A Parallel Primal-Dual Interior-Point Method for Semidefinite Programs Using Positive Definite Matrix Completion

Kazuhide Nakata[†], Makoto Yamashita[‡], Katsuki Fujisawa[⋆] and Masakazu Kojima[♯]

Abstract. A parallel computational method SDPARA-C is presented for SDPs (semidefinite programs). It combines two methods SDPARA and SDPA-C proposed by the authors who developed a software package SDPA. SDPARA is a parallel implementation of SDPA and it features parallel computation of the elements of the Schur complement equation system and a parallel Cholesky factorization of its coefficient matrix. SDPARA can effectively solve SDPs with a large number of equality constraints; however, it does not solve SDPs with a large scale matrix variable with similar effectiveness. SDPA-C is a primal-dual interior-point method using the positive definite matrix completion technique by Fukuda *et al*, and it performs effectively with SDPs with a large scale matrix variable, but not with a large number of equality constraints. SDPARA-C benefits from the strong performance of each of the two methods. Furthermore, SDPARA-C is designed to attain a high scalability by considering most of the expensive computations involved in the primal-dual interior-point method. Numerical experiments with the three parallel software packages SDPARA-C, SDPARA and PDSDP by Benson show that SDPARA-C efficiently solves SDPs with a large scale matrix variable as well as a large number of equality constraints with a small amount of memory.

Key words.

Semidefinite Program, Primal-Dual Interior-Point Method, Parallel Computation, Positive Definite Matrix Completion, Numerical Experiment, PC Cluster

† Department of Industrial Engineering and Management, Tokyo Institute of Technology, 2-12-1 Oh-Okayama, Meguro-ku, Tokyo 152-8552 Japan. *knakata@me.titech.ac.jp*

‡ Department of Mathematical and Computing Sciences, Tokyo Institute of Technology, 2-12-1 Oh-Okayama, Meguro-ku, Tokyo 152-8552 Japan. *Makoto.Yamashita@is.titech.ac.jp*

⋆ Department of Mathematical Sciences, Tokyo Denki University, Ishizuka, Hatoyama, Saitama 350-0394 Japan. *fujisawa@r.dendai.ac.jp*

♯ Department of Mathematical and Computing Sciences, Tokyo Institute of Technology, 2-12-1 Oh-Okayama, Meguro-ku, Tokyo 152-8552 Japan. *kojima@is.titech.ac.jp*

1 Introduction

Semidefinite programs (SDPs) have gained a lot of attention in recent years with applications arising from a diverse range of fields including structural optimization, system and control, statistics [4], financial engineering [14], and quantum chemistry [24]. The solution of an SDP has been found using primal-dual interior-point methods. The development of efficient primal-dual interior-point methods has become a key issue to handle SDPs. For more information on SDP and the primal-dual interior-point method, see e.g., the papers [21, 29].

Many software packages for solving SDPs with the primal-dual interior-point method have been developed and are now available on the Internet. These include SDPA [30] developed by the authors, and various other packages such as CDSP [6], SeDuMi [25] and SDPT3 [27]. These software packages are known to be successful to solve small to medium size SDPs. However, solving large-scale SDPs still remains a difficult problem. Some of the difficult issues involving large-scale SDPs are being unable to store the matrix variables in the computer memory or being unable to run the algorithm to completion within a practical time scale. A variety of methods have been proposed to address these issues, including methods in which an iterative solution is applied to the Schur complement equation system (a linear equation system needed to compute a search direction vector) [10, 23, 26], interior-point methods that exploit the sparse structure of the problem [8, 11], a dual interior-point method DSDP [2] and its parallel implementation PDSDP [1], the spectral bundle method [16], first-order nonlinear programming methods [9, 16] and a generalized augmented Lagrangian method [18].

The amount of data storage needed for the matrix variables of the primal and dual problems corresponds to the size of the data matrix in the standard form SDP and its dual problem. When the data matrix is sparse, the dual matrix variable inherits this sparsity but in general the matrix variable of the primal problem forms a completely dense matrix. This is a major drawback of the primal-dual interior-point method. To address this drawback, SDPA-C (the primal-dual interior-point method using (positive definite) matrix completion) was proposed [12, 22]. This method was developed by adapting SDPA to use (positive definite) matrix completion theory to perform a sparse factorization of the primal matrix variable, and by incorporating matrix computations that take advantage of this sparsity. Since this method does not store the primal matrix variable directly, the amount of memory used for the primal matrix variable can be greatly reduced.

The computation times can also be reduced because this method does not perform any computation in which the dense matrix is handled directly, so that SDPs having a large but sparse data matrix can be solved in a shorter time with less memory. But on the other hand, the coefficient matrix of the Schur complement equation system – which is solved at each iteration of the primal-dual interior-point method – usually forms a dense positive definite matrix even when the data matrix is sparse (see [3, 28] for some exceptional cases). Furthermore, its size matches the number of linear equality constraints in the primal problem. As a result, when SDPA-C is used for ordinary large-scale SDP problems, it does not reach a solution efficiently because there are bottlenecks in the computation of the coefficient matrix of the Schur complement equation system and in the Cholesky factorization.

In the paper [31], we proposed SDPARA (SemiDefinite Programming Algorithm PARAllel Version). Since the above bottlenecks occur when solving SDPs where the primal problem has a large number of linear equality constraints, SDPARA uses tools such as MPI [15] and ScaLAPACK [5] to apply parallel computing to the computation of the coefficient matrix in the Schur complement equation system and to the Cholesky factorization. As a result, the computation times and memory requirements relating to the coefficient matrix of the Schur complement equation system and the Cholesky factorization have been greatly reduced. However, with this method, since the dense primal matrix variables are stored and manipulated directly, the increased size of the SDP data

matrix results in requiring more time and memory for the computations relating to the primal matrix variables.

In this paper we propose SDPARA-C, which is obtained by combining and enhancing the primal-dual interior-point method of SDPA-C using matrix completion theory [12, 22] and the parallel primal-dual interior-point method of SDPARA [31]. This allows us to retain benefits of SDPA-C and SDPARA while compensating for their drawbacks. The Cholesky factorization of the coefficient matrix of the Schur complement equation system – which is solved at each iteration of the primal-dual interior-point method – is made to run in parallel in the same way as in [31]. Also, the computation of this coefficient matrix is improved so that the load is kept equally balanced based on the method proposed in [31]. Furthermore, we arrive at a highly scalable implementation by employing parallel processing for all the computations that take up most of the computation time in other places where bottlenecks are liable to occur. This makes it possible to solve SDPs with large sparse data matrices and large numbers of linear equality constraints in less time and with less memory.

This paper is organized as follows. In section 2, we present an overview of SDPA-C and SDPARA and their drawbacks. In section 3 we propose SDPARA-C. In section 4 we perform numerical experiments to evaluate the efficiency of SDPARA-C, and in section 5 we present our conclusions.

2   Previous studies

In this section we first discuss the standard interior-point method for solving SDPs. We then present an overview of techniques in which parallel processing is applied to the primal-dual interior-point method involving the use of matrix completion theory as proposed in the papers [12, 22], and the primal-dual interior-point method proposed in the paper [31], and we describe some issues associated with these techniques.

2.1   Solving SDPs with the primal-dual interior-point method

Let $\mathcal{S}^n$ denote the vector space of $n \times n$ symmetric matrices. For a pair of matrices $\boldsymbol{X}, \boldsymbol{Y} \in \mathcal{S}^n$, the inner product is defined as $\boldsymbol{X} \bullet \boldsymbol{Y} = \sum_{i=1}^{n} \sum_{j=1}^{n} X_{ij} Y_{ij}$. We use the notation $\boldsymbol{X} \in \mathcal{S}_+^n$ ($\mathcal{S}_{++}^n$) to indicate that $\boldsymbol{X} \in \mathcal{S}^n$ is positive semidefinite (or positive definite). Given $\boldsymbol{A}_p \in \mathcal{S}^n$ ($p = 0, 1, \ldots, m$) and $\boldsymbol{b} = (b_1, b_2, \ldots, b_m)^T \in \mathbb{R}^m$, the standard form SDP is written as follows:

$$\left.\begin{array}{ll} \text{minimize} & \boldsymbol{A}_0 \bullet \boldsymbol{X} \\ \text{subject to} & \boldsymbol{A}_p \bullet \boldsymbol{X} = b_p \ (p = 1, 2, \ldots, m), \ \boldsymbol{X} \in \mathcal{S}_+^n \end{array}\right\}. \tag{1}$$

The corresponding dual problem is as follows:

$$\left.\begin{array}{ll} \text{maximize} & \sum_{p=1}^{m} b_p z_p \\ \text{subject to} & \sum_{p=1}^{m} \boldsymbol{A}_p z_p + \boldsymbol{Y} = \boldsymbol{A}_0, \ \boldsymbol{Y} \in \mathcal{S}_+^n \end{array}\right\}. \tag{2}$$

It is known that primal-dual interior-point methods are capable of solving these problems in polynomial time. There have been proposed many primal-dual interior-point methods such as the path-following method and the Mehrotra-type predictor-corrector method, and various types of search directions used therein such as the HRVW/KSH/M search direction and the NT search direction. Although most of implementation of primal-dual interior-point methods employ

2

the Mehrotra-type predictor-corrector method due to its computational efficiency, we choose a simpler path-following method to effectively exploit sparsity of SDPs using the matrix completion technique; the Mehrotra-type predictor-corrector method, which is more sophisticated than the path-following method, would not fit the matrix completion technique [12, 22]. We also use the HRVW/KSH/M search direction. The HRVW/KSH/M search direction $(d\boldsymbol{X}, d\boldsymbol{Y}, d\boldsymbol{z}) \in \mathcal{S}^n \times \mathcal{S}^n \times \mathbb{R}^m$ is defined as "the Newton-type direction toward the central path", and is computed by reducing it to a system of linear equations in the vector variable $d\boldsymbol{z} \in \mathbb{R}^m$ known as the Schur complement equation system [11, 19]. The path-following primal-dual interior-point method with the use of the HRVW/KSH/M search direction can be summarized as described below.

Algorithm 2.1: The basic primal-dual interior-point method

Step 0: Set up parameters $\gamma_1, \gamma_2 \in (0, 1)$, choose a starting point $(\boldsymbol{X}, \boldsymbol{Y}, \boldsymbol{z}) \in \mathcal{S}^n_{++} \times \mathcal{S}^n_{++} \times \mathbb{R}^m$, and set $\mu := \frac{1}{n}\boldsymbol{X} \bullet \boldsymbol{Y}$.

Step 1: Compute the search direction $(d\boldsymbol{X}, d\boldsymbol{Y}, d\boldsymbol{z}) \in \mathcal{S}^n \times \mathcal{S}^n \times \mathbb{R}^m$.

