Parallel implementation of a semidefinite programming solver based on CSDP on a distributed memory cluster

I.D. Ivanov* E. de Klerk†

March 5, 2008

Abstract

In this paper we present the algorithmic framework and practical aspects of implementing a parallel version of a primal-dual semidefinite programming solver on a distributed memory computer cluster. Our implementation is based on the CSDP solver and uses a message passing interface (MPI), and the ScaLAPACK library. A new feature is implemented to deal with problems that have rank-one constraint matrices. We show that significant improvement is obtained for a test set of problems with rank one constraint matrices. Moreover, we show that very good parallel efficiency is obtained for large-scale problems where the number of linear equality constraints is very large compared to the block sizes of the positive semidefinite matrix variables.

Keywords: Semidefinite programming, interior point methods, parallel computing, distributed memory cluster

AMS classification: 90C22, 90C51

1 Introduction

Semidefinite programming (SDP) has been a very popular area in mathematical programming since the early 1990's. Applications include LMI's in control theory, the Lovász ϑ-function in combinatorial optimization, robust optimization, approximation of maximum cuts in graphs, satisfiability problems, polynomial optimization by approximation, electronic structure computations in quantum chemistry, and many more (see *e.g.* [15, 6]).

There are several interior-point SDP solvers available such as SeDuMi [25] , CSDP [3], SDPT3 [27], SDPA [29], and DSDP5 [2]. Unfortunately, it is well-known that sparsity in SDP data cannot be exploited as efficiently as in the linear programming (LP) case, and the sizes of problems that can be solved in practice are modest when compared to LP.

*Faculty of Electrical Engineering, Mathematic and Computer Sciences, Delft University of Technology, P.O. Box 5031, 2600 GA Delft, The Netherlands. I.D.Ivanov@tudelft.nl

†Department of Econometrics and OR, Tilburg University, Tilburg, the Netherlands. E.deKlerk@uvt.nl

As a result, parallel versions of several SDP software packages have been developed like PDSDP [1], SDPARA [30] and a parallel version of CSDP [5]. The first two are designed for PC clusters using MPI[1] and ScaLAPACK[2] and the last one is designed for a shared memory computer architecture [5]. PDSDP is a parallel version of the DSDP5 solver developed by Benson, Ye, and Zhang [2] 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 the H..K..M. search direction, as does CSDP.

In this paper we present a open source version of CSDP for a Beowulf[3] (distributed) computer cluster. It makes use of MPI, and of the ScaLAPACK library for parallel algebra computations. The new software is meant to complement the parallel version of CSDP [5] that is already available for shared memory machines. A beta version of our software is freely available at

`http://lyrawww.uvt.nl/~edeklerk/PCSDP/`

Our implementation also has a new built-in capability to deal efficiently with rank-one constraint matrices. As a result, significant speedup is achieved when computing the Schur complement matrix 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. The only option available so far is to use SDPT3 (no parallel implementation available (yet)) or the dual scaling algorithm implemented in PDSDP.

Outline

This paper is organized as follows: in the next section we motivate our choice of computer architecture for our implementation. In Section 3 we discuss the algorithmic framework behind the solver. In particular, we give details about how a rank-one structure of data matrices is exploited. Section 4 describes details about the parallel part of the solver. Numerical test results for our code on benchmark problems are presented in Section 5. In Section 6 we compare our implementation to the parallel solvers SDPARA [30] and PDSDP [1] that use the same computer architecture.

2 Choice of computer architecture

There are two main classifications 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.

[1]`http://www-unix.mcs.anl.gov/mpi/`
[2]`http://www.netlib.org/scalapack/`
[3]`http://www.beowulf.org/`

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 is the main difficulty with the Beowulf cluster. To run efficiently, problems have to be partitioned so that the pieces will run efficiently in 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 of at most one gigabyte, then the problem could run inefficiently. This issue with the dynamic load balancing is not a problem for shared memory computers.

The attraction to use Beowulf clusters lies in the low cost of both hardware and software, and in the control that builders and 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 the choice between 'shared or distributed' depends on the problem one is trying to solve. In our case the parts of the SDP algorithm suitable for parallel computation are computing the Schur complement matrix and its Cholesky factorization.

First, composing the Schur complement matrix 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 routines employed for Cholesky factorization are based on block-partitioned algorithms in order to minimize the frequency of data movement between different levels of the memory hierarchy.

3 Algorithmic Framework

We recall in this section the predictor-corrector primal-dual interior point algorithm implemented in the original code of CSDP [3]. Moreover, we review how low rank of data matrices may be exploited.

