ISSN 1342-2804

Research Reports on Mathematical and Computing Sciences

# Parallel solver for semidefinite programming problem having sparse Schur complement matrix

Makoto Yamashita, Katsuki Fujisawa, Mituhiro Fukuda, Kazuhide Nakata and Maho Nakata

September 2010, B–463

Department of Mathematical and Computing Sciences
Tokyo Institute of Technology

SERIES B: Operations Research

# Parallel solver for semidefinite programming problem having sparse Schur complement matrix

Makoto Yamashita[†], Katsuki Fujisawa[‡], Mituhiro Fukuda[♯], Kazuhide Nakata[♮] and Maho Nakata[*]

September, 2010

**Abstract**

SemiDefinite Programming (SDP) problem is one of the most central problems in mathematical programming. SDP provides a practical computation framework for many research fields. Some applications, however, require solving large-scale SDPs whose size exceeds the capacity of a single processor in terms of computational time and available memory. SDPARA (SemiDefinite Programming Algorithm paRAllel version) developed by Yamashita *et al.* was designed to solve such large-scale SDPs. Its parallel performance is remarkable for general SDPs in most cases. However, the parallel implementation is less successful in some sparse SDPs from the latest applications such as for Polynomial Optimization Problems (POPs) or Sensor Network Location (SNL) problems, since the previous SDPARA can not directly handle sparse Schur complement matrices (SCMs). In this paper, we focus on the sparsity of the SCM and propose new parallel implementations, the formula-cost-based distribution, and the replacement of the dense Cholesky factorization. Through numerical results, we confirm that these features are keys to solve SDPs having sparse SCMs faster on parallel computer systems, and they are further enhanced by multi-threading. In fact, SDPARA is implemented in order to explore parallelism in two fronts: MPI-based and multi-threading of CPU cores. The new SDPARA attains a good scalability in general, and found solutions of extremely large-scale SDPs arising from POPs which other solvers could not obtain before.

† Department of Mathematical and Computing Sciences, Tokyo Institute of Technology, 2-12-1-W8-29 Ookayama, Meguro-ku, Tokyo 152-8552, Japan. This research was partially supported by Grant-in-Aid for Young Scientists (B) 21710148.
Makoto.Yamashita@is.titech.ac.jp

‡ Department of Industrial and Systems Engineering, Chuo University, 1-13-27 Kasuga, Bunkyo-ku, Tokyo 112-8551, Japan. K. Fujisawa's research was partially supported by Grant-in-Aid for Scientific Research (C) 20510143.
fujisawa@indsys.chuo-u.ac.jp

♯ Department of Mathematical and Computing Sciences, Tokyo Institute of Technology, 2-12-1-W8-41 Ookayama, Meguro-ku, Tokyo 152-8552, Japan. M. Fukuda's research was partially supported by Grant-in-Aid for Scientific Research (B) 20340024.
mituhiro@is.titech.ac.jp

♮ Department of Industrial Engineering and Management, Tokyo Institute of Technology, 2-12-1-W9-60 Ookayama, Meguro-ku, Tokyo 152-8552, Japan. K. Nakata's research was partially supported by Grant-in-Aid for for Young Scientists (B) 22710136.
nakata.k.ac@m.titech.ac.jp

∗ Advanced Center for Computing and Communication, RIKEN, 2-1 Hirosawa Wako-city, Saitama 351-0198, Japan. M. Nakata's research was partially supported by Grant-in-Aid for Scientific Research (B) 21300017.
maho@riken.jp

# 1 Introduction

SemiDefinite Programming (SDP) problem is a fundamental problem in mathematical programming. In SDP, one needs to find an optimal solution which minimizes or maximizes a linear objective function over the intersection of cones of positive semidefinite symmetric matrices and an affine subspace. By its Lagrangian dual formulation, an SDP can also be regarded as an optimization problem whose feasible region can be described by linear matrix inequalities.

It is well-known that SDPs can be solved efficiently by polynomial-time algorithms, for example, by the primal-dual interior-point methods (PDIPMs) [4, 18, 22, 25, 28]. Owing to the existence of practical polynomial-time algorithms and high reliability of SDP solvers, SDPs have been used in many different applications. For example, SDP relaxation methods proposed in [16] provide practical numerical approximations for some NP-hard problems. Other important applications include Polynomial Optimization Problems (POPs) [23, 29], Sensor Network Location (SNL) problems [8, 30], and computation of electronic structures of small atoms and molecules in quantum chemistry [15, 26, 27]. The researches in these fields could not be so active without solid developments of computer software for SDP problems.

Through the extensions of PDIPMs from linear programming to SDPs [4, 18, 22, 25, 28], several SDP solvers have been developed in the last 15 years such as SDPA [36], CSDP [10], SeDuMi [31], and SDPT3 [32]. In particular, SDPA pioneered these software. Furthermore, they have continuously stimulated the growth of new application areas. However, while the capability of these SDP solvers improved along the years [36], the size of general SDP problems demanded in practical applications has become large-scale and far beyond the capability of a single processor in terms of computational time and available memory space. It is important to remark at this point that several applications require a reasonable accuracy for the solution of an SDP, which can be only obtained by a second-order method such as PDIPMs. For applications which require less accuracy, one can employ other existing methods [12, 17].

A breakthrough in solving general and large-scale SDPs was brought by the combination of the PDIPM and parallel computation. Numerical experiments indicate that the most time consuming parts of PDIPMs are devoted to forming and solving a linear system named *Schur Complement Equation*, which appears at each iteration of these methods. In particular, the computation of its coefficient matrix called *Schur Complement Matrix* (SCM) and its Cholesky factorization often occupy over 80% of the total computation time [35]. Following these observations, SDPARA [35] (a parallel version of SDPA [36]) and PDSDP [5] (a parallel version of DSDP [6, 7]) replaced the computation bottlenecks relevant to the SCM by their parallel implementation. They successfully reduced the total computation time by accelerations of parallel computing. Due to the difference on the base algorithm (PDIPM embedded in SDPARA is generally more stable than the dual interior-point method in PDSDP), SDPARA usually outperformed PDSDP and attained higher scalability [35]. More recently, two parallel versions of CSDP [10], which also implement PDIPMs, were developed. One based on OpenMP, which runs only on shared-memory parallel systems [11], and PCSDP 1.0r1 [19]. Still, SDPARA can solve large-scale SDPs [26] which other SDP solvers are not reported to solve.

The latest SDP applications, however, have brought a novel requirement for SDP solvers.

Frequently in SDPs arising from POPs or SNLs, the SCMs become sparse. Most elements of the SCM generated by these SDPs are zero and this structure critically affects the total performance of SDP solvers. Since parallel SDP solvers were primarily developed for fully dense SCMs, they result in poor performance for SDPs from POPs or SNLs. Even for sparse SCMs, the evaluation of the SCMs and their Cholesky factorizations still occupy significant portions of the total computation time as we will see later.

The most important improvements of SDPA, developed also by our group, from version 6.2.1 [34] to 7.3.1 [36] are the adoption of appropriate data structures for both sparse and dense SCMs, and the introduction of the sparse Cholesky factorization of MUMPS [1, 2, 3] which reduced drastically the computation time. By integrating the previous version of SDPARA (version 1.0.1) and the latest version of SDPA (version 7.3.1), we have newly developed SDPARA version 7.3.1[1]. To solve SDPs with sparse SCMs efficiently, SDPARA 7.3.1 employs new parallel schemes which were inherited from the parallel schemes devised in SDPARA 1.0.1.

The main purpose of this paper is to introduce the new parallel schemes of SDPARA 7.3.1 and verify their improvements by numerical experiments. SDPARA 7.3.1 adopts a novel compact storage scheme when the SCM of the SDP problem is sparse, which greatly reduces the memory consumption. In addition, this storage scheme is totally compatible with the formula-cost-based distribution proposed here, based on the three-formula technique of [13], and achieves an ideal load balance when computing the elements of a sparse SCM. In the numerical results, we can confirm a great scalability of the parallelism in this part. In SDPARA 1.0.1, a simpler row-wise distribution was implemented for this part [35], which SDPARA 7.3.1 also employs when the SCM is not sparse.

In the previous version SDPARA 1.0.1, there was a necessity of redistributing the elements of the SCM at the processors in two-dimensional block-cyclic distribution, which requires network communication, to achieve the best performance for ScaLAPACK [35]. When the SCM of the SDP problem is sparse, the novel SDPARA 7.3.1 employs the sparse Cholesky factorization of MUMPS [1, 2, 3] which assumes an assignment of the elements of the SCM to the processors compatible with its own computation, and therefore, do not require any network communication among processors. Unfortunately, the parallel scalability of the sparse Cholesky factorization is not equally well as the scalability of the SCM computation, but still significant.

Furthermore, a unique feature of SDPARA 7.3.1 is its dual parallelism which few optimization software take full advantage of. It adopts the MPI-based parallel computation and also the multi-threading parallel computation which is possible at the currently available multi-core CPUs. In particular, the numerical results show that this implementation provides an astonishing speedup for SDPARA 7.3.1 on some SDPs.

The numerical results of SDPARA 7.3.1 confirm high performance for large-scale SDP problems with both sparse and dense SCMs. Some problems, in fact, can be solved thanks to the storage of the SCM on the memory assigned to multiple processors by SDPARA 7.3.1. In particular, we believe that we solved the largest SDP from POP reported in literature with more than 700,000 equality constraints. The multi-threading plays a great role in the speedup as mentioned before.

[1]the SDPARA version reflects the embedded SDPA version since version 7.2.1; SDPARA 1.0.1 was based on SDPA 6.2.1.

Finally, we compare the performance with PCSDP 1.0r1 [19], which is the other serious competitor for SDPARA 7.3.1. Our solver resulted in a superior performance in all cases. It is numerically more stable and at least 2.6 faster, reaching more than 700 speedup in some cases. SDPARA 7.3.1 remarkably extends its capability for large-scale SDPs having sparse SCMs.