    Step 1a: Set $\mu := \gamma_1 \mu$ and compute the residuals $\boldsymbol{r} \in \mathbb{R}^m$, $\boldsymbol{R} \in \mathcal{S}^n$, $\boldsymbol{C} \in \mathbb{R}^{n \times n}$ defined as follows:
$$r_p := b_p - \boldsymbol{A}_p \bullet \boldsymbol{X} \quad (p = 1, 2, \ldots, m),$$
$$\boldsymbol{R} := \boldsymbol{A}_0 - \sum_{p=1}^m \boldsymbol{A}_p z_p - \boldsymbol{Y}, \qquad \boldsymbol{C} := \mu \boldsymbol{I} - \boldsymbol{X}\boldsymbol{Y}.$$

    Step 1b: Compute the $m \times m$ matrix $\boldsymbol{B}$ and the vector $\boldsymbol{s} \in \mathbb{R}^m$.
$$
\begin{aligned}
B_{pq} &:= \boldsymbol{A}_p \bullet \boldsymbol{X}\boldsymbol{A}_q\boldsymbol{Y}^{-1} \quad (p, q = 1, 2, \ldots, m), \\
s_p &:= r_p - \boldsymbol{A}_p \bullet (\boldsymbol{C} - \boldsymbol{X}\boldsymbol{R})\boldsymbol{Y}^{-1} \quad (p = 1, 2, \ldots, m).
\end{aligned}
\tag{3}
$$

    It is known that $\boldsymbol{B}$ is usually a fully dense positive definite symmetric matrix [11] (see [3, 28] for some exceptional cases).

    Step 1c: Solve the Schur complement equation system $\boldsymbol{B}d\boldsymbol{z} = \boldsymbol{s}$ to find $d\boldsymbol{z}$.

    Step 1d: Compute $d\boldsymbol{Y}$ and $d\boldsymbol{X}$ from $d\boldsymbol{z}$.
$$d\boldsymbol{Y} := \boldsymbol{R} - \sum_{p=1}^m \boldsymbol{A}_p dz_p, \quad \widetilde{d\boldsymbol{X}} := (\boldsymbol{C} - \boldsymbol{X}d\boldsymbol{Y})\boldsymbol{Y}^{-1}, \quad d\boldsymbol{X} := (\widetilde{d\boldsymbol{X}} + \widetilde{d\boldsymbol{X}}^T)/2.$$

Step 2: Determine the largest step size α_p, α_d in the search direction $(d\boldsymbol{X}, d\boldsymbol{Y}, d\boldsymbol{z})$.
$$\alpha_p := -1.0/\lambda_{\min}(\sqrt{\boldsymbol{X}}^{-1}d\boldsymbol{X}\sqrt{\boldsymbol{X}}^{-T}), \quad \alpha_d := -1.0/\lambda_{\min}(\sqrt{\boldsymbol{Y}}^{-1}d\boldsymbol{Y}\sqrt{\boldsymbol{Y}}^{-T}).$$

where $\sqrt{\boldsymbol{G}}$ denotes a matrix that satisfies $\sqrt{\boldsymbol{G}}\sqrt{\boldsymbol{G}}^T = \boldsymbol{G} \in \mathcal{S}^n_{++}$, and $\lambda_{\min}(\boldsymbol{H})$ is the minimum eigenvalue of matrix $\boldsymbol{H} \in \mathcal{S}^n$.

Step 3: Update the iteration point $(\boldsymbol{X}, \boldsymbol{Y}, \boldsymbol{z})$ from the search direction and the step size, and return to Step 1.
$$\boldsymbol{X} := \boldsymbol{X} + \gamma_2\alpha_p d\boldsymbol{X} \in \mathcal{S}^n_{++}, \ \ \boldsymbol{Y} := \boldsymbol{Y} + \gamma_2\alpha_d d\boldsymbol{Y} \in \mathcal{S}^n_{++}, \ \ \boldsymbol{z} := \boldsymbol{z} + \gamma_2\alpha_d d\boldsymbol{z}.$$

A characteristic of Algorithm 2.1 is that the matrices $\boldsymbol{X}$ and $\boldsymbol{Y}$ obtained at each iteration are positive definite, which is how the interior-point method gets its name. In practice, the efficiency of Algorithm 2.1 will greatly differ depending on what sort of algorithms and data structures are used. Our aim is to solve large-scale sparse SDPs with the smallest possible computation time and memory usage. To clarify the following discussion, we classify the parts that consume the bulk of the computation time into four types based on SDPA6.0 [30]

3

- Computing the value of each element in the dense matrix $\boldsymbol{B} \in \mathcal{S}_{++}^m$ at Step 1b.

- Performing the Cholesky factorization of $\boldsymbol{B} \in \mathcal{S}_{++}^m$ when solving the Schur complement equation system $\boldsymbol{B}d\boldsymbol{z} = \boldsymbol{s}$ at Step 1c.

- Computing the matrix variable $\widetilde{d\boldsymbol{X}}$ at Step 1d.

- Performing computations involving other $n \times n$ dense matrices such as $\boldsymbol{X}$ and $\boldsymbol{Y}^{-1}$.

These categories correspond to the parts where parallel processing is introduced as described later in this paper. When solving a large-scale SDP, the acquisition of memory also forms a bottleneck in many cases. Therefore we also classify the processes according to their memory requirements. When using Algorithm 2.1 to solve an SDP, the bulk of the memory consumption can be classified into the following two categories:

- The dense matrix $\boldsymbol{B} \in \mathcal{S}_{++}^m$

- $n \times n$ matrices such as the SDP matrix variables $\boldsymbol{X}$ and $\boldsymbol{Y}$ and their inverses.

In the remainder of this section, based on the aforementioned classifications of computation time and memory usage, we clarify the characteristics of SDPA-C involving the use of matrix completion technique [12, 22] and SDPARA [31] in which parallel processing is applied.

2.2   A primal-dual interior-point method using matrix completion

The papers [12, 22] propose a method in which matrix completion theory is applied to the primal-dual interior-point method in an efficient manner. These papers propose two methods – a completion method and a conversion method – but here we discuss the completion method (SDPA-C), which is related to the present paper.

In SDPA-C, crucial roles are played by the sparse factorization of the matrix variable $\boldsymbol{X}$ of the primal problem (1) and the matrix variable $\boldsymbol{Y}$ of the dual problem (2). To discuss this sparse factorization, we introduce a few concepts. First, we define the aggregate sparsity pattern of the input data matrices $\boldsymbol{A}_p$ $(p = 0, 1, \ldots, m)$ of SDPs (1) and (2). Assume V is the set $\{1, 2, \ldots, n\}$ of row/column indices of an $n \times n$ symmetric matrix. The aggregate sparsity pattern E of the input data matrices $\boldsymbol{A}_p$ $(p = 0, 1, \ldots, m)$ is the set defined as

$$E = \{(i,j) \in V \times V : i \neq j,\ [\boldsymbol{A}_p]_{ij} \neq 0,\ \exists p \in \{0, 1, 2, \ldots, m\}\},$$

where $[\boldsymbol{A}_p]_{ij}$ denotes the (i,j)th element of matrix $\boldsymbol{A}_p$. Here, if V is assumed to be a set of vertices and E is assumed to be a set of edges, then (V, E) describes a single graph. Next we discuss the extended sparsity pattern. The extended sparsity pattern F is a superset of the aggregate sparsity pattern E such that the graph (V, F) derived from V and F is a chordal graph. The definition and properties of chordal graphs are described in detail in the paper [13]. The primal-dual interior-point method described in this section uses this extended sparsity pattern to increase the computational efficiency. Accordingly, the extended sparsity pattern F should ideally be as sparse as possible. The construction of an extended sparsity pattern of this type is described in detail in the paper [12]. At an iteration point $\boldsymbol{X} \in \mathcal{S}_{++}^n$ of the primal-dual interior-point method, the parts that are not included in the extended sparsity pattern F – i.e., $\{X_{ij} : (i,j) \notin F\}$ are totally unrelated to the objective function $\boldsymbol{A}_0 \bullet \boldsymbol{X}$ of the primal problem (1) of the SDP or to the linear constraints $\boldsymbol{A}_p \bullet \boldsymbol{X} = b_p$ $(p = 1, 2, \ldots, m)$. In other words, these elements only affect the positive definiteness condition $\boldsymbol{X} \in \mathcal{S}_{++}^n$. Therefore, if a positive definite matrix $\widehat{\boldsymbol{X}}$ can be obtained by adding suitable

values to the parts $\{X_{ij} : (i,j) \notin F\}$ that are not included in the extended sparsity pattern F, then this matrix $\widehat{X}$ can be used as the iteration point. This process of arriving at a positive definite matrix by determining suitable values is called *(positive definite) matrix completion.* According to matrix completion theory, when (V, F) is a chordal graph, the matrix $\widehat{X}$ that has been subjected to positive definite matrix completion in the way to maximize its determinant value can be factorized as $\widehat{X} = M^{-T}M^{-1}$ [12]. Here M is a sparse matrix that only has nonzero elements in parts of the extended sparsity pattern F, and forms a matrix in which a suitable permutation is made to the rows and columns of a lower triangular matrix. Furthermore, the matrix variable Y can be factorized as $Y = NN^T$. Here N is a sparse matrix that only has nonzero elements in parts of the extended sparsity pattern F, and forms a matrix in which a suitable permutation is made to the rows and columns of a lower triangular matrix. In the primal-dual interior-point method that uses matrix completion theory as proposed in the paper [22], the computations at each step of Algorithm 2.1 are performed using this sparse factorization as much as possible. In particular, the sparse matrices M and N are stored instead of the conventional dense matrices X and Y^{-1}, and this sparsity is exploited when implementing the primal-dual interior-point method. Note also that both M^{-1} and N^{-1} become dense in general.

For this paper we fully updated SDPA-C software package which incorporates new matrix completion techniques based on SDPA 6.0 [30], which is the latest version of SDPA. Table 1 shows numerical results obtained when solving two SDPs with SDPA 6.0 and with SDPA-C. Problem A and B are SDP relaxations of maximum cut problems and maximum clique problems, respectively which are described in section 4. The size of problem A is $n = 1000, m = 1000$, and the size of problem B is $n = 1000, m = 1891$.