3.1 The semidefinite programming problem

Consider the semidefinite programming (SDP) problem formulation in primal form

$$\begin{aligned} \text{(P)} \qquad \max_{X} \qquad & \operatorname{trace}(CX) \\ \text{s.t.} \qquad & A(X) = b, \\ & X \succeq 0 \end{aligned} \tag{3.1}$$

where $X \succeq 0$ (or $X \in \mathcal{S}_n^+$) means that X is a symmetric positive semidefinite $n \times n$ matrix. We assume a representation of the linear operator A as

$$A(X) = \begin{bmatrix} \operatorname{trace}(A_1 X) \\ \operatorname{trace}(A_2 X) \\ \vdots \\ \operatorname{trace}(A_m X) \end{bmatrix}, \tag{3.2}$$

where the matrices $A_i \in \mathcal{S}_n$ (*i.e.* are symmetric $n \times n$) for $i = 1, \ldots, m$, 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 (P) is given by

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

where

$$A^*(y) = \sum_{i=1}^{m} y_i A_i,$$

is the adjoint of A with respect to the usual trace inner product. A primal X and dual (y, Z) are interior feasible solutions of (P) and (D), respectively, if they satisfy the constraints of (P) and (D) as well as $X \succ 0$ and $Z \succ 0$. (We use the notation $X \succ 0$ for $X \in \mathcal{S}_n$ to be positive definite.) The idea behind primal-dual IPM's is to 'follow' the central path

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

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

$$\begin{aligned} b - A(X) &= 0, \\ Z + C - A^*(y) &= 0, \\ ZX - \mu I &= 0, \\ Z, X &\succ 0, \end{aligned} \tag{3.3}$$

where I denotes the identity matrix of size $n \times n$, and $\mu > 0$ is a given value called the *barrier parameter.*

It is well-known that for each $\mu > 0$ (3.3) has an unique solution $(X(\mu), y(\mu), Z(\mu))$ assuming that there exists a interior feasible solution (X, y, Z) of the SDP and constraint matrices $A_i \in \mathcal{S}_n$ for $i = 1, \ldots, m$ are linearly independent. Moreover, in the limit as μ goes to 0, $(X(\mu), y(\mu), Z(\mu))$ converges to an optimal solution of the SDP.

3.2 The H..K..M search direction

The algorithm used in CSDP [3] 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 SDP solvers as SeDuMi [25] is to use a *self-dual embedding* [8] technique to obtain a feasible starting point on the central path. In our implementation we use the default CSDP starting point (a similar approach is used in [27]); see [3] for details. This initial point, say (X, y, Z), does not satisfy $A(X) = b$ or $A^*(y) - Z = C$ in general, so we define

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

The algorithm implemented in CSDP uses a predictor-corrector strategy. The predictor step is computed from:

$$\begin{aligned} -A(\Delta\widehat{X}) &= -R_p, \\ \Delta\widehat{Z} + C - A^*(\Delta\widehat{y}) &= -R_d, \\ Z\Delta\widehat{X} - \Delta\widehat{Z}X &= -ZX. \end{aligned} \tag{3.4}$$

Using the same approach as in [15], we can reduce the system of equations (3.4) to

$$A(Z^{-1}A^*(\Delta\widehat{y})X) = -b + A(Z^{-1}R_dX), \tag{3.5}$$
$$\Delta\widehat{Z} = A^*(\Delta\widehat{y}) - R_d, \tag{3.6}$$
$$\Delta\widehat{X} = -X + Z^{-1}R_dX - Z^{-1}A^*(\Delta\widehat{y})X. \tag{3.7}$$

If we introduce the notation

$$\begin{aligned} M &:= [A(Z^{-1}A_1X), A(Z^{-1}A_2X), \ldots, A(Z^{-1}A_mX)], \\ v &:= -b + A(Z^{-1}R_dX), \end{aligned} \tag{3.8}$$

then we can write (3.5) in matrix form as follows:

$$M\Delta\widehat{y} = v. \tag{3.9}$$

As Helmberg, Rendl, Vanderbei and Wolkowicz have shown, the Schur complement matrix M is symmetric and positive definite [15]. Thus we can compute the Cholesky factorization of M to solve the system of equations (3.9). By back substitution in (3.6) and (3.7) we can subsequently compute $\Delta\widehat{Z}$ and $\Delta\widehat{X}$ respectively. Note that in this case $\Delta\widehat{X}$ is not necessarily symmetric. In order to keep $X + \Delta\widehat{X}$ symmetric, we replace $\Delta\widehat{X}$ by $\frac{\Delta\widehat{X}+\Delta\widehat{X}^T}{2}$.

For the corrector step we solve the linear system

$$\begin{aligned} -A(\Delta\overline{X}) &= 0, \\ \Delta\overline{Z} + C - A^*(\Delta\overline{y}) &= 0, \\ Z\Delta\overline{X} + \Delta\overline{Z}X &= \mu I - \Delta\widehat{Z}\Delta\widehat{X}, \end{aligned} \tag{3.10}$$

where $\mu = \text{trace}\frac{XZ}{2n}$.

These equations have the same form as (3.4) and are solved similarly as before to obtain $(\Delta\overline{X}, \Delta\overline{y}, \Delta\overline{Z})$. Next we add the predictor and corrector step to compute the search directions:

$$\begin{aligned} \triangle X &= \Delta\widehat{X} + \Delta\overline{X}, \\ \triangle y &= \Delta\widehat{y} + \Delta\overline{y}, \\ \triangle Z &= \Delta\widehat{Z} + \Delta\overline{Z}. \end{aligned} \tag{3.11}$$

Next, we find the maximum step lengths α_P and α_D 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.

In practice, the Schur complement matrix M may become numerically singular even though X and Z are numerically nonsingular. In this case, CSDP returns to the previous solution, and executes a centering step with $\mu = \text{trace}\frac{XZ}{n}$.

