A high-performance software package for semidefinite programs: SDPA 7

Makoto Yamashita[†], Katsuki Fujisawa[‡], Kazuhide Nakata[◇], Maho Nakata[♯],
Mituhiro Fukuda[♮], Kazuhiro Kobayashi[*], and Kazushige Goto[+]

January, 2010

Abstract

The SDPA (SemiDefinite Programming Algorithm) Project launched in 1995 has been known to provide high-performance packages for solving large-scale Semidefinite Programs (SDPs). SDPA Ver. 6 solves efficiently large-scale dense SDPs, however, it required much computation time compared with other software packages, especially when the Schur complement matrix is sparse. SDPA Ver. 7 is now completely revised from SDPA Ver. 6 specially in the following three implementation; (i) modification of the storage of variables and memory access to handle variable matrices composed of a large number of sub-matrices, (ii) fast sparse Cholesky factorization for SDPs having a sparse Schur complement matrix, and (iii) parallel implementation on a multi-core processor with sophisticated techniques to reduce thread conflicts. As a consequence, SDPA Ver. 7 can efficiently solve SDPs arising from various fields with shorter time and less memory than Ver. 6 and other software packages. In addition, with the help of multiple precision libraries, SDPA-GMP, -QD and -DD are implemented based on SDPA to execute the primal-dual interior-point method with very accurate and stable computations.

The objective of this paper is to present brief explanations of SDPA Ver. 7 and to report its high performance for large-scale dense and sparse SDPs through numerical experiments compared with some other major software packages for general SDPs. Numerical experiments also show the astonishing numerical accuracy of SDPA-GMP, -QD and -DD.

Keywords: semidefinite program, primal-dual interior-point method, high-accuracy calculation

† Department of Mathematical and Computing Sciences, Tokyo Institute of Technology, 2-12-1-W8-29 Ookayama, Meguro-ku, Tokyo 152-8552, Japan.
`Makoto.Yamashita@is.titech.ac.jp`

‡ Department of Industrial and Systems Engineering, Chuo University, 1-13-27 Kasuga, Bunkyo-ku, Tokyo 112-8551, Japan.
`fujisawa@indsys.chuo-u.ac.jp`

◇ Department of Industrial Engineering and Management, Tokyo Institute of Technology, 2-12-1-W9-60 Ookayama, Meguro-ku, Tokyo 152-8552, Japan.
`nakata.k.ac@m.titech.ac.jp`

♯ Advanced Center for Computing and Communication, RIKEN, 2-1 Hirosawa, Wako-shi, Saitama 351-0198, Japan.
`maho@riken.jp`

♮ Global Edge Institute, Tokyo Institute of Technology, 2-12-1-S6-5 Ookayama, Meguro-ku, Tokyo 152-8550, Japan.
`mituhiro@is.titech.ac.jp`

∗ Center for Logistics Research, National Maritime Research Institute, 6-38-1 Shinkawa, Mitaka-shi, Tokyo 181-0004, Japan.
`kobayashi@nmri.go.jp`

+ Texas Advanced Computing Center, The University of Texas at Austin, 405 W 25th St 301, Austin, TX 78705-4831, United States.
kgoto@tacc.utexas.edu

1. Introduction

All started from a small Mathematica®program called *pinpal*. It was "translated" into a C++ programming code and named SDPA (SemiDefinite Programming Algorithm), a high-performance oriented software package to solve SemiDefinite Programming (SDP).

Table 1 demonstrates the significant progress of SDPA by showing how fast SDPA has become along the version upgrades. We applied each version of SDPA to SDPs chosen from the SDPLIB (the standard SDP benchmark problems) [6] and the SDP collections downloadable from Mittelmann's web page [25] (through this article we call this collection the Mittelmann's problem) on the same computer (Xeon 5550 (2.66GHz) x 2, 72GB Memory space). We excluded the Ver 1.0 from the result due to serious numerical instability from its primitive implementation. In particular, for mater-4, SDPA 7.3.1 attains more than 6,000 times of speed-up from SDPA 2.01.

Table 1: Computation time (in seconds) attained by each version of SDPA.

versions	mcp500-1	theta6	mater-4
2.01	569.2	2643.5	62501.7
3.20	126.8	216.3	7605.9
4.50	53.6	217.6	29601.9
5.01	23.8	212.0	31258.1
6.2.1	1.6	20.7	746.7
7.3.1	1.5	14.2	10.4

SDPA solves simultaneously the following standard form semidefinite program $\mathcal{P}$ and its dual $\mathcal{D}$:

$$\left\{\begin{array}{lll} \mathcal{P}: & \text{minimize} & \sum_{k=1}^{m} c_k x_k \\ & \text{subject to} & \boldsymbol{X} = \sum_{k=1}^{m} \boldsymbol{F}_k x_k - \boldsymbol{F}_0, \quad \boldsymbol{X} \succeq \boldsymbol{O}, \\ & & \boldsymbol{X} \in \mathcal{S}^n, \\ \mathcal{D}: & \text{maximize} & \boldsymbol{F}_0 \bullet \boldsymbol{Y} \\ & \text{subject to} & \boldsymbol{F}_k \bullet \boldsymbol{Y} = c_k \ (k = 1, 2, \ldots, m), \quad \boldsymbol{Y} \succeq \boldsymbol{O}, \\ & & \boldsymbol{Y} \in \mathcal{S}^n. \end{array}\right. \tag{1.1}$$

Here, $\mathcal{S}^n$ is the space of $n \times n$ real symmetric matrices (possibly with multiple diagonal blocks as detailed in Section 2). For $\boldsymbol{U}$ and $\boldsymbol{V}$ in $\mathcal{S}^n$, the Hilbert-Schmidt inner product $\boldsymbol{U} \bullet \boldsymbol{V}$ is defined as $\sum_{i=1}^{n}\sum_{j=1}^{n} U_{ij}V_{ij}$. The symbol $\boldsymbol{X} \succeq \boldsymbol{O}$ indicates that $\boldsymbol{X} \in \mathcal{S}^n$ is symmetric positive semidefinite. An instance of SDP is determined by $\boldsymbol{c} \in \mathbb{R}^m$ and $\boldsymbol{F}_k \in \mathcal{S}^n$ $(k = 0, 1, \ldots, m)$. When $(\boldsymbol{x}, \boldsymbol{X})$ is a feasible solution (or a minimum solution, respectively) of the primal problem $\mathcal{P}$, and $\boldsymbol{Y}$ is a feasible solution (or a maximum solution, respectively) of the dual problem $\mathcal{D}$, we call $(\boldsymbol{x}, \boldsymbol{X}, \boldsymbol{Y})$ a feasible solution (or an optimal solution, respectively) of the SDP.

There is an extensive list of publications related to the theory, algorithms, and applications of SDPs. A succinct description can be found for instance in [33, 38]. In particular, their applications in system and control theory [7], combinatorial optimization [13], quantum chemistry [31], polynomial optimization problems [20], sensor network problems [4, 17]

have been of extreme importance in posing challenging problems for the SDP codes.

Many algorithms have been proposed and many SDP codes have been implemented to solve SDPs. Among them, the current version 7.3.1 of SDPA implements the Mehrotra type predictor-corrector primal-dual interior-point method using the HKM (short for HRVW/KSH/M) search direction [14, 19, 26]. For the subsequent discussions, we place a simplified version of the path-following interior-point method implemented in SDPA [11]. In particular, the Schur Complement Matrix (SCM) of Step 2 will be often mentioned.

Framework 1.1 (Primal-Dual Interior-Point Method (PDIPM))

1. *Choose an initial point* $(\boldsymbol{x}, \boldsymbol{X}, \boldsymbol{Y})$ *with* $\boldsymbol{X} \succ \boldsymbol{O}$ *and* $\boldsymbol{Y} \succ \boldsymbol{O}$*. Set the parameter* $\gamma \in (0, 1)$*.*
2. *Compute the Schur Complement Matrix (SCM)* $\boldsymbol{B}$ *and the right hand side vector* $\boldsymbol{r}$ *of the Schur complement equation. The elements of* $\boldsymbol{B}$ *are of the form* $B_{ij} = (\boldsymbol{Y}\boldsymbol{F}_i\boldsymbol{X}^{-1}) \bullet \boldsymbol{F}_j$*.*
3. *Solve the Schur complement equation* $\boldsymbol{B}d\boldsymbol{x} = \boldsymbol{r}$ *to obtain* $d\boldsymbol{x}$*. Then compute* $d\boldsymbol{X}$ *and* $d\boldsymbol{Y}$ *for this* $d\boldsymbol{x}$*.*
4. *Compute the maximum step lengths* $\alpha_p = \max\{\alpha \in (0, 1] : \boldsymbol{X} + \alpha d\boldsymbol{X} \succeq \boldsymbol{O}\}$ *and* $\alpha_d = \max\{\alpha \in (0, 1] : \boldsymbol{Y} + \alpha d\boldsymbol{Y} \succeq \boldsymbol{O}\}$*.*
5. *Update the current point keeping the positive definiteness of* $\boldsymbol{X}$ *and* $\boldsymbol{Y}$*. Update* $(\boldsymbol{x}, \boldsymbol{X}, \boldsymbol{Y}) = (\boldsymbol{x} + \gamma\alpha_p d\boldsymbol{x}, \boldsymbol{X} + \gamma\alpha_p d\boldsymbol{X}, \boldsymbol{Y} + \gamma\alpha_d d\boldsymbol{Y})$*.*
6. *If the stopping criteria are satisfied, output* $(\boldsymbol{x}, \boldsymbol{X}, \boldsymbol{Y})$ *as an approximate optimal solution. Otherwise, return to Step 2.*

SDPA has the highest version number 7.3.1 among all generic SDP codes, due to its longest history which goes back to December of 1995. Table 2 summarizes the main characteristics of historical versions of SDPA.

Table 2: Details on main features of historical versions of SDPA.

versions	released years	main changes from the previous version	references
1.0	1995	C++ implementation of the primal-dual interior-point method using the HKM search direction	
2.0	1996	Implementation of the Mehrotra type predictor-corrector method	
3.0	1997	Novel formula to compute the SCM according to the sparsity of the data	[12]
4.0	1998	Full implementation of the above formula for all block matrices, callable library	
5.0	1999	Fast step-size computation using the bisection method to approximate minimum eigenvalues	[11]
6.0	2002	Replacing meschach with BLAS/ATLAS and LAPACK	[40]
7.0	2009	Major improvements are discussed in this article	

The main drive force for the development of new codes has been an existence of a demand for solving larger SDPs in a shorter time. Meanwhile, there is also a timid demand to solve ill-posed SDPs which require high precision floating-point arithmetic [24, 31]. The new features of the current version 7.3.1 and SDPA-GMP were provided to supply practical solutions to these demands.

The primary purpose of this article is to describe in details the new features of SDPA 7.3.1 accompanied by comparative numerical experiments. We summarize below the major contributions described in the following pages of this article.

