論文 / 著書情報
Article / Book Information

題目(和文)	半正定値計画問題に対する主双対内点法の並列実装
Title(English)	Parallel Implementation of Primal-Dual Interior-Point Methods for SemiDefinite Programming
著者(和文)	山下真
Author(English)	Makoto Yamashita
出典(和文)	学位:博士（理学）, 学位授与機関:東京工業大学, 報告番号:甲第5675号, 授与年月日:2004年3月26日, 学位の種別:課程博士, 審査員:
Citation(English)	Degree:Doctor of Science, Conferring organization: Tokyo Institute of Technology, Report number:甲第5675号, Conferred date:2004/3/26, Degree Type:Course doctor, Examiner:
学位種別(和文)	博士論文
Type(English)	Doctoral Thesis

Parallel Implementation of Primal-Dual Interior-Point Methods for SemiDefinite Programming

Makoto YAMASHITA

Submitted in partial fulfillments of
the requirement for the degree of
DOCTOR OF SCIENCE

Department of Mathematical and Computing Sciences
Tokyo Institute of Technology

March 2004

Acknowledgments

I would like to express my sincere thanks to my supervisor Professor Masakazu Kojima. He gave me the opportunity to study in the field of Semidefinite Programming. A lot of his advices have encouraged me to do study. Many and many of his ideas are included in this thesis. Surely, he has been a 'research-aholic'. His attitude towards research has taught me many things. I am gratefully thank to him for his warmth and considerations in researching.

My special thanks are due to Professor Katsuki Fujisawa of Tokyo Denki University. The two software in this thesis, SDPARA and SDPARA-C, are based on his software SDPA. His support has been indispensable for the development of these software.

My special thanks are also due to Dr. Kazuhide Nakata of Tokyo Institute of Technology for discussions in particular on the joint work in Chapter 4. SDPARA-C relies on a result of his fundamental research in the completion method.

I am grateful to Professor Satoshi Matsuoka of Tokyo Institute of Technology and the members of his laboratory who provided us high performance PC clusters for the numerical results of Chapter 3 and Chapter 4. Their support has often made up for my lack of knowledge in parallel computation.

I would like to thank Dr. Maho Nakata of Tokyo University, who provided us the large scale SDPs arisen from quantum chemistry for the numerical experiments in Chapter 3.

I would like to express thanks to Dr. Mituhiro Fukuda of New York University. He gave me essential improvements on my English for each international conference. He also gave me a priceless opportunity for overseas education to study quantum chemistry at New York University.

I am also very grateful for the assistance of Professor Bastiaan J. Braams, Professor Michael Overton and Ms. Zhengji Zhao of New York University when I was visiting there. In particular, Ms. Zhao gave me many advices even after I returned to Japan. They led my interest to the research in quantum chemistry. The period at New York University was one of my unforgettable and precious experience and reminded me about fascinating aspects of mathematics.

I am also indebted to Professor Sunyoung Kim of Ewha Women's University for discussions on Second-Order Cone Programming. Her valuable suggestions enhanced my understanding on Conic Programming.

It is a pleasure to acknowledge the encouragements and support from Professor Yan Dai of University of Illinois at Chicago, Professor Antoine Deza of McMaster University and Professor Takeaki Uno of National Institute of Informatics.

I would like also to thank to Dr. Akiko Takeda for her support during this long time since the Master program for a number of helpful suggestions.

My special thanks are due to Dr. Satoko Moriguchi. She looks sweet in her cute smile. She motivated me in doing research, and cheered me up many time during these three years. Her great support led me to the completion of the thesis.

I would like to thank Professor Yukio Takahashi, Professor Miyoshi Naoto, Professor Hidetoshi Shimodaira and Ms. Yuko Sadoyama for their support and all members of Kojima Laboratory, Takahashi Laboratory, Miyoshi Laboratory and Shimodaira Laboratory for their friendship and cooperations. I have really enjoyed a good time with them.

Finally, I wish to express my appreciation to my family for their support and encouragement in my daily life.

I am delighted to have the opportunity to meet with a lot of people through my research. It is my precious delight that I express my thanks to many and many people.

Makoto Yamashita
Tokyo, March, 2004

Contents

List of Tables

List of Figures

Conventions

Symbols

m	the number of equality constraints
n	the size of variable matrices
$\mathbb{R}^m$	the m-dimension real space
$\mathbb{S}^n$	the set of $n \times n$ symmetric matrices
$\boldsymbol{C} \in \mathbb{S}^n$	a coefficient matrix
$\boldsymbol{A}_k \in \mathbb{S}^n \ (k = 1, 2, \ldots, m)$	input data matrices
$b_1, b_2, \ldots, b_m$	right hand side values
$\boldsymbol{X} \in \mathbb{S}^n$	a primal variable matrix
$\boldsymbol{Y} \in \mathbb{S}^n$	a dual variable matrix
$\boldsymbol{z} \in \mathbb{R}^m$	a dual variable vector
$d\boldsymbol{X} \in \mathbb{S}^n$	a primal matrix of a search direction
$d\boldsymbol{Y} \in \mathbb{S}^n$	a dual matrix a search direction
$d\boldsymbol{z} \in \mathbb{R}^m$	a dual vector a search direction
$\boldsymbol{B}d\boldsymbol{z} = \boldsymbol{r}$	the Schur complement equation
$\boldsymbol{B} \in \mathbb{S}^m$	the Schur complement matrix
$\boldsymbol{r} \in \mathbb{R}^m$	the right hand side of the Schur complement equation
N	a number of available processors
u	a rank of processor
$\mathcal{B}$	the row indices of $\boldsymbol{B}$
$\mathcal{P}_u$	the row indices assigned to the uth processor
σ	a permutation
$\{\boldsymbol{X}\}_l$	the lth block of $\boldsymbol{X}$
$[\boldsymbol{X}]_{ij}$	the (i, j)th element of $\boldsymbol{X}$
sb	a size of belt
mb	a row-wise size of block
nb	a column-wise size of block
$G(V, E)$	a graph with a vertex set V and an edge set E
$\overline{G}(V, \overline{E})$	an aggregate graph with an aggregate edge set $\overline{E}$
$\overline{\boldsymbol{X}}$	a matrix assigned by $\overline{E}$
$\widehat{G}(V, \widehat{E})$	an extended graph with an extended edge set $\widehat{E}$
$\widehat{\boldsymbol{X}}$	a matrix assigned by $\widehat{E}$
$\boldsymbol{M}$	the sparse Cholesky Factorization of $\widehat{\boldsymbol{X}}$
$\boldsymbol{N}$	the sparse Cholesky Factorization of $\boldsymbol{Y}^{-1}$
$[\boldsymbol{X}]_{*k}$	the kth column of $\boldsymbol{X}$

$\overline{\mathcal{B}}$	row indices of $\boldsymbol{B}$ which break the simple row-wise distribution
$\mathcal{Q}_u^i$	the column indices assigned to the uth processor to compute the ith row of $\boldsymbol{B}$
$\mathcal{R}_u$	the column indices assigned to the uth processor to compute $d\boldsymbol{X}$

Notation

$\boldsymbol{A} \bullet \boldsymbol{X}$	the inner-product between $\boldsymbol{A} \in \mathbb{S}^n$ and $\boldsymbol{X} \in \mathbb{S}^n$, $\boldsymbol{A} \bullet \boldsymbol{X} = \sum_{i=1}^n \sum_{j=1}^n A_{ij} X_{ij}$
$\boldsymbol{X} \succeq \boldsymbol{O}$	to indicate $\boldsymbol{X} \in \mathbb{S}^n$ is positive semidefinite
$\boldsymbol{X} \succ \boldsymbol{O}$	to indicate $\boldsymbol{X} \in \mathbb{S}^n$ is positive definite

Acronyms

CG	Conjugate Gradient Methods
D-IPM	Dual Interior-Point Methods
IPM	Interior-Point Methods
MT-PC	Mehrotra Type Predictor-Corrector
PD-IPM	Primal-Dual Interior-Point Methods
SDP	SemiDefinite Programming
SDPA	SemiDefinite Programming Algorithm
SDPARA	SemiDefinite Programming Algorithm paRAllel version
SDPARA-C	SemiDefinite Programming Algorithm paRAllel version with the Completion Method
TD-BCD	Two-Dimensional Block-Cyclic Distribution

Abbreviation

ELEMENTS	evaluation of the Schur complement matrix
CHOLESKY	the Cholesky Factorization of the Schur complement matrix
PMATRIX	computation for $d\boldsymbol{X}$
DENSE	computation for $n \times n$ matrices

Chapter 1

Introduction

The motivation of the thesis stems from the question,

"how we solve larger SDPs in shorter time."

The question is described in a very simple and compact way. However, answering to the question in details is difficult, complicated and far from its appearance. To answer the question, the thesis presents two parallel computer software *SDPARA* (*SemiDefinite Programming Algorithm paRAllel version*) and *SDPARA-C* (*SemiDefinite Programming Algorithm paRAllel version with the Completion method*).

The term *SDP* is a standard abbreviation of *"SemiDefinite Programming"* or *"SemiDefinite Program"*, which is a principal subject in the thesis. To solve larger SDPs in a shorter time, we rely on *parallel computation*, which is another principal subject of the thesis. The field of parallel computation experienced dramatic accomplishments in the last decade. Particularly, clustering and grid-computing technologies, which have been promoting rapid growth of the field in recent years, provide enormous resources for numerical computations in mathematics.

The SDP is an established mathematical model in the field of optimization. The SDP we are addressing in the thesis is defined as the primal-dual standard form.

$$\text{SDP}: \begin{cases} \mathcal{P}: & \text{minimize} & \boldsymbol{C} \bullet \boldsymbol{X} \\ & \text{subject to} & \boldsymbol{A}_k \bullet \boldsymbol{X} = b_k \quad (k = 1, 2, \ldots, m), \\ & & \boldsymbol{X} \succeq \boldsymbol{O}. \\ \mathcal{D}: & \text{maximize} & \sum_{k=1}^{m} b_k z_k \\ & \text{subject to} & \sum_{k=1}^{m} \boldsymbol{A}_k z_k + \boldsymbol{Y} = \boldsymbol{C}, \\ & & \boldsymbol{Y} \succeq \boldsymbol{O}. \end{cases} ,$$

where $\boldsymbol{X}, \boldsymbol{Y}$ are $n \times n$ symmetric variable matrices and $\boldsymbol{z}$ is a variable vector in $\mathbb{R}^n$. The $n \times n$ symmetric matrices $\boldsymbol{C}, \boldsymbol{A}_1, \ldots, \boldsymbol{A}_m$ and the scalars $b_1, b_2, \ldots, b_m$ are given. We use the inner-product between two symmetric matrices $\boldsymbol{A} \bullet \boldsymbol{X} = \sum_{i=1}^{n} \sum_{j=1}^{n} A_{ij} X_{ij}$ and the notation $\boldsymbol{X} \succeq \boldsymbol{O}$ to indicate that a symmetric matrix $\boldsymbol{X}$ is a positive semidefinite matrix, that is, all of eigenvalues of $\boldsymbol{X}$ are non-negative. Chapter 2 describes more details about SDPs and the algorithmic framework for solving them.

In the last decade, powerful *Primal-Dual Interior-Point Methods* (PD-IPM) [35, 42, 57, 64] were proposed for SDPs, and several software [10, 67, 74] including SDPA [23] based on them were applied to many applications of SDPs from various fields. For example, Linear Matrix Inequality arisen from Lyapnov stability criteria in system and control theory [12] can be directly reformulated into an SDP. In financial engineering [53], SDPs are used to find a minimum variance keeping a mean above a given threshold. Robust optimization in engineering [7] utilizes SDPs to formulate optimization problems on uncertain situations.

Solving SDPs not only contributes their direct applications but also provides stable and efficient numerical methods to other optimization problems; SDPs are embedded to solve other optimization problems. In combinatorial optimization [27, 28, 72], SDPs generate approximate optimal values for Max Cut Problems, which are known as NP-complete problems, with an inexpensive computation cost. Successive Convex Relaxation Methods [43, 44, 45, 69] compute approximate optimal solutions of a non-convex optimization problem by solving many SDPs as their sub problems. The SDP has a lot of direct and indirect applications. Many survey papers [32, 72, 82, 83, etc.] on SDPs were also published.

Recently SDPs draw increasing attentions in the fields of data-mining [48] and quantum chemistry [61, 62, 91]. In particular, it is interesting to observe that SDPs in quantum chemistry are formulated through the principle of least action indicated in the textbook of Feynman [18]. We expect that more and more applications to various fields will be found in the next decade.

At the present time sandwiched between the last and next decades, SDPs we are facing become larger and larger. Some are beyond the effective range of the existing software working on a single processor, because they require enormous memory space and/or can not be solved in a practical computation time. Accuracy and flexibility in formulating SDP models from real problems are strongly dependent on how large number m of equality constraints and how large-size n of variable matrices are allowed to take in modeling. For example, if we pursuit higher accuracy of electron orbits in quantum chemistry, the number m of constraints increase rapidly so that we have to solve an SDP with an extremely large m, often greater than $10,000$. On the other hand, it is common in combinatorial optimization to encounter SDPs involving large-size variable matrices with some special structures. The larger m and n require much computation cost and more memory space to solve resulting SDPs numerically.

The main part of the thesis is the proposal of the two parallel software SDPARA and SDPARA-C for solving the large SDPs with high accuracy in a short time, and the reports of their excellent performance enhanced by experimental analyses. SDPARA is developed to solve large SDPs involving large m equality constraints in a short time, while SDPARA-C is designed for SDPs with large-size variable matrices. Throughout designing and implementing of two software, the following points are remarkable for their performance.

- In designing of SDPARA (for large m),
  1. The reduction of computation time in evaluation of so-called the Schur complement matrix $\boldsymbol{B}$, which is a bottleneck in PD-IPM, is brought by its row-wise distributed evaluation on multiple processors.
  2. The row-wise distribution keeps the efficient handling of the sparsity in input data matrices $\boldsymbol{A}_1, \boldsymbol{A}_2, \ldots, \boldsymbol{A}_m$ without yielding extra network communication.
  3. The parallel Cholesky factorization of the matrix $\boldsymbol{B}$ on the two-dimensional block-cyclic distribution enables us to acquire the solution of the Schur complement equation in a short time.
- In designing of SDPARA-C (for large n),
  1. We incorporate *the completion method* proposed in [24, 59] into SDPARA to reduce memory space for SDPs involving large-size but sparse input data matrices.
  2. Together with the distribution of the Schur complement matrix $\boldsymbol{B}$ on multiple processors, we can dramatically increase the size n of variable matrices involved in practically solvable SDPs.
  3. Hashed row-wise distribution for $\boldsymbol{B}$ and parallel computation of a search direction further shorten computation time.

Wide range experimental analyses considerably enhanced the performance of the two software. In particular, it is significant to take account of the capacity of physical network between multiple processors. In designing of SDPARA and SDPARA-C, lots of experimental analyses were done to reduce the amount of data transferred among processors. As a result, SDPARA and SDPARA-C achieved a high performance in numerical experiments as below.

- SDPARA achieved
  1. SDPARA on 64 processors can solve SDPs arisen from control theory or quantum chemistry more than 20 times faster than SDPA on a single processor.
  2. SDPARA can also solve SDPs with a huge number m of equality constraints arisen from quantum chemistry, which could not be solved before, in a practical time.
- SDPARA-C achieved
  1. SDPARA-C can solve SDPs whose sizes are beyond the capability of other existing software.

2. In particular, SDPARA-C can solve an SDP of Max Cut Problem with $40,000 \times 40,000$ variable matrices on lattice graph and $40,000$ equality constraints.

Thus, a combined use of SDPARA and SDPARA-C is our answer to the question "how we solve larger SDPs in shorter time." The proposal of the two software means plentiful applications in the above fields will be much accessible for us and they will enlarge our surrounding environments.

In Chapter 3, we discuss how we reduce computation time in *SDPARA* (*SemiDefinite Programming Algorithm paRAllel version*). Generally speaking, the most computation time to solve many SDPs by PD-IPM is occupied by construction of a system of linear equations, so-called the Schur complement equation which plays an essential role in computing a search direction in each iteration, and its solution. For example, this bottleneck regarding the Schur complement equation is serious in the case of quantum chemistry. In PD-IPM, we continue updating $(\boldsymbol{X}, \boldsymbol{Y}, \boldsymbol{z})$ along a search direction $(d\boldsymbol{X}, d\boldsymbol{Y}, d\boldsymbol{z})$ such that $(\boldsymbol{X}, \boldsymbol{Y}, \boldsymbol{z}) = (\boldsymbol{X}, \boldsymbol{Y}, \boldsymbol{z}) + (d\boldsymbol{X}, d\boldsymbol{Y}, d\boldsymbol{z})$ until $(\boldsymbol{X}, \boldsymbol{Y}, \boldsymbol{z})$ reaches an approximate optimal solution. The Schur complement equation $\boldsymbol{B}d\boldsymbol{z} = \boldsymbol{r}$ generates the component $d\boldsymbol{z}$ of the search direction. The Schur complement matrix $\boldsymbol{B}$ is an $m \times m$ fully dense positive definite matrix and its elements are in the form $B_{ij} = (\boldsymbol{X}\boldsymbol{A}_i\boldsymbol{Y}^{-1}) \bullet \boldsymbol{A}_j$. In SDPARA, we adopted row-wise distributed evaluation of the Schur complement matrix $\boldsymbol{B}$ on multiple processors to reduce the computation time. It should be emphasized that an independent evaluation of each row of $\boldsymbol{B}$ is available and the row-wise evaluation of $\boldsymbol{B}$ is also reasonable from viewpoints of parallel processing since all elements in the ith row share the common computation of $\boldsymbol{X}\boldsymbol{A}_i\boldsymbol{Y}^{-1}$. In addition, the row-wise distribution provides us a natural storage scheme for the large-size matrix $\boldsymbol{B}$ on distributed memory.

When applying parallel processing based on the row-wise distribution, however, a simple combination of a diagonal block structure with the exploitation of sparsity proposed by [22] for input data matrices is not consistent with less network communication to accumulate the evaluation results. SDPARA removed the sorting [22] of input data matrices to eliminate surplus communication, and accomplished significant reduction of computation time with excellent scalability in the evaluation of $\boldsymbol{B}$.

Moreover, the parallel Cholesky Factorization on multiple processors shortens computation time to acquire the solution of the Schur complement equation. Particularly, the redistribution from the row-wise distribution used for the evaluation of $\boldsymbol{B}$ to the two-dimensional block-cyclic distribution makes the parallel Cholesky Factorization more effective; the overhead of additional network communication caused by the redistribution is relatively minor.

Consequently, SDPARA solves extremely large-scale SDPs with a large number of equality constraints arisen from quantum chemistry in a short time which we could not attain before. SDPARA also solves some other large-scale SDPs very fast. Additional numerical results show that SDPARA attains satisfying scalability and load-balance with an advantage of the simple but powerful row-wise distribution of $\boldsymbol{B}$ and the parallel Cholesky Factorization on the two-dimensional block-cyclic distribution.

In Chapter 4, we propose *SDPARA-C* (*SemiDefinite Programming with the Completion method*) for another type of large-scale SDPs. SDP relaxations of combinatorial optimization problems often require to solve SDPs with large-size input data matrices $\boldsymbol{C}, \boldsymbol{A}_1, \ldots, \boldsymbol{A}_m$. If we apply SDPARA to such SDPs, the effectiveness of parallel computation of the Schur complement equation reduces and the portion in computation time to handle dense variable matrices becomes larger. Taking this disadvantage of SDPARA into account, we incorporated the completion method proposed in [24, 59] into SDPARA to exploit structural sparsity in input data matrices. The dual variable matrix $\boldsymbol{Y}$ inherits the sparsity of the input data matrices, but the primal variable matrix $\boldsymbol{X}$ becomes fully dense in general. In the completion method, many elements in $\boldsymbol{X}$ are neither explicitly computed nor stored in memory through the adoption of the sparsity structure in $\boldsymbol{X}$ characterized by the chordal graph property. Therefore, we can considerably save the growth of memory space even when the size n of variable matrices increases. In fact, the size n of variable matrices in SDPs which can be handled by SDPARA-C is far beyond the range of other software.

However, the row-wise distribution of SDPARA can not preserve its load-balance if we simply combine SDPARA with the completion method, because we can not directly access the elements of the dense matrices $\boldsymbol{X}$ and $\boldsymbol{Y}^{-1}$ in the completion method. Instead of $\boldsymbol{X}$ and $\boldsymbol{Y}^{-1}$, we hold their sparse Cholesky Factorization matrices. Thus the formula to evaluate $\boldsymbol{B}$ is modified to utilize the sparse factorization of $\boldsymbol{X}^{-1}$ and $\boldsymbol{Y}$ in substitutions for $\boldsymbol{X}$ and $\boldsymbol{Y}^{-1}$. Then computation cost for the ith row of $\boldsymbol{B}$ becomes heavily dependent on the number of non-zero column vectors of $\boldsymbol{A}_i$ so that the modification often breaks down the load-balance of the simple row-wise distribution. Specifically, in some SDPs arisen from combinatorial optimization, a sharp bias of the numbers of non-zero column vectors in input data matrices is common; hence the load-balance

of the simple row-wise distribution becomes worse. To overcome the difficulty, we apply hashed parallel processing to partition the computation of the elements $B_{ij} = (\boldsymbol{X}\boldsymbol{A}_i\boldsymbol{Y}^{-1}) \bullet \boldsymbol{A}_j$ $(j = 1, 2, \ldots, m)$ with many non-zero column vectors in $\boldsymbol{A}_i$. We divide the computation for such a row on multiple processors and accumulate the results to generate entire evaluation of $B_{ij}(j = 1, 2, \ldots, m)$ in the ith row of $\boldsymbol{B}$. The hashed row-wise distribution substantially improved load-balance in case of unbalanced input data matrices.

In addition, SDPARA-C replaced the serial computation of search direction $d\boldsymbol{X}$ with its parallel implementation. When the size n of matrices becomes large, we can not neglect the portion to compute $d\boldsymbol{X}$. The column-wise computation of $d\boldsymbol{X}$ in [59] provides us a practical method to apply parallel processing. We distribute some serial columns of $d\boldsymbol{X}$ to each processor instead of a stereotypic cyclic-distribution in order to reduce network communication cost for mutual broadcast of $d\boldsymbol{X}$ to retain entire elements on each processor for subsequent computation. Furthermore, minimizing the number of elements of $d\boldsymbol{X}$ to be broadcasted by the completion method cuts considerable amount of network communication.

In consequence, the reduction of memory space for dense matrices owing to the completion method and the distribution of $\boldsymbol{B}$ on distributed memory enable SDPARA-C to handle SDPs involving the extremely large-size n of variable matrices and solve them in a short time with the aid of parallel processing. SDPARA-C inherits the powerful performance from both SDPARA and the completion method.

Numerical results on PC cluster exhibit that computation time of SDPARA and the size of SDPs which SDPARA-C can handle are far superior to that of other existing software.

We start the thesis with describing some definitions related to SDPs and more features of PD-IPM in Chapter 2. Chapter 3 and 4 are devoted to SDPARA and SDPARA-C, respectively. Numerical results on them are also given there. Finally, we summarize the main assertions and results of the thesis and mention some future directions in Chapter 5.

Chapter 2

Semidefinite Programming and Primal-Dual Interior-Point Methods

In this chapter, we start from a mathematical definition of SDP and its significant characteristics in section 2.1. Then, section 2.2 describes an algorithmic framework of Primal-Dual Interior-Point Methods which are known as powerful and stable methods to solve SDPs numerically. The details of how to implement the framework on a single processor will be discussed in section 2.3 following the SDPA implementation. This chapter is devoted to introduce fundamental concepts in advance of the discussions for parallel implementation.

2.1 SemiDefinite Programming

We begin with some mathematical notations, before introducing a standard form of *SemiDefinite Programming* (SDP). Let $\mathbb{S}^n$ be the space of $n \times n$ symmetric matrices. We also use a notation $\boldsymbol{X} \succeq \boldsymbol{O} (\boldsymbol{X} \succ \boldsymbol{O})$ to indicate an $n \times n$ symmetric matrix $\boldsymbol{X}$ is positive semidefinite (positive definite), respectively. Additionally, we take an inner-product between two symmetric matrices, that is, $\boldsymbol{U} \bullet \boldsymbol{V} = \sum_{i=1}^{n} \sum_{j=1}^{n} U_{ij} V_{ij}$ for $\boldsymbol{U}, \boldsymbol{V} \in \mathbb{S}^n$.

Given a number of equality constraints m, a size of matrices n, a coefficient matrix $\boldsymbol{C} \in \mathbb{S}^n$, input data matrices $\boldsymbol{A}_1, \boldsymbol{A}_2, \ldots, \boldsymbol{A}_m \in \mathbb{S}^n$, and right hand side values $b_1, b_2, \ldots, b_m \in \mathbb{R}$, we introduce a standard form of SDP, the main interest which we are addressing,

$$
\text{SDP} : \left\{
\begin{array}{lll}
\mathcal{P} : & \text{minimize} & \boldsymbol{C} \bullet \boldsymbol{X} \\
& \text{subject to} & \boldsymbol{A}_k \bullet \boldsymbol{X} = b_k \quad (k = 1, 2, \ldots, m), \\
& & \boldsymbol{X} \succeq \boldsymbol{O}. \\
\mathcal{D} : & \text{maximize} & \sum_{k=1}^{m} b_k z_k \\
& \text{subject to} & \sum_{k=1}^{m} \boldsymbol{A}_k z_k + \boldsymbol{Y} = \boldsymbol{C}, \\
& & \boldsymbol{Y} \succeq \boldsymbol{O}.
\end{array}
\right.
$$

We know at a glance that the form embodies the primal-dual definition. The primal definition ($\mathcal{P}$) and the dual definition ($\mathcal{D}$) are equivalent in sense that we can acquire the same quality of information from both definitions. Hence, we have a selection on solving SDPs with only the one definition or solving them using both of the definitions.

In the primal definition ($\mathcal{P}$), we want to decide the variable matrix $\boldsymbol{X} \in \mathbb{S}^n$ which minimizes the linear objective function $\boldsymbol{C} \bullet \boldsymbol{X}$ satisfying the linear equality constraints $\boldsymbol{A}_k \bullet \boldsymbol{X} = b_k$ $(k = 1, 2, \ldots, m)$ and positive semidefiniteness of $\boldsymbol{X}$. Each linear constraints $\boldsymbol{A}_k \bullet \boldsymbol{X} = b_k$ draws a hyper-plane in the space $\mathbb{S}^n$, however, positive semidefiniteness is a nonlinear constraint.

On the other hand, in the dual definition ($\mathcal{D}$), we want to decide the variable matrix $\boldsymbol{Y} \in \mathbb{S}^n$ and the variable vector $\boldsymbol{z} \in \mathbb{R}^m$ which maximize the linear objective function $\sum_{k=1}^{m} b_k z_k$ satisfying the linear matrix equality $\sum_{k=1}^{m} \boldsymbol{A}_k z_k + \boldsymbol{Y} = \boldsymbol{C}$ and positive semidefiniteness of $\boldsymbol{Y}$. When we combine the linear matrix equality and the positive semidefiniteness by substituting $\boldsymbol{Y}$, the result ($\boldsymbol{C} - \sum_{k=1}^{m} \boldsymbol{A}_k z_k \succeq \boldsymbol{O}$) can be

viewed as a Linear Matrix Inequality (LMI). LMI plays an essential role for control theory [12]. Therefore, we can consider that SDP covers the field of LMI in itself.

Apart from the LMI, SDP covers many mathematical programmings. If we restrict $\mathbb{S}^n$ in the above definition into a space of $n \times n$ diagonal matrices, we have Linear Programming, the most fundamental programming in the field of mathematical programming. It is also well known that we can classify Second-Order Cone Programming and Quadratic Convex Optimization Programming into special cases of SDP.

In the above formulation, we call the matrix $\boldsymbol{X}$ (the pair of the matrix $\boldsymbol{Y}$ and the vector $\boldsymbol{z}$) a primal (dual) feasible solution or a primal (dual) feasible point if $\boldsymbol{X}$ ($\boldsymbol{Y}$ and $\boldsymbol{z}$) satisfy all constraints in $\mathcal{P}$ ($\mathcal{D}$), respectively. When $\boldsymbol{X}$ ($\boldsymbol{Y}$) is positive definite in addition to the feasibility, we call $\boldsymbol{X}$ ($\boldsymbol{Y}$ and $\boldsymbol{z}$) a primal (dual) interior point, respectively. If a feasible solution $\boldsymbol{X}$ ($\boldsymbol{Y}$ and $\boldsymbol{z}$) attains minimum (maximum), then we call a primal (dual) optimal solution, respectively. On conditions that $\boldsymbol{X}$ is a primal feasible solution (interior point, optimal solution) and the pair of $\boldsymbol{Y}$ and $\boldsymbol{z}$ is a dual feasible solution (interior point, optimal solution), we call $(\boldsymbol{X}, \boldsymbol{Y}, \boldsymbol{z}) \in \mathbb{S}^n \times \mathbb{S}^n \times \mathbb{R}^m$ a primal-dual feasible solution (a primal-dual interior point, a primal-dual optimal solution), or simply a feasible solution (an interior point, an optimal solution), respectively. The (weak) duality theorem shows the relation between the objective values of $\mathcal{P}$ and $\mathcal{D}$.

Theorem 2.1.1 (The (weak) Duality Theorem) : *For any feasible solution $(\boldsymbol{X}, \boldsymbol{Y}, \boldsymbol{z})$, an inequality between the primal and dual objective values,*

$$\boldsymbol{C} \bullet \boldsymbol{X} \leq \Sigma_{k=1}^{m} b_k z_k,$$

holds.

From the weak duality theorem, it is apparent that the coincidence of the objective values in addition to the feasibility stands for the optimum. However, in general SDPs, there may be duality gap between the optimal values of ($\mathcal{P}$) and ($\mathcal{D}$). An example with positive duality gap is mentioned in [72].

Throughout this thesis, we make the following two assumptions.

- A set of the input data matrices $\{\boldsymbol{A}_1, \boldsymbol{A}_2, \ldots, \boldsymbol{A}_m\}$ is linearly independent.
- There exists an interior point $(\boldsymbol{X}, \boldsymbol{Y}, \boldsymbol{z})$ for the SDP.