Table 1: The relative practical benefits of SDPA and SDPA-C

	Problem A		Problem B	
	SDPA	SDPA-C	SDPA	SDPA-C
Computation of B	2.2s	20.3s	82.0s	662.8s
Cholesky factorization of B	3.5s	3.7s	25.3s	34.1s
Computation of $\widetilde{dX}$	51.7s	9.2s	69.4s	32.6s
Dense matrix computation	96.1s	1.1s	125.7s	2.6s
Total computation time	157s	34s	308s	733s
Storage of B	8MB	8MB	27MB	27MB
Storage of $n \times n$ matrix	237MB	4MB	237MB	8MB
Total memory usage	258MB	14MB	279MB	39MB

To solve problem A, the SDPA software needed 15 iterations of the primal-dual interior-point method and the SDPA-C software needed 16. For problem B, SDPA needed 20 iterations and SDPA-C needed 27. The reason why SDPA-C needed more iterations is that SDPA is a Mehrotra-type predictor-corrector primal-dual interior-point method, whereas SDPA-C is a simpler path-following primal-dual interior-point method. In SDPA-C the dense matrices X and Y^{-1} are not stored directly, but instead the sparse factorizations $M^{-T}M^{-1}$ and $N^{-T}N^{-1}$ are used. Consequently, less time is needed for the computation of $\widetilde{dX}$ and the computations involving dense $n \times n$ matrices. Furthermore, there is no need to store any $n \times n$ dense matrices, so the memory requirements are correspondingly much lower than those of SDPA. However, more computation time is needed to evaluate each element of the coefficient matrix $B \in \mathcal{S}_{++}^m$ of the Schur complement equation system $Bdz = s$. This is because the matrices X and Y^{-1} are not stored directly, making it

impossible for SDPA-C to use computational methods that exploit the sparsity of the input data matrices $\boldsymbol{A}_p$ ($p = 1, 2, \ldots, m$) used in SDPA [11]. Similar computations are used for the Cholesky factorization of $\boldsymbol{B}$, so compared with conventional SDPA the computation time per iteration and the memory requirements are unchanged. Consequently there are some cases (e.g., problem A) where SDPA-C achieves substantial reductions in computation times and memory requirements compared with SDPA, while there are other cases (e.g., problem B) where although the memory requirements are substantially lower, the overall computation time is larger. In general to solve a sparse SDP with large n, SDPA-C needs much less memory than SDPA. On the other hand, the computation time of SDPA-C also includes the increased amount of computation needed to obtain the value of each element of $\boldsymbol{B} \in \mathcal{S}_{++}^m$, so the benefits of matrix completion can sometimes be marginal or even non-existent depending on the structure of the problem. In SDPs with a large m, most of the overall computation times and memory requirements are taken up by the Cholesky factorization of $\boldsymbol{B} \in \mathcal{S}_{++}^m$, so in such cases the use of matrix completion has little effect.

2.3 The parallel computation of primal-dual interior-point methods

The paper [31] proposes the SDPARA software which is based on SDPA and which solves SDPs by executing the primal-dual interior-point method on parallel CPUs (PC clusters). In SDPARA, two parts – the computation of the value of each element of the coefficient matrix $\boldsymbol{B} \in \mathcal{S}_{++}^m$ of the Schur complement equation system $\boldsymbol{B}dz = \boldsymbol{s}$ at step 1b of Algorithm 2.1 and the Cholesky factorization of $\boldsymbol{B} \in \mathcal{S}_{++}^m$ at step 1c – are performed by parallel processing.

We first describe how parallel processing can be used to obtain the values of each element in $\boldsymbol{B} \in \mathcal{S}_{++}^m$. Each element of $\boldsymbol{B}$ is defined by equation (3), where each row can be computed independently. Consequently, it can be processed in parallel by allocating each row to a different CPU. To represent the distribution of processing to a total of u CPUs, the m indices $\{1, 2, \cdots, m\}$ are partitioned into P_i ($i = 1, 2, \cdots, u$) groups. That is,

$$\cup_{i=1}^u P_i = \{1, 2, \cdots, m\}, \quad P_i \cap P_j = \emptyset \text{ if } i \neq j.$$

Here, the ith CPU computes each element in the pth row of $\boldsymbol{B}$ ($p \in P_i$). The ith CPU only needs enough memory to store the pth row of $\boldsymbol{B}$ ($p \in P_i$). The allocation of rows P_i ($i = 1, 2, \ldots, u$) to be processed by each CPU should be done so as to balance the loads as evenly as possible and reduce the amount of data transferred. In SDPARA the allocation is performed sequentially from the top down, mainly for practical reasons. In other words,

$$P_i = \{x \in \{1, 2, \cdots, m\} \mid x \equiv i \pmod{u}\} \qquad (i = 1, 2, \ldots, u). \tag{4}$$

In general, since m (the size of $\boldsymbol{B}$) is much larger than u (the number of CPUs), reasonably good load balancing can still be achieved with this simple method [31].

Next, we discuss how the Cholesky factorization of the coefficient matrix $\boldsymbol{B}$ of the Schur complement equation system $\boldsymbol{B}dz = \boldsymbol{s}$ can be performed in parallel. In SDPARA, this is done by calling the parallel Cholesky factorization routine in the ScaLAPACK parallel numerical computation library [5]. In the Cholesky factorization of the matrix, the load distribution and the quantity of data transferred are greatly affected by the way in which the matrix is partitioned. In ScaLA-PACK's parallel Cholesky factorization, the coefficient matrix $\boldsymbol{B}$ is partitioned by block-cyclic partitioning and each block is allocated to and stored in each CPU.

We now present the results of numerical experiments in which the ordinary characteristics of SDPARA are well expressed. Table 2 shows the numerical results obtained when two types of SDPs were solved with SDPA6.0 and SDPARA. With SDPARA the parallel processing was performed using 64 CPUs – the computation times and memory requirements are shown for one

of these CPUs. In SDPARA, the time taken up by transmitting data in block-cyclic partitioned format for the parallel Cholesky factorization of $B \in \mathcal{S}_{++}^m$ performed one row at a time on each CPU is included in the term "Computation of B". Problem C is an SDP relaxation of maximum clique problems which is described in section 4. The size of problem C is $n = 300, m = 4375$, while problem B is the same as the problem used in the numerical experiment of Table 1, with $n = 1000$ and $m = 1891$.

Table 2: Numerical comparison between SDPA and SDPARA

	Problem C		Problem B	
	SDPA	SDPARA	SDPA	SDPARA
Computation of B	126.6s	6.5s	82.0s	7.7s
Cholesky factorization of B	253.0s	15.1s	25.3s	2.9s
Computation of $\widetilde{dX}$	2.0s	2.0s	69.4s	69.0s
Dense matrix computation	5.0s	4.9s	125.7s	126.1s
Total computation time	395s	36s	308s	221s
Storage of B	146MB	5MB	27MB	1MB
Storage of $n \times n$ matrix	21MB	21MB	237MB	237MB
Total memory usage	179MB	58MB	279MB	265MB

As Table 2 shows, the time taken for the computation of B and the Cholesky factorization of B is greatly reduced. (The number of iterations in the primal-dial interior-point method is the same for each problem in SDPA and SDPARA.) Also, since $B \in \mathcal{S}_{++}^m$ is stored by partitioning it between the CPUs, the amount of memory needed to store B at each CPU is greatly reduced. On the other hand, there is no change in the computation times associated with the $n \times n$ dense matrix computation such as the computation of $\widetilde{dX}$. Also, the amount of memory needed to store the $n \times n$ matrix on each CPU is the same as in SDPA. In SDPARA, MPI is used for communication between the CPUs. Since additional time and memory are consumed when MPI is started up, it causes the overall computation times and memory requirements to increase slightly. In an SDP where m is large and n is small, as in problem C, it becomes possible to process the parts relating to the Schur complement equation system $Bdz = s$ – which would otherwise require a large amount of time and memory – very efficiently by employing parallel processing, and great reductions can also be made to the overall computation times and memory requirements. On the other hand, in a problem where n is large, as in problem B, large amounts of time and memory are consumed in the parts where the $n \times n$ dense matrix is handled. Since these parts are not performed in parallel, it is not possible to greatly reduce the overall computation times or memory requirements. Generally speaking, SDPARA can efficiently solve SDPs with large m where time is taken up by the Cholesky factorization of $B \in \mathcal{S}_{++}^m$ and dense SDPs where a large amount of time is needed to compute each element of B. But in an SDP with large n, the large amounts of computation time and memory taken up in handling the $n \times n$ dense matrix make it difficult for SDPARA to reach a solution in the same way as SDPA.

3 The parallel computation of primal-dual interior-point methods using matrix completion

In this section we propose the SDPARA-C software package, which is based on SDPA-C as described in section 2.2 and which solves SDP problems by executing the primal-dual interior-point

method on parallel CPUs (PC clusters). The parallel computation of $\boldsymbol{B} \in \mathcal{S}_{++}^m$ is discussed in section 3.1, and the parallel computation of $\widetilde{d\boldsymbol{X}}$ is discussed in section 3.2. After that, section 3.3 summarizes the characteristics of SDPARA-C.

3.1 Computation of the coefficient matrix

In the paper [22], three methods are proposed for computing the coefficient matrix $\boldsymbol{B} \in \mathcal{S}_{++}^m$ of the Schur complement equation system $\boldsymbol{B}d\boldsymbol{z} = \boldsymbol{s}$. We chose to develop our improved parallel method based on the method discussed in section 5.2 of the paper [22]. Each element of $\boldsymbol{B} \in \mathcal{S}_{++}^m$ and $\boldsymbol{s} \in \mathbb{R}^m$ computed in the following way:

$$
\left.
\begin{aligned}
B_{pq} &:= \sum_{k=1}^n (\boldsymbol{M}^{-T}\boldsymbol{M}^{-1}\boldsymbol{e}_k)^T \boldsymbol{A}_q (\boldsymbol{N}^{-T}\boldsymbol{N}^{-1}U_k[\boldsymbol{A}_p]) \\
&+ \sum_{k=1}^n (\boldsymbol{N}^{-T}\boldsymbol{N}^{-1}\boldsymbol{e}_k)^T \boldsymbol{A}_q (\boldsymbol{M}^{-T}\boldsymbol{M}^{-1}U_k[\boldsymbol{A}_p]) \quad (p, q = 1, 2, \ldots, m), \\
s_p &:= b_p + \sum_{k=1}^n (\boldsymbol{M}^{-T}\boldsymbol{M}^{-1}\boldsymbol{e}_k)^T \boldsymbol{R} (\boldsymbol{N}^{-T}\boldsymbol{N}^{-1}U_k[\boldsymbol{A}_p]) \\
&+ \sum_{k=1}^n (\boldsymbol{N}^{-T}\boldsymbol{N}^{-1}\boldsymbol{e}_k)^T \boldsymbol{R} (\boldsymbol{M}^{-T}\boldsymbol{M}^{-1}U_k[\boldsymbol{A}_p]) \\
&- \sum_{k=1}^n 2\mu \boldsymbol{e}_k^T \boldsymbol{N}^{-T}\boldsymbol{N}^{-1}U_k[\boldsymbol{A}_p] \qquad (p = 1, 2, \ldots, m).
\end{aligned}
\right\} \tag{5}
$$

The diagonal part of matrix $\boldsymbol{A}_p$ is denoted by $\boldsymbol{T}$, and the upper triangular part excluding the diagonal is denoted by $\boldsymbol{U}$. In equation (5), $U_k[\boldsymbol{A}_p]$ represents the kth column of the upper triangular matrix $\frac{1}{2}\boldsymbol{T} + \boldsymbol{U}$, and $\boldsymbol{e}_k \in \mathbb{R}^n$ denotes a vector in which the kth element is 1 and all the other elements are 0. This algorithm differs from the method proposed in section 5.2 of the paper [22] in two respects. One is that the computations are performed using only the upper triangular matrix part of matrix $\boldsymbol{A}_p$. When $U_k[\boldsymbol{A}_p]$ is a zero vector, there is no need to perform computations involving its index k. Comparing the conventional method and the method of equation (5), it is impossible to generalize which has fewer indices k to be computed. The other difference is that whereas $\widehat{\boldsymbol{X}}$ is subjected to a clique sparse factorization $\boldsymbol{M}^{-T}\boldsymbol{D}\boldsymbol{M}^{-1}$ in section 5.2 of the paper [22], we implemented the sparse factorization $\boldsymbol{M}^{-T}\boldsymbol{M}^{-1}$. Because the computation costs are almost same and the sparse factorization $\boldsymbol{M}^{-T}\boldsymbol{M}^{-1}$ is implemented more easily than the clique sparse factorization $\boldsymbol{M}^{-T}\boldsymbol{D}\boldsymbol{M}^{-1}$.

