA_jZ^{-1} + Z^{-1}A_jX),\ j = 1, ..., m, \quad (2.52)$$

and V, as in (2.14), is obtained from

$$V \;=\; \frac{1}{2}(XRZ^{-1} + Z^{-1}RX) - \mu Z^{-1} + X. \quad (2.53)$$

Hence, it follows from (2.15) and (2.52) that

$$\begin{aligned} M_{ij} &= \mathbf{Tr}(A_iV_j),\ i, j = 1, .., m, \\ &= \frac{1}{2}\left(\mathbf{Tr}(A_iXA_jZ^{-1}) + \mathbf{Tr}(A_iZ^{-1}A_jX)\right). \end{aligned} \quad (2.54)$$

Since Z^{-1}, X and A_i's are square symmetric matrices, the trace of the product $A_iXA_jZ^{-1}$ is invariant under cyclic permutations of the matrices involved, i.e.,

$$\mathbf{Tr}(A_iXA_jZ^{-1}) = \mathbf{Tr}(Z^{-1}A_iXA_j) = \mathbf{Tr}(A_jZ^{-1}A_iX). \quad (2.55)$$

Therefore, of we can write (2.54) as

$$M_{ij} = \frac{1}{2}\left(\mathbf{Tr}(A_jZ^{-1}A_iX) + \mathbf{Tr}(A_iZ^{-1}A_jX)\right) = \mathbf{Tr}(A_iZ^{-1}A_jX), \quad (2.56)$$

for all $i, j = 1, ..., m$. Note that interchanging the indexes i and j in (2.56) would not make a difference since M is a symmetric matrix for the HKM search direction. Using a similar argument as in (2.55) the elements of the vector h are computed from (2.15) and (2.53) by

$$h_i \;=\; r_i + \mathbf{Tr}\left(A_i(Z^{-1}RX - \mu Z^{-1} + X)\right),\ i = 1, ..., m. \quad (2.57)$$

In the first approach the m matrix products $Z^{-1}A_jX$ can be computed in $4mn^3 + \mathcal{O}(mn^2)$ flops. Given these m products (2.56) requires additionally $\frac{1}{2}m^2n^2 + \mathcal{O}(m^2n + mn^2)$ flops taking into account that $A_i, i = 1, ..., m$ are symmetric.

The second approach to compute the HKM direction is based on the following alternative way to compute M. Let L_X and $L_{Z^{-1}}$ denote the Cholesky factors of X and Z^{-1}, respectively. As a result we can write

$$M_{ij} \;=\; \mathbf{Tr}\left((L_X^TA_iL_{Z^{-1}})L_{Z^{-1}}^TA_jL_X\right),\ i, j = 1, .., m. \quad (2.58)$$

The elements of the vector h are obtained as in (2.53). Computation of matrix products $L_{Z^{-1}}^T A_i L_X, i = 1, ..., m$ takes $\frac{4}{3}mn^3 + \mathcal{O}(mn^2)$ flops. Since the matrix M is symmetric, only $\frac{m}{2}(m+1)$ inner products (2.58) of two nonsymmetric matrices needs to be computed, which takes $m^2n^2 + \mathcal{O}(m^2n + mn^2)$ flops.

The total number of $4mn^3 + \frac{1}{2}m^2n^2 + \mathcal{O}(mn^2 + m^2n)$ flops and $\frac{4}{3}mn^3 + m^2n^2 + \mathcal{O}(mn^2 + m^2n)$ flops are required, when computing M, using the first and the second approach, respectively.

2.3.3 NT

In this subsection we analyze the computational complexity to obtain the matrix M for the direction corresponding to map H_P with scaling matrix P, satisfying $P^T P = W^{-1}$, where

$$W := X^{\frac{1}{2}}(X^{\frac{1}{2}} Z X^{\frac{1}{2}})^{-\frac{1}{2}} X^{\frac{1}{2}}. \tag{2.59}$$

The resulting direction is widely referred to as the NT direction [89]. The NT direction and the corresponding path-following algorithms have been studied in several papers including Nesterov and Todd [89], Sturm and Zhang [101], and Todd, Toh and Tütüncü [105]. We derive in this subsection computational formulae for the NT direction based on the use of triangular matrices.

One possible choice for P is shown in the third line of Table 2.1 (on page 18). We use another choice, suggested in [105], for the scaling matrix P as follows. Let L_X denote the Cholesky factor of X and let

$$(L_X^T Z L_X)^{\frac{1}{2}} = Q D Q^T, \tag{2.60}$$

be the eigenvalue decomposition of the positive semidefinite matrix $(L_X^T Z L_X)^{\frac{1}{2}}$, where D is a diagonal matrix and Q is an orthogonal matrix. Define

$$P := D^{\frac{1}{2}} Q^T L_X^{-1}, \tag{2.61}$$

and

$$G := L_X^{-1} X^{\frac{1}{2}}. \tag{2.62}$$

The following lemma from [105] shows that G is an orthogonal matrix.

Lemma 2.3.1. *Suppose $B = CC^T$ is positive definite. Then $U := C^{-1}B^{\frac{1}{2}}$ is orthogonal.*

Proof: We see that $UU^T = C^{-1}BC^{-T} = C^{-1}CC^TC^{-T} = I$. □

Next we show that the scaling matrix P defined as in (2.61) satisfies $P^T P = W^{-1}$. Firstly, using (2.60) and (2.61) it follows that

$$P^T P = L_X^{-T} Q D Q^T L_X^{-1} = L_X^{-T} (L_X^T Z L_X)^{\frac{1}{2}} L_X^{-1}. \tag{2.63}$$

Secondly, by (2.62) we can write the matrix product $X^{\frac{1}{2}}ZX^{\frac{1}{2}}$ as

$$X^{\frac{1}{2}}ZX^{\frac{1}{2}} = G^T L_X^T Z L_X G.$$

Since G is orthogonal, we have

$$\left(X^{\frac{1}{2}}ZX^{\frac{1}{2}}\right)^{\frac{1}{2}} = G^T \left(L_X^T Z L_X\right)^{\frac{1}{2}} G.$$

Using this result and (2.59), W^{-1} can be written as

$$W^{-1} = X^{-\frac{1}{2}}(X^{\frac{1}{2}}ZX^{\frac{1}{2}})^{\frac{1}{2}}X^{-\frac{1}{2}} = L_X^{-T}\left(L_X^T Z L_X\right)^{\frac{1}{2}} L_X^{-1}. \tag{2.64}$$

Since the right hand sides of (2.63) and (2.64) are equal, it follows that $P^TP = W^{-1}$ holds for P defined as in (2.61).

It is shown in [105] that the NT direction is the solution of system (2.11) with $\mathcal{E}(\triangle X)$ and $\mathcal{F}(\triangle Z)$ equal to

$$\mathcal{E}(\triangle X) \;:=\; H_P(\triangle XZ) = \frac{1}{2}\left(P\triangle XZP^{-T} + P^{-1}Z\triangle XP^T\right), \tag{2.65}$$

and

$$\mathcal{F}(\triangle Z) \;:=\; H_P(X\triangle Z) = \frac{1}{2}\left(PX\triangle ZP^{-1} + P^{-T}\triangle ZXP^T\right), \tag{2.66}$$

respectively, and the direction specific expression for H as follows

$$H \;:=\; H_P(\mu I - XZ) = \mu I - \frac{1}{2}\left(PXZP^{-1} + P^{-T}ZXP^T\right). \tag{2.67}$$

We will require the following technical result for the NT search direction:

$$W^{-1}XW^{-1} = Z. \tag{2.68}$$

We can show that this is true by simply using the definition of W and $X = X^{\frac{1}{2}}X^{\frac{1}{2}}$ and expand $W^{-1}XW^{-1}$, namely

$$\begin{aligned} W^{-1}XW^{-1} &= X^{-\frac{1}{2}}(X^{\frac{1}{2}}ZX^{\frac{1}{2}})^{\frac{1}{2}}X^{-\frac{1}{2}}(X^{\frac{1}{2}}X^{\frac{1}{2}})X^{-\frac{1}{2}}(X^{\frac{1}{2}}ZX^{\frac{1}{2}})^{\frac{1}{2}}X^{-\frac{1}{2}} \\ &= X^{-\frac{1}{2}}(X^{\frac{1}{2}}ZX^{\frac{1}{2}})X^{-\frac{1}{2}} \\ &= Z. \end{aligned}$$

It is easy to see that (2.68) implies $X = WZW$. Due to (2.68), $PP^T = W^{-1}$ and the definition of P, it follows that

$$\begin{aligned} P^{-T}ZP^{-1} &= PXP^T, \\ &= D^{\frac{1}{2}}Q^T L_X^{-1} X L_X^{-T} Q D^{\frac{1}{2}}, \\ &= D^{\frac{1}{2}}Q^T (L_X^{-1}L_X)(L_X^T L_X^{-T}) Q D^{\frac{1}{2}}, \\ &= D^{\frac{1}{2}}Q^T Q D^{\frac{1}{2}}. \end{aligned}$$

Since Q is an orthogonal matrix, the last equality can be written as

$$P^{-T}ZP^{-1} = D. \tag{2.69}$$

Using (2.69) and $P^{-1}P = I$ we can reformulate H in (2.67) as

$$H = \mu I - \frac{1}{2}\left(PXZP^{-1} + P^{-T}ZXP^{T}\right) = \mu I - D^2.$$

Todd, Toh and Tütüncü in [105], using (2.65-2.67), have shown that for the NT direction, (2.11) is equivalent to

$$\begin{array}{rcl} \mathbf{Tr}(A_i\triangle X) & = & b - \mathbf{Tr}(A_iX), \quad i = 1, ..., m, \\ \sum_{i=1}^{m} \triangle y_i A_i + \triangle Z & = & C - Z - \sum_{i=1}^{m} y_i A_i, \\ W^{-1}\triangle X W^{-1} + \triangle Z & = & \mu X^{-1} - Z. \end{array} \tag{2.70}$$

Next, let us multiply the third equation in (2.70) on the left by W and on the right by WZ. Then taking into account that, by (2.68), $WX^{-1}WZ = I$, we get

$$\triangle XZ + W\triangle ZWZ = \mu I - XZ.$$

From this result it follows that (2.65) and (2.66) are equivalent to

$$\mathcal{E}(\triangle X) \quad := \quad \triangle XZ, \tag{2.71}$$

and

$$\mathcal{F}(\triangle Z) \quad := \quad W\triangle ZWZ, \tag{2.72}$$

respectively. The right hand side in the third equation of (2.70) is equivalent to the expression for H in (2.67), namely

$$H \quad := \quad \mu X^{-1} - Z. \tag{2.73}$$

Then using (2.71), (2.72) and (2.73) we can compute V_j and V, defined by (2.13) and (2.14), respectively as

$$V_j \quad = \quad \mathcal{E}^{-1}(\mathcal{F}(A_j)) = WA_jW, \quad j = 1, ..., m, \tag{2.74}$$

and

$$V \quad = \quad \mathcal{E}^{-1}(\mathcal{F}(R) - H) = WRW - \mu Z^{-1} + X. \tag{2.75}$$

The first approach for computing M, in the framework of the NT direction, uses expression (2.15), namely

$$M_{ij} \quad := \quad \mathbf{Tr}(A_iV_j), \; i, j = 1, ..., m.$$

The vector h, defined by (2.16), is computed from

$$h_i \quad := \quad r_i + \mathbf{Tr}(A_i V), \quad i = 1, ..., m.$$

Computing the $m+1$ matrices $V_j, j = 1, ..., m$ and V can be done in $2mn^3 + \mathcal{O}(mn^2+n^3)$ flops. Since M is symmetric, only $\frac{m}{2}(m+1)$ inner products $\mathbf{Tr}(A_i V_j)$ for $i, j = 1, ..., m$ need to be computed; moreover, as both factors in each inner product are symmetric, computation of these $\frac{m}{2}(m+1)$ inner products requires $\frac{1}{2}m^2n^2 + \mathcal{O}(m^2n + mn^2)$ flops.

The second approach to compute the matrix M is as follows. Let L_W denote the Cholesky factor of the matrix W, we can express the entries of M in (2.15) as

$$M_{ij} = \mathbf{Tr}\left((L_W^T A_i L_W) L_W^T A_j L_W\right), \; i, j = 1, ..., m. \tag{2.76}$$

The computation of the matrices $L_W^T A_i L_W, \; i = 1, ..., m$ requires $2mn^3 + \mathcal{O}(mn^2)$ flops. The matrix M is symmetric and both factors of each inner product (2.76) are symmetric matrices, we can compute the matrix product and the trace in (2.76) in $\frac{1}{2}m^2n^2 + \mathcal{O}(m^2n + mn^2)$ flops.

Finally, if we summarize the total amount of flops necessary to compute M, we get $2mn^3 + \frac{1}{2}m^2n^2 + \mathcal{O}(mn^2 + m^2n + n^3)$ flops and $\frac{2}{3}mn^3 + \frac{1}{2}m^2n^2 + \mathcal{O}(mn^2 + m^2n)$ flops by using the first and the second computational approach, respectively.

2.4 Search direction complexity

In what follows we describe the remaining steps involved in the computation of $(\triangle X, \triangle y, \triangle Z)$ for the three search directions. In Section (2.3) we presented two approaches for computing the matrix M and the vector h in (2.17) defined by (2.15) and (2.16), respectively. Each one of them implies a corresponding approach to compute the affine-scaling and corrector steps of the primal-dual direction. To simplify the presentation, we present the construction of these steps for the AHO, the HKM and the NT directions in terms of the first approach.

The flop counts of the three directions discussed in this section are expressed in a unified way, as proposed by Monteiro and Zanjácomo [83, p. 14]. These complexities do not include the flops spent on computing the matrix eigenvalue decompositions. As in the previous section we assume that all matrices are in dense format.

2.4.1 AHO

Recall from Section 2.3.1 results (2.35-2.38) and (2.42) for this direction. It follows that $\triangle y$ is the solution of the equation (2.17), namely $M \triangle y = h$ with M_{ij} and $h_i, \; i, j = 1, ..., m$ defined by (2.47) and (2.48), respectively. Computing $\triangle Z$ is done according to (2.19), namely

$$\triangle Z \quad = \quad R - \textstyle\sum_{i=1}^m \triangle y_i A_i,$$

where R is obtained from (2.8).

If we substitute (2.35-2.37) into the third equation from the system (2.11), we obtain

$$\begin{aligned} \triangle XZ + Z\triangle X &= 2H - (X\triangle Z + \triangle ZX) \\ &= 2\mu I - 2XZ - (X\triangle Z + \triangle ZX). \end{aligned} \tag{2.77}$$

Recall (2.34) and the property that matrices $XZ, ZX, Z^{\frac{1}{2}}XZ^{\frac{1}{2}}$ are similar, and so they have the same spectrum. As a result we can rewrite (2.77) as

$$Z^{\frac{1}{2}}\triangle XZ^{\frac{1}{2}} + Z^{\frac{1}{2}}\triangle XZ^{\frac{1}{2}} = 2\mu I - 2Z^{\frac{1}{2}}XZ^{\frac{1}{2}} - (X\triangle Z + \triangle ZX). \tag{2.78}$$

Next, if we rearrange the terms in (2.78) and multiply it on the right and on the left by $Z^{-\frac{1}{2}}$, we obtain

$$\triangle X = \mu Z^{-1} - X - \frac{1}{2}Z^{-\frac{1}{2}}(X\triangle Z + \triangle ZX)Z^{-\frac{1}{2}}. \tag{2.79}$$

Let U be defined by

$$U := \frac{1}{2}Z^{-\frac{1}{2}}(X\triangle Z + \triangle ZX)Z^{-\frac{1}{2}}.$$

Then, it is easy to see that U is the unique solution of the Lyapunov equation

$$ZU + UZ = X\triangle Z + \triangle ZX. \tag{2.80}$$

We can also write (2.80) as

$$U := \ll X\triangle Z + \triangle ZX \gg_Z . \tag{2.81}$$

If we substitute (2.81) into (2.79), we get the expression for $\triangle X$ for the AHO direction, namely

$$\triangle X = \mu Z^{-1} - X - \ll X\triangle Z + \triangle ZX \gg_Z .$$

Calculation of $(\triangle X, \triangle y, \triangle Z)$ requires $\frac{2}{3}m^3 + \mathcal{O}(mn^2 + n^3)$ flops.

For the Mehrotra predictor-corrector algorithm it is necessary first to compute the affine-scaling direction $(\triangle X^a, \triangle y^a, \triangle Z^a)$. It coincides with the above presented $(\triangle X, \triangle y, \triangle Z)$ when $\mu = 0$. This means that instead of H defined as in (2.37), it is given by

$$H := -H_I(XZ).$$

This change is not significant and does not change the complexity described above for $(\triangle X, \triangle y, \triangle Z)$. For the corrector direction we need to solve $M\triangle y^c = h^c$, see

e.g. Figure 2.1, where the elements of h^c are defined by (2.27). Matrix V^c, defined by (2.26), for the AHO direction is computed from

$$V^c := \mathcal{E}^{-1}(H_I(\triangle X \triangle Z) - \sigma\mu I).$$

Using (2.6) and (2.35) one can obtain V^c as a solution of the following Lyapunov equation

$$ZV^c + V^c Z = \triangle X \triangle Z + \triangle Z \triangle X - 2\sigma\mu I. \tag{2.82}$$

Using the notation introduced in (2.82) this means that

$$V^c = \ll \triangle X \triangle Z + \triangle Z \triangle X - 2\sigma\mu I \gg_Z .$$

It is possible to compute V^c using previously computed quantities for the affine-scaling direction, see Section 2.3.1. Recall (2.39), i.e., $Z = Q\Lambda Q^T$ is the eigenvalue decomposition of Z, and let $\widetilde{\triangle X} := Q^T \triangle X Q$, $\widetilde{\triangle Z} := Q^T \triangle Z Q$. Then (2.82) can be reformulated as

$$\Lambda V^c + V^c \Lambda = Q(\widetilde{\triangle X}\widetilde{\triangle Z} + \widetilde{\triangle Z}\widetilde{\triangle X})Q^T - 2\sigma\mu I, \tag{2.83}$$

or alternatively written as

$$V^c = Q \ll \widetilde{\triangle X}\widetilde{\triangle Z} + \widetilde{\triangle Z}\widetilde{\triangle X} - 2\mu I \gg_\Lambda Q^T.$$

The product $\widetilde{\triangle X}\widetilde{\triangle Z} = (Q^T \triangle X)(\triangle Z Q)$ can be computed in $4n^3 + \mathcal{O}(n^2)$ flops, since we may assume that $Q^T \triangle X$ is computed in the affine-scaling step. Subsequently, V^c can be found from (2.83) in $2n^3 + \mathcal{O}(n^2)$ flops. From (2.28) matrix $\triangle Z^c$ for the corrector step is computed by

$$\triangle Z^c \quad = \quad -\textstyle\sum_{j=1}^m \triangle y_j^c A_j.$$

To derive $\triangle X^c$ we use (2.29), (2.35) and (2.37). As a result we get

$$\triangle X^c = -\mathcal{E}^{-1}(X \triangle Z^c + \triangle Z^c X) - V^c. \tag{2.84}$$

Let

$$U^c := \mathcal{E}^{-1}(X \triangle Z^c + \triangle Z^c X). \tag{2.85}$$

Then, if the operator $\mathcal{E}$ is applied to both sides of (2.85) and by (2.35) follows that U^c is the unique solution of Lyapunov equation

$$ZU^c + U^c Z = X \triangle Z^c + \triangle Z^c X = Q\left[(Q^T X)\triangle Z^c Q\right] Q^T + \triangle Z^c Q (Q^T X), \tag{2.86}$$

where Q is an orthogonal matrix, see (2.39). Alternatively U^c can be expressed from (2.86) as

$$U^c := \ll X \triangle Z^c + \triangle Z^c X \gg_Z .$$

Finally, from (2.84) and (2.85) matrix $\triangle X^c$ is obtained by

$$\triangle X^c = -U^c - V^c.$$

Both $\triangle Z^c$ and $\triangle X^c$ above are computed in $6n^3 + \mathcal{O}(mn^2)$ flops using the precomputed expression $Q^T X$ from the affine-scaling step. Therefore, this corrector direction requires $12n^3 + \mathcal{O}(mn^2)$ flops in total.

2.4.2 HKM

In what follows, we continue the analysis from Section 2.3.2 and obtain the computational complexity of the HKM search direction using the Mehrotra predictor-corrector algorithm.

Recall results (2.49 - 2.53). Vector $\triangle y$ is solution of the equation (2.17), namely $M \triangle y = h$ with M_{ij} and h_i, $i, j = 1, ..., m$, defined by (2.56) and (2.57), respectively. Component $\triangle Z$ is computed from (2.19), namely

$$\triangle Z \quad = \quad R - \textstyle\sum_{i=1}^{m} \triangle y_i A_i.$$

Matrix R is obtained from (2.8). Using (2.49) we can compute $\triangle X$ from the expression

$$\triangle X = \mu Z^{-1} - X - \frac{1}{2}(X \triangle Z Z^{-1} + Z^{-1} \triangle Z X). \tag{2.87}$$

To compute the affine-scaling direction $(\triangle X^a, \triangle y^a, \triangle Z^a)$ for the Mehrotra predictor-corrector strategy, we fix $\mu = 0$ in (2.51). Therefore, we may assume the affine-scaling step has the same complexity as $(\triangle X, \triangle y, \triangle Z)$ described in Section 2.3.2.

We need to determine the computational complexity of the corrector step $(\triangle X^c, \triangle y^c, \triangle Z^c)$, for the HKM direction. From Lemma 2.2.1 follows that vector $\triangle y^c$ is a solution of the system $M \triangle y^c = h^c$, where h^c and V^c are computed by (2.27) and (2.26) for $P = Z^{\frac{1}{2}}$, namely

$$V^c := \mathcal{E}^{-1}(H_{Z^{\frac{1}{2}}}(\triangle X \triangle Z) - \sigma\mu I).$$

Using (2.49) and (2.50) one can obtain V^c from

$$Z^{\frac{1}{2}} V^c Z^{\frac{1}{2}} = \frac{1}{2}\left(Z^{\frac{1}{2}} \triangle X \triangle Z Z^{-\frac{1}{2}} + Z^{-\frac{1}{2}} \triangle Z \triangle X Z^{\frac{1}{2}}\right) - \sigma\mu I.$$

Next, multiplying the last equation from the left and the right by $Z^{-\frac{1}{2}}$ we get

$$V^c = \frac{1}{2}\left(\triangle X \triangle Z Z^{-1} + Z^{-1} \triangle Z \triangle X\right) - \sigma\mu Z^{-1}. \tag{2.88}$$

The elements of the vector h^c, using (2.27) and the argument in (2.56) for the trace of a symmetric matrix product, are given by

$$h_i^c = \mathbf{Tr}(A_i V^c) = \mathbf{Tr}\left(A_i(\triangle X \triangle Z Z^{-1} - \sigma\mu Z^{-1})\right), \; i = 1, ..., m.$$

The product $\triangle Z Z^{-1}$ has already been computed in (2.88). Matrix $\triangle Z^c$ is computed by (2.28) as

$$\triangle Z^c \quad = \quad -\sum_{j=1}^{m} \triangle y_j^c A_j,$$

and $\triangle X^c$, defined in (2.29), is obtained from

$$\triangle X^c = \sigma\mu Z^{-1} - \frac{1}{2}(\triangle X \triangle Z Z^{-1} + Z^{-1}\triangle Z \triangle X) - \frac{1}{2}(X \triangle Z^c Z^{-1} + Z^{-1}\triangle Z^c X). \tag{2.89}$$

Both $\triangle Z^c$ and $\triangle X^c$ can be computed in $4n^3 + \mathcal{O}(mn^2)$ flops. Hence, the corrector direction can be obtained in $6n^3 + \mathcal{O}(mn^2)$ flops.

2.4.3 NT

We derive in this subsection computational formulae for the NT search direction and its complexity for the Mehrotra predictor-corrector algorithm. Recall the direction specific expressions (2.71-2.75) from Section 2.3.3. As for the previous two search directions, $(\triangle X, \triangle y, \triangle Z)$ is a solution of system (2.11).

Using Lemma 2.2.1, vector $\triangle y$ is obtained by solving (2.17) in $m^3/3 + \mathcal{O}(m^2)$ flops. Due to (2.19) matrix $\triangle Z$ is computed from

$$\triangle Z \quad = \quad R - \sum_{i=1}^{m} \triangle y_i A_i,$$

where R is obtained from (2.8). It remains to show how $\triangle X$ can be computed. From (2.18) and (2.71-2.73) we get the relation

$$\triangle X = \mu Z^{-1} - X - W \triangle Z W. \tag{2.90}$$

Computational complexity of $\triangle X$ and $\triangle Z$ defined as above is $\mathcal{O}(mn^2 + n^3)$ flops.

