Parallel solver for semidefinite programming problem having sparse Schur complement matrix

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

August 2011

Abstract

A semidefinite programming (SDP) problem is one of the most central problems in mathematical programming. SDP provides an effective computation framework for many research fields. Some applications, however, require solving of a large-scale SDP whose size exceeds the capacity of a single processor in terms of computation time and available memory. SDPARA (SemiDefinite Programming Algorithm paRAllel package) developed by Yamashita *et al.* was designed to solve such large-scale SDPs. Its parallel performance is outstanding 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 cannot directly handle sparse Schur complement matrices (SCMs). In this paper, we improve SDPARA by focusing on the sparsity of the SCM and propose new parallel implementation, *i.e.*, the formula-cost-based distribution and a replacement of the dense Cholesky factorization. We numerically verify that these features are keys to solving SDPs having sparse SCMs more quickly on parallel computing systems. The performance is further enhanced by multi-threading. The new SDPARA attains considerable scalability in general. It also finds solutions for extremely large-scale SDPs arising from POPs which can not be obtained by other solvers.

† Department of Mathematical and Computing Sciences, Tokyo Institute of Technology, 2-12-1-W8-29 Ookayama, Meguro-ku, Tokyo 152-8552, Japan. M. Yamashita's 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 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

A semidefinite programming (SDP) problem is a fundamental one in mathematical programming. In SDP, an optimal solution must be found that minimizes or maximizes a linear objective function over the intersection of cones of positive semidefinite symmetric matrices and an affine subspace. An SDP can also be regarded as an optimization problem whose feasible region is defined by linear matrix inequalities that correspond to its Lagrangian dual formulation.

SDPs have been used in many applications, since practical polynomial-time algorithms such as primal-dual interior-point methods (PDIPMs) can be used to solve them [1, 18, 22, 25, 28]. For example, SDP relaxation methods proposed in Ref. [17] provide practical approximations for some non-determistic polynomial-time (NP)-hard problems. Other important applications include polynomial optimization problems (POPs) [23, 29], sensor network location (SNL) problems [8, 31], and computation of electronic structures of small atoms and molecules in quantum chemistry [16, 26, 27].

SDP solvers based on PDIPMs have been developed in the last 15 years, *e.g.*, SDPA [39], CSDP [10], SeDuMi [32], and SDPT3 [34]. SDPA was the first of these developed. However, while the capability of these solvers have improved over the years [39], the size of general SDP problems in practical applications has become large-scale and is far beyond the capability of a single processor in terms of computation time and available memory space. Note that some applications require a highly accurate solution, which can be only obtained by a second-order method such a PDIPM. For applications that require less precision, other methods [12, 30, 33, 36, 40] can be used.

A breakthrough in solving general and large-scale SDPs was reached through a combination of PDIPMs and parallel computation. Numerical experiments indicate that forming and solving a linear system named the *Schur complement equation* are the most time-consuming parts of PDIPMs. In particular, coefficient matrix of the system, called the *Schur complement matrix* (SCM), and its Cholesky factorization, which needs to be computed at every iteration of the PDIPMs, often consume over 80% of the total computation time [38]. SDPARA [38] (the parallel version of SDPA [39]) and PDSDP [5] (the parallel version of DSDP [6, 7]) resolved the computational bottlenecks associated with the SCM and thus reduced the required computation time. More recently, two parallel versions of CSDP [10] (one based on OpenMP, which runs only on shared-memory parallel systems [11], and PCSDP 1.0r1, which runs on distributed-memory parallel systems [19]) were developed. The size of the SDP arising from quantum chemistry [26] that SDPARA can solve is still larger than those solvable by the other solvers.

The latest SDP applications, however, have a different structure that can be exploited. SDPs arising from POPs or SNLs often have sparse SCMs. More than 90% of the SCM elements in these SDPs is zero, and this structure seriously affects the total performance of SDP solvers. Because parallel SDP solvers were primarily developed for fully dense SCMs, they perform poorly for such SDPs. Even for sparse SCMs, the evaluation of the SCMs and their Cholesky factorizations still consume significant portions of the entire computation time, as will described in the following section.

To solve SDPs with sparse SCMs, we have developed SDPARA version 7.3.1 by inte-

grating the previous version of SDPARA (version 1.0.1) and SDPA (version 7.3.1) [39] [1]. The improvements in the latest SDPA include the ability to handle data structures for both sparse and dense SCMs and utilization of the sparse Cholesky factorization of the MUltifrontal Massively Parallel sparse direct Solver (MUMPS) [2, 3, 4].

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.

The parallel distribution for sparse SCMs developed newly in SDPARA 7.3.1 is named *formula-cost-based distribution*. We estimate the computational load of the SCMs on the processors based on the three-formula method in Ref. [14] and execute parallel computations in a different way from SDPARA 1.0.1. As a result, SDPARA 7.3.1 achieves a remarkable load balance when evaluating a sparse SCM and also has shortened computation time.

In addition, SDPARA 7.3.1 inherits the sparse Cholesky factorization of MUMPS from SDPA 7.3.1. We modified the memory storage of the SCM to fit the sparse factorization so that the network communication for the redistribution of the SCM is reduced. The parallel scalability of the sparse Cholesky factorization is not as good as the scalability of the SCM computation, but significant reduction can still be observed.

An important aspect of SDPARA 7.3.1 is its hybrid parallelism, which is used in only a few optimization packages. It adopts the Message Passing Interface(MPI)-based parallel computation and the multi-threading parallel computation, which is known to be possible for multi-core CPUs. This is useful when the SCMs are both fully dense and sparse.

The numerical results of SDPARA 7.3.1 confirm high performance for large-scale SDP problems with both sparse and dense SCMs. These results are in accordance with the three improvements: the formula-cost-based distribution, the parallel sparse Cholesky factorization, and the hybrid parallel computations. In addition, we believe that we have solved the largest SDP from POP reported in literature with more than 700,000 equality constraints. This shows that SDPARA 7.3.1 has remarkably extended capability for large-scale SDPs having sparse SCMs.

This paper is organized as follows. In Section 2, we present the basic concepts for the subsequent sections by defining the SDP problem and describing a simplified scheme of a PDIPM. In Section 3, we give an overview of parallel schemes implemented in SDPARA 1.0.1 and describe its limitations when solving SDPs with sparse SCMs. Section 4, the main part of this paper, proposes the parallel schemes implemented in SDPARA 7.3.1. Section 5 reports numerical results for a PC cluster verifying the performance of SDPARA 7.3.1. We also compare SDPARA with another parallel SDP solver, PCSDP 1.0r1 [19]. We give some concluding remarks and topics for future work in Section 6.