1. New data structure (Section 2.1):
   SDPA 7.3.1 has a new data structure for sparse block diagonal matrices which allows more efficient arithmetic operations. The computational time was drastically reduced specially for problems which have a large number of diagonal submatrices. Many redundant dense variable matrices were eliminated. Hence, the overall memory usage can be less than half if compared to the previous version.
2. Sparse Cholesky factorization (Section 2.2):
   The SCM is fully dense in general even if data matrices are sparse. However, there are cases where SDPs have a large number of zero diagonal submatrices such as for SDP relaxation of polynomial optimization problems [18, 20]. Thus the SCM become sparse. SDPA 7.3.1 has now a sparse Cholesky factorization and it can automatically select either dense or sparse Cholesky factorizations accordingly with the sparsity of the SCM.
3. Multi-thread computation (Section 2.3):
   The acceleration using multi-thread computation is employed in two fronts. Firstly, we distribute the computation of each row of the SCM to each thread of multi-core processors. Secondly, we use the numerical linear algebra libraries which support multi-thread computation of matrix-matrix and matrix-vector multiplications. The speed-up obtained by these multi-thread computation is beyond one's expectation.
4. Very highly accurate calculation and/or good numerical stability (Section 4):
   The PDIPM sometimes encounters numerical instability near an optimal solution, mainly due to the factorization of an ill-conditioned SCM. SDPA-GMP, SDPA-QD and SDPA-DD incorporates MPACK (generic multiple-precision library for linear algebra) instead of usual double floating-point libraries: LAPACK and BLAS. The difference between SDPA-GMP, SDPA-QD and SDPA-DD is in the numerical precision supplied by the GMP (The GNU Multiple Precision Arithmetic Library), QD and DD [15], respectively. It is noteworthy to SDP researchers that solutions of sensitive SDP problems which other SDP codes fail are now obtainable by SDPA-GMP/-QD/-DD.

Section 3 is dedicated to the numerical experiments. When we compared with the major existing SDP codes on SDPs not having a particular structure or property, we confirmed SDPA 7.3.1 obtained a superior performance in terms of computational time, memory consumption, and numerical stability. In this section, we also analyze the effects of new features listed above.

The numerical results on high accurate solutions will be separately showed in Section 4, since the only public available software packages having more precisions than double precision are only from the SDPA Family. Section 5 will discuss some extensions of the SDPA Family. Finally, we will present the concluding remarks in Section 6.

The software packages SDPA, SDPA-GMP (-QD, -DD), SDPA-C (SDPA with the completion method) [28], SDPARA (a parallel version of SDPA) [39], and SDPARA-C [29] (the integration of SDPARA and SDPA-C), are called *SDPA Family*. All software packages can be downloaded from the SDPA web site:

http://sdpa.indsys.chuo-u.ac.jp/sdpa/

In addition, some of them are registered on the SDPA Online Solver and hence, users can use them via the Internet. Details on the Internet access will be described in Section 5. The Matlab interface, SDPA-M, is now incorporated into SDPA 7.3.1. This means that after compilation of the regular SDPA, one can also compile the mex files and call SDPA from the MATLAB.

2. New features of SDPA 7.3.1

In this section, we describe the new improvements introduced in the current version of SDPA 7.3.1.

2.1. Newly implemented data structures

We start with the introduction of block diagonal matrices. SDPA was the first SDP software package which could manage block diagonal matrices. This has been reflected in the fact that the *SDPA format* [10] is widely used as a representative SDP input form.

Now, let us suppose the input matrices $\boldsymbol{F}_0, \boldsymbol{F}_1, \ldots, \boldsymbol{F}_m$ share the same diagonal structure,

$$\boldsymbol{F} = \begin{pmatrix} \boldsymbol{F}^1 & \boldsymbol{O} & \cdots & \boldsymbol{O} \\ \boldsymbol{O} & \boldsymbol{F}^2 & \cdots & \boldsymbol{O} \\ \vdots & \vdots & \ddots & \vdots \\ \boldsymbol{O} & \boldsymbol{O} & \cdots & \boldsymbol{F}^\ell \end{pmatrix}.$$

Here $\boldsymbol{F}^b \in \mathcal{S}^{n_b}$ $(b = 1, 2, \ldots, \ell)$, and $n = \sum_{b=1}^{\ell} n_b$. From (1.1), it is clear that the matrix variables $\boldsymbol{X}$ and $\boldsymbol{Y}$ of size n stored internally in SDPA have exactly the same structure. Hereafter, we call each submatrix $\boldsymbol{F}^b$ $(b = 1, 2, \ldots, \ell)$ placed at the diagonal position a *diagonal submatrix*.

The techniques presented in this section are extremely effective when an SDP problem has a very large number ℓ of blocks. For example, Polynomial Optimization Problems (POP) [17, 20] often produce this type of SDPs for their relaxation problems. Another example is when an SDP has many non-negative linear constraints. These constraints are represented by a diagonal data matrix in SDPA format [10], and its diagonal elements can be interpreted as a collection of symmetric matrices of size one.

SDPA 7.3.1 adopts a new storage scheme for the diagonal block matrix structure which allows one to reduce both the memory usage and the computational time. The nonzero elements of a sparse diagonal submatrix are stored as triple vectors consisted by their row/column indices and values. SDPA 6.x always stores all ℓ diagonal submatrices $\boldsymbol{F}_k^b \in \mathcal{S}^{n_b}$ $(b = 1, 2, \ldots, \ell)$ without regarding if some of them might be the zero matrix. In contrast, SDPA 7.3.1 implements a more compact storage scheme which stores only the *nonzero* diagonal submatrices, and a list to the corresponding nonzero diagonal submatrix indices.

The influence of skipping zero diagonal submatrices might appear to have a scant effect. However, this new block diagonal matrix storage benefits the following three computational routines; evaluation of primal constraints, evaluation of dual constraints, and computation of the SCM. The total computational cost of these three routines often exceeds 80% of the whole computation cost, and can not be ignored.

Now, let us focus on the last routine of the three, since it is related to Section 2.2 and 2.3. SDPA employs the HKM search direction [14, 19, 26] in the PDIPM. At each iteration of

the PDIPM, we evaluate the SCM $\boldsymbol{B}$ whose elements are defined as $B_{ij} = (\boldsymbol{Y}\boldsymbol{F}_i\boldsymbol{X}^{-1}) \bullet \boldsymbol{F}_j$. Algorithms 2.1 and 2.2 are pseudo-codes for SDPA 6.x and 7.3.1, respectively, for this routine which takes into account the block diagonal matrix structure.

Algorithm 2.1
SCM computation in SDPA 6.x

$\boldsymbol{B} = \boldsymbol{O}$
for $b = 1, 2, \dots, \ell$
  for $i = 1, 2, \dots, m$
    for $j = 1, 2, \dots, m$
      $B_{ij} = B_{ij} + \boldsymbol{Y}^b \boldsymbol{F}_i^b (\boldsymbol{X}^b)^{-1} \bullet \boldsymbol{F}_j^b$
    end
  end
end

Algorithm 2.2
SCM computation in SDPA 7.3.1

$\boldsymbol{B} = \boldsymbol{O}$
for $b = 1, 2, \dots, \ell$
  for $i \in \{i \mid \boldsymbol{F}_i^b \neq \boldsymbol{O}\}$
    for $j \in \{j \mid \boldsymbol{F}_j^b \neq \boldsymbol{O}\}$
      $B_{ij} = B_{ij} + \boldsymbol{Y}^b \boldsymbol{F}_i^b (\boldsymbol{X}^b)^{-1} \bullet \boldsymbol{F}_j^b$
    end
  end
end

In SDPA 7.3.1 (Algorithm 2.2), the 3rd and 4th lines' "for" are executed only for *nonzero* diagonal submatrices, exploiting the sparsity of block structures. If we use $\bar{c}$ to denote the computational cost to compute the 5th line $B_{ij} = B_{ij} + \boldsymbol{Y}^b \boldsymbol{F}_i^b (\boldsymbol{X}^b)^{-1} \bullet \boldsymbol{F}_j^b$ and $\bar{m}$ to denote $\max_{b=1,2,\cdots,\ell} \#\{i \mid \boldsymbol{F}_i^b \neq \boldsymbol{O}\}$, then the cost to compute the SCM is $\mathcal{O}(\ell m^2 + \bar{c})$ for SDPA 6.x and $\mathcal{O}(\ell \bar{m}^2 + \bar{c})$ for SDPA 7.3.1, respectively. Since $\bar{m}$ is constant or significantly smaller than m in many SDP applications and $\bar{c}$ is not so large for SDPs having sparse SCM, this change has brought us a remarkable computation time reduction.

We now move to the discussion about dense matrices. In general, the variable matrices $\boldsymbol{X}^{-1}, \boldsymbol{Y} \in \mathcal{S}^n$ are dense even when the input data matrices $\boldsymbol{F}_k \in \mathcal{S}^n$ $(k = 1, 2, \dots, m)$ are sparse. SDPA 7.3.1 uses less than half of memory storage of SDPA 6.x for dense matrices. The number of stored dense matrices is reduced from 31 to only 15. The 11 auxiliary matrices in SDPA 6.x have been successfully combined into two auxiliary matrices in SDPA 7.3.1 by using a concept similar to object pool pattern developed in design pattern study. The rest of the reduction was made by eliminating redundant matrices related to the initial point input. If $n = 5,000$ and the input matrices are composed of one block, this effect leads to approximately 3GB of memory space reduction.

Furthermore, by optimizing the order of linear algebra operations used in the PDIPM, the number of multiplications between dense matrices of size n in one iteration is reduced from 10 (SDPA 6.x) to 8 (SDPA 7.3.1).

2.2. Sparse Cholesky factorization

As it is clear from the formula to compute the SCM $\boldsymbol{B}$

$$B_{ij} = (\boldsymbol{Y}\boldsymbol{F}_i\boldsymbol{X}^{-1}) \bullet \boldsymbol{F}_j = \sum_{b=1}^{\ell} (\boldsymbol{Y}^b \boldsymbol{F}_i^b (\boldsymbol{X}^{-1})^b) \bullet \boldsymbol{F}_j^b,$$

only the elements corresponding to the following indices of $\boldsymbol{B}$ becomes nonzero

$$\bigcup_{b=1}^{\ell} \left\{(i,j) \in \{1, 2, \dots, m\}^2 \middle| \ \boldsymbol{F}_i^b \neq \boldsymbol{O} \text{ and } \boldsymbol{F}_j^b \neq \boldsymbol{O}\right\}. \tag{2.1}$$

Therefore, if the input data matrices $\boldsymbol{F}_k$ $(k = 1, 2, \dots, m)$ consist of many zero diagonal submatrices, which is frequently the case for polynomial optimization problems [20], we

can expect that the SCM $\boldsymbol{B}$ becomes sparse. Even in these cases, SDPA 6.x performs the ordinary dense Cholesky factorization. To exploit the sparsity of the SCM $\boldsymbol{B}$, we newly implemented the sparse Cholesky factorization in SDPA 7.3.1.