The Mehrotra predictor-corrector algorithm for the central path defined for the NT direction will be described next. The affine-scaling direction $(\triangle X^a, \triangle y^a, \triangle Z^a)$ is equivalent to the direction $(\triangle X, \triangle y, \triangle Z)$ we just described, but with $\mu = 0$ in (2.73). Therefore, the complexity of computing $(\triangle X^a, \triangle y^a, \triangle Z^a)$ is the same as for $(\triangle X, \triangle y, \triangle Z)$ described above. Hence, we next analyze the corrector step for the NT direction.

Recall the expression for V^c in (2.26), namely

$$V^c := \mathcal{E}^{-1}(H_P(\triangle Z \triangle X) - \sigma\mu I).$$

Applying Lemma 2.2.1 to system (2.24) we see that $\triangle y^c$ is a solution of the system $M \triangle y^c = h^c$, where M is defined by (2.15) and h^c can be obtained from (2.27). Let

$$\widetilde{\triangle X} := P \triangle X P^T, \ \widetilde{\triangle Z} := P^{-T} \triangle Z P^{-1}.$$

If we multiply (2.90) with P, given by (2.61), on the left and P^T on the right, and taking into account that $W = P^{-1}P^{-T}$ and $P^{-T}ZP^{-1} = D$, see (2.69), we get

$$\widetilde{\triangle X} = \mu D^{-1} - D - \widetilde{\triangle Z}. \tag{2.91}$$

Using (2.65-2.69) and expression (2.91) for $\widetilde{\triangle X}$, we can write

$$V^c = P^{-1}\widetilde{V}^c P^{-T},$$

where $\widetilde{V}^c$ is the unique solution of the Lyapunov equation

$$\widetilde{V}^c D + D\widetilde{V}^c = (\mu D^{-1} - D - \widetilde{\triangle Z})\widetilde{\triangle Z} + \widetilde{\triangle Z}(\mu D^{-1} - D - \widetilde{\triangle Z}) - 2\sigma\mu I. \tag{2.92}$$

Equation (2.92) can be also written as

$$\widetilde{V}^c = \ll (\mu D^{-1} - D - \widetilde{\triangle Z})\widetilde{\triangle Z} + \widetilde{\triangle Z}(\mu D^{-1} - D - \widetilde{\triangle Z}) - 2\sigma\mu I \gg_D .$$

First, matrix $\widetilde{\triangle Z} := P^{-T}\triangle Z P^{-1}$ is computed in $2n^3 + \mathcal{O}(n^2)$ flops. Then the right hand side of the above Lyapunov equation (2.92) takes $n^3 + \mathcal{O}(n^2)$ flops. Next, the solution $\widetilde{V}^c$ is obtained in $\mathcal{O}(n^2)$ flops and the matrix V^c in $2n^3 + \mathcal{O}(n^2)$ flops. The matrix $\triangle Z^c$ is computed from (2.28) as

$$\triangle Z^c \quad = \quad -\textstyle\sum_{j=1}^m \triangle y_j^c A_j.$$

Finally, the expression for $\triangle X^c$, defined by (2.29), is obtained using results (2.71-2.73). Hence, it follows that

$$\triangle X^c = -W\triangle Z^c W - V^c.$$

Both $\triangle Z^c$ and $\triangle X^c$ are computed in $2n^3 + \mathcal{O}(mn^2)$ flops. Therefore, the corrector direction can be obtained in $7n^3 + \mathcal{O}(mn^2)$ flops.

2.4.4 Summary

In Table 2.2 we summarize the total computational complexities of the AHO, the HKM and the NT directions discussed in Sections 2.3 and 2.4.

The second column denotes which one of the two approaches is used to compute the matrix M. The third column shows the corresponding computational complexities for each direction. The overall complexity of the second approach is better, except for the HKM direction. In case $m < \frac{16}{3}n$, then even for the HKM direction the second computational approach will have better flop count.

Despite the good computational time, it has to be pointed out again that the second approach requires more memory space. This is a significant drawback when we have a large-scale SDO problems. The ratio between the amount of storage of the second approach compared with the first approach is at least two, see Monteiro et al. [83]. Therefore, the first approach is used in IPM's software for SDO, such as CSDP [18] and SDPA [113].

Search direction	Computational approach for M	Flop count for the predictor–corrector algorithm
AHO	First	$6mn^3 + m^2n^2 + \frac{2}{3}m^3 + \mathcal{O}(mn^2 + m^2n + n^3)$
$(P = I)$	Second	$\frac{11}{3}mn^3 + m^2n^2 + \frac{2}{3}m^3 + \mathcal{O}(mn^2 + m^2n + n^3)$
HKM	First	$4mn^3 + \frac{1}{2}m^2n^2 + \frac{1}{3}m^3 + \mathcal{O}(m^2n + mn^2 + n^3)$
$(P = Z^{\frac{1}{2}})$	Second	$\frac{4}{3}mn^3 + m^2n^2 + \frac{1}{3}m^3 + \mathcal{O}(m^2n + mn^2 + n^3)$
NT	First	$2mn^3 + m^2n^2 + \frac{1}{3}m^3 + \mathcal{O}(m^2n + mn^2 + n^3)$
$(P = W^{-\frac{1}{2}})$	Second	$\frac{2}{3}mn^3 + \frac{1}{2}m^2n^2 + \frac{1}{3}m^3 + \mathcal{O}(m^2n + mn^2 + n^3)$

Table 2.2: *Flop count results for the AHO, the HKM and the NT search directions using two different computational approaches.*

2.5 Other 'overhead' computations

The overall computational complexity in a primal-dual IPM is dominated by several operations depending on the particular structure of the SDO problem. If the constraint matrices $A_i, i = 1, ..., m$ are dense and m is far larger than n, more computational effort is spent in computation of the $m \times m$ matrix M from (2.15), e.g. see [20]. When A_i's are very sparse, factorizing[2] M becomes the dominant operation. Hence, an improvement of the time necessary for forming and factorizing this matrix could give a significant speedup of the total solution time.

In case of m is not far larger than n and the matrices A_i have a small number of big diagonal blocks, then the matrix operations on the positive semidefinite matrices X and Z can be the dominant ones, see e.g. [113]. In general the primal variable X is fully dense even if all the constraint matrices $A_i, i = 1, ..., m$ are sparse. As a result performing factorization on X can occupy a significant amount of time when m is not far larger than n. On the other hand, the dual matrix variable Z computed by

$$Z \quad = \quad C - \sum_{i}^{m} y_i A_i,$$

inherits the sparsity of the constraint matrices A_i $i = 1, ..., m$ and the data matrix C. This has been a critical disadvantage of primal-dual IPM methods compared to the dual interior-point method, which generates iterates only in the dual space. The dual-scaling IPM has lower computational cost per iteration compared with

[2]Depending on the search direction, M can be factorized by Cholesky or LU factorization.

primal-dual IPMs. However, it does not possess a super-linear convergence rate and has less accuracy in practice [114].

2.6 Conclusion

In this chapter we presented the computational complexity of three popular search directions for SDO, namely the AHO, the HKM, and the NT direction. Two approaches were presented for computing the matrix M. The second approach, introduced by Monteiro and Zanjácomo [83] takes less time, but requires more memory. Hence, it is less suited for large-scale problems.

All computational complexity results shown in this chapter are based on dense matrix computations. It has been shown in practice, see e.g. [112, chap. 2], that variations in problem structure seem to be more significant than problem size in determining the computational complexity of IPMs. Hence, exploiting structure is an important issue when one deals with software for solving SDO problems.

One popular approach to reduce the computational effort is to use block matrix structure of the data matrices, whenever possible, and store only the upper triangular part of all symmetric matrices, see e.g. [18, 113]. Another very successful approach used in practical computations is to use sparse data matrices and sparse linear algebra. This could lead to a significant reduction of the memory usage and of the number of computations necessary to solve certain types of large-scale SDO problems, with very sparse data matrices. Complexity results based on sparse computations are difficult to obtain, since the performance of such algorithms heavily depends on the problem's structure. Third, a significant reduction in practical computational difficulty can be achieved if a low-rank structure, where present, is exploited, see e.g. [107].

Practical experience suggested that a significant computational time is spent on computing and factorizing the matrix M. Speeding up these two computations has a big impact on the total solution time for large-scale problems. This motivated our objective in this thesis, namely to parallelize these computations.

Chapter 3

Parallel Architectures

In this chapter we present an overview of the parallel computer architectures and focus on the differences from the programming point of view. Suitability of the different parallel systems for the implementation of interior-point methods is also discussed. Finally, we motivate the choice of a parallel architecture for the implementation of a primal-dual IPM algorithm for SDO.

3.1 Parallel processing

The term parallel processing refers to the case when at least two processors cooperate by means of exchanging data while working on different parts of one and the same problem. Computers, which employ parallel processing, are referred to as parallel computers. There exist different classifications of parallel computer systems depending on: the number of processor units, CPU-to-memory access, interconnection networks, etc. More details on parallel computer architectures can be found in the survey by Duncan [38].

According to the criteria 'number of processing units', parallel systems with several thousands of processors are known as massively parallel, e.g. *BlueGene/L*[1] system by IBM[2] and *Jaguar - Cray XT4/XT3* by Cray Inc[3]. A computing system is said to be large-scale parallel system if it has several hundreds to thousands of processors. Most of the systems in these two categories are custom built, and are one-of-a-kind machines. Parallel computers with tens or hundreds of processing units, based on PCs, are considered small scale systems, e.g. the DAS3 computer[4].

[1] http://www.top500.org
[2] http://www.ibm.com
[3] http://www.cray.com/
[4] http://www.cs.vu.nl/das3/

In the past each processor used to have its own packaging, i.e., one CPU per microchip. Recently, multi-core processors were introduced, see Geer [49]. They contain multiple processing units in a single package. Examples of such CPUs are Intel[5] and AMD[6] dual-core and quad-core processors. As a result, the new generation of desktop and mobile PCs have a certain amount of parallel processing capability built-in. Therefore, parallel computing once again attracts significant interest from the scientific community. An up-to-date overview of the supercomputers can be found in [109].

3.1.1 Shared and distributed memory computers

An important classification of parallel computers is according to the CPU-to-memory access. Systems, where all of the processors have a direct access to the total amount of memory available, are called symmetric multiprocessors (SMP). Multi-core processors are considered to be a part of this class. When such computer systems are used for parallel processing, they are also called shared memory parallel computers, see Culler at al. [25]. From a hardware point of view such a system is a computer architecture where all processors have direct access to the common physical memory, see Figure 3.1.

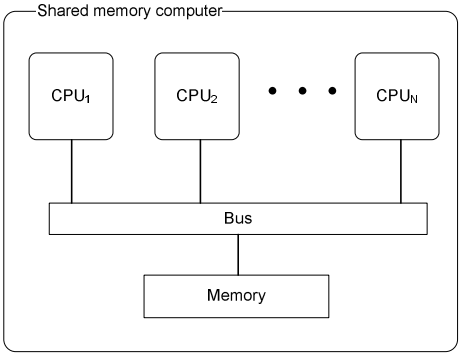

Figure 3.1: *Principle structure of a bus-based shared-memory parallel computer.*

In a programming sense, it describes a model where parallel tasks, at any moment, have the same "picture" of the memory, and can directly address and access the same logical memory locations, regardless of where the physical memory is actually located. Since the processors share the same address space, the knowledge of where data items are stored is no concern of the programmer. For a discussion of the challenges of programming shared-memory systems, see Pfister [91].
Shared memory parallel architectures can be further divided into two subclasses

[5] www.intel.com
[6] www.amd.com

based on memory access times: Uniform Memory Access (UMA) and Non-Uniform Memory Access (NUMA). The first one enjoys equal access times to all parts of the memory, whilst in the second one this is not the case.
Distributed memory computers are the second class of parallel computer systems, according to the criteria CPU-to-memory interconnection. The work of Seitz [96] pioneered this type of architecture. Each processor in such a system has its own local memory to which only this processor has a direct access, see Figure 3.2.

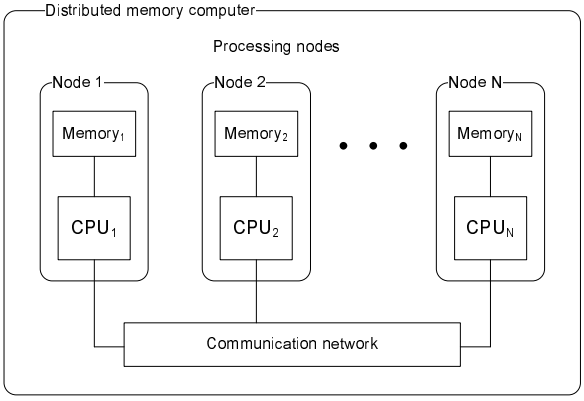

Figure 3.2: *Principle structure of a distributed memory parallel computer.*

From a programming point of view, no global memory address space exists across the processors. This means the user has to work with multiple address spaces, one for each local memory, and to explicitly organize the transfer of data from one address space to another when necessary. Such a computer is evidently more difficult to program than a shared memory parallel system or a single-CPU computer.
Most of the large scale and many small scale parallel systems are hybrid solutions in which several shared memory parallel computers are connected by means of an interconnection network. Such hybrid systems are non-uniform memory access (NUMA) architectures, see Figure 3.3.
A widely used classification of high-performance computer systems is the classification proposed by M.J Flynn [40]. According to Flynn's taxonomy, parallel computers have either SIMD (single instruction/multiple data) or MIMD (multiple instruction/multiple data) architecture, in practice.
In a SIMD parallel computer all processing units execute the same instruction on different data items, see Hillis and Tucker [61]. There is one control unit which is common for all processing units. This mode of operation fits quite well the computational requirements of many scientific applications. A common property of these applications is that they make use of the same operations on each element in large arrays of data. Such data-parallel operations are perfectly suited

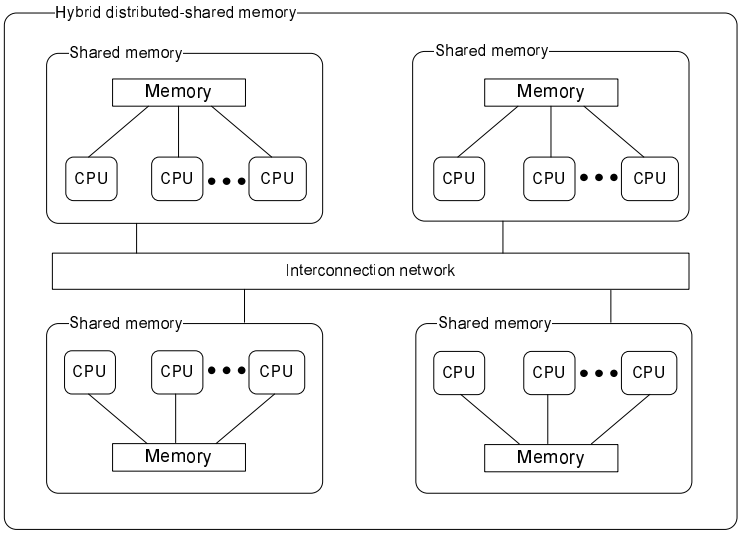

Figure 3.3: *Example of non-uniform memory access memory architecture.*

for execution in a SIMD mode. Despite the fact that parallel computers today are not SIMD, single instruction/multiple data is implemented in the SSE (Streaming SIMD Extensions) instruction set for the x86 processor architecture.
In contrast to the SIMD computer architecture, MIMD parallel computers could execute different instructions at the same time. Hence, the MIMD computers offer more flexibility than the SIMD computers. In programming sense they are generally more difficult to program and require more complex system software. The reason is the need for good coordination of the different processors. It is achieved either by the programmer or by the system software in the data parallel programming model.
In fact, most of the parallel computers make use of a combination of SIMD and MIMD modes of operation. While different processing nodes execute different instructions in a MIMD mode of operation, the SIMD processing mode is used within each processing node where all CPU units execute the same operation on different pieces of data. For more discussion on parallel computers from the programming point of view, see Dongarra et al. [36, Part I].

3.1.2 Distributed computing

Communication networks are an essential part of the computer technologies today. Computers interconnected by a network and running appropriate software can be operated as a distributed memory parallel computer, often called a cluster. Parallel processing on such systems is usually referred to as distributed comput-

ing, see Figure 3.4. Systems of this kind rely on a local area network (LAN) or a

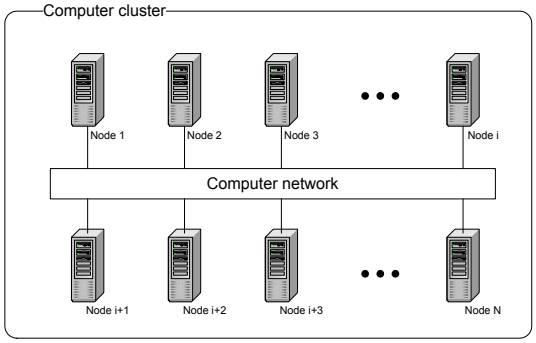

Figure 3.4: *A cluster of computers connected by a computer network used for distributed computing.*

high-speed, low-latency[7] network interconnect such as Myri-10G[8], Infiniband [1], etc. When a LAN is employed to connect a commodity hardware and software, one speaks of a Beowulf cluster [97, 98].
The system software needed for distributed computing is typically implementation of a message passing library, such as PVM[9] (Parallel Virtual Machine) or MPI[10] (Message Passing Interface). Parallel software, written in a standard programming language (C, FORTRAN, etc.), that makes subroutine calls to PVM or MPI libraries, is portable both to distributed computing systems and to shared memory parallel computers. The only difference is the time spent on communicating data between processors, see Gropp et al. [56].
There are two major applications of clusters: High Availability (HA) and High Performance Computing (HPC). The first one is used to ensure greater reliability of time-critical applications, while the second one is designed to provide greater computational power than a single computer. Since HA applications are not the subject of this thesis, we focus on HPC aspects of clusters, and particularly to their applications for interior-point method solvers for SDO.

3.1.3 Grid computing

The term *computer grid* was introduced in the mid 1990s as a concept describing a distributed computer infrastructure used in a coordinated way, see Foster et

[7] In a network, latency, a synonym for delay, is an expression of how much time it takes for a packet of data to make a round-trip from one designated point to another.
[8] Myricom, Inc. http://www.myri.com/.
[9] http://www.csm.ornl.gov/pvm/
[10] http://www.netlib.org/mpi/

al.[42]. The grid concept offers a model for solving massive computational problems by making use of a large numbers of unused computing resources and storage facilities, owned by different institutions, companies or individuals at different locations. The goal of this kind of computing is to solve large-scale computational problems such as earthquake simulation, weather modeling, financial modeling, and combinatorial optimization problems that are too big for any single supercomputer. A survey on using grids for large-scale science and engineering application was done by Johnston [63]. Results on using computational grids to solve large quadratic assignment problems (QAP) were presented by Anstreicher et al. [5].

Distributed facilities, taking part into the grid, retain the flexibility to work on multiple local smaller problems. There are no restrictions on the type of platforms, hardware/software architectures, and computer languages involved in the grid computing. A classification of grid resource management systems was introduced by Krauter et al. [73]. Example of a grid system is displayed on Figure 3.5.

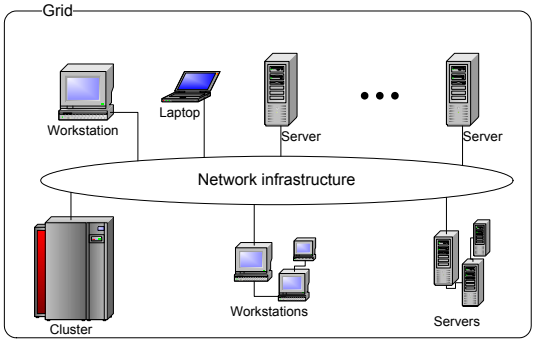

Figure 3.5: *Example of a grid computing infrastructure.*

The main challenge of the grid computing is to virtualize the view of the distributed computer resources in a way that a grid user essentially sees a single, large virtual computer. This is done by means of a grid middleware that takes care of integration of the participating facilities. At its core, the middleware is based on an open set of standards and protocols, e.g., the Open Grid Services Architecture (OGSA), that enables communication across heterogeneous, geographically dispersed environments. An example of such middleware is the Globus Toolkit [41]. It provides standard system components that support a wide variety of applications, without requiring a completely unique infrastructure to be developed for each application alone.

At first glance, grid computing and cluster computing look alike. The key distinction between them is mainly in the way resources are managed. In the case of clusters, a centralized resource manager takes care of allocating resources and

making the nodes working together as a single unified system. In the case of grids, each participating machine has its own resource manager that provides only a virtual single system view for the grid user, see e.g. [12, 43].

3.2 Suitability for IPM implementations

In the previous section we introduced a classification of the parallel computer architectures according to the number of processors, CPU-to-memory access and interconnection networks. However, there are only two main classes of multiprocessor 'supercomputers' today with respect to the programming model, namely shared memory and distributed memory architectures. The two architectures are suited to solving different kinds of problems.

Shared memory machines are best suited to so-called 'fine-grained' parallel computing, where all of the pieces of the problem are dependent on the results of the other processes. Distributed memory machines on the other hand are best suited to 'coarse-grained' problems, where each node can compute its piece of the problem with less frequent communication.

Another issue with distributed memory clusters is message passing. Since each node can only access its own memory space, there has to be a way for nodes to communicate with each other. Beowulf clusters use MPI to define how nodes communicate. An issue with MPI, however, is that there are two copies of data: one is on the node, and the other has been sent to a central server. The cluster must ensure that the data that each node is using is the latest.

Partitioning problems to solve them by distributed computations is the main difficulty with the Beowulf cluster. To run efficiently, problems have to be partitioned so that the pieces will run efficiently on the RAM, disk, networking, and other resources on each node. If nodes have a gigabyte of RAM but the problem's data set does not easily partition into pieces that run in a gigabyte, then the problem could run inefficiently. This issue with the dynamic load balancing is not a problem for shared memory computers.

The attraction of using Beowulf clusters lies in the low cost of both hardware and software and the control that builders/users have over their system. These clusters are cheaper, often by orders of magnitude, than single-node supercomputers. Advances in networking technologies and software in recent years have helped to level the field between massively parallel clusters and purpose-built supercomputers.

Ultimately the choice 'shared or distributed' depends on the problem one is trying to solve and on the available computing resources. In our case the parts of the SDO algorithm suitable for parallel computation are computing the Schur complement matrix and its Cholesky factorization.

Firstly, composing the matrix M as in (2.15) allows each node independently from the others to compute its piece of data and no communication is needed until the pieces of the matrix are assembled. This allows us to regard this computation

as a course-grained process. Secondly, the ScaLAPACK [14] routines employed for factorization of M are based on block-partitioned algorithms in order to minimize the frequency of data movement between different levels of the memory hierarchy.

3.3 Conclusion

In this chapter we have presented an overview of different parallel computer architectures. We discussed their differences from the programming point of view. Shared memory computers are easier to program but much more expensive than the distributed memory computer clusters. PCs with multi-core CPUs in most cases have RAM comparable to that of a desktop PC, so they should not be viewed as cheap shared-memory supercomputers yet. Distributed memory clusters are more difficult to program, but offer a significant amount of memory and computing power at lower price. They are easier to scale up too by simply adding a few additional nodes. These advantages make clusters the architecture of choice for most users today. Taking into account the strong and the weak points of both architectures, and the computations in IPMs that are most suitable for parallelization, a distributed memory cluster architecture was chosen for further software development of prima-dual interior-point methods for semidefinite optimization.

Chapter 4

Parallelization of IPM's

This chapter focuses on the different strategies to use parallel computations to speed-up interior-point methods and in particular primal-dual methods for SDO. Parallelization of IPM's is not a new concept and the first implementations of algorithms for linear optimization (LO) appeared in the early 1990's [67, 76]. Several parallel software packages for LO exist today. For example, pPCx by Coleman et al. [24] and OOPS by Gondzio and Sarkissian [54]. Both of them use primal-dual IPM algorithm with the Mehrotra predictor-corrector strategy designed to run on computer clusters via MPI routines.