The solution of the linear system (3.9) 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 $\mathcal{O}(mn^3 + m^2n^2)$ [20]. 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 [10]. For sparse A_i's with $\mathcal{O}(1)$ entries, the matrix $Z^{-1}A_iX$ can be computed in $\mathcal{O}(n^2)$ operations $(i = 1, \ldots, m)$. Additionally $\mathcal{O}(m^2)$ time is required for $A(\cdot)$ operations. Finally, the Schur complement matrix M is typically fully dense and its Cholesky factorization requires $\mathcal{O}(m^3)$ operations.

There are results on exploiting aggregate sparsity patterns of data matrices via matrix completion [11]. Results from the practical implementation of this approach (see [22]) show that it is efficient only on a very sparse SDP's where the aggregate sparsity pattern of the constraint matrices induces a sparse chordal graph. If this is not the case, a general SDP solver will have better performance. Therefore, we didn't consider exploiting this structure in our implementation since our aim was not a problem specific solver.

The overall computational complexity of presented primal-dual IPM is dominated by different operations depending on the particular structure of the SDP problem. For problems with $m \gg n$ more CPU time is spent in computation of the $m \times m$ Schur complement matrix M, and computation of the solution $\Delta\widehat{y}$ of (3.9). When the constraint matrices are very sparse, factoring the $m \times m$ matrix M becomes the dominant operation. From now on we will refer to the Cholesky factorization procedure as *Cholesky*. In case of $m \not\gg n$ and a small number of big diagonal blocks, the matrix operations on X and Z can be the dominant ones. In general the primal variable X is fully dense even if all the constraint matrices A_i $(i = 1, ..., m)$ are sparse. On the other hand, the dual matrix variable Z computed by

$$Z = \sum_{i=1}^{m} y_i A_i - C,$$

inherits the aggregate sparsity pattern of the constraint matrices A_i $(i = 1, ..., m)$ and C. Thus, although Z is typically also dense, the are several applications where it is sparse, e.g. for the Goemans-Williamson [12] SDP relaxation of the maximum cut problem on sparse graphs.

For such problmes, primal-dual IPM's are at a disadvantage when compared to dual interior-point methods which generate iterates only in the dual space. Despite lower computational cost per iteration, dual IPM's do not posses super-linear convergence rates and typically attain less accuracy in practice [30] than primal-dual methods.

When we have dense constraints, construction of M, called *Schur* from now on, can be the most time consuming computation in each iteration when $m \gg n$. To reduce the computational time of our code takes advantage of the approach by Fujisawa, Kojima, and Nakata [10]. As test results in [5, 29] suggest, the most computational time in general large scale problems using primal-dual IPM algorithms is occupied by constructing *Schur* and solving the linear system which involves *Cholesky*. This motivated our work toward employing distributed parallel computation for these computations.

3.3 Exploiting rank-one structure of constraint matrices

In this subsection one additional assumption will be made: 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. Our aim is to use this special structure of A_i's to speed up computation of *Schur* when using a primal-dual IPM. The approach we use is basically one introduced by Helmberg and Rendl [14]. Recall from (3.8) that

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

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

$$M_{ij} = ({a_i}^T Z^{-1} a_j)({a_i}^T X a_j) \qquad (i, j = 1, \ldots, m). \tag{3.12}$$

Precomputing ${a_i}^T Z^{-1}$ and ${a_i}^T X$ for each row leads to an $\mathcal{O}(mn^2 + m^2 n)$ arithmetic operations procedure for building M [14]. In practice this can be improved significantly for sparse a_i's. Note that it is a simple matter to extend this approach to the case where the matrices A_i have low rank (as opposed to rank 1).

In the rank-one case, we would like to store only the vectors a_i as opposed to the matrices $A_i = a_i a_i^T$. Unfortunately, the standard SDPA sparse input data format does not have an option to store rank-one matrices efficiently. For the purpose of our numerical experiments, we implemented the following 'hack': Since the SDPA input file accepts only block diagonal matrix structure, we save each vector $a_i, i = 1, ..., m$ as a diagonal matrix, and introduce an additional parameter that we call *rank1* into the file of CSDP input parameters *param.csdp*, to ensure that the constraint matrices are interpreted in the right way by our modified software (i.e. not as diagonal matrices, but as a rank-one matrices).

Of course, a more structural (future) solution would be to modify the standard SDPA input format to allow more types of data matrices.

4 Parallel implementation of CSDP

In this section we describe our parallel version of CSDP on a Beowulf cluster of PC's. The code is based on CSDP 5.0 and it enjoys the same 64-bit capability as the shared memory version [5].

The code is written in ANSI C with additional MPI directives and use of ScaLAPACK library for parallel algebra computations. We also assume that optimized BLAS and LAPACK libraries are available on all nodes. The latter are used for implementation of matrix multiplication, Cholesky factorization and other linear algebra operations. As we already mentioned in the previous section, the most time-consuming operations are the computation of the elements of Schur complement matrix M and its Cholesky factorization. Therefore, in our development we used parallelization to accelerate these two bottlenecks in the sequential algorithm. Next we describe how our software actually works.

We denote by N the number of 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*. When defined in this way, the processes form a one-dimensional array. To be able to use ScaLAPACK efficiently, it is useful to map this one-dimensional array of processes into a two-dimensional rectangular array, called often a process grid. Let this process grid have N_r rows and N_c columns, where $N_r \times N_c = N$. Each process is thus indexed by its row and column coordinates as $P_{row,col}$, with $0 \leq row \leq N_r - 1$ and $0 \leq col \leq N_c - 1$.