The new SDPARA 7.3.1 can be downloaded from the SDPA web site:

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

SDPARA 7.3.1 is now distributed under GPL. Furthermore, the SDPA Online Solver system described here [14] enables users to use SDPARA via the Internet with no charge. The SDPA Online Solver system consists of PC clusters with a pre-installed SDPARAs, where users only need to submit the SDP problem data via the Internet to receive its solution. The SDPA Online Solver is available at:

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

This paper is organized as follows. In Section 2, we prepare the basic concepts for the following sections defining the SDP problem and describing a simplified scheme of the PDIPM to solve it. In Section 3, we give a summary of the parallel schemes implemented in SDPARA 1.0.1, and indicates its limitations when solving SDPs with sparse SCMs. Section 4 is the core of this paper where novel parallel schemes implemented in SDPARA 7.3.1 are proposed. In Section 5, numerical results on a PC cluster are reported to verify the performance of SDPARA 7.3.1. In addition, we compare the performance with another parallel SDP solver PCSDP 1.0r1 [19]. We give some conclusion remarks and future works in Section 6.

All SDP problems for the numerical experiments in this paper are summarized in the section for numerical results, Section 5. The parallel computing environment we used is also described there.

# 2 Semidefinite programming problem and primal-dual interior-point methods

We begin this section with the definition of the standard form of SDP used in this paper. Then, we introduce a basic framework of Primal-Dual Interior-Point Methods (PDIPMs). PDIPMs are powerful methods for SDPs from both theoretical and practical viewpoints.

The standard form of SDP we address in this paper is defined by the following primal-dual pair.

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

The symbol $\mathbb{S}^n$ denotes the space of $n \times n$ symmetric matrices and $\boldsymbol{X} \succeq \boldsymbol{O}$ stands for $\boldsymbol{X} \in \mathbb{S}^n$ being positive semidefinite. The inner product between $\boldsymbol{U}$ and $\boldsymbol{V}$ in $\mathbb{S}^n$ is defined by $\boldsymbol{U} \bullet \boldsymbol{V} = \sum_{i=1}^{n} \sum_{j=1}^{n} U_{ij} V_{ij}$. In the primal problem $\mathcal{P}$, the vector $\boldsymbol{x} \in \mathbb{R}^m$ and the symmetric

matrix $\boldsymbol{X} \in \mathbb{S}^n$ are variables. Meanwhile, in the dual problem $\mathcal{D}$, $\boldsymbol{Y} \in \mathbb{S}^n$ is the variable matrix. The input data are $c_k \in \mathbb{R}$ $(k = 1, 2, \ldots, m)$ and $\boldsymbol{F}_k \in \mathbb{S}^n$ $(k = 0, 1, 2, \ldots, m)$.

The size of an SDP can be roughly estimated by two factors, the number of equality constraints $m$ of the dual problem $(\mathcal{D})$ and the dimension of variable matrices $n$. In this paper, we mainly address the case $m \gg n$, which includes many practical applications such as SNLs and quantum chemistry problems.

Under mild assumptions as Slater's condition, we can obtain an optimal criterion for the above pair of problems from the Karush-Kuhn-Tucker conditions; an optimal solution $(\boldsymbol{x}^*, \boldsymbol{X}^*, \boldsymbol{Y}^*)$ should satisfy the following system.

$$\text{KKT} \begin{cases} \boldsymbol{X}^* = \sum_{k=1}^m \boldsymbol{F}_k x_k^* - \boldsymbol{F}_0, & \text{(primal feasibility)} \\ \boldsymbol{F}_k \bullet \boldsymbol{Y}^* = c_k \ (k = 1, 2, \ldots, m), & \text{(dual feasibility)} \\ \sum_{k=1}^m c_k x_k^* = \boldsymbol{F}_0 \bullet \boldsymbol{Y}^*, & \text{(primal-dual gap condition)} \\ \boldsymbol{X}^* \succeq \boldsymbol{O}, \boldsymbol{Y}^* \succeq \boldsymbol{O}. & \text{(positive semidefinite conditions)} \end{cases} \tag{2}$$

Conversely, solutions of the above system are also optimal to (1).

PDIPMs search a point $(\boldsymbol{x}^*, \boldsymbol{X}^*, \boldsymbol{Y}^*)$ which satisfies the Karush-Kuhn-Tucker conditions by iterating a modified Newton method in the region where the variable matrices are positive definite. These methods solve both primal $(\mathcal{P})$ and dual $(\mathcal{D})$ problems simultaneously. For an iterate $(\boldsymbol{x}, \boldsymbol{X}, \boldsymbol{Y})$, we define the three residuals $\boldsymbol{P} = \boldsymbol{F}_0 - \sum_{k=1}^m \boldsymbol{F}_k x_k + \boldsymbol{X}$, $d_k = c_k - F_k \bullet \boldsymbol{Y}_k$ $(k = 1, 2, \ldots, m)$ and $g = \sum_{k=1}^m c_k x_k - \boldsymbol{F}_0 \bullet \boldsymbol{Y}$. These residuals are evaluated by their appropriate norms. We use $\boldsymbol{X} \succ \boldsymbol{O}$ to denote the positive definiteness of $\boldsymbol{X} \in \mathbb{S}^n$.

The basic framework of PDIPMs is outlined below.

## Framework of Primal-Dual Interior-Point Methods

Step 0. Choose an initial point $\boldsymbol{x}^0, \boldsymbol{X}^0, \boldsymbol{Y}^0$ with $\boldsymbol{X}^0 \succ \boldsymbol{O}$ and $\boldsymbol{Y}^0 \succ \boldsymbol{O}$. Choose a threshold $\epsilon > 0$ and parameters $0 < \beta < 1$ and $0 < \gamma < 1$. Set the iteration number $h = 0$.

Step 1. Compute a search direction $(d\boldsymbol{x}, d\boldsymbol{X}, d\boldsymbol{Y})$ by a modified Newton method toward a point which would have smaller residuals $\boldsymbol{P}, \boldsymbol{d}$ and $g$.

Step 2. To keep the positive definiteness, we evaluate the maximum length of possible step, $\alpha_p = \max\{\alpha : \boldsymbol{X}^h + \alpha_p d\boldsymbol{X} \succeq \boldsymbol{O}\}$ and $\alpha_d = \max\{\alpha : \boldsymbol{Y}^h + \alpha_d d\boldsymbol{Y} \succeq \boldsymbol{O}\}$.

Step 3. Update the current point by $(\boldsymbol{x}^{h+1}, \boldsymbol{X}^{h+1}, \boldsymbol{Y}^{h+1}) = (\boldsymbol{x}^h + \gamma\alpha_p d\boldsymbol{x}, \boldsymbol{X}^h + \gamma\alpha_p d\boldsymbol{X}, \boldsymbol{Y}^h + \gamma\alpha_d d\boldsymbol{Y})$. Set $h = h + 1$.

Step 4. If $\max\{||\boldsymbol{P}||, ||\boldsymbol{d}||, |g|\} < \epsilon$, then output $(\boldsymbol{x}^h, \boldsymbol{X}^h, \boldsymbol{Y}^h)$ as an optimal solution. Otherwise, return to Step 1.

Step 1 includes the principal computation bottlenecks of the above framework [35]. More specifically, when the current point is $(\boldsymbol{x}, \boldsymbol{X}, \boldsymbol{Y})$, the computation of the search direction $(d\boldsymbol{x}, d\boldsymbol{X}, d\boldsymbol{Y})$ can be reduced to

$$\begin{aligned} \boldsymbol{B}d\boldsymbol{x} &= \boldsymbol{r} \\ d\boldsymbol{X} &= \sum_{k=1}^m \boldsymbol{F}_k dx_k - \boldsymbol{P} \\ \widehat{d\boldsymbol{Y}} &= \boldsymbol{X}^{-1}(\boldsymbol{R} - d\boldsymbol{X}\boldsymbol{Y}), \qquad d\boldsymbol{Y} = (\widehat{d\boldsymbol{Y}} + \widehat{d\boldsymbol{Y}}^T)/2, \end{aligned} \tag{3}$$

where

$$B_{ij} = (\boldsymbol{X}^{-1}\boldsymbol{F}_i\boldsymbol{Y}) \bullet \boldsymbol{F}_j \qquad (i = 1, 2, \ldots, m,\ j = 1, 2, \ldots, m) \tag{4}$$

$$r_k = -d_k + \boldsymbol{F}_k \bullet (\boldsymbol{X}^{-1}(\boldsymbol{R} + \boldsymbol{P}\boldsymbol{Y})) \qquad (k = 1, 2, \ldots, m)$$

$$\boldsymbol{R} = \beta\frac{\boldsymbol{X} \bullet \boldsymbol{Y}}{n}\boldsymbol{I} - \boldsymbol{X}\boldsymbol{Y}. \tag{5}$$

Details of this formula reduction can be found in many papers, for example, [13, 22, 34, 35].

Most computation time of the framework is consumed by the first linear system (3), which is usually known as the Schur complement equation. Its coefficient matrix whose elements are evaluated by the formula (4) is the Schur Complement Matrix (SCM). Since the SCM is always positive definite through all the iterations, its Cholesky factorization is usually employed to obtain $d\boldsymbol{x}$. As indicated in [35], when SDPA on a single processor solved an SDP from control theory or combinatorial optimization, more than 80% of the total computation time was spent by the computation related to the SCM. The parallel implementation of SDPARA 1.0.1, which will be discussed next, mainly concentrated on the two components related to the SCM, the evaluation of the SCM and its Cholesky factorization.

# 3 Existing parallel schemes of SDPARA 1.0.1

We point out the two principal computation bottlenecks of PDIPMs and briefly describe how the previous version of SDPARA 1.0.1 replaced them with their parallel implementations. Finally, we show an SDP example having a sparse SCM and how SDPARA 1.0.1 loses its efficiency. These matters will serve as essential pieces to understand the next section where new parallel schemes will be proposed.

The two computational bottlenecks in PDIPMs are the evaluation of the SCM and its Cholesky factorization. Following the nomenclature in [35], the evaluation of the SCM will be referred *ELEMENTS component*, and its Cholesky factorization *CHOLESKY component*.

A noteworthy property of the evaluation formula (4) is that the computation of each row of the SCM is completely independent from the other rows. For the $i$th row, we first multiply $\boldsymbol{U} = \boldsymbol{X}^{-1}\boldsymbol{F}_i\boldsymbol{Y}$, then take the inner products $\boldsymbol{U} \bullet \boldsymbol{F}_j$ $(j = 1, 2, \ldots, m)$. By duplicating the input data matrices $\boldsymbol{F}_k$ $(k = 1, 2, \ldots, m)$ on all processors before the iterations and updating the variable matrices $\boldsymbol{X}$ and $\boldsymbol{Y}$ on each processor at every iteration, any processor can evaluate any row without network communications from other ones. This property motivated SDPARA 1.0.1 to adopt the *row-wise distribution* for ELEMENTS component [35]. Figure 1 displays an example of the row-wise distribution in which the SCM is $8 \times 8$ and the number of available processors is 4. Note that since the matrix is always symmetric, only the upper triangular part must be evaluated. In the row-wise distribution, all the processors are assigned to the rows in a cyclic manner.