In Section 5, all SDP problems for the numerical experiments in this paper are summarized. The density of the SCMs in the SDP problems and the parallel computing environment in the experiments are also described. Since the main experiments were done on a high-performance PC cluster, we also include the results for a more commonly available PC cluster in Appendix A as a reference for readers.

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

`http://sdpa.sourceforge.net/`

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

SDPARA 7.3.1 is now distributed under GPL. Furthermore, the SDPA Online Solver system described in Ref. [15] enables users to use SDPARA via the Internet at no charge. The SDPA Online Solver system consists of PC clusters with 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/`

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

The standard form of SDPs addressed in this paper is defined by the following primal-dual pair.

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

The symbol $\mathbb{S}^n$ denotes the space of $n \times n$ symmetric matrices and $\boldsymbol{X} \succeq \boldsymbol{O}$ ($\boldsymbol{X} \succ \boldsymbol{O}$) stands for $\boldsymbol{X} \in \mathbb{S}^n$ being positive semidefinite (positive definite). 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. 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 where $m \gg n$, which includes many practical applications such as SNLs and quantum chemistry problems.

Under suitable assumptions such 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 for the above system are also optimal to (1).

PDIPMs search for a point $(\boldsymbol{x}^*, \boldsymbol{X}^*, \boldsymbol{Y}^*)$ that 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. See Ref. [13] for details.

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 that would have smaller residuals $\boldsymbol{P}, \boldsymbol{d}$ and g.

Step 2. To keep the positive definiteness, evaluate the maximum lengths of possible steps $\alpha_p = \max\{\alpha : \boldsymbol{X}^h + \alpha_p d\boldsymbol{X} \succeq \boldsymbol{O}\}$ 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 approximate optimal solution. Otherwise, return to Step 1.

Step 1 contains the main bottlenecks of the above framework [38]. 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

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

where

$$
\begin{aligned}
B_{ij} &= (\boldsymbol{X}^{-1}\boldsymbol{F}_i\boldsymbol{Y}) \bullet \boldsymbol{F}_j \qquad (i = 1, 2, \ldots, m,\ j = 1, 2, \ldots, m) & (4) \\
r_k &= -d_k + \boldsymbol{F}_k \bullet (\boldsymbol{X}^{-1}(\boldsymbol{R} + \boldsymbol{P}\boldsymbol{Y})) \qquad (k = 1, 2, \ldots, m) \\
\boldsymbol{R} &= \beta \frac{\boldsymbol{X} \bullet \boldsymbol{Y}}{n} \boldsymbol{I} - \boldsymbol{X}\boldsymbol{Y}. & (5)
\end{aligned}
$$

Details on the derivation of the above computation scheme can be found in many papers [14, 22, 37, 38].

Most of the computation time of the framework is spent on the first linear system (3), which is generally known as the Schur complement equation. Its coefficient matrix is the Schur complement matrix (SCM), and the elements of SCM are evaluated by formula (4). Since the SCM is always positive definite, its Cholesky factorization is usually used to obtain $d\boldsymbol{x}$. Following the nomenclature in Ref. [38], the evaluation of the SCM will be referred to as the "ELEMENTS component", and its Cholesky factorization the "CHOLESKY component". The numerical results in Ref. [38] show that more than 80% of the total computation time is spent on the ELEMENTS and CHOLESKY components, when SDPA on a single processor is used to solve an SDP from control theory or combinatorial optimization (both of which usually have fully dense SCMs).

3 Existing parallel schemes of SDPARA 1.0.1

We describe in this section how the previous version of SDPARA (1.0.1) replaced these two bottlenecks of PDIPMs with their parallel implementations. Then, we show an SDP

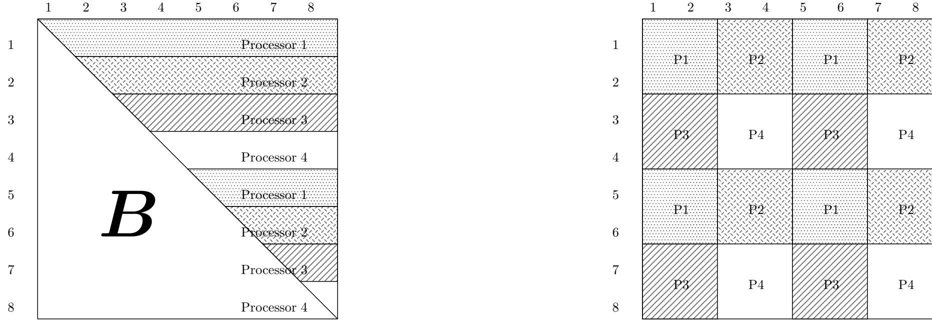

Figure 1: Parallel computation of Schur complement matrix $\boldsymbol{B}$.

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

example having a sparse SCM and how SDPARA 1.0.1 loses its effectiveness. These matters will be essential for understanding the parallel schemes proposed in the next section.

An important aspect of (4) is that the computation of each row of the SCM is independent of the other rows. For the ith row, we first multiply $\boldsymbol{U} = \boldsymbol{X}^{-1}\boldsymbol{F}_i\boldsymbol{Y}$, then take the inner products $\boldsymbol{U} \bullet \boldsymbol{F}_j$ ($j = 1, 2, \ldots, m$). We duplicate the input data matrices $\boldsymbol{F}_k$ ($k = 0, 1, \ldots, m$) on all processors before the PDIPM iterations and update the variable matrices $\boldsymbol{X}$ and $\boldsymbol{Y}$ on each processor at every iteration. Thus, any processor can evaluate any row without network communications from other processors. This property motivated the use in SDPARA 1.0.1 of a *row-wise distribution* for the ELEMENTS component [38]. Figure 1 displays an example of the row-wise distribution where the SCM is 8×8 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, we assign all the processors to the rows in a cyclic manner. More precisely, the pth processor evaluates the ith row of the SCM when $i - p$ is a multiple of the number of available processors u. If $\mathcal{R}_p$ denotes the set of row indexes assigned to the pth processor,

$$\mathcal{R}_p = \{i : 1 \leq i \leq m,\ i - p \text{ is a multiple of } u\}. \tag{6}$$

For the CHOLESKY component, which is the subsequent computation after the ELEMENTS component, we used the parallel Cholesky factorization supplied by the ScaLAPACK library [9]. To improve the performance of the parallel Cholesky factorization, SDPARA 1.0.1 redistributed the SCM from the row-wise distribution to a *two-dimensional block-cyclic distribution* whose style was assumed by ScaLAPACK for the matrix to be factorized. This distribution is illustrated in Figure 2, where the B_{17} and B_{28} elements are stored in the 2nd processor memory while the B_{34} and B_{78} elements are in the 4th processor memory.