Here, equation (5) can be used to compute $\boldsymbol{B} \in \mathcal{S}_{++}^m$ independently for each row in the same way as in the discussion of SDPARA in section 2.3. Therefore, in SDPARA-C it is possible to allocate the computation of $\boldsymbol{B} \in \mathcal{S}_{++}^m$ to each CPU one row at a time. This allocation is performed in the same way as in equation (4) of section 2.3. However, when using equation (5) to perform the computations, the number of nonzero column vectors in $U_k[\boldsymbol{A}_p]$ strongly affects the computational efficiency. Normally, since the number of nonzero vectors in $U_k[\boldsymbol{A}_p]$ is not fixed, it is possible that large variations may occur in the computation time on each CPU when the allocation to each CPU is performed as described above. For example, when there is just one identity matrix in the constraint matrices, this matrix may be regarded as a sparse matrix but equation (5) still has to be computed for every value of index k from 1 to n. To avoid such circumstances, it is better to process rows that are likely to involve lengthy computation and disrupt the load balance in parallel not by a single CPU but by multiple CPUs. The set of indices of such rows is defined as $Q \subset \{1, 2, \ldots, m\}$. The allocation K_i^p is set for every $p \in Q$ and every $i = 1, 2, \cdots, u$ as follows:

$$
\cup_{i=1}^u K_i^p = \{k \in \{1, 2, \ldots, n\} \mid U_k[\boldsymbol{A}_p] \neq \boldsymbol{0}\}, \quad K_i^p \cap K_j^p = \emptyset \text{ if } i \neq j
$$

At the ith CPU, the following computation is only performed for indices k contained in K_i^p in equation (5), with the results stored in a working vector $\boldsymbol{w} \in \mathbb{R}^m$ and a scalar $g \in \mathbb{R}$.

$$
\begin{aligned}
w_q := &\sum_{k \in K_i^p} (\boldsymbol{M}^{-T}\boldsymbol{M}^{-1}\boldsymbol{e}_k)^T \boldsymbol{A}_q (\boldsymbol{N}^{-T}\boldsymbol{N}^{-1}U_k[\boldsymbol{A}_p]) \\
&+ \sum_{k \in K_i^p} (\boldsymbol{N}^{-T}\boldsymbol{N}^{-1}\boldsymbol{e}_k)^T \boldsymbol{A}_q (\boldsymbol{M}^{-T}\boldsymbol{M}^{-1}U_k[\boldsymbol{A}_p]) \qquad (q = 1, 2, \ldots, m), \\
g := &\sum_{k \in K_i^p} (\boldsymbol{M}^{-T}\boldsymbol{M}^{-1}\boldsymbol{e}_k)^T \boldsymbol{R} (\boldsymbol{N}^{-T}\boldsymbol{N}^{-1}U_k[\boldsymbol{A}_p]) \\
&+ \sum_{k \in K_i^p} (\boldsymbol{N}^{-T}\boldsymbol{N}^{-1}\boldsymbol{e}_k)^T \boldsymbol{R} (\boldsymbol{M}^{-T}\boldsymbol{M}^{-1}U_k[\boldsymbol{A}_p]) \\
&- \sum_{k \in K_i^p} 2\mu \boldsymbol{e}_k^T \boldsymbol{N}^{-T}\boldsymbol{N}^{-1}U_k[\boldsymbol{A}_p].
\end{aligned}
$$

After that, the values of $\boldsymbol{w}$ and g computed at each CPU are sent to the jth CPU that was originally scheduled to process the vector of the pth row of $\boldsymbol{B} \in \mathcal{S}_{++}^m$ ($p \in P_j$), where they are added together. Therefore, the pth row of $\boldsymbol{B} \in \mathcal{S}_{++}^m$ is stored at the jth CPU ($p \in P_j$). Thus, the algorithm for computing the coefficient matrix $\boldsymbol{B} \in \mathcal{S}_{++}^m$ of the Schur complement equation system and the right hand side vector $\boldsymbol{s} \in \mathbb{R}^m$ at the ith CPU is as follows:

Computation of $\boldsymbol{B}$ and $\boldsymbol{s}$ at the ith CPU

> Set $\boldsymbol{B} := \boldsymbol{O}$, $\boldsymbol{s} := \boldsymbol{b}$
> for $\quad p \in Q$
> $\quad$ Set $\boldsymbol{w} = \boldsymbol{0}$ and $g = 0$
> $\quad$ for $\quad k \in K_i^p$
> $\quad\quad$ Compute $\boldsymbol{v}_1 := \boldsymbol{M}^{-T}\boldsymbol{M}^{-1}\boldsymbol{e}_k$ and $\boldsymbol{v}_2 := \boldsymbol{N}^{-T}\boldsymbol{N}^{-1}U_k[\boldsymbol{A}_p]$
> $\quad\quad$ Compute $\boldsymbol{v}_3 := \boldsymbol{N}^{-T}\boldsymbol{N}^{-1}\boldsymbol{e}_k$ and $\boldsymbol{v}_4 := \boldsymbol{M}^{-T}\boldsymbol{M}^{-1}U_k[\boldsymbol{A}_p]$
> $\quad\quad$ for $\quad q = 1, 2, \ldots, m$
> $\quad\quad\quad$ Compute $\boldsymbol{v}_1^T \boldsymbol{A}_q \boldsymbol{v}_2 + \boldsymbol{v}_3^T \boldsymbol{A}_q \boldsymbol{v}_4$ and add to w_q
> $\quad\quad$ end(for)
> $\quad\quad$ Compute $\boldsymbol{v}_1^T \boldsymbol{R} \boldsymbol{v}_2 + \boldsymbol{v}_3^T \boldsymbol{R} \boldsymbol{v}_4 - 2\mu \boldsymbol{e}_k^T \boldsymbol{v}_2$ and add to g
> $\quad$ end(for)
> $\quad$ Send $\boldsymbol{w}$ and g to the jth CPU ($p \in P_j$) and add them together
> end(for)
> for $\quad p \in P_i - Q$
> $\quad$ for $\quad k = 1, 2, \ldots, n \quad$ where $U_k[\boldsymbol{A}_p] \neq \boldsymbol{0}$
> $\quad\quad$ Compute $\boldsymbol{v}_1 := \boldsymbol{M}^{-T}\boldsymbol{M}^{-1}\boldsymbol{e}_k$ and $\boldsymbol{v}_2 := \boldsymbol{N}^{-T}\boldsymbol{N}^{-1}U_k[\boldsymbol{A}_p]$
> $\quad\quad$ Compute $\boldsymbol{v}_3 := \boldsymbol{N}^{-T}\boldsymbol{N}^{-1}\boldsymbol{e}_k$ and $\boldsymbol{v}_4 := \boldsymbol{M}^{-T}\boldsymbol{M}^{-1}U_k[\boldsymbol{A}_p]$
> $\quad\quad$ for $\quad q = 1, 2, \ldots, m$
> $\quad\quad\quad$ Compute $\boldsymbol{v}_1^T \boldsymbol{A}_q \boldsymbol{v}_2 + \boldsymbol{v}_3^T \boldsymbol{A}_q \boldsymbol{v}_4$ and add to B_{pq}
> $\quad\quad$ end(for)
> $\quad\quad$ Compute $\boldsymbol{v}_1^T \boldsymbol{R} \boldsymbol{v}_2 + \boldsymbol{v}_3^T \boldsymbol{R} \boldsymbol{v}_4 - 2\mu \boldsymbol{e}_k^T \boldsymbol{v}_2$ and add to s_p
> $\quad$ end(for)
> end(for)

As a result of this computation, the Schur complement equation system $\boldsymbol{B} \in \mathcal{S}_{++}^m$ is partitioned and stored row by row on each CPU. After that, $\boldsymbol{B} \in \mathcal{S}_{++}^m$ is sent to each CPU in block-cyclic partitioned form, and the Cholesky factorization is then performed in parallel.

3.2 Computation of the search direction

Here we discuss the parallel computation of the matrix variable $\widetilde{d\boldsymbol{X}}$ at step 1d of Algorithm 2.1. In the paper [22] it was proposed that $\widetilde{d\boldsymbol{X}}$ should be computed in separate columns as follows:

$$[\widetilde{d\boldsymbol{X}}]_{*k} := \mu \boldsymbol{N}^{-T}\boldsymbol{N}^{-1}\boldsymbol{e}_k - [\boldsymbol{X}]_{*k} - \boldsymbol{M}^{-T}\boldsymbol{M}^{-1}d\boldsymbol{Y}\boldsymbol{N}^{-T}\boldsymbol{N}^{-1}\boldsymbol{e}_k. \tag{6}$$

Here $[\boldsymbol{A}]_{*k}$ denotes the kth column of $\boldsymbol{A}$. It has been shown that if a nonlinear search is performed here instead of the linear search performed at step 3 of Algorithm 2.1, then it is only necessary to store the values of the extended sparsity pattern parts of $\widetilde{d\boldsymbol{X}}$. Since $[\widetilde{d\boldsymbol{X}}]_{*k}$ and $[\widetilde{d\boldsymbol{X}}]_{*k'}$ $(k \neq k')$ can be computed independently using the formula (6), it should be possible to split the matrix into columns and process these columns in parallel on multiple CPUs. The computational load associated with computing a single column is more or less constant. Therefore, by uniformly distributing columns to multiple CPUs, the computation can be performed in parallel with a uniformly balanced load.

The computation of columns of $\widetilde{d\boldsymbol{X}}$ at the ith CPU is performed as shown below. Here, the CPU allocations K_i $(i = 1, 2, \cdots, u)$ are determined as follows:

$$K_i = \{x \in \{1, 2, \cdots, n\} \mid (i-1)\left[\frac{n}{u}\right] < x \leq i[\frac{n}{u}]\} \quad (i = 1, 2, \ldots, u-1),$$
$$K_u = \{x \in \{1, 2, \cdots, n\} \mid (u-1)\left[\frac{n}{u}\right] < x\}.$$

and the ith CPU computes the pth column of $\widetilde{d\boldsymbol{X}}$ $(p \in K_i)$.