We can remove the first assumption if we pre-process a method such a Gaussian elimination. In the meantime, the second assumption, so-called the Slater condition is essential for the primal-dual formulation to establish the (strong) duality theorem

Theorem 2.1.2 (The (strong) Duality Theorem) : *Suppose that both ($\mathcal{P}$) and ($\mathcal{D}$) have an interior feasible solution. Then the feasible regions of ($\mathcal{P}$) and ($\mathcal{D}$) become compact set. Furthermore, the optimal values of ($\mathcal{P}$) and ($\mathcal{D}$) are equal.*

An excellent proof of the strong duality theorem can be found in [72]. The coincidence property of the optimal values is very useful, since we can add the property as one kind of constraints when we search an optimal solution. In other words, without the second assumption we need to search in wider space at which the two optimal objective values may not coincide.

2.2 Primal-Dual Interior-Point Methods

So far, many methods have been developed to solve SDPs. Here, we focus on Primal-Dual Interior-Point Methods (PD-IPM) among such methods. We will consider characteristics of other methods and comparisons with PD-IPM in section 2.4.

Historically, Interior-Point Methods (IPM) were introduced for the first time by Karmarkar [36, 37] to solve Linear Programming. Compared to simplex methods, an excellent virtue of IPM is that its computation cost can be bounded polynomial time. Furthermore, practical results have shown that IPM are superior to Ellipsoid Methods developed by Kchachiyan [38],which also possess the polynomial time property.

An remarkable breakthrough in the field of Interior-Point Methods was brought by Nesterov and Nemirovskii [63]. They established a concept of self-concordant barrier functions for general convex optimization programming. The concept plays an fundamental role in the proof that there exist polynomial time

methods for any general convex optimization programming that has their self-concordant barrier functions. Since not only SDP but also Second-Order Cone Programming and Linear Programming are sub-classes of general convex optimization programming, the proof ensures they can be solved effectively from the theoretical viewpoints.

PD-IPM were initially developed for Linear Complement Problems (LCP) by Kojima, *et al.* [41]. A significant improvement of PD-IPM is its high stability because PD-IPM solve both of the primal and the dual LCPs simultaneously based on the duality theorem. After the first impact of PD-IPM, they were extended to Linear Programming; See textbooks written by Wright [84] and by Ye [87] for the details of PD-IPM for Linear Programming. One flows of the further extensions of PD-IPM were made toward SDP [35, 42, 57, 64, 72]. Currently, many computer software including SDPA (SemiDefinite Programming Algorithm) are available to solve SDPs.

In advance of details how PD-IPM solve SDPs, we should make sure the optimal condition of the standard form of SDP. Under the second assumption in the previous section 2.1 (the Slater condition), it is known that the KKT condition provides a necessary and sufficient condition for $(\boldsymbol{X}, \boldsymbol{Y}, \boldsymbol{z})$ to be an optimal solution for an SDP.

$$\text{KKT:} \quad \begin{cases} \boldsymbol{A}_k \bullet \boldsymbol{X} = b_k \ (k = 1, 2, \ldots, m), \\ \sum_{k=1}^{m} \boldsymbol{A}_k z_k + \boldsymbol{Y} = \boldsymbol{C}, \\ \boldsymbol{X} \succeq \boldsymbol{O}, \quad \boldsymbol{Y} \succeq \boldsymbol{O}, \\ \boldsymbol{X}\boldsymbol{Y} = \boldsymbol{O}. \end{cases} \tag{2.1}$$

We can see that the KKT condition (2.1) for SDPs is composed of the four constraints. The first and the second constraints are exactly the primal and the dual equality constraints, respectively. The third constraint guarantees the positive semidefiniteness of matrices $\boldsymbol{X}$ and $\boldsymbol{Y}$. If these three constraints are satisfied, the last constraint, so-called complementarity condition, is equivalent that the primal and the dual optimal objective values attain the identical value. See [72] for the details.

Modifying the complementarity condition slightly with a perturbation parameter $\mu > 0$, we acquire an system which defines a central path.

$$\text{Central Path:} \quad \left\{ (\boldsymbol{X}, \boldsymbol{Y}, \boldsymbol{z}) \in \mathbb{S}^n \times \mathbb{S}^n \times \mathbb{R}^m : \begin{array}{l} \boldsymbol{A}_k \bullet \boldsymbol{X} = b_k \ (k = 1, 2, \ldots, m), \\ \sum_{k=1}^{m} \boldsymbol{A}_k z_k + \boldsymbol{Y} = \boldsymbol{C}, \\ \boldsymbol{X} \succeq \boldsymbol{O}, \quad \boldsymbol{Y} \succeq \boldsymbol{O}, \\ \boldsymbol{X}\boldsymbol{Y} = \mu \boldsymbol{I}. \end{array} \right\}, \tag{2.2}$$

where the matrix $\boldsymbol{I}$ is the $n \times n$ identity matrix. Obviously, we know that a point on the central path with $\mu = 0$ is an optimal solution, since we address the same constraints as the KKT condition (2.1). Two well-known facts are shown in [42]; the one is that if we fix the parameter μ to a positive number, there exists a unique point on the central path; the other is that the central path comprises of an continuous curve which converges an optimal solution as $\mu \to 0$. In addition, an equality $\mu = \boldsymbol{X} \bullet \boldsymbol{Y}/n$ holds for every point on the central path.

A main concept adopted by PD-IPM is that PD-IPM numerically traces the central path decreasing the perturbation parameter μ toward 0; when μ is sufficiently close to 0, PD-IPM reach to an approximate optimal solution. To describe an approximate optimal solution precisely, we define a feasibility error and a relative duality gap as follows.

$$\begin{aligned} \text{A feasibility error:} \quad & \max\{\text{a primal feasibility error}, \text{a dual feasibility error}\}, \\ \text{A relative duality gap:} \quad & \frac{|\text{objP} - \text{objD}|}{\max\{(|\text{objP}| + |\text{objD}|)/2.0, 1.0\}}, \end{aligned}$$

where

$$\begin{aligned} \text{A primal feasibility error:} \quad & \max\{|\boldsymbol{A}_k \bullet \boldsymbol{X} - b_k| : k = 1, 2, \ldots, m\}, \\ \text{A dual feasibility error:} \quad & \max\left\{ \left| [\Sigma_{k=1}^{m} \boldsymbol{A}_k z_k + \boldsymbol{Y} - \boldsymbol{C}]_{i,j} \right| : i, j = 1, 2, \ldots, n \right\}, \\ \text{objP:} \quad & \boldsymbol{C} \bullet \boldsymbol{X}, \\ \text{objD:} \quad & \Sigma_{k=1}^{m} b_k z_k. \end{aligned}$$

Other definition of a primal feasibility error and a dual feasibility error also can be employed, for example,

$$\begin{aligned} \text{An alternate primal feasibility error:} \quad & \left(\Sigma_{k=1}^{m} |\boldsymbol{A}_k \bullet \boldsymbol{X} - b_k|^2\right)^{1/2} \\ \text{An alternate dual feasibility error:} \quad & ||\Sigma_{k=1}^{m} \boldsymbol{A}_k z_k + \boldsymbol{Y} - \boldsymbol{C}||. \end{aligned}$$

We call $(\boldsymbol{X}, \boldsymbol{Y}, \boldsymbol{z})$ as an $\bar{\epsilon}$-feasible solution if the feasibility error is smaller than a small positive number $\bar{\epsilon}$ and $\boldsymbol{X}$ and $\boldsymbol{Y}$ are positive semidefinite. Moreover, if the relative gap of $(\boldsymbol{X}, \boldsymbol{Y}, \boldsymbol{z})$ is smaller than a small positive number ϵ^* in addition to $\bar{\epsilon}$-feasibility, we say $(\boldsymbol{X}, \boldsymbol{Y}, \boldsymbol{z})$ as an $(\bar{\epsilon}, \epsilon^*)$-optimal solution. At the time PD-IPM find an $(\bar{\epsilon}, \epsilon^*)$-optimal solution, PD-IPM terminate and give the $(\bar{\epsilon}, \epsilon^*)$-optimal solution as an approximate optimal solution. Usually, we take $\bar{\epsilon} = \epsilon^* = 1.0e-8$.

The final substantial element in PD-IPM is how we numerically trace the central path, that is, how we move the current point $(\boldsymbol{X}, \boldsymbol{Y}, \boldsymbol{z})$ to a next point $(\boldsymbol{X}_{next}, \boldsymbol{Y}_{next}, \boldsymbol{z}_{next})$. We introduce a search direction $(d\boldsymbol{X}, d\boldsymbol{Y}, d\boldsymbol{z})$ to update the current point as follow,

$$(\boldsymbol{X}_{next}, \boldsymbol{Y}_{next}, \boldsymbol{z}_{next}) = (\boldsymbol{X}, \boldsymbol{Y}, \boldsymbol{z}) + (d\boldsymbol{X}, d\boldsymbol{Y}, d\boldsymbol{z}). \tag{2.3}$$

Desired characteristics of the next point are that it will be close to the central path and that an approximate perturbation parameter $\mu_{next} = \boldsymbol{X}_{next} \bullet \boldsymbol{Y}_{next}/n$ will be smaller than current value $\mu_{current} = \boldsymbol{X} \bullet \boldsymbol{Y}/n$ to make the next point closer to an optimal solution at which the KKT condition (2.1) holds. Therefore, we apply the Newton method to the system (2.2) that defines the central path in order to obtain the search direction in (2.3), and acquire a system,

$$\begin{cases} \boldsymbol{A}_k \bullet (\boldsymbol{X} + d\boldsymbol{X}) = b_k \ (k = 1, 2, \ldots, m), \\ \sum_{k=1}^{m} \boldsymbol{A}_k (z_k + dz_k) + (\boldsymbol{Y} + d\boldsymbol{Y}) = \boldsymbol{C}, \\ (\boldsymbol{X} + d\boldsymbol{X})(\boldsymbol{Y} + d\boldsymbol{Y}) = \beta \ (\boldsymbol{X} \bullet \boldsymbol{Y}/n) \ \boldsymbol{I}, \end{cases}$$

with a parameter β from 0 to 1. If we set β close to 0, we obtain an aggressive search direction toward an optimal solution. Too aggressive search direction, however, suffers from a numerical unstability and a difficulty to attain the feasibility. Conversely, if we set β close to 1, the search direction leads the next point to a neighborhood of the central path, thus successive updates can be done on a better condition. Since we can control numerically the next point between an optimal solution and the central path by adjusting β, we call the parameter β the centering parameter. It should be emphasized that the above system for the search direction does not contain positive semidefiniteness of $\boldsymbol{X}$ and $\boldsymbol{Y}$, because it is difficult to apply the Newton method to the constraints that are not equality constraints. Consequently, we start from an initial point with $\boldsymbol{X} \succ \boldsymbol{O}, \boldsymbol{Y} \succ \boldsymbol{O}$ and keep positive definiteness by shrinking the search direction with step length $\alpha_p = \max\{\alpha : \boldsymbol{X} + \alpha \ d\boldsymbol{X} \succeq \boldsymbol{O}\}$ for the primal and $\alpha_d = \max\{\alpha : \boldsymbol{Y} + \alpha \ d\boldsymbol{Y} \succeq \boldsymbol{O}\}$ for the dual. The feature that we retain the positive definiteness during all the computation in PD-IPM is the origin of the name 'Interior-Point Methods.'

At the last of this section, we summarize the contents in the form of an algorithmic framework of PD-IPM.

Algorithmic Framework of Primal-Dual Interior-Point Methods

Step 0 (Initialization): Choose an initial point $(\boldsymbol{X}, \boldsymbol{Y}, \boldsymbol{z})$ satisfying $\boldsymbol{X} \succ \boldsymbol{O}$ and $\boldsymbol{Y} \succ \boldsymbol{O}$; for example, $\boldsymbol{X} = \boldsymbol{Y} = \boldsymbol{I}$ and $\boldsymbol{z} = \boldsymbol{0}$. Set parameters $0 < \beta < 1, 0 < \gamma < 1, \bar{\epsilon} > 0, \epsilon^* > 0$.

Step 1 (Terminally check): If $(\boldsymbol{X}, \boldsymbol{Y}, \boldsymbol{z})$ is an $(\bar{\epsilon}, \epsilon^*)$-optimal solution, we print out $(\boldsymbol{X}, \boldsymbol{Y}, \boldsymbol{z})$ as an approximation optimal solution and stop algorithm.

Step 2 (Next point search): We compute the search direction $(d\boldsymbol{X}, d\boldsymbol{Y}, d\boldsymbol{z})$ by the Newton method, and the step lengths α_p and α_d.

Step 3 (Update): We update the point by $(\boldsymbol{X}, \boldsymbol{Y}, \boldsymbol{z}) \leftarrow (\boldsymbol{X}, \boldsymbol{Y}, \boldsymbol{z}) + \gamma(\alpha_p \ d\boldsymbol{X}, \alpha_d \ d\boldsymbol{Y}, \alpha_d \ d\boldsymbol{z})$, and return to Step 1.

We regard the successive computations from Step 1 to Step 3 as one iteration. The parameters set by Step 0 affect the number of iterations that PD-IPM require to terminate the algorithm.

2.3 SDPA (SemiDefinite Programming Algorithm)

SDPA (SemiDefinite Programming Algorithm) [23] is software package for solving SDPs based on PD-IPM. SDPA is known as stable and efficient software.

The first version of SDPA was released in 1995. Throughout continuous improvements devoted to SDPA since the first release [20], SDPA contains many schemes to enhance its solvability. The major schemes among them are the exploitation sparsity of the input data matrices for the so-called Schur complement matrix and the Lanczos method to reduce the computation cost for the step length. Moreover, the latest version of SDPA (version 6.0) [85] takes advantage of the fast numerical libraries for linear algebra, LAPACK (Linear Algebra PACKage) [3] and ATLAS (Automatically Tuned Linear Algebra Software), to achieve further shorter computation time. In this section, we describe more descriptions of such schemes incorporated into SDPA from the viewpoints of implementation.

The source code and information of SDPA are maintained on the SDPA Home Page:
http://sdpa.is.titech.ac.jp/

2.3.1 Search Direction

As described in section 2.2, we need to solve the system by applying the Newton method to obtain the search direction $(d\boldsymbol{X}, d\boldsymbol{Y}, d\boldsymbol{z})$ in (2.3) at each iteration.

$$\begin{cases} \boldsymbol{A}_k \bullet (\boldsymbol{X} + d\boldsymbol{X}) = b_k \ (k = 1, 2, \ldots, m), \\ \sum_{k=1}^{m} \boldsymbol{A}_k (z_k + dz_k) + (\boldsymbol{Y} + d\boldsymbol{Y}) = \boldsymbol{C}, \\ (\boldsymbol{X} + d\boldsymbol{X})(\boldsymbol{Y} + d\boldsymbol{Y}) = \mu_c \boldsymbol{I}, \end{cases} \tag{2.4}$$

where $\mu_c = \beta \ (\boldsymbol{X} \bullet \boldsymbol{Y}/n)$. We ignore a minute nonlinear term $d\boldsymbol{X}d\boldsymbol{Y}$ in the third constraint of (2.4) and keep only the variable terms in the left hand side, thus we acquire the next system.

$$\begin{cases} \boldsymbol{A}_k \bullet d\boldsymbol{X} = b_k - \boldsymbol{A}_k \bullet \boldsymbol{X} \ (k = 1, 2, \ldots, m), \\ \sum_{k=1}^{m} \boldsymbol{A}_k dz_k + d\boldsymbol{Y} = \boldsymbol{C} - \sum_{k=1}^{m} \boldsymbol{A}_k z_k - \boldsymbol{Y}, \\ \boldsymbol{X}\ d\boldsymbol{Y} + d\boldsymbol{X}\ \boldsymbol{Y} = \mu_c \boldsymbol{I} - \boldsymbol{X}\boldsymbol{Y}, \end{cases} \tag{2.5}$$

However, we can not solve the system directly because of a conflict between the number of variables and that of equality constraints in the system. Since that an $n \times n$ symmetric matrix involves $n(n+1)/2$ free variables as its elements, we count the number of variables $(d\boldsymbol{X}, d\boldsymbol{Y}, d\boldsymbol{z})$ is $n(n+1)/2 + n(n+1)/2 + m$. On the other hand, the number of equality constraints is $m + n(n+1)/2 + n^2$, because we must consider the third constraint in the space of non-symmetric matrices while the second constraint in $\mathbb{S}^n$.

Some approaches has been proposed to overcome the conflict [76]. Monteiro and Zhang [57, 88] brought a general perspective to integrate many of such approaches. They replace the third constraint in (2.5) by its symmetrization

$$H_{\boldsymbol{P}}(\boldsymbol{X}\ d\boldsymbol{Y} + d\boldsymbol{X}\ \boldsymbol{Y}) = H_{\boldsymbol{P}}(\mu_c \boldsymbol{I} - \boldsymbol{X}\boldsymbol{Y}),$$

to reduce $n(n-1)/2$ equality, where $H_{\boldsymbol{P}}(\boldsymbol{X}) = (\boldsymbol{P}\boldsymbol{X}\boldsymbol{P}^{-1} + (\boldsymbol{P}\boldsymbol{X}\boldsymbol{P}^{-1})^T)/2$ is a symmetrization operator based on a $n \times n$ non-singular matrix $\boldsymbol{P}$. Three major search directions, HRVW/KSH/M [35, 42, 57], NT [64, 73], AHO [1] can be derived from the adequate choices of the matrix $\boldsymbol{P}$ as $\boldsymbol{Y}^{1/2}, \boldsymbol{X}^{1/2}(\boldsymbol{X}^{1/2}\boldsymbol{Y}\boldsymbol{X}^{1/2})^{-1/2}\boldsymbol{X}^{1/2}$, $\boldsymbol{I}$, respectively, where $\boldsymbol{X}^{1/2}$ is the $n \times n$ positive symmetric matrix that satisfies $\boldsymbol{X}^{1/2}\boldsymbol{X}^{1/2} = \boldsymbol{X}$.

The latest version of SDPA employs the HRVW/KSH/M search direction among them. In general, the NT direction has stronger theoretical features, for example, which can attain the least iteration number of PD-IPM. However, the reason why SDPA choose the HRVW/KSH/M direction is mainly the small computation cost in each iteration, which often outweighs the increment the number of iterations when we solve various SDPs.

In the HRVW/KSH/M search direction, we reduce the above system into

$$\begin{cases} \sum_{j=1}^{m} ((\boldsymbol{X}\boldsymbol{A}_i\boldsymbol{Y}^{-1})) \bullet \boldsymbol{A}_j dz_j = p_i - A_i \bullet ((\boldsymbol{R} - \boldsymbol{X}\boldsymbol{D})\boldsymbol{Y}^{-1}), \\ d\boldsymbol{Y} = \boldsymbol{D} - \sum_{k=1}^{m} \boldsymbol{A}_k dz_k, \\ \widehat{d\boldsymbol{X}} = (\boldsymbol{R} - \boldsymbol{X}d\boldsymbol{Y})\boldsymbol{Y}^{-1}, \qquad d\boldsymbol{X} = (\widehat{d\boldsymbol{X}} + \widehat{d\boldsymbol{X}}^T)/2, \end{cases} \tag{2.6}$$

where

$$p_k = b_k - \boldsymbol{A}_k \bullet \boldsymbol{X}, \quad \boldsymbol{D} = \boldsymbol{C} - \Sigma_{k=1}^{m} \boldsymbol{A}_k z_k - \boldsymbol{Y}, \quad \boldsymbol{R} = \mu_c \boldsymbol{I} - \boldsymbol{X}\boldsymbol{Y}. \tag{2.7}$$

Once we decide $d\boldsymbol{z}$, the second and the third equality in the reduced system in (2.6) express a direct computation for $d\boldsymbol{X}$ and $d\boldsymbol{Y}$. Note that $\widehat{d\boldsymbol{X}}$ is a intermediate matrix for $d\boldsymbol{X}$ to be a symmetric matrix. Therefore, a substantial computation concentrates on solving the first linear equation to obtain $d\boldsymbol{z}$. We call the first equation *the Schur complement equation* and denote as

$$\boldsymbol{B}\ d\boldsymbol{z} = \boldsymbol{r}, \tag{2.8}$$

where

$$B_{ij} = (\boldsymbol{X}\boldsymbol{A}_i\boldsymbol{Y}^{-1}) \bullet \boldsymbol{A}_j, \quad r_i = p_i - A_i \bullet ((\boldsymbol{R} - \boldsymbol{X}\boldsymbol{D})\boldsymbol{Y}^{-1}). \tag{2.9}$$

It is well-known that the coefficient matrix $\boldsymbol{B}$, so-called the Schur complement matrix, is always positive definite throughout all the iterations in PD-IPM. Hence, we can apply either the Cholesky Factorization [direct method] or Conjugate Gradient Methods (CG Methods) [iterative method] to acquire $d\boldsymbol{z}$ from the Schur complement equation. Details of a comparison between the Cholesky Factorization and CG Methods will be focused on in section 3.4.2.

SDPA adopts the Cholesky Factorization, because the convergence rate of CG methods Methods deeply depends on a condition number of $\boldsymbol{B}$ that becomes seriously worse as $(\boldsymbol{X}, \boldsymbol{Y}, \boldsymbol{z})$ closes to an optimal solution, although many researches have investigated to improve the convergence rate [15, 60, 79]. Of course, even the Cholesky Factorization case, some numerical errors accumulated during the computation of PD-IPM affect the accuracy of the Cholesky Factorization. When the positive definiteness of $\boldsymbol{B}$ can not be maintained due to numerical errors, SDPA terminates the algorithm and prints out the current point $(\boldsymbol{X}, \boldsymbol{Y}, \boldsymbol{z})$ as final information.

The evaluation of elements of $\boldsymbol{B}$ is another remarkable component with respect to the Schur complement equation. SDPA aggressively exploits the sparsity of the input data, and we describe the exploitation in the next subsection.

2.3.2 Exploiting Data Sparsity

How much we can reduce the computational cost in the evaluation of the Schur complement matrix $\boldsymbol{B}$ at each iteration immediately affects total computation time of PD-IPM, because the evaluation often occupies most of the total computation cost. Meanwhile, the input data matrices $\boldsymbol{A}_k$ $(k = 1, 2, \ldots, m)$ arisen from real problems often have a considerable sparsity, that is, the number of non-zero elements is relatively smaller than n^2. Without any consideration for the sparsity, it must be difficult to solve SDPs from real problems in a practical time.

We incorporate the adequate method that successfully exploits sparsity for reducing the cost of $\boldsymbol{B}$ into SDPA [22]. Here, we take a glance at how the computation cost of $B_{ij} = (\boldsymbol{X}\boldsymbol{A}_i\boldsymbol{Y}^{-1}) \bullet \boldsymbol{A}_j$ changes between the case $\boldsymbol{A}_i, \boldsymbol{A}_j$ are fully dense and the case they are considerably sparse. Here, we should exclude the computation cost of the inversion of $\boldsymbol{Y}$ from the comparison, because $\boldsymbol{Y}^{-1}$ is identical for all elements in $\boldsymbol{B}$ throughout each iteration.

First, let both $\boldsymbol{A}_i$ and $\boldsymbol{A}_j$ have n^2 non-zero elements, that is, fully dense. Then we need $(2n^3 + n^2)$ multiplications for B_{ij}. ($(2n^3)$ for twice matrix multiplications, in addition n^2 for inner-product.)

On the other hand, when both $\boldsymbol{A}_i$ and $\boldsymbol{A}_j$ are sparse, we denote that the number of the non-zero elements of the matrices are $nz(i)$ and $nz(j)$, respectively and $[\boldsymbol{A}_i]_{pq}$ stands for a (p, q) element in $\boldsymbol{A}_i$. We reformulate the formula for B_{ij} to clarify an effect of the sparsity.

$$\begin{aligned} B_{ij} &= (\boldsymbol{X}\boldsymbol{A}_i\boldsymbol{Y}^{-1}) \bullet \boldsymbol{A}_j \\ &= \Sigma_{p=1}^{n}\Sigma_{q=1}^{n}[\boldsymbol{X}\boldsymbol{A}_i\boldsymbol{Y}^{-1}]_{pq}[\boldsymbol{A}_j]_{pq} \\ &= \Sigma_{p=1}^{n}\Sigma_{q=1}^{n}(\Sigma_{r=1}^{n}\Sigma_{s=1}^{n}X_{pr}[\boldsymbol{A}_i]_{rs}Y_{sq}^{-1})[\boldsymbol{A}_j]_{pq}. \end{aligned}$$

Based on the reformulation, the computation cost can be estimated as $(2nz(i) + 1) \times nz(j)$ multiplication.

A difference between $(2nz(i) + 1) \times nz(j)$ and $2n^3 + n^2$ becomes more clearly, when $nz(i)$ and $nz(j)$ are far from fully dense n^2. If all input data matrices $\boldsymbol{A}_k$ $(k = 1, 2, \ldots, m)$ are sparse, the effect of the exploitation of the sparsity would give us an immeasurable impact.

Certainly, the simple analysis for the computation cost of B_{ij} described the above is not enough, because we do not consider an overhead which stems from a data structure to handle the sparse matrices and a possibility that all elements in the ith row of $\boldsymbol{B}$ can share a result of the multiplication $\boldsymbol{X}\boldsymbol{A}_i\boldsymbol{Y}^{-1}$. In this section, we do not turn into the more details; See [22] for more details.

Supported by the exploitation of the sparsity, SDPA successfully solves various SDPs as reported in [85]. Furthermore, another merit we exploit sparsity is that we can remarkably reduce the memory space to store input data matrices. Therefore, we have a significant potential to deal with large SDPs.

2.3.3 Mehrotra Type Predictor-Corrector

In order to enhance a qualification of the search direction, SDPA employs Mehrotra Type Predictor-Corrector (MT-PC) PD-IPM [55]. In MT-PC PD-IPM, we may attain second order convergence toward an optimal solution while we need to obtain the two search direction at each iteration, the predictor search direction $(d\boldsymbol{X}_p, d\boldsymbol{Y}_p, d\boldsymbol{z}_p)$ and the corrector search direction $(d\boldsymbol{X}_c, d\boldsymbol{Y}_c, d\boldsymbol{z}_c)$. MT-PC PD-IPM is distinct from Mizuno-Todd-Ye Predictor-Corrector PD-IPM [56], because the update from the current point $(\boldsymbol{X}, \boldsymbol{Y}, \boldsymbol{z})$ to the next point in MT-PC is brought by only the corrector direction, and the predictor direction aids only for obtaining the the corrector direction.

We describe the implementation of the search direction in SDPA based on MT-PC PD-IPM, which replaces the Step 2 in the algorithmic framework of PD-IPM in section 2.2.

Algorithmic Framework for the Search Direction based on Mehrotra Type Predictor-Corrector

Step 0 (Parameter Set): Before starting the first iteration, we set $0 < \beta^* \leq \bar{\beta} < 1$.

Step 1 (Predictor Step): If $(\boldsymbol{X}, \boldsymbol{Y}, \boldsymbol{z})$ is $\bar{\epsilon}$-feasible solution, we set a predictor centering parameter β_p as 0, else as $\bar{\beta}$. Let $(d\boldsymbol{X}_p, d\boldsymbol{Y}_p, d\boldsymbol{z}_p)$ be the solution $(d\boldsymbol{X}, d\boldsymbol{Y}, d\boldsymbol{z})$ of the Schur complement equation replaced β in (2.6) by β_p.

Step 2 (Corrector Step): Let

$$\beta_t = \left(\frac{(\boldsymbol{X} + d\boldsymbol{X}_p) \bullet (\boldsymbol{Y} + d\boldsymbol{Y}_p)}{\boldsymbol{X} \bullet \boldsymbol{Y}} \right)^2. \tag{2.10}$$

If $(\boldsymbol{X}, \boldsymbol{Y}, \boldsymbol{z})$ is an $\bar{\epsilon}$-feasible solution, we set a corrector centering parameter β_c as $\max\{\beta^*, \beta_t\}$, else as $\max\{\bar{\beta}, \beta_t\}$. Let $(d\boldsymbol{X}_c, d\boldsymbol{Y}_c, d\boldsymbol{z}_c)$ be the solution $(d\boldsymbol{X}, d\boldsymbol{Y}, d\boldsymbol{z})$ of the Schur complement equation replaced β in (2.6) by β_c and $\boldsymbol{R}$ in (2.7) by $\mu_c\boldsymbol{I} - \boldsymbol{X}\boldsymbol{Y} - d\boldsymbol{X}_p d\boldsymbol{Y}_p$.

Step 3 (Search Direction Set): Set the search direction based on MT-PC PD-IPM $(d\boldsymbol{X}, d\boldsymbol{Y}, d\boldsymbol{z})$ by the corrector search direction $(d\boldsymbol{X}_c, d\boldsymbol{Y}_c, d\boldsymbol{z}_c)$.

In the Step 2 of the above algorithmic framework, we utilize the predictor search direction $(d\boldsymbol{X}_p, d\boldsymbol{Y}_p, d\boldsymbol{z}_p)$ to obtain the corrector search direction $(d\boldsymbol{X}_c, d\boldsymbol{Y}_c, d\boldsymbol{z}_c)$ as mentioned. The replacement of $\boldsymbol{R}$ enables us to numerically trace the central path with an almost second-order approximation. A point we have to note regarding the computation cost is that we need to solve the Schur complement equation twice for the predictor search direction and the corrector search direction, however, we do twice computations with only once evaluation of the Schur complement matrix in (2.8). The reason is the replacement in μ and $\boldsymbol{R}$ affects the right hand side of the Schur complement equation but not the elements of the Schur complement matrix $\boldsymbol{B}$. Thus, a principal efficiency attained by MT-PC PD-IPM is originated from the reuse of $\boldsymbol{B}$.

The above definition of the auxiliary centering parameter β_t (2.10) is slightly different from an original definitions as

$$\beta_t = \left(\frac{(\boldsymbol{X} + \alpha_p^{pre} d\boldsymbol{X}_p) \bullet (\boldsymbol{Y} + \alpha_d^{pre} d\boldsymbol{Y}_p)}{\boldsymbol{X} \bullet \boldsymbol{Y}} \right)^3,$$

where

$$\alpha_p^{pre} = \max\{\alpha \in [0,1] : \boldsymbol{X} + \alpha\ d\boldsymbol{X}_p \succeq \boldsymbol{O}\}, \quad \alpha_d^{pre} = \max\{\alpha \in [0,1] : \boldsymbol{Y} + \alpha\ d\boldsymbol{Y}_p \succeq \boldsymbol{O}\}.$$

In the above definition(2.10), we take away the computation of α_p^{pre} and α_d^{pre} and alter the exponential from 3 to 2. These changes were first incorporated into SDPA empirically. Nevertheless, we can verify the changes highten the numerical stability of SDPA through numerical experiments. The choice of an auxiliary centering parameter may have strong dependence on a step length that we will discuss next.