Parallel versions of several SDO software packages have been developed too. For example PDSDP [10], SDPARA [114], SDPARA-C [85] and a parallel version of CSDP [20]. The first three are designed for PC clusters using MPI and ScaLAPACK[1], and the last one is designed for a shared memory computer architecture [20]. PDSDP is a parallel version of the DSDP5 solver developed by Benson, Ye, and Zhang [11] and it uses a dual scaling algorithm. SDPARA is a parallel version of SDPA for a computer cluster and employs a primal-dual interior-point method using Mehrotra[78] predictor-corrector strategy and HKM search direction, as does CSDP. SDPARA-C uses a path-following primal-dual IPM algorithm and HKM search direction. It is a parallel version of SDPA-C solver developed by Nakata et al. [84].

In this chapter we revisit the topic of the parallel computations in primal-dual IPM's for LO and SDO. We discuss the possibilities and respective trade-offs in the process of parallelizing interior-point algorithms for semidefinite optimization based on the AHO, HKM and NT search directions, described in Chapter 2.

[1] `http://www.netlib.org/scalapack/`

4.1 Identification of parallelism

A widely adopted approach for decomposing programs for parallelism contains three major components, see Dongarra et al. [36]. Firstly, we have to identify components that can run in parallel. Secondly, a strategy for decomposing the algorithm has to be chosen. Finally, a program is created according to the programming model and the computer architecture of choice.

The first task is to identify the places in the algorithm where there is parallelism to exploit. This means operations that can be handled independently without data sharing, with each of the computations running until the end, when they synchronize on the final result. Certainly, data sharing is not a problem if two computations both read the same data from a shared memory location. Therefore, for a data sharing to cause a problem, one of the computations must write into memory that the other accesses by either reading or writing, see Bernstein [13].

Another important task in parallel processing is to choose a strategy for decomposing the program into pieces that can be run in parallel. There are two possible approaches to do this. The first one is known as task parallelism. Different processors carry out different independent functions in the same time. Task parallelism is typically limited to small degrees of parallelism.

The second approach, called data parallelism, divides the data of a problem into regions and assigns different processors to compute the results for each region. This type of parallelism is more commonly used in interior-point methods codes for SDO since it exhibits a natural form of scalability. The computation per processor remains the same, a larger problem should require only modestly longer running time than a smaller problem on a smaller machine configuration. Such data parallelism is the one used in PDSDP, SDPARA [114], SDPARA-C and a parallel version of CSDP.

Data parallel loops represent the most important source of parallelism in scientific programs in general. The typical way to parallelize loops is to assign different iterations, or different blocks of iterations, to different processors. Here each iteration of the loop accesses a different element of the matrix M, so that there is no data sharing. In many cases, it is possible to achieve significant parallelism in the presence of so-called data races. A data race essentially means two or more processes that can run in parallel without synchronization until the end. The notion and conditions for existing of a data race were formalized in the 1960's by Bernstein [13].

Distributed computations often require the use of explicit message passing, e.g., using MPI library. In this case the SPMD style [62] is a convenient choice. In an SPMD program, all of the processors execute the same code, but apply the code to different portions of the data. All scalar variables are replicated on all of the processors and redundantly computed on each processor. In addition, explicit communication primitives are necessary in order to pass the shared data between processors. This is the case for LP solver OOPS[54] and SDO solvers PDSDP,

SDPARA and SDPARA-C.

4.2 Parallel primal-dual IPM's for LO

Semidefinite optimization contains as a special case the LO problem class, see e.g. De Klerk [27]. Therefore, the topic of parallel computations in IPM's for LO is related to the topic of parallel SDO. Next, we proceed by describing the differences in the computational aspect of primal-dual interior-point methods for LO and SDO. More details and discussion about the different IPM algorithms for LO can be found in the book of Roos, Terlaky and Vial [94].

We consider the primal linear optimization problem in standard form

$$\text{(LP)} \quad \min \left\{ c^T x : Ax = b, x \geq 0 \right\},$$

where $A \in \mathbb{R}^{m \times n}$ is real $m \times n$ matrix of rank m, vectors $x, c \in \mathbb{R}^n$ and $b \in \mathbb{R}^m$. The dual problem of (LP) is given by

$$\text{(LD)} \quad \max \left\{ b^T y : A^T y + s = c, s \geq 0 \right\},$$

with $y \in \mathbb{R}^m$ and $s \in \mathbb{R}^n$.

Problems (LP) and (LD) correspond to the SDO problems (P) and (D) defined in Chapter 2, respectively when matrices C, X, Z and $A_i, i = 1, ..., m$ are diagonal. It is well-known [94], that finding an optimal solution of (LP) and (LD) is equivalent solving the non-linear system of equations

$$\begin{aligned} Ax &= b, \quad x \geq 0, \\ A^T y + s &= c, \quad s \geq 0, \\ Xs &= 0, \end{aligned} \tag{4.1}$$

where $X \in \mathbb{R}^n$ is a diagonal matrix with entries $x_i, i = 1, ..., n$. Similar to the SDO case, in LO the non-linear third equation in (4.1) (so-called complementarity condition) is replaced by the equation $Xs = \mu e$, with parameter $\mu > 0$ and with e denoting the all-one vector $(1; 1; ...; 1)$ of length n. As a result the system (4.1) can be written as

$$\begin{aligned} Ax &= b, \quad x \geq 0, \\ A^T y + s &= c, \quad s \geq 0, \\ Xs &= \mu e. \end{aligned} \tag{4.2}$$

Note that the third equation in (4.2) corresponds to the third one in (2.3) for SDO. System (4.2) has a unique solution $(x(\mu), y(\mu), s(\mu))$ for each $\mu > 0$ and $x(\mu)$ is called the μ-center for (LP) and $y(\mu), s(\mu)$ is the μ-center for (LD). Assume strict feasibility, the set of μ-centers (with $\mu > 0$) defines the central path of (LP) and (LD). If $\mu \to 0$ then the limit of the central path exists and yields optimal solutions of (LP) and (LD)[94].

Primal-dual IPM methods for LO follow the central path approximatively. Given a strictly feasible solution (x, y, s) the goal is to obtain the primal and dual directions $\triangle x$ and $(\triangle y, \triangle s)$, respectively, by solving the following Newton system

$$\begin{bmatrix} A & 0 & 0 \\ 0 & A^T & I \\ S & 0 & X \end{bmatrix} \begin{bmatrix} \triangle x \\ \triangle y \\ \triangle s \end{bmatrix} = \begin{bmatrix} b - Ax \\ c - A^T y - s \\ \mu e - Xs \end{bmatrix}, \tag{4.3}$$

where $S \in \mathbb{R}^{n \times n}$ is a diagonal matrix with entries $s_i, i = 1, ..., n$. From the third equation in (4.3) we can express $\triangle s$ as

$$\triangle s = X^{-1} (\mu e - Xs - S \triangle x). \tag{4.4}$$

Substituting $\triangle s$ as in (4.4) into (4.3) leads to the so-called augmented system of the LO problem

$$\begin{bmatrix} -D^{-1} & A^T \\ A & 0 \end{bmatrix} \begin{bmatrix} \triangle x \\ \triangle y \end{bmatrix} = \begin{bmatrix} g \\ u \end{bmatrix}, \tag{4.5}$$

where

$$\begin{aligned} D &:= XS^{-1}, \\ g &:= c - A^T y - \mu X^{-1}. \\ u &:= b - Ax. \end{aligned}$$

Note that D is a diagonal matrix. From (4.5) we an obtain the expression for the primal direction $\triangle x$ as

$$\triangle x = DA^T \triangle y - Dg,$$

and $\triangle y$ is a solution of the following linear system

$$ADA^T \triangle y = ADg + u. \tag{4.6}$$

System (4.6) is symmetric and positive definite and can be written similar to SDO case as $M \triangle y = h$, where

$$M := ADA^T, \tag{4.7}$$

and

$$h := ADg + u.$$

The major difference with the SDO case is that in LO the computation of M consists of two matrix multiplications. One of these multiplications can be done very fast in practice because D is a diagonal matrix. In linear optimization case M is most often sparse, so the time to compute it is much less than the time needed for factorizing it. As a result there is no gain in using parallel computations to compute M for LO. On the other hand, solving the linear system (4.6) by parallel solver leads to a significant speed-up, see [54].

4.3 Parallel computation of the matrix M in SDO

Computation of the search direction in a primal-dual algorithm for SDO as in Figure 2.1 (on page 22) consists of sequential operations only, except for the computation of the elements $M_{ij}, i, j = 1, ..., m$ in (2.15). The matrix M is typically dense and its computation takes a big portion of the total running time on a large-scale SDO problems, see e.g. [114] and [20]. Therefore, M is a good candidate for parallel computation using the data parallelism model.

All three search directions presented in Chapter 2 result in a positive-definite matrix M, but only for the HKM and the NT direction it is also symmetric, see e.g. [103]. This means that for the AHO direction it is necessary to compute in the worst case at most $\frac{m}{2}(m-1)$ extra elements of M compared with the other two directions. Practical experiments with search directions for SDO done by Todd [103] suggests that the AHO, the HKM and the NT directions have the best performance in terms of number of iterations. Unfortunately, additional computational work needed to compute the typically large size M makes the AHO direction less attractive for practical parallel implementation. Hence, in the analysis that follows in this chapter, we assume that the HKM or the NT search direction is used.

Recall from Chapter 2 that computation of the matrix M is done as a matrix product of the type

$$M_{ij} = \mathbf{Tr}(A_i S_1 A_j S_2), \quad i, j = 1, ..., m, \tag{4.8}$$

where $S_1 = Z^{-1}, S_2 = X$ and $S_1 = S_2 = W$ for the HKM and the NT directions, respectively. A construction similar to (4.8) arises also in the dual-scaling interior-point algorithm behind the PDSDP solver. To compute the search direction, PDSDP involves the solution of a positive definite linear system of the type $M \triangle y = h$, see for details [10]. The matrix M in this case is defined as

$$M_{ij} := \mathbf{Tr}(S_2 A_i S_1 A_j), \quad i, j = 1, ..., m, \tag{4.9}$$

where $S_1 = S_2 = Z^{-1}$ [10]. Note that since all matrices involved in the product $S_2 A_i S_1 A_j$ are square, due to (2.55), it follows that (4.9) is equivalent to (4.8).

In what follows, we show several approaches to compute the matrix M in parallel. Assume we have a multiprocessor computer with N processors, where $N > 1$. We denote by $P_j, j = 0, .., N-1$ a corresponding process for each processor. Defined in this way processes form a one-dimensional array, i.e., a vector of processes. Next we outline two different ways used in SDO software to map the matrix M to the vector of processes.

Data distribution schemes

Since M is symmetric, only the lower (or upper) triangle need to be computed. Assume from now on that we compute only the lower triangle.

The first approach for distributed computation of M, defined as in (4.8), assigns in a cyclic manner the rows (or columns) to the processes. Without loss of generality, we may assume that it is done row-wise. At the beginning of the distributed computation, each of the N processes is scheduled to compute only one of the first N rows of matrix M. Next, the $(N+1)$-th row is assigned to process P_0, $(N+2)$-th row to P_1 and so on. In this way each processor can work independently on one row at the time, e.g., row i is computed by process P_r, where $r := (i \ mod \ N)$, $i = 1, ..., m$ and *mod* is the modulus operator. Figure 4.1 shows this kind of data distribution. It is also called a one-dimensional cyclic row

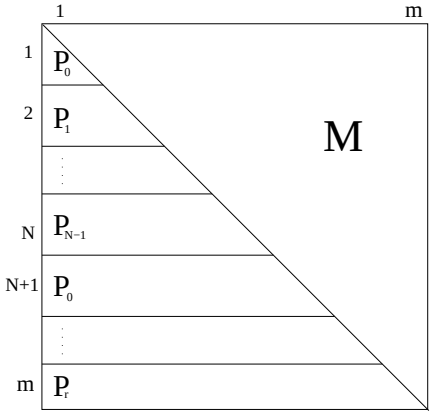

Figure 4.1: *One-dimensional cyclic row distribution.*

distribution [14, chap. 4] and it is used in SDPARA, SDPARA-C and PDSDP.

The shared memory version of CSDP uses similar approach, see Borchers and Young [20]. The difference is that the matrix M is subdivided into blocks of rows with size n_b, where $n_b \in [1, m)$. Each block of n_b consecutive rows is assigned for computation to one process only. When a static scheduling is used, the i-th row is assigned for computation to process P_r, where in this case $r := (\lfloor i/n_b \rfloor \ mod \ N)$. This type of distribution is also called a one-dimensional block-cyclic row distribution and is shown in Figure 4.2. Note that in the general case the last block will have $m - kn_b$ rows, where $k := \lfloor m/n_b \rfloor$. This type of data distribution contains the one-dimensional cyclic row distribution as a special case when $n_b = 1$. The shared memory version of the CSDP does not use a static scheduling of the blocks of rows, but a "self scheduling" approach. Each time a processor finishes a block, it begins to work on the next one that needs to be done. This provides better load balance than the static scheduling approach.

In both variants of the one-dimensional distribution, each row (or block of rows) of M is stored into the local memory of the processor that owns the corresponding process. Hence, none of the processors has the complete matrix M in

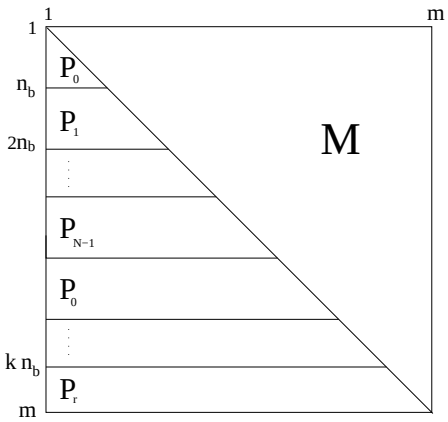

Figure 4.2: *One-dimensional block cyclic row distribution.*

its memory. The advantage of this one-dimensional approach is that the load balance among the processes is very good, see Kojima et al. [114]. The disadvantage is that the data distribution of M is not well suited for distributed factorization necessary in the next step of the primal-dual IPM algorithms.

Computational approaches

In both one-dimensional data distributions described above, all elements of one row of M are computed by one process only. Next we show different implementations of such row computations in the primal-dual IPM solvers for SDO.

The first approach is based on (4.8) and computes the elements in the i-th row of M for $i = 1, ..., m$, by the following steps:

(1) Compute the matrix product $A_i S_1$ once;

(2) Compute for $j = 1, ..., i$ the product $A_j S_2$;

(3) Compute $\mathbf{Tr}[(A_i S_1)(A_j S_2)], j = 1, ..., i$.

It is easy to see that processors work independently and no synchronization or data exchange is necessary between them until the end. Almost linear speedup can be expected using this type of parallel computations, see e.g. [114]. This approach is used in SDPARA solver.

The second approach for computing M uses expression (4.9). In this case the following steps are necessary to obtain each element of M in i-th row for $i = 1, ..., m$:

(1) Compute once the matrix product $S_2 A_i S_1$;

(2) Compute $\mathbf{Tr}[(S_2 A_i S_1) A_j], j = 1, ..., i$.

This way of computing M is used in PDSDP solver. For instances where computational time of the matrix products $A_j S_2$ for $j = 1, ..., i$ is dominant over the time of computing once $S_2 A_i S_1$, then the second approach would have an advantage. In general, both approaches have computational complexity $\mathcal{O}(mn^3 + m^2 n^2)$.

SDPARA-C uses a similar idea as mentioned above. The major difference is that matrices X and Z^{-1} are not explicitly stored, but only their Cholesky factors L_X and $L_{Z^{-1}}$ are accessible, i.e., X is implicitly available via $X := L_X L_X^T$ and $Z^{-1} := L_{Z^{-1}} L_{Z^{-1}}^T$. As a result, this approach is more computationally expensive than the other two we described above [85]. Hence, we omit its exact formulation. Detailed information about it can be found in Nakata et al. [85]

CSDP solver uses both the first and the second approach, depending on the density of the of the constraint matrices A_i, and it can dynamically switch between the two.

Computation of M is followed by its Cholesky factorization, if a direct solver is used for the linear system $M \triangle y = h$. It is well known [22] that parallel factorization routines achieve their best performance when the data matrix is partitioned into big and square blocks. This is not the case after both one-dimensional data distributions described earlier. Using them to compute M causes the data to be partitioned by rows, i.e., not in the desired shape for fast distributed factorization. Therefore, prior to the factorization step a data redistribution procedure is necessary. It involves inter-processor communication that causes additional communication overhead and loss of overall parallel performance. Discussion about how parallel performance is related to the data distributions can be found in [22].

One way to decrease the influence of the communication overhead is to use very high speed Myrinet or Infiniband network interconnects, if available. Solvers such as SDPARA, SDPARA-C and PDSDP rely on such interconnects for good parallel performance, because they perform a data redistribution procedure for M.

We propose another solution in case such interconnects are not available, which is the case on many PC clusters. We use a well-known map from the one-dimensional array of processes into a two-dimensional rectangular process array, called often a process grid. Then, the matrix M is directly computed with two-dimensional block cyclic data distribution, the most suitable for Cholesky factorization. As a result, we eliminate the need of data redistribution and achieve the best overall performance of computing M and factorization step together. We describe all details of our approach in Chapter 5.

4.4 Parallel solution of positive-definite linear systems

The first step of computing the search direction for SDO $(\triangle X, \triangle y, \triangle Z)$ involves solving the positive-definite linear system $M \triangle y = h$ to obtain $\triangle y$, see Chapter 2. This is the case not only for SDO but for primal-dual IPM algorithm for LO as

well. Algorithms for solving positive-definite linear systems can be divided into iterative methods and direct methods. Iterative methods such as conjugate gradient (CG), preconditioned conjugate gradient (PCG) and the generalized minimal residual method (GMRES) repeatedly refine an initial approximation to the solution of the linear system until they obtain an approximation which is close to the solution. A direct method as Cholesky factorization on the other hand factorizes the matrix M into a product of a lower triangular matrix L and its transpose, i.e., $M = LL^T$. Note if we compute the Cholesky factorization of M, we need to back solve two triangular linear systems $Lw = b$ and $L^T \triangle y = w$, where $w \in \mathbb{R}^m$, in order to obtain vector $\triangle y$.

If one wishes to exploit sparsity in solving the linear system $M \triangle y = h$, the parallelization becomes more complicated. Several issues arise. Firstly, an iterative solver such as PETSc [8] can always be used to obtain $\triangle y$. This requires a preconditioner, which not only depends on the problem but also is more difficult to parallelize. Work on this issue was done by Balay et al. [9] and Jones and Plassmann [64], who developed an efficient in practice parallel incomplete Cholesky factorization algorithm. Still, iterative methods for parallel IPM computations show weaker performance compared to the direct solvers [36].

Secondly, parallel direct-sparse solvers on a shared memory multiprocessor computers are available and can be used efficiently, for example PARDISO [95]. Such solvers are difficult to use on computer clusters, because data and computation are tricky to distribute to balance the load among processors, for example PSPASES [65]. A symbolic factorization phase is a potential serial bottleneck in addition to the sparse triangular system solves. Sun developed a group of parallel direct-sparse solvers for optimization using a multifrontal approach [102].

The third and widely used option in parallel IPM methods for SDO is the use of parallel dense direct solver with Cholesky factorization.

4.4.1 Linear optimization case

Large-scale LO problems often have a very sparse constraint matrix A that also has some particular block structure. Examples of such block structures are block-diagonal and block-angular. These block structures can be handled by a decomposition approach that allows a considerable computational efficiency trough parallel computations, see e.g. [76]. The matrix M, computed by (4.7), is typically sparse if the constraint matrices are sparse. When it is factored by Cholesky factorization, it suffers from fill-in. Since PMP^T is also symmetric and positive definite for any permutation matrix P, one can solve the reordered system

$$(PMP^T)(P \triangle y) = Ph.$$

The choice of P can have a significant effect on the amount of fill-in that occurs during the factorization. Hence, it is beneficial to reorder the rows and the columns of the matrix before performing the factorization. The problem of finding

the best ordering of a matrix in the sense of minimizing the fill-in is NP-complete, see e.g. [115]. Therefore, the reordering algorithms rely on heuristics. One of the most effective is the minimum degree ordering, see e.g. [50]. The use of such algorithms makes parallel direct-sparse Cholesky solvers for positive-definite linear systems very efficient in implementations of IPM methods for LO, see e.g. [24, 67].

A different approach was considered recently by Gondzio and Sarkissian [54] with respect to solving (4.6). They used an implicit inverse representation of ADA^T in order to solve the positive definite system (4.6). This implicit approach leads to a sparsity of the matrix product ADA^T similar to the one of A. Hence, it leads to better efficiency of computation of M and allows a straightforward parallelization of many computational steps when solving (4.6). The disadvantage of the implicit inverse schemes of ADA^T is that they are very sensitive to accuracy issues.

4.4.2 Semidefinite optimization case

Computation of the elements of M as in (4.8) for SDO often results in a fully dense matrix M. This is the case even when the constraint matrices $A_i, i = 1, ..., m$ and C are sparse, unlike large-scale LO problems. Hence, in SDO a procedure such as minimum degree ordering and parallel direct-sparse Cholesky solver would not be beneficial. Therefore, the parallel direct-dense solver with Cholesky factorization is a widely used option in parallel primal-dual IPM algorithms for SDO [10, 114].

The most important variations of algorithms for dense Cholesky factorization are right-looking (or *Gaxpy*) and left-looking (or *Outer product Cholesky*), see e.g. Golub and Van Loan [53, p.142-144]. Next, we give a brief description of both variants of Cholesky factorization.

Let L be the Cholesky factor of the matrix M. Recall that M is symmetric and we have assumed earlier that it is stored in lower triangular form. During the Cholesky factorization the elements of L overwrite the elements of M in the lower triangle.

Left-looking Cholesky factorization [Gaxpy Cholesky in Golub, Van Loan]

In the left-looking version of the algorithm for Cholesky factorization, the matrix M is traversed by columns from left to right. Each step updates the current column from previous columns are performed first, and then computations are performed on the current column. Figure 4.3 shows a snapshot of left-looking algorithm for Cholesky factorization at column k, where $k = 1, ..., m$.

The following operations are performed at column k of left-looking version of Cholesky factorization:

(1) Update column k from columns to the left $M_{ik} = M_{ik} - M_{ij}M_{kj}$ for $j = 1, ..., k-1$ and $i = j, ..., m$;

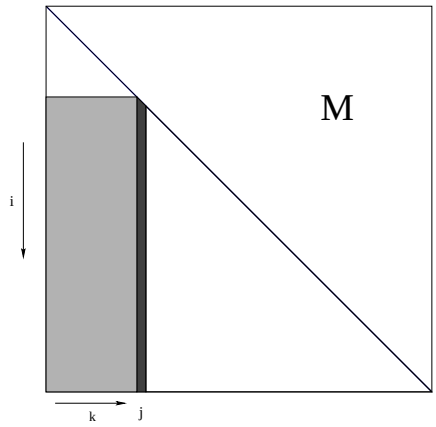

Figure 4.3: *Left-looking Cholesky factorization.*

(2) Reset $M_{kk} := \sqrt{M_{kk}}$;

(3) Compute elements of column k, i.e. $M_{ik} := M_{ik}/M_{kk}$ for $i = k+1, ..., m$.

Right-looking Cholesky factorization [Outer product Cholesky in Golub, Van Loan]

In the right-looking variant of the Cholesky factorization, similarly to the left-looking approach, the matrix is traversed by columns from left to right. This time at each step, computations are performed on the current column first, and then updates to columns to the right indexed by j are performed immediately, see Figure 4.4.

Assume the algorithm is computing currently column k, where $k = 1, ..., m$. The right-looking variant of the Cholesky factorization executes the following steps:

(1) Reset $M_{kk} := \sqrt{M_{kk}}$;

(2) Compute elements of column k as $M_{ik} := M_{ik}/M_{kk}$;

(3) Update all columns to the right $M_{ij} := M_{ij} - M_{ik}M_{jk}$ for $j = k+1, ..., m$ and $i = j, ..., m$.

In both left- and right-looking variant the matrix is traversed by columns from left to right. The difference between them is the way the updates are done. In the left-looking case they are done as late as possible. The right-looking version is doing them as soon as possible. Neither one of the variants of the Cholesky factorization is viewed as canonical or performs uniquely better on a single processor computer, see e.g. [79].