Computation of $\widetilde{d\boldsymbol{X}}$ at the ith CPU

Set $\widetilde{d\boldsymbol{X}} := -\boldsymbol{X}$
for $k \in K_i$
    Compute $\boldsymbol{v} := \mu \boldsymbol{N}^{-T}\boldsymbol{N}^{-1}\boldsymbol{e}_k - \boldsymbol{M}^{-T}\boldsymbol{M}^{-1}d\boldsymbol{Z}\boldsymbol{N}^{-T}\boldsymbol{N}^{-1}\boldsymbol{e}_k$
    Add $\boldsymbol{v}$ to $\widetilde{d\boldsymbol{X}}_{*k}$
end(for)
Transmit the column vector computed at each CPU to all the other CPUs

---

At the end of this algorithm, the column vector computed at each CPU is sent to all the other CPUs. At this time, since only the extended sparsity pattern parts of $\widetilde{d\boldsymbol{X}}$ are needed, these are the only parts that have to be sent to the other CPUs. Therefore, this part of the data transfer can be performed at high speed.

3.3 Characteristics

Table 3 shows the computation time of each part and the memory requirements when problem B used in the numerical experiments of Tables 1 and 2 ($n = 1000, m = 1891$) is solved with SDPARA-C. In this experiment, the parallel processing was performed using 64 CPUs. To facilitate comparison, the results obtained with SDPA 6.0, SDPA-C and SDPARA as shown in Tables 1 and 2 are also reproduced here. In the results obtained with SDPARA and SDPARA-C, the time taken up by transmitting data in block-cyclic partitioned format for the parallel Cholesky factorization of the coefficient matrix $\boldsymbol{B} \in \mathcal{S}_{++}^m$ of the Schur complement equation system performed one row at a time on each CPU is included in the term "Computation of $\boldsymbol{B}$". Also, the time taken to send the results of computing $\widetilde{d\boldsymbol{X}}$ obtained at each CPU to all the other CPUs is included in the term "Computation of $\widetilde{d\boldsymbol{X}}$".

Table 3: Numerical comparison between SDPA, SDPA-C, SDPARA and SDPARA-C

	Problem B			
	SDPA	SDPA-C	SDPARA	SDPARA-C
Computation of $\boldsymbol{B}$	82.0s	662.8s	7.7s	10.5s
Cholesky factorization of $\boldsymbol{B}$	25.3s	34.1s	2.9s	4.0s
Computation of $\widetilde{d\boldsymbol{X}}$	69.4s	32.6s	69.0s	2.4s
Dense matrix computation	125.7s	2.6s	126.1s	2.3s
Total computation time	308s	733s	221s	26s
Storage of $\boldsymbol{B}$	27MB	27MB	1MB	1MB
Storage of $n \times n$ matrix	237MB	8MB	237MB	8MB
Total memory usage	279MB	39MB	265MB	41MB

The number of iterations of the primal-dual interior-point method was 20 when the problem was solved by SDPA and SDPARA, and 27 when solved by SDPA-C and SDPARA-C. The reason why more iterations were required by the primal-dual interior-point methods using matrix completion is that these are simple path-following primal-dual interior-point methods. Section 2.1 mentions the four parts that take up most of the computation time and the two parts that take up most of the memory requirements when an SDP is solved by a primal-dual interior point method. In Table 3, we are able to confirm that the problem is processed efficiently in all these parts by SDPARA-C. SDPARA-C uses MPI for communication between CPUs in the same way as SDPARA. Since additional time and memory are consumed when MPI is started up and executed, it causes the overall computation times and memory requirements to increase. This is why the overall amount of memory used by SDPARA-C is greater than that of SDPA-C. As Table 3 shows, the computation times involved in the "Computation of $\boldsymbol{B}$" of SDPARA-C does not necessarily improve when it is compared to that of SDPARA. The next section shows the results of numerical experiments in which a variety of problems were solved by SDPARA-C.

4   Numerical experiments

This section presents the results of numerical experiments. In section 4.1, 4.2 and 4.3, the numerical experiments performed on the Presto III PC cluster at the Matsuoka Laboratory in the Tokyo Institute of Technology. Each node of this cluster consists of an Athlon 1900+ (1.6 GHz) CPU with 768 MB of memory. The nodes are connected together by a Myrinet 2000 network, which is faster and performs better than Gigabit Ethernet. In section 4.4 and 4.5, the numerical experiments performed on the SDPA PC cluster at the Fujisawa Laboratory in the Tokyo Denki University. Each node of this cluster consists of a Dual Athlon MP 2400+ (2GHz) CPU with 1GB of memory. The nodes are connected together by 1000BASE-T.

In these experiments we evaluated three types of software. One was the SDPARA-C software proposed in this paper. The second was SDPARA [31]. The third was PDSDP ver.4.6, as used in the numerical experiments of the paper [1], which employs the dual interior-point method on parallel CPUs. SDPARA-C and SDPARA used ScaLAPACK ver.1.7 and ATLAS ver.3.4.1. And the block size used with ScaLAPACK is 40 and other options are default settings. In all the experiments apart from those described in section 4.5, the starting point of the interior-point method was chosen to be $\boldsymbol{X} = \boldsymbol{Y} = 100\boldsymbol{I}, \boldsymbol{z} = \boldsymbol{0}$. As the termination conditions, the algorithms were run until the relative dual gap in SDPARA-C and SDPARA was 10^{-7} or less.

The SDPs used in the experiments were:

- SDP relaxations of randomly generated maximum cut problems.

- SDP relaxations of randomly generated maximum clique problems.

- Randomly generated norm minimization problems.

- Some benchmark problems from SDPLIB [7].

- Some benchmark problems from the 7th DIMACS implementation challenge.

The SDP relaxations of maximum cut problems and norm minimization problems were the same as the problems used in the experiments of the paper [22].

SDP relaxations of maximum cut problems: Assume $G = (V, E)$ is an undirected graph, where $V = \{1, 2, \ldots, n\}$ is a set of vertices and $E \subset \{(i, j) : i, j \in V, \ i < j\}$ is a set of edges. Also, assume that weights $C_{ij} = C_{ji}$ $((i, j) \in E)$ are also given. An SDP relaxation of a maximum cut problem is then given by

$$
\begin{aligned}
\text{minimize} \quad & -\boldsymbol{A}_0 \bullet \boldsymbol{X} \\
\text{subject to} \quad & \boldsymbol{E}_{ii} \bullet \boldsymbol{X} = 1/4 \ (i = 1, 2, \ldots, n), \\
& \boldsymbol{X} \in \mathcal{S}_+^n,
\end{aligned}
\tag{7}
$$

where $\boldsymbol{E}_{ii}$ is an $n \times n$ matrix where the element at (i, i) is 1 and all the other elements are 0. Also, we define $\boldsymbol{A}_0 = \operatorname{diag}(\boldsymbol{Ce}) - \boldsymbol{C}$.

SDP relaxations of maximum clique problems: Assume $G = (V, E)$ is an undirected graph as in the maximum cut problem. An SDP relaxation of the maximum clique SDP problem is then given by

$$
\begin{aligned}
\text{minimize} \quad & -\boldsymbol{E} \bullet \boldsymbol{X} \\
\text{subject to} \quad & \boldsymbol{E}_{ij} \bullet \boldsymbol{X} = 0 \ ((i, j) \notin E), \\
& \boldsymbol{I} \bullet \boldsymbol{X} = 1, \ \boldsymbol{X} \in \mathcal{S}_+^n,
\end{aligned}
\tag{8}
$$

where $\boldsymbol{E}$ is an $n \times n$ matrix where all the elements are 1, and $\boldsymbol{E}_{ij}$ is an $n \times n$ matrix where the (i, j)th and (j, i)th elements are 1 and all the other elements are 0. Since the aggregate sparsity pattern and extended sparsity pattern are dense patterns in this formulation, the problem is solved after performing the transformation proposed in section 6 of the paper [12].

Norm minimization problem: Assume $\boldsymbol{F}_i \in \mathbb{R}^{q \times r}$ $(i = 1, 2, \ldots, p)$. The norm minimization problem is then defined as follows:

$$
\text{minimize} \ \left\| \boldsymbol{F}_0 + \sum_{i=1}^p \boldsymbol{F}_i y_i \right\| \ \text{subject to} \ y_i \in \mathbb{R} \ (i = 1, 2, \ldots, p).
$$

where $\|\boldsymbol{G}\|$ is the spectral norm of $\boldsymbol{G}$; i.e., the square root of the maximum eigenvector of $\boldsymbol{G}^T \boldsymbol{G}$. This problem can be transformed into the following SDP:

$$
\begin{aligned}
\text{maximize} \quad & -z_{p+1} \\
\text{subject to} \quad & \sum_{i=1}^p \begin{pmatrix} \boldsymbol{O} & \boldsymbol{F}_i^T \\ \boldsymbol{F}_i & \boldsymbol{O} \end{pmatrix} z_i + \begin{pmatrix} \boldsymbol{I} & \boldsymbol{O} \\ \boldsymbol{O} & \boldsymbol{I} \end{pmatrix} z_{p+1} + \begin{pmatrix} \boldsymbol{O} & \boldsymbol{F}_0^T \\ \boldsymbol{F}_0 & \boldsymbol{O} \end{pmatrix} = \boldsymbol{Y}, \\
& \boldsymbol{Y} \in \mathcal{S}_+^{q+r}.
\end{aligned}
\tag{9}
$$

In section 4.1 we verify the scalability of parallel computations in SDPARA-C, SDPARA and PDSDP. In section 4.2 we investigate how the sparsity of the extended sparsity pattern of an SDP affects the efficiency of the three software packages. In section 4.3 we investigate how large SDPs can be solved. And finally in sections 4.4 and 4.5 we select a number of SDPs from the SDPLIB [7] problems and DIMACS challenge problems and solve them using SDPARA-C.