2.3.4 Step Length

To keep the positive definiteness of the variable matrices $\boldsymbol{X}$ and $\boldsymbol{Y}$ even after applying update in each iteration, we shrink the search direction $(d\boldsymbol{X}, d\boldsymbol{Y}, d\boldsymbol{z})$ by the step lengths α_p and α_d for $\mathcal{P}$ and $\mathcal{D}$, respectively. Shortly, the requirement for α_p and α_d is

$$\alpha_p = \max\{\alpha > 0 : \boldsymbol{X} + \alpha\ d\boldsymbol{X} \succeq \boldsymbol{O}\}, \quad \alpha_d = \max\{\alpha > 0 : \boldsymbol{Y} + \alpha\ d\boldsymbol{Y} \succeq \boldsymbol{O}\}.$$

The positive definiteness of $\boldsymbol{X}$ and $\boldsymbol{Y}$ ensures the step lengths must be positive number. Since the maximizations in both $\mathcal{P}$ and $\mathcal{D}$ are essentially identical computations, we focus on only for α_p in this section.

Fundamentally, we compute α_p through a rewrite of the maximization with the Cholesky Factorization $\boldsymbol{L}$ of $\boldsymbol{X}$, that is, $\boldsymbol{X} = \boldsymbol{L}\boldsymbol{L}^T$.

$$\begin{aligned}
\alpha_p &= \max\{\alpha > 0 : \boldsymbol{X} + \alpha\ d\boldsymbol{X} \succeq \boldsymbol{O}\} \\
&= \max\{\alpha > 0 : \boldsymbol{L}\boldsymbol{L}^T + \alpha\ d\boldsymbol{X} \succeq \boldsymbol{O}\} \\
&= \max\{\alpha > 0 : \boldsymbol{I} + \alpha \boldsymbol{L}^{-1} d\boldsymbol{X} \boldsymbol{L}^{-T} \succeq \boldsymbol{O}\} \\
&= -1/\lambda_{\min}(\boldsymbol{L}^{-1} d\boldsymbol{X} \boldsymbol{L}^{-T}),
\end{aligned}$$

where $\lambda_{\min}(\boldsymbol{A})$ is a minimum eigenvalue of the matrix $\boldsymbol{A}$. The last equality holds only if $\boldsymbol{L}^{-1} d\boldsymbol{X} \boldsymbol{L}^{-T}$ has negative eigenvalues. Otherwise $\boldsymbol{X} + \alpha \boldsymbol{dX}$ becomes a positive definite matrix for any $\alpha > 0$, hence we lose the reason for the shrinkage of the search direction. In the case, we assign a sufficiently large number for α_p such as 100.

If we obtain $\alpha_p > 1$ when the current point $(\boldsymbol{X}, \boldsymbol{Y}, \boldsymbol{z})$ is not $\bar{\epsilon}$-feasible solution, we restrict $\alpha_p = 1$ for the next point being a feasible solution. The restriction is justified the fact that we can always maintain the primal feasibility after once we reach the primal feasibility, except for the possibility of numerical errors.

The difficulty how we choose the step lengths comes also from a balance between a pursuit for the feasibility and a reduction of the relative gap. After we attain zero relative gap, it may be impossible to continue further iterations since the numerical stability of the Schur complement matrix may become extremely worse. To control the feasibility and the relative gap, we need to investigate an effect of the step length. Since the primal feasible error is

$$\max\{|p_k| : p_k = b_k - \boldsymbol{A}_k \bullet \boldsymbol{X}\ (k = 1, 2, \ldots, m)\}$$

and the primal component $d\boldsymbol{X}$ of the search direction satisfies $\boldsymbol{A}_k \bullet (\boldsymbol{X} + d\boldsymbol{X}) = b_k\ (k = 1, 2, \ldots, m)$, the primal feasible error on the next point with the step length α_p is

$$\begin{aligned}
& \text{[The next primal feasible error]} \\
= \ & \max\{|b_k - \boldsymbol{A}_k \bullet (\boldsymbol{X} + \alpha_p d\boldsymbol{X})| : (k = 1, 2, \ldots, m)\} \\
= \ & \max\{|b_k - \boldsymbol{A}_k \bullet \boldsymbol{X} - \alpha_p \boldsymbol{A}_k \bullet d\boldsymbol{X}| : (k = 1, 2, \ldots, m)\} \\
= \ & \max\{|b_k - \boldsymbol{A}_k \bullet \boldsymbol{X} - \alpha_p (b_k - \boldsymbol{A}_k \bullet \boldsymbol{X})| : (k = 1, 2, \ldots, m)\} \\
= \ & \max\{(1 - \alpha_p)|b_k - \boldsymbol{A}_k \bullet \boldsymbol{X}| : (k = 1, 2, \ldots, m)\} \\
= \ & (1 - \alpha_p) \max\{|p_k| : (k = 1, 2, \ldots, m)\} \\
= \ & (1 - \alpha_p) \times \text{ [The current primal feasible error]}
\end{aligned}$$

The relation that the next error is multiple of the current error ensures with factor of $(1 - \alpha_p)$ indicates that if we take $\alpha_p = 1$, the next error becomes zero. In addition, once we attain the feasibility, we can maintain the feasibility all the after iterations. In the same way, the dual feasibility error decreases in proportion to $(1 - \alpha_d)$. On the other hand, we estimate the relative gap by the formula $\boldsymbol{X} \bullet \boldsymbol{Y}$ since the difference between the primal and the dual objective value becomes $\boldsymbol{C} \bullet \boldsymbol{X} - \sum_{k=1}^{m} b_k z_k = \boldsymbol{X} \bullet \boldsymbol{Y}$ if $(\boldsymbol{X}, \boldsymbol{Y}, \boldsymbol{z})$ is a feasible solution. To make a reduction rate of the feasibility error smaller than an estimation rate of the relative gap $((\boldsymbol{X} + \alpha_p d\boldsymbol{X}) \bullet (\boldsymbol{Y} + \alpha_d d\boldsymbol{Y}))/(\boldsymbol{X} \bullet \boldsymbol{Y})$, we adopt the following scheme:

Algorithmic Framework for Adjustment of the Step Lengths

Step 0 (Initialization): Set initial step lengths by
$\alpha_p = \max\{\alpha > 0 : \boldsymbol{X} + \alpha\ d\boldsymbol{X} \succeq \boldsymbol{O}\}$, $\alpha_d = \max\{\alpha > 0 : \boldsymbol{Y} + \alpha\ d\boldsymbol{Y} \succeq \boldsymbol{O}\}$. If $(\boldsymbol{X}, \boldsymbol{Y}, \boldsymbol{z})$ is not $\bar{\epsilon}$-feasible solution, we modify $\alpha_p = \min\{\alpha_p, 1\}$ and $\alpha_d = \min\{\alpha_d, 1\}$. Choose a shrinking parameter $0 < \zeta < 1$

Step 1 (Termination check): If

$$\max\{1 - \alpha_p, 1 - \alpha_d\} \leq ((\boldsymbol{X} + \alpha_p d\boldsymbol{X}) \bullet (\boldsymbol{Y} + \alpha_d d\boldsymbol{Y}))/(\boldsymbol{X} \bullet \boldsymbol{Y})$$

is satisfied, we stop.

Step 2 (Update lengths): Let $\alpha_p \rightarrow \zeta\alpha_p$ and $\alpha_d \rightarrow \zeta\alpha_d$. Goto Step 1.

In SDPA, the actual implementation for the minimum eigenvalue of $\boldsymbol{L}^{-1} d\boldsymbol{X} \boldsymbol{L}^{-T}$ is the Lanczos method developed by [77]. A standard eigenvalue decomposition needs $O(n^3)$ operation, while the Lanczos method provides a cheaper computation with an enough approximation of the minimum eigenvalue.

2.3.5 Block Diagonal Structure

For many SDPs arisen from real problems, the coefficient matrix $\boldsymbol{C}$ and the input data matrices $\boldsymbol{A}_k$ $(k = 1, 2, \ldots, m)$ often have same non-zero structures other than simple sparsity. SDPA was designed to deal with a structure called the block diagonal structure.

For example, if a symmetric matrix $\boldsymbol{X}_{\rm ex}$ has a non-zero structure such that

$$\boldsymbol{X}_{\rm ex} = \left(\begin{array}{ccc|cc|cc} 1 & 2 & 3 & 0 & 0 & 0 & 0 \\ 2 & 4 & 5 & 0 & 0 & 0 & 0 \\ 3 & 3 & 6 & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 7 & 8 & 0 & 0 \\ 0 & 0 & 0 & 8 & 9 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & 0 & 10 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 11 \end{array}\right),$$

we had better treat it as three small matrices (a 3×3 matrix , a 2×2 matrix and a diagonal 2×2 matrix) than one big 7×7 matrix.

In general, the block diagonal structure for a symmetric matrix can be expressed by nBLOCK and bLOCKsTRUCT in SDPA. If a symmetric matrix $\boldsymbol{X}$ has h block diagonal matrices $\boldsymbol{X}_1 \in \mathbb{S}^{n_1}, \boldsymbol{X}_2 \in \mathbb{S}^{n_2}, \ldots \boldsymbol{X}_h \in \mathbb{S}^{n_h}$, that is,

$$\boldsymbol{X} = \left(\begin{array}{cccc} \boldsymbol{X}_1 & \boldsymbol{O} & \cdots & \boldsymbol{O} \\ \boldsymbol{O} & \boldsymbol{X}_2 & \cdots & \boldsymbol{O} \\ \vdots & \vdots & \ddots & \vdots \\ \boldsymbol{O} & \boldsymbol{O} & \cdots & \boldsymbol{X}_h \end{array}\right),$$

we define nBLOCK and bLOCKsTRUCT as

$$\begin{aligned} \text{nBLOCK} &= h, \\ \text{bLOCKsTRUCT} &= (\widehat{n_1}, \widehat{n_2}, \ldots, \widehat{n_h}), \\ \widehat{n_\ell} &= \begin{cases} n_\ell & \text{if } \boldsymbol{X}_\ell \text{ is a symmetric matrix,} \\ -n_\ell & \text{if } \boldsymbol{X}_\ell \text{ is a diagonal matrix.} \end{cases} \end{aligned}$$

The block diagonal structure for the above $\boldsymbol{X}_{\rm ex}$ can be expressed as

$$\text{nBLOCK} = 3, \quad \text{bLOCKsTRUCT} = (3, 2, -2).$$

A main merit of the adaptation of the block diagonal structure is a significant reduction of the computation cost. For instance, let $\boldsymbol{A}$ and $\boldsymbol{X}$ be two matrices sharing an identical block diagonal structure.

$$\boldsymbol{A} = \left(\begin{array}{cccc} \boldsymbol{A}_1 & \boldsymbol{O} & \cdots & \boldsymbol{O} \\ \boldsymbol{O} & \boldsymbol{A}_2 & \cdots & \boldsymbol{O} \\ \vdots & \vdots & \ddots & \vdots \\ \boldsymbol{O} & \boldsymbol{O} & \cdots & \boldsymbol{A}_h \end{array}\right), \quad \boldsymbol{X} = \left(\begin{array}{cccc} \boldsymbol{X}_1 & \boldsymbol{O} & \cdots & \boldsymbol{O} \\ \boldsymbol{O} & \boldsymbol{X}_2 & \cdots & \boldsymbol{O} \\ \vdots & \vdots & \ddots & \vdots \\ \boldsymbol{O} & \boldsymbol{O} & \cdots & \boldsymbol{X}_h \end{array}\right).$$

Then we can reduce multiplication and inner-product between $\boldsymbol{A}$ and $\boldsymbol{X}$ into

$$\begin{aligned} \boldsymbol{AX} &= \begin{pmatrix} \boldsymbol{A}_1\boldsymbol{X}_1 & \boldsymbol{O} & \cdots & \boldsymbol{O} \\ \boldsymbol{O} & \boldsymbol{A}_2\boldsymbol{X}_2 & \cdots & \boldsymbol{O} \\ \vdots & \vdots & \ddots & \vdots \\ \boldsymbol{O} & \boldsymbol{O} & \cdots & \boldsymbol{A}_h\boldsymbol{X}_h \end{pmatrix}, \\ \boldsymbol{A} \bullet \boldsymbol{X} &= \boldsymbol{A}_1 \bullet \boldsymbol{X}_1 + \boldsymbol{A}_2 \bullet \boldsymbol{X}_2 + \cdots + \boldsymbol{A}_h \bullet \boldsymbol{X}_h. \end{aligned}$$

Furthermore, we can detect a positive semidefiniteness of $\boldsymbol{X}$ by smaller sub-matrices.

$$\boldsymbol{X} \succeq \boldsymbol{O} \Leftrightarrow \boldsymbol{X}_1 \succeq \boldsymbol{O}, \boldsymbol{X}_2 \succeq \boldsymbol{O}, \ldots, \boldsymbol{X}_h \succeq \boldsymbol{O}.$$

As a result we do with only sub-matrices computations which require cheaper cost than one large matrix computations.

A combination of the exploitation of the sparsity and the block diagonal structure described here cut down the computation cost of SDPA. A significant effect can be observed when we apply SDPA to SDPs arisen from real problems such as Control Problem and Theta Function Problem described in section 3.2.1.

2.4 Other Methods and Software for Semidefinite Programming

PD-IPM described in this section have been a strong position in the field of SDP since its first proposal [42]. Until now, PD-IPM have been implemented in some computer software for SDPs. SDPA is a representative one of them, and the other representative software of PD-IPM are SDPT3[74], SeDuMi[67] and CSDP[10].

SDPT3, a MATLAB software, achieves an outstanding performance that is comparable to SDPA. A main algorithmic framework in SDPT3 is almost the same as SDPA, for instance, the HRVW/KSH/M direction, however, a point SDPT3 is superior to SDPA is that SDPT3 directly solves Second-Order Cone Programming, namely deals with the Cartesian product of SDP, Second-Order Cone Programming and Linear Programming.

SeDuMi is also a MATLAB software owing to the advantage of PD-IPM. SeDuMi utilizes Self-Dual embedding and the NT search direction instead of the HRVW/KSH/M direction. With a schema which holds the variable matrices in their Cholesky Factorization form, SeDuMi attains excellent stability far beyond other software.

CSDP is written in C language, while SDPA is in C++ language. Thus, CSDP can ignore overheads derived from C++ or MATLAB. Since CSDP and SDPA have their callable library and MATLAB interface, we call their implementation from many program, including MATLAB [21].

PD-IPM have been verified very strong from viewpoints of both theory and practice by the above software, however, they are not perfect for all SDPs. Suppose that the coefficient matrix $\boldsymbol{C}$ and all the input data matrices $\boldsymbol{A}_k$ $(k = 1, 2, \ldots, m)$ have the same sparsity, that is, the same non-zero structure. The dual variable matrix $\boldsymbol{Y}$ inherits the sparsity since $\boldsymbol{Y} = \boldsymbol{C} - \sum_{k=1}^{m} \boldsymbol{A}_k$. On the other hand, the primal variable matrix $\boldsymbol{X}$ becomes usually full dense matrix even when the sparsity is remarkable. In order to overcome the fully density in the primal matrix, two methods have been proposed in IPM framework. Nakata, *et al.* proposed a new PD-IPM by means of a combination with *the Completion Method* [24, 60]. The proposition is also incorporated into parallel implementation as described in chapter 4. The other method is *Dual IPM* (D-IPM) introduced by Benson *et al.* [6]. In D-IPM framework, they consider in only the dual space $\mathcal{D}$ without the primal space $\mathcal{P}$. Therefore, in general, D-IPM only attains lower stability than PD-IPM based on the dual theorem. They implement D-IPM into software package, DSDP[5]. Removing the memory storage for the fully dense primal variable matrix $\boldsymbol{X}$ results in a successful reduction of computation memory space. DSDP shows its strong power for SDP relaxations arisen from graph theory at which we may not require high accuracy. The details of D-IPM will be discussed in section 3.6.1.

A main difficulty of IPM is that the most computation costs of either PD-IPM or D-IPM are usually occupied by the Schur complement equation. To overcome the difficulty, many other researches have been developed [78].

Among other methods, an approach that deals with SDP from the viewpoint of nonlinear optimal programming (NLP) are remarkable. Krishnan, *et al.* [46, 47] focus a fact that positive semidefiniteness of a symmetric matrix can be expressed semi-infinite linear constraints and propose Linear Programming

approaches to SDP with cutting plane methods. Lagrangian method, a common method in the field of NLP, is also utilized [25]. Kocvara, *et al.* apply penalty methods to augmented Lagrangian method. PENNON [39, 40] implemented by them searches an optimal solution by alternating a movement of a variable matrix and an update of Lagrangian multipliers. Spectral Bundle method proposed by Helmberg *et al.* [34] is also implemented into software, SBMethod [33], based on the Lagrangian method. They convert an SDP to an eigenvalue optimization problem. Furthermore, Burer, *et al.* [13, 14] introduce a first-order log-barrier method utilizing gradient information, a fundamental concept in NLP.

More and more other methods and software can be found from Helmberg's WWW site [32]. These uncountable researches prove the importance and the attraction of SDPs which we are addressing in this thesis.

Chapter 3

Parallel Implementation of Primal-Dual Interior-Point Methods

In this chapter, we propose parallel software SDPARA (SemiDefinite Programming Algorithm paRAllel version). We start the chapter from some background and historical description of parallel computation in section 3.1. Then we move on section 3.2, a central part of this chapter. In section 3.2, after pointing out bottlenecks of PD-IPM on a single processor, we describe how the bottlenecks have been replaced by their parallel implementation of SDPARA. Section 3.3 is devoted for the details of parallel computation environments. In section 3.4, we investigate effects of the parallel implementation by preliminary numerical experiments. Then, full-scale numerical experiments on various SDPs, in particular, on SDPs arisen from quantum chemistry in section 3.5 and a comparison with another parallel SDP solver in section 3.6 will show the high performance of SDPARA. Finally, we provide a theoretical validity of parallel implementation of SDPARA in section 3.7.

3.1 Background of Parallel Computation

As stated in section 2.1, SDPs have many applications from real issues. However, more accurate descriptions or more input datas for the models make extremely large SDPs. In particular, SDPs arisen from quantum chemistry can have arbitrary number of constraints, if we prepare a quite number of base wave functions as described in section 3.5.2.

Such large scale SDPs are impossible to solve on a single processor mainly due to two reasons. The first reason is that memory space we can attach on a single processor is limited. The other reason is that computation time required for large scale SDPs may not be practical even if we have enough memory to store them. The questions we have to ask in this thesis are that how to store large SDPs, and that how to solve in a short time.

On the other hand, over these few years, a considerable number of researches have been devoted on parallel computation. We can verify noticeable accomplishments in the field of parallel computation through Top 500 [80] list which reports the 500 strongest computation resources in the world. Historically speaking, an event which made quite an impact on people was that a parallel computer 'Deep Blue' beated the world chess champion in 1997. After that, parallel computation has been brought to public attention as computation resources. What we are interested in here is whether we can solve the large scale SDPs in short time if we incorporate parallel computation into PD-IPM.

Before we turn to investigate the possibility of parallel computation for SDPs, let us draw our attention to parallel computation itself.

The history of mechanical calculators can date back to automatically additions and subtractions by Pascal in 1649 and multiplications and divisions by Leibniz in 1674. After almost 300 years from Pascal, the first electronic calculator 'ABC (Atanasoff-Berry Computer)' was developed by Atanasoff and Berry in 1942. In 1946, ENIAC (Electronic Numerical Integrator And Calculator) by Mauchly and Eckart proved that the power of electricity enables us to compute effectively. From mathematical views, important contributions were achieved by Turing and von Neumann. Turing [81] established a fundamental concept of *computability*,

what we can compute or can not. A lot of number of mechanical devices for computation proposed by von Neumann in 1945 affect almost all the present computers.

The first super computer to persist high performance computing was CDC6600 in 1964. In succession, the Cray-I regarded as a synonym for super computer was built in 1976. CDC6600 and Cray-I were parallel computers which we can categorize into vector parallel computer. They were designed to achieve high performance, in particular, for vector computations arisen from technological fields; they had specialized processors for vector computations. Since the outstanding micro processors evolutions in 1980s, multiple attempts were made for parallel computation based on commercially-designated micro processors. Subsequently, in 1990s, the super parallel computer with hundreds of thousands of processors emerged.

At the present time, we have two key technological factors in parallel computation, *Cluster* and *Grid Computing*. Cluster technology has aimed for a realization of multiple computation resources, such as personal computers, work stations and super computers, lumped together in a big virtual parallel computation resource by connections through the network. On the other hand, the term Grid is derived from an electronic grid. When we use an electronic power from a plug, we usually do not worry about where the electricity is produced. In the same way, Grid technology explores a possibility whether we can use computation resources without any attention where the computation is done. Cluster and Grid computing technologies will provide us more easiness ways for parallel computation and massive computation resource in near future.

In the above paragraphs, we have reviewed the short history. In turn, we take look at categorizations of parallel computation. In general, we have multiple processors and multiple data on parallel computation. A relation between data and operators on processors gives us a categorization called *SIMD (Single Instruction stream and Multiple Data stream)* and *MIMD(Multiple Instruction stream and Multiple Data stream)*. For example, suppose that we have two processors, P1 and P2. If we compute $a_1 = b_1 + c_1$ on P1 and $a_2 = b_2 + c_2$ on P2 simultaneously, we call SIMD since P1 and P2 apply the same operation (namely addition). On the other hand, when $a_1 = b_1 + c_1$ on P1 and $a_2 = b_2 \times c_2$ on P2, we call MIMD, because of the difference of operators (namely addition and multiplication). The other categorization is from memory space which corresponds to processors. On *SMP* (*Symmetric MultiProcessors*) system, all processors share one large memory space. Conversely, on *distributed memory* system, each processor posses their own memory space. We usually require no communication over memory space on SMP system, while distributed memory easily enables us to retain large memory space.

Another perspective we are concerned about parallel computation is performance measures. Common measures are *scalability* and *load-balance*. It may be natural we expect that we can solve problems N times faster as compared to a single processor when we use N processors. The scalability is a measure how much faster parallel computations solve problems than a single processor. If N processors are used, a desirable scalability is N, but it is often impossible due to communication between multiple processors and components which we should not replace with parallel computation from a single computation because of algorithmic frameworks. Generally speaking, a reduction of communication generates higher scalability. On the other hand, after counting the number of operation on each processors through parallel computation, we can evaluate the performance by load-balance, which is the ratio of the highest operation counts and the lowest counts in N processors. The load-balance close to 1 indicates that we distribute computations on the processors averagely. Clearly, the load-balance and the scalability has strong correlation. Approaches to better load-balance are surely results in higher scalability, since there may be no redundant computation time on each processor. What we have to note here is, however, that grain sizes is one subject. The grain size is the unit size into which we divide the original problem for the distribution over multiple processors. Small grain sizes are helpful for better load-balance and give worse scalability because of incremental communication. Therefore, better load-balance and higher scalability are sometimes difficult for us to acquire simultaneously.

So far, a lot of applications on parallel computation have been developed through significant number of supports over the past years. Among them, crucial researches are examined in the field of data mining and image processing. In addition, a drastically growth of parallel computation is continued on pattern matting for DNA analysis. In the field of numerical analysis, finite element methods arisen from differential equation has relatively long history on parallel computation.

We are convinced that high performances achieved by parallel computation enable us to solve extremely large-scale SDPs in a short time which we could not attain so far. Before discussing detail implementation to solve SDPs on parallel computation, we should inquire into bottlenecks when we solve SDPs on a single

processor, since the bottlenecks must be targets to be replaced with their parallel implementation.

3.2 Parallelization

3.2.1 Bottlenecks of Primal-Dual Interior-Point Methods on a Single Processor

To gain a sufficient effect on parallel computation, there are somethings to keep our eyes on. Clearly, a reduction of computation time offered by parallel computation is considerably attractive to solve large SDPs, however, we also need to consider overheads of communication between multiple processors. It is said in general that only 10 percent of source codes occupies 90 percent of computation time. A significant task we are facing is that we have to decide which components of PD-IPM we apply parallel computation. Here, let us examine to figure out bottlenecks of PD-IPM on various SDPs on a single processor, then we consider replacements for the bottlenecks with their parallel implementation.

From combinations of theoretical estimations for computation cost and empirical results on some SDPs, we categorize the four components of PD-IPM which may be bottlenecks mainly from the algorithmic framework of PD-IPM (section 2.1) and the search direction (section 2.3.1).

1. ELEMENTS
   The Evaluations of elements of the Schur complement matrix $\boldsymbol{B}$ with formula $B_{ij} = \boldsymbol{A}_i \bullet (\boldsymbol{X}\boldsymbol{A}_j\boldsymbol{Y}^{-1})$ in (2.9).

2. CHOLESKY
   The Cholesky Factorization of the Schur complement matrix $\boldsymbol{B}$.

3. PMATRIX
   The computation for $d\boldsymbol{X}$, the primal variable matrix in the search direction in the framework of the HRVW/KSH/M direction based on the last formula (2.6).

4. DENSE
   The other portions required by computation for primal and dual dense variable matrices, $\boldsymbol{X}$ and $\boldsymbol{Y}$. For example, the Lanczos method for the step length (section 2.3.4), multiplications for the right hand side in the Schur complement equation (2.9) and the inversion of $\boldsymbol{Y}$.

We summarize estimations of computation cost for each component in Table 3.1, regarding the number of equality constraints m and the matrix size n for $\boldsymbol{X}$ and $\boldsymbol{Y}$. What we want to emphasize is that the

Table 3.1: Estimations of computation cost for each component

ELEMENTS	$\mathcal{O}(mn^3 + m^2n^2)$
CHOLESKY	$\mathcal{O}(m^3)$
PMATRIX	$\mathcal{O}(n^3)$
DENSE	$\mathcal{O}(n^3)$

most computation time at each iteration of PD-IPM in general SDPs is occupied by the evaluations of the elements of the Schur complement matrix $\boldsymbol{B}$, ELEMENTS, and its Cholesky Factorization, CHOLESKY. Although the former computation heavily depends on how sparse the input data matrices $\boldsymbol{A}_1, \boldsymbol{A}_2, \ldots, \boldsymbol{A}_m$ are, it often occupies the largest portion of the computational time, even though we exploit the sparsity [22]. (The estimations in the above table were made on an assumption that in the case all input data matrices $\boldsymbol{A}_1, \boldsymbol{A}_2, \ldots, \boldsymbol{A}_m$ are fully dense.) In addition, the Schur complement matrix $\boldsymbol{B}$ usually becomes fully dense no matter how sparse input data matrices are. Since the Cholesky Factorization of $\boldsymbol{B}$ needs $m^3/3$ multiplication, it sometimes takes longer computation time than the evaluation of the elements of $\boldsymbol{B}$ especially when the input data matrices are sparse and/or the number m of the equality constraints of $\mathcal{P}$ is much larger than the size n of the variable matrices. The reason why we divide $\mathcal{O}(n^3)$ computation into two portions, PMATRIX and DENSE, may not be clear here, however, this division will be necessary in the next chapter where we incorporate the completion method with PD-IPM.

To confirm the above fact from numerical results, we pick up three characteristic problems from SDPLIB [11], 'control11', 'theta6' and 'maxG51' arisen from Control Problem, Theta Function Problem, Max Cut Problem, respectively. Shortly, let us examine how these problems are formulated into SDPs. See [12, 35, 72] for details of the formulations and their backgrounds.

SDPs arisen from Control Theory

Let $\boldsymbol{A}$ be a symmetric matrix. Suppose that time-dependent vector $\boldsymbol{u}(t)$ is controlled under the following differential equation.

$$\frac{d\boldsymbol{u}(t)}{dt} = \boldsymbol{A}\boldsymbol{u}(t). \tag{3.1}$$

It is known that a stability condition $||\boldsymbol{u}(t)|| \to 0$ $(t \to \infty)$ holds provided that all eigenvalues of $\boldsymbol{A}$ are negative. Furthermore, we assume $\boldsymbol{A}$ is a linear combination of symmetric matrices $\boldsymbol{A}_1, \boldsymbol{A}_2, \ldots, \boldsymbol{A}_m$, that is, $\boldsymbol{A} = \sum_{k=1}^{m} x_k \boldsymbol{A}_k$. Then we can detect whether the stability in (3.1) holds or not by an optimal value of the SDP.

$$\text{minimize} \quad \lambda \quad \text{subject to} \quad \boldsymbol{X} = -\sum_{k=1}^{m} x_k \boldsymbol{A}_k + \lambda \boldsymbol{I}, \quad \boldsymbol{X} \succeq \boldsymbol{O}.$$

Since λ is an upper bound of the eigenvalues of $\boldsymbol{A}$, we conclude that if the optimal value λ in the above SDP is negative, there exists $\boldsymbol{x}$ which stabilizes the differential equation, *i.e.*, all the eigenvalues of $\boldsymbol{A}$ are negative. Conversely, $||\boldsymbol{u}(t)||$ can not be bounded if λ is positive.

More general cases are discussed in [12]. In the paper, we do not enforce the input matrices $\boldsymbol{A}_1, \boldsymbol{A}_2, \ldots, \boldsymbol{A}_m$ be symmetric matrices. To consider the stability of the more general differential equation, we transform a stability condition derived from Lyapnov function into the following SDP in which we have a positive semidefinite matrix $\boldsymbol{Y}$ to be determined.

$$\text{minimize} \quad \lambda \quad \text{subject to} \quad \boldsymbol{A}_k^T \boldsymbol{Y} + \boldsymbol{Y}\boldsymbol{A}_k \preceq \boldsymbol{O} \ (k = 1, 2, \ldots, m), \quad \lambda \boldsymbol{I} \succeq \boldsymbol{Y} \succeq \boldsymbol{O}. \tag{3.2}$$

'control11' in SDPLIB is an SDP formulated in this general way. We often say Control Problem instead of 'SDPs arisen from Control Theory.'

Theta Function Problem

Let $G = (V, E)$ be an undirected graph with a vertex set $V = \{1, 2, \ldots, n\}$ and an edge set $E \subset \{(i, j) : i, j \in V, i < j\}$. We define *the theta function* $\theta(G)$ on G as an optimal value of the following SDP.

$$\begin{array}{rlll} \theta(G) = & \text{maximize} & \boldsymbol{e}\boldsymbol{e}^T \bullet \boldsymbol{X} & \\ & \text{subject to} & \boldsymbol{I} \bullet \boldsymbol{X} = 1, & \\ & & \boldsymbol{E}_{ij} \bullet \boldsymbol{X} = 0, & (i, j) \notin E \\ & & \boldsymbol{X} \succeq \boldsymbol{O}, & \end{array}$$

where $\boldsymbol{e}$ is a vector in $\mathbb{R}^n$ whose *all* components equal one and $\boldsymbol{E}_{ij}$ is a matrix in $\mathbb{S}^n$ whose *only* (i, j) and (j, i) components equal one and other components are zero. An important property is that $\theta(G)$ is an upper bound of the maximum size of a stable set and a lower bound of the coloring number. However, computations for the two numbers, the maximum size of a stable set and the coloring number, are NP-complete problem. Since $\theta(G)$ is an optimal value of the SDP which can be solved in polynomial time by PD-IPM, $\theta(G)$ provides an efficient approximation value for the two complicated numbers.