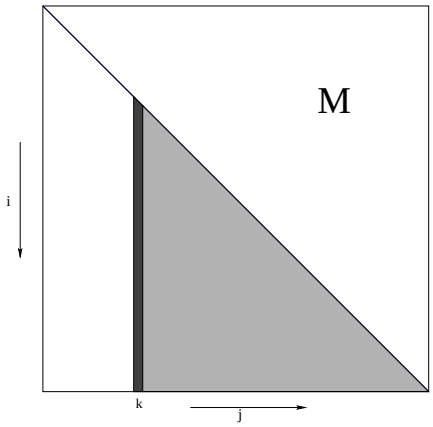

Figure 4.4: *Right-looking Cholesky factorization.*

Block-partitioned form Cholesky factorization

Better performance and scalability of the Cholesky factorization is achieved in practice if a column-wise block-partitioned approach is used, see e.g. [37]. A right-looking factorization in block-partitioned form is implemented in LAPACK routine DPOTRF, see Figure 4.5.

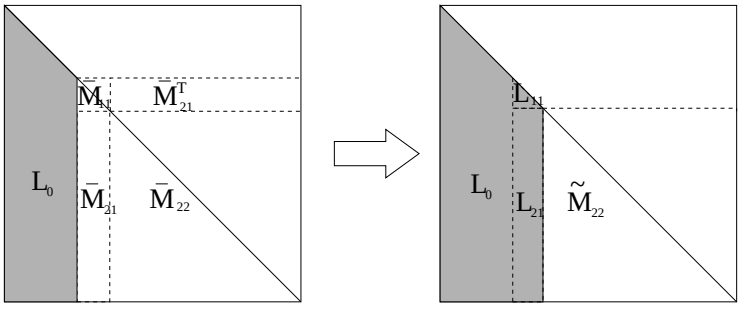

Figure 4.5: *Right-looking block-partitioned form of Cholesky factorization.*

Assume that at the k-th step of the factorization, matrix M has a partition as in Fig.4.5. Here L_0 is the already computed part of the Cholesky factor L, and $\bar{M}$ is the remaining $m_k \times m_k$ symmetric diagonal block matrix of M to be

factorized next. Let $\bar{M}$ be partitioned as in Figure 4.5:

$$\bar{M} = \begin{bmatrix} \bar{M}_{11} & \bar{M}_{21}^T \\ \bar{M}_{21} & \bar{M}_{22} \end{bmatrix} = \begin{bmatrix} L_{11} & \mathbf{0} \\ L_{21} & L_{22} \end{bmatrix} \begin{bmatrix} L_{11}^T & L_{21}^T \\ \mathbf{0} & L_{22}^T \end{bmatrix} = \begin{bmatrix} L_{11}L_{11}^T & L_{11}L_{21}^T \\ L_{21}L_{11}^T & L_{21}L_{21}^T + L_{22}L_{22}^T \end{bmatrix},$$

where matrices $\bar{M}_{11}$ and L_{11} are $m_b \times m_b$, $\bar{M}_{21}$ and L_{21} are $(m_k - m_b) \times m_b$, $\bar{M}_{22}$ is $(m_k - m_b) \times (m_k - m_b)$ and $m_b > 1$ is the size of the column panel (block). Note that the matrix L_{11} is lower triangular.

The order of operations performed at the k-th step of right-looking block-partitioned version of the Cholesky factorization is:

(1) Compute the Cholesky factorization of the diagonal block $\bar{M}_{11}$ using right-looking or left-looking strategy described above. As a result L_{11} is obtained;

(2) Compute the column panel $L_{21} = \bar{M}_{21} \left(L_{11}^T\right)^{-1}$;

(3) Update all the rest of the matrix $\bar{M}$ to the right, so $\tilde{M}_{22} = \bar{M}_{22} - L_{21}L_{21}^T$.

Several packages provide routines for block-partitioned variants of parallel Cholesky routines for distributed computations on clusters, such as ScaLAPACK and PSPASES [65]. Since, one of the aims of our implementation is to be portable to many different computer platforms we choose the ScaLAPACK library. It provides a parallel right-looking block-partitioned Cholesky routine PDPOTRF[2]. In order to improve the parallel numerical factorization performance this routine makes use of fast matrix multiplications based on a Level-3 BLAS (basic linear-algebra subprogram) routines. The solvers SDPARA, SDPARA-C and PDSDP all make use of this routine from the ScaLAPACK library.

4.5 Overhead

Apart from computing the matrix M and performing its Cholesky factorization in parallel, there is another operation that can benefit from the parallel processing. As we already discussed in Section 2.5 of Chapter 2, the primal variable X is dense even if all the constraint matrices $A_i, i = 1, ..., m$ are sparse. As a result, the Cholesky factorization of X can occupy a significant amount of time when m is not far larger than n, i.e., we have a very large size of the matrix variables X and Z compared with the number of constraints.

Factorization of a large-scale matrix X on a shared-memory parallel computer is a straightforward operation since each processor has direct access to every

[2]http://www.netlib.org/scalapack/

element of X. Unfortunately, on a computer cluster there are two major difficulties in factorizing X using distributed computation. The first problem arises from the fact that all nodes in the cluster contain and use X locally for all redundant sequential computations in a primal-dual algorithm for SDO. In case X has to be factorized in parallel, a distributed copy of it has to be created. As a result, each node will require extra memory to store its local part of the distributed X. The second disadvantage is that after X factorized, the resulting factor L_X from $X := L_X L_X^T$ remains distributed, i.e., none of the nodes has all elements of it. Since all of the nodes require a local updated copy of L_X, a significant amount of inter-processor communication is necessary to assemble a complete L_X on every node of the cluster. The benefit from the distributed factorization of X is lost in most cases due to the extra communication overhead. Hence, factorization of X is usually left as a local computation on every node as in the case of the SDPARA solver.

For certain classes of SDO problems such as combinatorial optimization problems, where the constraint matrices $A_i, i = 1, ..., m$ and the matrix C are very sparse, exploiting the structure of the problems to obtain sparse X and its factorization have been successful. Work in this direction was done by Fukuda et al. [47] and Nakata et al. [84]. They introduced the positive-definite matrix completion method for primal-dual IPM's. This method uses matrix completion theory to exploit the sparsity pattern of C and $A_i, i = 1, ..., m$, and perform a sparse factorization on the primal matrix variable X. A key element of it is that X is not stored explicitly, so there is less demand for memory in practice. This matrix completion technique is employed by the SDPARA-C parallel solver [85]. A down side of positive-definite matrix completion is the increased computational cost for obtaining the elements of M. Therefore, the benefits of it can be in many cases marginal and even nonexistent, depending on the structure of the problem, see e.g. [85].

Since the latest generation of processors have more than one computing core, in other words in-build shared memory parallel processing features, those can be used for speeding up the Cholesky factorization of X. Standard packages for linear algebra as Intel MKL[3], AMD ACML[4] and ATLAS[5] BLAS already offer enhanced Cholesky factorization routines optimized for multi-core processors, see e.g. Kurzak and Dongarra [74]. As a result, cluster based IPM solvers for SDO could evolve into a 'hybrid' type parallel software. On a global level they can take advantage of the distributed cluster computations, but locally, on a node level, they will be able to perform small-scale shared memory computations.

It is a well-known fact that high performance does not automatically follow from parallel implementation. To achieve the highest possible performance one must take a number of other considerations into account. Firstly, it is very important to balance the loads on the components of the computing configuration

[3] http://www.intel.com/software/products/mkl
[4] http://www.amd.com/acml
[5] http://math-atlas.sourceforge.net/

so that no single component dominates the running time. Secondly, solving very large problems requires that the computation scales well to large numbers of parallel processors, i.e., the implementation requires a certain level of craftsmanship.

4.6 Conclusion

Despite the effort and some solid developments, the use of parallelism in interior-point methods has not been as wide spread as their sequential version. One of the reasons is that to run a parallel version of IPM solver for SDO, specialized and thus expensive hardware was necessary. With the recent progress of processor technologies towards multi-core CPUs and high speed network connectivity, parallel computations are within the reach of many new users.

Another reason for the slow progress is the inherently sequential nature of interior-point methods for SDO. Apart from the computing in parallel the matrix M and solving the linear system in $\triangle y$, no other parts allow profitable concurrent computations. Despite this, there is still room for improvement such as exploiting the low-rank structure. Finally, computing the matrix M using a row-by-row technique might be efficient for certain type of problems, but it introduces undesirable communication overhead.

Chapter 5

Implementation of a Primal-Dual IPM for SDO using the HKM Search Direction

In this chapter we present the algorithmic framework and practical aspects of implementing a distributed parallel version of a primal-dual semidefinite programming solver on a computer cluster. Our implementation is based on the sequential solver CSDP5.0 [18] and uses a predictor-corrector strategy with the HKM search direction.

We propose a different computational approach for the matrix M than the other interior-point solvers for SDO. Instead of one-dimensional data distribution (see Section 4.3) when computing M we use a two-dimensional distribution. In this way we decrease the communication overhead on computer clusters with slow interconnects such as 100Mbps (Megabits per second) and 1Gbps (Gigabits per second) ethernet. The infrastructure for ethernet is cheaper by factor of six to twenty compared with the high performance computing (HP) 10Gbps interconnects as Myrinet or Infiniband [1]. Our aim is to achieve better utilization of the existing non-HP network and computational infrastructure, and in this way lower the cost of the parallel computing.

A new feature is implemented to deal explicitly with SDO problems that have rank-one constraint matrices. As a result a significant speedup is achieved when computing M for such problems. To the best of our knowledge our software is the first parallel primal-dual interior-point solver to exploit rank-one structure explicitly. The only option available so far is to use SDPT3 (no parallel implementation available yet) or the dual scaling algorithm implemented in PDSDP [10].

Our solver preserves the 64bit computational arithmetic feature of CSDP5.0 and makes it possible to solve very large semidefinite optimization problems that

lie beyond the 32bit addressing space (or 4GB memory). In this way we make it possible to solve large-scale SDO problems, that require much more RAM (random access memory) than 4GB, on a cluster of 64bit PC's with at most 4GB of memory each. Such large problems usually require an expensive shared memory multiprocessor computer with a large amount of RAM. Hence, our solver offers a cost effective way to solve large-scale SDO problems using clusters.

Next we present the algorithmic framework behind our implementation and its features.

5.1 The Algorithm

There are different ways to formulate the primal and the dual SDO problem. One widely used approach is to define (P) and (D) as in Chapter 2, i.e., to consider the primal problem to be a minimization and the dual problem a maximization one. Helmberg et al. [60] proposed a definition of the primal SDO problem as a maximization of the objective function and the dual as a minimization of the dual objective. Such approach was adopted for practical implementations of primal-dual IPM's by Yamashita et al. [113] and Borchers [18]. This alternative approach introduces only minor changes to what we already presented with respect to the interior-point methods for SDO in Chapter 2. Since we use the CSDP 5.0 [18] solver as a base for our development, we will use this second approach to describe the algorithmic framework of our solver.

We consider the semidefinite optimization (SDO) problem formulation in primal form

$$\begin{array}{lll} \text{(SDP)} & \max\limits_{X} & \mathbf{Tr}(CX) \\ & \text{s.t.} & \mathcal{A}(X) = b, \\ & & X \succeq 0, \end{array} \tag{5.1}$$

where $X \in \mathbb{S}^n_+$, and $\mathcal{A}(\cdot)$ is a linear operator defined by

$$\mathcal{A}(X) = \begin{bmatrix} \mathbf{Tr}(A_1X) \\ \mathbf{Tr}(A_2X) \\ \vdots \\ \mathbf{Tr}(A_mX) \end{bmatrix}. \tag{5.2}$$

The formulation (5.1) is obtained by replacing C by -C in (2.1). All constraint matrices $A_i \in \mathbb{S}^n$ (*i.e.* are symmetric $n \times n$) for $i = 1, \cdots, m$, as is the matrix C. Thus n denotes the size of the matrix variables and m the number of equality constraints. The dual of the (SDP) problem is given by

$$\begin{array}{lll} \text{(SDD)} & \min\limits_{y,Z} & b^T y \\ & \text{s.t.} & \mathcal{A}^*(y) - Z = C, \\ & & Z \succeq 0, \end{array}$$

where

$$\mathcal{A}^*(y) = \sum_i^m y_i A_i,$$

is the adjoint of $\mathcal{A}(\cdot)$ with respect to the usual trace product. The primal X and the dual (y, Z) variables are interior feasible solutions of (SDP) and (SDD), respectively, if they satisfy their constraints as well as $X \succ 0$ and $Z \succ 0$. As we already described in details in Chapter 2, the idea behind primal-dual IPM's is to 'follow' the central path

$$\mathcal{C} := \left\{(X(\mu), y(\mu), Z(\mu)) \in \mathbb{S}^n_+ \times \mathbb{R}^m \times \mathbb{S}^n_+ : \mu > 0\right\},$$

where each $(X(\mu), y(\mu), Z(\mu))$ is the solution of the system of equations

$$\begin{aligned} b - \mathcal{A}(X) &= 0, \\ Z + C - \mathcal{A}^*(y) &= 0, \\ ZX - \mu I &= 0, \\ Z, X &\succ 0, \end{aligned} \tag{5.3}$$

where I denotes the identity matrix of size $n \times n$, and $\mu > 0$.

The algorithm used in CSDP5.0 is an *infeasible-start method* and it is designed to work with starting point $X \succ 0, Z \succ 0$ that is not necessarily feasible. An alternative approach used in some SDO solvers as SeDuMi [99] is to use a *self-dual embedding* [34] technique to obtain a feasible starting point on the central path. In our implementation we use the default CSDP5.0 starting point (a similar approach is used in [107]):

$$\begin{aligned} X &= \alpha I, \\ y &= 0, \\ Z &= \beta I, \end{aligned} \tag{5.4}$$

where

$$\begin{aligned} \alpha &:= n \max_k \left(\frac{1 + \mid b_k \mid}{1 + \|A_k\|_F} \right), \qquad k = 1, ..., m, \\ \beta &:= \frac{1}{\sqrt{n}} \left(1 + \max(\max_k(\|A_k\|_F), \|C\|_F) \right), \qquad k = 1, ..., m, \end{aligned}$$

and $\|\cdot\|_F$ denotes the Frobenius norm of a matrix and $|\cdot|$ the absolute value of a real number. It is easy to see that this initial point may not satisfy $\mathcal{A}(X) = b$ or $\mathcal{A}^*(y) - Z = C$, so we write (2.7) and (2.8) for (SDP) and (SDD), respectively as

$$\begin{aligned} R_p &:= b - \mathcal{A}(X), \\ R_d &:= Z + C - \mathcal{A}^*(y). \end{aligned}$$

The algorithm implemented in CSDP5.0 uses the Mehrotra predictor-corrector strategy. The affine-scaling step is computed from:

$$\begin{aligned} -\mathcal{A}(\triangle X^a) &= -R_p, \\ \triangle Z^a - \mathcal{A}^*(\triangle y^a) &= -R_d, \\ Z\triangle X^a + \triangle Z^a X &= -ZX. \end{aligned} \tag{5.5}$$

Using the same approach as in [60], we can reduce the system of equations (5.5) to

$$\begin{aligned} \mathcal{A}(Z^{-1}\mathcal{A}^*(\triangle y^a)X) &= -b + \mathcal{A}(Z^{-1}R_d X), \\ \triangle Z^a &= \mathcal{A}^*(\triangle y^a) - R_d, \\ \triangle X^a &= -X + Z^{-1}R_d X - Z^{-1}\mathcal{A}^*(\triangle y^a)X. \end{aligned} \tag{5.6}$$

If we define

$$\begin{aligned} M &:= [\mathcal{A}(Z^{-1}A_1X), \mathcal{A}(Z^{-1}A_2X), \ldots, \mathcal{A}(Z^{-1}A_mX)], \\ h &:= -b + \mathcal{A}(Z^{-1}R_dX), \end{aligned} \tag{5.7}$$

then we can write the first equation in (5.6) in a matrix form as follows:

$$M\triangle y^a = h. \tag{5.8}$$

Note that (5.8) was obtained by reducing the system of equations (5.5). Therefore, in the literature (5.8) often refereed a Schur complement equation, and the matrix M, defined by (5.7), is called Schur complement matrix.

As Helmberg, Rendl, Vanderbei and Wolkowicz have shown, the matrix M is symmetric and positive definite [60]. Thus we can compute the Cholesky factorization of M to solve the system of equations (5.8). By back substitution of $\triangle y^a$ into the second and the third equation of (5.6) we can subsequently compute $\triangle Z^a$ and $\triangle X^a$, respectively. Note that in this case $\triangle X^a$ obtained by the third equation in (5.6) is not necessarily symmetric. In order to keep X symmetric we need to symmetrize $\triangle X^a$. In case of the HKM search direction this is done by

$$\triangle X^a = \frac{\triangle X^a + (\triangle X^a)^T}{2}. \tag{5.9}$$

As a result, $\triangle X^a$ obtained by (5.9) results in the same expression as (2.87) in Section 2.4.2 from Chapter 2 for $\mu = 0$, namely

$$\triangle X^a = -X - \frac{1}{2}(X\triangle Z^a Z^{-1} + Z^{-1}\triangle Z^a X). \tag{5.10}$$

To show the equivalence between (5.9) and (5.10), we first express $\mathcal{A}^*(\triangle y^a)$ from the second equation in (5.6) and substitute it into the righthand-side of the third equation of the system (5.6). As a result we get

$$\triangle X^a = -X + Z^{-1}R_dX - Z^{-1}(\triangle Z^a + R_d)X = -X - Z^{-1}\triangle Z^a X. \tag{5.11}$$

Since the matrices X and Z^{-1} are symmetric, we can rewrite (5.9) using (5.11) as

$$\begin{aligned}\frac{\triangle X^a + (\triangle X^a)^T}{2} &= \frac{1}{2}\left(-2X - Z^{-1}\triangle Z^a X - (Z^{-1}\triangle Z^a X)^T\right)\\ &= -X - \frac{1}{2}\left(Z^{-1}\triangle Z^a X + X\triangle Z^a Z^{-1}\right),\end{aligned}$$

which is the same as (5.10).

For the corrector step one needs to solve the linear system

$$\begin{aligned}-\mathcal{A}(\triangle X^c) &= 0,\\ \triangle Z^c - \mathcal{A}^*(\triangle y^c) &= 0,\\ Z\triangle X^c + \triangle Z^c X &= \mu I - \triangle Z^a \triangle X^a,\end{aligned} \tag{5.12}$$

where $\mu = \frac{\mathbf{Tr}(XZ)}{2n}$.

The equation system (5.12) is solved similarly to (5.5), and $(\triangle X^c, \triangle y^c, \triangle Z^c)$ is obtained from

$$\begin{aligned}\mathcal{A}(Z^{-1}\mathcal{A}^*(\triangle y^c)X) &= \mu\mathcal{A}(Z^{-1}) - \mathcal{A}(Z^{-1}\triangle Z^a\triangle X^a),\\ \triangle Z^c &= \mathcal{A}^*(\triangle y^c),\\ \triangle X^c &= \mu Z^{-1} - Z^{-1}\mathcal{A}^*(\triangle y^c)X - Z^{-1}\triangle Z^a\triangle X^a.\end{aligned} \tag{5.13}$$

Using the same notation as before,

$$h^c := \mu\mathcal{A}(Z^{-1}) - \mathcal{A}(Z^{-1}\triangle Z^a\triangle X^a), \tag{5.14}$$

we can write the first equation in (5.13) in a matrix form as follows:

$$M\triangle y^a = h^c. \tag{5.15}$$

Note that the matrix M, defined by (5.7), is the same as for the affine-scaling direction. Hence, it is not necessary to compute the Cholesky factorization of M again and we can solve (5.15) immediately. By back substitution of $\triangle y^c$ into the second and the third equation of (5.13) we can subsequently compute $\triangle Z^c$ and $\triangle X^c$, respectively. In this case, $\triangle X^a$ obtained by the third equation in (5.13) again is not necessarily symmetric. We apply the same approach to symmetrize it as on the affine-scaling step, namely

$$\triangle X^c = \frac{\triangle X^c + (\triangle X^c)^T}{2}. \tag{5.16}$$

It can be shown similarly to the previous case that the resulting expression from (5.16) is the same as (2.89) in Chapter 2, for $\sigma = 1$, i.e.,

$$\triangle X^c = \mu Z^{-1} - \frac{1}{2}(\triangle X^a\triangle Z^a Z^{-1} + Z^{-1}\triangle Z^a\triangle X^a) - \frac{1}{2}(X\triangle Z^c Z^{-1} + Z^{-1}\triangle Z^c X).$$

Next, we add the affine-scaling and corrector step to compute the search directions:

$$\begin{aligned} \triangle X &= \triangle X^a + \triangle X^c, \\ \triangle y &= \triangle y^a + \triangle y^c, \\ \triangle Z &= \triangle Z^a + \triangle Z^c. \end{aligned} \tag{5.17}$$

Finally, the maximum step lengths of the steps α_P and α_D are computed such that the update $(X + \alpha_P \triangle X, y + \alpha_D \triangle y, Z + \alpha_D \triangle Z)$ results in a feasible primal–dual point, see (2.31) and (2.32) in Chapter 2.

In practice, the Schur complement matrix M may become numerically singular even though X and Z are numerically nonsingular. In this case, the algorithm returns to the previous solution, and executes a corrector step with $\mu = \frac{\mathbf{Tr}(XZ)}{n}$.

The default stopping criteria are the following

$$\begin{aligned} \frac{\mid \mathbf{Tr}(CX) - b^T y \mid}{1 + |b^T y|} &< 10^{-7}, \\ \frac{\|\mathcal{A}(X) - b\|_2}{1 + \|b\|_2} &< 10^{-7}, \\ \frac{\|\mathcal{A}^*(y) - Z - C\|_F}{1 + \|C\|_F} &< 10^{-7}, \\ Z, X &\succeq 0, \end{aligned} \tag{5.18}$$

where $\|\cdot\|_2$ denotes the Euclidean vector norm.

The solution of the linear system (5.8) involves the construction and the Cholesky factorization of the Schur complement matrix $M \succ 0$ that has size m by m. For dense X, Z and A_i $i = 1, \ldots, m$, the worst-case complexity in computing M is $\mathcal{O}(mn^3 + m^2n^2)$ [83]. In practice the constraint matrices are very sparse and we exploit sparsity in the construction of the Schur complement matrix in the same way as in [46]. For sparse A_i's with $\mathcal{O}(1)$ entries, the matrix $Z^{-1}A_iX$ for $i = 1, ..., m$ can be computed in $\mathcal{O}(n^2)$ operations $(i = 1, \ldots, m)$. Additionally $\mathcal{O}(m^2)$ time is required for $\mathcal{A}(\cdot)$ operations. Finally, the Schur complement matrix M is typically fully dense and its Cholesky factorization requires $\mathcal{O}(m^3)$ operations.

5.2 The approach

In this section we describe our distributed IPM solver for SDO that we will call from now on *PCSDP*. The code is based on CSDP5.0 and it enjoys the same 64-bit capability as the shared memory version of Borchers and Young [20]. It is written in ANSI C with additional MPI[1] directives and use of ScaLAPACK[22] library for

[1] http://www.netlib.org/mpi/

parallel algebra computations. The sequential part of the PCSDP solver is kept the same as in CSDP5.0. It makes use of optimized BLAS [15] and LAPACK [4] libraries. The latter are used for implementation of non-parallel operations such as matrix multiplication, Cholesky factorization of the primal and the dual variable X and Z, respectively, as well as other linear algebra operations.

PCSDP makes use of distributed computational routines for the following operations:

- Computation of the matrix M;
- Cholesky factorization of M;
- Solution of the linear systems (5.8) and (5.15).

We already mentioned in the previous chapter that often (but not always) the most time-consuming operations are the first two in this list. Therefore, in our development we used parallelization to accelerate these two bottlenecks in the sequential algorithm. Solving systems (5.8) and (5.15) by a distributed algorithm is essential, because in this way M is stored only in a distributed way. As a result, the software does not require that any of the computer nodes of the cluster should be able to accommodate the whole matrix into its local memory space. Next, we give more details on the implementation.

Assume that we have N processors available, and we attach a corresponding process $P_j, j = 0, .., N-1$ to each one of them. We call process P_0 a *root* process. When PCSDP starts, the *root* node sends a copy of the execution and (input) data files to the $N-1$ nodes left. Each node independently proceeds to allocate space for the variables X, y, Z locally. Then all nodes start the execution of their copy of the code, until computation of the Schur complement matrix M is reached. At that moment, the computational process synchronizes and the distributed computation of M is started. We give more details on this process in the next section. After the matrix M is computed, the algorithm proceeds with parallel Cholesky factorization and solution of the linear system (5.8), both performed by ScaLAPACK routines. The resulting vector $\triangle y^a$ appears in a distributed format. We use a redistribution procedure that makes a copy of $\triangle y^a$ on all N nodes. From that point on all nodes continue with their local redundant computations of the primal-dual IPM algorithm until the solution of (5.15) is reached. Since M is factorized already we just solve (5.15) by distributed routine and make a local copy of $\triangle y^c$ in the same way as for the affine-scaling step. All the nodes resume their sequential computation of the interior-point algorithm until M has to be computed again, or PCSDP terminates if the stopping criteria described the previous section are satisfied.