We employ the sparse Cholesky factorization implemented in MUMPS [1, 2, 3]. However, just calling its routines is not so sophisticated. Since the performance of the sparse Cholesky factorization strongly depends on the sparsity of $\boldsymbol{B}$, we should apply the dense Cholesky factorization in the case most elements of $\boldsymbol{B}$ are non-zero. In SDPA 7.3.1, we adopt some criteria to perform either the dense Cholesky factorization or the sparse Cholesky factorization to the matrix $\boldsymbol{B}$.

We take a look at the criteria from the simplest ones. Since the sparsity of $\boldsymbol{B}$ is defined by (2.1), in the case an input matrix $\boldsymbol{F}_k$ has all non-zero blocks, *i.e.,* $\boldsymbol{F}_k^b \neq \boldsymbol{O}$, $\forall b = 1, \ldots, \ell$, we should select the dense Cholesky factorization. The situation in which the ratio between the number of non-zero elements of (2.1) and m^2 (fully dense) exceeds some threshold, for example 0.7, is also another case to consider the dense Cholesky factorization.

It is known that the sparse Cholesky factorizations usually produce additional non-zeros elements called fill-in's other than (2.1). To perform the sparse Cholesky factorization more effectively, we should apply a re-ordering of rows/columns to $\boldsymbol{B}$ to reduce its fill-in's which by itself is an NP-hard problem. Thus, we usually employ one of several heuristic methods for re-orderings, such as AMD (Approximate Minimum Degree) or AMF (Approximate Minimum Fill). These methods are the standard heuristics called in the analysis phase of MUMPS, and their computational costs are reasonably cheap in the framework of PDIPM. The obtained ordering determines the number of additional fill-in's. When the ratio between the number of additional fill-in's plus the original non-zeros in (2.1), and m^2 exceeds another threshold, for example 0.8, we switch to the dense Cholesky factorization. Even though the threshold 0.8 seems too dense, we verified from preliminary experiments that the multiple frontal method implemented in MUMPS gives better performance than the normal dense Cholesky factorization; the multi frontal method automatically applies the dense Cholesky factorization to some dense parts of $\boldsymbol{B}$ even when the whole matrix is sufficiently sparse.

Another reason why we adopt MUMPS is because it can estimate the number of floating-point operations required for the elimination process, which occupies most of the computation time of the multiple frontal method. This estimation enables us to compare the computational cost of the sparse Cholesky factorization and the dense Cholesky factorization. Since the cost of the dense Cholesky factorization to $\boldsymbol{B}$ is approximately proportional to $\frac{1}{3}m^3$, we finally adopt the sparse Cholesky factorization when the following inequality holds:

$$
\mathit{sparse_cost} < \frac{1}{3} m^3 \times \mathit{sd_ratio},
$$

where $\mathit{sparse_cost}$ is the cost estimated by MUMPS and $\mathit{sd_ratio}$ is some sparse/dense ratio. Based on numerical experiments, we set $\mathit{sd_ratio} = 0.85$.

We decide if we should adopt the sparse or the dense Cholesky factorization only once before the main iterations of the PDIPM, because the sparse structure of $\boldsymbol{B}$ is invariant during all the iterations. The introduction of the sparse Cholesky factorization provides a significant improvement on the performance of SDPA 7.3.1 for SDPs with sparse SCMs, while it still maintains the existing fine performance for dense SCMs.

2.3. Mutil-thread computation

Multi-core processors is one of major evolutions in computing technology and delivers outstanding performance in high-performance computing. Modern multi-core processors are usually composed of two or more independent and general-purpose cores. The performance gained by multi-core processors is heavily dependent on implementation such as multi-threading. Multi-threading paradigm is becoming very popular thanks to some application programming interfaces such as OpenMP and pthread. They provide simple frameworks to obtain parallel advantage for optimization software packages. We can execute multiple threads in parallel and expect to solve SDPs in a shorter time.

Here, we discuss in details how SDPA 7.3.1 exploits the multi-threading when determining all elements of the SCM. Fujisawa *et al.* [12] proposed an efficient method for the computation of all elements of $\boldsymbol{B}$ when a problem is large scale and sparse. They have introduced three kinds of formula, $\mathcal{F}$-1, $\mathcal{F}$-2 and $\mathcal{F}$-3, for the computation of B_{ij} in accordance to the sparsity of $\boldsymbol{F}_k^b$'s $(b = 1, 2, \ldots, \ell)$.

$\mathcal{F}$-1: If $\boldsymbol{F}_i^b$ and $\boldsymbol{F}_j^b$ are dense, compute $B_{ij} = B_{ij} + \boldsymbol{Y}^b \boldsymbol{F}_i^b (\boldsymbol{X}^b)^{-1} \bullet \boldsymbol{F}_j^b$.

$\mathcal{F}$-2: If $\boldsymbol{F}_i^b$ is dense and $\boldsymbol{F}_j^b$ is sparse, compute

$$B_{ij} = B_{ij} + \sum_{\alpha=1}^{n_b} \sum_{\beta=1}^{n_b} [\boldsymbol{F}_j^b]_{\alpha\beta} \left(\sum_{\gamma=1}^{n_b} \boldsymbol{Y}_{\alpha\gamma}^b [\boldsymbol{F}_i^b (\boldsymbol{X}^b)^{-1}]_{\gamma\beta} \right).$$

$\mathcal{F}$-3: If $\boldsymbol{F}_i^b$ and $\boldsymbol{F}_j^b$ are sparse, compute

$$B_{ij} = B_{ij} + \sum_{\gamma=1}^{n_b} \sum_{\epsilon=1}^{n_b} \left(\sum_{\alpha=1}^{n_b} \sum_{\beta=1}^{n_b} [\boldsymbol{F}_i^b]_{\alpha\beta} \boldsymbol{Y}_{\alpha\gamma}^b (\boldsymbol{X}^b)_{\beta\epsilon}^{-1} \right) [\boldsymbol{F}_j^b]_{\gamma\epsilon}.$$

In order to employ the above formula, we need to perform the following preprocessing only once after loading the problem into the memory space.

Preprocessing for the computation of the SCM

Step 1: Count the number f_k^b of nonzero elements in $\boldsymbol{F}_k^b$ $(k = 1, 2, \ldots, m,\ b = 1, 2, \ldots, \ell)$.

Step 2: Assume that we use only one formula when computing each row of $\boldsymbol{B}$. Estimate the computational costs of formula $\mathcal{F}$-1, $\mathcal{F}$-2 and $\mathcal{F}$-3 according to f_k^b's and determine which formula we use for each row of $\boldsymbol{B}$.

Successful numerical results have been reported on these implementations for SDPA [11, 12, 40]. These formula have been also adopted in CSDP [5] and SDPT3 [34].

Note that $\boldsymbol{B}$ is a symmetric matrix and we compute only the upper (lower) triangular part of it. In SDPA 7.3.1, each row of $\boldsymbol{B}$ is assigned to a single thread, in other words, each row is neither divided into several parts nor assigned to multiple threads. Algorithm 2.3 shows a pseudo-code for the multi-thread computation of the SCM $\boldsymbol{B}$ employed in SDPA 7.3.1.

Algorithm 2.3 (Multi-threading of the SCM in SDPA 7.3.1)

$\boldsymbol{B} = \boldsymbol{O}$,
generate p threads
for $b = 1, 2, \ldots, \ell$
for $i = 1, 2, \ldots, m$
wait until finding an idle thread
for $j \in \{k \mid \boldsymbol{F}_k^b \neq \boldsymbol{O}\}$
compute B_{ij} by using the formula $\mathcal{F}$-1, $\mathcal{F}$-2 or $\mathcal{F}$-3 on the idle thread
end
end
end
terminate all threads

Note that p is the maximum number of available threads, and m is the number of constraints. If a thread is idle, which means that no row of $\boldsymbol{B}$ is assigned to the thread yet or the computation of the last assigned row has already finished, we assign a new row to the idle thread. Therefore this algorithm also has the function of automatic load balancing between multiple CPU cores. When we use a quad-core CPU and generate four threads, we can expect that the maximum speed-up of four is attained.

A different multi-thread computation can be obtained almost immediately if we use correctly the recent numerical linear algebra libraries. SDPA 7.3.1 usually utilizes optimized BLAS. BLAS (Basic Linear Algebra Subprograms) library provides standard interface for performing basic vector and matrix operations. We highly recommend to use optimized BLAS libraries: Automatically Tuned Linear Algebra Software (ATLAS), GotoBLAS and Intel Math Kernel Library (MKL). For example, the problem maxG32 in Table 5 in Section 3.1 takes 1053.2 seconds using BLAS/LAPACK 3.2.1, but only 85.7 seconds using GotoBLAS 1.34.

The formula $\mathcal{F}$-1 contains two matrix-matrix multiplications, one is the multiplication of the dense matrix $\boldsymbol{Y}$ and the sparse matrix $\boldsymbol{F}_i$, the other is the multiplication of two dense matrices $(\boldsymbol{Y}\boldsymbol{F}_i)$ and $\boldsymbol{X}^{-1}$. The latter can be accelerated by utilizing an optimized BLAS library. GotoBLAS is a highly optimized BLAS library and it assumes that it can occupy all CPU cores and cashes especially when computing the multiplication of two dense matrices. On the other hand, $\mathcal{F}$-3 formula frequently accesses CPU cashes. Therefore, a simultaneous computation of $\mathcal{F}$-1 and $\mathcal{F}$-3 may cause serious resource conflicts, which dramatically lowers the performance of the multi-threading for the SCM. However, we can avoid these kinds of serious resource conflicts. If one thread starts the computation of $\mathcal{F}$-1, we suspend all other threads until the thread finishes $\mathcal{F}$-1. SDPA 7.3.1 is carefully implemented so that it can derive better performance from multi-threading.

3. Numerical experiments of SDPA 7.3.1

In this section, we evaluate the performance of SDPA 7.3.1 with three existing software packages: CSDP 6.0.1 [5], SDPT3 4.0β [34], and SeDuMi 1.21 [32]. Through these numerical experiments, we can conclude that SDPA 7.3.1 is the fastest general-purpose software package for SDPs. Then, in Section 3.2, we discuss how the new features described in Section 2 affect these results and why SDPA 7.3.1 is so efficient compared to other software packages.

The SDPs we used for the performance evaluation were selected from the SDPLIB [6], the 7th DIMACS Implementation Challenge [23], Mittelmann's benchmark problems [25],

SDP relaxation problems of polynomial optimization problems generated by SparsePOP [36], SDPs arising from quantum chemistry [31] and sensor network problems generated by SFSDP [17]. Table 3 shows some definitions and terms used to explain the performance of the SDPA 7.3.1.