For CHOLESKY component, which is the subsequent computation after ELEMENTS component, we employed the parallel Cholesky factorization supplied by ScaLAPACK library [9]. To intensify the performance of the parallel Cholesky factorization, SDPARA 1.0.1 redistributes the SCM from the row-wise distribution to the *two-dimensional block-cyclic distribution* whose style was assumed by ScaLAPACK for the matrix to be factorized. This

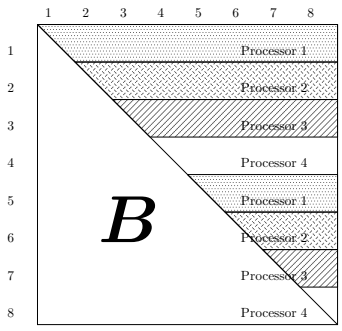


Figure 1: Parallel computation of the Schur complement matrix $\boldsymbol{B}$.

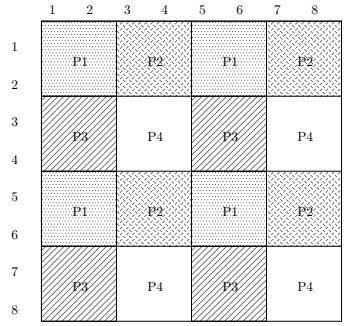


Figure 2: Two-dimensional block-cyclic distribution of the Schur complement matrix $\boldsymbol{B}$.

Table 1: Computation time (in seconds) for an SDP from quantum chemistry (Be.1S.SV.pqgt1t2p) by SDPARA 1.0.1 on different number of processors.

| # processors | 1 | 2 | 4 | 8 | 16 |
|---|---|---|---|---|---|
| ELEMENTS | 17236.91 | 8546.58 | 4333.07 | 2163.03 | 1069.04 |
| CHOLESKY | 191.72 | 96.47 | 57.96 | 39.17 | 23.49 |
| total | 17727.03 | 9050.55 | 4682.63 | 2492.25 | 1389.38 |

distribution is illustrated in Figure 2 in which $B_{17}$ and $B_{28}$ elements are stored by the 2nd processor, while $B_{34}$ and $B_{78}$ elements by the 4th processor.

The row-wise distribution and the parallel Cholesky factorization are the major features of SDPARA 1.0.1 to reduce the total computation time. Table 1 shows ELEMENTS time, CHOLESKY time and the total computation time when we applied SDPARA 1.0.1 to an SDP problem from quantum chemistry, 'Be.1S.SV.pqgt1t2p' [26]. See Table 5 for the sizes of this SDP. The row-wise distribution seems to be a simple scheme to achieve good load-balance, however, we can verify that it produces almost a linear scalability for ELEMENTS component. That is, the 16 times speedup on 16 processors, which produced the 12 times speedup of the total computation time. These numbers are called *scalability* in parallel computation.

SDPARA 1.0.1 is very fast on some classes of SDPs. However, it has been observed through recent years that SDPARA 1.0.1 loses its merit for the latest applications such as POPs or SNLs. Table 2 gives the computation time for the SDP problem 'BroydenBand30' generated by SparsePOP [33]. SDPA 7.3.1, which only runs on a single processor, saves both memory space and computation time compared to SDPARA 1.0.1 even using 16 processors.

The main difference between the problems in Tables 1 and 2 is the sparsity of the SCM. Figure 3 sketches the positions of nonzeros elements of the SCM for SDPs from quantum chemistry (Be.1S.SV.pqgt1t2p) and POP (BroydenBand40), respectively. The left figure is an example of a fully dense SCM. On the contrary, the right figure indicates that the SCM is sparse.

The sparsity of the SCM comes from the so-called diagonal block structure of input data

Table 2: Computation time (in seconds) for an SDP from POP (BroydenBand30) by SDPARA 1.0.1 and SDPA 7.3.1 on different number of processors. 'O.M.' stands for out of memory.

| | SDPARA 1.0.1 | | | | | SDPA 7.3.1 |
|---|---|---|---|---|---|---|
| # processors | 1 | 2 | 4 | 8 | 16 | 1 |
| ELEMENTS | O.M. | 211.61 | 106.42 | 56.46 | 24.62 | 212.05 |
| CHOLESKY | O.M. | 4911.94 | 2455.88 | 1419.86 | 643.23 | 326.71 |
| total | O.M. | 5591.48 | 2937.71 | 1779.13 | 823.58 | 553.51 |

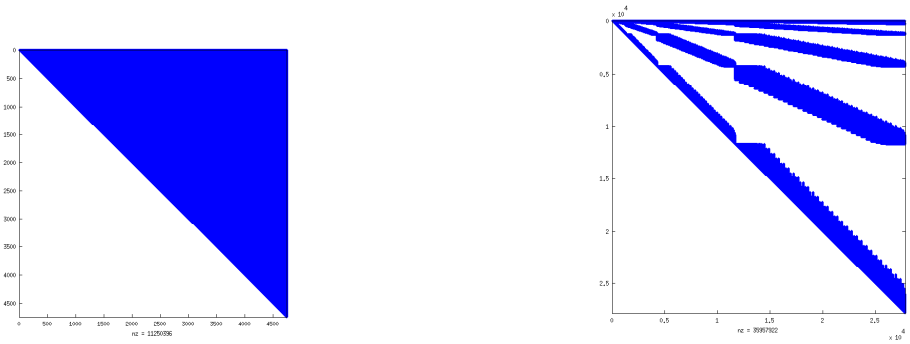

Figure 3: Nonzero elements of the Schur complement matrices (only the upper triangular parts) in SDPs from quantum chemistry (left) and POP (right).

matrices. Suppose that all the input data matrices $\boldsymbol{F}_k$ $(k = 1, 2, \ldots, m)$ share the same diagonal block structure with matrix sizes $n_1, n_2, \ldots, n_{\bar{\ell}}$. More precisely, suppose that each matrix $\boldsymbol{F}_k$ can be decomposed into sub-matrices $[\boldsymbol{F}_k]^\ell \in \mathbb{S}^{n_\ell}$ $(\ell = 1, 2, \ldots, \bar{\ell})$ positioned at its diagonal,

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

Then, necessarily the variable matrices $\boldsymbol{X}$ and $\boldsymbol{Y}$ also share the same structure and can be decomposed into $[\boldsymbol{X}]^\ell$ and $[\boldsymbol{Y}]^\ell \in \mathbb{S}^{n_\ell} (\ell = 1, 2, \ldots, \bar{\ell})$. Consequently, the evaluation formula (4) can come down to the sum over sub-matrices,

$$\begin{aligned} B_{ij} &= (\boldsymbol{X}^{-1}\boldsymbol{F}_i\boldsymbol{Y}) \bullet \boldsymbol{F}_j \\ &= \sum_{\ell=1}^{\bar{\ell}} \left([\boldsymbol{X}^{-1}]^\ell [\boldsymbol{F}_i]^\ell [\boldsymbol{Y}]^\ell\right) \bullet [\boldsymbol{F}_j]^\ell. \qquad (i = 1, 2, \ldots, m,\ j = 1, 2, \ldots, m) \end{aligned} \tag{6}$$

This breakdown demonstrates that $B_{ij}$ becomes zero if $[\boldsymbol{F}_i]^\ell$ or $[\boldsymbol{F}_j]^\ell$ is the zero matrix for all $\ell = 1, 2, \ldots, \bar{\ell}$. We should remark here that in POPs or SNLs, larger SDPs have stronger tendency of this phenomenon and generate sparser SCMs. This phenomenon can be checked in Table 5.

The critical drawback of SDPARA 1.0.1 is that the parallel schemes described above are restricted to fully dense SCMs. The new version of SDPARA 7.3.1 overcomes this drawback by introducing new parallel schemes for ELEMENTS and CHOLESKY components, enhancing the functions to handle sparse SCMs of SDPA 7.3.1.

# 4 New features of SDPARA 7.3.1

The new SDPARA 7.3.1 has remarkable enhancements compared to the previous version. It speeds up the computational time for SDPs specially with sparse SCMs. We begin detailing the storage scheme which prioritizes the load-balance between the processors for the SCM computation. And then discuss its sparse Cholesky factorization. Finally, we explain the MPI and multi-threading computation of these routines.

## 4.1 A new storage and load-balance schemes for the parallel computation of sparse Schur complement matrices

To discuss the new parallel schemes, this section starts from how we store a sparse SCM. Compared to the two-dimensional block-cyclic distribution for the fully dense matrix case, more elaborated storage managements should be provided. From (6), the non-zero pattern $S$ of the SCM is determined by

$$S = \bigcup_{\ell=1}^{\bar{\ell}} \left\{(i,j) |\ [\boldsymbol{F}_i]^\ell \neq \boldsymbol{O} \text{ and } [\boldsymbol{F}_j]^\ell \neq \boldsymbol{O}, 1 \leq i \leq j \leq m\right\}. \tag{7}$$

The patterns $S$ are illustrated in Figure 3. The sparsity found in the pattern $S$ is called correlative sparsity in [21] and it is directly related to the performance of the sparse Cholesky factorization. Note that $S$ is invariant through all the iterations of PDIPMs. Therefore, by counting the number of elements in $S$, we can allocate a memory space for the triples $(i, j, B_{ij})$ for each $(i, j) \in S$ in advance.

We use the symbol $|S|$ to denote the number of elements in $S$. Then the density of the SCM is defined by

$$\text{density} = \frac{2|S| - m}{m^2}.$$

In this definition, $S$ and $|S|$ are defined only for the upper triangular part while the density is considered for the whole matrix. For instance, the density of the right figure in Figure 3 is only 9.26%, and we should avoid applying the dense Cholesky factorization to it.

The number of fill-in affects the performance of the sparse Cholesky factorization. However, for simplicity, we do not consider it in detail in this paper. As we will describe in the next subsection, we employ MUMPS [1, 2, 3] for the sparse Cholesky factorization and MUMPS automatically minimizes the fill-in by some heuristics. Actual implementation also involves some criteria based on the number of fill-in similarly.