SDP relaxation for Max Cut Problem

Again, we start from an undirected graph $G = (V, E)$ with a vertex set $V = \{1, 2, \ldots, n\}$ and an edge set $E \subset \{(i, j) : i, j \in V\}$. Additionally, we attach a weight value $c(i, j) = c(j, i)$ on each edge in E. In Max Cut Problem, we want to separate the edge set V into two disjoint sets V_1 and V_2 with maximizing weight across the two sets. More precisely, Max Cut Problem is described in the following mathematical definition.

$$\max\{\Sigma_{i \in V_1, j \in V_2} c(i, j) : V_1 \cup V_2 = V, V_1 \cap V_2 = \phi\}$$

We introduce a variable x_i on each vertex to indicate the vertex is separated into either $V_1 = \{i \in V : x_i = 1\}$ or $V_2 = \{i \in V : x_i = -1\}$. Let C be a matrix in $\mathbb{S}^n$ whose elements are weight values, $C_{ij} = c(i, j) = c(j, i)$. Then we reformulate Max Cut Problem into an integer quadratic optimization problem.

$$\max\{\Sigma_{i=1}^{n} \Sigma_{j=1}^{n} c_{ij}(1 - x_i x_j) : x_i = \pm 1 \ (i = 1, 2, \ldots, n)\}$$

Regarding multiplication $x_i x_j$ as x_{ij}, (i,j) element of a symmetric matrix $\boldsymbol{X}$, we converts the constraint $x_i = \pm 1$ $(i = 1, 2, \ldots, n)$ into the three constraints $\boldsymbol{X}_{ii} = 1, \boldsymbol{X} \succeq \boldsymbol{O}$ and $rank(\boldsymbol{X}) = 1$. Especially, the rank condition for the matrix makes Max Cut Problem difficult to be solved. Max Cut Problem as well as the maximum size of a stable set are NP-complete Problems.

However, a disregard for the rank condition generates an SDP relaxation for Max Cut Problem.

$$\max\{\widehat{\boldsymbol{C}} \bullet \boldsymbol{X} : \boldsymbol{E}_{ii} \bullet \boldsymbol{X} = 1 \ (i = 1, 2, \ldots, n), \boldsymbol{X} \succeq \boldsymbol{O}\},$$

where $\widehat{\boldsymbol{C}}_{ij} = \delta_{ij} \sum_{k=1}^{n} \boldsymbol{C}_{ik} - \boldsymbol{C}_{ij}$ and $\boldsymbol{E}_{ii}$ is a matrix whose only ith diagonal elements are equal to one and other elements are zero. Consequently, a relation between the SDP relaxation and its original Max Cut Problem is similar as between Theta function and coloring number, since the SDP relaxation gives an efficient lower bound of optimal value of Max Cut Problem.

Goemans and William [27, 28] first introduced SDP relaxations for Max Cut Problem, and then their effectiveness and efficiency have been deeply investigated. We often use 'Max Cut Problem' to indicate SDP relaxation for Max Cut Problem, because we focus on only SDP in this thesis and original Max Cut Problem (NP-complete Problem) is beyond our interest.

Now, we return to our main concern, bottlenecks of PD-IPM executed by a single processor on 'control11', 'theta6' and 'maxG51' whose sizes are shown in Table 3.2. m is the number of equality constraints of $\mathcal{P}$, nBLOCK and bLOCKsTRUCT define the structure of $\boldsymbol{X}$ and $\boldsymbol{Y}$. Table 3.3 shows how much proportion of

Table 3.2: SDPs from SDPLIB

name	m	nBLOCK	bLOCKsTRUCT
control11	1596	2	(110,55)
theta6	4375	1	(300)
thetaG51	6910	1	(1001)

the total computation time in second is occupied by the above four categorized components of PD-IPM and the other parts denoted by "Others". "Total" denotes the total computation time in second. We executed SDPA 6.0 on a single Pentium 4 (2.2 GHz) processor and 1GB memory under Linux 2.4.18.

Table 3.3: Performance of SDPA 6.0 for control11, theta6 and maxG51 on a single processor

	control11		theta6		maxG51	
	time	ratio	time	ratio	time	ratio
ELEMENTS	463.2	91.6%	78.3	26.1%	1.5	1.0 %
CHOLESKY	31.7	6.2%	209.8	70.1%	3.0	2.1 %
PMATRIX	1.8	0.3%	1.8	0.6%	47.3	33.7%
DENSE	1.0	0.2%	4.1	1.3%	86.5	61.7%
Others	7.2	1.4%	5.13	1.7%	1.8	1.3%
Total	505.2	100.0%	292.3	100.0%	140.2	100.0 %

Table 3.3 explicitly indicates that about 90% of computation time was spent for ELEMENTS in control11 and about 70% for CHOLESKY in theta6, respectively. In both cases, the computation time spent in the other portions of SDPA is less than 5%. In solving many SDPs by the SDPA, ELEMENTS and CHOLESKY occupy most of the computation time. Therefore, applying parallel computation to these two components ELEMENTS and CHOLESKY is quite reasonable to shorten the total computation time.

This is not true, however, in the case maxG51. Max Cut Problem described above has two significant structures. First, the size n of the variable matrices $\boldsymbol{X}$ and $\boldsymbol{Y}$ is as large as the number m of the equality constraints in $\mathcal{P}$. The other is that the input data matrices $\boldsymbol{A}_1, \boldsymbol{A}_2, \ldots, \boldsymbol{A}_m$ have only one element in their diagonal positions. Due to the two structures, parallel implementation for ELEMENTS and CHOLESKY cannot carry out its parallel computation effectively.

In this section, we will focus on replacing ELEMENTS and CHOLESKY, the components regarding the Schur complement $\boldsymbol{B}$ in (2.8) to obtain the search direction, with their parallel implementation. After the numerical results of the parallelization, we discuss the component of PMATRIX with advantage of the Completion Method in the next chapter to overcome the difficulties in Max Cut Problem.

3.2.2 Parallel Evaluation of Elements of the Schur Complement Matrix

A purpose of this section is to explain details of parallel implementation for ELEMENTS component of PD-IPM. In the subsequent section, we describe for CHOLESKY component. We assume the number of available processors is N and attach the numbers from 1 through N to each processor. We start from the most simple case in which we do not adopt exploitation of sparsity in the input data matrices proposed in [22] and the diagonal block structure described in section 2.3.5.

Since each element of the Schur complement matrix $\boldsymbol{B}$ is of the form $B_{ij} = (\boldsymbol{X}\boldsymbol{A}_i\boldsymbol{Y}^{-1}) \bullet \boldsymbol{A}_j$, all elements B_{ij} $(j = 1, 2, \ldots, m)$ in the ith row share a common matrix $(\boldsymbol{X}\boldsymbol{A}_i\boldsymbol{Y}^{-1})$. If two different processors shared the computation of those elements, they would need to compute the entire matrix $(\boldsymbol{X}\boldsymbol{A}_i\boldsymbol{Y}^{-1})$ in duplicate or they would need to transfer partial elements of the matrix to each other. Hence, to avoid duplicate computation of $(\boldsymbol{X}\boldsymbol{A}_i\boldsymbol{Y}^{-1})$ and communication time between different processors, it is reasonable to require a single processor to compute the entire matrix $(\boldsymbol{X}\boldsymbol{A}_i\boldsymbol{Y}^{-1})$ and all elements B_{ij} $(j = 1, 2, \ldots, m)$ in the ith row. On the other hand, if $k \neq i$ then we can compute the matrix $(\boldsymbol{X}\boldsymbol{A}_k\boldsymbol{Y}^{-1})$ and the elements B_{kj} $(j = 1, 2, \ldots, m)$ independently from them. Thus we assign the computation of the elements in each row of $\boldsymbol{B}$ to a single processor. To be precise, let $\mathcal{B} = \{1, 2, \ldots, m\}$ be the row indices of $\boldsymbol{B}$ and the index sets $\mathcal{P}_u \subset \mathcal{B}$ for the uth processor. In other word, we assign the computation for all the elements in the ith row of $\boldsymbol{B}$ $(i \in \mathcal{P}_u)$ to the uth processor. Generally speaking, we divide $\mathcal{B}$ into disjoint sets $\mathcal{P}_u$ $(u = 1, 2, \ldots, N)$, that is,

$$\mathcal{B} = \cup_{u=1}^{N} \mathcal{P}_u, \quad \mathcal{P}_u \cap \mathcal{P}_v = \phi \ (u \neq v).$$

From the viewpoints for better load-balance, $\mathcal{B}$ should be equalized into the sets $\mathcal{P}_u$ $(u = 1, 2, \ldots, N)$. Shortly, we may attain higher scalability when we choose the cardinality of each set $\mathcal{P}_u$ is as close as to m/N. A concept we adopt to level the load-balance is that we divide $\mathcal{B}$ into some belts and assign each processor to each belt in a sequential order. Concretely, we first define the wth belt $\mathcal{B}_w \subset \mathcal{B}$ with a constant size sb in the Schur complement matrix $\boldsymbol{B}$.

$$\mathcal{B}_w = \{i : (w-1) \times sb + 1 \leq i \leq \min\{m, w \times sb\}\}$$

Surely, we hold $\lceil m/sb \rceil$ belts in $\boldsymbol{B}$, where $\lceil t \rceil$ is a ceiling number of t. Only the upper bound of the last belt (the $\lceil m/sb \rceil$th belt) may be the cardinality m of $\boldsymbol{B}$. Then, we compose $\mathcal{P}_u$ $(u = 1, 2, \ldots, N)$ of some belts in rotation,

$$\mathcal{P}_u = \cup\{i : i \in \mathcal{B}_w, \ w\%N = u\}, \tag{3.3}$$

where $a\%b$ denotes the remainder of a divided by b if it is non-zero or b if it is zero. We call this division of $\mathcal{B}$ *row-wise distribution*.

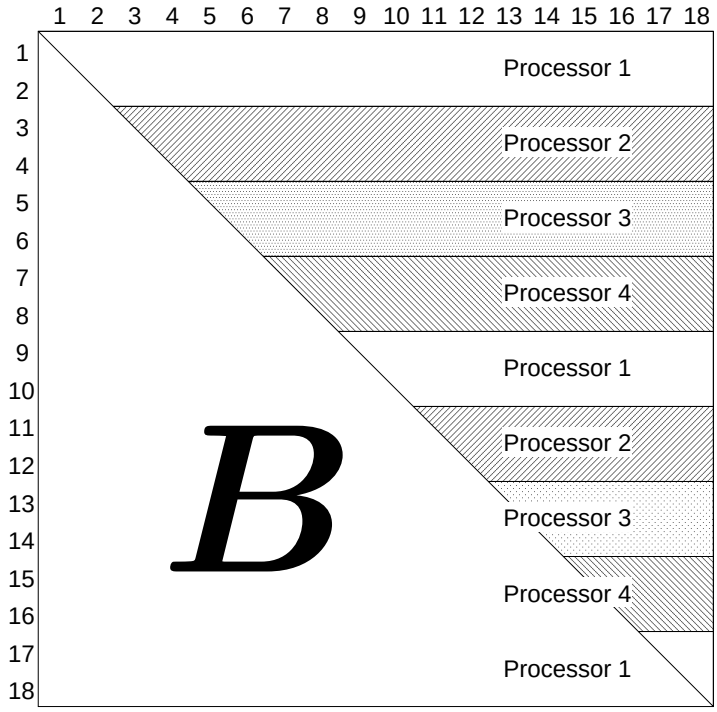
Figure 3.1 will enable us to understand the concept more clearly than the above precise explanation. We illustrate in Figure 3.1 the case when $\boldsymbol{B}$ is a 18×18 matrix, the number of processors is $N = 4$ and the size of belt is $sb = 2$. For instance, B_{35} and B_{39} are computed by the processor 2, and $B_{14,15}$ and $B_{14,18}$ are computed by the processor 3. Note that $\boldsymbol{B}$ is always symmetric since its elements are formed of $B_{ij} = (\boldsymbol{X}\boldsymbol{A}_i\boldsymbol{Y}^{-1}) \bullet \boldsymbol{A}_j$. Therefore, we need to compute only an upper triangular part of $\boldsymbol{B}$.

We want to emphasize that the above processor assignment enables us to distribute memory space to store $\boldsymbol{B}$ in the straight way. The rows indicated by $\mathcal{P}_u$ is stored in memory space on the uth processor. From viewpoints of not only computation cost but also memory space, we can easily achieve well load-balance.

Furthermore, communication between multiple processors is not necessary for the evaluations of the Schur complement matrix, provided that we distribute information of $\boldsymbol{X}, \boldsymbol{Y}, \ \boldsymbol{A}_1, \boldsymbol{A}_2, \ldots, \boldsymbol{A}_m$ on all the processors before the evaluations. This significant characteristic results in surprising high scalability as reported in numerical results.

Although the above description of the row-wise distribution is very simple, it contains enough basic concepts for actual implementation. However, the exploitation of the input data matrices sparsity by [22] and the block diagonal structure make evaluations of $\boldsymbol{B}$ difficult.

Figure 3.1: Evaluation of the Schur complement matrix $\boldsymbol{B}$



For a moment, we take a look how the block diagonal structure affect the formula $B_{ij} = (\boldsymbol{X}\boldsymbol{A}_i\boldsymbol{Y}^{-1}) \bullet \boldsymbol{A}_j$. Let $\{\boldsymbol{X}\}_l$ denote the lth diagonal block matrix of $\boldsymbol{X}$. Here, we consider the case that the block structure comprises of two blocks, then

$$\begin{aligned}
B_{ij} &= (\boldsymbol{X}\boldsymbol{A}_i\boldsymbol{Y}^{-1}) \bullet \boldsymbol{A}_j \\
&= \left(\begin{bmatrix} \{\boldsymbol{X}\}_1 & \\ & \{\boldsymbol{X}\}_2 \end{bmatrix} \begin{bmatrix} \{\boldsymbol{A}_i\}_1 & \\ & \{\boldsymbol{A}_i\}_2 \end{bmatrix} \begin{bmatrix} \{\boldsymbol{Y}^{-1}\}_1 & \\ & \{\boldsymbol{Y}^{-1}\}_2 \end{bmatrix} \right) \bullet \begin{bmatrix} \{\boldsymbol{A}_j\}_1 & \\ & \{\boldsymbol{A}_j\}_2 \end{bmatrix} \\
&= \left(\begin{bmatrix} \{\boldsymbol{X}\}_1\{\boldsymbol{A}_i\}_1\{\boldsymbol{Y}^{-1}\}_1 & \\ & \{\boldsymbol{X}\}_2\{\boldsymbol{A}_i\}_2\{\boldsymbol{Y}^{-1}\}_2 \end{bmatrix} \right) \bullet \begin{bmatrix} \{\boldsymbol{A}_j\}_1 & \\ & \{\boldsymbol{A}_j\}_2 \end{bmatrix} \\
&= \{\boldsymbol{X}\}_1\{\boldsymbol{A}_i\}_1\{\boldsymbol{Y}^{-1}\}_1 \bullet \{\boldsymbol{A}_j\}_1 + \{\boldsymbol{X}\}_2\{\boldsymbol{A}_i\}_2\{\boldsymbol{Y}^{-1}\}_2 \bullet \{\boldsymbol{A}_j\}_2 \\
&= B_{ij}^{(1)} + B_{ij}^{(2)},
\end{aligned}$$

where $\boldsymbol{B}^{(1)}$ and $\boldsymbol{B}^{(2)}$ are the sub Schur complement matrices generated from 1st and 2nd diagonal blocks, respectively. We can extend this result for a more general case with h diagonal blocks,

$$\boldsymbol{B} = \Sigma_{l=1}^{h} \boldsymbol{B}^{(l)}.$$

Therefore, it will be reasonable that we compute B_{ij} over all the diagonal blocks by only the uth processor ($i \in \mathcal{P}_u$). Otherwise, if we compute $B_{ij}^{(l)}$ on the uth processor and $B_{ij}^{(k)}$ on the vth processor, a communication between the uth and the vth processors is necessary for the addition $B_{ij} = B_{ij}^{(l)} + B_{ij}^{(k)} + \cdots$ and a storage of the result on memory space of the uth processor.

On the other hand, a notable feature in the exploitation of the sparsity by [22] is that they propose three formula $\mathcal{F}_1, \mathcal{F}_2, \mathcal{F}_3$ to evaluate elements of $\boldsymbol{B}$. A combination of this feature and the block diagonal structure makes parallel processing over distributed memory more complicated. Suppose that $\boldsymbol{A}_i$ and $\boldsymbol{A}_j$ are sufficiently sparse, then they use the formula $\mathcal{F}_3$ to evaluate B_{ij} from $\boldsymbol{A}_i$ and $\boldsymbol{A}_j$ directly.

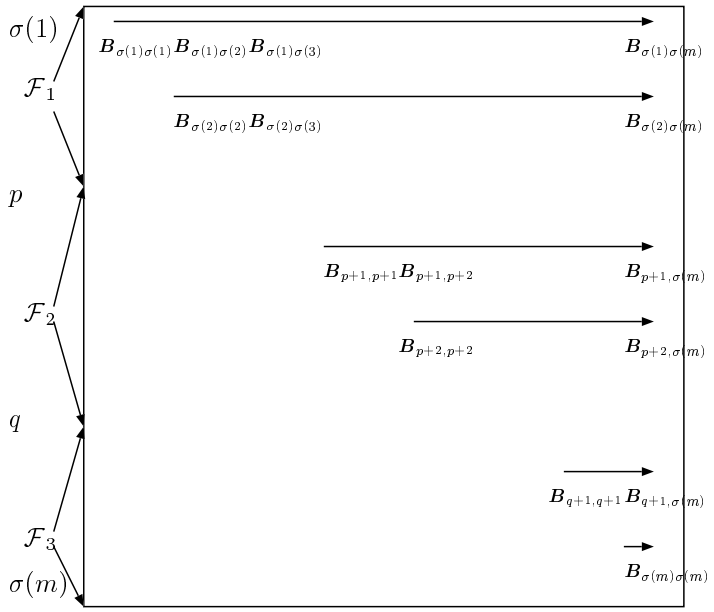
$$B_{ij} = (\boldsymbol{X}\boldsymbol{A}_i\boldsymbol{Y}^{-1}) \bullet \boldsymbol{A}_j$$

$$
\begin{aligned}
&= \Sigma_{p=1}^{n}\Sigma_{q=1}^{n}[\boldsymbol{X}\boldsymbol{A}_i\boldsymbol{Y}^{-1}]_{pq}[\boldsymbol{A}_j]_{pq} \\
&= \Sigma_{p=1}^{n}\Sigma_{q=1}^{n}(\Sigma_{r=1}^{n}\Sigma_{s=1}^{n}X_{pr}[\boldsymbol{A}_i]_{rs}Y_{sq}^{-1})[\boldsymbol{A}_j]_{pq}
\end{aligned}
$$

However, if the $\boldsymbol{A}_i$ is almost fully dense, an usage of a temporary storage $\boldsymbol{T} = \boldsymbol{X}\boldsymbol{A}_i\boldsymbol{Y}^{-1}$ is clearly smart since we do computation $B_{ij} = \boldsymbol{T} \bullet \boldsymbol{A}_j$ over all $j = 1, 2, \ldots, m$ with only inner-product, that is, the formula $\mathcal{F}_1$. They exploit $\mathcal{F}_2$ formula to compute B_{ij} with dense $\boldsymbol{A}_i$ and sparse $\boldsymbol{A}_j$. Furthermore, they estimate the computation cost for each formula regarding the number of required multiplications and overheads for sparse data structures. We store the input data matrices into special data structures to cut memory space and computation cost of zero elements.

They have proved that the most efficient manner to evaluate $\boldsymbol{B}$ based on the estimation is as follow. At first, they sort the input data matrices by their density with a permutation σ so that $\boldsymbol{A}_{\sigma(1)}, \boldsymbol{A}_{\sigma(2)}, \ldots, \boldsymbol{A}_{\sigma(m)}$ satisfy $nz(\sigma(i)) \geq nz(\sigma(j))$ for $\sigma(i) \leq \sigma(j)$, where $nz(\sigma(i))$ is the number of non-zero elements of $\boldsymbol{A}_{\sigma(i)}$. A key-point of their proof is that there exist two indices $p, q (1 \leq p \leq q \leq m)$ such that we can attain the minimum computation cost when we apply the formulas $\mathcal{F}_k$ for each row $\mathcal{B}_k$, where $\mathcal{B}_1 = \{1 \leq \sigma(i) \leq p\}, \mathcal{B}_2 = \{p \leq \sigma(i) \leq q\}, \mathcal{B}_3 = \{q \leq \sigma(i) \leq m\}$. We do not change the formula on each row since, for example, $\mathcal{F}_1$ is the most efficient if we compute the temporary matrix $\boldsymbol{T} = \boldsymbol{X}\boldsymbol{A}_i\boldsymbol{Y}^{-1}$ even once. Consequently, they evaluate the elements of $\boldsymbol{B}$ in the order of the permutation σ as Figure 3.2.

Figure 3.2: Order of elements of $\boldsymbol{B}$ and their formula



In the combination of the sparsity exploitation and the block diagonal structure, it is promising that we sort the input data matrices on non-zero number of each sub-block matrices. Let $\sigma(i, l)$ be a permutation for the lth block so that $\{\boldsymbol{A}_{\sigma(i,l)}\}_l$ has more non-zero number than $\{\boldsymbol{A}_{\sigma(j,l)}\}_l$ if $\sigma(i, l) \leq \sigma(j, l)$.

Here, we consider a plain model which has $m = 4$ constraints and $h = 2$ block diagonal matrices. Thus, the Schur complement matrix becomes 4×4 symmetric matrix. In addition, suppose $\sigma(i, 1)$ and $\sigma(i, 2)$ for the permutation of 1st block and 2nd block as follow.

$$
\begin{aligned}
&\sigma(1,1) = 2,\ \sigma(2,1) = 1,\ \sigma(3,1) = 4,\ \sigma(4,1) = 3, \\
&\sigma(1,2) = 3,\ \sigma(2,2) = 2,\ \sigma(3,2) = 1,\ \sigma(4,2) = 4.
\end{aligned}
$$

In other word, $\boldsymbol{A}_{\sigma(4,1)}$ is the 1st block of $\boldsymbol{A}_3$, *i.e.* $\{\boldsymbol{A}_3\}_1$. Furthermore, we assume that we use two processors P_1, P_2 and set the size of belt is $sb = 1$. Figure 3.3 for the 1st block and Figure 3.4 for the 2nd block show the order of evaluations of elements of $\boldsymbol{B}^{(1)}$ and $\boldsymbol{B}^{(2)}$, respectively, with the original row number in $\boldsymbol{B}$ and

the assigned processors for each row. As space is limited, we use an abbreviation $\sigma^l_{i,j} = B^{(l)}_{\sigma(i,l),\sigma(j,l)}$ in these figures.

Figure 3.3: Original row number and assign processors for 1st block

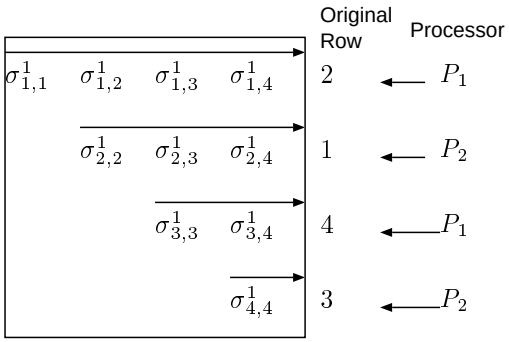
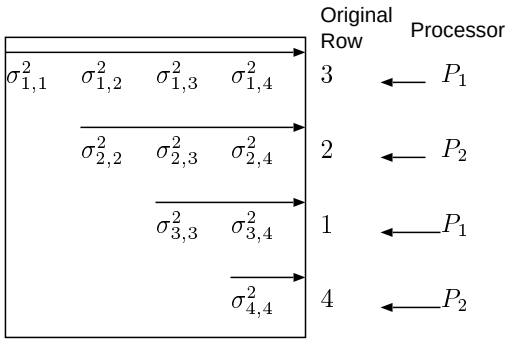


Figure 3.4: Original row number and assign processors for 2nd block



The difficulty we need to focus on here is clarified by the computation for B_{14}

$$B_{14} = B^{(1)}_{14} + B^{(2)}_{14} = B^{(1)}_{\sigma(2,1)\sigma(3,1)} + B^{(2)}_{\sigma(3,2)\sigma(4,2)} = \sigma^1_{2,3} + \sigma^2_{3,4}$$

From these figures, $\sigma^1_{2,3}$ is computed by the processor P_2, while $\sigma^2_{3,4}$ is computed by the processor P_1. Therefore, a communication between P_1 and P_2 is required to add the two elements for B_{14}.

It is apparent that the difficulty is originated from the different permutations in the order of density at each block. It is desired on parallel processing that the Schur complement matrix can be evaluation with as small communication as possible. On the same time, we need to retain the advantage of the formulas $\mathcal{F}_1, \mathcal{F}_2, \mathcal{F}_2$.

The following corollary derived from Theory 3.1 in [22] provides an important concept for the evaluation of the Schur complement $\boldsymbol{B}$ without any communication between multiple processors.

Corollary 3.2.1. *Suppose that we pre-process the selection of the formula $\mathcal{F}_1, \mathcal{F}_2, \mathcal{F}_3$ for all pair (i,l), the ith row and the lth block, based on Theory 3.1 in [22] before the evaluation of the Schur complement matrix $\boldsymbol{B}$. Then we can attain the minimum computation for $\boldsymbol{B}$ without any communication by an manner that $B^{(l)}_{ij}$ evaluated on ith row by the formula for (i,l) if $\{A_i\}_l$ has more non-zero elements than $\{A_j\}_l$.*

Due to the corollary, we develop an algorithm for the evaluation of $\boldsymbol{B}$.

Algorithmic Framework for Evaluation of the Schur Complement Matrix

Before starting the first iteration of PD-IPM, we select $\mathcal{F}_{(i,l)}$ from $\mathcal{F}_1, \mathcal{F}_2, \mathcal{F}_3$ for the ith row and the lth block. In every iterations of PD-IPM, we compute $\boldsymbol{B}$ as follow.

Set $\boldsymbol{B} = \boldsymbol{O}$
For $l = 1, 2, \ldots, h$ (iterator for block)
  For $i = 1, 2, \ldots, m$ (iterator for row)
    For $j = 1, 2, \ldots, m$ (iterator for column)
      If $nz(\{A_i\}_l) \geq nz(\{A_j\}_l)$ then
        Compute $B^{(l)}_{ij}$ by $\mathcal{F}_{(i,l)}$
        $B_{ij} \leftarrow B_{ij} + B^{(l)}_{ij}$

Based on the above algorithm, we can easily divide the computation and subsequently the memory space into the row-wise distribution on multiple processors. Furthermore, the algorithm enables us to retain the row-wise division proposed by $\mathcal{P}_u$ in (3.3) from viewpoints of both computation cost and memory

space. It should be emphasized that no communication between multiple processors is required in the above algorithm.

However, the algorithm has a drawback that we do not have the guarantee of symmetric property of the Schur complement matrix $\boldsymbol{B}$ since, for example $B_{12}^{(1)}$ is computed on the fist row and $B_{12}^{(2)}$ is computed on the second row, respectively. Some communication is required to symmetrize $\boldsymbol{B}$ by an addition the lower part into the upper part. Usually, communication time for the symmetrization is very smaller than computation time for each element of $\boldsymbol{B}$. Therefore we can justify the communication by the remarkable reduction of the computation time for each element owing to parallel computation.

Nevertheless, in case almost all the input data matrices are considerably sparse, we had better compute each component ignoring the density instead of the symmetrization of $\boldsymbol{B}$. In other words, we compute $B_{ij}^{(l)}$ $(i \leq j)$ over all blocks in the ith row without comparison between $nz(\{\boldsymbol{A}_i\}_l)$ and $nz(\{\boldsymbol{A}_j\}_l)$, because the computation time may become too short compared to the communication for the symmetrization. Theta Function Problem explained in section 3.2.1 where all the input data matrices other than one identity matrix has only two non-zero elements is good example to discuss this case.

We end up this subsection with the summary of parallel version of algorithmic framework.

Parallel Processing for Evaluation of the Schur Complement Matrix on the uth Processor

Before starting the first iteration of PD-IPM, we select $\mathcal{F}_{(i,l)}$ from $\mathcal{F}_1, \mathcal{F}_2, \mathcal{F}_3$ for the ith row and the lth block.
Prepare a flag sym.

If $\mathcal{F}_1$ or $\mathcal{F}_2$ are selected more than once
then $sym = 1$,
else $sym = 0$.

Reserve memory space to store rows assigned by $\mathcal{P}_u$.
At each iteration of PD-IPM, we compute as follow.

$\boldsymbol{B} = \boldsymbol{O}$
For $l = 1, 2, \ldots, h$ (iterator for block)
For $i \in \mathcal{P}_u$ (iterator for row)
For $j = 1, 2, \ldots, m$ (iterator for column)
If $nz\{A_i\}_l \geq nz\{A_j\}_l$ or $sym = 0$ then
Compute $B_{ij}^{(l)}$ by $\mathcal{F}_{(i,l)}$
$B_{ij} \leftarrow B_{ij} + B_{ij}^{(l)}$
If $sym = 1$ then
Symmetrize $\boldsymbol{B}$ over distributed memory

3.2.3 Parallel Cholesky Factorization and Two-Dimensional Block-Cyclic Distribution

After the computation of the elements of the Schur complement matrix $\boldsymbol{B}$, we apply the parallel Cholesky Factorization, provided by ScaLAPACK [8], to $\boldsymbol{B}$ in order to solve the Schur complement equation $\boldsymbol{B}d\boldsymbol{z} = \boldsymbol{r}$. The matrix $\boldsymbol{B}$ is always positive definite throughout all the iterations of PD-IPM. Since ScaLAPACK assumes the elements of a positive definite matrix to be factorized are distributed according to *the two-dimensional block-cyclic distribution*, over distributed memory, what SDPARA needs to do before calling the parallel Cholesky Factorization routine is to redistribute the elements of $\boldsymbol{B}$ as ScaLAPACK assumes. In this paper, we abbreviate the two-dimensional block-cyclic distribution to TD-BCD. To estimate amount of communication for the parallel Cholesky Factorization as precisely as possible, we describe more details of TD-BCD and how we apply the Cholesky Factorization on parallel processing. In this description, zero-origin is smart for the attachment of numbers. Usually, we use one-origin and count, for example, the 1st row, the 2nd row, ..., the mth row. On the other hand, in zero-origin, we start from 0, that is, the 0th row, the 1st row, ..., the $(m-1)$th row. Zero-origin has an advantage of removing troublesome of remainders and floor numbers arisen from divisions of integers.