Table 3: Definitions on numerical experiments.

m	number of constraints
nBLOCK	number of blocks [10]
bLOCKsTRUCT	block structure vector [10]
n	total matrix size
ELEMENTS	computation of all elements of the SCM: $\mathcal{O}(mn^3 + m^2n^2)$
CHOLESKY	Cholesky factorization of the SCM: $\mathcal{O}(m^3)$
OTHERS	other matrix-matrix multiplications: $\mathcal{O}(n^3)$

Table 4 shows the sizes of mostly large problems among the SDPs we solved. In the "bLOCKsTRUCT" column, a positive number means the size of an SDP block, while a negative number indicates the size of a diagonal (LP) block. For example, "11 $\times$4966" means there are 4966 matrices of size 11×11 and "-6606" means we have 6606 matrices of size 1×1 as one diagonal block.

All numerical experiments of this section were performed on the following environment:

CPU	:	Intel Xeon X5550 2.67GHz (2 CPUs, 8 total cores)
Memory	:	72GB
OS	:	Fedora 11 64bit Linux
Compiler	:	gcc/g++/gfortran 4.4.1
MATLAB version	:	7.8.0.347 (R2009a)
Numerical Libraries	:	GotoBLAS 1.34, MUMPS 4.9 [1, 2, 3] (for SDPA).

3.1. Comparison with other software packages on benchmark problems

Table 5 shows the CPU time in seconds and the number of iterations required for each software package on SDPLIB and DIMACS problems. We set a running time limit of one hour.

For all software packages, we use their own default parameters. For instance, SDPA 7.3.1 decides that the optimal solution is obtained when the relative gap and feasibility errors become smaller than 1.0×10^{-7}. See [10] for details. In addition, we set the number of available threads as 8. The number of threads affects only optimized BLAS for software packages, except SDPA 7.3.1. In SDPA 7.3.1, the multiple-threads are used not only by optimized BLAS but also for parallel evaluation of the SCM.

In the 26 solved problems in Table 5 , SDPA 7.3.1 was slower only on 7 problems than CSDP 6.0.1, and on 4 problems than SDPT3 4.0β. Compared with SeDuMi 1.21, in most cases, it was at least 2.4 times faster.

Table 6 shows the results on Mittelmann's problems and polynomial optimization problems. We set the running time limit to one hour or two hours in this case. Again, among the 33 solved problems, SDPA 7.3.1 was slower only on 4 problems than SDPT3 4.0β, and on 2 problems than SeDuMi 1.21. CSDP 6.0.1 was extremely slow specially for the BroydenTri problems.

Table 4: Sizes of some large SDPs in Table 5,6, and 8.

name	m	nBLOCK	bLOCKsTRUCT
		SDPLIB [6]	
control11	1596	2	(110, 55)
maxG60	7000	1	(7000)
qpG51	1000	1	(2000)
thetaG51	6910	1	(1001)
		DIMACS Challenge [23]	
hamming_10_2	23041	1	(1024)
		Mittelmann's Problem [25]	
mater-6	20463	4968	(11 ×4966, 1 ×2)
vibra5	3280	3	(1760, 1761, -3280)
ice_2.0	8113	1	(8113)
p_auss2_3.0	9115	1	(9115)
rabmo	5004	2	(220 -6606)
		Polynomial Optimization Problem [36]	
BroydenTri800	15974	800	(10 ×798, 4 ×1, -3)
BroydenTri900	17974	900	(10 ×898, 4 ×1, -3)
		Quantum Chemistry [31]	
NH3+.2A2".STO6G.pqgt1t2p	2964	22	(744 ×2, 224 ×4, ..., -154)
Be.1S.SV.pqgt1t2p	4743	22	(1062 ×2, 324 ×4, ..., -190)
		Sensor Network Location Problem [17]	
d2s4Kn0r01a4	31630	3885	(43 ×2, 36 ×1, ..., -31392)
s5000n0r05g2FD_R	33061	4631	(73 ×1, 65 ×1, 64 ×2, ...)

Now, we move our concerns to numerical stability and memory consumption. As Table 8 indicates, SDPA 7.3.1 is the fastest code which also has numerical stability and low consumption of memory. This table gives comparative numerical results for the quantum chemistry (first two problems) [31] and sensor network problems (last two problems) [17]. It lists the computational time, number of iterations, memory usage and the DIMACS errors, standardized at the 7th DIMACS Implementation Challenge [23] as specified in Table 7.

SDPA 7.3.1 is at least 2 times faster than SeDuMi 1.21, 7 times faster than CSDP 6.0.1 or SDPT3 4.0β. In the particular case of "d2s4Kn0r01a4", SDPA 7.3.1 is 100 times faster than SDPT3 4.0β, and 110 times faster than CSDP 6.0.1. SDPA 7.3.1 uses more memory than CSDP 6.0.1 for the quantum chemistry problems, because it momentarily allocates memory space for the intermediate matrices $\boldsymbol{Y}\boldsymbol{F}_i\boldsymbol{X}^{-1}$ in $\mathcal{F}$-1 and $\mathcal{F}$-2 formula by multi-thread computation (Section 2.3). Both SDPT3 4.0β and SeDuMi 1.21 use at least 3 times more memory than SDPA 7.3.1 due to the overhead of MATLAB. SDPT3 4.0β has a prohibitive memory usage for the sensor network problems. Finally, in terms of accuracy, SDPA 7.3.1 and SeDuMi 1.21 produce competing accuracy in (primal-dual) feasibility and duality gaps. SDPT3 4.0β comes next and CSDP 6.0.1 possibly has an internal bug for the very last iteration before reporting the DIMACS errors for the last two problems.

The above tables lead us to conclude that SDPA 7.3.1 attains the highest performance among the four representative software packages for most SDPs.

Table 5: Computational results for large instances from SDPLIB and DIMACS problems. CPU time in seconds and number of iterations. Time limit of one hour.

problem	SDPA 7.3.1	CSDP 6.0.1	SDPT3 4.0β	SeDuMi 1.21
		SDPLIB		
arch8	**0.8(25)**	0.9(25)	2.6(25)	3.4(28)
control11	**35.9(47)**	49.4(26)	52.8(26)	86.9(45)
equalG11	**6.8(17)**	16.6(22)	16.9(17)	150.3(16)
equalG51	**16.0(23)**	39.0(29)	30.5(18)	676.0(30)
gpp500-1	**2.3(19)**	8.2(36)	6.6(21)	115.2(40)
gpp500-4	**3.0(19)**	4.0(21)	5.8(17)	81.5(25)
maxG11	**6.6(16)**	7.1(16)	7.6(15)	92.9(13)
maxG32	85.7(17)	63.2(17)	**62.0(15)**	3094.6(14)
maxG51	**8.6(16)**	12.7(17)	19.0(17)	227.7(16)
maxG55	**578.1(17)**	643.5(18)	1120.7(17)	≥ 3600
maxG60	1483.8(17)	**1445.5(18)**	2411.23(16)	≥ 3600
mcp500-1	**1.5(16)**	1.7(16)	2.6(15)	30.2(16)
mcp500-4	1.8(15)	**1.6(15)**	3.4(13)	27.9(14)
qpG11	28.5(16)	33.5(17)	**8.4(15)**	1275.5(14)
qpG51	52.2(19)	75.9(20)	**19.6(17)**	3894.4(22)
ss30	4.3(22)	**3.8(22)**	8.8(21)	20.6(23)
theta5	**7.3(18)**	8.3(17)	10.4(14)	112.3(16)
theta6	**14.2(18)**	19.4(17)	23.8(14)	326.7(16)
thetaG11	**12.3(22)**	23.6(23)	26.7(18)	156.9(15)
thetaG51	**86.8(28)**	256.3(35)	517.5(37)	2579.1(19)
truss8	1.4(20)	**1.0(20)**	2.1(16)	2.2(23)
		DIMACS		
torusg3-15	464.3(24)	**249.7(18)**	325.2(16)	≥ 3600
filter48_socp	29.0(35)	**18.6(48)**	68.2(40)	77.0(31)
copo23	**17.7(24)**	48.6(23)	47.9(20)	92.2(16)
hamming_10_2	**1105.2(18)**	1511.4(18)	1183.2(10)	≥ 3600
hamming_8_3_4	**334.7(15)**	423.9(14)	387.3(9)	≥ 3600

In the next subsection, we investigate how SDPA 7.3.1 can produce such performance based on the new features discussed in Section 2.

3.2. The effect of the new features

To assess the effects of the new features, we first remark that the computation of each iteration of SDPA 7.3.1 can be divided into three major parts (ELEMENTS, CHOLESKY, and OTHERS) as listed in Table 3.

In Table 9, we compare the computational results for some problems in Tables 5 and 6 using SDPA 6.2.1 and SDPA 7.3.1.

The computation time spent by ELEMENTS in the problem mater-5 is shortened from 838.5s to 13.0s. An important property of mater-5 is that it has many small blocks, 2438 matrices of size 11×11. The numbers $m_i = \#\{b | \boldsymbol{F}_i^b \neq \boldsymbol{O}\}$ of non-zero submatrices in $\boldsymbol{F}_1, \ldots, \boldsymbol{F}_m$ are 2438 (full) in only 3 matrices $\boldsymbol{F}_1, \boldsymbol{F}_2, \boldsymbol{F}_3$ among all $m = 10143$ matrices, while at most 6 in other 10140 matrices. In particular, for 8452 matrices, the number is only 4. Therefore, the impact of new data structure introduced at the first half of Section 2.1

Table 6: Computational results for large instances from Mittelmann's problems and polynomial optimization problems. CPU time in seconds and number of iterations. Time limit of one or two hours.