In the new parallel scheme, we assign ELEMENTS computation to each processor by dividing $S$ into subsets $S_1, S_2, \ldots, S_u$ where $u$ is the number of available processors. Hence, each element is evaluated on a fixed processor through all the diagonal blocks. This concept is mainly due to the network communication overheads as justified next. Let $[B_{ij}]^\ell$ $(\ell = 1, 2, \ldots, \bar{\ell})$ denote the partial value of the SCM relevant to the $\ell$th block (*i.e.*, $B_{ij} = \sum_{\ell=1}^{\bar{\ell}} [B_{ij}]^\ell$). If $[B_{ij}]^1$ is evaluated on the 1st processor, $[B_{ij}]^2$ on the 2nd processor, and $B_{ij}$ is supposed to be stored on the 3rd processor, then $[B_{ij}]^1$ and $[B_{ij}]^2$ can not be summed without network commutations to the 3rd processor. We can not underestimate this network overhead, since $|S|$ is beyond 1 billion for extremely large-scale problems which SDPARA is designed to solve. In addition, the number of blocks considered for the composition of each element $B_{ij}$ varies for each $(i, j)$. Therefore, the network flow over all the processors will be too complicated. These factors prevent SDPARA from attaining better scalability, and hence we assign the computation of each element through all the diagonal blocks to one processor.

Now, what we have to consider is how to divide $S$ to $S_1, S_2, \ldots, S_u$. To derive best performance of the subsequent CHOLESKY component described below, sequential elements of $S$ should be assigned to the same processor. To be more precise, we first enumerate consecutively the elements of $S$ in a row-wise way by $\mathcal{N}(i, j)$. The upper numbers in Figure 4 show an example of $\mathcal{N}(i, j)$ for the SCM with size $10 \times 10$ and $|S|$ being 24. With this notation, the division is determined to satisfy

$$\mathcal{N}(i, j) < \mathcal{N}(i', j') \text{ for } (i, j) \in S_p,\ (i', j') \in S_{p'}.\ (1 \leq p < p' \leq u)$$

The division of $S$ is now equivalent to finding delimiter points $d_0 = 0, d_1, d_2, \ldots, d_{u-1}, d_u = |S|$. From the sequential element policy for CHOLESKY component, each subset $S_p$ for the $p$th processor can be expressed in another way,

$$S_p = \{(i, j) \in S : d_{p-1} < \mathcal{N}(i, j) \leq d_p\}. \quad (p = 1, 2, \ldots, u)$$

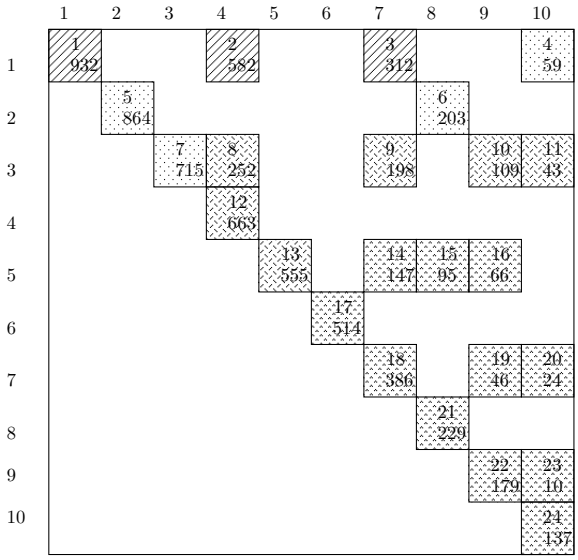


Figure 4: An example of sparse Schur complement matrix with row-wise indexes $\mathcal{N}(i,j)$ (upper numbers) and estimated costs $\sum_{\ell=1}^{\bar{\ell}}[F_{ij}]^\ell$ (lower numbers).

The key for ELEMENTS component to be processed faster is to have a perfect load-balance among the processors. Let $[\mathcal{F}_{ij}]^\ell$ denote the computation cost for the evaluation of $[B_{ij}]^\ell$. Then, the computation load on the $p$th processor $\mathcal{L}_p$ is calculated by

$$\mathcal{L}_p = \sum_{(i,j)\in S_p} \sum_{\ell=1}^{\bar{\ell}} [\mathcal{F}_{ij}]^\ell. \quad (p = 1, 2, \ldots, u)$$

When $[\mathcal{F}_{ij}]^\ell$ is given, a simple heuristic is enough to determine $d_0, d_1, \ldots, d_u$ which makes all of $\mathcal{L}_p$ $(p = 1, \ldots, u)$ close to their average $\sum_{p=1}^{u} \mathcal{L}_p/u$, since $|S|$ is usually larger than $u$. In large-scale SDPs, $|S|$ can exceeds millions, while $u$ is usually at most 1000.

Our main interest is now moved to $[\mathcal{F}_{ij}]^\ell$, since a reasonably accurate estimation for $[\mathcal{F}_{ij}]^\ell$ is essential to obtain $\mathcal{L}_p$. This estimation is provided by an outstanding method originally implemented in SDPA [13]. The remarkable feature in SDPA is that it selects the evaluation formula for $[B_{ij}]^\ell$ from the three candidates, $\mathcal{F}_1, \mathcal{F}_2$ and $\mathcal{F}_3$,

$$\begin{aligned}
\mathcal{F}_1 &: [\boldsymbol{U}]^\ell = [\boldsymbol{X}^{-1}]^\ell [\boldsymbol{F}_i]^\ell [\boldsymbol{Y}]^\ell, \quad [B_{ij}]^\ell = \sum_{\alpha,\beta} [\boldsymbol{U}]^\ell_{\alpha,\beta} [\boldsymbol{F}_j]^\ell_{\alpha,\beta} \\
\mathcal{F}_2 &: [\boldsymbol{U}]^\ell = [\boldsymbol{F}_i]^\ell [\boldsymbol{Y}]^\ell, \quad [B_{ij}]^\ell = \sum_{\alpha,\beta} \sum_{\gamma} [\boldsymbol{X}^{-1}]^\ell_{\alpha,\gamma} [\boldsymbol{U}]^\ell_{\gamma,\beta} [\boldsymbol{F}_j]^\ell_{\alpha,\beta} \\
\mathcal{F}_3 &: [B_{ij}]^\ell = \sum_{\alpha,\beta} \sum_{\gamma,\delta} [\boldsymbol{X}^{-1}]^\ell_{\alpha,\gamma} [\boldsymbol{F}_i]^\ell_{\gamma,\delta} [\boldsymbol{Y}]^\ell_{\delta,\beta} [\boldsymbol{F}_j]^\ell_{\alpha,\beta},
\end{aligned}$$

where $[\boldsymbol{F}_i]^\ell_{\alpha,\beta}$ is the $(\alpha,\beta)$ element of $[\boldsymbol{F}_i]^\ell$. Although each formula generates the correct value of $[B_{ij}]^\ell$, the computation time considerably differs depending on the number of nonzero elements of $[\boldsymbol{F}_k]^\ell$ $(k = 1, 2, \ldots, m)$. The formula $\mathcal{F}_1$ is effective when both $[\boldsymbol{F}_i]^\ell$

and $[\boldsymbol{F}_j]^\ell$ are dense, while $\mathcal{F}_3$ works well when both $[\boldsymbol{F}_i]^\ell$ and $[\boldsymbol{F}_j]^\ell$ are sparse. SDPA automatically selects the best formula from the three; For details, refer to [13].

Note that all three formulas are composed of floating-point multiplications and additions. Since the computation cost for floating-point multiplications is generally greater than the computation cost for floating-point additions, each computation cost for the three formulas is approximately proportional to the number of floating-point multiplications. The number of nonzero elements of $[\boldsymbol{F}_k]^\ell$ $(k = 1, 2, \ldots, m)$ is vital to estimate $[\mathcal{F}_{ij}]^\ell$ and consequently to determine $S_1, S_2, \ldots, S_u$.

We call the distribution $S_1, S_2, \ldots, S_u$ based on $[\mathcal{F}_{ij}]^\ell$ the *formula-cost-based distribution*. Figure 4 also serves as an example; the lower numbers indicate the estimated costs $\sum_{\ell=1}^{\bar{\ell}} [\mathcal{F}_{ij}]^\ell$ of the corresponding elements. If $u = 4$ and the delimiter points are $d_0 = 0$, $d_1 = 3$, $d_2 = 7$, $d_3 = 13$, and $d_4 = 24$, then $\mathcal{L}_1 = 1826$, $\mathcal{L}_2 = 1841$, $\mathcal{L}_3 = 1820$, and $\mathcal{L}_4 = 1833$. The computation load on each processor is close to their average 1830. Since we compute $[\boldsymbol{U}]^\ell$ for $\mathcal{F}_1$ and $\mathcal{F}_2$ at the computation of the diagonal elements $[B_{ii}]^\ell$ for $i = 1, 2, \ldots, m$, its computation cost is embedded to the cost of $[B_{ii}]^\ell$.

## 4.2 Sparse Cholesky factorization of the Schur complement matrices

After ELEMENTS component, SDPARA 7.3.1 proceeds to CHOLESKY component. Instead of the parallel dense Cholesky factorization of ScaLAPACK, we employ the parallel *sparse* Cholesky factorization of MUMPS [1, 2, 3]. At the present, MUMPS is the sole existing software which can attain reasonable scalability for the sparse factorization. MUMPS assumes that the matrix to be factorized is distributed under some specific style. Under the *distributed assembled matrix distribution*, the matrix should be stored with the style of triples $(i, j, B_{ij})$ of consecutive elements in the row-wise direction. It means that the memory storage of this distribution is completely consistent with the formula-cost-based distribution considered previously. Hence, on the contrary of cases for dense SCMs (Section 3), we do not redistribute the matrix prior to CHOLESKY component. This reduces the total network communication.

Unfortunately, the scalability of the parallel sparse Cholesky factorization of MUMPS is lower than the scalability of the dense factorization of ScaLAPACK due to the complicated framework of the multiple frontal methods adopted in MUMPS. When the SCM is weakly sparse, the dense factorization sometimes becomes faster than the sparse factorization on more processors.

SDPARA 7.3.1 can automatically select which factorization is better with the information supplied by MUMPS and scalability information obtained by preliminary numerical experiments. In practice, most SDPs are almost fully dense or extremely sparse, and cases in which sensitive automatic selection is required is rare. Hence either the dense factorization or the sparse factorization is usually selected irrelevantly to the number of available processors.