4.1   Scalability

In this section we discuss the results of measuring the computation times required to solve a number of SDPs with clusters of 1, 2, 4, 8, 16, 32 and 64 PCs. The problems that were solved are listed in Table 4. In this table, n and m respectively represent the size of matrix variables $\boldsymbol{X}$ and $\boldsymbol{Y}$ and the number of linear constraints in the primal problem (1).
   3

Table 4: Problems solved in the experiments. See Figures 1, 2, 3 and Table 5

Problem	n	m	
Cut($10 * 100$)	1000	1000	10×100 lattice graph maximum cut (7)
Cut($10 * 500$)	5000	5000	10×500 lattice graph maximum cut (7)
Clique($10 * 100$)	1000	1891	10×100 lattice graph maximum clique (8)
Clique($10 * 200$)	2000	3791	10×200 lattice graph maximum clique (8)
Norm($10 * 990$)	1000	11	Norm minimization of ten 10×990 matrices (9)
Norm($5 * 995$)	1000	11	Norm minimization of ten 5×995 matrices (9)
qpG11	1600	800	SDPLIB[7] problem
maxG51	1000	1000	SDPLIB[7] problem
control10	150	1326	SDPLIB[7] problem
theta6	300	4375	SDPLIB[7] problem

In Figure 1, the horizontal axis shows the number of CPUs and the vertical axis shows the logarithmic computation time. From the numerical results it can be confirmed that parallel computation performed with SDPARA-C has a very high level of scalability. Although the scalability becomes worse as the number of CPUs increases and the computation time is in the region of a few tens of seconds, this is due to factors such as disruption of the load balance of each CPU and an increase in the proportion of time taken up by data transfers.

Next, we performed similar numerical experiments with SDPARA. The results of these experiments are shown in Figure 2. In SDPARA, the "Cut($10 * 500$)" and "Clique($10 * 200$)" problems could not be solved because there was insufficient memory. As mentioned in section 2.3, in SD-PARA the computation of the coefficient matrix $\boldsymbol{B} \in \mathcal{S}_{++}^m$ of the Schur complement equation system of size m and the Cholesky factorization thereof are performed in parallel. Therefore, the effect of parallel processing is very large for the "control10" and "theta6" problems where m is large compared to n. On the other hand, in the other problems where m is at most only twice as large as n, parallel processing had almost no effect whatsoever. Compared with the numerical results of SDPARA-C shown in Figure 1, SDPARA produced solutions faster than SDPARA-C for the two SDPs with small n ("control10" and "theta6"). In the "Clique($10 * 100$)" problem, the computation time of SDPARA was smaller with fewer CPUs, but with a large number of CPUs the computation time was lower in SDPARA-C. This is because parallel processing is applied to more parts in SDPARA-C, so it is more able to reap the benefits of parallel processing. For the other problems, SDPARA-C performed better.
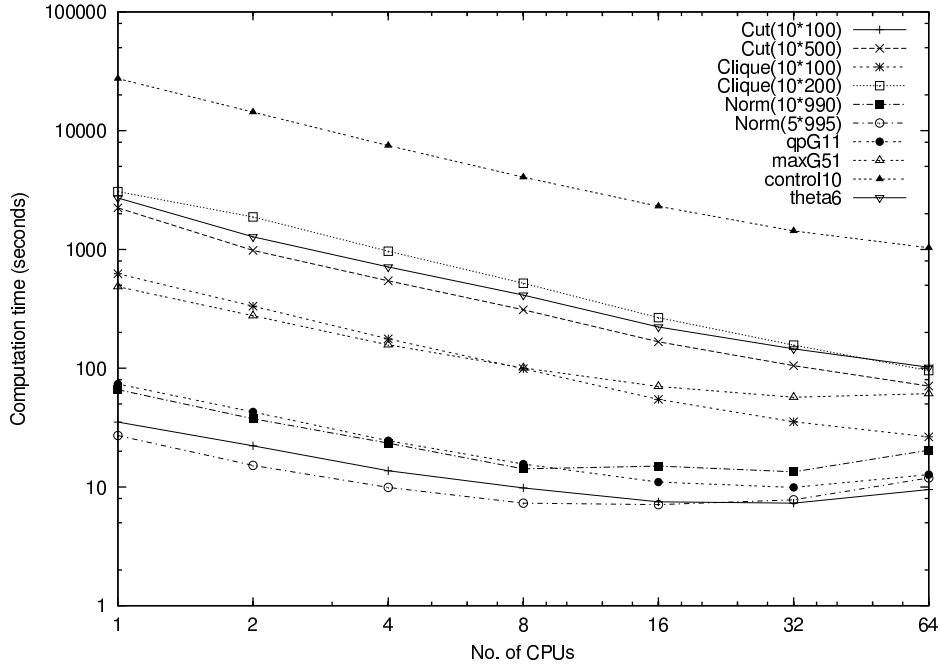
13

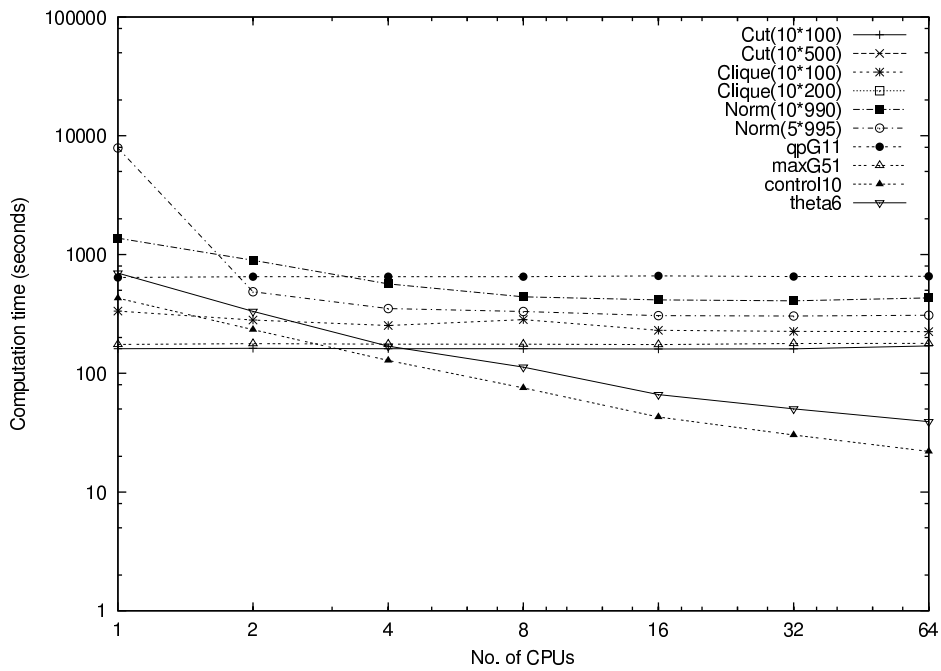Figure 1: Scalability of SDPARA-C applied to problems listed in Table 4

Figure 2: Scalability of SDPARA applied to problems listed in Table 4

Next, we performed similar numerical experiments with PDSDP [1]. The results of these experiments are shown in Figure 3. PDSDP is a software package that employs the dual interior-point method on parallel CPUs. However, according to the numerical results shown in Figure 3, it appears that this algorithm is not particularly scalable for some problems. Its performance

14

is similar to that of SDPARA for SDPs such as "control10" and "theta6" where m is large and parallel computation works effectively. We compare these numerical results with those obtained for SDPARA-C as shown in Figure 1. The dual interior-point method employed in PDSDP uses the sparsity of the SDP to perform the computation in the same way as the primal-dual interior-point method that uses matrix completion as employed by SDPARA-C. Consequently, when solved on a single CPU, the computation time of PDSDP is often found to be close to the results obtained for the computation time of SDPARA-C. However, when using 64 CPUs, the results show that SDPARA-C is better for 9 out of the 10 problems (i.e., all of them except "control10").
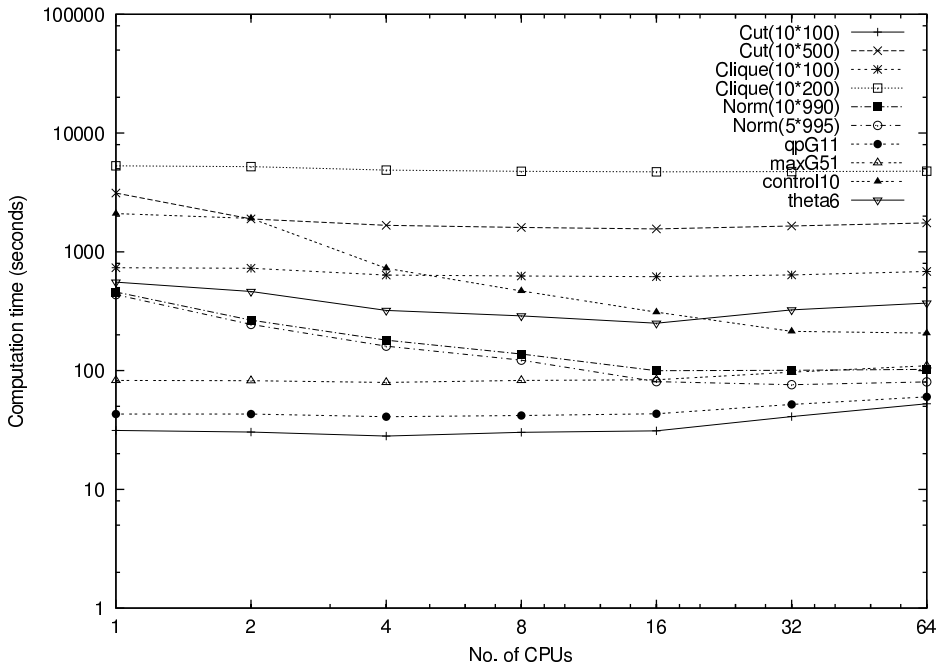

Figure 3: Scalability of PDSDP applied to problems listed in Table 4

Based on these results, we have been able to verify that SDPARA-C has better scalability than SDPARA and PDSDP.

In addition to Figures 1, 2 and 3, Table 5 shows the parallel efficiency of SDPARA-C, SDPARA and PDSDP applied to each problem between the cluster of 1 and 8 CPUs and the shortest computation time of SDPARA-C, SDPARA and PDSDP applied to each problem over the cluster of 1, 2, 4, 8, 16, 32 and 64 CPUs. The parallel efficiency is defined by

$$\frac{\text{the cpu time in a single machine}}{\text{time on 8cpus} \times 8}.$$

The column "parallel eff." denotes the parallel efficiency and the column "#cpus" denotes the number of CPUs at which the shortest computation time was attained, and "M" signifies that the problem could not be solved due to insufficient memory.