The software starts execution by sending a copy of the execution and data files to all N nodes. Each node independently proceeds to allocate space for the variables X, y, Z locally, and starts the execution of its copy of the code, until computation of the Schur complement matrix M is reached.

We use a distributed storage for M during the complete solution cycle of the algorithm. As a result the software does not require that any of the nodes should be able to accommodate the whole matrix in its memory space. It has to be mentioned that M is stored on distributed memory only in one storage format, and not in two different formats as in SDPARA. We only use a two-dimensional block cyclic data distribution required later on by ScaLAPACK; see Figure 1. In particular, the matrix M is subdivided into blocks of size $NB \times NB$ for a suitable value

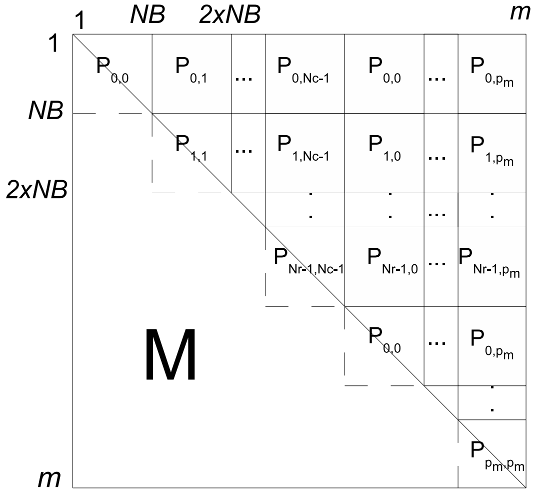

Figure 1: Two-dimensional block cyclic data distribution over two dimensional process grid.

of NB and the assignment of block to processes are as shown in the figure. Since the matrix M

is symmetric, we need only to compute the upper triangular part of it.

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

5 Numerical results

In this section we present numerical results testing our rank-one feature on the sequential CSDP code. We also show results in the last subsection from tests of our parallel implementation on a number of selected SDP test problems taken from SDPLIB, DIMACS and other sets of benchmark problems.

5.1 Experimental results for the sequential code on rank-one problems

In this section we test the practical efficiency of our modified sequential CSDP code to deal with rank-one constraints. From now on we will refer to this version as CSDP-R1. The numerical experiments were executed on PC with Intel x86 P4 (3.0GHz) CPU and 3GB of memory running the Linux operation system. Optimized BLAS and LAPACK libraries were used. The code was compiled with gcc 3.4.5 using the default values of all parameters from CSDP 5.0.[4] The input data format used was our modified SDPA sparse format for rank-one constraint matrices.

SDP problems with rank-one (or low rank) data matrices arise from several applications, including truss topology optimization [21], combinatorial optimization [12], sum-of-squares decompositions of polynomials [17], and the educational testing problem [28].

The performance of both CSDP and CSDP-R1 was tested on SDP instances with rank-one data matrices taken from [7]. These SDP problems approximate the minima of a set of twenty univariate test functions from Hansen et al. [13] on an interval, by first approximating the functions using Lagrange interpolation, and subsequently minimizing the Lagrange polynomials using SDP. These SDP problems only involve rank-one data matrices by using an SDP formulation due to Löfberg and Parrilo [17]. These rank-one problems were not run in parallel due to their relatively small sizes, i.e. the goal of these experiments was only to illustrate the potential gains of exploiting the rank-one structure.

These problems are representative of rank-one instances where the data matrices are dense.

The data in Tables 1 and 2 describes the results from our experiments with respect to the number of (Chebyshev) interpolation points 20, 40, 60 80, 100, 150 and 200 for CSDP and CSDP-R1, respectively. The sizes of the SDP problems roughly depend on the number 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 $m/2$.

[4] `http://infohost.nmt.edu/~borchers/csdp.html/`

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

Table 1: Solution times in seconds for CSDP for twenty rank-one test problems from [7].

The solution times in Tables 1 and 2 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 SDP problem. With increasing the number of interpolation points, the difference in solution times becomes apparent. When the order is 80, 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 200, CSDP-R1 obtains solutions about 2.7 times faster on average than the standard version of CSDP. These results clearly indicate that the simple construction (3.12) for exploiting rank-one structure when computing *Schur* in a primal-dual IPM algorithm can result 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 CSDP and CSDP-R1. The test was only on the test functions test1 and test17 functions, as the computational times vary only slightly for the different instances. The results are shown in Table 3. 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 running times with difference within a 2-3 tenths of a second for DSDP, CSDP and CSDP-R1. When the order is 100 then DSDP is faster than CSDP-R1 by a factor of three. When the size of the