5.2.1 Computing the Schur Complement Matrix

The efficiency of the Cholesky factorization, from the ScaLAPACK library, is achieved when matrix M is in two-dimensional block cyclic data distribution.

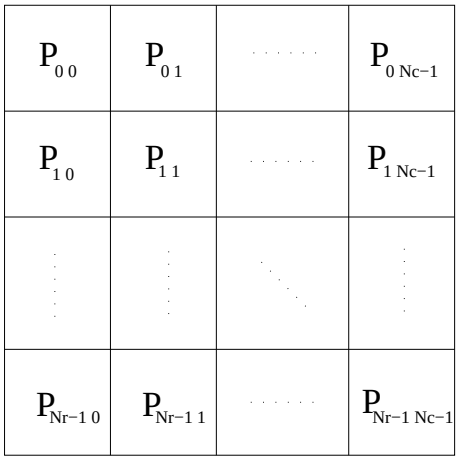

Figure 5.1: *Two dimensional process grid $N_r \times N_c$.*

The traditional approach used in the SDPARA, SDPARA-C and PDSDP solvers is to compute M using one-dimensional cyclic row distribution. Subsequently, its elements are redistributed to the desired two-dimensional distribution, see Section 4.3 in Chapter 4. This introduces additional communication overhead for large-scale SDO problems. To be able to avoid this overhead, we propose a completely different approach than the other SDO solvers.

Instead of using a one-dimensional array of processes, we map the N processes into a two-dimensional rectangular array, often called a *process grid*. Let this process grid have $N_r \geq 1$ rows and $N_c \geq 1$ columns, where $N_r N_c = N$. Figure 5.1 shows such process grid. Each process is indexed by its row and column coordinates as P_{rc} with $0 \leq r \leq N_r - 1$ and $0 \leq c \leq N_c - 1$. It is easy to see that when $N_c = 1$ we have as a special case a one-dimensional array of processes. By using such process grid we can use the same two-dimensional block cyclic data distribution for computing the matrix M, factorizing it and solving both linear systems (5.8) and (5.15). In this way no data redistribution between the processors is necessary with respect to the Schur complement matrix.

To be able to map matrix M on the process grid, it is first subdivided into blocks of size $n_b \times n_b$ for a suitable value of $n_b > 1$ and the assignment of block to processes are as shown on Figure 5.2. The reason to choose square blocks is that in this way the Cholesky factorization that follows will reach its best parallel performance. Recall from Chapter 4 that M is symmetric and we need only to compute its lower triangular part. The element M_{ij} is assigned for computation

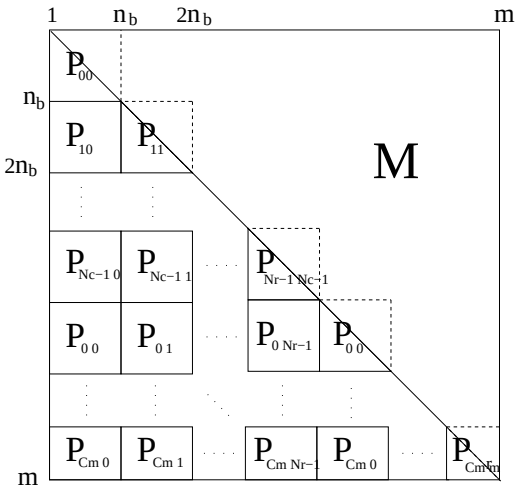

Figure 5.2: *Two-dimensional block cyclic data distribution over two dimensional process grid.*

by process P_{rc}, where the specific values for r and c can be obtained by

$$\begin{aligned} r &:= \left\lfloor \frac{i-1}{n_b} \right\rfloor mod\ N_r, \quad i = 1, ..., m, \\ c &:= \left\lfloor \frac{j-1}{n_b} \right\rfloor mod\ N_c, \quad j = i, ..., m. \end{aligned}$$

Each processor computes and stores in its local memory only the elements that were assigned to it. During the complete solution cycle of the algorithm, those elements stay local and are not needed by the other processors. As a result, PCSDP does not require that any of the nodes should be able to accommodate the whole matrix into its memory space.

A good load balance is difficult to achieve when a two-dimensional block cyclic layout is used. The quality of the load balance depends on the choices of n_b, N_r and N_c. If we choose $n_b = 1, N_c = 1$ and $N_r = N$ we will have a very good load balance, i.e., we will have a one-dimensional block row distribution. On the other hand, the distributed Cholesky factorization of M, performed by ScaLAPACK routine, reaches its best performance when the process grid is as square as possible ($N_r \approx N_c$). Later, we show that the use of the square process grid is very suitable for Beowulf clusters with GBit ethernet interconnect. Such interconnects have relatively high latency ($50-100\mu s$) compared with Myrinet or Infiniband networks ($6 - 9\mu s$).

5.2.2 Parallel Cholesky factorization of the matrix M

In PCSDP we used a distributed routine for Cholesky factorization provided by the ScaLAPACK library. The aim was better portability on different computational platforms (Linux, UNIX, AIX, etc.), hence the choice of the library package. ScaLAPACK also includes routines suitable for solving in parallel both large-scale linear systems (5.8) and (5.15).

The block-partitioned form of Cholesky factorization is known to have in practice very good performance and scalability, see e.g. [37]. The routine we use to factorize M is PDPOTRF and it is a parallel version of the right-looking factorization in block-partitioned form is implemented in LAPACK routine DPOTRF, presented in Chapter 4. Details about the actual algorithm and parallel implementation behind PDPOTRF can be found in Choi et al. [23].

5.2.3 Exploiting rank-one structure in SDO

In this subsection we describe the built-in capability of PCSDP to deal efficiently with rank-one constraint matrices. So far no primal-dual interior-point solver for SDO offers this option, only the dual-scaling algorithm implemented in PDSDP.

Assume the constraint matrices have the form $A_i = a_i {a_i}^T$, $a_i \in \mathbb{R}^n$ and $i = 1, ..., m$. We will refer to this type of structure as 'rank-one'. It appears in many large-scale problems coming from combinatorial optimization problems and optimization of univariate functions by interpolation [29]. Our aim next is to use this special structure of A_i's to speed up computation of the matrix M when using a primal-dual IPM for SDO. The approach we use is basically the one proposed by Helmberg and Rendl [59]. The elements of the matrix M, defined by (5.7), can be computed from

$$M_{ij} \quad = \quad \mathbf{Tr}(A_i Z^{-1} A_j X) \qquad (i, j = 1, \ldots, m).$$

Since $A_i = a_i {a_i}^T$, this reduces to

$$M_{ij} \quad = \quad ({a_i}^T Z^{-1} a_j)({a_i}^T X a_j) \qquad (i, j = 1, \ldots, m). \tag{5.19}$$

Computing ${a_i}^T Z^{-1}$ and ${a_i}^T X$ only once for each row, leads to $\mathcal{O}(mn^2 + m^2 n)$ arithmetic operations, see e.g. [59]. In practice this can be improved significantly for sparse a_i's.

We would like to store only the vectors a_i as opposed to the matrices $A_i = a_i a_i^T$, because this obviously will reduce the storage requirements by factor of n. On the other hand, we still want to use the standard SDPA sparse input data format [113], which does not have an option to store rank-one matrices efficiently. To overcome this difficulty we propose a storage of each vector $a_i, i = 1, ..., m$ as a diagonal matrix. In addition we introduce in PCSDP a new parameter call *rank1* in the parameters setup file of our software. It ensures that the rank-one constraint matrices are interpreted by PDSDP in the right way (i.e. not as diagonal matrices, but as a rank-one matrices defined by a vector).

Note that rank-one constraints matrices, stored as vectors, are only implicitly available. Our software never computes them explicitly as $A_i = a_i {a_i}^T$, $i = 1, ..., m$. In this way a significant reduction of the memory requirements of PCSDP could be achieved when A_i's are dense. In the next chapter we present numerical experiments with SDO problems with rank-one SDO structure and give more details.

5.3 Conclusion

In this chapter we presented the algorithmic framework behind our implementation of a parallel primal-dual IPM solver for semidefinite optimization problems. We took different approaches in distributed computation of the Schur complement matrix M than the other existing parallel software packages. As a result, the communication overhead is eliminated after M is computed. We also solve the Schur complement systems, that involve M, in parallel. Therefore, PCSDP does not require that any of the nodes in the cluster should be able to accommodate into its memory the whole $m \times m$ matrix M. This is an important result for problems with very large number of constraints.

Our software also offers 64bit computational arithmetic. It makes it possible to solve large-scale semidefinite optimization problems that lie beyond the 32bit addressing space (or 4GB of memory). In the next chapter we solve problems that require more than 4 GB of memory to store only the Schur complement matrix M.

PCSDP can explicitly deal with the rank-one structures in the constraint matrices. Exploiting such structure, when possible, leads to a significant speed-up in the computations of the SDO problems.

Chapter 6

Computational Results

In this chapter we present results from the numerical tests of our parallel IPM solver for SDO, PCSDP. The software was developed and tested on the DAS3 cluster at the Delft University of Technology, The Netherlands. Each node of the cluster has two 64bit AMD[1] Opteron 2.4 GHz CPUs, running ClusterVisionOS Linux, and has 4 GB memory. Communication between the nodes relies on a 1GBit ethernet network, which is used as an interconnect and network file transport. All parameter values in PCSDP were set to the values described in Chapter 5.

As is customary, we measured speedups S and parallel efficiencies E, defined by

$$S = \frac{T_1}{T_N},$$

and

$$E = \frac{S}{N} = \frac{T_1}{(NT_N)}, \tag{6.1}$$

where T_1 and T_N are the times to run a code on one processor and N processors respectively. Note that $0 \leq E \leq 1$ and $E = 1$ corresponds to perfect speedup.

6.1 SDO benchmark problems

We selected for our numerical tests twenty medium and large-scale SDO problems from eight different applications. These applications are: control theory, the crossing number of complete bipartite graphs, the maximum cut problem, the *Lovász* ϑ-function, the min k-uncut problem, calculating electronic structures

[1] http://www.amd.com/

in quantum chemistry, and semidefinite relaxation of quadratic assignment and traveling salesman problems. All of the test instances are from a standard benchmark suites, except the last five. They are newly generated instances by De Klerk at al. [32, 35]. More details about the selected set of instances can be found in Table 6.1. In this table m is the number of constraints, n denotes the size of the primal matrix variable X, and n_{max} is the size of the largest block in the common matrix data structure of the constraint matrices A_i. For more details on the block structured matrices and the sparse SDPA format see Yamashita et al. [113, Section 4.6]. We also give in the fifth column the optimal values of the objective function, corresponding to the problems.

Name	m	n	n_{max}	Optimal value
control10	1326	150	100	$3.8533E+01$
control11	1596	165	110	$3.1959E+01$
theta8	7905	400	400	$7.3953559E+01$
theta42	5986	200	200	$2.3931707E+01$
theta62	13390	300	300	$2.9641248E+01$
theta82	23872	400	400	$3.4366889E+1$
thetaG51	6910	1001	1001	$3.49000E+02$
maxG51	1000	1000	1000	$4.003809E+03$
hamming_8_3_4	16129	256	256	$2.560000e+01$
hamming_9_5_6	53761	512	512	$8.5333332E+01$
hamming_10_2	23041	1024	1024	$1.024E+02$
hamming_11_2	56321	2048	2048	$1.7066666E+02$
fap09	15225	174	174	$-1.0797803E+01$
CH4.1A1.STO6G	24503	630	324	$1.3021808E+01$
LiF.1Sigma.STO6G	7230	5990	1450	$-1.1558005E+02$
Esc64a	517	4618	4097	$9.7749966E+01$
crossing_n8	239	620	380	$5.859985E+00$
crossing_n9	1366	3805	2438	$7.735211E+00$
GR17tsp	3673	6666	17	$2.0063685e+03$
BAYS29tsp	6526	15981	29	$1.9997581e+03$

Table 6.1: *Selected SDO benchmark problems.*

Problems control10 and control11 are from control theory and they are the

largest of this type in SDPLIB test set [19]. The instance maxG51 is a medium size max-cut problem chosen from the same benchmark set. ThetaG51, theta42, theta62, theta8, and theta82 are Lovász ϑ problems. The first one is from SDPLIB, while the others were generated by Toh and Kojima [106]. The instances hamming_8_3_4, hamming_9_5_6, hamming_10_2 and hamming_11_2 compute the ϑ function of Hamming graphs. All four of them are from the DIMACS [80] library of mixed semidefinite-quadratic-linear programs, as is the min k-uncut test problem fap09. The fourteenth and fifteenth instances are CH4.1A1.STO6G and LiF.1Sigma.STO6G, respectively, and they come from calculating electronic structures in quantum chemistry [86]. Instance Esc64a is semidefinite relaxation of the quadratic assignment problem taken from QAPLIB [21], see for details [35]. Both crossing_n8 and crossing_n9 problems compute a lower bound of the crossing number of complete bipartite graphs (see Chapter 1, on page 7). The last two instances GR17tsp and BAYS21tsp are SDO relaxations of the traveling salesman problem generated by De Klerk at al. [33].

The initial point used during the numerical experiments is as described in (5.4) and the stopping criteria is (5.18).

Recall from Chapter 5, that PCSDP uses distributed computation of the matrix M and its Cholesky factorization. Therefore in the numerical experiments that follow, along with the total running time, we measure also the time spent on these two components computed in parallel. The time spent for computing the Schur complement matrix M and its Cholesky factorization are denoted in the tables as *Schur* and *Cholesky*, respectively.

Fourteen of the selected benchmark semidefinite problems could be solved using from one to sixty four processors on the DAS3 cluster in TU Delft. Table 6.2 gives the time in seconds for these instances. At least 4 nodes were required to accommodate the problem data matrices for the large-scale SDO problems CH4, hamming_9_5_6, hamming_10_2, hamming_11_2, theta82 and crossing_n9. The running times in seconds for these six instances are presented in Table 6.4. *Phase* denotes the three different components we measure: *Schur*, *Cholesky* and the total running time, denoted as *Total*. Columns after the fifth one in Table 6.2 denote the actual time depending on the number of CPUs used. Instances CH4.1A1.STO6G and LiF.1Sigma.STO6G are abbreviated in the tables as CH4 and LiF, respectively.

Computing the parallel efficiencies E, using expression (6.1), was not possible for all of the selected twenty test problems. In the case of crossing_n9, despite the relatively small number of constrains (see Table 6.4), the computational resources this instance requires are significant. This makes numerical tests with crossing_n9 a nearly impossible task on a small number of nodes. Computing the parallel efficiencies was not possible for some other test problems too, though: when the number of constraints exceed $20,000$, the memory required to accommodate the problem exceeded the available memory on an individual node. Therefore, Table 6.3 presents only the parallel efficiencies of the problems that can run on one node, i.e., for the fourteen instances in Table 6.2.

Problem	m	n_{max}	Phase	CPUs						
				1	2	4	8	16	32	64
control10	1326	100	Schur	216.4	113.2	76.5	40.2	27.9	17.4	12.8
			Cholesky	9.9	6.6	6.1	5.3	3.7	3.0	2.7
			Total	240.2	132.6	107.3	56.4	39.0	32.9	24.8
control11	1596	110	Schur	340.7	169.5	116.4	62.5	50.2	29.2	21.5
			Cholesky	17.7	8.4	6.8	6.7	4.6	5.2	3.5
			Total	377.2	192.0	136.7	82.0	67.9	49.3	35.8
crossing_n8	239	380	Schur	561.5	309.8	271.0	146.5	127.4	72.7	67.5
			Cholesky	0.2	0.3	0.5	0.9	1.1	0.4	1.4
			Total	617.1	369.0	329.1	208.9	190.0	137.8	133.4
theta42	5986	200	Schur	78.5	35.7	20.7	11.6	6.8	3.8	2.8
			Cholesky	555.3	173.4	112.8	81.9	47.3	43.6	25.1
			Total	696.9	248.9	164.4	116.5	74.4	65.5	45.7
theta8	7905	400	Schur	163.1	84.2	44.6	22.4	13.8	8.0	6.2
			Cholesky	1445.1	403.2	251.6	174.2	99.1	88.3	48.0
			Total	1730.9	596.3	359.6	245.5	156.8	135.7	91.1
theta62	13390	300	Schur	521.1	259.8	131.9	67.2	39.4	19.9	13.6
			Cholesky	4977.2	1693.2	1002.0	621.5	346.9	287.3	146.3
			Total	5807.8	2152.4	1292.4	796.7	482.1	408.8	238.9
thetaG51	6910	1001	Schur	1003.4	482.6	252.4	128.1	68.3	35.7	21.4
			Cholesky	2249.7	616.1	401.3	278.2	159.3	143.6	79.9
			Total	3700.5	1454.3	989.1	713.6	525.9	472.4	394.0
maxG51	1000	1000	Schur	1.2	1.0	0.5	0.3	0.2	0.1	0.1
			Cholesky	2.6	2.1	2.0	2.0	1.9	1.9	1.8
			Total	112.2	110.7	111.2	111.8	111.6	111.7	113.5
hamming_8_3_4	16129	256	Schur	622.9	282.1	172.1	91.9	47.0	25.5	18.1
			Cholesky	6938.8	2502.3	1452.0	875.0	465.0	334.3	197.1
			Total	8258.6	3025.9	1806.0	1101.3	658.1	561.6	313.0
fap09	15225	174	Schur	5682.2	3087.1	2013.3	1044.3	715.6	359.2	299.7
			Cholesky	37827.1	12614.4	7492.7	4496.0	2532.1	1836.5	1009.7
			Total	46613.5	17033.4	10518.1	6288.0	3932.4	3163.6	1857.4
LiF	7230	1450	Schur	187397.1	101331.7	52501.5	26023.2	15115.56	9469.1	6705.2
			Cholesky	2317.0	672.9	479.6	304.9	283.0	208.5	213.4
			Total	192080.7	104382.5	56209.8	28578.0	17601.1	11769.3	9224.4
Esc64a	517	4097	Schur	19944.91	15586.06	13895.43	12500.78	12222.75	11857.14	11805.26
			Cholesky	459.41	419.00	339.81	335.81	300.28	286.54	262.90
			Total	29144.51	28414.97	26056.74	24498.22	24378.90	23828.02	23095.44
GR17tsp	3673	6394	Schur	201.6	86.4	43.3	22.3	15.4	9.3	6.3
			Cholesky	617.4	224.5	135.9	108.7	74.9	67.9	46.9
			Total	1072.2	517.1	323.9	251.4	193.8	207.7	179.8
BAYS29tsp	6526	15169	Schur	16880.7	8170.1	4187.1	2862.5	1502.2	801.1	421.6
			Cholesky	3568.9	1075.9	759.8	509.3	470.5	350.4	345.5
			Total	21384.1	10937.3	5502.9	3928.9	2495.3	1673.7	1311.9

Table 6.2: *Running times for the selected SDO benchmark problems for PCSDP.*

In Table 6.3 several problems show parallel efficiency beyond one. This is caused by the difference in the number of iterations when running on one CPU as well as the cache issues in the computing nodes that additionally speedup the parallel computations in parallel.

For problems control10, control11, and crossing_n8 computing the Schur complement matrix is the dominant operation and it scales relatively well with the number of the processors up to 16. However this is not the case with Cholesky factorization of M and the total running time. When the number of processors is 16, 32 and 64 the parallel efficiency results are poor. The reason is that the

$m \times m$ Schur complement matrix is fully dense and its Cholesky factorization does not inherit much from the structure of the problem. Although the block structure and the sparsity of X, Z and A_i, $i = 1, ..., m$ are effectively utilized in the matrix multiplications, they do not affect the scalability of distributed computing of *Cholesky*. Additionally, the high latency of the 1GBit ethernet network

Problem	m	n_{max}	Phase	CPUs						
				1	2	4	8	16	32	64
control10	1326	100	Schur	1	0.96	0.71	0.67	0.48	0.39	0.26
			Cholesky	1	0.75	0.40	0.23	0.16	0.10	0.06
			Total	1	0.91	0.56	0.53	0.38	0.23	0.15
control11	1596	110	Schur	1	1.01	0.73	0.68	0.48	0.36	0.25
			Cholesky	1	1.05	0.65	0.33	0.24	0.11	0.08
			Total	1	0.98	0.69	0.57	0.35	0.24	0.16
crossing_n8	239	380	Schur	1	0.91	0.52	0.48	0.28	0.24	0.13
			Cholesky	1	0.32	0.10	0.03	0.01	0.01	0.00
			Total	1	0.84	0.47	0.37	0.20	0.14	0.07
theta42	5986	200	Schur	1	1.10	0.95	0.85	0.73	0.65	0.43
			Cholesky	1	1.60	1.23	0.85	0.73	0.40	0.35
			Total	1	1.40	1.06	0.75	0.59	0.33	0.24
theta8	7905	400	Schur	1	0.97	0.91	0.91	0.74	0.63	0.41
			Cholesky	1	1.79	1.44	1.04	0.91	0.51	0.47
			Total	1	1.45	1.20	0.88	0.69	0.40	0.30
theta62	13390	300	Schur	1	1.00	0.99	0.97	0.83	0.82	0.60
			Cholesky	1	1.47	1.24	1.00	0.90	0.54	0.53
			Total	1	1.35	1.12	0.91	0.75	0.44	0.38
thetaG51	6910	1001	Schur	1	1.04	0.99	0.98	0.92	0.88	0.73
			Cholesky	1	1.83	1.40	1.01	0.88	0.49	0.44
			Total	1	1.27	0.94	0.65	0.44	0.24	0.15
maxG51	1000	1000	Schur	1	0.65	0.58	0.60	0.43	0.39	0.24
			Cholesky	1	0.61	0.33	0.16	0.09	0.04	0.02
			Total	1	0.50	0.25	0.12	0.06	0.03	0.02
hamming_8_3_4	16129	256	Schur	1	1.10	0.90	0.85	0.83	0.76	0.54
			Cholesky	1	1.39	1.19	0.99	0.93	0.65	0.55
			Total	1	1.36	1.14	0.94	0.78	0.46	0.41
fap09	15225	174	Schur	1	0.92	0.71	0.68	0.50	0.49	0.30
			Cholesky	1	1.50	1.26	1.05	0.93	0.64	0.59
			Total	1	1.37	1.11	0.93	0.74	0.46	0.39
LiF	7230	1450	Schur	1	0.92	0.89	0.90	0.77	0.62	0.44
			Cholesky	1	1.72	1.21	0.95	0.51	0.35	0.17
			Total	1	0.92	0.85	0.84	0.68	0.51	0.33
Esc64a	517	4097	Schur	1	0.63	0.36	0.20	0.10	0.05	0.03
			Cholesky	1	0.55	0.34	0.17	0.10	0.05	0.03
			Total	1	0.51	0.28	0.15	0.07	0.04	0.02
GR17tsp	3673	6394	Schur	1	1.17	1.16	1.13	0.82	0.68	0.50
			Cholesky	1	1.38	1.14	0.71	0.52	0.28	0.21
			Total	1	1.04	0.83	0.53	0.35	0.16	0.09
BAYS29tsp	6526	15169	Schur	1	1.03	1.01	0.74	0.70	0.66	0.63
			Cholesky	1	1.66	1.17	0.88	0.47	0.32	0.16
			Total	1	0.97	0.97	0.68	0.53	0.40	0.25

Table 6.3: *Parallel efficiencies of PCSDP for the selected SDO problems.*

interconnect between the nodes results in relatively slow message delivery times. Therefore, when the problem has fewer than $5,000$ constraints and matrix size $n < 3,000$, like control10, control11, and crossing_n8, it is not very efficient to solve it by distributed memory cluster using more than 8 processors.

Poor scalability of *Cholesky* is also observed for the SDO relaxations of the

traveling salesman problem GR17tsp and BAYS21tsp. The parallel efficiency of *Schur* remains at least 0.5 in all of the test cases, but the overall scalability drops rapidly when more than 16 processors are used. Both problems have large sizes of n compared to the number of constraints m.

The *Lovász* ϑ-type instances theta42 and theta8 as well as thetaG51 also exhibit poor parallel efficiency. The difference with the problems discussed in the previous paragraph is that scalability of both components *Schur* and *Cholesky* remain quite reasonable up to 64 processors, but the total running time scales poorly above 16 CPUs. This is to be expected due to the smaller portion of time *Schur* and *Cholesky* occupy compared to the matrix operations involving the primal and dual matrix variables X and Z.