problem	SDPA 7.3.1	CSDP 6.0.1	SDPT3 4.0β	SeDuMi 1.21
		Mittelmann's Problems		
mater-4	**6.1(28)**	36.9(27)	29.9(28)	32.2(26)
mater-5	**14.7(30)**	237.9(29)	72.0(29)	84.3(28)
mater-6	**34.8(36)**	1673.2(33)	160.0(28)	187.9(31)
trto3	**1.7(21)**	5.0(37)	4.1(25)	44.2(59)
trto4	**7.1(23)**	29.3(39)	25.1(33)	501.1(73)
trto5	**59.8(22)**	737.3(57)	188.1(29)	$\geq$ 3600
vibra3	**4.7(32)**	10.1(42)	10.0(32)	100.8(69)
vibra4	**19.5(35)**	74.5(50)	68.9(46)	1043.5(95)
vibra5	**194.0(40)**	1452.2(61)	788.5(63)	$\geq$ 3600
neosfbr20	**143.6(25)**	239.5(25)	530.2(23)	$\geq$ 3600
cnhil8	4.3(20)	5.6(19)	**3.6(24)**	28.2(18)
cnhil10	29.6(22)	73.3(21)	**26.9(23)**	627.2(18)
G59mc	**600.7(18)**	860.4(20)	1284.5(19)	$\geq$ 3600
neu3	**297.9(39)**	2357.5(68)	346.3(76)	2567.4(24)
neu3g	**341.3(37)**	3182.2(65)	358.9(50)	3365.8(25)
rendl1_600_0	**5.3(30)**	7.5(20)	9.7(25)	105.3(28)
sensor_1000	**61.4(33)**	236.1(61)	783.8(33)	$\geq$ 3600
taha1b	**164.6(33)**	589.7(47)	704.5(34)	$\geq$ 3600
yalsdp	**103.9(18)**	318.3(16)	237.5(13)	678.6(16)
checker_1.5	**414.6(23)**	773.0(29)	476.9(23)	$\geq$ 3600
foot	**169.6(37)**	373.2(35)	262.6(24)	$\geq$ 3600
hand	**27.2(21)**	97.1(20)	53.8(17)	767.9(18)
ice_2.0	4852.7(26)	$\geq$ 3600	**3784.8(27)**	$\geq$ 3600
p_auss2_3.0	6065.1(24)	$\geq$ 3600	**5218.7(27)**	$\geq$ 3600
rabmo	**26.8(21)**	121.0(28)	316.3(53)	681.2(21)
reimer5	**179.2(20)**	4360.7(68)	2382.9(31)	1403.1(16)
chs_500	**5.3(28)**	$\geq$ 3600	12.1(23)	8.3(22)
nonc_500	3.0(28)	846.6(31)	5.9(27)	**2.1(22)**
ros_500	2.8(22)	806.3(23)	4.3(17)	**1.6(17)**
		Polynomial Optimization Problems		
BroydenTri600	**2.5(20)**	$\geq$ 3600	12.2(18)	8.7(14)
BroydenTri700	**3.1(21)**	$\geq$ 3600	14.6(18)	10.8(15)
BroydenTri800	**3.4(20)**	$\geq$ 3600	16.8(18)	12.7(14)
BroydenTri900	**3.8(20)**	$\geq$ 3600	19.6(18)	15.3(14)

can be observed prominently for mater-5.

The memory management for dense variable matrices discussed in the latter half of Section 2.1 can be monitored in maxG60. The size of the variable matrices is 7000×7000. Since the memory space of the dense matrices is reduced from 31 to 15, this should save approximately 5.8 GB of memory space. The result for maxG60 in Table 9 is consistent with this estimation.

Table 7: Standardized DIMACS errors for the SDP software outputs [23] .

Error 1	$\dfrac{\sqrt{\sum_{k=1}^{m}(\boldsymbol{F}_k \bullet \boldsymbol{Y} - c_k)^2}}{1+\max\{\lvert c_k\rvert \ : \ k=1,2,\ldots,m\}}$
Error 2	$\max\left\{0, \dfrac{\lambda_{\min}(\boldsymbol{Y})}{1+\max\{\lvert c_k\rvert \ : \ k=1,2,\ldots,m\}}\right\}$
Error 3	$\dfrac{\lVert \boldsymbol{X} - \sum_{k=1}^{m}\boldsymbol{F}_k x_k + \boldsymbol{F}_0\rVert_f}{1+\max\{\lvert[\boldsymbol{F}_0]_{ij}\rvert \ : \ i,j=1,2,\ldots,n\}}$
Error 4	$\max\left\{0, \dfrac{\lambda_{\min}(\boldsymbol{X})}{1+\max\{\lvert c_k\rvert \ : \ k=1,2,\ldots,m\}}\right\}$
Error 5	$\frac{\sum_{k=1}^{m} c_k x_k - \boldsymbol{F}_0 \bullet \boldsymbol{Y}}{1+\lvert\sum_{k=1}^{m} c_k x_k\rvert + \lvert\boldsymbol{F}_0 \bullet \boldsymbol{Y}\rvert}$
Error 6	$\dfrac{\boldsymbol{X} \bullet \boldsymbol{Y}}{1+\lvert\sum_{k=1}^{m} c_k x_k\rvert + \lvert\boldsymbol{F}_0 \bullet \boldsymbol{Y}\rvert}$

$\|\cdot\|_f$ is the norm defined as the sum of Frobenius norms of each block diagonal matrix. $\lambda_{\min}(\cdot)$ is the smallest eigenvalue of the matrix.

In BroydenTri600, the computation time on CHOLESKY is greatly shortened. This is brought by the sparse Cholesky factorization of Section 2.2. Figure 3.2 displays the non-zero patterns of the upper triangular part of the SCM; its density is only 0.40%. From the figure, it is apparent that we should apply the sparse Cholesky factorization instead the dense factorization. We note that the reason why CSDP is very slow for polynomial optimization problems and sensor network problems in Tables 6 and 8 comes from the dense factorizations. CSDP always applies the dense factorization without regarding the sparsity of the SCM. SDPT3 first stores the SCM in the fully-dense structure and then apply the sparse Cholesky factorization. Hence, SDPT3 requires a dreadfully large memory space for sensor network problems.

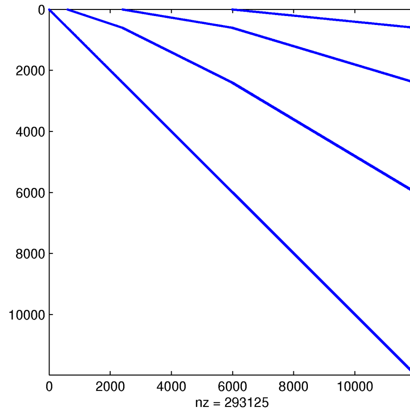

Figure 1: Non-zero patterns of the SCM generated from BroydenTri600.

Finally, Table 10 shows the efficiency of the multi-thread computation (Section 2.3). In particular, in the 3rd line of each instance of the SDP, we generated 8 threads for the computation of the SCM $\boldsymbol{B}$, and 8 threads for the optimized BLAS, GotoBLAS. As it is clear from

Table 8: Comparative results of time, memory usage, and DIMACS errors for the quantum chemistry and sensor network problems when solving by four SDP packages.

problem		SDPA 7.3.1	CSDP 6.0.1	SDPT3 4.0β	SeDuMi 1.21
NH3+.2A2".STO6G	time(#iter.)	**495.3(31)**	**5675.6(51)**	**4882.9(40)**	**1357.1(30)**
.pqgt1t2p	memory (bytes)	1004M	568M	3676M	4065M
	Error 1	2.87e−10	1.35e−09	9.27e−07	6.98e−10
	Error 2	0.00e+00	0.00e+00	0.00e+00	0.00E+00
	Error 3	2.90e−08	9.78e−10	1.26e−11	0.00E+00
	Error 4	0.00e+00	0.00e+00	0.00e+00	2.43e−13
	Error 5	4.39e−09	7.33e−09	6.64e−08	8.45e−09
	Error 6	2.60e−08	8.19e−09	9.27e−10	1.16e−08
Be.1S.SV.pqgt1t2p	time(#iter.)	**2238.3(38)**	**15592.3(40)**	**15513.8(37)**	**5550.4(38)**
	memory (bytes)	1253M	744M	3723M	4723M
	Error 1	1.95e−09	8.41e−11	5.49e−05	9.74e−10
	Error 2	0.00e+00	0.00e+00	0.00e+00	0.00E+00
	Error 3	6.31e−08	8.44e−10	3.95e−07	0.00E+00
	Error 4	0.00e+00	0.00e+00	0.00e+00	3.37e−14
	Error 5	7.09e−09	4.60e−09	6.21e−05	2.02e−08
	Error 6	2.87e−08	4.84e−09	6.92e−05	2.39e−08
d2s4Kn0r01a4	time(#iter.)	**45.7(35)**	**5162.4(27)**	**4900.3(53)**	**92.6(42)**
	memory (bytes)	1093M	8006M	63181M	3254M
	Error 1	5.07e−14	5.64e−37	3.13e−05	9.12e−11
	Error 2	0.00e+00	0.00e+00	0.00e+00	0.00E+00
	Error 3	5.67e−06	6.13e+03	9.55e−09	0.00E+00
	Error 4	0.00e+00	0.00e+00	0.00e+00	1.76e−12
	Error 5	1.67e−08	7.42e−54	2.11e−05	4.36e−10
	Error 6	1.29e−07	2.17e−36	1.96e−05	1.36e−09
s5000n0r05g2FD_R	time(#iter.)	**284.7(37)**	**6510.9(31)**	**4601.6(37)**	**1005.0(62)**
	memory (bytes)	2127M	8730M	100762M	4914M
	Error 1	1.03e−14	2.23e−37	1.91e−05	1.65e−10
	Error 2	0.00e+00	0.00e+00	0.00e+00	0.00E+00
	Error 3	3.95e−06	1.14e+04	1.33e−11	0.00E+00
	Error 4	0.00e+00	0.00e+00	0.00e+00	1.52e−13
	Error 5	2.97e−08	9.30e−37	3.50e−04	7.06e−10
	Error 6	1.21e−07	2.82e−36	6.32e−04	2.12e−09

Table 9: Comparison between SDPA 6.2.1 and SDPA 7.3.1. CPU time in seconds and memory space in bytes.

problem	version	ELEMENTS	CHOLESKY	OTHERS	Total	memory
mater-5	SDPA 6.2.1	838.5	1101.6	6056.9	7997.0	2786M
	SDPA 7.3.1	13.0	2.6	6.0	21.6	885M
maxG60	SDPA 6.2.1	27.4	193.4	11607.0	11827.9	12285M
	SDPA 7.3.1	46.8	27.5	1358.7	1433.1	6124M
BroydenTri600	SDPA 6.2.1	200.3	31147.8	1208.9	32557.0	1694M
	SDPA 7.3.1	3.3	0.8	1.3	5.4	794M

the table, increasing the threads from 1 to 8 for the optimized BLAS, reduces the computation time of CHOLESKY and OTHERS. The computation of ELEMENTS needs an extra comment. For example, in control11, we need to employ the $\mathcal{F}$-1 formula for some constraint matrices $\boldsymbol{F}_k$. Without the thread management discussed in Section 2.3, a conflict of mem-

ory access between threads would lower the parallel efficiency in ELEMENTS. When the number of constraints m is large, this multi-threaded parallel computation for ELEMENTS works well. Since most computation time is devoted to ELEMENTS in most problems of SDPLIB and Mittelmann's problems, we can expect that SDPA 7.3.1 with multi-threading solves them in the shortest time. As pointed in Section 3.1, however, SDPA 7.3.1 with multi-threading consumes slightly more memory space.

Table 10: Computational time (in seconds) of each major part of SDPA 7.3.1 when changing the number of threads when computing the SCM (ELEMENTS) and using the GotoBLAS.

problem	threads SCM $\boldsymbol{B}$	threads GotoBLAS	ELEMENTS	CHOLESKY	OTHERS	Total
control11	1	1	77.3	6.8	0.3	86.1
	1	8	64.6	1.3	0.3	68.0
	8	8	29.6	4.1	0.2	35.9
thetaG51	1	1	134.7	308.6	46.0	494.9
	1	8	136.2	43.6	7.7	193.9
	8	8	26.2	45.8	7.6	86.8
rabmo	1	1	56.3	88.1	0.4	146.6
	1	8	59.9	13.4	0.1	75.2
	8	8	11.2	13.5	0.1	26.8

In the SDPs of this subsection, we could describe the effects of new features. For general SDPs, however, it is difficult to measure the effects separately, since the total computation time is usually a combination of them. For example, if an SDP has many small blocks and $\bar{m}$ (the maximum number of nonzero blocks in each input matrices) is relatively small, then both the new data structure and the sparse Cholesky factorization will be beneficial. Consequently, the high performance of SDPA 7.3.1 in Section 3.1 is attained by these combined effects.