Problem	20	40	60	80	100	150	200
$test1$	0.08	0.40	1.36	3.44	7.59	31.58	92.90
$test2$	0.06	0.42	1.38	3.46	7.51	31.96	98.28
$test3$	0.07	0.39	1.37	3.57	7.14	30.47	98.08
$test4$	0.06	0.39	1.28	3.36	7.25	31.22	88.95
$test5$	0.07	0.43	1.36	3.59	7.60	31.18	94.78
$test6$	0.07	0.40	1.38	3.47	7.63	33.74	93.25
$test7$	0.07	0.42	1.37	3.64	7.63	34.62	96.39
$test8$	0.06	0.39	1.34	3.66	7.14	30.48	92.77
$test9$	0.07	0.42	1.35	3.55	7.50	32.36	93.95
$test10$	0.07	0.40	1.33	3.47	7.71	32.38	92.15
$test11$	0.07	0.40	1.34	3.65	7.49	34.43	97.44
$test12$	0.06	0.39	1.25	3.64	7.15	31.96	99.44
$test13$	0.06	0.38	1.26	3.51	7.35	30.93	92.60
$test14$	0.06	0.39	1.31	3.52	7.33	33.03	95.05
$test15$	0.07	0.45	1.51	4.08	8.54	35.10	102.84
$test16$	0.08	0.54	1.69	4.64	9.65	41.63	112.57
$test17$	0.07	0.48	1.56	4.23	8.69	37.34	108.08
$test18$	0.08	0.49	1.65	4.04	8.72	35.01	98.89
$test19$	0.06	0.39	1.35	3.61	7.53	32.02	93.05
$test20$	0.06	0.38	1.28	3.53	7.66	30.88	94.87

Table 2: Solution times in seconds for CSDP-R1 for twenty rank-one test problems from [7].

Solver	20	40	60	80	100	150	200
CSDP	0.06	0.66	2.58	7.65	17.33	81.45	254.47
CSDP-R1	0.08	0.41	1.38	3.46	7.63	31.69	93.12
DSDP	0.15	0.34	0.76	1.27	2.48	6.75	14.80

Table 3: Running times in seconds for CSDP, CSDP-R1 and DSDP for one rank-one test problem from [7].

SDP problem further increases, the gap between the two algorithms grows as one might expect. For 200 interpolation points, CSDP-R1 is slower than DSDP by a factor close to 10.

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

5.2 Experimental results on parallel code

In this section we present results on numerical tests of our parallel implementation of CSDP using MPI, which we will refer to as PCSDP. The software was developed and tested on the DAS3 Beowulf cluster at the Delft University of Technology. Each node of the cluster has two 64bit AMD Opteron 2.4 GHz CPU, 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 through a 10GBit switch.

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

$$S = \frac{T_1}{T_N} \quad \text{and} \quad E = \frac{S}{N} = \frac{T_1}{(NT_N)}, \tag{5.13}$$

where T_1 and T_N are the times to run a code on one processor and N processors respectively. In principle one has $0 \leq E \leq 1$ with $E = 1$ corresponding to perfect speedup, but $E > 1$ may occur in practice due to cache issues.

Results on SDP benchmark problems

We selected for our tests fourteen medium and large-scale SDP problems from five different applications. These applications are: control theory, the maximum cut problem, the *Lovász* ϑ-function, the min k-uncut problem, and calculating electronic structures in quantum chemistry. All of the test instances are from standard benchmark suites.

Problems control10 and control11 are from control theory and they are the largest of this type in the SDPLIB test set [4]. The instance maxG51 is a medium size maximum cut problem chosen from the same benchmark set. ThetaG51, theta42, theta62, theta8, and theta82 are *Lovász* ϑ problems. The first three are from SDPLIB, while the others were generated by Toh and Kojima [26]. 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 [18] library of mixed semidefinite-quadratic-linear programs as is the min k-uncut test problem fap09. The sixteenth instance is CH4.1A1.STO6G, that comes form calculating electronic structures in quantum chemistry [23]. The initial point and the stopping criteria were the CSDP [3] default choices.

Tables 4 and 5 give the time in seconds for computing Schur complement matrix M, Cholesky factorization of M and total solution time using one to sixty four processors. Note that we did not exploit rank-one structure in these computations.

In these tables m is the number of constraints as before, and n_{max} is the size of the largest block in the primal matrix variable X. Computing *Schur* (the Schur complement matrix) for problems control10 and control11 is the most time consuming operation. For the rest of problems except maxG51, the time to compute the Cholesky factorization of M dominates. The parallel efficiencies were calculated using (5.13). This was not possible for all of the test problems, 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 4 presents only results for the problem that could run on 1 node.

For problems control10 and control11, computing the Schur complement matrix is the dominant task and it scales relatively well with the number of the processors. However this is not the