Another example that sufferers from low parallel-to-nonparallel ratio is the max-cut problem maxG51, where $n = m$ and $n = 1000$ (i.e. relatively small). The constraint matrices themselves are very sparse (and rank-one). As a result computing the elements of M and its Cholesky factorization takes a very small part in the total solution time. The dominant operations in this case are factorization of the primal matrix variable. Therefore, the overall scalability is poor and solving such a problem using primal-dual IPM solver is very inefficient. Similar problems with the scalability of this problem was observed in [114].

Instance Esc64a also gives poor scalability results. The reason is that the number of constraints m is very low compared with the size n of the primal and dual matrix variables. As a result the matrix M is small compared to the X and Z matrices and computing it occupies at most 50% of the total solution time on more than one processor. In this case (sequential) matrix multiplications also take a significant amount of time. Hence, the worse scalability of the total running time. Despite its non-impressive scalability, the solution of instance Esc64a done by PCSDP achieved a significant improvement of the lower bound of this QAP problem, see for details De Klerk et al. [35].

Numerical results so far clearly indicate that problems with the number of constraints under $5,000$ and small n or ones from max-cut are not solved very efficiently on more than 8 processors by PCSDP on a computer cluster. Much better results are obtained for the large-scale instances such as theta62, fap09, LiF, and hamming_8_3_4. Here the parallel efficiency of the running time is above 0.44 for almost the whole range of CPU's used between 2 and 32. Only on 64 CPU's it is slightly lower, at least 0.33. Both the Cholesky factorization and construction of the Schur complement matrix scale well with the number of processors for all four problems. When these two computations dominate in the overall running time, one sees a good overall scalability of the problem.

As we already mentioned above, we were not able to solve the largest test problems CH4, hamming_9_5_6, hamming_10_2, hamming_11_2, theta82 and crossing_n9 on fewer than 4 nodes. In Table 6.4 we therefore present only the running times in seconds when 4, 16, 32 and 64 processors were used. At least 4 nodes were required to accommodate the Schur complement matrix for problems with around $24,000$ constraints like CH4, hamming_10_2 and theta82. They have a small di-

Problem	m	n_{max}	Phase	CPUs				
				4	8	16	32	64
CH4	24503	324	Schur	2433.7	1150.7	666.4	339.8	280.8
			Cholesky	12584.1	6601.7	3511.0	3038.7	1404.8
			Total	16144.0	8497.1	4814.1	3944.2	2280.5
hamming_9_5_6	53761	512	Schur	*	*	736.2	363.9	262.9
			Cholesky	*	*	14165.1	11793.7	4395.6
			Total	*	*	16277.3	13343.8	5793.0
hamming_10_2	23041	1024	Schur	491.0	254.9	168.6	89.21	62.0
			Cholesky	4943.6	2826.3	1538.6	1334.5	567.3
			Total	6023.8	3552.3	2143.7	1813.6	1010.9
hamming_11_2	56321	2048	Schur	*	*	1072.6	638.0	484.2
			Cholesky	*	*	20005.6	16334.9	6037.5
			Total	*	*	23929.6	19516.0	9198.6
theta82	23872	400	Schur	487.6	242.2	146.1	70.9	49.5
			Cholesky	5127.3	2892.3	1591.1	1366.3	580.2
			Total	6079.8	3476.1	2037.4	1694.9	876.6
crossing_n9	1366	2438	Schur	not run	not run	125220.9	76095.0	67931.8
			Cholesky	not run	not run	7.6	7.5	5.6
			Total	not run	not run	143311.3	94750.4	85837.8

Table 6.4: *Running times (in seconds) for the selected large-scale SDO benchmark problems for our solver PCSDP ('*' - means lack of memory)*

mension of the primal and dual matrix variables, hence the parallel operations dominate. As a result, very good scalability is observed not only in computing M and its Cholesky factorization but on the total running time as well. For the truly large-scale problems hamming_9_5_6 and hamming_11_2, at least 16 nodes were needed due to the amount of memory required. They both have more than $50,000$ constraints and 32bit addressing would not be enough to address the elements of the Schur complement matrix M. In this case m is far larger than n and the solver efficiently solved them with a good scalability in terms of running times between 16 and 64 CPUs.

Instance crossing_n9 was solved by using the DAS3 cluster located in Vrije Universiteit (VU) Amsterdam. It has a Myri10G interconnect which is at least 10 times faster than ethernet 1GBit, available at the TU Delft. Each node of the VU cluster has two 64bit dual-core AMD Opteron 2.4 GHz CPUs, and 4GB RAM. The software platform and libraries are exactly the same as on all other experiments. Despite the use of Myri10G, significant resources were required in order to solve crossing_n9 instance. Therefore, we give only the running times for 16, 32 and 64 processors.

Using PCSDP we solved one additional instance named solveC133b which is not included in Table 6.1. It is an SDO relaxation of a 3-assignment problem and is generated by De Klerk [28]. The number of constraints of solveC133b is $m = 75,174$ with dimension $n = 154,937$ of the primal variable X, and $n_{max} = 67$.

Despite the large value of n this problem has a diagonal block in the A_i's of size 152,726. Instance solveC133b was successfully solved by DAS3 using 64 CPUs. The time measures as follows: $Schur = 4,373.57$ sec, $Cholesky = 33,513.50$ sec, and $Total = 52,155.42$ sec. This instance is the largest one solved so far by our PCSDP solver.

6.1.1 Comparison with other IPM solvers

We compare the performance of our parallel software PCSDP[2] with the two other distributed memory SDO solvers freely available, namely SDPARA-1.0.1 by Yamashita *et al.* [114] and PDSDP-5.8 by Benson [10]. The interconnect used was ethernet 1Gbit. The same optimized BLACS[3] and ScaLAPACK-1.8.0 parallel libraries were used.

We selected only benchmark problems from Table 6.2 which could be solved with the same accuracy as PCSDP by SDPARA and PDSDP. Instance crossing_n8 was considered too small in size for a parallel solver, and was therefore omitted from the list of test problems. Large-scale instances LiF, Esc64a, BAYS20tsp and the ones from Table 6.4 require a significant amount of time to obtain their solution. We did not have the necessary time window to solve them with SDPARA and PDSDP on the DAS3 cluster.

Table 6.5 presents the running times, in seconds, for PCSDP, SDPARA and PDSDP using between 1 and 64 processors. In cases where a program could not solve some instance, due to lack of memory, we indicated this in the table by '*'. We did not measure the different parallel components, as the times spent on computing the Schur complement matrix and its Cholesky factorization.

Results from Table 6.5 show that on a relatively small-scale SDO problems such as control10 and control11, SDPARA and PDSDP perform better in terms of total running time. This to be expected, since they both use the same one-dimensional way of computing the Schur complement matrix M. Their approach has better load balance and the necessary network communication for redistributing M is small, due to the small number of constraints these problems have.

Two other instances where PCSDP performs worse than the other software are thetaG51 and maxG51. Due to the specific problem structure, thetaG51 is solved on average 30% faster by SDPARA compared by PCSDP. The solution times of PDSDP are by factor of 3 to 5 greater than for our solver on this problem. On the other hand, the opposite is true for maxG51. The reason is that this type of problems are very sparse and the dual-scaling interior-point algorithm, as in PDSDP, is known to solve them much more efficiently than the primal-dual algorithm implemented in PCSDP and SDPARA.

PCSDP gives better results compared to the other two solvers for the medium sized instances theta42, theta8 and GR17tsp, using from 1 to 64 processors. The

[2]Available at: http://lyrawww.uvt.nl/∼edeklerk/PCSDP/
[3]http://www.netlib.org/blacs/

Problem	m	n_{max}	Solver	CPUs						
				1	2	4	8	16	32	64
control10	1326	100	pcsdp	240.2	132.6	107.3	56.4	39.0	32.9	24.8
			sdpara	109.4	91.6	57.8	37.1	23.6	18.6	20.0
			pdsdp	132.0	84.0	67.9	57.4	37.5	29.6	27.0
control11	3028	250	pcsdp	377.2	192.0	136.7	82.0	67.9	49.3	35.8
			sdpara	279.5	154.1	95.2	58.1	37.5	28.3	27.9
			pdsdp	184.3	125.0	98.4	79.8	70.7	56.4	39.0
theta42	5986	200	pcsdp	696.9	248.9	164.4	116.5	74.4	65.5	45.7
			sdpara	846.4	329.9	202.9	154.2	89.1	82.7	57.6
			pdsdp	1052.0	454.2	248.8	165.9	96.7	85.3	65.6
theta8	7905	400	pcsdp	1730.9	596.3	359.6	245.5	156.8	135.7	91.1
			sdpara	2111.7	768.7	467.7	310.8	191.8	157.7	112.9
			pdsdp	2213.0	912.7	558.4	328.5	211.7	167.8	118.5
theta62	13390	300	pcsdp	5807.8	2152.4	1292.4	796.7	482.1	408.8	238.9
			sdpara	*	*	1820.2	1143.6	607.0	547.6	287.5
			pdsdp	*	*	*	2276.0	998.4	617.2	376.0
thetaG51	6910	1001	pcsdp	3700.5	1454.3	989.1	713.6	525.9	472.4	394.0
			sdpara	2579.0	1057.2	697.1	524.0	381.1	346.9	274.8
			pdsdp	10530.0	6811.1	2975.1	2306.5	1426.0	1133.6	943.3
maxG51	1000	1000	pcsdp	112.2	110.7	111.2	111.8	111.6	111.7	113.5
			sdpara	97.3	98.9	102.5	100.8	99.8	102.2	106.0
			pdsdp	31.6	21.6	15.0	12.9	11.6	11.0	11.7
GR17tsp	3673	6394	pcsdp	1072.2	517.1	323.9	251.4	193.8	207.7	179.8
			sdpara	2785.3	1857.8	938.7	501.3	361.1	248.4	196.0
			pdsdp	1243.9	684.2	362.1	291.2	257.1	219.1	191.8
hamming_8_3_4	16129	256	pcsdp	8258.6	3025.9	1806.0	1101.3	658.1	561.6	313.0
			sdpara	*	*	2763.7	1697.6	930.8	805.1	441.1
			pdsdp	*	*	*	1155.6	567.9	390.5	194.8
fap09	15225	174	pcsdp	46613.5	17033.4	10518.1	6288.0	3932.4	3163.6	1857.4
			sdpara	*	*	*	42247.9	22298.9	12232.1	6218.9
			pdsdp	*	*	*	36940.0	15830.5	9646.1	5385.2

Table 6.5: *Running times in seconds for the selected SDO problems solved by PCSDP, SDPARA and PDSDP.*

communication overhead caused by redistributing the Schur complement matrix on these problems is already significant and adds up in the total running time in the case of SDPARA and PDSDP, so they have worse overall performance than PCSDP.

Of all three solvers only PCSDP was able to solve the instances theta62, hamming_8_3_4 and fap09 in the whole range from 1 to 64 CPUs. The advantage in memory usage comes from the two-dimensional approach in computing the matrix M in PCSDP, that does not require local storage of the matrix M. The Schur complement matrix always remains distributed over the process grid, see

Chapter 5 for details. This is not the case for SDPARA and PDSDP, therefore they have high memory requirements in order to solve large-scale SDO problems.

For the instances theta62 and fap09, PCSDP outperforms the other two programs by a factor of at least 3. Only for hamming_8_3_4 does PDSDP give somewhat better times with between 16 and 64 processors, which is a result of the problems structure.

The overall verdict after the test is that on a large-scale instances, especially when the size of the Schur complement matrix is very large, and the network is not a low latency one, the two-dimensional data approach in our solver results in better running times compared with SDPARA and PDSDP. This approach also leads to less memory usage and makes it possible large-scale SDO problems to be solved on fewer computing nodes.

6.1.2 Discussion

In Chapter 5 we presented our two-dimensional approach for computing the matrix M. It involves the use of a two-dimensional process grid instead of one-dimensional one as in all other distributed SDO solvers. Note that when we refer from now on to the process grid or a 'grid', we mean the two-dimensional way processes are mapped on a program level, i.e., as explained in Section 5.2.1. The notion of process grid and 'grid' does not imply by any means the notion presented in Chapter 3.

Recall that the main argument why we claim that our solution is better suited for cluster (i.e. distributed) computing was that we would decrease the communication overhead after computing *Schur*. To demonstrate our point we implemented a separate experimental function for PCSDP to compute M using one-dimensional cyclic row distribution. Four benchmark test problems were chosen from Table 6.1 from different sizes and coming from different applications: theta6, control11, thetaG51 and hamming_8_3_4. We run both the standard function computing M in PCSDP and the modified one. Since the routine for *Cholesky* is the same, we recorded only the running time for *Schur* and the total solution time, denoted by $Total$. For the one-dimensional computation case we need to execute also a redistribution procedure to achieve the desired two-dimensional form of M before its Cholesky factorization. For that purpose we use the standard routine *pdgemr2d* from ScaLAPACK library. It is set up to give exactly the same block structure as the standard PCSDP will have when computing M.

Table 6.6 contains the results for computing *Schur* (matrix M) for the selected test problems using from 2 to 64 processors. In the column 'grid type' 1D and 2D mean two-dimensional computation and one-dimensional computation, respectively.

We added to the time for computing elements of M using 1D approach also the time spent on data redistribution by the *pdgemr2d* routine. In this way we have a precise measure of the total cost computing the Schur complement matrix up to the beginning of the Cholesky factorization. Figure 6.1 depicts the computational

Problem	m	n_{max}	Grid type	CPUs					
				2	4	8	16	32	64
theta6	4375	300	$2D$	19.11	12.01	5.79	4.10	2.80	1.81
			$1D$	14.62	7.64	4.63	2.61	1.84	1.83
control11	1326	100	$2D$	234.16	125.00	71.03	53.09	30.57	21.66
			$1D$	197.01	99.76	54.60	25.52	13.43	8.16
thetaG51	6910	1001	$2D$	487.91	247.11	124.44	66.65	35.05	20.96
			$1D$	578.16	372.38	314.96	121.45	156.85	102.87
hamming_8_3_4	16129	256	$2D$	309.19	186.58	96.20	57.52	27.44	19.32
			$1D$	371.91	231.65	171.75	91.79	50.04	28.86

Table 6.6: *Computing the matrix M of selected SDO problems solved by PCSDP using 1D vs 2D process grid.*

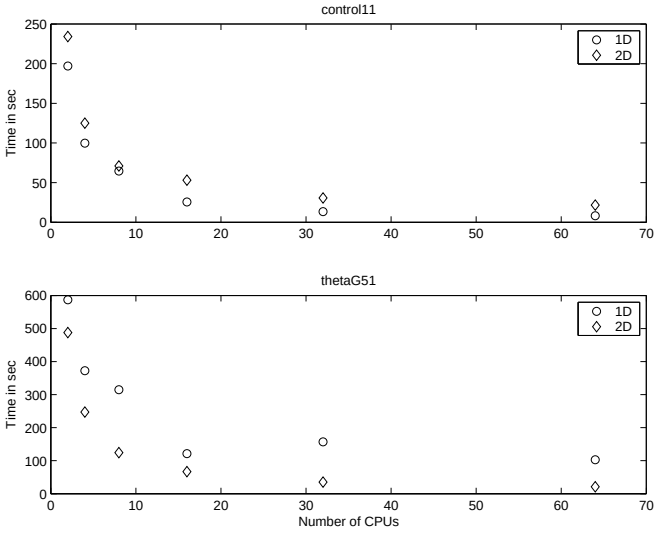

Figure 6.1: *Computing the matrix M of control11 and thetaG51 using 1D vs 2D process grid.*

time (in seconds) of M for the control11 and thetaG51 instances using 1D and 2D approach for 1, 2, 4, 8, 32 and 64 processors.

When the problem has a small size such as theta6 and control11, the one-dimensional computation has an advantage. When we have a medium or large sized SDO problem such as thetaG51 and hamming_8_3_4, the results show significant decrease in the computation times. Communication overhead for redistribution of M becomes significant in those cases and the 2D approach wins independently of the number of processors used.

Table 6.7 depicts the total running time for the four test problems using 2D and 1D approaches. The results follow the same trend as when computing M. For the small-sized problems theta6 and control11 it would take longer with the standard PCSDP to obtain the final result, than with the use of 1D function. In the case of instances thetaG51 and hamming_8_3_4, the 2D approach gives better results on a computer cluster with lower speed interconnect, such as the 1GBit ethernet in DAS3.

Problem	m	n_{max}	Grid type	CPUs					
				2	4	8	16	32	64
theta6	4375	300	$2D$	376.71	202.21	122.93	83.68	75.75	49.94
			$1D$	360.3	195.62	114.96	79.45	70.43	45.93
control11	1326	100	$2D$	310.55	191.96	129.84	106.94	83.34	75.30
			$1D$	127.26	166.18	119.2	76.59	69.71	58.93
thetaG51	6910	1001	$2D$	5368.77	3633.84	2498.57	2041.32	1943.69	1715.75
			$1D$	5392.17	3704.06	2603.43	2118.79	2107.29	1768.70
hamming_8_3_4	16129	256	$2D$	16526.56	10719.20	5391.22	2462.25	1433.83	685.14
			$1D$	16656.02	10898.30	5443.07	2598.05	1498.83	692.59

Table 6.7: *Total running time of selected SDO problems solved by PCSDP using 1D vs 2D process grid.*

One option in PCSDP that we didn't mention so far is the possibility to reconfigure the shape of the two-dimensional process grid, displayed on Fig 5.1. The number of rows N_r and the number of columns N_c could be changed. Unfortunately, changing the shape make the performance of ScaLAPACK routines worse, i.e., the time to compute *Cholesky* increases. This implies that such an option is highly undesirable when Cholesky factorization of M is the most time consuming operation. For the cases where *Schur* takes the biggest portion and *Cholesky* a very small part of the total running time, such an option might be worth trying.

6.2 Problems with rank-one structure

In Chapter 5 we described how PCSDP can exploit rank-one constraint matrices. For certain classes of SDO problems such as combinatorial optimization relaxations (for example max-cut [59]) and optimization of polynomials using SDO [75], the resulting semidefinite optimization problems have rank-one structure. Computation of the matrix M may be simplified (see Section 5.2.3) if the rank-one structure is exploited.

In this section we test the efficiency of PCSDP when dealing with rank-one A_i's, and compare the results with the standard sequential solvers CSDP5.0 and DSDP. For this purpose we modified PCSDP to be able to run as a sequential

solver and to deal explicitly with rank-one constraints. From now on we will call this sequential version of our solver as PCSDPr. The numerical experiments with these three solvers were executed on a PC with Intel x86 P4 (3.0GHz) CPU and 3GB of memory running Linux operating system. Optimized BLAS and LAPACK libraries were used. All of the software was compiled with gcc 3.4.5 and the default values of all parameters were used. The input data format used is the standard SDPA sparse (dat-s) format for CSDP5.0 and our modified[4] SDPA sparse format

Problem	Function f	$[a,b]$	$\max_{x\in[a,b]} f(x)$	Global maximizer(s)
test1	$-\frac{1}{6}x6 + \frac{52}{25}x5 - \frac{39}{80}x4$ $-\frac{71}{10}x3 + \frac{79}{20}x2 + x - \frac{1}{10}$	[-1.5,11]	29,763.233	10
test2	$-\sin x - \sin\frac{10}{3}x$	[2.7,7.5]	1.899599	5.145735
test3	$\sum_{k=1} 5k\sin((k+1)x+k)$	[-10,10]	12.03124	-6.7745761 -0.491391 5.791785
test4	$(16x2 - 24x + 5)e^{-x}$	[1.9,3.9]	3.85045	2.868034
test5	$(-3x + 1.4)\sin 18x$	[0,1.2]	1.48907	0.96609
test6	$(x + \sin x)e^{-x2}$	[-10,10]	0.824239	0.67956
test7	$-\sin x - \sin\frac{10}{3}x$ $-\ln x + 0.84x - 3$	[2.7,7.5]	1.6013	5.19978
test8	$\sum_{k=1} 5k\cos((k+1)x+k)$	[-10,10]	14.508	- 7.083506 -0.800321 5.48286
test9	$-\sin x - \sin\frac{2}{3}x$	[3.1,20.4]	1.90596	17.039
test10	$x\sin x$	[0,10]	7.91673	7.9787
test11	$2\cos x + \cos 2x$	[-1.57,6.28]	1.5	2.09439 4.18879
test12	$-\sin 3x - \cos 3x$	[0,6.28]	1	π 4.712389
test13	$x^{2/3} + (1-x2)^{1/3}$	[0.001,0.99]	1.5874	$1/\sqrt{2}$
test14	$e^{-x}\sin 2\pi x$	[0,4]	0.788685	0.224885
test15	$(-x2 + 5x - 6)/(x2 + 1)$	[-5,5]	0.03553	2.41422
test16	$-2(x-3)2 - e^{-x2/2}$	[-3,3]	-0.0111090	3
test17	$-x6 + 15x4 - 27x2 - 250$	[-4,4]	-7	-3 3
test18	$\begin{cases} -(x-2)2, & x \le 3; \\ -2\ln(x-2) - 1, & \text{otherwise.} \end{cases}$	[0,6]	0	2
test19	$\sin 3x - x - 1$	[0,6.5]	7.81567	5.87287
test20	$(x - \sin x)e^{-x2}$	[-10,10]	0.0634905	1.195137

Table 6.8: *Twenty test functions from P. Hansen et al. [58].*

for PCSDPr. In the case of DSDP all data was provided by using its MATLAB interface.

[4]We gave details about the modifications in Section 5.2.3.

6.2.1 Optimization of univariate functions on bounded interval by interpolation

The performance of both CSDP5.0 and PCSDPr was evaluated on all SDO instances displayed in Table 6.8, taken from [29].

These SDO problems approximate the minima of a set of twenty univariate test functions from Hansen et al. [58] on an interval, by first approximating the functions using Lagrange interpolation, and subsequently minimizing the Lagrange polynomials using SDO. These SDO problems only involve rank-one data matrices by using a formulation due to Löfberg and Parrilo [75].

The data in Tables 6.9 and 6.10 describes the results from our experiments with respect to the number of (Chebyshev) interpolation points for CSDP5.0 and PCSDPr, respectively. The sizes of the SDO problems depend on the number

Problem	Objective	Interpolation points						
		20	40	60	80	100	150	200
*test*1	$2.9763233e+04$	0.05	0.58	2.30	7.01	16.05	76.87	243.78
*test*2	$1.8995993e+00$	0.06	0.62	2.30	7.16	16.19	78.38	250.86
*test*3	$1.2031249e+01$	0.06	0.58	2.46	7.32	15.54	80.11	235.20
*test*4	$3.8504507e+00$	0.05	0.56	2.21	6.68	15.75	73.06	251.04
*test*5	$1.4890725e+00$	0.06	0.62	2.37	7.40	16.88	76.90	244.48
*test*6	$8.2423940e-01$	0.06	0.57	2.31	7.03	16.87	82.48	254.22
*test*7	$1.6013075e+00$	0.06	0.62	2.38	7.44	18.13	85.20	258.01
*test*8	$1.4508008e+01$	0.05	0.60	2.34	7.19	15.78	78.82	254.26
*test*9	$1.9059611e+00$	0.05	0.64	2.43	7.41	16.92	79.42	255.00
*test*10	$7.9167274e+00$	0.06	0.62	2.45	7.62	17.20	76.33	245.32
*test*11	$1.5000000e+00$	0.05	0.62	2.50	7.86	17.06	82.95	265.09
*test*12	$1.0000000e+00$	0.07	0.61	2.37	7.65	17.02	80.48	254.71
*test*13	$1.5874011e+00$	0.06	0.62	2.42	7.33	16.26	79.07	251.49
*test*14	$7.8868539e-01$	0.06	0.61	2.38	7.45	16.95	80.61	261.72
*test*15	$3.5533901e-02$	0.07	0.73	2.80	8.40	19.15	86.17	272.15
*test*16	$-1.110901e-02$	0.07	0.76	3.03	9.07	21.50	98.18	321.54
*test*17	$-7.0000000e+00$	0.06	0.76	2.88	8.84	20.71	101.68	300.42
*test*18	0	0.07	0.72	2.59	8.45	18.58	89.28	267.22
*test*19	$7.8156745e+00$	0.07	0.61	2.48	7.53	17.03	82.57	264.96
*test*20	$6.3490529e-02$	0.07	0.58	2.36	6.94	17.24	79.33	239.42

Table 6.9: *Solution times in seconds for CSDP5.0 for twenty rank-one test problems.*

of interpolation points as follows: the number of linear equality constraints m equals the number of interpolation points, and there are two positive semidefinite blocks of size roughly $m/2$. We also give the optimal objective value for each test function.