4. Ultra high accurate versions of SDPA: SDPA-GMP, SDPA-QD, and SDPA-DD

One necessity for obtaining high accurate solutions of SDPs arises from quantum physical/chemical systems [31]. The PDIPM, which is regarded as a high precision method for solving SDPs, can typically provide only seven to eight significant digits of certainty for the optimal values. This is only sufficient for very small systems, and not sufficient for highly correlated, degenerated and/or larger systems in general. For an ill-behaved SDP instance, the accuracy which an PDIPM can attain is restricted to only three or less digits. [9] lists some other SDP instances which require accurate calculations.

There are mainly two reasons we face numerical difficulties when solving numerically an SDP. First, to implement a PDIPM as a computer software, the real number arithmetic is usually replaced by the IEEE 754 double precision having a limited precision; approximately 16 significant digits. Therefore, errors accumulated over the iterations often generate inaccurate search directions, producing points which do not satisfy the constraints accurately. The accumulated errors also cause a premature breakdown of the PDIPM bringing a disaster to the Cholesky factorization, since the SCM becomes ill-conditioned near an optimal

solution and hence very sensitive to even a subtle numerical error. Second, we may be solving a problem which is not guaranteed to be theoretically solvable by PDIPM because it may not converge correctly. That is the case when a problem does not satisfy the Slater condition or the linear independence of matrix constraints.

To resolve mainly the first difficulty in a practical way, we have developed new multiple-precision arithmetic versions of SDPA: SDPA-GMP, SDPA-QD and SDPA-DD by implementing a multiple-precision linear algebra library named MPACK [30]. The difference between these three software packages will be discussed later. The three software packages carry PDIPM out with higher accuracy provided via MPACK, and can solve some difficult problems with an amazing accuracy as shown in numerical results of Section 4.3.

4.1. Multi-precision BLAS and LAPACK libraries: MPACK

We have implemented MPACK for a general purpose use on linear algebra. It is composed of MBLAS and MLAPACK (multiple-precision versions of BLAS and LAPACK, respectively). Their interface looks similar to BLAS and LAPACK. The naming rule has been changed for individual routines: prefixes `s,d` or `c,z` in BLAS and LAPACK have been replaced with `R` and `C`, *e.g*, the matrix-matrix multiplication routine `dgemm` → `Rgemm`.

The numerical accuracy of MPACK is determined by the base arithmetic library. MPACK currently supports three arithmetic libraries, the GNU Multi-Precision Arithmetic (GMP) Library, Double-Double (DD) and Quad-Double (QD) Arithmetic Library [15]. GMP is a package which can handle numbers with arbitrary significant bits and DD and QD libraries support approximately 32 and 64 significant digits, respectively.

MPACK is now freely available under LGPL and a subset version of MPACK is included in SDPA-GMP/-QD/-DD for convenience. Although the first motivation of MPACK was SDPA-GMP, MPACK has a considerable potential of becoming a powerful tool in scientific areas which demand high accuracy in basic linear algebras.

4.2. SDPA-GMP/-QD/-DD

As the names indicate, SDPA-GMP/-QD/-DD utilize MPACK with GMP, QD and DD, respectively, instead of BLAS/LAPACK. Owing to the development of MPACK, we can keep minimal the difference between the source codes of SDPA and SDPA-GMP/-QD/-DD.

Using different arithmetic libraries affects not only the numerical accuracy but also their computation time. From our experience, SDPA-GMP is the slowest, and SDPA-QD is from 1.2 to 2 times faster than SDPA-GMP. Finally SDPA-DD is approximately 10 times faster than SDPA-GMP. Roughly speaking, the SDPA-GMP calculations take approximately 100 times to 1000 times longer than SDPA's ones. Exemplifying, the problem Truss1002_no_blocks took the longest time to solve in Section 4.3; about 15.5 days.

Even though SDPA-GMP is very slow (a study for reducing its calculation time is ongoing), its accuracy has resulted in interesting researches. SDPA-GMP was first used in the quantum physical/chemical systems by Nakata *et al.* [31]. In Waki *et al.*'s study [37], they used approximately 900 significant digits to apply the PDIPM to SDPs which do not satisfy the Slater condition. SDPA-GMP occasionally can solve such problems, too.

4.3. Numerical Results

There are two sets of problems we solved. Both sets can not be solved by other existing SDP codes such as SDPA and SeDuMi.

The first set of SDPs is from quantum physics/chemistry: the Hubbard model at the strong correlation limit [31]. In our example, we calculated the one-dimensional and

nearest neighbor hopping with periodic boundary condition, half-filled case, S^2=0 and $U/t = \infty, 100000, 10000, 1000$ by the RDM method. The number of the electrons is varied from 6 to 10. We used the P, Q, G conditions or P, Q, G, and $T2'$ conditions as the N-representability conditions. The inclusion of the G condition produces an exact result for the infinite U/t, since the G condition includes the electron-electron repulsion part of the Hubbard Hamiltonian. In this case, the optimal value (the ground state energy) should be zero. When U/t is large, the corresponding SDP becomes extremely difficult to solve. Note that these problems are nearly quantum physically degenerated. There are 5, 14 and 42 degeneracies for the 6, 8 and 10 sites/electrons systems, respectively. This degeneracy is considered to be one reason of numerical difficulties for SDP codes. As a reference, the exact ground state energy for 6 and 8 electrons for $U/t = 10,000$ by solving the Schrödinger equation are -1.721110×10^{-3} and -2.260436×10^{-3}, respectively.

The second set consists of some numerically difficult instances from [9]. We used several software packages and computers to solve these problems. Since Mittelmann *et al.* [24] and de Klerk *et al.* [9] had already employed SDPA-GMP to solve 17 problems of [9], we solved only the larger problems here.

Table 11 shows the sizes of problems solved by SDPA-GMP or SDPA-QD. Table 12 shows the optimal values and elapsing times. Finally, Table 13 gives the relative duality gaps, and the primal and dual feasibility errors [10]. The significant digits of SDPA-GMP and for SDPA-QD are approximately 77 and 63, and their machine epsilons are 8.6e-78 and 1.2e-63, respectively. As described above, the optimal values of Hubbard models are 0 for the case $U/t = \infty$ due to the chemical property of the G condition. Therefore, the optimal values obtained by SDPA-GMP in Table 12 attain the accuracy of the order 1.0×10^{-30}. In addition, it is predicted that the optimal values of the Hubbard models would be magnified almost proportional to U/t. More precisely, if the optimal value is -1.72×10^{-4} for $U/t = 100,000$, then we should approximately obtain -1.72×10^{-3} for $U/t = 10,000$, and -1.72×10^{-2} for $U/t = 1,000$. The results by SDPA-GMP reproduced this asymptotic behavior very well.

Table 14 shows the comparison between the objective values reported in [9] and SDPA-GMP/-QD[1]. For all problems in this table, SDPA-GMP/-QD successfully updated the accuracy. Even though the computation time of Truss1002_no_blocks was very long as 15 days, SDPA-GMP with some appropriate parameters could update the significant digits from 3 to 14.

From Table 13, we conclude that the relative duality gaps are very small, and almost half of them are smaller than 1e-75. Even for the largest ones such as Truss502_no_blocks (7.6e-16) and Truss1002_no_blocks (5.0e-16), their feasibility errors are both very small, very close to the machine epsilon: 1e-78 in this case.

At the end of this section, we should point out that a simple comparison of these results might be difficult. This is because other software packages (including CSDP, SDPT3 and SeDuMi) can not solve these problems, and we can not compare directly these values with them. In addition, the stopping criteria of SDPA-GMP/-QD/-DD are based on the KKT condition. For example, the dual feasibility is checked by $\max_{k=1,2,\ldots,m}\{|\boldsymbol{F}_k \bullet \boldsymbol{Y} - c_k|\} < \epsilon$ with a given parameter ϵ. Even if the current point satisfies the dual and primal feasibility and duality gap with $\epsilon = 10^{-30}$, it might not be an optimal solution. This is the case of

[1]The results shown at the web site `http://lyrawww.uvt.nl/~sotirovr/library/` of [9] have already been updated according to our results.

Table 11: The sizes of SDP instances. "*" should be replaced by Inf, 100000, 10000, or 1000.

problems	m	nBLOCK	bLOCKsTRUCT
hubbard_X6N6p.*.pqg	948	13	(72, 36 ×4, 15 ×4, 6 ×4, -94)
hubbard_X6N6p.*.pqgt1t2p	948	21	(312 ×2, 90 ×4, 72, 36 ×4, 20 ×2, 15 ×4, 6 ×4, -94)
hubbard_X8N8p.*.pqg	2964	13	(128, 64 ×4, 28 ×4, 8 ×4, -154)
hubbard_X8N8p.*.pqgt1t2p	2964	21	(744 ×2, 224 ×4, 128, 64 ×4, 56 ×2, 28 ×4, 8 ×4, -154)
hubbard_X10N10p.*.pqg	7230	14	(200, 100 ×4, 45 ×4, 10 ×4, -230)
QAP_Esc64a_red	517	7	(65 ×7, -5128)
Schrijver_A(37,15)	468	1327	many small blocks of size 1 to 38
Laurent_A(50,15)	2057	6016	many small blocks of size 1 to 52
Laurent_A(50,23)	607	1754	many small blocks of size 1 to 52
TSPbays29	6090	14	(29 ×14, -13456)
Truss502_no_blocks	3	1	(1509, -3)
Truss1002_no_blocks	3	1	(3009, -3)

Table 12: Optimal values and elapsed times of difficult SDPs solved by SDPA-GMP or SDPA-QD. The computers used were `(a) AMD Opteron 250`, and `(b) Intel Xeon X5365`.

problems	software	optimal value	time (s)	computer
hubbard_X6N6p.Inf.pqg	SDPA-GMP	-8.4022210139931402e-31	225	a
hubbard_X6N6p.100000.pqg	SDPA-GMP	-2.1353988200647472e-04	2349	a
hubbard_X6N6p.10000.pqg	SDPA-GMP	-2.1353985768649223e-03	2193	a
hubbard_X6N6p.1000.pqg	SDPA-GMP	-2.1353742577700272e-02	2138	a
hubbard_X6N6p.Inf.pqgt1t2p	SDPA-GMP	-2.9537164216756805e-30	23223	a
hubbard_X6N6p.100000.pqgt1t2p	SDPA-GMP	-1.7249397045806836e-04	80211	a
hubbard_X6N6p.10000.pqgt1t2p	SDPA-GMP	-1.7249951195749524e-03	80195	a
hubbard_X6N6p.1000.pqgt1t2p	SDPA-GMP	-1.7255360310431303e-02	81396	a
hubbard_X8N8p.Inf.pqgt1t2p	SDPA-QD	-4.2783999570339451e-30	594446	b
hubbard_X8N8p.100000.pqgt1t2p	SDPA-QD	-2.2675986731298406e-04	754492	b
hubbard_X8N8p.10000.pqgt1t2p	SDPA-QD	-2.2676738944543175e-03	772093	b
hubbard_X8N8p.1000.pqgt1t2p	SDPA-QD	-2.2684122478330654e-02	781767	b
hubbard_X10N10p.Inf.pqg	SDPA-GMP	-5.6274162400011421e-30	18452	a
QAP_Esc64a_red	SDPA-QD	9.7750000000000000e+01	1581	b
Schrijver_A(37,15)	SDPA-QD	-1.4069999999999891e+03	2099	b
Laurent_A(50,15)	SDPA-QD	-1.9712600652510334e-09	63090	b
Laurent_A(50,23)	SDPA-QD	-2.5985639398573229e-13	6729	b
TSPbays29	SDPA-GMP	1.9997655161769048e+03	525689	b
Truss502_no_blocks	SDPA-GMP	1.61523565346747e+06	155040	b
Truss1002_no_blocks	SDPA-GMP	9.8233067733903e+06	1339663	b

SDPs which do not satisfy the Slater condition. Although we can conclude the correctness of SDPA-GMP for the Hubbard model from chemical insights, more investigations on the numerical correctness for such exceptional SDPs are the subject of our future investigations.