We applied SDPARA 1.0.1 to an SDP problem from quantum chemistry, 'Be.1S.SV.pqgt1t2p' [26]. Table 1 shows the ELEMENTS, CHOLESKY, and total computation times when solving it. See Table 5 in Section 5 for the sizes of this SDP. The row-wise distribution seems to be a simple scheme to achieve satisfactory load balance, but surprisingly it produced an almost linear scalability for the ELEMENTS component, *i.e.*, 16-times speedup on 16 processors. Consequently, it produced 12-times speedup for the total computation time. These speedup factors are called *scalability* in parallel computation.

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

# Processors	1	2	4	8	16
ELEMENTS	17,236.91	8546.58	4333.07	2163.03	1069.04
CHOLESKY	191.72	96.47	57.96	39.17	23.49
Total	17,727.03	9050.55	4682.63	2492.25	1389.38

Table 2: Computation time (in seconds) for SDP from POP (BroydenBand30) by SDPARA 1.0.1 and SDPA 7.3.1 on different numbers 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

SDPARA 1.0.1 is fast for some groups of SDPs. However, we have observed over recent years that SDPARA 1.0.1 loses its advantage for the latest applications such as POPs or SNLs. Table 2 gives the computation time for the SDP problem 'BroydenBand30' generated by SparsePOP [35]. 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 when the latter uses 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 non-zeros elements of SCMs for SDPs from quantum chemistry 'Be.1S.SV.pqgt1t2p' and POP 'BroydenBand40', respectively. The left figure is an example of a fully dense SCM and 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 matrices. Suppose that all the input data matrices $\boldsymbol{F}_k$ $(k = 0, 1, \ldots, m)$ share the same

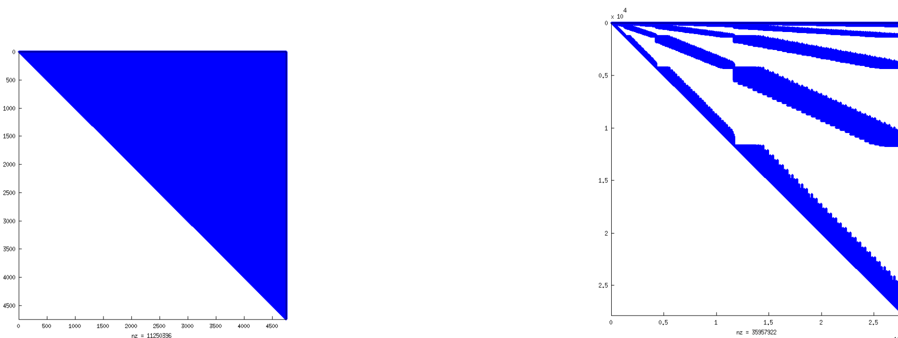

Figure 3: Non-zero elements of Schur complement matrices (only upper triangular parts) in SDPs from quantum chemistry (left) and POP (right).

diagonal block structure with matrix sizes $n_1, n_2, \ldots, n_{\bar{\ell}}$. More precisely, we 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:

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

In practice, the matrix sizes $n_1, n_2, \ldots, n_{\bar{\ell}}$ are determined by the applications and are given as input for SDP solvers. For example, the diagonal block structures of SDPs arising from quantum chemistry correspond to $P, Q, G, T1, T2'$ conditions. The sparse conversion described in Ref. [20] converts an SDP into an equivalent SDP with multiple smaller diagonal block matrices that can be input to SDP solvers.

When all the input data matrices share the same block diagonal structure, it can be easily checked that the variable matrices $\boldsymbol{X}$ and $\boldsymbol{Y}$ also share the same structure and that they 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 be decomposed to a sum over sub-matrices:

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

This breakdown demonstrates that B_{ij} becomes zero if $[\boldsymbol{F}_i]^\ell$ or $[\boldsymbol{F}_j]^\ell$ is the zero matrix for all $\ell = 1, 2, \ldots, \bar{\ell}$, so the non-zero pattern S of the SCM is determined by

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

Figure 3 illustrates this pattern. We use the symbol $|S|$ to denote the number of elements in S. The density of the SCM is then 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%, so we should avoid applying the row-wise distribution and the dense Cholesky factorization. Note that larger SDPs in POPs or SNLs have a stronger tendency to produce this situation and generate sparser SCMs. This phenomenon can be observed in Table 6. Thus, the larger SDPs become, the more we need to process the sparse SCMs in an appropriate manner.

4 New features of SDPARA 7.3.1

The new SDPARA, version 7.3.1, overcomes the limitation of SDPARA 1.0.1 in which parallel schemes are restricted to fully dense SCMs. We begin by detailing the formula-cost-based distribution that prioritizes the load balance among the processors for the sparse SCM computation. Then, we consider its Cholesky factorization. We also describe the MPI and multi-threading computation of these routines.

4.1 New load balance schemes for sparse Schur complement matrices

For the sparse matrix case, more elaborate storage management should be provided compared to the two-dimensional block-cyclic distribution for the fully dense matrix. Since the (sparsity) pattern S of the SCM is invariant through all the iterations of PDIPMs, by counting the number of elements in S, we can allocate memory space for the triples (i, j, B_{ij}) for each $(i, j) \in S$ in advance.

In the new parallel distribution for the sparse SCMs, we assign the ELEMENTS computation to each processor by dividing S into disjoint subsets $S_1, S_2, \ldots, S_u$, where u is the number of available processors. Thus, each element of the SCM is evaluated on a fixed processor through all the diagonal blocks. This concept was conceived of to address 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 ℓth 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 communications to the 3rd processor. We can not underestimate this network overhead, since $|S|$ is beyond 100 million for large-scale problems which SDPARA is designed to solve (see Table 6). In addition, the number of blocks considered for the composition of B_{ij}, *i.e.*, $\#\{\ell : [B_{ij}]^\ell \neq 0 \ (\ell = 1, 2, \ldots, \bar{\ell})\}$, varies for each $(i, j) \in S$. Therefore, the network flow over all the processors will be too complicated. These factors would prevent SDPARA from attaining better scalability. Hence, we assign the computation of each element through all the diagonal blocks to one processor.

We should consider an appropriate division of S to $S_1, S_2, \ldots, S_u$. To obtain the best performance of the CHOLESKY component described below, sequential elements of S should be assigned to the same processor. We first number 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 a 10×10 SCM with $|S|$ being 24. With this notation, the division should be in the form of

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

with delimiter points $d_0 = 0, d_1, d_2, \ldots, d_{u-1}, d_u = |S|$.