Problem	m	n_{max}	*Phase*	1	2	4	8	16	32	64
	1326	100	Schur	280.35	144.66	77.7	43.98	31.72	21.17	12.32
control10			Cholesky	34.57	24.15	19.52	15.58	10.73	9.57	8.26
			Total	344.00	196.01	124.39	87.82	71.62	62.33	56.61
	1596	110	Schur	489.53	243.16	125.00	71.03	53.09	30.57	21.66
control11			Cholesky	74.61	40.17	30.98	23.48	15.09	14.5	9.69
			Total	605.29	310.55	191.96	129.84	106.94	83.34	75.30
	7905	400	Schur	183.87	97.42	50.57	27.32	17.78	12.28	9.98
theta8			Cholesky	3336.99	2375.22	1276.52	524.33	274.71	239.74	100.14
			Total	3673.97	2592.06	1426.64	635.77	372.40	327.14	184.52
	5986	200	Schur	88.18	43.46	22.75	12.14	7.93	4.23	3.82
theta42			Cholesky	1489.70	827.11	452.74	226.59	122.00	107.09	48.65
			Total	1647.20	915.63	509.96	256.04	153.66	132.71	73.08
	13390	300	Schur	508.96	256.00	150.42	72.20	43.81	21.62	14.85
theta62			Cholesky	15094.69	13065.39	6790.26	2746.30	1422.07	923.26	355.69
			Total	15927.84	13530.76	7100.53	2935.48	1571.48	1034.38	459.62
	6910	1001	Schur	971.22	487.91	247.11	124.44	66.65	35.05	20.96
thetaG51			Cholesky	5258.25	3341.09	1852.00	826.88	436.09	369.94	165.19
			Total	7721.57	5368.77	3633.84	2498.57	2041.32	1943.69	1715.75
	1000	1000	Schur	1.29	1.10	0.62	0.28	0.19	0.12	0.08
maxG51			Cholesky	7.95	6.28	4.90	4.54	3.78	3.85	3.74
			Total	641.45	640.46	639.83	635.04	632.19	632.04	632.92
	16129	256	Schur	629.04	309.19	186.58	96.20	57.52	24.44	19.32
hamming_8_3_4			Cholesky	22980.47	15960.96	10340.12	5151.16	2278.29	1298.03	511.16
			Total	29237.21	16526.65	10719.20	5391.22	2462.25	1433.83	685.14
	15225	174	Schur	6656.76	3548.41	2476.86	1226.86	869.74	413.42	335.49
fap09			Cholesky	113529.39	97491.86	46902.41	23145.18	11928.57	7140.35	2714.18
			Total	122334.83	102429.12	50434.07	25155.29	13490.96	8153.83	3621.23

Table 4: Running times (in seconds) for the selected SDP benchmark problems for our solver PCSDP.

case with Cholesky factorization of M and the total running time, especially when the number of processors is 16, 32 and 64. The $m \times m$ Schur complement matrix is fully dense in general 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 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 network interconnect means relatively high delivery times for messages between nodes. Therefore, when the problem has fewer than $10,000$ constraints, like control10 and control11, it is not very efficient to solve it by distributed memory cluster. The

Problem	m	n_{max}	*Phase*	4	8	16	32	64
CH4	24503	324	Schur	2407.36	1256.58	817.43	383.41	263.69
			Cholesky	11507.03	6581.86	3541.88	2963.46	1263.78
			Total	14987.14	8630.98	5060.78	3963.46	2113.39
hamming_9_5_6	53761	512	Schur	$*$	$*$	701.34	390.59	277.57
			Cholesky	$*$	$*$	108753.44	72082.25	26032.19
			Total	$*$	$*$	110908.28	73736.39	27570.8
hamming_10_2	23041	1024	Schur	535.18	324.41	242.94	160.87	136.19
			Cholesky	33259.07	18460.04	9007.85	5232.26	1952.26
			Total	35023.35	19908.98	10398.88	6460.06	3165.4
hamming_11_2	56321	2048	Schur	$*$	$*$	1632.33	1167.66	1056.20
			Cholesky	$*$	$*$	160248.71	94922.18	36750.65
			Total	$*$	$*$	171201.57	104380.48	46087.89
theta82	23872	400	Schur	473.78	261.15	161.91	80.49	55.58
			Cholesky	35116.75	17637.60	9436.25	5677.68	2045.98
			Total	36090.25	18283.34	9936.71	6067.4	2403.21

Table 5: Running times (in seconds) for the selected large-scale SDP benchmark problems for our solver PCSDP ('*' - means lack of memory)

same issue is observed also in the middle-sized *Lovász* ϑ-type problems theta42, theta5, theta6 and theta8. The story is similar for the problem thetaG51. This is to be expected due to the relatively large dimensions of the primal and dual matrix variables X and Z.

Another example that suffers from poor parallel efficiency is the max-cut problem maxG51, where $n = m$. The constraints matrices 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 in a distributed memory computers is very inefficient. A similar problem with the scalability of maxG51 was observed for the solver SDPARA in [30].

Much better results are obtained for the large scale instances as theta62, fap09 and hamming_8_3_4. Only for 32 CPUs is the performance relatively poor due to the non square grid geometry in this case i.e. $N_r = 8, N_c = 4$. Both the Cholesky factorization and construction of the Schur complement matrix scale well with the number of processors for all three 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_5, hamming_10_2, hamming_11_2 and theta82 on one processor. In Table 5 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 also a small dimension 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 Schur complement matrix M. In this case $m \gg n$ and the solver efficiently solved them with a good scalability in terms of running times between 16 and 64 CPUs.

6 Comparison with other parallel SDP solvers

We compare the performance of our SDP software PCSDP with two other distributed memory solvers, namely SDPARA by Yamashita *et al.* [30] and PDSDP by Benson [1]. All three codes were run on the DAS3 Beowulf cluster at the Delft University of Technology.

The stopping criteria for PCSDP were as follows:

$$\frac{|\operatorname{trace}(CX) - b^T y|}{1 + |b^T y|} \leq 10^{-7}, \quad \frac{\|A(X) - b\|}{1 + \|b\|} \leq 10^{-7}, \quad \frac{\|A^*(y) - Z - C\|_F}{1 + \|C\|_F} \leq 10^{-7}.$$

In order to have comparable stopping criteria for SDPARA and PDSDP we set the SDPARA parameters 'epsilonDash' and 'epsilonStar' to 10^{-7}, and used the PDSDP parameter settings: DSDPSetGapTolerance = 10^{-7}, DSDPSetRTolerance = 10^{-6}, and DSDPSetPTolerance=10^{-6}, the last two being default values.