Table 3 shows a preliminary numerical experiment using SDPARA 7.3.1. The SDP solved here is 'BroydenBand600', an SDP relaxation problem of POP generated by SparsePOP [33]. Compared to a single processor which needs 4505 seconds, SDPARA on 16 processors solves the problem in only 345 seconds. In other words, we obtain an 11.73

Table 3: Computation time (in seconds) of SDPARA 7.3.1 on 'BroydenBand600' on different number of processors.

| # processors | 1 | 2 | 4 | 8 | 16 |
|---|---|---|---|---|---|
| ELEMENTS | 4505.22 | 2345.16 | 1194.63 | 640.67 | 345.12 |
| CHOLESKY | 7495.95 | 4555.90 | 2776.02 | 1810.49 | 1356.10 |
| Total | 12367.39 | 7186.17 | 4200.83 | 2667.58 | 1933.37 |

times speedup. This reduction is mainly derived by the formula-cost-based distribution. For CHOLESKY component, the resultant speedup is only 5.52. However, the unnecessity of the redistribution before performing the CHOLESKY component is of great importance. Consequently, SDPARA 7.3.1 on 16 processors attains the 6.39 times speedup.

We should note that the SDP 'BroydenBand600' solved here is much larger than the SDP 'BroydenBand30' solved in Table 2. SDPARA 1.0.1 used in Table 2 could not handle 'BroydenBand600' due to lack of memory. A fact that was overcome by the sparse SCM and its sparse Cholesky factorization.

The performance of SDPARA 7.3.1 has already become better for sparse SCM than other parallel SDP solvers. However, this performance will be further improved by multi-threading discussed in the next section.

## 4.3 Multi-threading acceleration on MPI-based parallel computing

In recent years, it has become common that processors feature multi-core. This change has brought a multi-threading speedup to SDPA 7.3.1 [36].

The new SDPARA 7.3.1 combines MPI-based parallel computing discussed so far and multi-threading enhancement in ELEMENTS component. We first discuss the combination of multi-threading and MPI-based computation for dense SCMs and then move to sparse SCMs. For CHOLESKY component, preliminary numerical experiments showed that using optimized and multi-threaded BLAS is enough to obtain some benefits from the multi-threading.

As described in Section 3 for dense SCMs, the row-wise distribution is vital to SDPARA 1.0.1 to process ELEMENTS component in a short time. More precisely, the $i$th row of the SCM is evaluated by the $p$th processor when $i - p$ is a multiple of the number of available processors $u$. Let $\mathcal{R}_p$ denote the set of row indexes assigned to the $p$th processor,

$$\mathcal{R}_p = \{i : 1 \leq i \leq m,\ i - p \text{ is a multiple of } u\}.$$

All cores on each processor share the memory space allocated to $\mathcal{R}_p$, hence, each thread can store its own computed elements of the SCM into the memory space attached to the processor. Therefore, the task of each thread on the $p$th processor can be summarized as follows. Let $i_s$ be the smallest number in $\mathcal{R}_p$. First pick up $i_s$ and remove it from $\mathcal{R}_p$. Then the thread evaluates the $i_s$th row of the SCM. Continue this process until $\mathcal{R}_p$ becomes empty. By controlling the access of all the threads to $\mathcal{R}_p$, we can guarantee that each row

is evaluated by only one thread. This strategy produces a better load-balance if compared to the case of row-wise distribution on multi-threading.

It is a natural question why we do not employ this strategy not only in multi-threading but also in the MPI-based parallel computing. The reason is the difference of memory access. If we adopt this strategy for the MPI-based computing, we can not fix $\mathcal{R}_p$. Unbalanced sizes of $\mathcal{R}_p$ cause complex network communications of the redistribution prior to CHOLESKY component. Since the load-balance of the row-wise distribution in MPI-based is already nice as indicated in Table 1, the complex communication just becomes an overhead.

Next, we move to sparse SCMs. In the formula-cost-based distribution, the division of $S$ to $S_1, S_2, \ldots, S_u$ decides the computation assigned to each processor. The strategy adopted for multi-threading here is the sub-divisions of $S_1, S_2, \ldots, S_u$ again. Let $v$ be the number of threads. We suppose that all the processors have the same number of threads. On the $p$th processor, we apply the formula-cost-based distribution to $S_p$ and obtain the sub-divisions $S_p^1, S_p^2, \ldots, S_p^v$. Then the $q$th thread on the $p$th processor evaluates the elements of $\boldsymbol{B}$ specified by $S_p^q$. An important point we should emphasize is that this strategy keeps the independence of each thread. Without any communication to other threads, each thread can concentrate on its own task. This attribute yields better load-balance, hence higher scalability.

Table 4 shows the result of the combination of MPI-based parallel computing and multi-threading. The SCM is fully dense for 'Be.1S.SV.pqgt1t2p', while the density of 'Broyden-Band600' is only 0.559%. At first, we focus on the dense case ('Be.1S.SV.pqgt1t2p') on a single processor. The computation time for ELEMENTS is reduced from 10984 seconds to 1585 seconds; we obtain the 6.93 times speedup with 8 threads. Using 8 threads on the whole 16 processors (128 threads in total), the time is just 109.87 seconds; the scalability on 128 threads reaches 99.9. This excellent scalability and the effect of multi-threading BLAS for CHOLESKY component bring the remarkable efficiency to the total computation time; by 128 threads, we finally attain the 61.2 times speedup in the total time.

The formula-cost-based distribution for the sparse case ('BroydenBand600') has a more elaborate distribution. The scalability for ELEMENTS component is, however, still kept as high as 75.0 on 128 threads. Even though CHOLESKY component is not accelerated so greatly, SDPARA 7.3.1 can achieve the 8.28 times speedup in the total time, even in sparse case.

The remarkable scalability of ELEMENTS component is, of course, derived from the good load-balance (Section 4.1). The load-balance over all the threads is shown in Figure 5. These are the results of SDPs in Table 4 on 16 processors with 8 threads. The horizontal axis is the sequential thread numbers over all the processors. The $q$th thread on $p$th processor is mapped to the thread number $(p-1)\cdot v + q$, where $v = 8$ now. The vertical axis is the computation time of ELEMENTS component for one iteration of the PDIPM.

For the dense case, the maximum/minimum ratio on 128 threads is just $\frac{2.84}{2.72} = 1.04$. This exquisite load-balance generates the outstanding 99.9 times speedup. On the other hand, for the sparse case, the shortest thread time is much shorter than the second shortest. The formula-cost-based distribution is based on the estimation of $\mathcal{F}_1, \mathcal{F}_2$ and $\mathcal{F}_3$ (Section 4.1). Even though this estimation is already established taking the effect of sparse data structures, memory access for sparse data structures is sometimes too complicated to predict completely. It might be inevitable to incur such an undesirable situation. By excluding the 1st thread, the maximum/minimum ratio is $\frac{1.94}{1.06} = 1.80$. As a consequence, we obtained the reasonable

Table 4: Performance of SDPARA 7.3.1 on multiple processors with multiple threads for a dense SCM (Be.1S.SV.pqgt1t2p) and sparse SCM (BroydenBand600).

| | | # processors | 1 | 2 | 4 | 8 | 16 |
|---|---|---|---|---|---|---|---|
| problem | # threads | | | | | | |
| Be.1S.SV.pqgt1t2p | 1 | ELEMENTS | 10984.21 | 5524.70 | 2763.75 | 1430.10 | 722.08 |
| | | CHOLESKY | 161.08 | 81.02 | 44.77 | 31.83 | 19.21 |
| | | Total | 11355.65 | 5812.68 | 3005.04 | 1657.17 | 932.64 |
| | 2 | ELEMENTS | 5466.67 | 2746.67 | 1417.01 | 714.22 | 361.67 |
| | | CHOLESKY | 109.28 | 47.81 | 30.35 | 23.35 | 15.14 |
| | | Total | 5713.40 | 2931.11 | 1576.99 | 861.28 | 497.01 |
| | 4 | ELEMENTS | 2736.80 | 1412.62 | 691.33 | 358.99 | 178.60 |
| | | CHOLESKY | 78.90 | 32.55 | 21.71 | 18.84 | 12.41 |
| | | Total | 2913.74 | 1544.25 | 799.46 | 460.61 | 268.09 |
| | 8 | ELEMENTS | 1584.97 | 819.63 | 413.59 | 206.12 | 109.87 |
| | | CHOLESKY | 67.93 | 25.64 | 19.02 | 16.64 | 12.35 |
| | | Total | 1736.17 | 927.87 | 505.83 | 288.39 | 185.62 |
| BroydenBand600 | 1 | ELEMENTS | 4505.22 | 2345.16 | 1194.63 | 640.67 | 345.12 |
| | | CHOLESKY | 7495.95 | 4555.90 | 2776.02 | 1810.49 | 1356.10 |
| | | Total | 12367.39 | 7186.17 | 4200.83 | 2667.58 | 1933.37 |
| | 2 | ELEMENTS | 2956.41 | 1471.20 | 735.05 | 387.69 | 204.13 |
| | | CHOLESKY | 6006.12 | 3736.66 | 2368.92 | 1636.86 | 1272.09 |
| | | Total | 9333.16 | 5486.32 | 3316.80 | 2235.10 | 1702.98 |
| | 4 | ELEMENTS | 1928.32 | 844.87 | 376.20 | 206.12 | 105.98 |
| | | CHOLESKY | 5121.50 | 3236.48 | 2098.46 | 1464.52 | 1201.91 |
| | | Total | 7397.88 | 4344.94 | 2687.83 | 1876.75 | 1530.61 |
| | 8 | ELEMENTS | 1020.10 | 457.31 | 202.23 | 112.49 | 60.09 |
| | | CHOLESKY | 4560.67 | 2973.21 | 1975.04 | 1440.76 | 1210.11 |
| | | Total | 5929.96 | 3697.20 | 2390.19 | 1759.99 | 1493.65 |

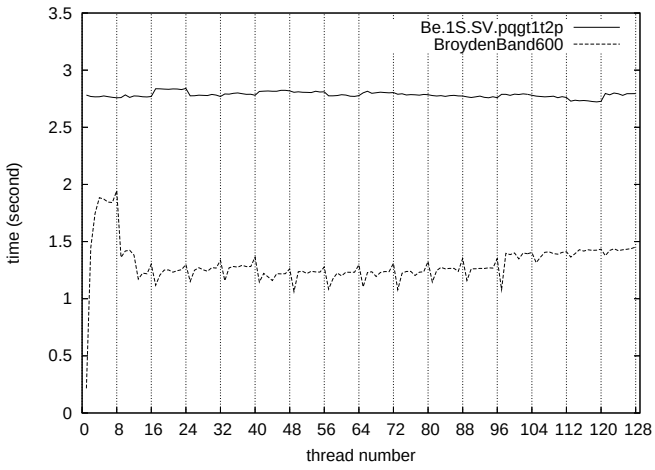


Figure 5: Load balance for ELEMENTS component of SDPARA 7.3.1 for the problems in Table 4.

scalability 75.0 on 128 threads.