To decide the delimiter points, we take the computation cost of each element in S into account. Let $[\mathcal{F}_{ij}]^\ell$ denote the computation cost for the evaluation of $[B_{ij}]^\ell$. Then, we calculate the computation load on the pth processor $\mathcal{L}_p$ by

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

The estimation for $[\mathcal{F}_{ij}]^\ell$ can be obtained by a method originally implemented in SDPA [14]. SDPA selects the smallest-cost formula for $[B_{ij}]^\ell$ from three candidates, $\mathcal{F}_1, \mathcal{F}_2$, and $\mathcal{F}_3$,

$$\mathcal{F}_1 : \quad [\boldsymbol{U}]^\ell = [\boldsymbol{X}^{-1}]^\ell [\boldsymbol{F}_i]^\ell [\boldsymbol{Y}]^\ell, \quad [B_{ij}]^\ell = \sum_{\alpha,\beta} [\boldsymbol{U}]^\ell_{\alpha,\beta} [\boldsymbol{F}_j]^\ell_{\alpha,\beta}$$

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

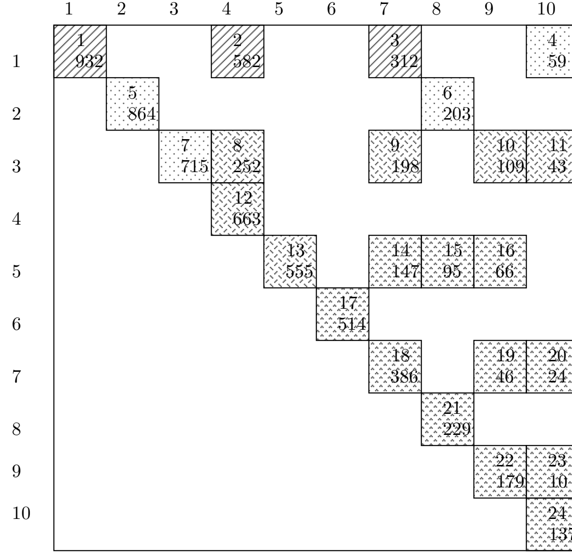

Figure 4: Example of sparse Schur complement matrix with row-wise indexes $\mathcal{N}(i,j)$ (upper numbers) and estimated costs $\sum_{\ell=1}^{\bar{\ell}}[\mathcal{F}_{ij}]^\ell$ (lower numbers).

$$\mathcal{F}_3: \qquad [B_{ij}]^\ell = \sum_{\alpha,\beta}\sum_{\gamma,\delta}[\boldsymbol{X}^{-1}]^\ell_{\alpha,\gamma}[\boldsymbol{F}_i]^\ell_{\gamma,\delta}[\boldsymbol{Y}]^\ell_{\delta,\beta}[\boldsymbol{F}_j]^\ell_{\alpha,\beta},$$

where $[\boldsymbol{F}_i]^\ell_{\alpha,\beta}$ is the (α,β) 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 non-zero elements for $[\boldsymbol{F}_k]^\ell$. 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. Note that all three formulas are composed of floating-point multiplications and additions. Since the computation cost for floating-point multiplications is greater than that of additions, each computation cost for the three formulas is approximately proportional to the number of floating-point multiplications. Therefore, the number of non-zero elements of $[\boldsymbol{F}_k]^\ell$ enables $[\mathcal{F}_{ij}]^\ell$ to be estimated.

After all $[\mathcal{F}_{ij}]^\ell$ are computed, a simple heuristic is sufficient to determine $d_0, d_1, \ldots, d_u$ that makes all of $\mathcal{L}_p$ $(p = 1,\ldots,u)$ close to their average $\widehat{\mathcal{L}} = \sum_{p=1}^{u}\mathcal{L}_p/u$, since $|S|$ can exceed millions in large-scale SDPs (see Table 6) while u is usually at most 1000. Based on this observation, the heuristic implemented in SDPARA is a simple one. For the pth processor, we enumerate $\mathcal{L}_p(d) = \sum_{(i,j)\in S: d_{p-1}<\mathcal{N}(i,j)\le d}\sum_{\ell=1}^{\bar{\ell}}[\mathcal{F}_{ij}]^\ell$ from $d = d_{p-1}+1$ until we find $\widehat{d}$ such that $\mathcal{L}_p(\widehat{d}) > \widehat{\mathcal{L}}$. If $\mathcal{L}_p(\widehat{d})$ is closer to $\widehat{\mathcal{L}}$ than $\mathcal{L}_p(\widehat{d}-1)$, we set $d_p = \widehat{d}$; otherwise, $d_p = \widehat{d}-1$.

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

4.2 Sparse Cholesky factorization of Schur complement matrices

Instead of the parallel dense Cholesky factorization of ScaLAPACK, SDPARA 7.3.1 uses the parallel *sparse* Cholesky factorization of MUMPS [2, 3, 4]. At present, MUMPS is the only software that can attain reasonable scalability for sparse factorization. MUMPS assumes that the matrix to be factorized is distributed under the *distributed assembled matrix distribution*, *i.e.*, the matrix should be stored in the style of triples (i, j, B_{ij}) of consecutive elements in the row-wise direction. The memory storage of this distribution is consistent with the formula-cost-based distribution considered before. Therefore, in contrast to dense SCMs (Section 3), we do not redistribute the matrix prior to the CHOLESKY component. This reduces the total network communication, since such redistribution would move the data of S, which exceeds millions, to scattered locations over all the processors, as similarly described in the previous sub-section.

The scalability of the parallel sparse Cholesky factorization is lower than that of the dense factorization, since MUMPS uses a complicated framework of multiple frontal methods. When the SCM is weakly sparse, the dense factorization is sometimes faster than the sparse factorization when more processors are used. SDPARA 7.3.1 can automatically determine which factorization is better with the information supplied by MUMPS and scalability information obtained by preliminary numerical experiments. In practice, however, most SDPs are almost fully dense or extremely sparse, and sensitive automatic selection is not often required. Thus, whether SDPARA uses the dense factorization or the sparse factorization is often determined based on the density of $|S|$, and this decision is usually independent of the number of available processors. More precisely, when the density is greater than 70%, SDPARA selects the dense factorization.

Table 3 shows the results of preliminary numerical experiment using SDPARA 7.3.1. The SDP solved here was 'BroydenBand600', an SDP relaxation problem of POP generated by SparsePOP [35]. Compared to a single processor, which needed 4505 seconds for the ELEMENTS component, SDPARA on 16 processors solved the problem in only 345 seconds. We obtained an 11.73-times speedup. This reduction is derived from the formula-cost-based distribution. For the CHOLESKY component, the resultant speedup was only 5.52. However, not having to redistribute the matrix before performing the CHOLESKY component is one of the advantages of SDPARA 7.3.1. Consequently, SDPARA 7.3.1 on 16 processors attains a 6.39-times speedup.