Generally speaking, TD-BCD adopted by ScaLAPACK distributes a large matrix on distributed memory space in the following style. Suppose that the number of available processors is $P \times Q = N$. In other words,

we have attached a row number P_r $(0 \leq P_r \leq P-1)$ and a column number Q_c $(0 \leq Q_c \leq Q-1)$ on each processor in the way $P(0,0), P(0,1), \ldots, P(0,Q-1), P(1,0), \ldots, P(P-1,Q-1)$. Furthermore, we assume the size of the matrix $\boldsymbol{B}$ to be factorized is $m \times n$. ScaLAPACK routines demand parameters, block sizes mb and nb, that we choose depending on network environments for numerical experiments.

Thus, the (i,j) element of $\boldsymbol{B}$, that is B_{ij} (Note that $0 \leq i \leq m-1, 0 \leq j \leq n-1$ because of zero-origin), is stored in the $(m \bmod m_b, n \bmod n_b)$ position of the $(\lfloor \lfloor m/m_b \rfloor / P \rfloor, \lfloor \lfloor n/n_b \rfloor / Q \rfloor)$ block on the $(\lfloor m/m_b \bmod P \rfloor, \lfloor n/n_b \bmod Q \rfloor)$ processor. We use the notations $a \ mod \ b$ to denote the remainder when we divide the integer a by b, and $\lfloor x \rfloor$ to denote the largest integer that does not exceed x.

An example shown in Figure 3.5, where we have a 9×14 matrix and 3×2 processors and assign parameters $mb = 2$ and $nb = 3$, will help us to understand this complicated distribution. Figure 3.6 shows the same distribution of the matrix by sorting from the viewpoints of the memory space on each processor.

Figure 3.5: Two-dimensional block-cyclic distribution

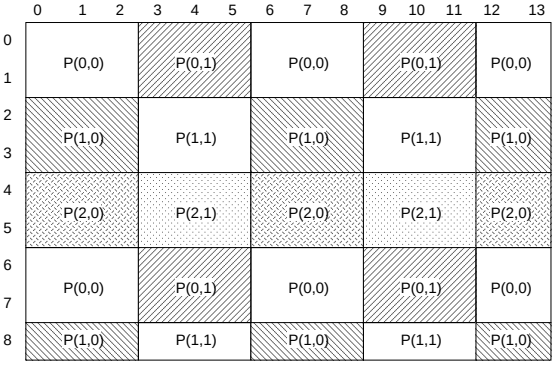
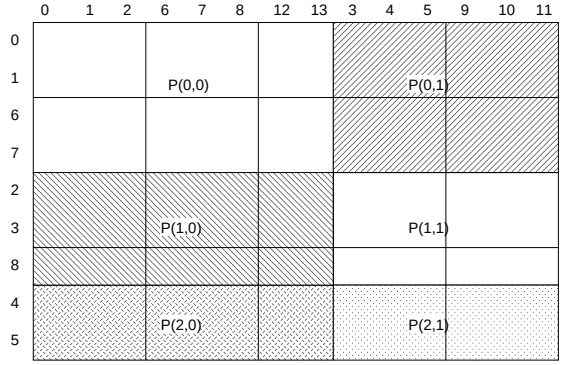


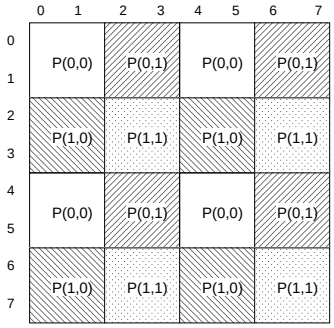
Figure 3.6: Reposition of two-dimensional block-cyclic distribution



With the assistance of TD-BCD, ScaLAPACK attains well-considered load-balance of both memory space and computation cost. See [16] to consult more details of excellent features being capable by ScaLAPACK.

Fortunately, we can restrict our situation to simpler case for more clear estimation of communication between multiple processors, since we only apply the parallel Cholesky Factorization. The Schur complement matrix $\boldsymbol{B}$ is always a symmetric matrix, therefore, a restriction $m = n$ and parameters assignment as $mb = nb$ are sufficiently reasonable. In order for plainness, we assume $P = Q$ that the number of available processors is necessary to be an square of some integer, $N = P^2$, and m is a multiple number of mb. Figure 3.7 shows the case that $m = 8, mb = 2, P = 2$ with 4 processors. A comparison between Figure 3.5 and Figure 3.7 exhibits how much simpler our restrictions and assumptions change the distributions.

Figure 3.7: Two-dimensional block-cyclic distribution for the Cholesky Factorization



It is important to understand the parallel Cholesky Factorization is that the Cholesky Factorization can be decomposed into each block factorization. Let $\boldsymbol{B}$ be factorized as $\boldsymbol{B} = \boldsymbol{U}^T\boldsymbol{U}$, where $\boldsymbol{U}$ is an upper

triangular matrix. Furthermore, we suppose that we partition $\boldsymbol{B}$ and $\boldsymbol{U}$ into 4 sub-block.

$$\boldsymbol{B} = \begin{pmatrix} \boldsymbol{B}_{11} & \boldsymbol{B}_{12} \\ \boldsymbol{B}_{12}^T & \boldsymbol{B}_{22} \end{pmatrix}, \quad \boldsymbol{U} = \begin{pmatrix} \boldsymbol{U}_{11} & \boldsymbol{U}_{12} \\ & \boldsymbol{U}_{22} \end{pmatrix},$$

where the size of matrices $\boldsymbol{B}_{11}, \boldsymbol{B}_{12}, \boldsymbol{B}_{12}^T$ and $\boldsymbol{B}_{22}$ are $mb \times mb, mb \times (m-mb), (m-mb) \times mb$ and $(m-mb) \times (m-mb)$, respectively. For instance in Figure 3.7, 4 elements, B_{00}, B_{01}, B_{10} and B_{11} comprise of the left-upper block $\boldsymbol{B}_{11}$ and the right-lower block $\boldsymbol{B}_{22}$ has 6 rows and 6 columns. Then we virtually factorize $\boldsymbol{B}$ in block-oriented.

$$\begin{aligned} \boldsymbol{B} &= \begin{pmatrix} \boldsymbol{B}_{11} & \boldsymbol{B}_{12} \\ \boldsymbol{B}_{12}^T & \boldsymbol{B}_{22} \end{pmatrix} \\ = \quad \boldsymbol{U}^T\boldsymbol{U} &= \begin{pmatrix} \boldsymbol{U}_{11}^T & \\ \boldsymbol{U}_{12}^T & \boldsymbol{U}_{22}^T \end{pmatrix} \begin{pmatrix} \boldsymbol{U}_{11} & \boldsymbol{U}_{12} \\ & \boldsymbol{U}_{22} \end{pmatrix} = \begin{pmatrix} \boldsymbol{U}_{11}^T\boldsymbol{U}_{11} & \boldsymbol{U}_{11}^T\boldsymbol{U}_{12} \\ \boldsymbol{U}_{12}^T\boldsymbol{U}_{11} & \boldsymbol{U}_{12}^T\boldsymbol{U}_{12} + \boldsymbol{U}_{22}^T\boldsymbol{U}_{22} \end{pmatrix}. \end{aligned}$$

Consequently, we arrive at the block-oriented Cholesky Factorization.

Algorithmic Framework for the block-oriented Cholesky Factorization

Step 1 (Inner Cholesky Factorization): Apply the non block-oriented Cholesky Factorization to one block $\boldsymbol{B}_{11}$ and acquire $\boldsymbol{U}_{11}$.

Step 2 (Computation of the Upper Part): We compute $\boldsymbol{U}_{12} = (\boldsymbol{U}_{11}^T)^{-1}\boldsymbol{B}_{12}$

Step 3 (Update the Right-Lower Block): Update $\boldsymbol{B}_{22} \Leftarrow \boldsymbol{B}_{22} - \boldsymbol{U}_{12}^T\boldsymbol{U}_{12} = \boldsymbol{U}_{22}^T\boldsymbol{U}_{22}$.

Step 4 (Next Block): Partition $\boldsymbol{B}_{22}$ into smaller blocks and apply the block-oriented Cholesky Factorization to $\boldsymbol{B}_{22}$ to obtain $\boldsymbol{U}_{22}$.

In brief, we take a look what will be computed when we apply the above algorithm to $\boldsymbol{B}$ in Figure 3.7. First, we apply the Cholesky Factorization implemented for a single processor to the $mb \times mb$ left-upper block $\boldsymbol{B}_{11}$, which is composed of B_{00}, B_{01}, B_{10} and B_{11}, on Processor $P(0,0)$. The $mb \times mb$ left-upper block is factorized on a single processor without any communication. Second, the information of $\boldsymbol{U}_{11}$ is sent to all processors involved in the computation for $\boldsymbol{U}_{12}$, that is, the processors $P(0,0)$ and $P(0,1)$. Therefore, the message from $P(0,0)$ to $P(0,1)$ is necessary in this step. Furthermore, we send $\boldsymbol{U}_{12}$ and its transpose for the update of the right-lower block $\boldsymbol{B}_{22}$. We should recognize is that the update for 4 elements B_{26}, B_{27}, B_{36} and B_{37}, requires only $U_{02}, U_{03}, U_{12}, U_{13}, U_{06}, U_{07}, U_{16}$ and U_{17}, not all elements of $\boldsymbol{U}_{12}$. This characteristic significantly reduces the amount of communication for having $\boldsymbol{U}_{12}$. Finally, we repeat m/mb times of these computation to apply the Cholesky Factorization to all components of $\boldsymbol{B}$; 4 times in the Figure 3.7.

We estimate total computation cost and communication needed for the parallel Cholesky Factorization over all processors, in a general matrix $\boldsymbol{B}$, and summarize in Table 3.4.

Table 3.4: Total estimation of computation cost and communication over all processors

	Computation Cost	Communication
Step 1	$\frac{1}{3} \times m \times mb^2$	0
Step 2	$\frac{1}{4} \times m^2 \times mb$	$(P-1) \times m \times mb$
Step 3	$\frac{1}{3} \times m^3$	$(P-1) \times m^2$

Based on Table 3.4, we advance average computation cost and communication on each processor, since we are focusing on parallel processing. At most P and P^2 processors join the computation in Step 2 and Step 3, respectively. Meanwhile, estimations of communication on each processor will become very difficult, because it seriously depends on topological structures of network environments. Here, we make a simple assumption that a processor in a row or column of processor position can broadcast their elements without any effect for other rows or columns. Concretely saying, we can broadcast two messages simultaneously over two rows, from $P(i_1, j_1)$ to $P(i_1, j_2)$ $(j_2 = 0, 1, \ldots, P-1)$ and from $P(i_2, j_2)$ to $P(i_2, j_2)$ $(j_2 = 0, 1, \ldots, P-1)$. Based on this assumption, we can divide the amount of communication by at most P processors. In step

Table 3.5: Estimation of computation cost and communication on each processor

	Computation Cost	Communication
Step 1	$\frac{1}{3} \times m \times mb^2$	0
Step 2	$\frac{1}{4} \times m^2 \times mb/P$	$(P-1) \times m \times mb$
Step 3	$\frac{1}{3} \times m^3/P^2$	$(P-1) \times m^2/P$

1, the computation is done on only a single processor. Hence, other processors are waiting the only one processor. The results of estimation regarding computation cost and communication on each processor are shown in Table 3.5.

Since we consider large-scale matrix to be factorized, we take mb/m and P/m as zero. As a result, the parallel Cholesky Factorization requires $\frac{1}{3} \times m \times mb^3 + \frac{1}{4} \times m^2 \times mb/P + \frac{1}{3} \times m^3/P^2$ computation costs and $(P-1) \times m \times (mb + m/P)$ communication. We can not neglect that each transmission of message has overhead for processing of network protocol. Let h be an overhead for the protocol at each transmission of size mb. Then, the amount of communication with the overhead turns out to be $(P-1) \times m \times (mb + m/P) \times (1 + (m/mb) \times h)$ for the reason that we have m/mb blocks.

Therefore, we are facing a trade-off between the computation cost and communication. Smaller block size decreases the computation cost, at the same time increases the amount of communication cost. By choosing an appropriate block size mb depending on network configurations, we will verify through numerical results that the parallel Cholesky Factorization undoubtedly reduces computation time overcoming the overheads of communication between multiple processors. In particular, an effect that the primal term of the computation cost has the denominator $P^2 = N$ is clearly meaningful from the viewpoint of parallel processing.

Similar analysis as the above can be seen in [29], which discusses only the case $mb = 1$.

3.2.4 SDPARA (SemiDefinite Programming Algorithm paRAllel Version)

In sections 3.2.2 and 3.2.3, we have investigated the concepts to overcome the bottlenecks of PD-IPM on a single processor, constructing and solving the Schur complement matrix, with significant advantage of parallel computation. Based on these fundamental concepts, we have implemented SDPARA (SemiDefinite Programming Algorithm paRAllel version) on multiple processors and distributed memory. SDPARA as well as SDPA can be downloaded from the SDPA Home Page:

http://sdpa.is.titech.ac.jp/

In the following section, we will exhibit the high performance provided by SDPARA through numerical results.

To implement SDPARA, we utilize two libraries, MPI (Message Passing Interface) for communication between multiple processors, and ScaLAPACK(Scalable Linear Algebra PACKage) for the parallel Cholesky Factorization described in section 3.2.3. MPI is de facto standard communication library for parallel computer composed of multiple processors, and designed to aim the standard regarding efficiency, portability, and flexibility on parallel processing. The manifold benefits from MPI enable us implement parallel software without studying complicated protocols for network programming; we can concentrate on data distribution and numerical algorithms. ScaLAPACK utilizes MPI as a method for communication. Other than ScaLAPACK, there are many software to apply linear algebra on parallel precessing, for example, PLAPACK [66], TAO (Toolkit for Advance Optimization) [71]. The reason we choose ScaLAPACK is that we can directly access memory space allocated on each processor. The merit is very important, in particular, for constructing the Schur complement matrix. Since we hold an exact position in memory space to store a computation result in advance by only information retained on each processor without any information of a entire matrix, we really compute the Schur complement matrix without communication between multiple processors, resulting in excellent scalability.

Before showing the numerical results, we should reconsider the flow of the data distribution. In order to evaluate the Schur complement matrix as mentioned in section 3.2.2, every processor needs to maintain all input data, $\boldsymbol{A}_1, \boldsymbol{A}_2, \ldots, \boldsymbol{A}_m, \boldsymbol{C}, b_1, b_2, \ldots, b_m$. Since these data are constant for all iterations of PD-IPM, we distribute these data over all processors at the beginning of the execution of SDPARA. Furthermore, some portions for the Schur complement matrix, $\boldsymbol{X}, \boldsymbol{Y}$, and $\boldsymbol{z}$, are also maintained by each processor throughout all the iterations of PD-IPM.

On the other hand, the Schur complement matrix $\boldsymbol{B}$ itself is allocated on distributed memory. We prepare two type memory allocation for the Schur complement matrix $\boldsymbol{B}$, the row-wise distribution for evaluation and TD-BCD for applying the parallel Cholesky Factorization. In each iteration of PD-IPM, we redistribute the Schur complement matrix from row-wise distribution to TD-BCD. After solving the Schur complement equation on parallel processing, the result vector $d\boldsymbol{z}$ is broadcast for all processors to obtain $d\boldsymbol{X}, d\boldsymbol{Y}$ and each processor progresses subsequent components.

After all, the data flow over distributed memory is wrapped as follow. Note that only one processor is required to access an input file to read given datas. Here, we call the processor as the first processor.

The Distribution of Data Flow of SDPARA

Step 1 (Parameter Setting): Set the belt size (sb) and the block size (mb). These values must coincide over all the processors.

Step 2 (Input Data Matrices): The first processor reads the given datas, $m, n, \boldsymbol{A}_1, \boldsymbol{A}_2,,\ldots, \boldsymbol{A}_m, \boldsymbol{C}$, $b_1, b_2, \ldots, b_m$ from an input file. Then, the first processor broadcasts these data to all the processors.

Step 3 (Memory Allocation): Each processor allocates the memory space for the current point $(\boldsymbol{X}, \boldsymbol{Y}, \boldsymbol{z})$ and the search direction $(d\boldsymbol{X}, d\boldsymbol{Y}, d\boldsymbol{z})$. In addition, each processor allocates the memory space for the Schur complement matrix, $\boldsymbol{B}_1$ for the row-wise and $\boldsymbol{B}_2, d\boldsymbol{z}_2$ for TD-BCD defined by m, n, sb, mb and the rank of the processor. We use $d\boldsymbol{z}_2$ for $d\boldsymbol{z}$ allocated on distributed memory.

Step 4 (Main Loop Start): If the terminal condition of PD-IPM is satisfied, the first processor prints out the result, and all the processor release their memory space, finally, stop the algorithm.

Step 5 (Evaluating the Schur complement matrix): Each processor evaluates the assigned elements of the Schur complements and stores them in $\boldsymbol{B}_1$ independently from other processors.

Step 6 (The Parallel Cholesky Factorization): Redistribute the Schur complement matrix from $\boldsymbol{B}_1$ to $\boldsymbol{B}_2$. If needed, $\boldsymbol{B}_2$ is symmetrized. Then, apply the parallel Cholesky Factorization provided ScaLAPACK to $\boldsymbol{B}_2$. Subsequently, apply forwarding and backwarding substitution to obtain $d\boldsymbol{z}_2$.

Step 7 (Search Direction and Update): Copy from $d\boldsymbol{z}_2$ over distributed memory composed of multiple processors to $d\boldsymbol{z}$ on the first processor. Then the first processor broadcast $d\boldsymbol{z}$ to all the processors. Finally, each processor progresses the computation of PD-IPM to acquire the rest components of the search direction $(d\boldsymbol{X}, d\boldsymbol{Y})$ and updates $(\boldsymbol{X}, \boldsymbol{Y})$. Goto Step 4.

Since ELEMENTS component (Step 5) is done without any communication, the most portion of communication is originated from CHOLESKY part (Step6). Therefore CHOLESKY part may be strongly affected by the network environments for numerical experiments. One point to be noticed is that we synchronize all the processors in only Step 4, and Step 6 in Main Loop, and other components of PD-IPM are done on each processor independently and individually. Therefore the overhead required for the synchronization has already been minimized.

Numerical results in the following sections will prove the high performance of SDPARA, in particular, from the viewpoints of the meaningful reduction of the computation time and the scalability.

3.3 Computation Environment for Numerical Experiments

Here, let us mention the computation environments for our numerical experiments before moving on to numerical results.

We are very fortunate to have the opportunities to execute our numerical experiments on the three PC-clusters, namely, Presto-I , Presto-III and sdpa. Presto-I and Presto-III are assembled and maintained by Matsuoka lab, Tokyo Institute of Technology, while sdpa cluster is supported by Fujisawa lab, Tokyo Denki University. PC-cluster is promising in cluster technology mentioned in section 3.1 and categorized into distributed memory system.

A PC-cluster is composed of multiple Personal Computers connected by LAN (Local Area Network) and designed to achieve high performance owing to parallel computation. PC-clusters are superior to specialized parallel computers form many standpoints. First, almost all physical components of PC-clusters

are commonplace parts, therefore, it is relatively easy to assemble PC-clusters. Second, when we are faced with update after a passage of time since built-up, we can replace some components by a process of easy gradation, not all components at once. A life span of PC-clusters may be longer than we expect to specialized parallel computers. In addition, we are able to apply the same knowledge on PC-cluster as on a single Personal Computer, for instance, Operating Systems, programming tools and programming languages.

We summarize the above PC-cluster specs in Table 3.6.

Table 3.6: PC-cluster specs

Name	Presto-I	Presto-III	sdpa
Number of nodes	64	256	45
Processor	Celeron 1.4 GHz	Athlon 1.6 GHz	Athlon 1.2 GHz
Number of Processors on each node	1	2	2
Memory Space on each node	384 MB	768 MB	2 GB (8 nodes) 1 GB (37 nodes)
Network	100 Base-TX Ethernet	Myrinet-2000	Gigabit Ethernet
Performance	20 GFLOPS	760 GFLOPS	105 GFLOPS

The Operating Systems on all the three PC-clusters are *Linux*, which has been developed since 1981. Traditionally speaking, Linux comes from Linus. B. Trovalds hobby. However, it is obvious that the state-of-art Linux has gone beyond borders of hobbies. At present time, Linux is considered as a strong and stable Operating System. Linux provides sufficient flexibilities on network environments and powerful instruments for numerical computations.

Here, we pick up some ingredients from each PC-cluster. PAPI [52] enables us to measure load-balance on Presto-I. Since PAPI supplies detail information of microprocessor, we can count the number of operations executed on each processor involved in parallel computation. The environment for parallel computation on sdpa-cluster is essentially under a supervision of SCore [65]. SCore developed by RWCP (Real World Computing Partnership) is comprised of communication library, administration and programming tools, run-time environments and other components to realize parallel computation on PC-cluster more near-at-hand. Myrinet-2000 developed by Myricom [58] strongly enhances the network performance of Presto III. The reason of the high capacity is shortly that Myrinet loads special network processors and special switch to connect each nodes. In particular, the special network processor can input-communication and output-communication simultaneously.

Since we have a chance that we run on three PC-clusters, we can select the PC-cluster depending on objectives of numerical results. We select Presto-I on experimental stage of a parallel implementation. Based on the number of operations executed on each processor reported by PAPI, we adjust load-balance to attain higher scalability. Subsequently, we test on sdpa-cluster with faster network than Presto-I. We can obtain the effect caused by the difference of network speed. Finally, we try the largest problem on Presto-III, because Presto-III has the highest performance in the three PC-clusters as seen in Table 3.6. Unless otherwise stated, all numerical experiments in this thesis were done on Presto-III.

3.4 Preliminary Numerical Experiments and Evaluations of SDPARA

In advance of full-scale numerical experiments for some benchmark SDPs and extremely large SDPs, we have done preliminary experiments to understand fundamental characteristics of the parallel implementation of SDPARA. To achieve high performance for full-scale experiments, reflections of the knowledge from the preliminary experiments are necessary.

The first subject is a choice of the belt size to allocate the evaluation of the Schur complement on each processor. Then, we compare the Cholesky Factorization and Conjugate Gradient Method to solve the

Schur complement equation from the viewpoint of parallel processing. Finally, we investigate an effect of network capacity to SDPARA. We pick up control11 and theta6 to measure performance of SDPARA, since the most time consuming components of the two SDPs have been replaced by their parallel implementation in SDPARA.

3.4.1 Issues of Belt Size

As we have mentioned in section 3.2.2, the strategy adopted by SDPARA to allocate of the evaluation of the Schur complement matrix $\boldsymbol{B}$ to each processor is the row-wise distribution. Precisely saying, the uth processor evaluate all elements in the ith row ($i \in \mathcal{P}_u$). Let m be the size of $\boldsymbol{B}$ and N be the number of available processors, respectively. Then $\mathcal{P}_u$ is composed of some belts,

$$\mathcal{P}_u = \cup\{i : i \in \mathcal{B}_w, w\%N = u\}, \quad \text{where } \mathcal{B}_w = \{i : (w-1) \times sb \leq i \leq \min\{m, w \times sb\}\}.$$

Therefore, we have a choice of the belt size sb. In this subsection, we explore the most adequate sb.

We have applied SDPARA to control11 and theta6 with changing sb from 1 to 32 on Presto I with 32 processors. The reason we have selected Presto I is that we can count the number of operators executed on each processors by PAPI [52]. It means the details of the load-balance are available. Table 3.7 shows the lowest and highest operation counts in 32 processors and their ratio with respect to ELEMENTS, the component of PD-IPM to evaluate $\boldsymbol{B}$.

Table 3.7: Effect of belt size on load-balance

size of belt		1	2	4	8	16	32
control11	lowest	2.54e+10	2.48e+10	2.40e+10	2.32e+10	2.25e+10	1.50e+10
	highest	2.67e+10	2.66e+10	2.85e+10	3.12e+10	3.62e+10	4.03e+10
	ratio	1.05	1.07	1.18	1.34	1.36	2.68
theta6	lowest	1.21e+9	1.21e+9	1.19e+9	1.17e+9	1.10e+9	1.00e+9
	highest	1.25e+9	1.26e+9	1.27e+9	1.31e+9	1.35e+9	1.49e+9
	ratio	1.03	1.04	1.06	1.11	1.22	1.49

For both control11 and theta6, the smaller sb attains the lower ratio. In particular, the case $sb = 1$ results in the most adequate load-balance. On the other hand, if we set sb more than 8, the load-balance is rather worse, which means we have idle processors.

We continue further investigation on control11, since ELEMENTS is the most time consuming components for control11. Figure 3.8 shows the operation counts for ELEMENTS on each processor by ascending order. Thus, the smaller gradient indicates better load-balance. From Figure 3.8, it becomes an apparent conclusion again that the most adequate size is $sb = 1$. In control11, there are various sparsity from very sparse to almost fully-dense in $\boldsymbol{A}_1, \boldsymbol{A}_2, \ldots, \boldsymbol{A}_m$. The big steps in the counts in the case $sb = 16, 32$ come from the characteristic. Therefore, the well-leveled load-balance is necessary to attain high performance on parallel processing decreasing the idle processors.

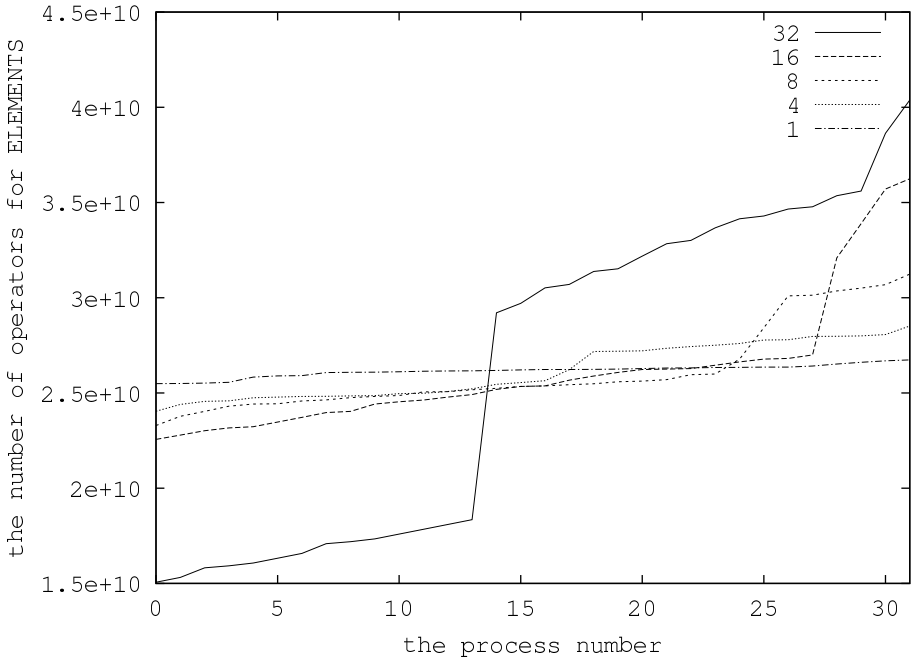
The conclusion in this subsection is that we should fix $sb = 1$ in SDPARA to achieve high performance in full-scale numerical experiments.

3.4.2 Conjugate Gradient Method for the Schur Complement Equation

After constructing the Schur complement matrix $\boldsymbol{B}$, a subsequent task is to solve the Schur complement equation $\boldsymbol{B}d\boldsymbol{z} = \boldsymbol{r}$ to obtain the dual component $d\boldsymbol{z}$ of the search direction. Since it is well-known that the Schur complement matrix $\boldsymbol{B}$ is always positive definite through all the iterations of PD-IPM [20], we immediately hit upon the idea of the Cholesky Factorization as a direct method and *Conjugate Gradient Method* (CG method) as an iterative method. So far, many researches have been devoted to incorporate CG method with PD-IPM [60, 79]. However, since their numerical results were done on a single processor, CG method on parallel processing has not been discussed. We compare between the Cholesky Factorization and CG method from the viewpoints of not only PD-IPM but also parallel processing.

Since we have already estimated the computation cost and the amount of communication of the parallel Cholesky Factorization in section 3.2.3, we focus on an estimation of CG method on the same assumption.

Figure 3.8: Effect of size of belt on operation counts on each processor (control11)



Suppose that we solve an equation $\boldsymbol{Ax} = \boldsymbol{b}$ to obtain a vector $\boldsymbol{x} \in \mathbb{R}^m$, where the coefficient matrix $\boldsymbol{A}$ is an $m \times m$ positive definite matrix. In short, we replace $\boldsymbol{B}$ by $\boldsymbol{A}$, $d\boldsymbol{z}$ by $\boldsymbol{x}$ and $\boldsymbol{r}$ by $\boldsymbol{b}$ in the Schur complement equation (2.8). In CG method, we seek to minimize the convex function

$$\phi(\boldsymbol{x}) = \frac{1}{2}(\boldsymbol{x} - \boldsymbol{A}^{-1}\boldsymbol{b})^T \boldsymbol{A}(\boldsymbol{x} - \boldsymbol{A}^{-1}\boldsymbol{b}), \tag{3.4}$$

instead of the direct Factorization of $\boldsymbol{A}$. $\phi(\boldsymbol{x})$ attains its minimum value zero if and only if $\boldsymbol{x} = \boldsymbol{A}^{-1}\boldsymbol{b}$, which stands for the solution of the equation. In CG method, we continue a line search with a steepest descendant direction $-\nabla\phi(\boldsymbol{x})$ until we reach the solution, where $\nabla\phi(\boldsymbol{x}) = \boldsymbol{Ax} - \boldsymbol{b}$ is a gradient vector of $\phi(\boldsymbol{x})$. The step length α of the line search is defined to satisfy $\frac{\partial}{\partial\alpha}\phi(\boldsymbol{x} + \alpha(-\nabla\phi(\boldsymbol{x}))) = 0$. Therefore, a simple CG method is designed as follow.