These stopping criteria are not identical, but comparable. Since our primary interest is the parallel efficiency of the codes and not the solution time on one processor, this suffices for our purposes.

Table 6 gives the respective running times on up to 64 cpu's for the three codes, and Table 7 the corresponding parallel efficiencies. Note that the parallel efficiencies could only be calculated for those problems that could be solved on one processor.

The parallel efficiencies in Table 7 are quite similar from PCSDP and SDPARA, as one might expect due to the similar algorithmic frameworks. However, it is clear from the instance fap09 that the codes may sometimes perform quite differently (see Table 6). The explanation is that the 2D process grid approach of PCSDP to forming the Schur complement matrix pays dividends here. In particular, the time spent by SDPARA for redistributing M is given in Table 8. Note that as much as 8,000 seconds is spent by SDPARA in this process for 8CPUs.

The parallel efficiencies of DSDP in Table 7 are also very similar to those of the primal-dual codes, even though the corresponding running times in Table 6 can be quite different. This is logical, since the three codes exploit parallelism in the same places, namely in forming and solving the respective Schur complement systems. This results in roughly the same speedup per iteration for the three codes.

7 Future developments

The ability of PCSDP to exploit rank-one structure in the data matrices may be extended to data matrices with 'low' rank. This is already done in some versions of the code DSDP [2] by way of spectral decompositions of the data matrices. However, it would be much more efficient

Problem	m	n_{max}	Solver	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
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: Running times in seconds for the selected SDP problems solved by PCSDP, SDPARA and PDSDP ('*' - means lack of memory).

if the user could specify the low rank structure directly in the input file. This highlights the need for a standard SDP input format that allows the user to directly specify more information on the data matrices than is currently possible with the sparse SDPA input format.

Acknowledgement

The authors would like to thank Professors Masakazu Kojima and Katsuki Fujisawa for their kind assistance with the installation of SDPARA on the DAS3 Beowulf cluster at the Delft University of Technology.

Problem	m	n_{max}	Solver	1	2	4	8	16	32	64
control10	1326	100	pcsdp	1	0.91	0.56	0.53	0.38	0.23	0.15
			sdpara	1	0.60	0.47	0.37	0.29	0.18	0.09
			pdsdp	1	0.79	0.49	0.29	0.22	0.14	0.08
control11	3028	250	pcsdp	1	0.98	0.69	0.57	0.35	0.24	0.16
			sdpara	1	0.91	0.73	0.60	0.47	0.31	0.16
			pdsdp	1	0.74	0.47	0.29	0.16	0.10	0.07
theta42	5986	200	pcsdp	1	1.40	1.06	0.75	0.59	0.33	0.24
			sdpara	1	1.28	1.04	0.69	0.59	0.32	0.23
			pdsdp	1	1.16	1.06	0.79	0.68	0.39	0.25
theta8	7905	400	pcsdp	1	1.45	1.20	0.88	0.69	0.40	0.30
			sdpara	1	1.37	1.11	0.85	0.69	0.42	0.29
			pdsdp	1	1.21	0.99	0.84	0.65	0.41	0.29
theta62	13390	300	pcsdp	1	1.34	1.12	0.91	0.75	0.44	0.38
			sdpara	*	*	*	*	*	*	*
			pdsdp	*	*	*	*	*	*	*
thetaG51	6910	1001	pcsdp	1	1.27	0.94	0.65	0.44	0.24	0.15
			sdpara	1	1.22	0.92	0.62	0.42	0.23	0.15
			pdsdp	1	0.77	0.88	0.57	0.46	0.29	0.17
maxG51	1000	1000	pcsdp	1	0.50	0.25	0.12	0.06	0.03	0.02
			sdpara	1	0.49	0.24	0.12	0.06	0.03	0.01
			pdsdp	1	0.73	0.53	0.31	0.17	0.09	0.04
hamming_8_3_4	16129	256	pcsdp	1	1.36	1.14	0.93	0.78	0.46	0.41
			sdpara	*	*	*	*	*	*	*
			pdsdp	*	*	*	*	*	*	*
fap09	15225	174	pcsdp	1	1.36	1.11	0.93	0.74	0.46	0.39
			sdpara	*	*	*	*	*	*	*
			pdsdp	*	*	*	*	*	*	*

Table 7: Parallel efficiencies for the selected SDP problems solved by PCSDP, SDPARA and PDSDP. A * indicates that the problem did not run on one processor and that the parallel efficiency could therefore not be calculated.

8CPUs	16CPUs	32CPUs	64CPUs
8028	3500	1234	1139

Table 8: Time (in seconds) spent by SDPARA to redistribute the Schur complement matrix over the processors for the instance fap09.

References

[1] S. J. Benson. Parallel computing on semidefinite programs. Technical Report ANL/MCS-P939-0302, Mathematics and Computer Science Division, Argonne National Laboratory, April 2003.