We should also consider the fill-in of the Cholesky factorization. Through the factorization process, we have non-zero elements out of S in general, and such elements are called fill-in. The number of fill-in affects the factorization performance. Table 6 shows the number of fill-in reported by MUMPS. MUMPS automatically minimizes the fill-in by heuristics; its default heuristic method is the symmetric approximate minimum degree permutation (SYMAMD). The analysis part based on SYMAMD requires about 100 seconds in the case of 'BroydenBand600'. This analysis is done only once before the main iteration of PDIPMs, but it is not parallelized. Thus, the percentage of the total computation time spent on this analysis increases from 0.8% on 1 node to 5.1% on 16 nodes. However, the low occurrence of fill-in in POPs and SNLs makes MUMPS much faster than simply applying the dense Cholesky factorization. Hence, SDPARA attains reduction of the total computation time.

We also note that the SDP 'BroydenBand600' solved here is much larger than the SDP 'BroydenBand30' shown in Table 2. SDPARA 1.0.1 could not handle 'BroydenBand600'

Table 3: Computation time (in seconds) for 'BroydenBand600' by SDPARA 7.3.1 on different numbers 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

due to lack of memory. This indicates that handling sparse SCMs appropriately enables SDPARA 7.3.1 to solve larger SDPs.

4.3 Multi-threading acceleration on MPI-based parallel computing

Multi-core architectures has become common in processors recently. This change has brought a multi-threading speedup to SDPA 7.3.1 [39]. The new SDPARA 7.3.1 combines the MPI-based parallel computing discussed so far and multi-threading enhancement in the ELEMENTS component of SDPARA 7.3.1. We first discuss the combination for dense SCMs and then move to sparse SCMs. For the CHOLESKY component, preliminary numerical experiments showed that using optimized and multi-threaded Basic Linear Algebra Subprograms (BLAS) libraries is enough to obtain benefits from the multi-threading.

As described in Section 3, the row-wise distribution is essential for reducing the computation time of the ELEMENTS component for dense SCMs. In this distribution, $\mathcal{R}_p$ in (6) denotes the set of row indexes assigned to the pth processor. All cores on the pth processor share the memory space allocated to $\mathcal{R}_p$, so each thread can store its own computed elements of the SCM into the memory space attached to the processor. Therefore, we can describe the task of each thread on the pth processor as follows. Let i_s be the smallest index in $\mathcal{R}_p$. First, choose i_s and remove it from $\mathcal{R}_p$. Then, evaluate the i_sth 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. A question may be why we do not also use this strategy in the MPI-based parallel computing. The reason is the difference in the memory access. If we use this strategy for the MPI-based computing, we can not fix $\mathcal{R}_p$. Unbalanced sizes of $\mathcal{R}_p$ produce complex network communications on the redistribution prior to the CHOLESKY component. Since the load balance of the row-wise distribution in the MPI-based computing is already satisfactory as indicated in Table 1, such complex communications would cause an overhead.

Next, we move to sparse SCMs. In the formula-cost-based distribution, the division of S into $S_1, S_2, \ldots, S_u$ decides the computation assigned to each processor. The strategy used for multi-threading here is related to the sub-divisions of $S_1, S_2, \ldots, S_u$. Let v be the number of threads. We assume that all the processors have the same number of threads. On the pth processor, we apply the formula-cost-based distribution to S_p and obtain the sub-divisions $S_p^1, S_p^2, \ldots, S_p^v$. The qth thread on the pth processor then evaluates the elements

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

		# Processors	1	2	4	8	16
Problem	# Threads						
Be.1S.SV.pqgt1t2p	1	ELEMENTS	10,984.21	5524.70	2763.75	1430.10	722.08
		CHOLESKY	161.08	81.02	44.77	31.83	19.21
		Total	11,355.65	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	12,367.39	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

of $\boldsymbol{B}$ specified by S_p^q. This strategy keeps the independence of each thread without any communication to other threads. This attribute should yield better load balance and thus higher scalability.

Table 4 shows the results of the combination of MPI-based parallel computing and multi-threading. The SCM was fully dense for 'Be.1S.SV.pqgt1t2p' while only 0.559% dense for 'BroydenBand600'. We first focus on the dense case 'Be.1S.SV.pqgt1t2p' on a single processor. The computation time for ELEMENTS was reduced from 10,984 seconds to 1585 seconds; we obtained 6.93-times speedup with 8 threads. When we used 8 threads on all 16 processors (128 threads in total), the ELEMENTS time was only 109.87 seconds; the scalability for 128 threads reached 99.9. This extremely high scalability for the ELEMENTS component and the effect of multi-threading BLAS for the CHOLESKY component remarkably improves the efficiency of the total computation. As a final result, we achieved 61.2-times speedup in the total time by using 128 threads.

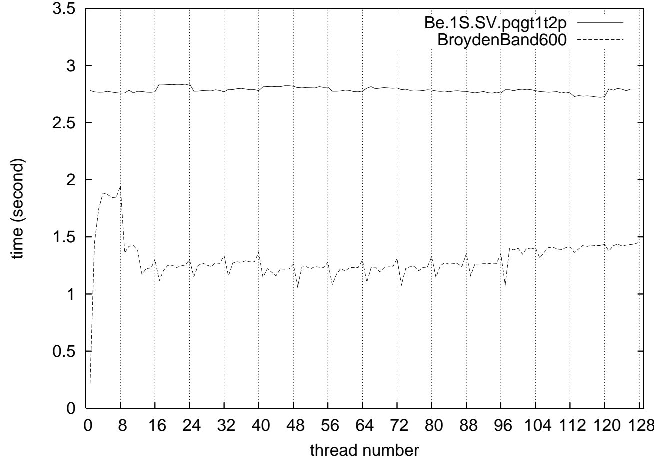

Figure 5: Load balance for ELEMENTS component of SDPARA 7.3.1 for SDPs in Table 4.

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

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

For the dense case, the maximum/minimum ratio on 128 threads was $\frac{2.84}{2.72} = 1.04$. This notable load balance generates the above 99.9-times speedup. For the sparse case, the shortest thread time was 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 has already taken the effect of sparse data structures into account, memory access for sparse data structures is sometimes too complex to be predicted. When we excluded the 1st thread, the maximum/minimum ratio was $\frac{1.94}{1.06} = 1.80$. As a result, we obtained the reasonable scalability of 75.0 on 128 threads.