# 5 Numerical Results

The SDPs we used for the numerical experiments through the paper are summarized in Table 5. They can be categorized into four classes: POP, SNL, QC and Mittelmann. The first class 'POP' problems are the SDP relaxation problems generated by SparsePOP [33]. For example, 'BroydenBand30' was generated by the SparsePOP command `sparsePOP('BroydenBand(30)');` with its default parameters. The 'SNL' class consists of Sensor Network Localization problems generated by SFSDP [20]. The problem 'sfsdp30000' has 30,000 sensors scattered in the two dimensional square $[0,1] \times [0,1]$. The noise threshold for the sensor-sensor or sensor-anchor distance is set to 10% and the radio range is set to 0.1, respectively. Details for 'QC' (quantum chemistry) class can be found in [26]. The constraints of these SDPs are based on the $P$, $Q$, $G$, $T1$, $T2'$ conditions and the optimal solutions of these SDPs give approximate electronic structures of considered atoms/molecules. The SDPs of 'Mittelmann' class are chosen from the benchmark problems at Mittelmann's website [24]. In 'Mittelmann' class, we excluded small problems which can be solved in less than 500 seconds on a single processor with a single thread. Meanwhile, some small-size problems in POPs and SNLs are chosen to compare SDPARA results with another parallel SDP solver PCSDP [19]. The 3rd column in Table 5 corresponds to the number of equality constraints of $(\mathcal{D})$ in the standard form (1), while the 4th column $n$ is the dimension of $\boldsymbol{X}$ and $\boldsymbol{Y}$. The 5th column describes the number of blocks in the diagonal block structure. Here, we do not count the blocks whose sizes are one, since they can be evaluated in ELEMENTS component separately by a simpler way as linear programming. The 6th column $n_{\max}$ is the largest block size defined by $n_{\max} = \max\{n_1, n_2, \ldots, n_{\bar{\ell}}\}$. The last column shows the density of the SCM. This column indicates that larger SDPs have sparser SCMs in POP or SNL and fully dense ones in QC or Mittelmann.

All numerical experiments in this paper were executed on a PC cluster composed of 16

Table 5: SDP problems for the numerical experiments.

| class | name | $m$ | $n$ | blocks | $n_{\max}$ | density of the SCM |
|---|---|---|---|---|---|---|
| POP | BroydenBand30 | 19931 | 2880 | 24 | 120 | 12.9% |
| | BroydenBand600 | 471371 | 71280 | 594 | 120 | $5.59 \times 10^{-1}$% |
| | BroydenBand800 | 629771 | 95280 | 794 | 120 | $4.18 \times 10^{-1}$% |
| | BroydenBand900 | 708971 | 107280 | 894 | 120 | $3.72 \times 10^{-1}$% |
| | ChainedSingular500 | 9974 | 4980 | 498 | 10 | $4.81 \times 10^{-1}$% |
| | ChainedSingular20000 | 399974 | 199980 | 19998 | 10 | $1.20 \times 10^{-2}$% |
| | ChainedSingular30000 | 599974 | 299980 | 29998 | 10 | $8.00 \times 10^{-3}$% |
| SNL | sfsdp500 | 7933 | 12004 | 423 | 21 | $9.44 \times 10^{-1}$% |
| | sfsdp30000 | 452217 | 808272 | 29887 | 44 | $7.87 \times 10^{-3}$% |
| | sfsdp35000 | 527096 | 943151 | 34887 | 43 | $6.52 \times 10^{-3}$% |
| QC | Be.1S.SV.pqgt1t2p | 4743 | 4444 | 21 | 1062 | 100% |
| | N.4P.DZ.pqgt1t2p | 7230 | 6010 | 21 | 1460 | 100% |
| Mittelmann | butcher | 6434 | 22842 | 1 | 330 | 100% |
| | neu3g | 8007 | 462 | 1 | 462 | 100% |
| | reimer5 | 6187 | 102606 | 1 | 462 | 100% |
| | shmup5 | 1800 | 11041 | 2 | 3721 | 100% |
| | taha1b | 8007 | 1609 | 21 | 286 | 100% |

nodes. Each node has 48 GB memory space and 2 Xeon X5460 (3.16GHz : 4 CPU cores) processors. Therefore, the maximal number of CPU cores for multi-threading on each node is 8. All the nodes are connected by Myrinet-10G network interface. The selection of an optimized BLAS significantly affects the total performance. We adopt GotoBLAS 1.26. The stopping tolerance is usually set to $\epsilon = 1.0 \times 10^{-7}$ (see Section 2). Only for SNL problems, the stopping tolerance is relaxed to $\epsilon = 1.0 \times 10^{-5}$. We do not need high accuracy for SNL problems since the relaxed tolerance is enough to generate a good starting point for the posterior local methods.

Tables 6 and 7 show the computation time of SDPARA 7.3.1 for SDPs with sparse SCMs and dense SCMs, respectively. We changed the number of nodes by 1, 2, 4, 8, 16, and the number of threads by 1 and 8. In the table, 'O.M.' stands for Out of Memory. In addition, 'thread problem due to MUMPS error' means that SDPARA 7.3.1 fails due to MUMPS, which is not thread-safe in some parts. Hence it can not adequately compute the Cholesky factorization in multi-threading computation for some SDPs.

Next, we discuss the results of Table 6 in more details. First, for 'BroydenBand800' and 'BroydenBand900', SDPARA 7.3.1 can attain reasonable scalability. These SDPs are in the same class of the SDP in Tables 3 and 4, but larger than it. Note that using one or two nodes, we could not solve them due to the lack of memory. We claim that the new SDPARA is the first SDP solver that can solve problems of this class of SDPs with this size. In particular, 'BroydenBand900' which has $m = 708,971$ equality constraints can be solved in about half hour. Without exploiting the sparsity of the SCM, even SDPARA 7.3.1 could not store this extremely large-scale SDP in memory.

Table 6: Computation time (in seconds) of SDPARA 7.3.1 for SDP problems with sparse SCMs using 1,2,4,8,16 processors, and 1 and 8 threads. 'O.M' means out of memory.

| | | # processors | 1 | 2 | 4 | 8 | 16 |
|---|---|---|---|---|---|---|---|
| problem | # threads | | | | | | |
| BroydenBand800 | 1 | ELEMENTS | O.M. | O.M. | 1661.73 | 903.53 | 478.25 |
| | | CHOLESKY | O.M. | O.M. | 3745.28 | 2295.04 | 1690.84 |
| | | Total | O.M. | O.M. | 5733.85 | 3490.69 | 2454.14 |
| | 8 | ELEMENTS | O.M. | O.M. | O.M. | 155.22 | 84.96 |
| | | CHOLESKY | O.M. | O.M. | O.M. | 1790.92 | 1437.99 |
| | | Total | O.M. | O.M. | O.M. | 2246.97 | 1795.20 |
| BroydenBand900 | 1 | ELEMENTS | O.M. | O.M. | 2001.92 | 1016.01 | 546.94 |
| | | CHOLESKY | O.M. | O.M. | 4253.95 | 2522.83 | 1785.76 |
| | | Total | O.M. | O.M. | 6648.20 | 3874.47 | 2692.98 |
| | 8 | ELEMENTS | O.M. | O.M. | O.M. | O.M. | 105.62 |
| | | CHOLESKY | O.M. | O.M. | O.M. | O.M. | 1500.01 |
| | | Total | O.M. | O.M. | O.M. | O.M. | 1932.74 |
| ChainedSingular20000 | 1 | ELEMENTS | 26.37 | 18.45 | 9.02 | 5.85 | 3.06 |
| | | CHOLESKY | 65.59 | 40.69 | 24.04 | 14.19 | 9.29 |
| | | Total | 152.92 | 123.77 | 89.01 | 73.93 | 185.11 |
| | 8 | ELEMENTS | 30.84 | 27.05 | 25.79 | 25.57 | 25.42 |
| | | CHOLESKY | 82.12 | 52.33 | 30.49 | 18.08 | 12.51 |
| | | Total | 242.17 | 216.81 | 174.45 | 150.65 | 288.41 |
| ChainedSingular30000 | 1 | ELEMENTS | 40.72 | 25.41 | 14.30 | 9.45 | 5.23 |
| | | CHOLESKY | 99.32 | 62.06 | 36.04 | 21.15 | 11.84 |
| | | Total | 231.50 | 184.57 | 134.47 | 111.48 | 229.80 |
| | 8 | ELEMENTS | 46.14 | 46.63 | 44.14 | 43.02 | 41.59 |
| | | CHOLESKY | 125.97 | 77.39 | 45.58 | 26.72 | 15.03 |
| | | Total | 379.34 | 327.78 | 259.32 | 231.94 | 381.46 |
| sfsdp30000 | 1 | ELEMENTS | 81.30 | 49.39 | 30.00 | 21.14 | 16.27 |
| | | CHOLESKY | 189.86 | 117.78 | 69.13 | 42.97 | 40.50 |
| | | Total | 492.46 | 390.37 | 294.41 | 254.89 | 241.41 |
| | 8 | threads problem due to MUMPS | | | | | |
| sfsdp35000 | 1 | ELEMENTS | 90.10 | 54.93 | 33.76 | 23.37 | 18.37 |
| | | CHOLESKY | 213.01 | 139.67 | 74.50 | 49.03 | 43.51 |
| | | Total | 564.36 | 458.54 | 338.90 | 296.99 | 281.23 |
| | 8 | threads problem due to MUMPS | | | | | |

Table 7: Computation time (in seconds) of SDPARA 7.3.1 for SDP problems with fully dense SCMs using 1,2,4,8,16 processors, and 1 and 8 threads.