Simple CG method

Set $\boldsymbol{x} = \boldsymbol{0}$.
Compute $\nabla\phi(\boldsymbol{x}) = \boldsymbol{Ax} - \boldsymbol{b}$.
While $||\nabla\phi(\boldsymbol{x})|| \neq 0$
    Compute $\alpha = \frac{\nabla\phi(\boldsymbol{x})^T \nabla\phi(\boldsymbol{x})}{\nabla\phi(\boldsymbol{x})^T \boldsymbol{A} \nabla\phi(\boldsymbol{x})}$.
    Update $\boldsymbol{x} \leftarrow \boldsymbol{x} + \alpha(-\nabla\phi(\boldsymbol{x}))$.
    Compute $\nabla\phi(\boldsymbol{x}) = \boldsymbol{Ax} - \boldsymbol{b}$.
end(While)

Let k be the inner iteration number to terminate the while loop of CG method. We add 'inner' to distinguish from the iteration number of PD-IPM. We call the iteration number of PD-IPM 'outer' iteration number if needed. Theoretically speaking, k is bounded above by m, in particular, k is expected to be as small as possible. However, CG method sometimes can not terminate due to numerical errors. Therefore, CG method must be designed to terminate if the norm of the residual vector $\boldsymbol{r} = \boldsymbol{b} - \boldsymbol{Ax}$ is smaller than a priori precision ϵ, for example 1.0×10^{-8}, or if k exceeds a priori iteration number *maxiter*, for example

10000. Taking computation efficiency into account, we have implemented the following practical CG method on parallel processing. The algorithmic framework is based on the method described in [29].

Practical CG method

Choose parameters ϵ and $maxiter$.
Set $\boldsymbol{x} = 0, \boldsymbol{p} = \boldsymbol{0}, \boldsymbol{r} = \boldsymbol{b}, \beta = 0, \gamma = \boldsymbol{b}^T\boldsymbol{b}$, and $k = 0$.
While $\gamma > \epsilon$ and $k < maxiter$
    Update $\boldsymbol{p} \leftarrow \boldsymbol{r} + \beta\boldsymbol{p}$.
    Compute $\boldsymbol{q} = \boldsymbol{A}\boldsymbol{p}$.
    Compute $\alpha = \gamma/(\boldsymbol{p}^T\boldsymbol{q})$.
    Update $\boldsymbol{x} \leftarrow \boldsymbol{x} + \alpha\boldsymbol{p}$.
    Update $\boldsymbol{r} \leftarrow \boldsymbol{r} - \alpha\boldsymbol{q}$.
    Compute $\delta = \boldsymbol{r}^T\boldsymbol{r}$.
    Compute $\beta = \delta/\gamma$.
    Update $\gamma \leftarrow \delta$.
end(While)

In CG method, the heaviest computation concentrates on multiplications between a matrix and a vector. In the simple CG method, two multiplications are required. We can reduce the multiplication to only once in the practical CG method by the auxiliary vectors $\boldsymbol{p}, \boldsymbol{q}, \boldsymbol{r}$.

Regarding memory space over distributed memory space, the Schur complement matrix $\boldsymbol{A}$ which may become extremely large should be stored on Two-dimensional Block-Cyclic Distribution (TD-BCD) in the same way as the parallel Cholesky Factorization, because the memory space is divided equally into each processor. In addition, TD-BCD is adequate to attain high performance of multiplication with vector $\boldsymbol{q}$ provided that the vectors $\boldsymbol{x}, \boldsymbol{p}, \boldsymbol{q}, \boldsymbol{r}$ are also stored in the style of TD-BCD. The matrix $\boldsymbol{A}$ is distributed over all processors, however, $\boldsymbol{x}, \boldsymbol{p}, \boldsymbol{q}, \boldsymbol{r}$ are not over all processors. Suppose $P \times Q = N$ processors are available and we attach the process name on each processor, $P(0,0), P(0,1), \ldots, P(0,Q-1), P(1,0), P(1,1), \ldots, P(1,Q-1), \ldots, P(P-1,0), P(P-1,1), \ldots, P(P-1,Q-1)$. Following the style of TD-BCD described in section 3.2.3, a vector regarded as an $m \times 1$ matrix is usually stored in the processors on the first column, $P(0,0), P(1,0), \ldots, P(P-1,0)$. Since all the vectors $\boldsymbol{x}, \boldsymbol{p}, \boldsymbol{q}, \boldsymbol{r}$ are stored in the same distribution, the additions between two vectors, $\boldsymbol{p} \leftarrow \boldsymbol{r} + \beta\boldsymbol{p}, \boldsymbol{x} \leftarrow \boldsymbol{x} + \alpha\boldsymbol{p}, \boldsymbol{r} \leftarrow \boldsymbol{r} - \alpha\boldsymbol{p}$, can be done with only broadcasts of the scalar value α, β. The inner-products $\boldsymbol{p}^T\boldsymbol{q}, \boldsymbol{r}^T\boldsymbol{r}$ also require only cheap communication. Therefore, the most portions of the communication as well as the computation cost are occupied by the multiplication $\boldsymbol{A}\boldsymbol{p}$.

To estimate on the same assumption as the Cholesky Factorization in section 3.2.3, we assume that $N = P^2$ processors are available, the block size for column and row are identical $mb = nb$, and each communication is divided into mb size transmissions and each transmission requires an overhead h on each transmission. Furthermore, for plainness, suppose that m is a multiple number of mb.

Since one iteration of CG iteration is comprised of one multiplications between a matrix and a vector, three additions between two vectors, two inner-products between two vectors, and broadcasts of scalar value γ, we estimate each part of computation cost and amount of communication on each processor and summarize in Table 3.8. The computation cost is counted by the number of scalar multiplication.

Table 3.8: Estimation of computation cost and communication in one iteration of Conjugate Gradient method on each processor

	Computation Cost	Communication
multiplication	m^2/P^2	$2m + m \times P$
addition	m/P	P
inner-product	m/P	$2 \times P$
broadcast	0	$2 \times P$

Now, we can roughly compare the total estimation of computation cost and amount of communication

in primary term on each processor with the overhead h between the Cholesky Factorization and CG method in Table 3.9, where k is the inner iteration number required by CG method.

Table 3.9: Estimation of computation cost and communication of the Cholesky Factorization and Conjugate Gradient method on each processor

Cholesky	Computation Cost	$\frac{1}{3} \times m^3/P^2$
	Communication	$m^2 \times (1 + (m/mb) \times h)$
CG method	Computation Cost	$2 \times k \times m^2/P^2$
	Communication	$2 \times k \times (m \times (P+2) \times (1 + (m/mb) \times h))$

CG method must be called twice in each outer iteration for the predictor and the corrector search directions, hence the multiplier 2 is considered in Table 3.9. From the estimation in Table 3.9, the computation cost of CG method is smaller than the Cholesky Factorization when the iteration number k is bounded by $k \leq m/6$. In addition, since the number of available processors is at most 64 in our numerical environments, we have $P = 8$ at most. Therefore, the condition $k \leq m/20$ is sufficient for the communication of CG method to be smaller than that of the Cholesky Factorization. Hence, it is apparent that the comparison is determined by the iteration number of k.

We have applied CG method to control11 replacing the Cholesky Factorization in PD-IPM to count actual iteration counts k. In Figure 3.9, the horizontal axis indicates the outer iteration count of PD-IPM, and the vertical axis indicates the inner iteration k required at each outer iteration. While we need to solve the Schur complement twice for the predictor and the corrector search directions, the iteration counts only for the corrector directions are plotted in Figure 3.9. In control11, we have $m = 1596$ equality constraints. Therefore, k should be bounded by 266 in the computation cost or by 79 in the communication for CG method to be superior to the Cholesky Factorization. However, the threshold is easily broken when the outer iteration number is greater than 12. In particular, after 15 outer iteration, the inner iteration counts reach the pre-set maximum to terminate CG method even if a enough convergence is not attained. The super-abundant inner iterations are conductive to expensive computation costs.

Table 3.10 shows the time regarding CHOLESKY components when we apply SDPARA to control11 and theta6 changing the number of processors. Note that the implementation of CHOLESKY components are replaced by CG method in the row of 'cg'. CG method requires tens times computation time of the Cholesky Factorization. Furthermore, we can make a guess based on the above estimation and the super abundant iteration number that the total communication of CG method far exceeds that of the Cholesky Factorization.

Generally speaking, the iteration counts of CG method depend on a condition number of the coefficient matrix. More precisely, a convergence rate of the sequence $\{\boldsymbol{x}_k : k = 1, 2, \ldots\}$ generated by the practical CG method to the solution can be estimated as

$$\left|\left|\boldsymbol{x}_k - \boldsymbol{A}^{-1}\boldsymbol{b}\right|\right|_{\boldsymbol{A}} \quad \leq \quad 2\left|\left|\boldsymbol{A}^{-1}\boldsymbol{b}\right|\right|_{\boldsymbol{A}} \left(\frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}\right)^k,$$

where $||\boldsymbol{x}||_{\boldsymbol{A}} = \sqrt{\boldsymbol{x}^T\boldsymbol{A}\boldsymbol{x}}$ and κ is a condition number of $\boldsymbol{A}$. Therefore, a small condition number assures a rapid convergence to the solution. See [29] for more details.

However, it is well-known that the condition number of the Schur complement matrix remarkably grows up when the outer iteration count increases. Until now, many approach has been developed to bound the growth of the condition number. The preprocessing techniques are familiar methods, for example, diagonal scaling [90] and incomplete Cholesky Factorization [50, 51]. A general framework for pre-processing from the viewpoints of incomplete orthogonalization was discussed in [89]. Recently, a division method based on eigenvalues of the Schur complement matrix was proposed by Toh [78] . However, these methods are too complicated to implement on distributed memory. Therefore, we can not now prevent CG method from the explosive iteration number due to the large condition number.

Nevertheless, CG method is not always inferior to the Cholesky Factorization from all standpoints. Parallel processing brings out a point at which CG method beats the the Cholesky Factorization, that is, scalability. Figure 3.10 shows the scalability computed from Table 3.10. In the small number of processors,

Figure 3.9: Inner iteration number of Conjugate Gradient method (control11)

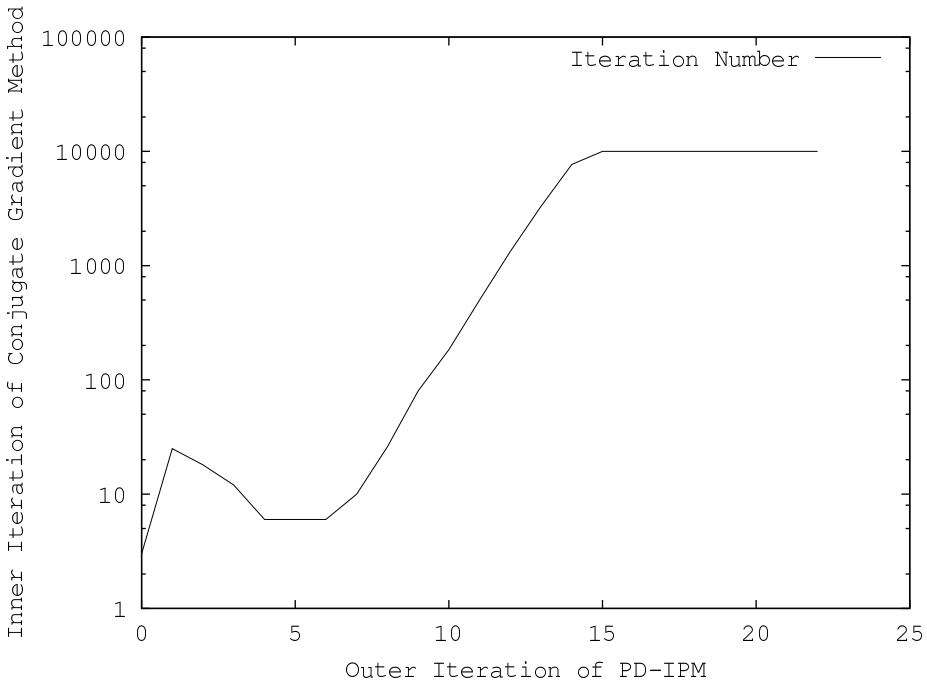


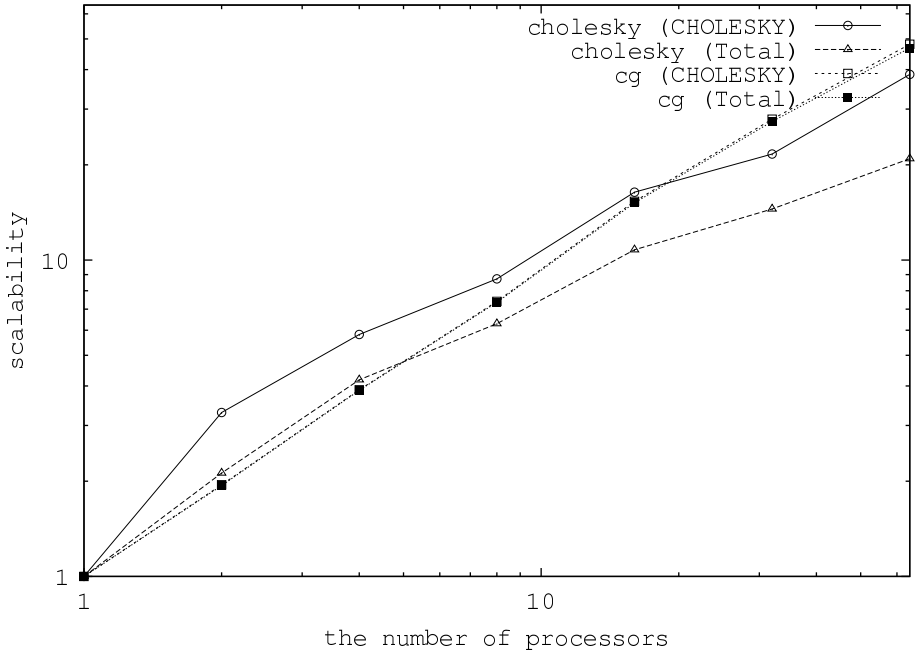
Table 3.10: Time for the Cholesky Factorization ('cholesky' row) and Conjugate Gradient method ('cg' row)

the number of processors		1	2	4	8	16	32	64
control11	cholesky	67	30	18	14	8	7	4
CHOLESKY	cg	17111	9554	4122	2412	1450	997	564
control11	cholesky	698	368	200	113	65	43	28
Total	cg	17734	9907	4298	2510	1507	1032	593
theta6	cholesky	541	164	93	62	33	25	14
CHOLESKY	cg	30995	15903	7968	4187	2022	1112	644
theta6	cholesky	711	335	170	113	66	49	34
Total	cg	31193	16104	8054	4243	2057	1138	666

the scalability of the Cholesky Factorization is higher than CG method. However, the situation becomes reverse when the number of available processors increases. The reason why the Cholesky Factorization can not attain higher scalability comes from the structure of the block-oriented Cholesky Factorization mentioned in section 3.2.3. In particular, the factorization in the most right and the lowest block which is the last block to be factorized is done on only one processor and other processors are idle. Therefore the well load-balance is difficult to attain by the Cholesky Factorization. On the other hand, in CG method, the similar situation does not occur, because the most of computation cost is occupied in the multiplication between a matrix and a vector and the computation of the multiplication without block-oriented can be equally distributed on all the processors.

From the above comparison, the conclusion toward full-scale numerical results is that we should adopt the Cholesky Factorization as the current implementation of SDPARA because of the small computation cost. However, it must not be neglected that CG method is advantageous when the higher scalability is necessary without regarding total computation

Figure 3.10: Scalability of the Cholesky Factorization and Conjugate Gradient Method (theta6)



3.4.3 Effect of Network Environment on Primal-Dual Interior-Point Methods

In this subsection, we investigate an effect of the physical network environments on SDPARA, in particular, on the time of consuming components, ELEMENTS and CHOLESKY. We have applied SDPARA to control11 and theta6 on the three PC-clusters, Presto I, sdpa and Presto III. The network environments of the PC-clusters are Ethernet, Gigabit-Ethernet, Myrinet, respectively. More detail specs of the PC-clusters have already been described in section 3.3. Table 3.11 shows the numerical results regarding ELEMENTS and CHOLESKY and Total time changing the number of processors. On Presto I, we can not solve control11 with 4 processors because of some communication error and theta6 with 1 processor owing to lack of memory space. In this subsection, we can fix the size of belt $sb = 1$ from the results of the previous numerical experiments.

As we have pointed out in section 3.2.2, ELEMENTS can be computed without any communication between multiple processors due to the excellent advantages of the row-wise distribution. Therefore, SDPARA attains almost linear scalability independently from the network environment.

However, CHOLESKY requires a lot of communication over Two-dimensional block-cyclic distribution (TD-BCD). Figure 3.11 shows the scalability of CHOLESKY on each cluster. Presto III which has the fastest network in the three PC-clusters does not lose the scalability even when we increase the number of processors up to 32. Over 2 Giga bps capacity generated by Myrinet plays an essential role to achieve the excellent scalability. On the other hand, Ethernet on Presto I can not retain the scalability on 32 processors. It means that the overheads on Ethernet can not be covered by the power of parallel processing.

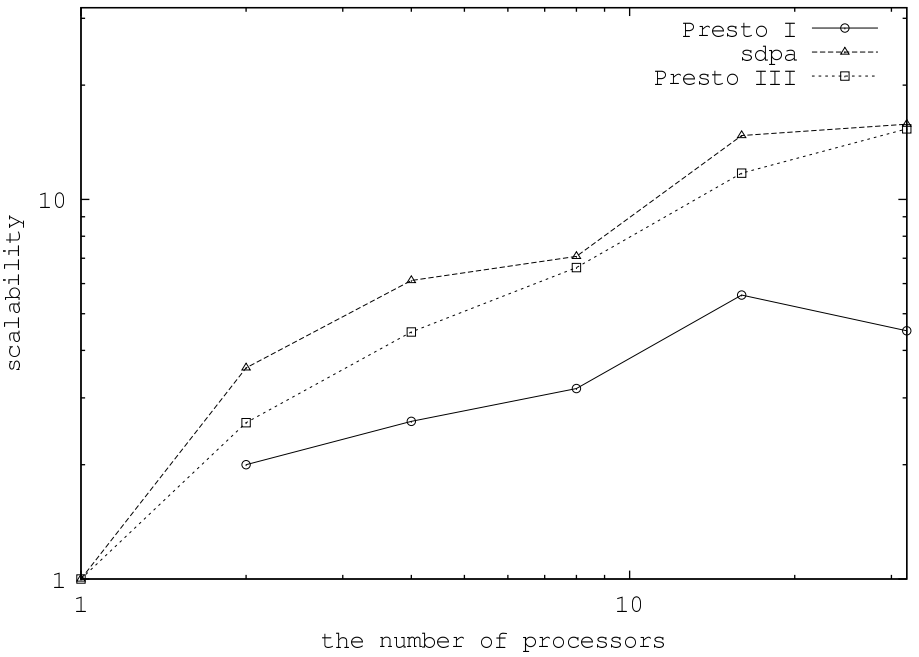
More detail of Figure 3.11 lets us know that the performance of CHOLESKY on Presto III is better when the number of processors is square of some integer. The slops regarding increment of the number of available processors of $2 \rightarrow 4$ and $8 \rightarrow 16$ are steeper than that of $4 \rightarrow 8$ and $16 \rightarrow 32$. The reason comes from the structure of TD-BCD on which the parallel Cholesky Factorization based. TD-BCD can attain its best performance when we have square processor grid, that is, $P = Q = \sqrt{N}$ as described in section 3.2.3. However, the performance on sdpa-cluster has little relevant to the characteristic of the TD-BCD structure. It is possible to consider the reason is that the network library of sdpa-cluster is under SCore library. SCore may affect the total communication over multiple processors.

Let us summarize the effect of the network environment for SDPARA. SDPARA has two parallel compo-

Table 3.11: Performance of SDPARA on each cluster

the number of processors		1	2	4	8	16	32
control11	Presto I	1143.6	569.6	*	140.3	69.3	34.4
ELEMENTS	sdpa	683.6	341.6	169.8	84.9	41.9	20.7
	Presto III	603.4	293.4	146.8	73.6	35.9	17.9
control11	Presto I	150.4	106.5	*	94.4	69.0	84.9
CHOLESKY	sdpa	92.4	38.3	24.3	23.7	13.9	13.8
	Presto III	54.5	29.2	18.7	15.4	10.1	9.1
control11	Presto I	1342.2	816.9	*	329.4	205.3	182.6
Total	sdpa	809.3	761.6	239.4	142.3	78.19	55.3
	Presto III	685.3	363.1	195.0	112.1	66.6	42.9
theta6	Presto I	*	65.8	30.3	14.3	7.0	3.5
ELEMENTS	sdpa	137.7	62.4	29.8	14.8	7.16	3.5
	Presto III	166.0	102.6	60.3	35.5	18.6	9.4
theta6	Presto I	*	533.4	409.9	336.1	190.6	236.8
CHOLESKY	sdpa	760.3	211.3	124.4	107.4	51.6	48.2
	Presto III	417.3	161.7	93.3	63.1	35.6	27.2
theta6	Presto I	*	819.4	613.9	458.9	279.3	314.7
Total	sdpa	953.7	398.2	212.5	165.0	87.2	75.2
	Presto III	600.6	339.9	166.9	111.4	66.7	49.1

Figure 3.11: Scalability on each cluster (theta6,CHOLESKY)



nents, ELEMENTS and CHOLESKY. ELEMENTS can be done without any communication, therefore the scalability of ELEMENTS is independent from the network environments. On the other hand, CHOLESKY requires a lot of communication on TD-BCD. Thus, CHOLESKY is strongly affected by the network environments. From the results for components of PD-IPM, we can make sure that the high capacity network plays a significant role to attain excellent scalability in total computation time for SDPARA as well as other general parallel software. It is important from the viewpoints of parallel processing to explore whether the

parallel implementation quite depends on the network capacity or not.

3.5 Numerical Results

In this section, we report full-scale numerical results of SDPARA. The SDPs that we tested are divided into the following two types. The SDPs of the first type are selected from SDPLIB benchmark problems [11], while the other type are SDPs arisen from quantum chemistry. Almost all our numerical experiments in this paper were mainly executed on Presto III, unless we indicate.

3.5.1 SDPs from SDPLIB

We selected control10,11, theta5,6 , thetaG51 and maxG51 from SDPLIB. Three of them were also picked up in section 3.2.1. Their sizes are shown in Table 3.12. The problems control10 and control11 are Control Problems and they are the largest problems of this type in SDPLIB, while theta5, theta6 and thetaG51 are Theta Function Problems. In particular, thetaG51 requires the most computation time among all problems of SDPLIB. The problem maxG51 is a Max Cut Problem. The formulations of these SDPs have been described in section 3.2.1. The numerical results on SDPARA applied to these problems are shown in Table 3.13 in form of required time in second for the components of PD-IPM categorized in section 3.2.1. In Table 3.12, m is the number of equality constraints of $\mathcal{P}$, nBLOCK and bLOCKsTRUCT define the structure of $\boldsymbol{X}$ and $\boldsymbol{Y}$. In Table 3.13, '*' indicates lack of memory, and we skip the components which can be computed in less than 20 seconds even by a single processor.

Table 3.12: SDPs picked up from SDPLIB

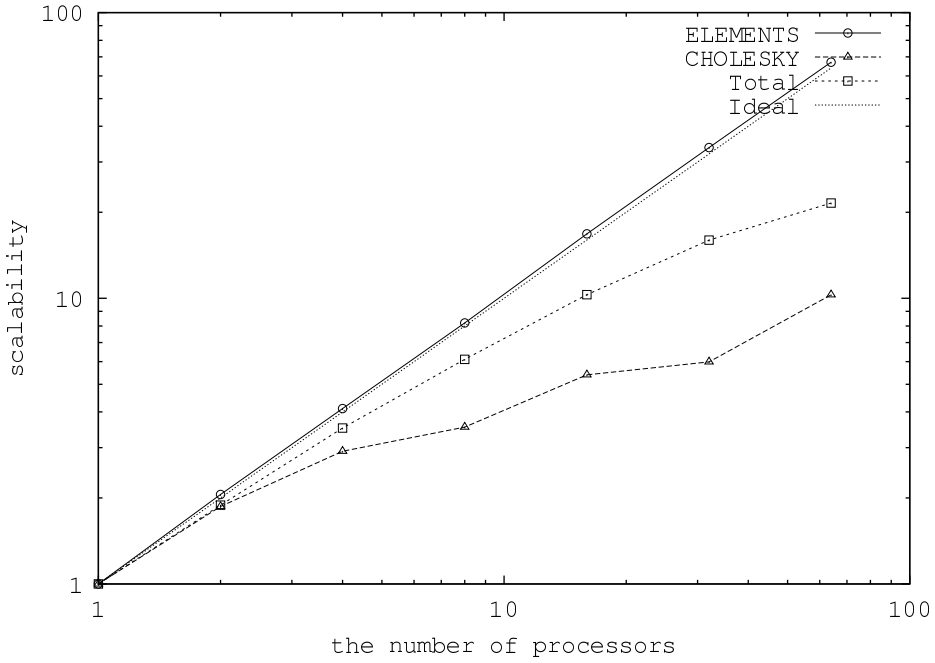
name	m	nBLOCK	bLOCKsTRUCT
control10	1326	2	(100,50)
control11	1596	2	(110,55)
theta5	3028	1	(250)
theta6	4375	1	(300)
thetaG51	6910	1	(1001)
maxG51	1000	1	(1000)

In control10 and control11, the most of the computation time is spent in ELEMENTS (the computation of the elements of the Schur complement matrix $\boldsymbol{B}$). We observe an excellent scalability, the ratio of real time to solve a problem with respect to the number of processor used, especially in ELEMENTS, as illustrated in Figure 3.12. For example, SDPARA with 8 processors solved control11 6.1 times faster than SDPARA with a single processor, and the case with 64 processors solved the problem 22 times faster than the case with a single processor, respectively. The $m \times m$ Schur complement matrix is always a fully dense and its Cholesky Factorization does not depend on the block structure of the test problem described as nBLOCK (the number of blocks) and bLOCKsTRUCT (the block diagonal structure) in Table 3.2. Although the block structure and the sparsity of $\boldsymbol{X}$, $\boldsymbol{Y}$ and $\boldsymbol{A}_i$ are effectively utilized in the multiplication and inner products, they do little affect to the scalability of ELEMENTS. The scalability sometimes exceeds the ratio of the number of processors. This unusual phenomenon happened probably because as we increased the processors the memory space for each processor to access decreased so that the access speed to memory became faster.

In theta5, theta6 and thetaG51, most of the computation time are spent for CHOLESKY (the parallel Cholesky Factorization of the Schur complement matrix $\boldsymbol{B}$). We observe again high scalability in the numerical results on these problems. For instance, SDPARA with 8 processors solved theta6 5.3 times faster than SDPARA with a single processor, and the case with 64 processors solved the problem 15 times faster than the case with a single processor, respectively.

In contrast to the excellent scalability described above for Control Problems and Theta Function Problems, it is unfortunate that SDPARA can not attain any scalability for the Max Cut Problem, maxG51. As mentioned in the implementation of SDPARA, we have replaced ELEMENTS and CHOLESKY by their parallel implementation. However, the most consuming components when we executed SDPARA on

Figure 3.12: Scalability for Control11



maxG51 are PMATRIX and DENSE. We will investigate parallel implementation for PMATRIX component in chapter 4 with the completion method.

3.5.2 SDPs arisen from Quantum Chemistry

To examine the performance of SDPARA more widely, we employ SDPs arisen from quantum chemistry. Here, we take a look at the formulation of the SDPs to grab their characteristic.

In the field of quantum chemistry, the ground-state energy has been a meaningful subject over many decades. In short, the ground-state energy can be considered as system energy in the most stable state of a molecule. The ground-state energy state plays an essential role to estimate quantity of heat caused by chemical reaction. So far, Hartree-Fock approximation and Full Configuration-Interaction were developed to obtain the ground-state energy See [68] by Szabo and Ostlund for more details.

In 1955, Mayer [54] first pointed out that the system energy can be fundamentally described by the effect between two electrons, that is, two-body reduced density matrix (2-RDM). Then, in 1963, a concept of N-representability was introduced by Coleman [17] to represent the state in not only one electron but also multiple electrons, N electrons. A remarkable study in Coleman's accomplishment was he indicated that N-representability lies on a strong relation with SemiDefinite Programming. At that time, however, a powerful algorithm such as PD-IPM had not been developed yet. Poor algorithms could not carry out the expensive computation cost of the SDP formulations in N-representability. Thirty years later, in 2000, practical scale numerical results in descriptions of 2-RDM have been reported in Nakata, *et al.* [61, 62] based on the significant developments of PD-IPM and its implementation, SDPA [23]. Furthermore, Zhao, *et al.* [91] successfully reduced the order of the equality constraints m in the SDP formulation. In [61, 62], the primal SDP formulation of is adopted to describe the conditions of quantum chemistry, while Zhao, *et al.* adopts dual SDP formulation in [91]. In addition, portions of 3-RDM are taken into consideration in [91] so that an optimal value generated by an SDP will become closer to the ground-state energy. Here, we follow the SDP formulation discussed in [61, 62, 91].

We start from a single-electron wave function $\psi(\boldsymbol{r})$, where $\boldsymbol{r}$ is the location of the electron in three real dimension $\mathbb{R}^3$. Though a single-electron wave function must include a condition with respect to spins of electrons, the discussion here is advanced without regarding the spin; we assume that the wave functions

Table 3.13: Performance of SDPARA on multiple processors

the number of processors		1	2	4	8	16	32	64
control10	ELEMENTS	400.5	208.2	108.3	56.3	27.5	14.0	7.5
	CHOLESKY	31.8	18.2	12.3	10.2	6.5	6.1	3.7
	Total	440.8	233.9	127.8	73.5	41.0	27.3	19.8
control11	ELEMENTS	603.4	293.4	146.8	73.6	35.9	17.9	9.0
	CHOLESKY	54.5	29.2	18.7	15.4	10.1	9.1	5.3
	Total	685.3	363.1	195.0	112.1	66.6	42.9	31.8
theta5	ELEMENTS	72.4	46.6	27.6	16.7	8.7	4.4	2.5
	CHOLESKY	140.8	58.8	36.0	26.0	14.9	12.5	6.9
	Total	222.5	113.2	70.8	49.7	30.6	24.0	16.7
theta6	ELEMENTS	166.0	102.6	60.3	35.5	18.6	9.4	5.5
	CHOLESKY	417.3	161.7	93.3	63.1	35.6	27.2	17.3
	Total	600.6	339.9	166.9	111.4	66.7	49.1	35.5
thetaG51	ELEMENTS	*	1627.1	473.4	265.9	131.5	68.0	36.6
	CHOLESKY	*	962.3	557.2	334.8	181.5	125.4	75.9
	PMATRIX	*	145.0	101.4	101.4	101.4	101.2	101.3
	DENSE	*	204.4	184.5	184.9	184.6	184.3	185.0
	Total	*	3786.5	1344.3	911.9	624.9	505.8	424.1
maxG51	PMATRIX	55.5	55.9	55.5	55.6	55.7	38.4	35.0
	DENSE	108.8	109.5	109.0	109.3	109.2	77.6	69.8
	Total	176.7	178.4	176.2	188.1	183.4	129.9	161.7

$\psi_i(\boldsymbol{r})$ $(i = 1, 2, \ldots, N)$ have already included spin information.