5 Numerical Results

Tables 5 and 6 summarize the SDPs with their sizes and sparsity, which we used for the numerical experiments in this paper. We categorize the SDPs into four groups: POP, SNL, QC, and Mittelmann. The first group, 'POP' problems, are the SDP relaxation problems generated by SparsePOP [35]. For example, 'BroydenBand30' is generated by the SparsePOP command `sparsePOP('BroydenBand(30)');` with its default parameters. The 'SNL' group consists of sensor network localization problems generated by SFSDP [21]. The problem 'sfsdp30000' has 30,000 sensors scattered in the two dimensional square $[0,1]\times[0,1]$.

Table 5: SDP sizes for numerical experiments.

Group	Name	m	n	# Blocks	$n_{\max}$
POP	BroydenBand30	19,931	2880	24	120
	BroydenBand600	471,371	71,280	594	120
	BroydenBand800	629,771	95,280	794	120
	BroydenBand900	708,971	107,280	894	120
	ChainedSingular500	9974	4980	498	10
	ChainedSingular20000	399,974	199,980	19,998	10
	ChainedSingular30000	599,974	299,980	29,998	10
SNL	sfsdp500	7933	12,004	423	21
	sfsdp30000	452,217	808,272	29,887	44
	sfsdp35000	527,096	943,151	34,887	43
QC	Be.1S.SV.pqgt1t2p	4743	4444	21	1062
	N.4P.DZ.pqgt1t2p	7230	6010	21	1460
Mittelmann	butcher	6434	22,842	1	330
	neu3g	8007	462	1	462
	reimer5	6187	102,606	1	462
	shmup5	1800	11,041	2	3721
	taha1b	8007	1609	21	286

The noise threshold for the sensor-sensor or sensor-anchor distance was set to 10%, and the radio range was set to 0.1. Details for the 'QC' (quantum chemistry) group can be found in Ref. [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 the considered atoms/molecules. The SDPs of the 'Mittelmann' group were chosen from the benchmark problems at Mittelmann's website [24]. In the 'Mittelmann' group, we excluded small problems solvable in less than 500 seconds on a single processor with a single thread. In the POP and SNL group, some small-sized problems were chosen to compare the results of SDPARA with those of another parallel SDP solver, PCSDP [19].

In Table 5, the 3rd column corresponds to the number of equality constraints of $(\mathcal{D})$ in the standard form (1), while the 4th column n is the dimensions of $\boldsymbol{X}$ and $\boldsymbol{Y}$. The 5th column describes the number of blocks in the diagonal block structure. We did not include the blocks whose sizes were one, since they can be evaluated in the ELEMENTS component separately by a simpler way as linear programming. The 6th column, $n_{\max}$, is the largest block size defined by $n_{\max} = \max\{n_1, n_2, \ldots, n_{\bar{\ell}}\}$.

Table 6 shows the density of corresponding SCMs. Its columns provide the number of $|S|$, the number of fill-in and their sum with the corresponding density against m^2. Note that this table shows that larger SDPs have sparser SCMs in POP or SNL. We excluded QC and Mittelmann, since the SCMs in these groups are always fully dense.

All numerical experiments presented in this paper, except in Appendix A, were executed on a PC cluster composed of 16 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 a Myrinet-10G network

Table 6: Number of non-zeros and density of sparse SCMs ('K' and 'M' indicate thousands and millions, respectively.)

Group	Name	$\|S\|$		(# Fill-in)		$\|S\|$+(# Fill-in)	
POP	BroydenBand30	25M	(12.8%)	15M	(7.47%)	40M	(20.3%)
	BroydenBand600	621M	$(5.59 \times 10^{-1}\%)$	191M	$(1.72 \times 10^{-1}\%)$	812M	$(7.31 \times 10^{-1}\%)$
	BroydenBand800	830M	$(4.18 \times 10^{-1}\%)$	280M	$(1.41 \times 10^{-1}\%)$	1,110M	$(5.60 \times 10^{-1}\%)$
	BroydenBand900	934M	$(3.72 \times 10^{-2}\%)$	321M	$(1.28 \times 10^{-3}\%)$	1,255M	$(5.00 \times 10^{-2}\%)$
	ChainedSingular500	234K	$(4.81 \times 10^{-1}\%)$	0	(0%)	234K	$(4.81 \times 10^{-1}\%)$
	ChainedSingular20000	9M	$(1.20 \times 10^{-2}\%)$	3M	$(3.43 \times 10^{-3}\%)$	12M	$(1.54 \times 10^{-2}\%)$
	ChainedSingular30000	14M	$(8.00 \times 10^{-3}\%)$	4M	$(2.30 \times 10^{-3}\%)$	18M	$(1.03 \times 10^{-2}\%)$
SNL	sfsdp500	593K	$(9.44 \times 10^{-1}\%)$	47K	$(7.62 \times 10^{-2}\%)$	641K	(1.02%)
	sfsdp30000	16M	$(7.87 \times 10^{-3}\%)$	32K	$(1.57 \times 10^{-5}\%)$	16M	$(7.89 \times 10^{-3}\%)$
	sfsdp35000	18M	$(6.52 \times 10^{-3}\%)$	33K	$(1.18 \times 10^{-5}\%)$	18M	$(6.53 \times 10^{-3}\%)$

interface. The optimized BLAS we used was GotoBLAS 1.26. The stopping tolerance was usually set to $\epsilon = 1.0 \times 10^{-7}$ (see Section 2). Only for SNL problems, the stopping tolerance was relaxed to $\epsilon = 1.0 \times 10^{-5}$. We need only moderate accuracy for SNL problems, since the relaxed tolerance is sufficient to generate an appropriate starting point for the posterior local optimization methods. The results of low-accuracy SDP methods are not included in this paper. From a preliminary experiment on 'sfsdp500', SDPLR [12] did not obtain the weak accuracy $\epsilon = 1.0 \times 10^{-3}$, even 1 hour of computation, while SDPA can reach $\epsilon = 1.0 \times 10^{-5}$ in 10 seconds. We also investigated the performance of low-accuracy SDP solvers [30, 33, 36, 40]. However, their performances were very sensitive to the types of SDP to be solved; they were not successful for the SDPs from POPs or SNLs.

Tables 7 and 8 show the computation time of SDPARA 7.3.1 for SDPs with sparse SCMs and dense SCMs, respectively. We changed the number of nodes to 1, 2, 4, 8, or 16 and the number of threads to 1 or 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 failed due to MUMPS, which is not thread-safe in some parts and hence cannot adequately compute the Cholesky factorization in multi-threading computation for some SDPs.