4.2   The sparsity of SDPs

In this section we examine how SDPARA-C is affected by the sparsity of a given SDP. All the problems were solved by 64 CPUs. We produced SDP relaxations of maximum cut problems (7) in which the sparsity ρ of the extended sparsity pattern was varied while the size of the SDP was

15

Table 5: Parallel efficiency and shortest computation times required for SDPARA-C, SDPARA and PDSDP to solve problems listed in Table 4

	SDPARA-C			SDPARA			PDSDP		
Problem	parallel eff. (%)	# cpus	time (s)	parallel eff. (%)	# cpus	time (s)	parallel eff. (%)	# cpus	time (s)
Cut(10 * 100)	44.9	32	7.3	12.5	16	160.1	13.0	4	28.1
Cut(10 * 500)	90.1	64	70.7			M	24.4	16	1559.6
Clique(10 * 100)	78.8	64	26.4	14.7	64	224.3	14.7	16	618.6
Clique(10 * 200)	73.9	64	95.6			M	13.9	16	4718.0
Norm(10 * 990)	58.1	32	13.4	39.0	32	407.2	41.6	16	99.8
Norm(5 * 995)	46.4	16	7.1	29.6	32	303.3	44.5	32	76.1
qpG11	58.9	32	9.9	12.3	8	650.4	12.9	4	40.9
maxG51	60.4	32	56.9	12.4	16	174.6	12.5	4	79.4
control10	84.5	64	1034.6	71.2	64	22.0	56.1	64	207.0
theta6	82.3	64	101.9	77.3	64	39.1	24.0	16	250.5

fixed at $n = m = 1000$. Figure 4 shows the relationship between the sparsity of the SDP and the computation time when these problems were solved by SDPARA-C, SDPARA and PDSDP, and Figure 5 shows the relationship between the sparsity of the SDP and the amount of memory needed to solve it. In Figures. 4 and 5, the horizontal axis is a logarithmic representation of the sparsity $\rho(1 \leq \rho \leq 1000)$, and the vertical axis is a logarithmic representation of the computation time (or memory usage).
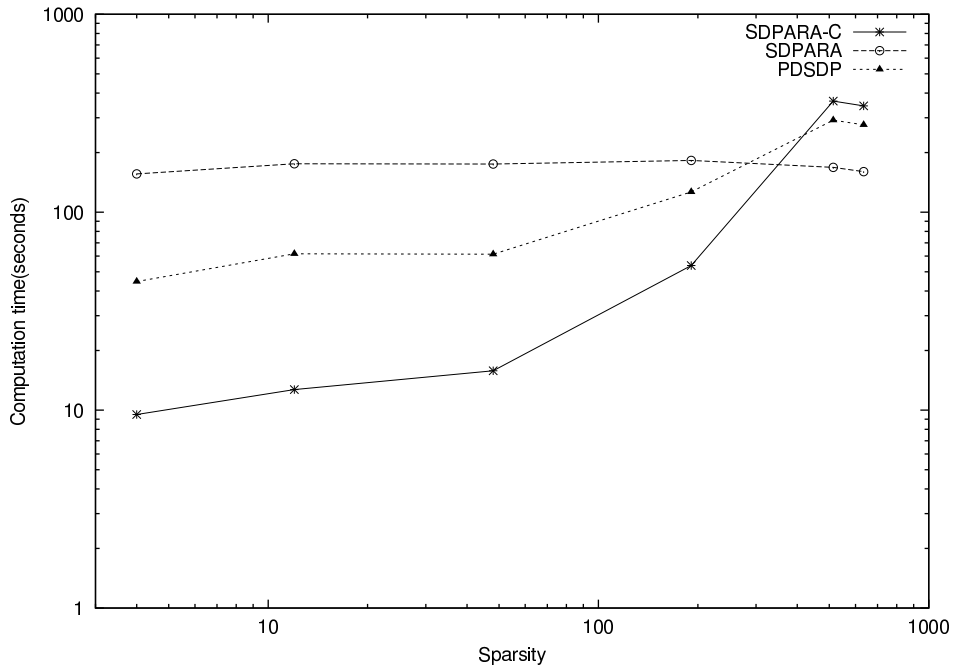

Figure 4: Sparsity and computation time required for SDPARA-C, SDPARA and PDSDP to solve SDP relaxations of maximum cut problems (64 CPUs)
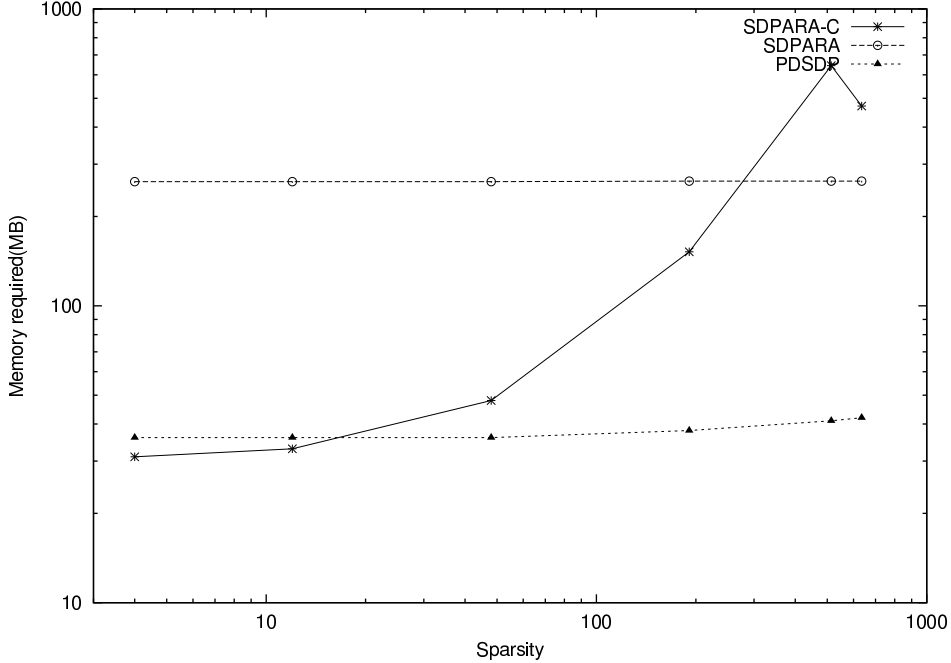
16

Figure 5: Sparsity and memory requirements for SDP relaxations of maximum cut problems by SDPARA-C, SDPARA and PDSDP (64 CPUs)

The results in Figure 4 show that as ρ increases, the computation time of SDPARA-C and PDSDP also increases. On the other hand, since SDPARA is implemented without much consideration of sparsity of SDPs, the computation time is almost completely independent of the sparsity of an SDP to be solved. The results in Figure 5 show that the memory requirements per processor of SDPARA-C depend on the sparsity of the SDP, whereas the memory requirements per processor of SDPARA and PDSDP remain more or less fixed.

By comparing the results obtained with these three software packages, it can be seen that SDPARA-C has the shortest computation times and smallest memory requirements when the SDP to be solved is sparse. On the other hand, when the SDP to be solved has little sparsity, the shortest computation times were achieved with SDPARA and the smallest memory requirements were obtained with PDSDP. The reason why SDPARA-C requires more memory than SDPARA when the SDP has less sparsity is because it makes duplicate copies of variables to increase the computational efficiency. PDSDP is characterized in that its memory requirements are very low regardless of how sparse the SDP to be solved is.

4.3 Large-size SDPs

In this section we investigate how large an SDP's matrix variable can be made before it becomes impossible for SDPARA-C to solve the problem. In the following numerical experiments, parallel computation is performed using all 64 CPUs. The SDPs used in the numerical experiments were the SDP relaxation of a maximum cut lattice graph problem (7), the SDP relaxation of a maximum clique lattice graph problem (8), and a norm minimization problem (9). The size of the SDP relaxation of the maximum cut problem from a $k_1 \times k_2$ lattice is $n = m = k_1 k_2$. As an indicator of the sparsity of an SDP, we consider the average number ρ of nonzero elements per row of a sparse matrix having nonzero elements in the extended sparsity pattern parts. In this problem,

$\rho \le 2\min(k_1, k_2) + 1$. Therefore, by fixing k_1 at 10 and varying k_2 from 1000 to 4000, it is possible to produce SDPs with larger matrix variables (i.e., larger n) without increasing the sparsity ρ. Similarly, the size and sparsity of SDP relaxation of maximum clique problems with a $k_1 \times k_2$ lattice are respectively: $n = k_1 k_2$, $m = 2k_1 k_2 - k_1 - k_2 + 1$, $\rho \le 2\min(k_1, k_2) + 2$. Therefore, by fixing k_1 at 10 and varying k_2 from 500 to 2000, it is possible to produce SDPs with larger matrix variables without increasing the sparsity. Also, the size of a norm minimization problem derived from a matrix of size $k_1 \times k_2$ is $n = k_1 + k_2$. When the number of matrices is fixed at 10, $m = 11$ and $\rho \approx 2\min(k_1, k_2) + 1$. Therefore, by fixing k_1 at 10 and varying k_2 from 9990 to 39990, it is possible to produce SDPs with larger matrix variables without increasing the sparsity; thus $\rho \approx 21$ for all cases in Table 6.

Table 6: Numerical results on SDPARA-C applied to large-size SDPs (64 CPUs)

Problem	n	m	time (s)	memory (MB)
Cut(10 * 1000)	10000	10000	274.3	126
Cut(10 * 2000)	20000	20000	1328.2	276
Cut(10 * 4000)	40000	40000	7462.0	720
Clique(10 * 500)	5000	9491	639.5	119
Clique(10 * 1000)	10000	18991	3033.2	259
Clique(10 * 2000)	20000	37991	15329.0	669
Norm(10 * 9990)	10000	11	409.5	164
Norm(10 * 19990)	20000	11	1800.9	304
Norm(10 * 39990)	40000	11	7706.0	583

Table 6 shows the computation times and memory requirements per processor needed when SDPARA-C is used to solve SDP relaxations of maximum cut problems for large lattice graphs, SDP relaxations of maximum clique problems for large lattice graphs and norm minimization problems for large matrices. As one would expect, the computation times and memory requirements increase as n gets larger. When n was 2000 or more, SDPARA was unable to solve any of these three types of problem due to a lack of memory. Also, PDSDP had insufficient memory to solve SDP relaxations of maximum cut problems with $n \ge 10000$ or SDP relaxations of maximum clique problems and norm minimization problems with $n \ge 5000$. However, SDPARA-C was able to solve SDP relaxations of maximum cut problems with $n = 40000$, SDP relaxations of maximum clique problems with $n = 20000$, and norm minimization problems with $n = 40000$. Thus in the case of very sparse SDPs as solved in this section, solving with SDPARA-C allows an optimal solution to be obtained to very large-scale SDPs.

4.4 The SDPLIB problems

We performed numerical experiments with a number of problems selected from SDPLIB [7], which is a set of standard benchmark SDP problems. All the problems were processed in parallel on 8, 16 and 32 CPUs. However, in the "equalG11", "theta6" and "equalG51" problems, the constraint matrices include a matrix in which all the elements are 1, so these problems were solved after making the transformation proposed in section 6 of the paper [12]. The values of n and m in Table 7 indicate the size of the SDP in the same way as in section 4.1. Also, ρ is an index representing the sparsity of the SDP as in section 4.3. Cases in which the problem could not be solved due to insufficient memory are indicated as "M", and cases in which no optimal solution

had been found after 200 iterations are indicated as "I".