Problem	Objective	Interpolation points						
		20	40	60	80	100	150	200
*test*1	$2.9763233e+04$	0.08	0.40	1.36	3.44	7.59	31.58	92.90
*test*2	$1.8995993e+00$	0.06	0.42	1.38	3.46	7.51	31.96	98.28
*test*3	$1.2031249e+01$	0.07	0.39	1.37	3.57	7.14	30.47	98.08
*test*4	$3.8504507e+00$	0.06	0.39	1.28	3.36	7.25	31.22	88.95
*test*5	$1.4890725e+00$	0.07	0.43	1.36	3.59	7.60	31.18	94.78
*test*6	$8.2423940e-01$	0.07	0.40	1.38	3.47	7.63	33.74	93.25
*test*7	$1.6013075e+00$	0.07	0.42	1.37	3.64	7.63	34.62	96.39
*test*8	$1.4508008e+01$	0.06	0.39	1.34	3.66	7.14	30.48	92.77
*test*9	$1.9059611e+00$	0.07	0.42	1.35	3.55	7.50	32.36	93.95
*test*10	$7.9167274e+00$	0.07	0.40	1.33	3.47	7.71	32.38	92.15
*test*11	$1.5000000e+00$	0.07	0.40	1.34	3.65	7.49	34.43	97.44
*test*12	$1.0000000e+00$	0.06	0.39	1.25	3.64	7.15	31.96	99.44
*test*13	$1.5874011e+00$	0.06	0.38	1.26	3.51	7.35	30.93	92.60
*test*14	$7.8868539e-01$	0.06	0.39	1.31	3.52	7.33	33.03	95.05
*test*15	$3.5533901e-02$	0.07	0.45	1.51	4.08	8.54	35.10	102.84
*test*16	$-1.110901e-02$	0.08	0.54	1.69	4.64	9.65	41.63	112.57
*test*17	$-7.0000000e+00$	0.07	0.48	1.56	4.23	8.69	37.34	108.08
*test*18	0	0.08	0.49	1.65	4.04	8.72	35.01	98.89
*test*19	$7.8156745e+00$	0.06	0.39	1.35	3.61	7.53	32.02	93.05
*test*20	$6.3490529e-02$	0.06	0.38	1.28	3.53	7.66	30.88	94.87

Table 6.10: *Solution times in seconds for PCSDPr for twenty rank-one test problems.*

The solution times in Tables 6.9 and 6.10 show that when we have up to 40 interpolation points, the times are of the same order. This is to be expected due to the small size of the SDO problem. With increasing the number of interpolation points, the difference in solution times becomes apparent. When the order is 60, we notice roughly a 50% reduction in solution time when exploiting rank-one structure. This percentage increases with increase of the number of interpolation points. For order 100, PCSDPr obtains solution almost twice faster on average than the standard version of CSDP5.0. The difference in solution time is almost 3 times in favor of PCSDPr for 200 interpolation points. These results clearly indicate that the simple construction (5.19) for exploiting rank-one structure when computing *Schur* in a primal-dual IPM algorithm results in a very significant speedup in practice.

Next, we included in our tests the fastest known approach to solve rank-one problems, namely using the DSDP solver. We performed tests with DSDP using its Matlab 6.5 interface and Linux operating system on the same PC as the test of CSDP5.0 and PCSDPr. The test was only on the test functions test1 and test17 functions, as the computational times vary only slightly for the different instances.

Problem	Solver	Phase	Interpolation points						
			20	40	60	80	100	150	200
test1	CSDP5.0	Reading input	0.01	0.08	0.28	0.64	1.28	4.38	10.69
		Solver	0.05	0.58	2.30	7.01	16.05	76.87	243.78
		Total	0.06	0.66	2.58	7.65	17.33	81.45	254.47
	PCSDPr	Reading input	< 0.01	0.01	0.02	0.02	0.04	0.11	0.22
		Solver	0.08	0.40	1.36	3.44	7.59	31.58	92.90
		Total	0.08	0.41	1.38	3.46	7.63	31.69	93.12
	DSDP	Total	0.15	0.34	0.76	1.27	2.48	6.75	14.80
test17	CSDP5.0	Reading input	0.01	0.09	0.28	0.67	1.31	4.43	10.50
		Solver	0.06	0.76	2.88	8.84	20.71	101.68	300.42
		Total	0.07	0.85	3.16	9.51	22.02	106.11	310.92
	PCSDPr	Reading input	< 0.01	0.01	0.02	0.02	0.05	0.11	0.20
		Solver	0.07	0.48	1.56	4.23	8.69	37.34	108.08
		Total	0.07	0.49	1.58	4.25	8.74	37.45	108.28
	DSDP	Total	0.08	0.20	0.44	0.88	1.49	4.78	11.19

Table 6.11: *Running times in seconds for CSDP5.0, PCSDPr and DSDP for one rank-one test problem.*

The results are shown in Table 6.11. All times are in seconds and we were only interested in the total running time. We see that for order 20 and 40 we have a

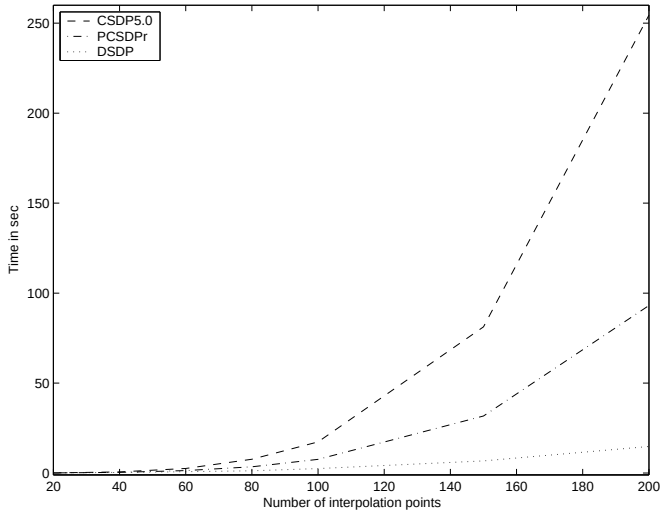

Figure 6.2: *Solution times (in seconds) for CSDP5.0, PCSDPr and DSDP for instance test1.*

running times with difference within a 2-3 tenths of a second for DSDP, CSDP5.0 and PCSDPr. When the order is 100 then DSDP is faster than PCSDPr by a factor of three. The difference is easy to observe on Figure 6.2 that depicts the total solution time (Total) in seconds for problem test1.

When the size of the SDO problem further increases, the gap between the two algorithms grows as one might expect. For 200 interpolation points, PCSDPr is slower than DSDP by a factor close to 10.

In summary, from all our numerical experiments so far exploiting rank-one structure in PCSDP still does not compete well with the dual scaling algorithm implemented in DSDP, but it does make the gap smaller than before.

Experimental results with PCSDP

The tests of the rank-one feature in sequential code PCSDPr above suggest a good speedup in practice, so we made it available in our parallel code PCSDP too. Next we present results from the numerical tests of our parallel implementation PCSDP on the DAS3 cluster. The aim of the experiments is to show what is the scalability of computing the matrix M in parallel when exploiting rank-one structure. The number of processors used is between 1 and 32. The reason we exclude the 64 CPU's case is that the problems in Table 6.8 are relatively small-scale. They are also of similar sizes and we therefore only consider only one problem: *test2*. In the PCSDP parameters file we introduced a variable (called *rank1*)[5] that can switch the rank-one feature of the solver 'on' and 'off'.

Exploiting rank-one structure of the constraints matrices will affect only computation of the Schur complement matrix M. Therefore we measure only the time of forming *Schur* and the total running time, marked as *Total*. We exclude from our tests cases with fewer than 80 interpolation points.

Interpolation points	Phase	CPUs					
		1	2	4	8	16	32
80	*Schur*	3.30	2.26	1.62	0.82	0.48	0.25
	Total	5.42	4.54	4.09	3.72	3.45	3.62
100	*Schur*	7.99	5.50	3.91	2.01	1.20	0.67
	Total	11.76	9.44	8.21	6.28	6.41	6.68
150	*Schur*	41.71	29.83	16.70	10.18	6.26	3.35
	Total	51.97	40.20	30.20	23.17	20.37	24.43
200	*Schur*	145.28	94.64	67.11	36.30	22.56	11.34
	Total	176.38	126.29	98.87	68.66	56.09	54.89

Table 6.12: *Running times for test2 problem (in seconds) for PCSDP with rank-one option 'off'.*

[5]See for more details Section 5.2.3.

Tables 6.12 and 6.13 contain the running times in seconds for *test2* when rank-one feature is 'off' and when is 'on', respectively. The number of interpolation points used for our tests is between 80 and 200. Figure 6.3 depicts the results for

Interpolation points	Phase	CPUs					
		1	2	4	8	16	32
80	*Schur*	1.27	0.61	0.35	0.21	0.16	0.09
	Total	3.19	3.08	2.65	2.47	2.07	2.16
100	*Schur*	3.71	2.29	1.53	0.75	0.40	0.19
	Total	6.48	5.81	5.16	4.70	4.57	4.90
150	*Schur*	15.77	10.93	5.77	3.81	2.00	0.98
	Total	28.95	21.94	17.10	15.28	13.67	13.63
200	*Schur*	55.50	34.81	21.62	13.00	6.63	3.30
	Total	81.44	61.82	48.73	41.75	35.11	32.68

Table 6.13: *Running times for test2 problem (in seconds) for PCSDP with rank-one option 'on'.*

both *Schur* and *Total* for 200 interpolation points with rank-one feature 'on' and 'off'. With increasing the number of processors, results for computing *Schur* using

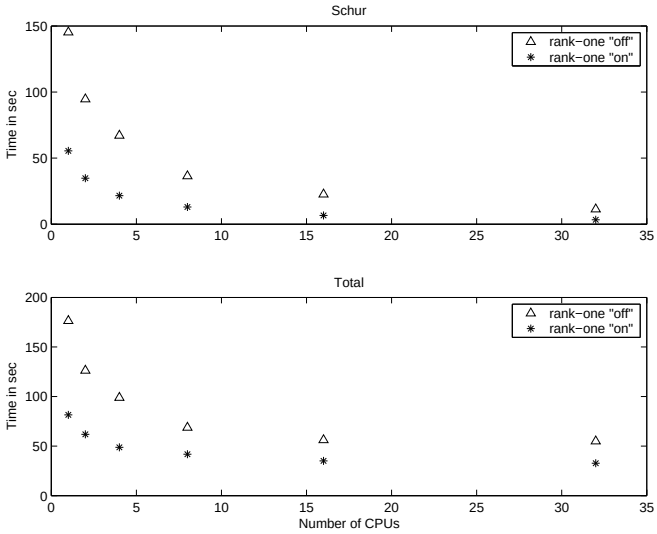

Figure 6.3: *Results for test2 (Schur and Total) when 200 interpolation points are used.*

the rank-one feature show similar speedup compared to the sequential case. The

times differ by a factor of at least 3 for all the range of CPUs used. Unfortunately, this is not the case for the total running time.

When the rank-one option is 'on' PCSDP runs roughly twice as fast when compared to the standard approach for 2 and 4 processors. Further increasing the number of processors the reduction is less than a factor 1.5. Compared to DSDP, even with the parallel implementation and rank-one option 'on' PCSDP does not outperform it. The reasons are that the total running time is affected by the additional communication time needed and factorization of a dense primal variable X. Due to the good speedup of *Schur*, it does not remain the dominant operation in $Total$ when the number of processors is above 4. The size of the problem also plays a role in this poor scalability of the total running time. All the test problems we had were of a small size and $m = 200$.

We investigate next how well computing of M scales with respect to the number of processors used, depending on the use of the rank-one option. The parallel efficiencies E were calculated for *Schur* from Tables 6.12 and 6.13 by expression (6.1). The results are presented in Table 6.14. For better illustrations of the

Interpolation points	rank-one feature	CPUs					
		1	2	4	8	16	32
80	off	1	0.73	0.51	0.50	0.42	0.41
	on	1	1.03	0.89	0.74	0.47	0.42
100	off	1	0.73	0.51	0.50	0.41	0.37
	on	1	0.81	0.61	0.62	0.58	0.58
150	off	1	0.70	0.62	0.51	0.42	0.39
	on	1	0.72	0.68	0.52	0.49	0.50
200	off	1	0.77	0.54	0.50	0.40	0.40
	on	1	0.80	0.64	0.53	0.52	0.52

Table 6.14: *Parallel efficiency of PCSDP computing the matrix M for test2 problem.*

results, we depict in Figure 6.4 the parallel efficiency of *Schur* for test2 when 200 interpolation points are used.

From the results it is easy to see that exploiting the rank-one structure improves not only the running time, but also the parallel efficiency of computing the elements of the matrix M.

6.3 Conclusion

This chapter presented a numerical test of our primal-dual interior-point SDO solver on a set of benchmark problems. Results suggests that it achieves a good overall parallel efficiency for medium and large-sized semidefinite problems. The best results were obtained for instances, where computation of the Schur complement matrix M and its Cholesky factorization are dominant operations.

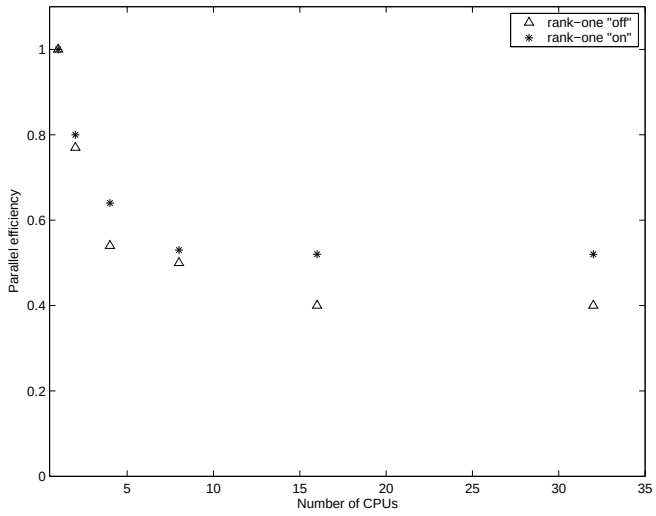

Figure 6.4: *Parallel efficiencies for computing M for 200 interpolation points for problem test2.*

PCSDP was primary designed for cluster computing, where only an ethernet network interconnect is available. The two-dimensional approach in computing the matrix M, implemented in the solver, proved to be better then the one-dimensional approach for this type of parallel computer architecture. Our approach improved the locality of the computations and decreased the communication overhead.

The use of the rank-one feature in our solver brings a significant computational benefit for the semidefinite problems that have rank-one structure in their constraint matrices. Exploiting this structure in PCSDP improves not only the running time in a primal-dual IPM solver, but also improves the parallel efficiency computing the elements of the Schur complement matrix M.

PCSDP, with its 64bit computing capability, makes it possible to solve large-scale SDO problems, that require a total amount of memory beyond the available RAM on any single PC in the cluster. In this way, our solver offers a cost-effective way to solve even larger future semidefinite optimization problems.

Chapter 7

Conclusions

This thesis addressed the natural limitations of primal-dual interior-point methods for SDO. Practical experience suggests [20, 114] that a significant amount of computational time in a primal-dual IPM for SDO is spent on computing and factorizing the typically dense Schur complement matrix. Reduction of the time spent on these two computations has a big impact on the total solution time for large scale problems. This motivated our work towards employing a parallel approach for these computations.

In Chapter 3 we give an overview of the different parallel computer architectures. There are two possibilities from the programming point of view: distributed or shared memory platform. As our favorite we nominated the distributed memory cluster architecture due to its low cost, its flexibility, and the fact that it available to many users, inside as well as outside the academic community.

Despite the effort and some solid developments, the use of parallelism in interior-point algorithms has not been a popular choice. Therefore, in Chapter 4 we performed a detailed evaluation of the parallel approaches in interior-point methods and their distributed implementations available so far. We identified the places where one could still achieve an improvement.

Results from our analysis moved us in the direction of developing a primal-dual interior point solver for semidefinite optimization, that we name PCSDP, designed for a computer cluster platform. It is based on the well-known sequential solver CSDP[18]. PCSDP requires only low cost computational hardware and network interconnects. Another advantage is that it offers also a 64bit computational arithmetic and is portable to different software UNIX-like platforms.

Our software incorporates a two-dimensional approach in distributed computation of the Schur complement matrix, which is different compared to all existing parallel SDO packages. It is of scientific value since it results in minimization of the total communication overhead for slow networks. PCSDP is capable of solving large scale SDO problems with memory requirements beyond the available RAM

of any computing node in the cluster. We offer also a feature not available in other parallel solvers for SDO, namely, to exploit rank-one structure in constraint matrices of the problem.

Results from the benchmark experiments proved that PCSDP enjoys a good overall parallel efficiency for medium and large-sized semidefinite problems. It is most efficient on problems that have a large number of constraints compared to the size of the primal and dual matrix variables. The best results were obtained for instances, where computation of the Schur complement matrix M and its Cholesky factorization are dominant operations. Its two-dimensional approach in computing the matrix M improved the locality of the computations, which is important on slow networks, such as ethernet. The use of the rank-one feature brought a significant computational benefit for problems that have rank-one structure in their constant matrices. Overall, PCSDP proved itself as a step forward in the development of parallel interior-point solvers for semidefinite optimization.

7.1 Directions for further research

One of the major performance limitations in parallel implementations of the primal-dual IPMs for semidefinite optimization is the inherently sequential nature of interior-point methods. Despite that, there is still room for improvement. One of the problems that still does not have a solution in the distributed implementations for SDO is the time-consuming Cholesky factorization of the primal and the dual matrix variables.

A possible answer may be on the way with the recent progress of processor technologies towards multi-core CPUs. Most parallel machines in the near future will be hybrids, combining nodes containing a modest number of commodity CPU's sharing memory in a distributed-memory cluster system. Such systems will open the horizon for a hybrid software too. For example, PCSDP can be modified into a distributed solver, which locally performs shared memory parallel computations on the multi-core nodes. Such local parallel computations can speed-up additionally all non-distributed parallel matrix operations including the primal and dual matrix factorizations.

Looking even further into the future of high-performance parallel computations, one could not miss the latest innovation in supercomputer systems: incorporating FPGA[1] technology. It provides massive algorithm acceleration through *hardware-based* implementation of compute-intensive algorithmic tasks. Modern FPGAs have the ability to be reprogrammed at 'run time', and this leads to the idea of reconfigurable computing (or systems)– CPUs that reconfigure themselves to suit the task at hand.

This high-performance computer technology is already implemented in the

[1] A field-programmable gate array (FPGA) is a semiconductor device that contains a programmable logic components called "logic blocks", and programmable interconnects.

Cray XT5h[2] supercomputer. By using FPGA-based acceleration, it can obtain $40 - 50$ times improvement over conventional processors for certain algorithms. Typical applications of FPGAs nowadays are digital signal processing, medical imaging, computer vision, speech recognition, cryptography, bioinformatics and a growing range of other areas. Semidefinite optimization, and particulary interior-point algorithms, might be a future application of this new high-performance hybrid computer technology.

[2]http://www.cray.com/products/xt5

Bibliography

[1] Infiniband Trade Association, web page `http://www.infinibandta.org/home`.

[2] F. Alizadeh. *Combinatorial optimization with interior point methods and semi-definite matrices*. PhD thesis, University of Minnesota, Minneapolis, MN, USA, 1992.

[3] F. Alizadeh, J.-P. A. Haeberly, and M. L. Overton. Primal-dual interior-point methods for semidefinite programming: Convergence rates, stability and numerical results. *SIAM Journal on Optimization*, 8(3):746–768, 1998.

[4] E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. Demmel, J. J. Dongarra, J. D. Croz, S. Hammarling, A. Greenbaum, A. McKenney, and D. Sorensen. *LAPACK Users' guide (third ed.)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1999.

[5] K. Anstreicher, N. Brixius, J.-P. Goux, and J. Linderoth. Solving large quadratic assignment problems on computational grids. *Mathematical Programming*, 91(3, Ser. B):563–588, 2002.

[6] K. M. Anstreicher. Recent advances in the solution of quadratic assignment problems. *Mathematical Programming*, 97(1-2, Ser. B):27–42, 2003.

[7] K. M. Anstreicher, J. Ji, F. A. Potra, and Y. Ye. Average performance of a self-dual interior point algorithm for linear programming. In *Complexity in numerical optimization*, pages 1–15. World Sci. Publ., River Edge, NJ, 1993.

[8] S. Balay, K. Buschelman, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, B. F. Smith, and H. Zhang. PETSc Web page, 2001. `http://www.mcs.anl.gov/petsc`.

[9] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith. Efficient management of parallelism in object-oriented numerical software libraries. In *Modern software tools for scientific computing*, pages 163–202. Birkhauser Boston Inc., Cambridge, MA, USA, 1997.

[10] S. J. Benson. Parallel computing on semidefinite programs. Technical Report ANL/MCS-P939-0302, Mathematics and Computer Science Division, Argonne National Laboratory, April 2003.

[11] S. J. Benson, Y. Ye, and X. Zhang. Solving large-scale sparse semidefinite programs for combinatorial optimization. *SIAM Journal on Optimization*, 10(2):443–461, 2000.

[12] F. Berman, G. Fox, and A. J. G. Hey. *Grid Computing: Making the Global Infrastructure a Reality*. John Wiley & Sons, Inc., New York, NY, USA, 2003.

[13] A. Bernstein. Analysis of programs for parallel processing. *IEEE Transactions on Electronic Computers*, EC-15(5):757–763, 1966.

[14] L. S. Blackford, J. Choi, A. Cleary, E. D'Azeuedo, J. Demmel, I. Dhillon, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK user's guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1997.

[15] L. S. Blackford, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, R. Pozo, K. Remington, and R. C. Whaley. An updated set of Basic Linear Algebra Subprograms (BLAS). *ACM Transactions on Mathematical Software*, 28(2):135–151, June 2002.

[16] L. Blum, F. Cucker, M. Shub, and S. Smale. *Complexity and real computation*. Springer-Verlag New York, Inc., 1998.

[17] L. Blum, M. Shub, and S. Smale. On a theory of computation and complexity over the real numbers: NP-completeness, recursive functions and universal machines. *Bulletin of the American Mathematical Society (New Series)*, 21(1):1–46, 1989.

[18] B. Borchers. CSDP, a C library for semidefinite programming. *Optimization Methods and Software*, 11/12(1-4):613–623, 1999.

[19] B. Borchers. SDPLIB 1.2, library of semidefinite programming test problems. *Optimization Methods and Software*, 11/12(1-4):683–690, 1999.

[20] B. Borchers and J. G. Young. Implementation of a primal-dual method for SDP on a shared memory parallel architecture. *Computational Optimization and Applications*, 37(3):355–369, 2007.

[21] R. E. Burkard, S. E. Karisch, and F. Rendl. QAPLIB—a quadratic assignment problem library. *Journal of Global Optimization*, 10(4):391–403, 1997; see also `http://www.seas.upenn.edu/qaplib/`.

[22] J. Choi, J. Demmel, I. Dhillon, J. Dongarra, S. Ostrouchov, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. ScaLAPACK: A portable linear algebra library for distributed memory computers – design issues and performance. Technical report, Knoxville, TN 37996, USA, 1995.

[23] J. Choi, J. J. Dongarra, L. S. Ostrouchov, A. P. Petitet, D. W. Walker, and R. C. Whaley. Design and implementation of the ScaLAPACK LU, QR, and Cholesky factorization routines. *Scientific Programming*, 5(3):173–184, 1996.

[24] T. Coleman, J. Czyzyk, C. Sun, M. Wagner, and S. Wright. pPCx: Parallel software for linear programming. In *Proceedings of the Eighth 30 SIAM Conference on Parallel Processing in Scientific Computing*. SIAM Publications, 1997.

[25] D. E. Culler, A. Gupta, and J. P. Singh. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[26] D. Cvetkovic, M. Cangalovic, and V. Kovacevic-Vujcic. Semidefinite programming methods for the symmetric traveling salesman problem. In *Proceedings of the 7th International IPCO Conference on Integer Programming and Combinatorial Optimization*, pages 126–136, Springer-Verlag, London, UK, 1999.

[27] E. de Klerk. *Aspects of Semidefinite Programming: Interior Point Algorithms and Selected Applications*, volume 65 of *Applied Optimization*. Kluwer Academic Publishers, March 2002.