Next, we discuss the results of Table 7 in more detail. For 'BroydenBand800' and 'BroydenBand900', SDPARA 7.3.1 can attain reasonable scalability. These SDPs are in the same SDP group as those in Tables 3 and 4 but are larger. Note that we could not solve them when using one or two nodes due to lack of memory. The new SDPARA is the first SDP solver that can solve SDPs of this group with this size. In particular, 'BroydenBand900', which has $m = 708,971$ equality constraints, can be solved in about half an hour. Without exploiting the sparsity of the SCM, even SDPARA 7.3.1 would not be able to store this large-scale SDP in memory.

We should comment on the 'O.M.' that sometimes occurred on 8 threads, but that did not happen when a single thread was used. For all the threads to 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 a single thread is used, the memory space is already over 40-GB, which is close to the maximum amount of 48-GB. Therefore, we exhausted the memory space when using 8 threads.

In the 'ChainedSingular' problems, SDPARA 7.3.1 failed to attain parallel efficiency since

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

		# Processors	1	2	4	8	16
Problem	# Threads						
N.4P.DZ.pqgt1t2p	1	ELEMENTS	35,551.82	17,844.27	8948.75	4485.27	2267.93
		CHOLESKY	629.71	278.17	150.17	95.01	54.62
		Total	36,276.29	18,655.51	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

the structure of the SCM for these problems is very simple. When we apply SYMAMD to the SCM prior to the sparse Cholesky factorization, we can detect that its factorization will be a band-diagonal matrix with a remarkably small bandwidth. Consequently, the amount of parallel computation is not significant. However, the formula-cost-based distribution for the ELEMENT component is still useful in these problems on a single thread. In a multi-threading case, all the threads encounter many conditional jumps through many small diagonal blocks. Therefore, managing the threads becomes an obstacle that lowers their parallel efficiency.

The SDP 'sfsdp35000' is the largest SDP that SFSDP can generate on 48-GB memory space. Even this largest SDP can be solved by SDPARA 7.3.1 in only 300 seconds.

We move our focus now to Table 8, in which the SDPs have fully dense SCMs. SDPARA 7.3.1 attained high scalability in both the ELEMENTS and CHOLESKY components and, as a result, the total computation time was drastically shortened, except for the 'shmup5' SDP. The speedup for 'N.4p.DZ.pqgt1t2p' on 128 threads was over 60-times. Its scalability of 69.4 was higher than the 61.2 of 'Be.1S.SV.pqgt1t2p' (Table 4). A remarkable feature of SDPARA 7.3.1 is that it can achieve higher scalability when larger SDPs are solved. In 'reimer5', the percentage of the ELEMENTS and CHOLESKY components in the total time was decreased from $\frac{906+199}{1140} = 0.97$ to $\frac{10+10}{28} = 0.72$ when we increased the number of processors and threads. This is another indicator that the parallel schemes of SDPARA 7.3.1 are significantly reinforced by multi-threading.

The problem 'shmup5' is exceptional because $m < n_{\max}$. The chief computational bottleneck in this case is the matrix multiplications such as in (5). Therefore, we can expect that even though the row-wise distribution and the parallel Cholesky factorization still reduce the computation time, their contribution is relatively small. However, note that multi-threading did substantially reduce the total time for this SDP.

Finally, Table 9 compares the performance of SDPARA 7.3.1 with that of PCSDP 1.0r1 [19]. PCSDP is another parallel SDP solver, based on CSDP developed by Borchers [10]. We do not consider PDSDP [5], since it has not been updated in the last five years. In the SDPARA 7.3.1 experiments, we used 1 or 8 threads, while we set it 8 threads for PCSDP 1.0r1; thus, the difference due to the performance of BLAS disappears when SDPARA 7.3.1 uses 8 threads. Since PCSDP 1.0r1 does not print out the required computation time, we used the Linux `time` command instead, so the smallest time unit reportable is one second when the total time exceeds one hour. We set the default parameters of SDPARA 7.3.1 and PCSDP 1.0r1, respectively, except for SNLs where the stopping tolerance was set to $\epsilon = 1.0 \times 10^{-5}$.

SDPARA 7.3.1 on a single thread solved all SDPs with sparse SCMs faster than PCSDP 1.0r1 on 8 threads whenever both solvers reached optimal solutions. PCSDP 1.0r1 could not solve three of the problems due to insufficient memory. Furthermore, multi-threading widened the gap between SDPARA 7.3.1 and PCSDP 1.0r1 for 'BroydenBand30', while for problems such as 'ChainedSingular500' or 'sfsdp500', the variable matrices sizes were too small to obtain an advantage 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 in dense factorization than in a sparse one, the scalability of PCSDP 1.0r1 seems to be better than that of SDPARA 7.3.1. However, if we focus on the computation time itself, the difference is obvious. Furthermore, the behavior of PCSDP 1.0r1 is some-

times unstable. For example, PCSDP 1.0r1 on 2 processors solved 'sfsdp500' 5 times faster than on a single processor. An unknown reason makes PCSDP 1.0r1 unusually slow on a single processor.

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

6 Concluding remarks and future work

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 these schemes successfully reduce the total computation time. In addition, multi-threading substantially enhanced the parallel performance of SDPARA 7.3.1 for all cases. In particular, SDPARA 7.3.1 solved the extremely large-scale BroydenBand-type SDPs that other solvers cannot run. We expect this solid ability to solve SDPs will enrich research on POPs and SNLs and extend the range of SDP applications.

The results of this paper motivate future research topics including the following two. One challenge is how we solve SDPs with m(number of equality constraints) $< n$(size of block-diagonal variable m
, which includes many practical applications. As pointed out, SDPARA usually addresses SDPs with $m > n$, where we mainly resolved the bottlenecks related to the SCM. Even in cases where $m < n$, SDPARA might be accelerated by distributing the variable matrices $\boldsymbol{X}$ and $\boldsymbol{Y}$ over all the processors. However, this distribution would require complex network communication. Whether this network overhead can be justified by the reduction of the bottlenecks related to $\boldsymbol{X}$ and $\boldsymbol{Y}$ remains in question. The other issue is whether there is be space for further improvements on multi-threading. Simultaneous access of same data by multiple threads sometimes interferes with the performance. Knowledge about the memory hierarchy in computer architecture will provide important information for this case.

Appendix A: Numerical results on a more commonly available system

The numerical results in Section 5 were obtained on our best PC cluster (hereafter called "Cluster1"). Since we aimed to solve the largest SDPs in a short time, the specifications on Cluster1 are considerably advanced; in particular, all the nodes are connected with a fast network.