Table 7: Numerical results on SDPARA-C, SDPARA and PDSDP applied to SDPLIB problems

		SDPARA-C			SDPARA			PDSDP		
Problem	$n, \quad m, \quad \rho$	# cpus	time (s)	mem (MB)	# cpus	time (s)	mem (MB)	# cpus	time (s)	mem (MB)
thetaG11	801,2401,23	32	95.6	13	32	184.8	159		I	
maxG32	2000,2000,32	16	115.5	26			M	8	227.6	58
equalG11	801,801,35	32	252.0	115	16	133.1	161	16	258.7	33
qpG51	2000,1000,67	32	189.8	83	8	1210.9	955	8	490.5	46
control11	165,1596,74	32	2902.5	28	32	124.0	14	32	475.7	20
maxG51	1000,1000,134	16	95.2	79	8	173.1	243	8	106.1	18
thetaG51	1001,6910,137	32	1870.1	113			M		I	
theta6	300,4375,271	32	347.3	29	32	188.3	36	8	495.8	37
equalG51	1001,1001,534	32	683.3	178	16	241.2	251	16	624.9	51

Table 7 shows the computation times and memory requirements per processor when the shortest computation times are achieved over clusters of 8, 16 and 32 CPUs. We have verified that SDPARA-C attains higher scalability compared to SDPARA and PDSDP. The SDPs shown in Table 7 are listed in order of increasing sparsity ρ. In cases where the SDP to be solved is sparse (i.e., the problems near the top of Table 7), SDPARA-C was able to reach solutions in the shortest computation times and with the smallest memory requirements. In cases where the SDP to be solved is not very sparse (i.e., the problems near the bottom of Table 7), the shortest computation times tended to be achieved with SDPARA and the smallest memory requirements tended to be achieved with PDSDP.

4.5 DIMACS

We performed numerical experiments with four problems selected from the 7th DIMACS challenge. All the problems were processed in parallel on 8, 16 and 32 CPUs. Table 8 shows the computation times and memory requirements per processor when the shortest computation times are achieved over clusters of 8, 16 and 32 CPUs. Since these problems tend to have optimal solutions with large values, we used the starting conditions $\boldsymbol{X} = \boldsymbol{Y} = 10^7 \boldsymbol{I}, \boldsymbol{z} = \boldsymbol{0}$. From the results of these numerical experiments as shown in Table 8, we were able to confirm that the smallest computation times were achieved with SDPARA-C, and that the smallest memory requirements were obtained with PDSDP, in cases where the SDP to be solved is large.

5 Conclusion

In this paper, by applying parallel processing techniques to the primal-dual interior-point method using matrix completion theory as proposed in the paper [12, 22], we have developed the SDPARA-C software package which runs on parallel CPUs (PC clusters). SDPARA-C is able to solve large-scale SDPs (SDPs with sparse large-scale data matrices and a large number of linear constraints in the primal problem (1)) while using less memory. In particular, we have reduced its memory requirements and computation time by performing computations based on matrix completion theory that exploit the sparsity of the problem without performing computations on dense matrices, and by making use of parallel Cholesky factorization. By introducing parallel processing in the

Table 8: Numerical results on SDPARA-C, SDPARA and PDSDP applied to DIMACS problems

Problem	$n, \quad m, \quad \rho$	SDPARA-C			SDPARA			PDSDP		
		# cpus	time (s)	mem (MB)	# cpus	time (s)	mem (MB)	# cpus	time (s)	mem (MB)
torusg3-8	512,512,78	8	26.4	26	8	46.5	66	8	18.0	7
toruspm3-8-50	512,512,78	8	27.5	26	8	53.4	66	8	18.9	7
torusg3-15	3375,3375,212	32	826.7	439			M	8	1945.4	159
toruspm3-15-50	3375,3375,212	32	866.3	439			M	8	1744.9	159

parts where bottlenecks have occurred in conventional software packages, we have also succeeded in making substantial reductions to the overall computation times.

We have conducted numerical experiments to compare the performance of SDPARA-C with that of SDPARA [31] and PDSDP [1] in a variety of different problems. From the results of these experiments, we have verified that SDPARA-C exhibits very high scalability. We have also found that SDPARA-C can solve problems in much less time and with much less memory than other software packages when the extended sparsity pattern is very sparse, and as a result we have successfully solved large-scale problems in which the size of the matrix variable is of the order of tens of thousands – an impossible task for conventional software packages.

As a topic for further study, it would be worthwhile allocating the parallel processing of the coefficient matrix of the Schur complement equation system proposed in section 3.1 so that the load balance of each CPU becomes more uniform (bearing in mind the need for an allocation that can be achieved with a small communication overhead). Numerical experiments have confirmed that that good results can be achieved even with the relatively simple method proposed in this paper, but there is probably still room for improvement in this respect.

When solving huge scale SDPs, memory is likely to become a bottleneck. In SDPARA-C, the coefficient matrix of the Schur complement equation system is partitioned between each CPU, but the other matrices (including the input data matrix and the sparse factorizations of the matrix variables in the primal and dual problems) are copied across to and stored on every CPU. This memory bottleneck could be eliminated by partitioning each of the matrices copied in this way and distributing the parts across the CPUs. However, since this would result in greater data communication and synchronization overheads, it would probably result in larger computation times. Therefore, further study is needed to evaluate the benefits and drawbacks of partitioning the data in this way.

SDPARA-C has the property of being able to solve sparse SDPs efficiently, and SDPARA has the property of being able to solve non-sparse SDPs efficiently. In other words, SDPARA-C and SDPARA have a complementary relationship. This fact has been confirmed from the numerical experiments of section 4.2 and section 4.4. The two algorithms used in SDPARA and SDPARA-C could be combined into a single convenient software package by automatically deciding which algorithm can solve a given SDP more efficiently, and then solving the problem with the selected algorithm. To make this decision, it would be necessary to estimate the computation times and memory requirements of each algorithm. However, estimating computation times in parallel processing is not that easy to do.

Acknowledgment

References

[1] S.J. Benson, Parallel Computing on Semidefinite Programs, Preprint ANL/MCS-P939-0302, http://www.msc.anl.gov/~benson/dsdp/pdsdp.ps (2002).

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

[3] A. Ben-Tal, M. Kocvara, A. Nemirovskii and J. Zowe, Free material optimization via semidefinite programming: the multiload case with contact conditions, *SIAM Journal on Optimization*, 9 (1999) 813-832.

[4] A. Ben-Tal and A. Nemirovskii, Lectures on Modern Convex Optimization Analysis, Algorithms, and Engineering Applications, *SIAM*, Philadelphia (2001).

[5] 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, *SIAM*, Philadelphia (1997).

[6] B. Borchers, CSDP 2.3 user's guide, *Optimization Methods and Software*, 11 & 12 (1999) 597–611. Available at http://www.nmt.edu/~borchers/csdp.html.

[7] B. Borchers, SDPLIB 1.2, a library of semidefinite programming test problems, *Optimization Methods and Software*, 11 & 12 (1999) 683-690.

[8] S. Burer, Semidefinite programming in the space of partial positive semidefinite matrices, *SIAM Journal on Optimization*, 14 (2003) 139-172.

[9] S. Burer, R.D.C. Monteiro, and Y. Zhang, A computational study of a gradient-based logbarrier algorithm for a class of large-scale SDPs, *Mathematical Programming*, 95 (2003) 359–379.

[10] C. Choi and Y. Ye, Solving sparse semidefinite programs using the dual scaling algorithm with an iterative solver, Manuscript, Department of Management Sciences, University of Iowa, Iowa City, IA 52242 (2000).

[11] K. Fujisawa, M. Kojima and K. Nakata, Exploiting sparsity in primal-dual interior-point methods for semidefinite programming, *Mathematical Programming*, 79 (1997) 235–253.

[12] M. Fukuda, M. Kojima, K. Murota and K. Nakata, Exploiting sparsity in semidefinite programming via matrix completion I: General framework, *SIAM Journal on Optimization*, 11 (2000) 647–674.

[13] D.R. Fulkerson and J.W.H. Gross, Incidence matrices and interval graphs, *Pacific Journal of Mathematics*, 15 (1965) 835–855.

[14] D. Goldfarb and G.Iyengar, Robust portfolio selection problems, *Mathematics of Operations Research*, 28 (2003) 1–38.

[15] W. Gropp, E. Lusk and A. Skjellum, Using MPI: Portable Parallel Programming with the Message-Passing Interface, *MIT Press*, Massachusetts (1994).

[16] C. Helmberg and F. Rendl, A spectral bundle method for semidefinite programming, *SIAM Journal on Optimization*, 10 (2000) 673–696.

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

[18] M. Kocvara and M. Stingl, PENNON - A Code for Convex Nonlinear and Semidefinite Programming, *Optimization Methods and Software*, 18 (2003) 317–333.

[19] M. Kojima, S. Shindoh and S. Hara, Interior-point methods for the monotone semidefinite linear complementarity problem in symmetric matrices, *SIAM Journal on Optimization*, 7 (1997) 86–125.

[20] R. D. C. Monteiro, Primal-dual path-following algorithms for semidefinite programming, *SIAM Journal on Optimization*, 7 (1997) 663–678.

[21] R. D. C. Monteiro, First- and second-order methods for semidefinite programming, *Mathematical Programming*, 97 (2003) 209–244.

[22] K. Nakata, K. Fujisawa, M. Fukuda, M. Kojima and K. Murota, Exploiting sparsity in semidefinite programming via matrix completion II: Implementation and numerical results, *Mathematical Programming*, 95 (2003) 303–327.

[23] K. Nakata, K. Fujisawa and M. Kojima, Using the conjugate gradient method in interior-point methods for semidefinite programs (in Japanese), *Proceedings of the Institute of Statistical Mathematics*, 46 (1998) 297–316.

[24] M. Nakata, H. Nakatsuji, M. Ehara, M. Fukuda, K. Nakata and K. Fujisawa, Variational calculations of fermion second-order reduced density matrices by semidefinite programming algorithm, *Journal of Chemical Physics*, 114 (2001) 8282-8292.

[25] J. F. Sturm, Using SeDuMi 1.02, a MATLAB toolbox for optimization over symmetric cones, *Optimization Methods and Software*, 11 & 12 (1999) 625–653.

[26] K. C. Toh and M. Kojima, Solving some large scale semidefinite programs via conjugate residual method, *SIAM Journal on Optimization*, 12 (2002) 669–691.

[27] K. C. Toh, M. J. Todd and R. H. Tütüncü, SDPT3 — a MATLAB software package for semidefinite programming, version 1.3, *Optimization Methods and Software*, 11 & 12 (1999) 545–581. Available at http://www.math.nus.edu.sg/~mattohkc.

[28] H. Waki, S. Kim, M. Kojima and M. Muramatsu, Sums of squares and semidefinite programming relaxations for polynomial optimization problems with structured sparsity, Research Report B-411, Department of Mathematical and Computing Sciences, Tokyo Institute of Technology, Meguro, Tokyo 152-8552 (2005).

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

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

[31] M. Yamashita, K. Fujisawa and M. Kojima, SDPARA: SemiDefinte Programming Algorithm PARAllel version, *Parallel Computing*, 29 (2003) 1053–1067.