[2] S. J. Benson, Y. Ye, and X. Zhang. Solving large-scale sparse semidefinite programs for combinatorial optimization. *SIAM J. Optim.*, 10(2):443–461 (electronic), 2000.

[3] B. Borchers. CSDP, a C library for semidefinite programming. *Optim. Methods Softw.*, 11/12(1-4):613–623, 1999.

[4] B. Borchers. SDPLIB 1.2, library of semidefinite programming test problems. *Optim. Methods Softw.*, 11/12(1-4):683–690, 1999.

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

[6] E. de Klerk. *Aspects of semidefinite programming, Interior point algorithms and selected applications*, volume 65 of *Applied Optimization*. Kluwer Academic Publishers, Dordrecht, 2002.

[7] E. de Klerk, G. Elabwabi, and D. den Hertog. Optimization of univariate functions on bounded intervals by interpolation and semidefinite programming. CentER Discussion paper 2006-26, Tilburg University, The Netherlands, April 2006. Available at *Optimization Online*.

[8] E. de Klerk, C. Roos, and T. Terlaky. Initialization in semidefinite programming via a self-dual skew-symmetric embedding. *Oper. Res. Lett.*, 20(5):213–221, 1997.

[9] G. Elabwabi. *Topics in global optimization using semidefinite optimization.* PhD thesis, Delft University of Technology, to be published (2007).

[10] K. Fujisawa, M. Kojima, and K. Nakata. Exploiting sparsity in primal–dual interior-point methods for semidefinite programming. *Math. Programming*, 79(1-3, Ser. B):235–253, 1997.

[11] M. Fukuda, M. Kojima, K. Mucro, and K. Nakata. Exploiting sparsity in semidefinite programming via matrix completion. I. General framework. *SIAM J. Optim.*, 11(3):647–674 (electronic), 2000/01.

[12] M.X. Goemans and D.P. Williamson. Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. *Journal of the ACM*, 42(6):1115–1145, 1995.

[13] P. Hansen, B. Jaumard, and S.-H. Lu. Global optimization of univariate Lipschitz functions. II. New algorithms and computational comparison. *Math. Programming*, 55(3, Ser. A):273–292, 1992.

[14] C. Helmberg and F. Rendl. Solving quadratic $(0, 1)$-problems by semidefinite programs and cutting planes. *Math. Programming*, 82(3, Ser. A):291–315, 1998.

[15] C. Helmberg, F. Rendl, R. J. Vanderbei, and H. Wolkowicz. An interior-point method for semidefinite programming. *SIAM J. Optim.*, 6(2):342–361, 1996.

[16] M. Kojima, S. Shindoh, and S. Hara. Interior-point methods for the monotone semidefinite linear complementarity problem in symmetric matrices. *SIAM J. Optim.*, 7(1):86–125, 1997.

[17] J. Löfberg and P. Parrilo. From coefficients to samples: a new approach to SOS optimization. In *IEEE Conference on Decision and Control*, December 2004.

[18] H. D. Mittelman. Dimacs challenge benchmarks. `http://plato.asu.edu/dimacs/`, 2000.

[19] R. D. C. Monteiro. Primal-dual path-following algorithms for semidefinite programming. *SIAM J. Optim.*, 7(3):663–678, 1997.

[20] 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. *Optim. Methods Softw.*, 11/12(1-4):91–140, 1999.

[21] M. Ohsaki, K. Fujisawa, N. Katoh and Y. Kanno, Semi-definite programming for topology optimization of trusses under multiple eigenvalue constraints. *Comp. Meth. Appl. Mech. Engng.*, 180: 203–217, 1999.

[22] K. Nakata, K. Fujisawa, M. Fukuda, M. Kojima, and K. Mucro. Exploiting sparsity in semidefinite programming via matrix completion. II. Implementation and numerical results. *Math. Program.*, 95(2, Ser. B):303–327, 2003.

[23] 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. *The Journal of Chemical Physics*, 114(19):8282–8292, 2001.

[24] Y. Nesterov. Squared functional systems and optimization problems. In *High performance optimization*, volume 33 of *Appl. Optim.*, pages 405–440. Kluwer Acad. Publ., Dordrecht, 2000.

[25] J. F. Sturm. Using SeDuMi 1.02, a MATLAB toolbox for optimization over symmetric cones. *Optim. Methods Softw.*, 11/12(1-4):625–653, 1999.

[26] K. C. Toh and M. Kojima. Solving some large scale semidefinite programs via the conjugate residual method. *SIAM J. Optim.*, 12(3):669–691 (electronic), 2002.

[27] K. C. Toh, M. J. Todd, and R. H. Tütüncü. SDPT3—a MATLAB software package for semidefinite programming, version 1.3. *Optim. Methods Softw.*, 11/12(1-4):545–581, 1999.

[28] L. Vandenberghe and S. Boyd. Semidefinite programming. *SIAM Review*, 38:49–95, 1996.

[29] M. Yamashita, K. Fujisawa, and M. Kojima. Implementation and evaluation of SDPA 6.0 (semidefinite programming algorithm 6.0). *Optim. Methods Softw.*, 18(4):491–505, 2003.

[30] M. Yamashita, K. Fujisawa, and M. Kojima. SDPARA: SemiDefinite Programming Algorithm paRAllel version. *Parallel Comput.*, 29(8):1053–1067, 2003.