| | | # processors | 1 | 2 | 4 | 8 | 16 |
|---|---|---|---|---|---|---|---|
| problem | # threads | | | | | | |
| N.4P.DZ.pqgt1t2p | 1 | ELEMENTS | 35551.82 | 17844.27 | 8948.75 | 4485.27 | 2267.93 |
| | | CHOLESKY | 629.71 | 278.17 | 150.17 | 95.01 | 54.62 |
| | | Total | 36276.29 | 18655.51 | 9604.72 | 5071.99 | 2803.00 |
| | 8 | ELEMENTS | 5261.17 | 2665.11 | 1360.39 | 691.21 | 305.85 |
| | | CHOLESKY | 254.92 | 71.22 | 50.00 | 41.81 | 28.23 |
| | | Total | 5716.05 | 2931.07 | 1577.61 | 887.01 | 522.37 |
| butcher | 1 | ELEMENTS | 512.57 | 255.95 | 128.00 | 68.43 | 35.42 |
| | | CHOLESKY | 514.75 | 224.64 | 124.94 | 85.92 | 49.29 |
| | | Total | 1080.94 | 523.24 | 278.09 | 170.23 | 96.60 |
| | 8 | ELEMENTS | 70.67 | 35.87 | 18.63 | 10.01 | 5.31 |
| | | CHOLESKY | 201.54 | 60.20 | 44.92 | 40.39 | 25.22 |
| | | Total | 320.48 | 131.14 | 86.33 | 64.11 | 39.30 |
| neu3g | 1 | ELEMENTS | 1612.06 | 808.03 | 405.10 | 203.65 | 102.93 |
| | | CHOLESKY | 825.92 | 358.26 | 190.67 | 119.40 | 67.30 |
| | | Total | 2509.47 | 1217.32 | 630.59 | 344.19 | 185.18 |
| | 8 | ELEMENTS | 227.06 | 114.15 | 57.35 | 29.09 | 15.01 |
| | | CHOLESKY | 332.24 | 87.23 | 59.39 | 49.09 | 32.72 |
| | | Total | 614.78 | 244.44 | 144.91 | 94.36 | 57.91 |
| reimer5 | 1 | ELEMENTS | 906.37 | 455.74 | 236.18 | 119.05 | 62.59 |
| | | CHOLESKY | 199.94 | 92.71 | 50.09 | 32.77 | 19.19 |
| | | Total | 1140.65 | 578.64 | 306.24 | 166.36 | 92.09 |
| | 8 | ELEMENTS | 136.16 | 69.06 | 35.98 | 18.58 | 10.06 |
| | | CHOLESKY | 81.60 | 25.26 | 17.86 | 15.44 | 10.45 |
| | | Total | 246.38 | 120.86 | 70.51 | 45.66 | 28.25 |
| shmup5 | 1 | ELEMENTS | 1066.97 | 545.27 | 262.23 | 138.09 | 76.02 |
| | | CHOLESKY | 10.59 | 7.23 | 4.25 | 3.70 | 2.63 |
| | | Total | 9072.93 | 8283.13 | 7557.27 | 7432.77 | 7371.42 |
| | 8 | ELEMENTS | 171.74 | 113.56 | 83.04 | 67.65 | 61.97 |
| | | CHOLESKY | 4.77 | 3.85 | 3.24 | 3.46 | 2.89 |
| | | Total | 1518.61 | 1463.46 | 1432.64 | 1421.46 | 1439.77 |
| taha1b | 1 | ELEMENTS | 308.91 | 161.42 | 78.88 | 39.64 | 20.11 |
| | | CHOLESKY | 785.28 | 338.58 | 180.27 | 112.96 | 63.4 |
| | | Total | 1158.88 | 545.18 | 288.61 | 169.29 | 94.54 |
| | 8 | ELEMENTS | 47.21 | 25.95 | 14.14 | 7.97 | 4.65 |
| | | CHOLESKY | 312.41 | 83.67 | 56.93 | 47.57 | 31.36 |
| | | Total | 412.72 | 151.08 | 97.94 | 71.08 | 46.02 |

We need a small comment about some 'O.M.'s on 8 threads, which not occur when solving by a single thread. To make all the threads concentrate on their assigned computation, each thread must allocate its own memory space for the temporary matrix $[\boldsymbol{U}]^{\ell}$ of $\mathcal{F}_1$ and $\mathcal{F}_2$ (see Section 4.1). When we execute by a single thread, the memory space is already over 40GB, which is close to the maximum amount of 48GB. Therefore, we exhaust the memory space when using 8 threads.

In the ChainedSingular problems, SDPARA 7.3.1 fails to attain its parallel efficiency, since the structure of the SCM for these problems is too simple. When we apply a SYMAMD (symmetric approximate minimum degree permutation) to the SCM prior to the sparse Cholesky factorization, we can detect that its factorization will be a band-diagonal matrix with a very small bandwidth. Consequently, the portion of the parallel computation is not so significant. However, the formula-cost-based distribution for ELEMENT component is still effective in these problems on a single thread. In multiple-threading case, all the threads encounter many conditional jumps through many small diagonal blocks, and therefore managing the threads becomes an obstacle which lowers parallel efficiency.

The SDP 'sfsdp35000' is the largest SDP which SFSDP can generate on 48GB memory space. Even for this largest SDP, SDPARA 7.3.1 can solve it in only 300 seconds. SDPARA is becoming a powerful tool for SNL researches.

Now, we move our focus to Table 7, in which the SDPs have fully dense SCMs. SDPARA 7.3.1 successfully attains high scalability in both ELEMENTS and CHOLESKY components and, as a result, the total computation time is astonishingly shortened, except one SDP, 'shmup5'. The speedup for 'N.4p.DZ.pqgt1t2p' on 128 threads goes over 60 times. Its scalability 69.4 is higher than 61.2 of 'Be.1S.SV.pqgt1t2p' (Table 4). A remarkable property of SDPARA 7.3.1 is that it can attain higher scalability when larger SDPs are solved. In 'reimer5', the portion of ELEMENTS and CHOLESKY in the total time is decreased from $\frac{906+199}{1140} = 0.97$ to $\frac{10+10}{28} = 0.72$, by the increment of processors and threads. This is another evidence which indicates the parallel schemes of SDPARA 7.3.1 are greatly reinforced by multi-threading.

The problem 'shmup5' is exceptional because $m < n_{\max}$. In this case, the chief computation bottleneck to be considered is the matrix multiplications in (5). Therefore, we can expect that even though the row-wise distribution and the parallel Cholesky factorization still shrink their time, their contribution is relatively small. The results in the table fit this inference. However, we should note that multi-threading considerably reduces the total time for this SDP. This reduction is an effect of GotoBLAS.

Finally, Table 8 compares the performance of SDPARA 7.3.1 with PCSDP 1.0r1 [19]. PCSDP is another parallel SDP solver, whose base is CSDP developed by Borchers [10]. Here, we do not include PDSDP [5], since it is not updated in the last five years. In the SDPARA 7.3.1 experiments, we use 1 and 8 threads, while we fix it to 8 for PCSDP 1.0r1; hence the difference due to the performance of BLAS is absence when the thread number of SDPARA 7.3.1 is 8. Since PCSDP 1.0r1 does not print out the required computation time, we use the Linux `time` command instead and thus the smallest time unit is one second when the total time exceeds one hour. We adopt the default parameters of SDPARA 7.3.1 and PCSDP 1.0r1, respectively, except for SNLs where the stopping tolerance is set to $\epsilon = 1.0 \times 10^{-5}$.

SDPARA 7.3.1 on a single thread can solve all SDPs with sparse SCMs faster than PCSDP 1.0r1 on 8 threads whenever is comparable. The latter solver cannot solve three

problems due to memory insufficiency. Furthermore, multi-threading widens the margin between SDPARA 7.3.1 and PCSDP 1.0r1 for 'BroydenBand30', meanwhile for problems such as 'ChainedSingular500' or 'sfsdp500', their variable matrices sizes are too small to obtain merits from multi-threading. The main advantage of SDPARA 7.3.1 over PCSDP 1.0r1 is that the former can handle sparse SCMs. In other words, PCSDP 1.0r1 always applies the dense parallel Cholesky factorization. Since it is easier to achieve higher parallel efficiency for the dense factorization than for the sparse one, the scalability of PCSDP 1.0r1 seems better than SDPARA 7.3.1. However, if we pay attention to the computation time itself, the difference is clear. Furthermore, the behavior of PCSDP 1.0r1 is sometimes unstable. For example, PCSDP 1.0r1 on 2 processors solves 5 times faster than a single processor for 'sfsdp500'. Some unknown reason makes PCSDP 1.0r1 very slow on a single processor. In addition PCSDP 1.0r1 could not store large instances of POP or SNL problems due to memory shortage.

For dense SCM cases, the computation time of SDPARA 7.3.1 with a single thread is also already shorter than PCSDP 1.0r1. SDPARA 7.3.1 successfully obtains great benefits from multi-threading which widens its speed over PCSDP 1.0r1. Hence, SDPARA 7.3.1 can solve 'N.4P.DZ.pgqt1t2p' 8.96 times faster than PCSDP 1.0r1 on the whole 128 threads.

# 6 Concluding remarks and future works

In this paper, we have discussed the new parallel schemes implemented in SDPARA 7.3.1; the formula-cost-based distribution and the sparse Cholesky factorization for SDPs with sparse SCMs. Through numerical experiments, we have verified that the new schemes successfully reduce the total computation time and allows to solve large-scale SDPs which can not be stored in the memory available for a single processor. In addition, multi-threading has substantially enhanced the parallel performance of SDPARA 7.3.1 for all cases. In particular, SDPARA 7.3.1 has solved extremely large-scale BroydenBand-type SDPs which other solvers could not perform. We expect this solid solubility will enrich researches on POPs and SNLs and extend the range of SDP applications.

The results of the current paper motivate future works. We mention two of them here. One challenge is how we solve SDPs with (number of equality constraints) $m < n$ (size of block-diagonal variable matrices), which includes many practical applications, such as 'shmup5' in shorter time. As pointed out, SDPARA usually addresses SDPs with $m > n$ where we mainly resolve the bottlenecks related to the SCM. Even in cases where $m < n$, it might be possible to accelerated SDPARA by distributing the variable matrices $\boldsymbol{X}$ and $\boldsymbol{Y}$ among all the processors. However, this distribution would require complicated network communication. It will be a question whether this network overhead can be justified by the reduction of the bottlenecks related to $\boldsymbol{X}$ and $\boldsymbol{Y}$ or not. The other issue is that there might be a space for further improvements on multi-threading. Accessing to same data simultaneously by multiple threads sometimes interferes on better performance and higher accuracy. The knowledge about memory hierarchy in the computer architecture will be important clues for this case.