For a reference, we report the numerical results for a more commonly available cluster. This cluster is our least-powered one at present (hereafter called "Cluster2") and is composed of two nodes. Each node has 24-GB memory space and a Xeon W3520 (2.67 GHz : 4 CPU cores) processor, and the nodes are connected by 1-Gbps Ethernet.

The main differences between Cluster1 and Cluster2 are the number of available threads on each node and the network speed. Cluster1 has 8 threads while Cluster2 has 4. All the

Table 9: Computation time comparison (in seconds) between SDPARA 7.3.1 (1 or 8 threads) and PCSDP 1.0r1 (8 threads) on 1, 2, 4, 8, or 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)	11,355.65	5812.68	3005.04	1657.17	932.64
	SDPARA(8)	1736.17	927.87	505.83	288.39	185.62
	PCSDP(8)	12,704.00	6575.00	3485.84	1972.93	1165.95
N.4P.DZ.pgqt1t2	SDPARA(1)	36,276.29	18,655.51	9604.72	5071.99	2803.00
	SDPARA(8)	5716.05	2931.07	1577.61	887.01	522.37
	PCSDP(8)	53,577.00	27,672.00	15,178.00	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)	11,350.00	11,005.00	10,480.00	11,122.00	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

Table 10: Performance of SDPARA 7.3.1 for 'Be.1S.SV.pqgt1t2p' and 'BroydenBand30' on two clusters. (Time unit is seconds; numbers between parentheses are corresponding scalabilities.)

Problem/Cluster	# Processor(s)	#Thread	ELEMENTS	CHOLESKY	Total
Be.1S.SV.pqgt1t2p	1	1	10,984.20 (1.00)	161.08 (1.00)	11,355.64 (1.00)
/Cluster1	2	4	1412.61 (7.77)	32.54 (4.95)	1544.24 (7.35)
		8	819.63 (13.40)	25.63 (6.28)	927.87 (12.24)
Be.1S.SV.pqgt1t2p	1	1	12,956.03 (1.00)	188.30 (1.00)	13,327.05 (1.00)
/Cluster2	2	4	2397.92 (5.41)	67.09 (2.80)	2592.92 (5.14)
BroydenBand30	1	1	175.55 (1.00)	373.81 (1.00)	564.55 (1.00)
/Cluster1	2	4	33.41 (5.25)	145.94 (2.56)	189.80 (2.97)
		8	22.54 (7.79)	134.55 (2.77)	167.73 (3.37)
BroydenBand30	1	1	164.36 (1.00)	355.07 (1.00)	526.54 (1.00)
/Cluster2	2	4	33.49 (4.90)	190.43 (1.86)	229.45 (2.29)

nodes in Cluster1 are connected with a high-speed network (Myrinet-10G), while those in Cluster2 are with a standard network (1-Gbps Ethernet).

Table 10 reports the numerical results obtained on Cluster1 and Cluster2 for 'Be.1S.SV.pqgt1t2p' and 'BroydenBand30', which were selected from Table 5 as cases of dense and sparse SCMs. We selected the smallest problem, 'BroydenBand30', since 'BroydenBand600' in Table 4 consumes 36-GB memory space, beyond the available 24-GB memory space of each node. In Table 10, the first column shows the problem name and the cluster. The second and third columns indicate the numbers of nodes and threads, respectively. The fourth, fifth, and sixth columns report the computation time with corresponding scalability on ELEMENTS, CHOLESKY, and the total time, respectively.

We observe that the CHOLESKY component was affected by the network speed. The scalability on 2 nodes with 4 threads dropped from 4.95 on Cluster1 to 2.80 on Cluster2 in the dense case, and from 2.56 to 1.86 in the sparse case. However, applying parallel computation contributed to reducing in the computation time of this component.

For the ELEMENTS component on 2 nodes with 4 threads, the scalability of Cluster2 was lower than that of Cluster1, since Cluster2 uses the maximum number of threads and Cluster1 still has remaining threads (by the same reason, the scalability of Cluster1 on 2 nodes with 8 threads has lower efficiency than on 2 nodes with 4 threads). However, the drop in the scalability of Cluster2 from that of Cluster1 is small, from 7.77 to 5.41 in the dense case and from 5.25 to 4.90 in the sparse case. Since the ELEMENTS component does not involve network communication between threads, SDPARA 7.3.1 can maintain high scalability through the hybrid distribution and the formula-cost-based distribution.

Note that even on the commonly available cluster with at most 8 threads, SDPARA 7.3.1 can obtain 5.25-times speedup in the dense case and 2.29-times speedup in the sparse case. Thus, SDPARA 7.3.1 enables remarkable reduction of the total computation time. Furthermore, SDPARA 7.3.1 can attain higher performance on higher-specification clusters.

References

[1] 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 .

[2] 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.

[3] 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.

[4] 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.

[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. Fukuda, K. Kobayashi, M. Kojima, K. Nakata, M. Nakata, and M. Yamashita, SDPA (SemiDefinite Programming Algorithm) and SDPA-GMP User's Manual — Version 7.1.1, Research Report B-448, Dept. of Math. and Comp. Sciences, Tokyo Institute of Technology, Oh-Okayama, Meguro, Tokyo 152-8552, June 18th 2008 updated.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[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, M. Mevissen, and M. Yamashita, Exploiting sparsity in linear and nonlinear matrix inequalities via positive semidefinite matrix completion, Research Report B-452, Dept. of Math. and Comput. Sciences, Tokyo Institute of Technology, Oh-Okayama, Meguro, Tokyo 152-8552, January 2009.

[21] 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.

[22] M. Kojima, S. Shindoh, and S. Hara, Interior-point methods for the monotone semidefinite linear complementarity problems in symmetric matrices, *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 (14 pages).

[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] J. Povh, F. Rendl, and A. Wiegele, A boundary point method to solve semidefinite programming, *Computing* **78**, (2006), 277–286.

[31] 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.

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

[33] K.-C. Toh, Solving large scale semidefinite programs via an iterative solver on the augemented systems, *SIAM J. Optim* **14**, (2003), 670–698.

[34] K.-C. Toh, M. J. Todd, 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.

[35] 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)**, (2009), article 15 (13 pages).

[36] Z. Wen, D. Goldfarb, and W. Yin, Alternating direction augmented Lagrangian methods for semidefinite programming, *Math. Program. Comput.* **2**, (2010), 203–230.

[37] 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.

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

[39] 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.

[40] X. Y. Zhao, D. F. Sun and K. C. Toh, A Newton-CG augmented Lagrangian method for semidefinite programming, *SIAM J. Optim.* **20**, (2010), 1737-1765.