A fundamental constraint which $\psi(\boldsymbol{r})$ must satisfy is *the Schrödinger equation*,

$$\mathcal{H}\psi(\boldsymbol{r}) = \epsilon\psi(\boldsymbol{r}).$$

The system energy ϵ and $\psi(\boldsymbol{r})$ can be regarded as an eigenvalue and an eigen function with respect to the Hamiltonian operator $\mathcal{H}$ which includes a term for the kinetic energy and a term for the potential energy received from protons. Furthermore, $\psi(\boldsymbol{r})$ needs to be normalized, because $\psi(\boldsymbol{r})$ includes just one electron in all space $\mathbb{R}^3$.

$$\int \psi(\boldsymbol{r})^* \psi(\boldsymbol{r}) d\boldsymbol{r} = 1. \tag{3.5}$$

We use '*' to denote the conjugate value (to be consistent in spin of each electron), and omit an explicit integral region, since it is always $\mathbb{R}^3$. Note that we integrate in three dimension; to be precise, the integration is

$$\int \psi(\boldsymbol{r}) d\boldsymbol{r} = \int_{\mathbb{R}}\int_{\mathbb{R}}\int_{\mathbb{R}} \psi(\boldsymbol{r}_x, \boldsymbol{r}_y, \boldsymbol{r}_z) d\boldsymbol{r}_x d\boldsymbol{r}_y d\boldsymbol{r}_z.$$

The normalization (3.5) gives us another viewpoints of the system energy,

$$\epsilon = \epsilon \int \psi(\boldsymbol{r})^* \psi(\boldsymbol{r}) d\boldsymbol{r} = \int \psi(\boldsymbol{r})^* \mathcal{H}\psi(\boldsymbol{r}) d\boldsymbol{r}$$

Since the smallest eigenvalue corresponds to the ground-state energy in the system, the following minimization problem will give us the ground-state energy.

$$\begin{array}{lll} \min & \epsilon = & \int \psi(\boldsymbol{r})^* \mathcal{H}\psi(\boldsymbol{r}) d\boldsymbol{r} \\ \text{subject to} & & \int \psi(\boldsymbol{r})^* \psi(\boldsymbol{r}) d\boldsymbol{r} = 1. \end{array} \tag{3.6}$$

Until now, the subject matter is only single-electron; thus the above argument is further simple. However, the molecules whose ground-state energy is our interest have multiple electrons, for example LiF, NH_2, therefore the situations become very complicated, in particular, because of *the Pauli's exclusion principle*.

Let N be the number of electrons in a molecule. Suppose that we have single-electron wave functions $\psi_i(\boldsymbol{r})$ $(i = 1, 2, \ldots, N)$. It is common that an N-electron wave function $\psi(\boldsymbol{r}_1, \boldsymbol{r}_2, \ldots, \boldsymbol{r}_N)$ is composed in the form of *the Slater determinant* to satisfy the Pauli's principle.

$$\psi(\boldsymbol{r}_1, \boldsymbol{r}_2, \ldots, \boldsymbol{r}_N) = \frac{1}{\sqrt{N!}} det \begin{pmatrix} \psi_1(\boldsymbol{r}_1) & \psi_2(\boldsymbol{r}_1) & \cdots & \psi_N(\boldsymbol{r}_1) \\ \psi_1(\boldsymbol{r}_2) & \psi_2(\boldsymbol{r}_2) & \cdots & \psi_N(\boldsymbol{r}_2) \\ \vdots & \vdots & \ddots & \vdots \\ \psi_1(\boldsymbol{r}_N) & \psi_2(\boldsymbol{r}_N) & \cdots & \psi_N(\boldsymbol{r}_N) \end{pmatrix}.$$

The factor $\sqrt{N!}$ is required for ψ to be normalized (3.5). The Slater determinant ensures that any wave function is independent from each other ($\psi_i \neq \psi_j$ if $i \neq j$) and no electron exists on the two wave functions ($\boldsymbol{r}_i \neq \boldsymbol{r}_j$ if $i \neq j$) simultaneously.

When we have a normalized N-electron wave function $\psi(\boldsymbol{r}_1, \boldsymbol{r}_2, \ldots, \boldsymbol{r}_N)$, we can regard it as a density function. More precisely,

$$\psi(\boldsymbol{r}_1, \boldsymbol{r}_2, \ldots, \boldsymbol{r}_N)\psi^*(\boldsymbol{r}_1, \boldsymbol{r}_2, \ldots, \boldsymbol{r}_N)d\boldsymbol{r}_1 d\boldsymbol{r}_2 \ldots d\boldsymbol{r}_N$$

is the possibility that the 1st electron resides the cube centered $\boldsymbol{r}_1$ with the width $d\boldsymbol{r}_1$ on a side, the 2nd electron resides the cube centered $\boldsymbol{r}_2$ with the width $d\boldsymbol{r}_2$ on a side, ..., and the Nth electron resides the cube centered $\boldsymbol{r}_N$ with the width $d\boldsymbol{r}_N$ on a side.

If our focus on the 1st electron, the *1st-order Reduced Density Matrix (1-RDM)*, γ, has advantageous to shed light on the relation for SemiDefinite Programming.

$$\gamma(\boldsymbol{r}_1, \boldsymbol{r}'_1) = N \int \psi(\boldsymbol{r}_1, \boldsymbol{r}_2, \ldots, \boldsymbol{r}_N)\psi^*(\boldsymbol{r}'_1, \boldsymbol{r}_2, \ldots, \boldsymbol{r}_N)d\boldsymbol{r}_1 d\boldsymbol{r}_2 \ldots d\boldsymbol{r}_N.$$

The factor N is required so that the integration over $d\boldsymbol{r}_1$ with $\boldsymbol{r}_1 = \boldsymbol{r}'_1$,

$$\int [\gamma(\boldsymbol{r}_1 \boldsymbol{r}'_1)]_{\boldsymbol{r}'_1 = \boldsymbol{r}_1} d\boldsymbol{r}_1 = \int \rho(\boldsymbol{r}_1) d\boldsymbol{r}_1,$$

becomes the electron number N, where

$$\rho(\boldsymbol{r}_1) = N \int \psi(\boldsymbol{r}_1, \boldsymbol{r}_2, \ldots, \boldsymbol{r}_N)\psi^*(\boldsymbol{r}_1, \boldsymbol{r}_2, \ldots, \boldsymbol{r}_N)d\boldsymbol{r}_1 d\boldsymbol{r}_2 \ldots d\boldsymbol{r}_N$$

is called *the reduced density function* for single-electron. Note that 1-RDM is a generalization of the reduced density function.

Since the 1-RDM is a continuous matrix, we construct a discrete representation with single-electron base wave functions $\phi_1(\boldsymbol{r}), \phi_2(\boldsymbol{r}), \ldots, \phi_K(\boldsymbol{r})$,

$$\gamma_{ij} = \int \phi_i^*(\boldsymbol{r}_1)\gamma(\boldsymbol{r}_1, \boldsymbol{r}'_1)\phi_j(\boldsymbol{r}'_1)d\boldsymbol{r}_1 d\boldsymbol{r}'_1.$$

Conversely, if the set of the base wave functions is complete,

$$\gamma(\boldsymbol{r}_1, \boldsymbol{r}'_1) = \Sigma_{ij}\phi_i(\boldsymbol{r}_1)\gamma_{ij}\phi_j^*(\boldsymbol{r}'_1).$$

Although many attempts have been made at the base wave functions $\phi_i(\boldsymbol{r})(i = 1, 2, \ldots, K)$, we will not take them up in detail here as space is limited. STO-3G and STO-6G are common base wave functions sets and we have selected STO-6G. See [68] for more base wave functions sets.

Now, we examine a reformulation of the above minimization problem to obtain the ground-state energy with the discrete representation of 1-RDM. What we have to recognize here is that the Hamiltonian operator $\mathcal{H}$ must be modified to include reactions between two electrons.

$$\mathcal{H} = \Sigma_{i=1}^N \mathcal{H}_i^1 + \frac{1}{2}\Sigma_{i=1}^N \Sigma_{j=1}^N \mathcal{H}_{ij}^2$$

$\mathcal{H}_i^1$ is the Hamiltonian operator with respect to the ith electron for its kinetic power and its potential energy from protons, and $\mathcal{H}_{ij}^2$ stands for the reaction between two electrons (the ith and the jth). For instance,

coulomb force is a strong effect between two electrons. Since we focus on 1-RDM here, we first ignore $\mathcal{H}^2_{ij}$ here, then we back at $\mathcal{H}^2_{ij}$.

First, the objective function in (3.6) will be described in 1-RDM.

$$\begin{aligned} \epsilon &= \int \psi^*(\boldsymbol{r}_1, \boldsymbol{r}_2, \ldots, \boldsymbol{r}_N)\mathcal{H}\psi(\boldsymbol{r}_1, \boldsymbol{r}_2, \ldots, \boldsymbol{r}_N)d\boldsymbol{r}_1 d\boldsymbol{r}_2 \ldots d\boldsymbol{r}_N \\ &= \boldsymbol{h} \bullet \gamma, \end{aligned}$$

where the matrix $\boldsymbol{h}$ is composed of elements $h_{ij} = \int \phi_i^*(\boldsymbol{r}_1)\mathcal{H}_1^1\phi_j(\boldsymbol{r}_1)d\boldsymbol{r}_1$. Since we select STO-6G as the base wave function sets, the matrix $\boldsymbol{h}$ can be calculated by Gaussian98 [19]. Therefore, we can assume that $\boldsymbol{h}$ is given after the selection of the base wave function sets.

Furthermore, the integral over all space in three dimensions must coincide with the total number of electrons N.

$$\begin{aligned} N &= N\int \psi^*(\boldsymbol{r}_1, \boldsymbol{r}_2, \ldots, \boldsymbol{r}_N)\psi(\boldsymbol{r}_1, \boldsymbol{r}_2, \ldots, \boldsymbol{r}_N)d\boldsymbol{r}_1 d\boldsymbol{r}_2 \ldots d\boldsymbol{r}_N \\ &= \Sigma_i \gamma_{ii} = \boldsymbol{I} \bullet \gamma. \end{aligned}$$

Finally, any single-electron $\psi(\boldsymbol{r})$ wave function has the possibility of electron existence in the range from 0 to 1. If $\psi(\boldsymbol{r})$ is decomposed into a linear combination of the basic wave functions, $\psi(\boldsymbol{r}) = \Sigma_i x_i \phi_i(\boldsymbol{r})$, the condition will be replaced by the positive semidefiniteness of 1-RDM. The possibility that the 1st electron resides in $\psi(\boldsymbol{r})$ is

$$\int \psi^*(\boldsymbol{r}_1)\gamma(\boldsymbol{r}_1, \boldsymbol{r}_1')\psi(\boldsymbol{r}_1')d\boldsymbol{r}_1 d\boldsymbol{r}_1' = \Sigma_{ij} x_i^* \gamma_{ij} x_j.$$

Since the possibility for any $\boldsymbol{x}$ in the range from 0 to 1, we obtain the positive semidefinite conditions,

$$\boldsymbol{O} \preceq \gamma \preceq \boldsymbol{I}.$$

Now, the continuous minimization problem (3.6) is reduced into the SDP.

$$\begin{array}{lll} \min & \epsilon = & \boldsymbol{h} \bullet \gamma \\ \text{subject to} & & \boldsymbol{I} \bullet \gamma = N \\ & & \boldsymbol{O} \preceq \gamma \preceq \boldsymbol{I} \end{array}$$

.

As we have mentioned, however, 1-RDM is not enough to represent the coulomb force between two-electrons. Therefore, we employ 2-RDM Γ and its discrete representation.

$$\begin{aligned} \Gamma(\boldsymbol{r}_1, \boldsymbol{r}_2, \boldsymbol{r}_1', \boldsymbol{r}_2') &= N(N-1)\int \psi(\boldsymbol{r}_1, \boldsymbol{r}_2, \boldsymbol{r}_3, \ldots, \boldsymbol{r}_N)\psi^*(\boldsymbol{r}_1', \boldsymbol{r}_2', \boldsymbol{r}_3, \ldots, \boldsymbol{r}_N)d\boldsymbol{r}_3 \ldots d\boldsymbol{r}_N \\ \Gamma_{ij;kl} &= \int \frac{1}{\sqrt{2}}\left(\phi_i^*(\boldsymbol{r}_1)\phi_j^*(\boldsymbol{r}_2) - \phi_j^*(\boldsymbol{r}_1)\phi_i^*(\boldsymbol{r}_2)\right)\Gamma(\boldsymbol{r}_1, \boldsymbol{r}_2, \boldsymbol{r}_1', \boldsymbol{r}_2') \\ &\quad \frac{1}{\sqrt{2}}\left(\phi_k(\boldsymbol{r}_1')\phi_l(\boldsymbol{r}_2') - \phi_l(\boldsymbol{r}_1')\phi_k(\boldsymbol{r}_2')\right)d\boldsymbol{r}_1 d\boldsymbol{r}_2 d\boldsymbol{r}_1' d\boldsymbol{r}_2' \end{aligned}$$

Note that $\frac{1}{\sqrt{2}}\left(\phi_i(\boldsymbol{r}_1)\phi_j(\boldsymbol{r}_2) - \phi_j(\boldsymbol{r}_1)\phi_i(\boldsymbol{r}_2)\right)$ is the Slate determinant which is comprised of $\phi_i(\boldsymbol{r})$ and $\phi_j(\boldsymbol{r})$. With 2-RDM, the objective function is modified as

$$\epsilon = \boldsymbol{h} \bullet \gamma + \frac{1}{2}\boldsymbol{H} \bullet \Gamma,$$

where

$$\begin{aligned} H_{ij;kl} &= \int \frac{1}{\sqrt{2}}\left(\phi_i^*(\boldsymbol{r}_1)\phi_j^*(\boldsymbol{r}_2) - \phi_j^*(\boldsymbol{r}_1)\phi_i^*(\boldsymbol{r}_2)\right)\mathcal{H}_{12}^2 \\ &\quad \frac{1}{\sqrt{2}}\left(\phi_k(\boldsymbol{r}_1)\phi_l(\boldsymbol{r}_2) - \phi_l(\boldsymbol{r}_1)\phi_k(\boldsymbol{r}_2)\right)d\boldsymbol{r}_1 d\boldsymbol{r}_2 \end{aligned}$$

If we adopt higher-RDMs than 2-RDM, an accuracy of an optimal value to the exact ground-state energy will better. At the same time, however, the SDP becomes extremely large and we can not solve it. It is reported in [91] that 3-RDM is enough to acquire a sufficient accuracy from standpoints of quantum chemistry, in addition that 3-RDM can be expressed by linear combinations of 2-RDM elements.

The above discussions can be summarized into the formulation of SDP.

$$
\begin{array}{lll}
\min & \epsilon = & \boldsymbol{h} \bullet \gamma + \frac{1}{2}\boldsymbol{H} \bullet \Gamma \\
\text{subject to} & & \boldsymbol{I} \bullet \gamma = N \\
& & \Sigma_k \Gamma_{ik;jk} = \gamma_{ij} \quad (i = 1, \ldots, K,\ j = 1, \ldots, K) \\
& & \boldsymbol{O} \preceq \gamma \preceq \boldsymbol{I} \\
& & \boldsymbol{O} \preceq \Gamma \preceq \boldsymbol{I} \\
& & \text{and 3-RDM conditions.}
\end{array}
$$

Regarding computation cost, the above SDP grows in extremely rapid speed with respect to the number K of the basic wave functions. The number m of equality constraints and the size n of variable matrices are estimated as $\mathcal{O}(K^6)$ and $\mathcal{O}(K^4)$, respectively. Zhao, *et al.* [91] reduces them to $m = \mathcal{O}(K^4)$ and $n = \mathcal{O}(K^4)$ in significant order based on the dual formulation of SDP. Even when they utilize the reduction, the SDPs are still very large.

However, the extremely largeness of the SDPs arisen from quantum chemistry are in the right place for SDPARA. Thorough the numerical results, we will verify that SDPARA successfully solves such large SDPs.

3.5.3 Numerical Results for Quantum Chemistry

The SDPs given in Table 3.14 are from quantum chemistry [61, 62]. The characteristic of this type of SDPs is that the number m of equality constraints of $\mathcal{P}$ can be very large. In the largest problem with $m = 24503$, we need to store a 24503×24503 matrix for the Schur complement matrix $\boldsymbol{B}$ on distributed memory. The matrix requires about 9GB memory to store, so that we need at least 16 processors on Presto III to solve the problem. Table 3.15 shows the numerical results on SDPARA applied to the problems listed in Table 3.14. ('*' indicates lack of memory.) It is clear that as more processors we used, faster we solved each problem.

Table 3.14: SDPs arisen from quantum chemistry

System.Status.Basis	m	nBLOCK	bLOCKsTRUCT
$BH_3.{}^1A_1$.STO-6G	2897	2	(120,120)
$HF^+.{}^2\Pi$.STO-6G	4871	3	(66,66,144)
$NH_2.{}^2A_1$.STO-6G	8993	3	(91,91,196)
$LiF.{}^1\Sigma$.STO-6G	15313	3	(120,120,256)
$CH_4.{}^1A_1$.STO-6G	24503	3	(153,153,324)

It should be also emphasized that as the size of the problems becomes larger, SDPARA attains higher scalability; as the number of processors is increased from 8 to 64, 1/2.5 reduction of the real time to solve the smallest problem BH_3 is attained while 1/5.2 reduction is attained in the larger problem case LiF. We illustrate the scalability attained by SDPARA for LiF in Figure 3.13.

3.5.4 Load-Balance

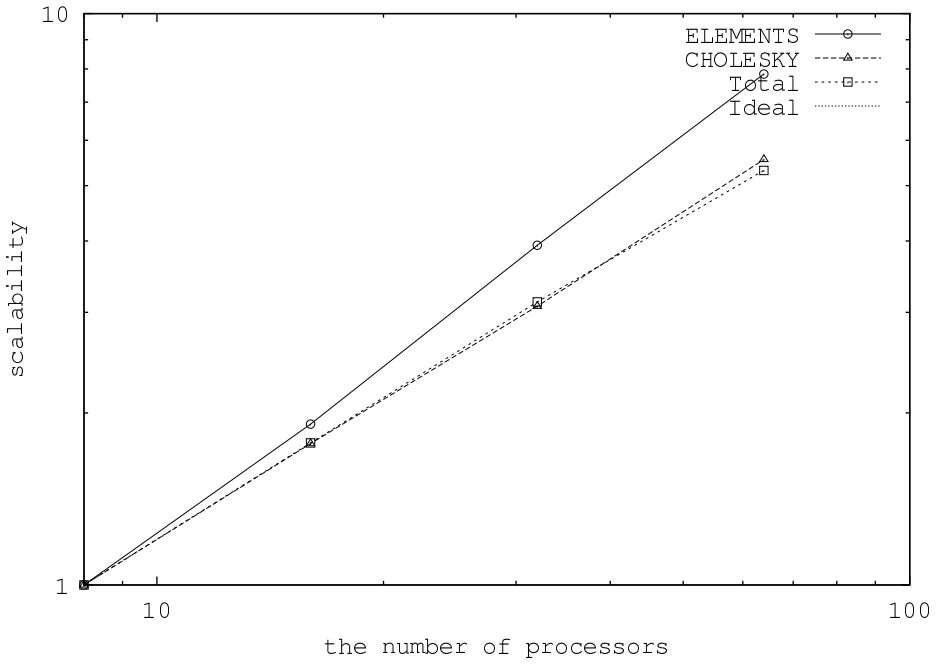
We have measured load-balance of SDPARA over 64 processors by using PAPI [52] on Presto I. Table 3.16 shows the lowest and highest CPU operation counts in 64 processors and their ratio regarding two parallel components. In addition, we post Figure 3.14 for more details of operation counts on each processor in the case of HF. (In this figure, we sort the order by operation counts.)

We observe that the ratios in Total operation counts are bounded by 1.40. Especially, in control11 and theta6, SDPARA shows excellent load-balance in ELEMENTS. Therefore we can conclude that SDPARA attains reasonable load-balance although it adopts the simple parallel implementation described in section 3.2.2.

Table 3.15: Performance of SDPARA on multiple processors for SDPs arisen from quantum chemistry

the number of processors		8	16	32	64
BH_3.$^1A_1$.STO-6G	ELEMENTS	38.7	19.1	11.3	5.4
	CHOLESKY	24.0	13.5	10.9	6.6
	Total	66.7	36.7	26.5	16.6
HF^+.$^2\Pi$.STO-6G	ELEMENTS	109.7	56.3	28.2	14.1
	CHOLESKY	86.8	49.8	33.7	19.7
	Total	210.4	120.6	76.7	48.8
NH_2.$^2A_1$.STO-6G	ELEMENTS	406.8	236.5	125.6	61.4
	CHOLESKY	452.6	289.0	185.2	108.0
	Total	908.2	573.6	358.9	219.4
LiF.$^1\Sigma$.STO-6G	ELEMENTS	1454.8	760.7	370.0	185.6
	CHOLESKY	2245.1	1270.1	730.1	404.2
	Total	3830.9	2160.8	1224.8	720.7
CH_4.$^1A_1$.STO-6G	ELEMENTS	*	1695.5	907.1	426.5
	CHOLESKY	*	4369.5	2743.3	1248.1
	Total	*	6367.7	3952.4	1980.3

Figure 3.13: Scalability for LiF



3.6 Comparison with Other Software

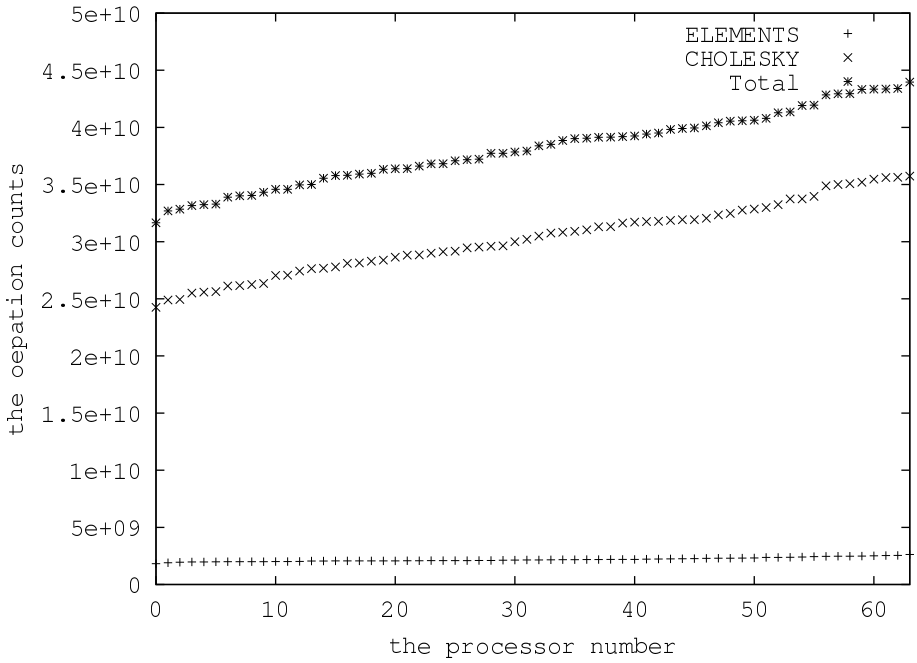
3.6.1 PDSDP

In this section, we compare the performance of our SDPARA with PDSDP [4] through some numerical results. PDSDP is a parallel version of DSDP, an SDP solver developed by Benson and Ye [5]. In our best knowledge, PDSDP had been the only parallel solver for general SDPs before we implemented SDPARA. There is a major difference between SDPARA and PDSDP. The difference lies in their algorithmic frameworks; SDPARA is based on PD-IPM with the use of the HRVW/KSH/M search direction as described in section 2, while PDSDP is based on *D-IPM* (dual-scaling interior-point methods).

Table 3.16: Load-balance of SDPARA on 64 processors

problems		lowest	highest	ratio
control11	ELEMENTS	1.24×10^{10}	1.35×10^{10}	1.09
	CHOLESKY	1.62×10^{9}	4.30×10^{9}	2.65
	Total	2.10×10^{10}	2.72×10^{10}	1.29
theta6	ELEMENTS	6.06×10^{8}	6.43×10^{8}	1.06
	CHOLESKY	1.67×10^{10}	2.68×10^{10}	1.60
	Total	4.19×10^{10}	5.21×10^{10}	1.24
$HF^{+}.^{2}\Pi$.STO-6G	ELEMENTS	1.82×10^{9}	2.62×10^{9}	1.43
	CHOLESKY	2.42×10^{10}	3.57×10^{10}	1.42
	Total	3.16×10^{10}	4.40×10^{10}	1.39

Figure 3.14: Operation count on each processor for HF



Now, we take a look at chief characteristics of D-IPM. D-IPM was proposed in [6] to overcome a drawback of PD-IPM that the primal variable matrix $\boldsymbol{X}$ is a full dense matrix in general. Conversely, the dual variable matrix $\boldsymbol{Y}$ inherits the sparsity of the input data matrices.

$$\boldsymbol{Y} = \boldsymbol{C} - \Sigma_{k=1}^{m} \boldsymbol{A}_k z_k.$$

Therefore, it is natural that we had better carry out the interior-point methods with only dual space if possible. In other words, the method may reduce considerable amount of memory space to store the dense matrix $\boldsymbol{X}$ and be capable of exploiting the sparsity of the input data matrices much directly.

In the derivation of the HRVW/KSH/M search direction in section 2.3.1, we apply the symmetrization operator $H_{\boldsymbol{P}}(\boldsymbol{X}) = (\boldsymbol{P}\boldsymbol{X}\boldsymbol{P}^{-1} + (\boldsymbol{P}\boldsymbol{X}\boldsymbol{P}^{-1})^T)/2$ to $\boldsymbol{X}d\boldsymbol{Y} + d\boldsymbol{X}\boldsymbol{Y} = \mu\boldsymbol{I} - \boldsymbol{X}\boldsymbol{Y}$ originated from $\boldsymbol{X}\boldsymbol{Y} = \mu\boldsymbol{I}$. In stead of the replacement for $\boldsymbol{X}\boldsymbol{Y} = \mu\boldsymbol{I}$, we consider an equivalent equation $\boldsymbol{X} = \mu\boldsymbol{Y}^{-1}$ in D-IPM. Hence, the central path is comprised of

$$\begin{aligned} \{\boldsymbol{X}, \boldsymbol{Y}, \boldsymbol{z} \quad : \quad & \boldsymbol{A}_k \bullet \boldsymbol{X} = b_k (k = 1, 2, \ldots, m), \Sigma_{k=1}^{m} \boldsymbol{A}_k z_k + \boldsymbol{Y} = \boldsymbol{C}, \\ & \boldsymbol{X} \succeq \boldsymbol{O}, \boldsymbol{Y} \succeq \boldsymbol{O}, \ \boldsymbol{X} = \mu \boldsymbol{Y}^{-1}\}. \end{aligned}$$

A characteristic feature of D-IPM is that we progress without the primal variable $\boldsymbol{X}$, however, the use of $\boldsymbol{X}$ facilitates a derivation of the search direction computed in D-IPM.

Let $(d\boldsymbol{X}, d\boldsymbol{Y}, d\boldsymbol{z})$ be the search direction. Then, we solve the following system with the Newton methods.

$$\left\{ \begin{array}{l} \boldsymbol{A}_k \bullet (\boldsymbol{X} + d\boldsymbol{X}) = b_k \ (k = 1, 2, \ldots, m) \\ \Sigma_{k=1}^{m} \boldsymbol{A}_k (z_k + dz_k) + (\boldsymbol{Y} + d\boldsymbol{Y}) = \boldsymbol{C} \\ \boldsymbol{X} + d\boldsymbol{X} = \mu (\boldsymbol{Y} + d\boldsymbol{Y})^{-1} \end{array} \right. \tag{3.7}$$

Since the last equation contains nonlinear term, we use an equation

$$(\boldsymbol{Y} + d\boldsymbol{Y})^{-1} = \boldsymbol{Y}^{-1} - \boldsymbol{Y}^{-1} d\boldsymbol{Y} \boldsymbol{Y}^{-1},$$

which holds in the first order approximation.

With the first and the second equations in (3.7), we construct the equivalent system.

$$\left\{ \begin{array}{l} \sum_{j=1}^{m} (\boldsymbol{Y}^{-1} \boldsymbol{A}_i \boldsymbol{Y}^{-1}) \bullet \boldsymbol{A}_j dz_j = b_i/\mu - A_i \bullet (\boldsymbol{Y}^{-1} - \boldsymbol{Y}^{-1} \boldsymbol{D} \boldsymbol{Y}^{-1}), \\ d\boldsymbol{Y} = \boldsymbol{D} - \sum_{k=1}^{m} \boldsymbol{A}_k dz_k, \end{array} \right.$$

where

$$\boldsymbol{D} = \boldsymbol{C} - \Sigma_{k=1}^{m} \boldsymbol{A}_k z_k - \boldsymbol{Y}.$$

This transformation indicates that we remove the dense matrices $\boldsymbol{X}$ and $d\boldsymbol{X}$. Following the manner for PD-IPM, we call the first equation the Schur complement equation for D-IPM. Furthermore, we call its coefficient matrix $\boldsymbol{B}$ the Schur complement matrix for D-IPM whose elements are in the form of $B_{ij} = (\boldsymbol{Y}^{-1} \boldsymbol{A}_i \boldsymbol{Y}^{-1}) \bullet \boldsymbol{A}_j$. Solving the Schur complement equation to obtain $d\boldsymbol{z}$ and a subsequent substitution to obtain $d\boldsymbol{Y}$ result in the comprehensive search direction $(d\boldsymbol{Y}, d\boldsymbol{z})$ required for D-IPM which solves SDPs from only dual space information. To progress D-IPM, we also need information to reduce μ toward 0, which is approximately estimated as $\mu = \frac{\boldsymbol{X} \bullet \boldsymbol{Y}}{n}$ in PD-IPM. This update is done by an appropriate choice of a potential function proposed in [70, 75] to assure a convergence of the sequence generated by D-IPM iterations.