Table 8: Computation time comparison (in seconds) between SDPARA 7.3.1 (1 and 8 threads) and PCSDP 1.0r1 (8 threads) on 1,2,4,8,16 processors for SDP problems with sparse and fully dense SCMs. 'O.M' means out of memory.

| | # processors | 1 | 2 | 4 | 8 | 16 |
|---|---|---|---|---|---|---|
| problem | solver | | | | | |
| BroydenBand30 | SDPARA(1) | 564.56 | 312.42 | 293.12 | 165.68 | 142.66 |
| | SDPARA(8) | 259.54 | 167.74 | 156.79 | 91.45 | 81.60 |
| | PCSDP(8) | O.M. | 1098.10 | 677.67 | 431.60 | 284.91 |
| BroydenBand900 | SDPARA(1) | O.M. | O.M. | 6648.20 | 3874.47 | 2692.98 |
| | SDPARA(8) | O.M. | O.M. | O.M. | O.M. | 1932.74 |
| | PCSDP(8) | O.M. | | | | |
| ChainedSingular 500 | SDPARA(1) | 2.82 | 2.57 | 2.27 | 2.16 | 2.12 |
| | SDPARA(8) | 5.39 | 5.13 | 4.65 | 4.54 | 4.48 |
| | PCSDP(8) | 4133.00 | 1460.66 | 766.12 | 354.42 | 226.01 |
| ChainedSingular 30000 | SDPARA(1) | 231.50 | 184.57 | 134.47 | 111.48 | 229.80 |
| | SDPARA(8) | 379.34 | 327.78 | 259.32 | 231.94 | 381.46 |
| | PCSDP(8) | O.M. | | | | |
| sfsdp500 | SDPARA(1) | 3.83 | 2.76 | 2.22 | 1.75 | 4.55 |
| | SDPARA(8) | 6.33 | 4.67 | 3.89 | 3.11 | 11.80 |
| | PCSDP(8) | 1018.59 | 201.92 | 127.02 | 92.91 | 67.68 |
| sfsdp35000 | SDPARA(1) | 564.36 | 458.54 | 338.90 | 296.99 | 281.23 |
| | SDPARA(8) | threads problem due to MUMPS | | | | |
| | PCSDP(8) | O.M. | | | | |
| Be.1S.SV pqgt1t2p | SDPARA(1) | 11355.65 | 5812.68 | 3005.04 | 1657.17 | 932.64 |
| | SDPARA(8) | 1736.17 | 927.87 | 505.83 | 288.39 | 185.62 |
| | PCSDP(8) | 12704.00 | 6575.00 | 3485.84 | 1972.93 | 1165.95 |
| N.4P.DZ.pgqt1t2 | SDPARA(1) | 36276.29 | 18655.51 | 9604.72 | 5071.99 | 2803.00 |
| | SDPARA(8) | 5716.05 | 2931.07 | 1577.61 | 887.01 | 522.37 |
| | PCSDP(8) | 53577.00 | 27672.00 | 15178.00 | 8174.00 | 4682.00 |
| butcher | SDPARA(1) | 1080.94 | 523.24 | 278.09 | 170.23 | 96.60 |
| | SDPARA(8) | 320.48 | 131.14 | 86.33 | 64.11 | 39.30 |
| | PCSDP(8) | 1805.67 | 779.43 | 483.73 | 263.48 | 198.43 |
| neu3g | SDPARA(1) | 2509.47 | 1217.32 | 630.59 | 344.19 | 185.18 |
| | SDPARA(8) | 614.78 | 244.44 | 144.91 | 94.36 | 57.91 |
| | PCSDP(8) | 4745.00 | 1809.45 | 969.18 | 581.70 | 327.49 |
| reimer5 | SDPARA(1) | 1140.65 | 578.64 | 306.24 | 166.36 | 92.09 |
| | SDPARA(8) | 246.38 | 120.86 | 70.51 | 45.66 | 28.25 |
| | PCSDP(8) | 4869.00 | 2847.09 | 1984.54 | 1572.26 | 1323.23 |
| shmup5 | SDPARA(1) | 9072.93 | 8283.13 | 7557.27 | 7432.72 | 7371.42 |
| | SDPARA(8) | 1518.61 | 1463.46 | 1432.64 | 1421.46 | 1439.77 |
| | PCSDP(8) | 11350.00 | 11005.00 | 10480.00 | 11122.00 | 8030.00 |
| taha1b | SDPARA(1) | 1158.88 | 545.18 | 288.61 | 169.29 | 94.54 |
| | SDPARA(8) | 412.72 | 151.08 | 97.94 | 71.08 | 46.02 |
| | PCSDP(8) | 1757.47 | 497.56 | 271.70 | 191.22 | 119.60 |

# References


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

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

[3] P. R. Amestoy, A. Guermouche, J.-Y. L'Excellent, and S. Pralet, Hybrid scheduling for the parallel solution of linear systems, *Parallel Comput.* **32(2)**, (2006), 136–156.

[4] F. Alizadeh, J.-P. A. Haeberly, and M. L. Overton, Primal-dual interior-point methods for semidefinite programming: Convergence rates, stability and numerical results, *SIAM J. Optim.* **8(3)**, (1998), 746–768 .

[5] S. J. Benson, Parallel computing on semidefinite programs, *Preprint ANL/MCS-P939-0302* (2002).

[6] S. J. Benson and Y. Ye, Algorithm 875: DSDP5 – Software for semidefinite programming, *ACM Trans. Math. Software* **34(3)**, (2008), article 16 (20 pages).

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

[8] P. Biswas and Y. Ye, Semidefinite programming for ad hoc wireless sensor network localization, *in Proceedings of the third international symposium on information processing in sensor networks*, ACM press, (2004), 46–54.

[9] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley, ScaLAPACK Users' Guide, *Society for Industrial and Applied Mathematics*, (1997), Philadelphia.

[10] B. Borchers, CSDP, a C library for semidefinite programming, *Optim. Methods Software*, **11 & 12(1-4)**, (1999), 613–623.

[11] B. Borchers and J. G. Young, Implementation of a primal-dual method for SDP on a shared memory parallel architecture, *Comput. Optim. Appl.* **37(3)**, (2007), 355–369.

[12] S. Burer and R. D. C. Monteiro, A nonlinear programming algorithm for solving semidefinite programs via low-rank factorization, *Math. Program. Series B* **95(2)**, (2003), 329–357.

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

[14] K. Fujisawa, K. Nakata, M. Yamashita, and M. Fukuda, SDPA project: Solving large-scale semidefinite programs, *J. Oper. Research Soc. Jap.* **50(4)**, (2007), 278–298.

[15] M. Fukuda, B. J. Braams, M. Nakata, M. L. Overton, J. K. Percus, M. Yamashita, and Z. Zhao, Large-scale semidefinite programs in electronic structure calculation, *Math. Program. Series B* **109(2-3)**, (2007), 553–580.

[16] M. X. Goemans and D. P. Williamson, Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming, *J. Assoc. Comput. Mach.* **42(6)**, (1995), 1115–1145.

[17] C. Helmberg and F. Rendl, A spectral bundle method for semidefinite programming, *SIAM J. Optim.* **10(3)**, (2000), 673–696.

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

[19] I. D. Ivanov and E. de Klerk, Parallel implementation of a semidefinite programming solver based on CSDP in a distributed memory cluster, *Optim. Methods Software* **25(3)**, (2010), 405–420.

[20] S. Kim, M. Kojima, H. Waki, and M. Yamashita, SFSDP: a Sparse version of Full SemiDefinite Programming relaxation for sensor network localization problems, Research Report B-457, Dept. of Math. and Comp. Sciences, Tokyo Institute of Technology, Oh-Okayama, Meguro, Tokyo 152-8552, July 2009.

[21] K. Kobayashi, S. Kim, and M. Kojima, Correlative sparsity in primal-dual interior-point methods for LP, SDP and SOCP, *Appl. Math. Optim.* **58(1)**, (2008), 69–88.

[22] M. Kojima, S. Shindoh, and S. Hara, Interior-point methods for the monotone semidefinite linear complementarity problems, *SIAM J. Optim.* **7**, (1997), 86-125.

[23] J. B. Lasserre: Global optimization with polynomials and the problems of moments, *SIAM J. Optim.*, **11(3)**, (2001), 796–817.

[24] H. D. Mittelmann, Additional SDP test problems, http://plato.asu.edu/ftp/sdp/00README

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

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

[27] M. Nakata, H. Nakatsuji, M. Ehara, M. Fukuda, K. Nakata, and K. Fujisawa, Variational calculations of fermion second-order deduced density matrices by semidefinite programming algorithm, *J. Chem. Phys.* **114**, (2001), 8282–8292.

[28] Yu. E. Nesterov and M. J. Todd, Primal-dual interior-point methods for self-scaled cones, *SIAM J. Optim.* **8(2)**, (1998), 324–364.

[29] P. A. Parrilo, Semidefinite programming relaxations for semialgebraic problems, *Math. Program. Series B* **96(2)**, (2003), 293–320.

[30] A. M. C. So and Y. Ye, Theory of semidefinite programming for sensor network localization, *Math. Program. Series B* **109(2-3)**, (2007), 367–384.

[31] J. F. Strum, Using SeDuMi 1.02, a MATLAB toolbox for optimization over symmetric cones, *Optim. Methods Software* **11 & 12(1-4)**, (1999), 625–653.

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

[33] H. Waki, S. Kim, M. Kojima, M. Muramatsu, and H. Sugimoto, Algorithm 883: sparsePOP: A Sparse Semidefinite Programming Relaxation of Polynomial Optimization Problems, *ACM Trans. Math. Software* **35(2)**, (2008), article 15 (13 pages).

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

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

[36] M. Yamashita, K. Fujisawa, K. Nakata, M. Nakata, M. Fukuda, K. Kobayashi, and K. Goto, A high-performance software package for semidefinite programs: SDPA 7. Research Report B-460, Dept. of Math. and Comp. Sciences, Tokyo Institute of Technology, Oh-Okayama, Meguro, Tokyo 152-8552, January 2010.