5. Complementary extensions of SDPA

5.1. The SDPA online solver

As shown in the numerical results of Sections 3 and 4, the solvability of SDPA and SDPA-GMP is outstanding. Upgrading the software packages, however, sometimes may cause difficulties when users want to install them. For example, the performance of SDPA is

Table 13: The relative duality gap, primal and dual feasibility errors [10] by SDPA-GMP or SDPA-QD.

problems	software	relative gap	p.feas error	d.feas error
hubbard_X6N6p.Inf.pqg	SDPA-GMP	4.636e-29	9.060e-31	2.240e-75
hubbard_X6N6p.100000.pqg	SDPA-GMP	7.232e-77	3.707e-31	5.401e-48
hubbard_X6N6p.10000.pqg	SDPA-GMP	2.757e-77	1.315e-31	2.509e-49
hubbard_X6N6p.1000.pqg	SDPA-GMP	1.120e-76	1.893e-31	1.066e-50
hubbard_X6N6p.Inf.pqgt1t2p	SDPA-GMP	1.979e-32	5.896e-31	5.130e-73
hubbard_X6N6p.100000.pqgt1t2p	SDPA-GMP	1.834e-76	7.052e-31	9.596e-58
hubbard_X6N6p.10000.pqgt1t2p	SDPA-GMP	5.644e-78	3.406e-31	3.557e-62
hubbard_X6N6p.1000.pqgt1t2p	SDPA-GMP	2.866e-76	3.396e-31	1.969e-65
hubbard_X8N8p.Inf.pqgt1t2p	SDPA-QD	7.241e-66	5.987e-31	6.937e-54
hubbard_X8N8p.100000.pqgt1t2p	SDPA-QD	1.687e-60	4.633e-31	2.377e-46
hubbard_X8N8p.10000.pqgt1t2p	SDPA-QD	5.723e-61	2.5277-31	3.232e-48
hubbard_X8N8p.1000.pqgt1t2p	SDPA-QD	2.707e-62	1.704e-31	1.279e-51
hubbard_X10N10p.Inf.pqg	SDPA-GMP	1.182e-77	8.862e-31	1.821e-67
QAP_Esc64a_red	SDPA-QD	6.925e-63	3.857e-31	4.143e-42
Schrijver_A(37,15)	SDPA-QD	1.382e-65	4.826e-31	1.018e-45
Laurent_A(50,15)	SDPA-QD	1.769e-74	4.552e-31	8.151e-43
Laurent_A(50,23)	SDPA-QD	1.080e-78	6.284e-31	9.926e-39
TSPbays29	SDPA-GMP	3.455e-80	2.909e-31	4.928e-55
Truss502_no_blocks	SDPA-GMP	7.597e-16	7.122e-74	2.974e-63
Truss1002_no_blocks	SDPA-GMP	5.000e-16	2.067e-74	1.202e-60

Table 14: The optimal values reported in [9] and obtained by SDPA-GMP/-QD.

problem	optimal values [9]	optimal values [SDPA-GMP/-QD]
QAP_Esc64a_red	9.774e+01	9.7750000000000000e+01
Schrijver_A(37,15)	-14070e+03	-1.4069999999999891e+03
Laurent_A(50,15)	-6.4e-09	-1.9712600652510334e-09
Laurent_A(50,23)	-1e-11	-2.5985639398573229e-13
TSPbays29	1.9997e+03	1.9997655161769048e+03
Truss502_no_blocks	1.6152356e+06	1.61523565346747e+06
Truss1002_no_blocks	9.82e+06	9.8233067733903e+06

affected by how users compile the optimized BLAS. In addition, solving large-scale optimization problems requires a huge amount of computational power.

For these reasons, we have developed a grid portal system and provide online software services for some software packages of the SDPA Family. We call this system the *SDPA Online Solver*. In short, users first send their own SDP problems to the SDPA Online Solver from their PC via the Internet. Then the SDPA Online Solver assigns the task of solving the SDP by SDPA to the computers of the Online Solver system. Finally, the result of the SDPA execution can be browsed by the users' PC and/or can be downloaded.

Figure 2 displays the web interface of the SDPA Online Solver. It is very easy to use the SDPA Online Solver. Users who do not have time to compile SDPA or huge computer power can also gain the advantage of SDPA. One can access it from the following Web site: `http://sdpa.indsys.chuo-u.ac.jp/portal/`.

Currently, over 70 users are registered, and several thousand SDPs have been solved in the SDPA Online Solver. Among the SDPA Family, SDPA, SDPA-GMP, SDPARA and SDPARA-C are available now. SDPARA (SemiDefinite Programming Algorithm paRAllel version) [39] is a parallel version of SDPA and solves SDPs with the help of the MPI (Message Passing Interface) and ScaLAPACK (Scalable LAPACK). SDPARA-C [29] is a parallel version of the SDPA-C [28]; a variant of SDPARA which incorporates the positive definite matrix completion technique.

These two parallel software packages are designed to solve extremely large-scale SDPs and usually require huge computer power. Therefore, users who want to use especially these two software packages but not have a parallel computing environment will find the SDPA Online Solver very useful.

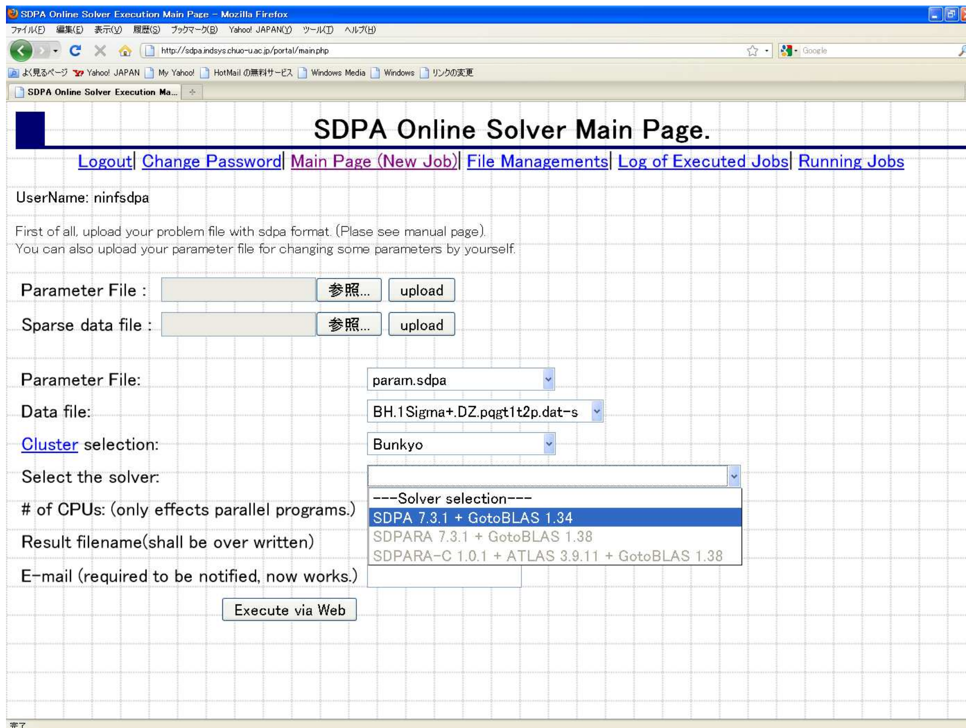

Figure 2: The SDPA Online Solver

5.2. SDPA second-order cone programs solver

Second-Order Cone Program (SOCP) is the problem of minimizing a linear function over the intersection of a direct product of several second-order cones and an affine space. Recently, SOCP is in a focus of attention because many applications can be represented as SOCPs and it has a nice structure which enables us to design the interior-point algorithm for it [21].

The second-order cones can be embedded in the cones of positive semidefinite matrices. Thus, an SOCP can be expressed as an SDP. However, it is better to use the algorithm that solves the SOCP directly because it has a better worst-case complexity than the algorithm to solve the SDP [35].

Let $\mathcal{K}_b$ $(b = 1, 2, \ldots, \ell)$ be the second-order cones defined as follows:

$$\mathcal{K}_b = \left\{ \boldsymbol{x}^b = \left(x_0^b, \boldsymbol{x}_1^b\right) \in \mathbb{R} \times \mathbb{R}^{k_b - 1} \mid \left(x_0^b\right)^2 - \boldsymbol{x}_1^b \cdot \boldsymbol{x}_1^b \geq 0, x_0^b \geq 0 \right\}.$$

Let $\boldsymbol{x} \succeq_{\mathcal{K}_b} \boldsymbol{0}$ denote that $\boldsymbol{x} \in \mathcal{K}_b$. The standard form second-order cone program and its dual are given as follows:

$$
\text{SOCP} \begin{cases}
\mathcal{P}: & \text{minimize} & \displaystyle\sum_{k=1}^{m} c_k z_k \\
& \text{subject to} & \boldsymbol{x}^b = \displaystyle\sum_{k=1}^{m} \boldsymbol{f}_k^b z_k - \boldsymbol{f}_0^b, \\
& & \boldsymbol{x}^b \succeq_{\mathcal{K}_b} \boldsymbol{0},\ \boldsymbol{x}^b \in \mathbb{R}^{k_b}\ (b = 1, 2, \ldots, \ell), \\
\mathcal{D}: & \text{maximize} & \displaystyle\sum_{b=1}^{\ell} \boldsymbol{f}_0^b \cdot \boldsymbol{y}^b \\
& \text{subject to} & \displaystyle\sum_{b=1}^{\ell} \boldsymbol{f}_k^b \cdot \boldsymbol{y}^b = c_k, \\
& & \boldsymbol{y}^b \succeq_{\mathcal{K}_b} \boldsymbol{0},\ \boldsymbol{y}^b \in \mathbb{R}^{k_b}\ (k = 1, 2, \ldots, m;\ b = 1, 2, \ldots, \ell).
\end{cases}
$$

Here, the inner product $\boldsymbol{u}\cdot\boldsymbol{v}$ for $\boldsymbol{u}$ and $\boldsymbol{v}$ in $\mathbb{R}^n$ is defined as $\boldsymbol{u}\cdot\boldsymbol{v} = \sum_{i=1}^{n} u_i v_i$. The constraint subvectors $\boldsymbol{f}_k^b$ $(b = 1, 2, \ldots, \ell)$ are specified as the diagonal submatrices $\boldsymbol{F}_k^b$ $(b = 1, 2, \ldots, \ell)$ in the SDPA format. More specifically, the ith component of $\boldsymbol{f}_k^b$ is specified by the (i, i)th component of $\boldsymbol{F}_k^b$: $[\boldsymbol{f}_k^b]_i = [\boldsymbol{F}_k^b]_{ii}$.