[28] E. de Klerk. Private communication. 2008.

[29] E. de Klerk, G. Elabwabi, and D. den Hertog. Optimization of univariate functions on bounded intervals by interpolation and semidefinite programming. Discussion Paper 2006-26, Tilburg University, Center for Economic Research, April 2006; available at `http://ideas.repec.org/p/dgr/kubcen/200626.html`.

[30] E. de Klerk, J. Maharry, D. V. Pasechnik, R. B. Richter, and G. Salazar. Improved bounds for the crossing numbers of $K_{m,n}$ and K_n. *SIAM Journal on Discrete Mathematics*, 20(1):189–202, 2006.

[31] E. de Klerk and D. V. Pasechnik. Approximation of the stability number of a graph via copositive programming. *SIAM Journal on Optimization*, 12(4):875–892, 2002.

[32] E. de Klerk, D. V. Pasechnik, and A. Schrijver. Reduction of symmetric semidefinite programs using the regular $*$-representation. *Mathematical Programming*, 109(2-3, Ser. B):613–624, 2007.

[33] E. de Klerk, D. V. Pasechnik, and R. Sotirov. On semidefinite programming relaxations of the traveling salesman problem. Discussion Paper 2007-101, Tilburg University, Center for Economic Research, December 2007; available at `http://ideas.repec.org/p/dgr/kubcen/2007101.html`.

[34] E. de Klerk, C. Roos, and T. Terlaky. Initialization in semidefinite programming via a self-dual skew-symmetric embedding. *Operations Research Letters*, 20(5):213–221, 1997.

[35] E. de Klerk and R. Sotirov. Exploiting group symmetry in semidefinite programming relaxations of the quadratic assignment problem. Discussion Paper 2007-44, Tilburg University, Center for Economic Research, June 2007; available at `http://ideas.repec.org/p/dgr/kubcen/200744.html`.

[36] J. Dongarra, I. Foster, G. Fox, W. Gropp, K. Kennedy, L. Torczon, and A. White. *Sourcebook of parallel computing*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[37] J. J. Dongarra, L. S. Duff, D. C. Sorensen, and H. A. V. Vorst. *Numerical Linear Algebra for High Performance Computers*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1998.

[38] R. Duncan. A survey of parallel computer architectures. *Computer*, 23(2):5–16, 1990.

[39] A. V. Fiacco and G. P. McCormick. *Nonlinear Programming: Sequential Unconstrained Minimization Techniques*. John Wiley & Sons, New York, 1968. Reprint: Volume 4 of *SIAM Classics in Applied Mathematics*, SIAM Publications, Philadelphia, Pennsylvania, 1990.

[40] M. J. Flynn. Some computer organisations and their effectiviness. *IEEE Transactions on Computers*, C-21:948–960, September 1972.

[41] I. Foster and C. Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, 1997.

[42] I. Foster and C. Kesselman. *The Grid 2: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers Inc., San Francisco, USA, 2003.

[43] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the Grid: Enabling scalable virtual organization. *International Journal of High Performance Computing Applications*, 15(3):200–222, 2001.

[44] K. R. Frisch. The logarithmic potential method for convex programming. Memorandum, Institute of Economics, University of Oslo, Norway, May 1955.

[45] K. R. Frisch. The logarithmic potential method for solving linear programming problems. Memorandum, Institute of Economics, University of Oslo, Norway, 1955.

[46] K. Fujisawa, M. Kojima, and K. Nakata. Exploiting sparsity in primal–dual interior-point methods for semidefinite programming. *Mathematical Programming*, 79(1-3, Ser. B):235–253, 1997.

[47] M. Fukuda, M. Kojima, K. Mucro, and K. Nakata. Exploiting sparsity in semidefinite programming via matrix completion. I. General framework. *SIAM Journal on Optimization*, 11(3):647–674, 2000.

[48] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co., 2003.

[49] D. Geer. Industry trends: Chip makers turn to multicore processors. *Computer*, 38(5):11–13, 2005.

[50] A. George and J. W. Liu. The evolution of the minimum degree ordering algorithm. *SIAM Review*, 31(1):1–19, 1989.

[51] P. E. Gill, W. Murray, M. A. Saunders, J. A. Tomlin, and M. H. Wright. On projected Newton barrier methods for linear programming and an equivalence to Karmarkar's projective method. *Mathematical Programming*, 36(2):183–209, 1986.

[52] M. X. Goemans and D. P. Williamson. Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. *Journal of the Association for Computing Machinery*, 42(6):1115–1145, 1995.

[53] G. H. Golub and C. F. Van Loan. *Matrix computations (3rd ed.).* Johns Hopkins University Press, 1996.

[54] J. Gondzio and R. Sarkissian. Parallel interior-point solver for structured linear programs. *Mathematical Programming*, 96(3, Ser. A):561–584, 2003.

[55] A. Graham. *Kronecker products and matrix calculus: with applications.* Ellis Horwood Ltd., Chichester, 1981. Ellis Horwood Series in Mathematics and its Applications.

[56] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI (2nd ed.): portable parallel programming with the message-passing interface.* MIT Press, Cambridge, MA, USA, 1999.

[57] R. K. Guy. The decline and fall of Zarankiewicz's theorem. In *Proof Techniques in Graph Theory (Procceedings of the Second Annual Arbor Graph Theory Conference, Ann Arbor, Michigan, 1968)*, pages 63–69. Academic Press, New York, 1969.

[58] P. Hansen, B. Jaumard, and S.-H. Lu. Global optimization of univariate Lipschitz functions. II. New algorithms and computational comparison. *Mathematical Programming*, 55(3, Ser. A):273–292, 1992.

[59] C. Helmberg and F. Rendl. Solving quadratic $(0, 1)$-problems by semidefinite programs and cutting planes. *Mathematical Programming*, 82(3, Ser. A):291–315, 1998.

[60] C. Helmberg, F. Rendl, R. J. Vanderbei, and H. Wolkowicz. An interior-point method for semidefinite programming. *SIAM Journal on Optimization*, 6(2):342–361, 1996.

[61] W. D. Hillis and L. W. Tucker. The CM-5 Connection Machine: a scalable supercomputer. *Communications of the ACM*, 36(11):31–40, 1993.

[62] L. H. Jamieson, D. Gannon, and R. J. Douglass, editors. *The characteristics of parallel algorithms*. Massachusetts Institute of Technology, Cambridge, MA, USA, 1987.

[63] W. E. Johnston. Using computing and data grids for large-scale science and engineering. *International Journal of High Performance Computing Applications*, 15(3):223–242, 2001.

[64] M. T. Jones and P. E. Plassmann. An improved incomplete cholesky factorization. *ACM Transactions on Mathematical Software*, 21(1):5–17, 1995.

[65] M. Joshi, G. Karypis, V. Kumar, A. Gupta, and F. Gustavson. PSPASES: An efficient and scalable parallel sparse direct solver. In *Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing*. San Antonio, Texas, USA, 1999.

[66] N. Karmarkar. A new polynomial-time algorithm for linear programming. *Combinatorica*, 4(4):373–395, 1984.

[67] G. Karypis, A. Gupta, and V. Kumar. A parallel formulation of interior point algorithms. In *Supercomputing '94: Proceedings of the 1994 conference on Supercomputing*, pages 204–213. IEEE Computer Society Press, USA, 1994.

[68] L. G. Khaciyan. A polynomial algorithm in linear programming. *Doklady Akademiia Nauk SSSR*, 244:1093–1096, 1979. Translated into English in *Soviet Mathematics Doklady* 20, 191-194.

[69] V. Klee and G. J. Minty. "How good is the simplex algorithm?". In *Inequalities III*, pages 159–175. Academic Press, New York, USA, 1972.

[70] M. Kojima, S. Mizuno, and A. Yoshise. A primal-dual interior point algorithm for linear programming. In *Progress in mathematical programming*, pages 29–47. Springer, New York, 1989.

[71] M. Kojima, S. Shindoh, and S. Hara. Interior-point methods for the monotone semidefinite linear complementarity problem in symmetric matrices. *SIAM Journal on Optimization*, 7(1):86–125, 1997.

[72] M. Kočvara and M. Stingl. On the solution of large-scale SDP problems by the modified barrier method using iterative solvers. *Mathematical Programming*, 109(2):413–444, 2007.

[73] K. Krauter, R. Buyya, and M. Maheswaran. A taxonomy and survey of grid resource management systems for distributed computing. *Software – Practice and Experience*, 32(2):135–164, 2002.

[74] J. Kurzak and J. Dongarra. Implementing linear algebra routines on multi-core processors with pipelining and a look ahead. In *Applied Parallel Computing. State of the Art in Scientific Computing*, volume 4699 of *Lecture Notes in Computer Science*, pages 147–156. Springer, 2007.

[75] J. Löfberg and P. Parrilo. From coefficients to samples: a new approach to SOS optimization. In *IEEE Conference on Decision and Control*, December 2004.

[76] I. J. Lustig and G. Li. An implementation of a parallel primal–dual interior–point method for block–structured linear programs. *Computational Optimization and Applications*, 1(2):141–161, 1992.

[77] N. Meggido. Pathways to the optimal set in linear programming. In *Progress in Mathematical Programming Interior-point and related methods*, pages 131–158. Springer-Verlag Inc., New York, USA, 1988.

[78] S. Mehrotra. On the implementation of a primal-dual interior point method. *SIAM Journal on Optimization*, 2(4):575–601, 1992.

[79] V. Menon and K. Pingali. Look left, look right, look left again: an application of fractal symbolic analysis to linear algebra code restructuring. *International Journal of Parallel Programming*, 32(6):501–523, 2004.

[80] H. D. Mittelman. Dimacs challenge benchmarks, 2000; available at `http://plato.asu.edu/dimacs/`.

[81] R. D. C. Monteiro. Primal-dual path-following algorithms for semidefinite programming. *SIAM Journal on Optimization*, 7(3):663–678, 1997.

[82] R. D. C. Monteiro and I. Adler. Interior path following primal-dual algorithms. I. Linear programming. *Mathematical Programming*, 44(1, Ser. A):27–41, 1989.

[83] R. D. C. Monteiro and P. Zanjácomo. Implementation of primal-dual methods for semidefinite programming based on Monteiro and Tsuchiya Newton directions and their variants. *Optimization Methods and Software*, 11/12(1-4):91–140, 1999.

[84] K. Nakata, K. Fujisawa, M. Fukuda, M. Kojima, and K. Mucro. Exploiting sparsity in semidefinite programming via matrix completion. II. Implementation and numerical results. *Mathematical Programming*, 95(2, Ser. B):303–327, 2003.

[85] K. Nakata, M. Yamashita, K. Fujisawa, and M. Kojima. A parallel primal-dual interior-point method for semidefinite programs using positive definite matrix completion. *Parallel Computing*, 32(1):24–43, 2006.

[86] M. Nakata, H. Nakatsuji, M. Ehara, M. Fukuda, K. Nakata, and K. Fujisawa. Variational calculations of fermion second-order reduced density matrices by semidefinite programming algorithm. *Journal of Chemical Physics*, 114(19):8282–8292, 2001.

[87] Y. E. Nesterov and A. Nemirovskii. *Interior-point polynomial algorithms in convex programming*, volume 13 of *SIAM Studies in Applied Mathematics*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 1994.

[88] Y. E. Nesterov and M. J. Todd. Self-scaled barriers and interior-point methods for convex programming. *Mathematics of Operations Research*, 22(1):1–42, 1997.

[89] Y. E. Nesterov and M. J. Todd. Primal-dual interior-point methods for self-scaled cones. *SIAM Journal on Optimization*, 8(2):324–364, 1998.

[90] C. H. Papadimitriou and K. Steiglitz. *Combinatorial optimization: algorithms and complexity*. Prentice-Hall Inc., Upper Saddle River, NJ, USA, 1982.

[91] G. F. Pfister. *In search of clusters (2nd ed.)*. Prentice-Hall Inc., Upper Saddle River, NJ, USA, 1998.

[92] J. Renegar. A polynomial-time algorithm, based on Newton's method, for linear programming. *Mathematical Programming*, 40(1, Ser. A):59–93, 1988.

[93] R. T. Rockafellar. Lagrange multipliers and optimality. *SIAM Review*, 35(2):183–238, 1993.

[94] C. Roos, T. Terlaky, and J.-P. Vial. *Interior-point methods for linear optimization*. Springer, New York, 2006. Second edition of *Theory and algorithms for linear optimization*, Wiley, Chichester, 1997.

[95] O. Schenk and K. Gärtner. Solving unsymmetric sparse systems of linear equations with PARDISO. *Future Generation Computer Systems*, 20(3):475–487, 2004.

[96] C. L. Seitz. The cosmic cube. *Communications of the ACM*, 28(1):22–33, 1985.

[97] T. L. Sterling, J. Salmon, D. J. Becker, and D. F. Savarese. *How to build a Beowulf: a guide to the implementation and application of PC clusters*. MIT Press, Cambridge, MA, USA, 1999.

[98] T. L. Sterling, D. F. Savarese, D. J. Becker, J. E. Dorband, U. A. Ranawake, and C. V. Packer. BEOWULF: A parallel workstation for scientific computation. In *Proceedings of the 24th International Conference on Parallel Processing*, pages I:11–14, Oconomowoc, WI, 1995.

[99] J. F. Sturm. Using SeDuMi 1.02, a MATLAB toolbox for optimization over symmetric cones. *Optimization Methods and Software*, 11/12(1-4):625–653, 1999.

[100] J. F. Sturm. Implementation of interior–point methods for mixed semidefinite and second order cone optimization problems. *Optimization Methods and Software*, 17(6):1105–1154, 2002.

[101] J. F. Sturm and S. Zhang. Symmetric primal-dual path-following algorithms for semidefinite programming. *Applied Numerical Mathematics*, 29(3):301–315, 1999.

[102] C. Sun. Parallel sparse orthogonal factorization on distributed-memory multiprocessors. *SIAM Journal on Scientific Computing*, 17(3):666–685, 1996.

[103] M. J. Todd. A study of search directions in primal-dual interior-point methods for semidefinite programming. *Optimization Methods and Software*, 11/12(1-4):1–46, 1999.

[104] M. J. Todd. Semidefinite optimization. *Acta Numerica*, 10:515–560, 2001.

[105] M. J. Todd, K. C. Toh, and R. H. Tütüncü. On the Nesterov–Todd direction in semidefinite programming. *SIAM Journal on Optimization*, 8(3):769–796, 1998.

[106] K. C. Toh and M. Kojima. Solving some large-scale semidefinite programs via the conjugate residual method. *SIAM Journal on Optimization*, 12(3):669–691, 2002.

[107] K. C. Toh, M. J. Todd, and R. H. Tütüncü. SDPT3—a MATLAB software package for semidefinite programming, version 1.3. *Optimization Methods and Software*, 11/12(1-4):545–581, 1999.

[108] J. F. Traub and H. Woźniakowski. Complexity of linear programming. *Operations Research Letters*, 1(2):59–62, 1981/82.

[109] A. van der Steen. Overview of recent supercomputers. Technical report, 2007; available at `http://www.euroben.nl/reports/`.

[110] L. Vandenberghe and S. Boyd. Semidefinite programming. *SIAM Review*, 38(1):49–95, 1996.

[111] S. A. Vavasis. *Nonlinear optimization*, volume 8 of *International Series of Monographs on Computer Science*. Oxford University Press, New York, 1991.

[112] S. J. Wright. *Primal-dual interior-point methods*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, USA, 1997.

[113] M. Yamashita, K. Fujisawa, and M. Kojima. Implementation and evaluation of SDPA 6.0 (SemiDefinite Programming Algorithm 6.0). *Optimization Methods and Software*, 18(4):491–505, 2003.

[114] M. Yamashita, K. Fujisawa, and M. Kojima. SDPARA: SemiDefinite Programming Algorithm paRAllel version. *Parallel Computing. Theory and Applications*, 29(8):1053–1067, 2003.

[115] M. Yannakakis. Computing the minimum fill-in is NP-complete. *SIAM Journal on Algebraic and Discrete Methods*, 2(1):77–79, 1981.

[116] K. Zarankiewicz. On a problem of P. Turan concerning graphs. *Polska Akademia Nauk. Fundamenta Mathematicae*, 41:137–145, 1954.

[117] Y. Zhang. On extending some primal–dual interior-point algorithms from linear programming to semidefinite programming. *SIAM Journal on Optimization*, 8(2):365–386, 1998.

[118] Q. Zhao, S. E. Karisch, F. Rendl, and H. Wolkowicz. Semidefinite programming relaxations for the quadratic assignment problem. *Journal of Combinatorial Optimization*, 2(1):71–109, 1998.

[119] Z. Zhao, B. J. Braams, M. Fukuda, M. L. Overton, and J. K. Percus. The reduced density matrix method for electronic structure calculations and the role of three-index representability conditions. *Journal of Chemical Physics*, 120(5):2095–2104, 2004.

Index

Summary

This thesis deals with solving large-scale semidefinite optimization (SDO) problems by using interior-point methods and parallel computations.

Firstly, we revisit the topic of the computational complexity of the primal-dual interior-point method (IPM) for SDO. Our main focus is on the time per iteration complexity for the three most popular search directions in semidefinite optimization solvers: the Alizadeh–Haeberly–Overton, the Helmberg–Kojima–Monteiro and the Nesterov–Todd directions. We outline the most computationally intensive operations involved in a path-following primal-dual interior-point algorithm using the Mehrotra predictor-corrector strategy. Practical experience suggested that a significant computational time is spend on computing and factorizing the Schur complement matrix M. Speeding up these two computations has a big impact on the total solution time for large-scale problems. This motivated our objective in this thesis, namely to parallelize these computations.

As a next step, we review the different parallel computer architectures from the programming point of view. We also discuss the suitability of the parallel systems for practical implementation of interior-point methods. Finally, we motivate our choice of a computer cluster architecture for development of a new distributed parallel primal-dual IPM solver for semidefinite optimization.

In order to choose a suitable design for our solver, we review the related work and problems in parallelization of interior-point methods for linear and semidefinite optimization. We focus on the different strategies to identify and use parallel computation to speed-up interior-point algorithms, in particular primal-dual methods for SDO. We give a discussion on the computational overhead caused by performing tasks in parallel.

In Chapter 5, we present the algorithmic framework behind our newly developed primal-dual interior-point method solver for SDO using the Helmberg–Kojima–Monteiro search direction. We introduce there a different parallel computational approach for computing the Schur complement matrix than the existing IPM solvers. The two-dimensional data distribution used in our solver aims to minimize the communication overhead caused by the parallel computations.

This approach improves the locality of the data and requires less overall memory storage. This is an important result for problems with very large number of constraints.

We also describe the new feature in our software that allows to deal with rank-one constraint matrices in SDO problems.

Chapter 6 presents the computational results from the numerical tests of our software PCSDP. We compare the running times and the scalability between our IPM solver and other parallel solvers for SDO, namely SDPARA and PDSDP. Results suggests that PCSDP achieves a good overall parallel efficiency for middle and large sized semidefinite problems. The best results were obtained for instances, where computation of the Schur complement matrix and its Cholesky factorization are dominant operations.

Results from the test of the rank-one feature in our solver are presented too. Exploiting rank-one structure resulted in a significant computational benefit for the semidefinite problems that have such structure in their constraint matrices.

PCSDP, with its 64bit computing capability, makes it possible to solve large-scale SDO problems, that require a total amount of memory beyond the available RAM on any single PC in the cluster. In this way, our solver offers a cost-effective way to solve even larger future SDO problems.

Samenvatting

Dit proefschrift gaat over het oplossen van grootschalige semidefiniete optimaliserings (SDO) problemen met inwendige punt methoden en het gebruik daarbij van parallelle berekeningen.

Om te beginnen staan we stil bij de rekenkundige complexiteit van primaal-duale inwendige punt methoden (IPM) voor SDO. We richten ons daarbij vooral op de rekentijd die per iteratie nodig is voor de drie meest populaire zoekrichtingen in SDO pakketten, te weten: de Alizadeh–Haeberly–Overton, de Helmberg–Kojima–Monteiro en de Nesterov–Todd zoekrichting. Wij schetsen ook de meest tijdvergende operaties voor padvolgende inwendige punt methoden die de predictor-corrector strategie van Mehrotra gebruiken.

Ervaringen in de praktijk doen vermoeden dat een significant deel van de rekentijd nodig is voor het berekenenen en factoriseren van de Schur complement matrix M. Het versnellen van de laatstgenoemde berekeningen heeft een grote invloed op de totale rekentijd voor het oplossen van grootschalige problemen. Dit verklaart het doel van dit proefschrift, namelijk om deze berekeningen te parallelliseren.

Vervolgens geven we een overzicht van de verschillende parallelle computer-architecturen met het oog op programmeringsaspecten. We bespreken ook de geschiktheid van deze architecturen voor de practische implementatie van inwendige punt methoden. Tenslotte motiveren we de gemaakte keus van een computer cluster architectuur voor het ontwikkelen van een nieuwe gedistribueerde parallelle primaal-duale IPM voor het oplossen van semidefiniete optimaliseringsproblemen.

Teneinde te komen tot een geschikt en verantwoord ontwerp voor onze code bespreken we gerelateerd bestaand werk, en de problemen die zich voordoen bij het parallelliseren van inwendige punt methoden, zowel voor lineaire en semidefiniete optimalisering. We focusseren daarbij op de verschillende strategieën om parallelle berekeningen te gebruiken voor het versnellen van inwendige punt algoritmen, in het bijzonder primaal-duale methoden voor semidefiniete optimalisering. We bespreken ook de onvermijdelijke rekenkundige overhead die optreedt bij het parallel

uitvoeren van deelberekeningen.

In Hoofdstuk 5 presenteren we het algoritmisch geraamte van de nieuw ontworpen primaal-duale inwendige punt methode voor SDO, gebaseerd op de Helmberg–Kojima–Monteiro zoekrichting. We introduceren een nieuwe benadering voor het parallelliseren van de berekening van de Schur complement matrix, die verschilt van de aanpak in bestaande SDO pakketten. De 2-dimensionale data distributie in onze code is bedoeld om de communicatie overhead te minimaliseren. Deze benadering verbetert het lokaal beschikbaar zijn van benodigde data, en vereist minder geheugen. Dit is van groot belang, met name voor voor problemen met een groot aantal beperkingen.

Onze software (PCSDP genoemd) heeft een andere opvallende eigenschap, namelijk dat als constraint matrix diadisch is (i.e., rang 1 heeft), dit in de berekeningen wordt benut.

Hoofdstuk 6 bevat testresultaten voor PCDSP. We vergelijken de rekentijden en de schaalbaarheid van PCDSP met die van twee bekende andere pakketten voor SDO, namelijk SDPARA en PDSDP. De resultaten laten zien dat PCSDP over het geheel genomen goed presteert voor grootschalige en minder grootschalige problemen. De beste resultaten worden verkregen voor problemen waarvoor de de rekentijd voor de berekening van de Schur complement matrix en de Cholesky factorisatie relatief groot is en die van de andere rekentaken domineert.

Wij testen ook de eigenschap van PCSDP om het optreden van diadische constraint matrices te exploreren. Dit blijkt te leiden tot een significante besparing in rekentijd.

PCSDP, met zijn 64-bit rekencapaciteit, maakt het mogelijk om grootschalige SDO problemen op te lossen, problemen die niet kunnen worden opgelost op een enkele PC omdat het vereiste RAM-geheugen dat van een gewone PC ver overschrijdt. PCSDP biedt zodoende een doeltreffend middel om in de toekomst grote SDO problemen op te lossen.

Curriculum Vitae

Ivan Ivanov was born in Silistra, Bulgaria on January 18, 1977. He graduated in 1996 the Vocational High School of Mechanics "Vladimir Komarov" - Silistra. The same year, he was enrolled in the Faculty of Automatics in the Technical University of Sofia, Bulgaria. He received his MSc degree in Automatics and Control Engineering in 2001.

Between January and August 2003, he did an internship in the "Group of Mixed-Signal Circuits and Systems" at Philips Research Laboratories (NatLab) in Eindhoven, The Netherlands. In September 2003 he became a researcher in the Optimization group at Delft University of Technology, with supervisor Prof. Cornelis Roos and scientific advisor Dr. Etienne de Klerk. His research in the area of interior-point algorithms for semidefinite optimization has led to the present PhD thesis.

During his stay in TU Delft, he completed the course program for PhD's of the Dutch Network on the Mathematics of Operations Research (LNMB). In April 2008, he joined Emerson Process Management in Rijswijk.

His research interests include: Interior-point methods, Linear and Semidefinite optimization, parallel computational algorithms, high-performance optimization.