To solve SOCPs, we are developing another version of SDPA: SDPA-SOCP. SDPA-SOCP solves the $\mathcal{P}$-$\mathcal{D}$ pair of SOCP by the Mehrotra type predictor-corrector primal-dual interior-point method using the HKM search direction [35]. SDPA-SOCP will be embedded in SDPA and SDPA will be able to solve the linear optimization problem over a mixture of semidefinite cones, second-order cones and nonnegative orthant.

SDPA-SOCP is expected to be distributed in a near future at the SDPA project web site.

6. Concluding remarks

We described the new features implemented in SDPA 7.3.1 in detail with extensive numerical experiments. The new data structure for sparse block diagonal matrices is introduced, which leads to a significant reduction of the computation cost, and memory usage. Another important feature is the implementation of the sparse Cholesky factorization for SDPs with sparse SCM. The acceleration using multi-thread computation is employed in the computation of each row of the SCM and the matrix-matrix and matrix-vector multiplications. From the computational results, we conclude that SDPA 7.3.1 is the fastest general-purpose software for SDPs.

Ultra high accurate versions of SDPA: SDPA-GMP, SDPA-QD and SDPA-DD have been developed employing multiple-precision arithmetics. They provide highly accurate solutions for SDPs arising from quantum chemistry which require accurate solutions.

We are going to implement the next version of SDPA-C and SDPARA based on these improvements. In particular, since MUMPS can work with MPI on multiple processors, we expect SDPARA will be an extremely fast software package for SDPs with sparse SCM. Its performance can be further improved by multi-threading.

SDPA and the other codes to solve general SDPs have been refined through many years. However, we recognize that there is still ample room for future improvements. For example,

they do not have a proper preprocessing for sparse problems to accelerate the computation such as its is common for LPs. However, there exist many successful results which can be combined with these codes. More specifically, the sparsity of an SDP problem can be explored in diverse ways: matrix completion using chordal structures [17, 28], matrix representation in different spaces [16, 22], or reduction by group symmetry [8, 27]. An existing difficult is, of course, to elaborate a procedure to choose the best preprocessing for each incoming sparse problem. These studies are stimulating our motivation for the next version of SDPA, SDPA 8.

Acknowledgments

K. F.'s and M. F.'s researches were supported by Grand-in-Aid for Scientific Research (B) 19310096. The later also by 20340024. M. N.'s and M. Y.'s researches were partially supported by Grand-in-Aid for Young Scientists (B) 21300017 and 21710148, respectively. M. N. was also supported by The Special Postdoctoral Researchers' Program of RIKEN. M. N. is thankful for Dr. Hayato Waki, Dr. Henry Cohn, Mr. Jeechul Woo, and Dr. Hans D. Mittelmann for their bug reports, discussions during developments of SDPA-GMP, -QD, -DD and MPACK.

References

[1] Amestoy, P.R., Duff, I.S., L'Excellent, J.-Y.: Multifrontal parallel distributed symmetric and unsymmetric solvers. Comput. Methods Appl. Mech. Eng. **184**, 501–520 (2000)

[2] Amestoy, P.R., Duff, I.S., L'Excellent, J.-Y., Koster, J.: A fully asynchronous multifrontal solver using distributed dynamic scheduling. SIAM J. Matrix Anal. Appl. **23**, 15–41 (2001)

[3] Amestoy, P.R., Guermouche, A., L'Excellent, J.-Y., Pralet, S.: Hybrid scheduling for the parallel solution of linear systems. Parallel Computing **32**, 136–156 (2006)

[4] Biswas, P., Ye, Y.: Semidefinite programming for ad hoc wireless sensor network localization. In: Proceedings of the Third International Symposium on Information Processing in Sensor Networks, pp. 46–54. ACM Press (2004)

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

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

[7] Boyd, S., El Ghaoui, L., Feron, E., Balakrishnan, V.: Linear Matrix Inequalities in System and Control Theory. SIAM, Philadelphia (1994)

[8] de Klerk, E., Pasechnik, D.V., Schrijver A.: Reduction of symmetric semidefinite programs using the regular $*$-representation. Math. Program. **109**, 613–624 (2007)

[9] de Klerk, E., Sotirov, R.: A new library of structured semidefinite programming instances. Optim. Methods Softw. **24**, 959–971 (2009)

[10] Fujisawa, K., Fukuda, M., Kobayashi, K., Kojima, M., Nakata, K., Nakata, M., Yamashita, M.: SDPA (SemiDefinite Programming Algorithm) User's Manual — Version 7.0.5, Research Report B-448, Department of Mathematical and Computing Sciences, Tokyo Institute of Technology, August 2008

[11] Fujisawa, K., Fukuda, M., Kojima, M., Nakata, K.: Numerical evaluation of SDPA (SemiDefinite Programming Algorithm). In: Frenk, H., Roos, K., Terlaky, T., Zhang, S. (eds.) High Performance Optimization, pp. 267–301. Kluwer Academic Publishers, Massachusetts (2000)

[12] Fujisawa, K., Kojima, M., Nakata, K.: Exploiting sparsity in primal-dual interior-point methods for semidefinite programming. Math. Program. **79**, 235–253 (1997)

[13] Goemans, M.X., Williamson, D.P.: Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. J. Assoc. Comput. Mach. **42**, 1115–1145 (1995)

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

[15] Hida, Y., Li, X.S., Bailey, D.H.: Quad-double arithmetic: Algorithms, implementation, and application, Technical Report LBNL-46996, Lawrence Berkeley National Laboratory, Berkeley, CA 94720, Oct. (2000)

[16] Kim, S., Kojima, M., Mevissen, M., Yamashita, M.: Exploiting sparsity in linear and nonlinear matrix inequalities via positive semidefinite matrix completion, Research Report B-452, Department of Mathematical and Computing Sciences, Tokyo Institute of Technology, January 2009

[17] Kim, S., Kojima, M., Waki, H.: Exploiting sparsity in SDP relaxation for sensor network localization. SIAM J. Optim. **20**, 192–215 (2009)

[18] Kobayashi, K., Kim, S., Kojima, M.: Correlative sparsity in primal-dual interior-point methods for LP, SDP and SOCP. Appl. Math. Optim. **58**, 69–88 (2008)

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

[20] Lasserre, J.B.: Global optimization with polynomials and the problems of moments. SIAM J. Optim. **11**, 796–817 (2001)

[21] Lobo, M.S., Vandenberghe, L., Boyd, S., Lebret, H.: Applications of second-order cone programming. Linear Algebra Appl. **284**, 193–228 (1998)

[22] Löfberg, J.: Dualize it: Software for automatic primal and dual conversions of conic programs. Optim. Methods Softw. **24**, 313–325 (2009)

[23] Mittelmann, H.D.: An independent benchmarking of SDP and SOCP solvers. Math. Program. **95**, 407–430 (2003)

[24] Mittelmann, H.D., Vallentin, F.: High accuracy semidefinite programming bounds for kissing numbers. arXiv:0902.1105v2 (2009)

[25] Mittelmann, H.D.: Sparse SDP Problems,
`http://plato.asu.edu/ftp/sparse_sdp.html`

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

[27] Murota, K., Kanno, Y., Kojima, M., Kojima, S.: A numerical algorithm for block-diagonal decomposition of matrix ∗-algebras, part I: Proposed approach and application to semidefinite programming. Japan J. Industrial Appl. Math., to appear

[28] Nakata, K., Fujisawa, K., Fukuda, M., Kojima, M., Murota, K.: Exploiting sparsity in semidefinite programming via matrix completion II: Implementation and numerical

results. Math. Program. **95**, 303–327 (2003)

[29] Nakata, K., Yamashita, M., Fujisawa, F., Kojima, M.: A parallel primal-dual interior-point method for semidefinite programs using positive definite matrix completion. Parallel Comput. **32**, 24–43 (2006)

[30] Nakata, M.: The MPACK: Multiple precision arithmetic BLAS (MBLAS) and LAPACK (MLAPACK). `http://mplapack.sourceforge.net/`

[31] Nakata, M., Braams, B.J., Fujisawa, K., Fukuda, M., Percus, J.K., Yamashita, M., Zhao, Z.: Variational calculation of second-order reduced density matrices by strong N-representability conditions and an accurate semidefinite programming solver. J. Chem. Phys. **128**, 164113 (2008)

[32] Strum, J.F.: SeDuMi 1.02, a MATLAB toolbox for optimization over symmetric cones. Optim. Methods Softw. **11/12**, 625–653 (1999)

[33] Todd, M.J.: Semidefinite optimization. Acta Numer. **10**, 515–560 (2001)

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

[35] Tsuchiya, T.: A convergence analysis of the scaling-invariant primal-dual path-following algorithm for second-order cone programming. Optim. Methods Softw. **11/12**, 141–182 (1999)

[36] Waki, H., Kim, S., Kojima, M., Muramatsu, M., Sugimoto, H.: Algorithm 883: SparsePOP: A Sparse Semidefinite Programming Relaxation of Polynomial Optimization Problems. ACM Trans. Math. Softw. **35**, 90-04 (2009)

[37] Waki, H., Nakata, M., Muramatsu, M.: Strange behaviors of interior-point methods for solving semidefinite programming problems in polynomial optimization. Comput. Optim. Appl., to appear.

[38] Wolkowicz, H., Saigal, R., Vandenberghe, L.: Handbook of Semidefinite Programming: Theory, Algorithms, and Applications. Kluwer Academic Publishers, Massachusetts (2000)

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

[40] Yamashita, M., Fujisawa, K., Kojima, M.: Implementation and evaluation of SDPA6.0 (SemiDefinite Programming Algorithm 6.0). Optim. Methods Softw. **18**, 491–505 (2003)