An algorithmic framework of D-IPM is very similar to that of PD-IPM. Starting with positive definite matrix $\boldsymbol{Y}$, we continue iterations which are comprised of evaluating the Schur complement matrix, solving the Schur complement equation, computing the step length to keep positive definiteness of $\boldsymbol{Y}$, updating the variables $(\boldsymbol{Y}, \boldsymbol{z})$ to the next point, and estimation of the new μ with the potential function, until we reduce μ to sufficiently close to 0.

As mentioned above, the attractive characteristic of D-IPM is that it may release us from the computation cost and the memory space arisen from the primal dense $\boldsymbol{X}$. If the information $\boldsymbol{X}$ is surely required, we can estimate from the dual matrix $\boldsymbol{Y}$ and the equation $\boldsymbol{X} = \mu \boldsymbol{Y}^{-1}$. However, the high stability of PD-IPM and the strong results of the primal-dual pair formulation, for instance, the duality theorem described in section 2.1 may be lost in D-IPM because of lack of the primal information.

Another excellent property implemented in PDSDP in a combination with D-IPM is a low-rank property, which stands for that PDSDP effectively constructs the Schur complement matrix if the ranks of input data matrices $\boldsymbol{A}_k (k = 1, 2, \ldots, m)$ are relatively small compared to the matrix size n. Let r_i and r_j be ranks of $\boldsymbol{A}_i$ and $\boldsymbol{A}_j$, respectively. In addition, we decompose $\boldsymbol{A}_i$ and $\boldsymbol{A}_j$ through their eigenvalue decomposition in the form,

$$\boldsymbol{A}_i = \Sigma_{p=1}^{r_i} \xi_p \boldsymbol{u}_p \boldsymbol{u}_p^T, \quad \boldsymbol{A}_j = \Sigma_{q=1}^{r_j} \zeta_q \boldsymbol{v}_q \boldsymbol{v}_q^T,$$

where ξ_p, ζ_q are eigenvalues and $\boldsymbol{u}_p, \boldsymbol{v}_q$ are eigenvectors of $\boldsymbol{A}_i, \boldsymbol{A}_j$, respectively. Then, we can rewrite the formula to compute the elements of the Schur matrix $\boldsymbol{B}$,

$$\begin{array}{rcl} B_{ij} & = & (\boldsymbol{Y}^{-1} \boldsymbol{A}_i \boldsymbol{Y}^{-1}) \bullet \boldsymbol{A}_j \\ & = & \left(\boldsymbol{Y}^{-1} \left(\Sigma_{p=1}^{r_i} \xi_p \boldsymbol{u}_p \boldsymbol{u}_p^T \right) \boldsymbol{Y}^{-1} \right) \bullet \Sigma_{q=1}^{r_j} \zeta_q \boldsymbol{v}_q \boldsymbol{v}_q^T \\ & = & \Sigma_{p=1}^{r_i} \Sigma_{q=1}^{r_j} \ \xi_p \zeta_q (\boldsymbol{v}_q^T \boldsymbol{Y}^{-1} \boldsymbol{u}_p)^2. \end{array}$$

Therefore, the smaller r_i and r_j leads to less computation cost required for evaluation B_{ij}. In particular, all the input data matrices $\boldsymbol{A}_1, \ldots, \boldsymbol{A}_m$ has only 1 element in Max Cut Problems described in section 3.2.1, since PDSDP shows the strong power of its low-rank property for such cases.

The components to which PDSDP pay attention to apply parallel computation are similar that of SDPARA. Even though using D-IPM with low-rank property, the bottlenecks remain to the evaluation of the Schur complement matrix $\boldsymbol{B}$ and solving the Schur complement equation. In the first implementation of PDSDP before we implemented SDPARA, PDSDP adopted Conjugate Gradient Method to solve the equation on parallel processing. However, PDSDP replaced with the parallel Cholesky Factorization to attain higher accuracy, because of the same reason described in section 3.4.2.

In general, D-IPM has advantage for SDPs that have special structure in the input data matrices, while PD-IPM is capable of being applied for various SDPs. The numerical results in the following subsection, however, shows the higher performance of SDPARA than PDSDP.

3.6.2 SDPs from SDPLIB

We applied SDPARA and PDSDP with changing the number of processors to the SDP problems control10, control11, theta5, theta6, thetaG51 and maxG51 selected from SDPLIB. Their sizes are shown in Table 3.12. The total time required by them to solve the problems is shown in Table 3.17 and the scalability for control11 and theta6 are shown in Figure 3.15.

Table 3.17: Comparison of computation time (seconds) between SDPARA and PDSDP on multiple processors

the number of process		1	2	4	8	16	32	64
control10	SDPARA	441	235	129	75	43	32	25
	PDSDP	2090	1993	732	463	314	207	200
control11	SDPARA	685	363	195	112	67	43	32
	PDSDP	3319	3346	1369	757	531	340	299
theta5	SDPARA	223	114	72	51	35	26	20
	PDSDP	221	181	131	125	117	154	177
theta6	SDPARA	601	342	168	113	68	51	38
	PDSDP	586	471	328	296	253	323	375
maxG51	SDPARA	176	179	177	189	184	185	190
	PDSDP	85	83	80	83	83	95	108

In Control Problems, SDPARA achieves both faster total time and higher scalability than PDSDP. In particular, when we use 64 processors to solve control11, SDPARA is 64 times faster than PDSDP. The overwhelming difference is mainly due to the fact that SDPARA computes the elements of the Schur complement matrix $\boldsymbol{B}$ on parallel processing without any communication between the processors. This enables SDPARA to obtain high scalability for Control Problems which spend most of the computation time on the evaluation of the Schur complement matrix (see Table 3.3).

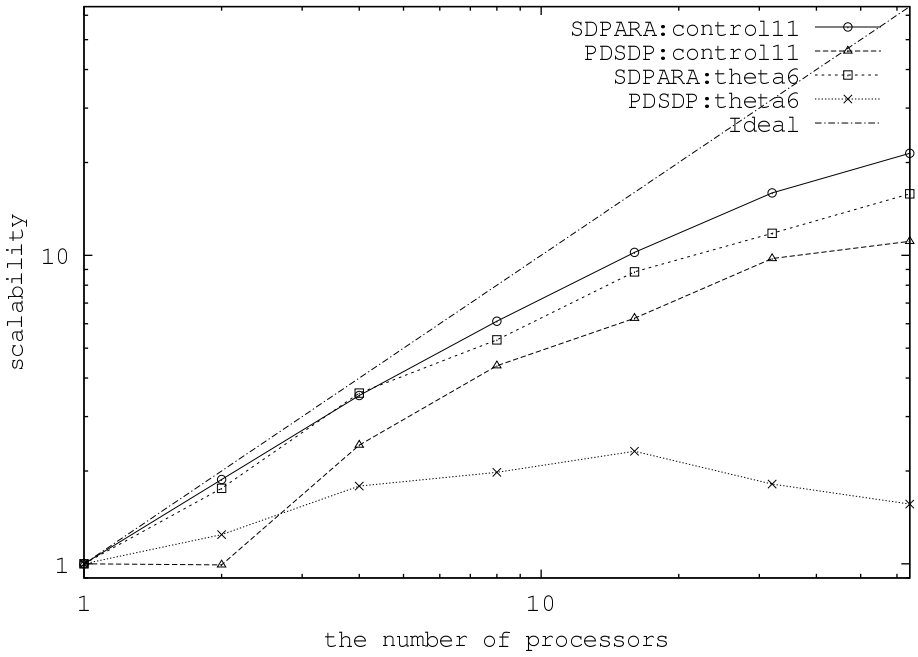
In the problems theta5 and theta6 which are Theta Function Problems, SDPARA is also faster than PDSDP. Since the algorithmic frameworks of the parallel Cholesky Factorization adopted by both software are essential the same, the difference between SDPARA and PDSDP indicates that the iteration number of SDPARA is smaller than that of PDSDP owing to the remarkable stability of PD-IPM.

In maxG11 and maxG51, however, PDSDP is obviously faster than SDPARA. This is mainly because the size n of the matrix variables $\boldsymbol{X}$ and $\boldsymbol{Y}$ is as large as the number m of the equality constraints in these problems (see Table 3.12), so that the main computation time is occupied by operations to the matrix variables $\boldsymbol{X}$ and $\boldsymbol{Y}$, PMATRIX and DENSE components, not by ELEMENTS and CHOLESKY. Hence SDPARA can not attain much scalability. For such SDPs, we can not expect SDPARA to work effectively. On the other hand, the dual scaling algorithm adopted by PDSDP effectively exploits the special sparsity of the input data, and attains the shorter computation time than SDPARA.

3.6.3 Quantum Chemistry

SDPARA and PDSDP were also applied on SDPs arisen from quantum chemistry listed in Table 3.14. Table 3.18 shows their numerical results. In addition, Figure 3.16 displays the scalability on NH_2 and LiF. Note that we use more than 8 processors, because the SDPs arisen from quantum chemistry have a tendency to

Figure 3.15: Scalability of SDPARA and PDSDP for control11 and theta6



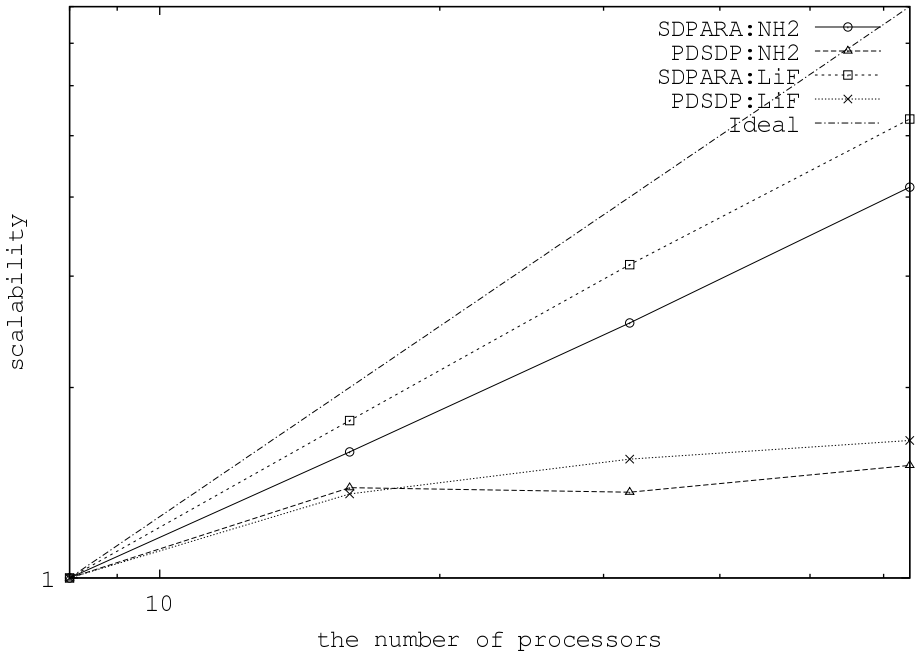
become large SDPs with large number of equality constraints. In Table 3.18, SDPARA can not solve CH_4 with 8 processors due to lack of memory space indicated by '*'.

Table 3.18: Performance of SDPARA and PDSDP for SDPs arisen from quantum chemistry

the number of processors		8	16	32	64
BH_3.$^1A_1$.STO-6G	SDPARA	67	37	27	17
	PDSDP	361	277	255	266
HF^+.$^2\Pi$.STO-6G	SDPARA	210	121	77	49
	PDSDP	813	663	682	612
NH_2.$^2A_1$.STO-6G	SDPARA	908	574	359	219
	PDSDP	3719	2677	2722	2469
LiF.$^1\Sigma$.STO-6G	SDPARA	3831	2161	1225	721
	PDSDP	16918	12458	10977	10258
CH_4.$^1A_1$.STO-6G	SDPARA	*	6368	3952	1980
	SDPARA	31065	25184	19359	15744

Table 3.18 makes it clear that SDPARA solves all SDPs many times faster than PDSDP. Particularly, CH_4 is extremely large SDP which could not solve by other software than SDPARA and PDSDP. Therefore, at the current time, the computation time for CH_4 attained by SDPARA is the shortest time. Furthermore, the scalability of SDPARA exceeds that of PDSDP. Though PDSDP slightly raises the scalability with more than 16 processors, SDPARA can lengthen the scalability up to 64 processors in the straight way. The main difference between SDPARA and PDSDP is their algorithmic framework, PD-IPM and D-IPM, respectively. It is possible to consider that the stability of PD-IPM enables SDPARA to attain such a higher scalability.

Figure 3.16: Scalability of SDPARA and PDSDP on NH_2 and LiF



3.7 Theoretical Validity of Parallel Implementation in SDPARA

We have already described details of the parallel implementation of SDPARA (in section 3.2) and reported its numerical results (in section 3.4,3.5). However, we have several alternative approaches in the course of developments of SDPARA. In this section, we discuss a theoretical validity of selected approaches in SDPARA focusing on a trade-off between reductions of computation time owing to parallel processing and overheads of communication between multiple processors, and comparing with other alternative approaches.

Throughout discussions in this section, we use three values t_f, t_v and t_m in Table 3.19 regarding computation and communication time. The unit size of these values is second. Here, we assume a type of all

Table 3.19: Computation and Communication Time (second)

t_f	Computation time for a multiplication between two values
t_v	Transfer time to send a value
t_m	Latency

data values stored in memory space is 'double' type which requires 64 bits. In other words, t_f stands for time to compute a multiplication between two 'double' values, t_v stands for the reciprocal of the number of 'double' values which can be sent in a second (the reciprocal of bandwidth), and t_m stands for time to send the first 'double' value from one processor to another processor, respectively.

When real values are required to compare with alternative approaches after we establish theoretical measures, we use values attained in numerical experiments executed on PC-cluster, Presto III. As described in section 3.3, the important feature of the PC-cluster is that all nodes are connected by Myrinet. Myrinet provides a high network capacity whose bandwidth is 2 Giga bit per second at its theoretical limits. Therefore, communication time t_v is not a so heavy burden as compared to computation time t_f. Approximate orders of the above three values on Presto III are $t_f = 10^{-9}, t_v = 10^{-8}, t_m = 10^{-5}$. (Note that the unit sizes of these values are second. For example, a multiplication between two 'double' values requires about 10^{-9} second.) These values may be changed when we use a different PC-cluster.

SDPARA has two parallel components, ELEMENTS (in section 3.2.2) and CHOLESKY (in section

3.2.3). We will concentrate on the two components in this section. First, we discuss ELEMENTS component which evaluates elements of the Schur complement matrix on parallel processing. In advance of listing up alternative approaches, we review the algorithmic framework implemented in SDPARA.

Evaluation of the Schur Complement Matrix on the uth Processor

Before starting the first iteration of PD-IPM, we select $\mathcal{F}_{(i,l)}$ from $\mathcal{F}_1, \mathcal{F}_2, \mathcal{F}_3$ for the ith row and the lth block.
Prepare a flag sym. If $\mathcal{F}_1$ or $\mathcal{F}_2$ are chosen more than once then $sym = 1$, else $sym = 0$.
Reserve memory space to store rows assigned by $\mathcal{P}_u$.
At each iteration of PD-IPM, we compute as follow.

$\boldsymbol{B} = \boldsymbol{O}$
For $l = 1, 2, \ldots, h$ (iterator for block)
  For $i \in \mathcal{P}_u$ (iterator for row)
    For $j = 1, 2, \ldots, m$ (iterator for column)
      If $nz\{A_i\}_l \geq nz\{A_j\}_l$ or $sym = 0$ then
        Compute $B_{ij}^{(l)}$ by $\mathcal{F}_{(i,l)}$
        $B_{ij} \leftarrow B_{ij} + B_{ij}^{(l)}$
If $sym = 1$ then symmetrize $\boldsymbol{B}$ on distributed memory

In the course of implementation, we had two alternative approaches below.

1. [SORT vs. NON-SORT]

   In SDPARA, we do not use the sorting of input data matrices by their density proposed in [22] (NON-SORT). Therefore, an alternative approach can be the evaluation with the sorting (SORT).

2. [Automatic Symmetrization]

   We symmetrize the Schur complement matrix on distributed memory if required. The other alternative approaches are one that evaluates all elements in the upper triangular instead of the symmetrization (SYM-OFF), and one that always uses the symmetrization for any SDP (SYM-ON).

We start from a comparison between SORT and NON-SORT. As described in section 3.2.2, the manner to evaluate the Schur complement matrix proposed in Fujisawa *et al.* [22] is as follow. Let $\{\boldsymbol{A}_i\}_l$ denotes the lth diagonal block of $\boldsymbol{A}_i$. For the lth diagonal block ($l = 1, 2, \ldots, h$), they first sort $\{\boldsymbol{A}_1\}_l, \{\boldsymbol{A}_2\}_l, \ldots, \{\boldsymbol{A}_m\}_l$ by their density with a permutation σ so that $\{\boldsymbol{A}_{\sigma(1,l)}\}_l, \{\boldsymbol{A}_{\sigma(2,l)}\}_l, \ldots, \{\boldsymbol{A}_{\sigma(m,l)}\}_l$ satisfy $nz(\sigma(i,l)) \geq nz(\sigma(j,l))$ for $\sigma(i,l) \leq \sigma(j,l)$, where $nz(\sigma(i,l))$ is the number of non-zero elements of $\{\boldsymbol{A}_{\sigma(i,l)}\}_l$. Then they compute the elements of the sub Schur complement matrix $B^{(l)}_{\sigma(i,l)\sigma(j,l)}$ if $\sigma(i,l) \leq \sigma(j,l)$. The remarkable point of their manner is a selection of appropriate formulas depending on the sparsity of the input matrices from three formulas, $\mathcal{F}_1$ for dense, $\mathcal{F}_2$ for mildly dense and $\mathcal{F}_3$ for sparse. The algorithmic framework for SORT will be described below. Here, we do not consider the symmetrization to concentrate on SORT vs. NON-SORT.

Evaluation of the Schur Complement Matrix on the uth Processor [with SORT]

Before starting the first iteration of PD-IPM, compute a permutation $\sigma(i,l)$ and thresholds p_l, q_l.
Reserve memory space to store rows assigned by $\mathcal{P}_u$.
At each iteration of PD-IPM, we compute as follow.

$\boldsymbol{B} = \boldsymbol{O}$
For $l = 1, 2, \ldots, h$ (iterator for block)
  For $\widehat{i} \in \mathcal{P}_u$ (iterator for row)
    $\boldsymbol{b} = \boldsymbol{0}$
    $i \leftarrow \sigma(\widehat{i}, l)$
    If $i < p_l$ then $\mathcal{F} \leftarrow \mathcal{F}_1$, else if $i < q_l$ then $\mathcal{F} \leftarrow \mathcal{F}_2$, else $\mathcal{F} \leftarrow \mathcal{F}_3$
    For $\widehat{j} = \widehat{i}, \widehat{i} + 1 \ldots, m$ (iterator for column)
      $j \leftarrow \sigma(\widehat{j}, l)$

```
            Compute B_ij^(l) by F and b_j ← B_ij^(l)
        End (For)
        send b to the vth processor (i ∈ P_v)
        add b to the ith row of B on the vth processor (i ∈ P_v)
    End (For)
End (For)
```

As we have pointed in section 3.2.2, SORT requires communication because the evaluation of an element may executed on a processor which is different from the processor where the result of the element must be stored. The example in section 3.2.2 depicted this conflict more clearly. Therefore, the communication cost may be a heavy burden of SORT.

To compare SORT and NON-SORT, we can not neglect that the computation cost for $B_{ij}^{(l)}$ $(i, j = 1, 2, \ldots, m)$ and their addition are the same in the either approach because of Corollary 3.2.1. Furthermore, since the number h of diagonal block and the iteration number of PD-IPM become the same, we need only to compare the computation cost at inside of the For loop with respect to $\widehat{i}$ of SORT and i of NON-SORT. We summarized each overhead of NON-SORT and SORT compared with the other approach in Table 3.20 and 3.21, respectively.

Table 3.20: Overhead of NON-SORT

operation	the number of required operation	time for one operation
pick up of $\mathcal{F}$	m/N	t_{pick}
pick up of $nz(i,l), nz(j,l)$	$m(m+1)/N$	t_{pick}
comparison of $nz(i,l), nz(j,l)$	m^2/N	t_{cmp}

Table 3.21: Overhead of SORT

operation	the number of required operation	time for one operation
assignment of $\boldsymbol{b} = \boldsymbol{0}$	m/N	t_{assign}
pick up of i	m/N	t_{pick}
pick up of j	$m(m-1)/(2N)$	t_{pick}
comparison to define $\mathcal{F}$	$2m/N$	t_{cmp}
transfer of $\boldsymbol{b}$	m/N	t_{send}

Let N be the number of all available processors. We assume the ideal shortest communication time between N processors, that is, N source processors can send and N destination processors can receive simultaneously. This simple assumption is very useful to estimate the lower bound of t_{send} in Table 3.21. Furthermore, each processor transfers $\boldsymbol{b}$ to itself with a possibility $1/N$, we can remove the transfer. Then t_{send} can be estimated as follow.

$$t_{send} \quad \geq \quad (mt_v + t_m) \times (N-1)/N.$$

Hence the total difference between T_{SORT} (the total time for SORT) and $T_{NON-SORT}$ (the total time for NON-SORT) becomes

$$\begin{aligned}
& T_{SORT} - T_{NON-SORT} \\
= \; & \{mt_{assign}/N + m(m+1)t_{pick}/(2N) + 2mt_{cmp}/N + m/N \times (mt_v + t_m)(N-1)/N\} \\
& -(mt_{pick}/N + m(m+1)t_{pick}/N + m^2 t_{cmp}/N) \\
= \; & m/N \times \{t_{assign} - (m+3)t_{pick}/2 + (mt_v + t_m)(N-1)/N - (m-2)t_{cmp}\} \\
\geq \; & m/N \times (mt_v(N-1) - ((m-2)t_{cmp} + (m+3)t_{pick}/2) \\
\geq \; & m/N \times (mt_v(N-1)/N - m(t_{cmp} + t_{pick})) \\
= \; & m^2/N \times (t_v(N-1)/N - (t_{cmp} + t_{pick})).
\end{aligned}$$

Therefore, when $N \geq 2$, the total time for NON-SORT is shorter than that for SORT if $t_v \geq (t_{cmp} + t_{pick})$. From numerical experiments on Presto III, we obtained $t_v \geq 3.2 \times 10^{-8}$ and $t_{cmp} + t_{pick} \leq 1.5 \times 10^{-8}$.

Furthermore, the value of t_{send} used in the above estimation is the ideal lower bound and can not be attained in practical situations. Consequently, we obtain $T_{NON-SORT} \leq T_{SORT}$, which means that the selection of NON-SORT approach is valid.

Next, we move on a validity of Automatic Symmetrization. When we use SYM-ON, we compute either $B_{ij}^{(l)}$ or $B_{ji}^{(l)}$ depending on the comparison between $nz(i,l)$ and $nz(j,l)$. After the evaluation of all elements, we need to symmetrize the total Schur complement matrix $\boldsymbol{B} = \Sigma_{l=1}^{h} \boldsymbol{B}^{(l)}$ by $B_{ij} \leftarrow (B_{ij} + B_{ji})/2 \quad (i < j)$. The computation of $B_{ij}^{(l)}$ can be computed the appropriate formula from $\mathcal{F}_1, \mathcal{F}_2, \mathcal{F}_3$, but the communication cost is required to symmetrize $\boldsymbol{B}$ because it is stored on the distributed memory. Generally speaking, the symmetrization consumes communication time $T_{sym} = m^2 t_v / N$.

On the other hand, SYM-OFF approach always computes $B_{ij}^{(l)}$ when $i \leq j$. Therefore, no communication is necessary for $\boldsymbol{B}$ to be a symmetric matrix. However, there is a possibility that the appropriate formula is not selected for some j, since the formula to compute $B_{ij}^{(l)}$ is decided by only i. We use $\mathcal{G}^{(1,l)}, \mathcal{G}^{(2,l)}, \mathcal{G}^{(3,l)}$ to denote the sets of row indices correspond to $\mathcal{F}_1, \mathcal{F}_2, \mathcal{F}_3$, respectively. To be precise,

$$\mathcal{G}^{(p,l)} = \{i : 1 \leq i \leq m, \mathcal{F}_{(i,l)} = \mathcal{F}_p\} \quad \text{for} \quad p = 1, 2, 3.$$

Furthermore, we define $m^{(p,l)}$ for the cardinality of $\mathcal{G}^{(p,l)}$ (Note that $m^{(1,l)} + m^{(2,l)} + m^{(3,l)} = m,\ l = 1, 2, \ldots, h$). and $nz^{(p,l)}$ for sum of nonzero elements of $\{\boldsymbol{A}_i\}_l$, that is, $nz^{(p,l)} = \Sigma_{i \in \mathcal{G}^{(p,l)}}\ nz(i,l)$. When we use SYM-ON, the number of elements of $\boldsymbol{B}^{(l)}$ evaluated by each formula is counted as Table 3.22. In the

Table 3.22: Number of elements of $\boldsymbol{B}^{(l)}$ for each formula in the case of the appropriate selection

$i \quad \backslash \quad j$	$\mathcal{G}^{(1,l)}$	$\mathcal{G}^{(2,l)}$	$\mathcal{G}^{(3,l)}$
$\mathcal{F}_1, \mathcal{G}^{(1,l)}$	$m^{(1,l)}(m^{(1,l)}m_1 + 1)/2$	$m^{(1,l)}m^{(2,l)}$	$m^{(1,l)}m^{(3,l)}$
$\mathcal{F}_2, \mathcal{G}^{(2,l)}$		$m^{(2,l)}(m^{(2,l)} + 1)/2$	$m^{(2,l)}m^{(3,l)}$
$\mathcal{F}_3, \mathcal{G}^{(3,l)}$			$m^{(3,l)}(m^{(3,l)} + 1)/2$

case of SYM-OFF, if we compute $B_{ij}^{(l)}$ without the comparison of $nz(i,l)$ and $nz(j,l)$, the following three non-appropriate selection may happen.

1. The elements $\boldsymbol{B}_{ij}^{(l)}$ which should be computed by $\mathcal{F}_1$ may be computed by $\mathcal{F}_2$. The computation cost increases $(n_l + 1)nz(j,l) - nz(j,l)$.

2. The elements $\boldsymbol{B}_{ij}^{(l)}$ which should be computed by $\mathcal{F}_1$ may be computed by $\mathcal{F}_3$. The computation cost increases $(2nz(i,l) + 1)nz(j,l) - nz(j,l)$.

3. The elements $\boldsymbol{B}_{ij}^{(l)}$ which should be computed by $\mathcal{F}_2$ may be computed by $\mathcal{F}_3$. The computation cost increases $(2nz(i,l) + 1)nz(j,l) - (n_l + 1)nz(j,l)$.

In the above, n_l is the size of the lth diagonal matrix $\{\boldsymbol{X}\}_l$. The increment costs are estimated based on the formulas proposed $\mathcal{F}_1, \mathcal{F}_2$ and $\mathcal{F}_3$ in [22]. Here, we assume that the order of $nz(1,l), nz(2,l), \ldots, nz(m,l)$ follows uniform distribution and all $\{\boldsymbol{A}_1\}_l, \{\boldsymbol{A}_2\}_l, \ldots, \{\boldsymbol{A}_m\}_l$ are stored in sparse data structures. The second assumption implies that an overhead of the sparse data structures need not be taken into consideration.

Hence, the total increment of computation cost T_{inc} on each processor in SYM-OFF approach can be estimated as

$$\begin{aligned} T_{inc} &= \Sigma_{l=1}^{h} \frac{t_f}{2N} \{ \Sigma_{i \in \mathcal{G}^{(1,l)}} \Sigma_{i \in \mathcal{G}^{(2,l)}} ((n_l + 1)nz(j,l) - nz(j,l)) \\ &\qquad + \Sigma_{i \in \mathcal{G}^{(1,l)}} \Sigma_{i \in \mathcal{G}^{(3,l)}} ((2nz(i,l) + 1)nz(j,l) - nz(j,l)) \\ &\qquad + \Sigma_{i \in \mathcal{G}^{(2,l)}} \Sigma_{i \in \mathcal{G}^{(3,l)}} ((2nz(i,l) + 1)nz(j,l) - (n_l + 1)nz(j,l)) \} \\ &= \Sigma_{l=1}^{h} \frac{t_f}{2N} \{ n_l m^{(1,l)} nz^{(2,l)} + 2nz^{(1,l)} nz^{(3,l)} + 2(nz^{(2,l)} - n_l m^{(2,l)}) nz^{(3,l)} \}. \end{aligned}$$

Therefore, the following strategy is reasonable.

If $T_{inc} < T_{sym}$, we adopt SYM-OFF.
Else we adopt SYM-ON.

From numerical experiments on the two SDPs, control11 and theta6, on Presto III, we obtained the real time in second with respect to the computation of $B_{ij}^{(l)}(i, j = 1, 2, \ldots, m,\ l = 1, 2, \ldots, h)$ [COMPUTE] and the symmetrization of $\boldsymbol{B}$ [SYM] on SYM-ON and SYM-OFF (Table 3.23,3.24,3.25 and 3.26). The real time in the tables are the total time over all the iterations of PD-IPM. The iteration number of control11 and theta6 are 48 and 18, respectively. In addition, the time for COMPUTE and SYM are the same through all the iterations. For example, COMPUTE time in each iteration with respect to SYM-ON for control11 on a single processor is $596.0/48 = 12.4$.

Table 3.23: Performance of SYM-ON for control11

the number of processors	1	2	4	8	16	32	64
COMPUTE	596.0	300.2	149.9	73.9	36.7	18.3	9.2
SYM	5.4	11.1	7.8	5.5	2.2	1.7	0.6
Total	690.3	367.6	198.0	112.4	63.8	42.8	28.5

Table 3.24: Performance of SYM-OFF for control11

the number of processors	1	2	4	8	16	32	64
COMPUTE	716.6	363.0	192.4	105.0	60.9	37.0	27.9
SYM	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Total	805.2	417.8	232.3	137.2	85.5	59.3	46.6

Table 3.25: Performance of SYM-ON for theta6

the number of processors	1	2	4	8	16	32	64
COMPUTE	122.3	58.2	28.8	14.8	7.6	4.4	2.7
SYM	18.0	32.1	24.1	17.0	6.8	5.0	1.8
Total	726.5	364.8	190.3	127.4	71.8	53.2	37.0

Table 3.26: Performance of SYM-OFF for theta6

the number of processors	1	2	4	8	16	32	64
COMPUTE	122.0	56.9	27.6	13.6	6.8	3.4	1.8
SYM	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Total	707.9	331.2	165.5	109.4	64.1	48.0	34.1

First, we consider control11 ($m = 1596, bLOCKsTRUCT = [110, 55]$). From the statics of control11

Table 3.27: Statics of control11

1st block	$m^{(1,1)}$	1571	$m^{(2,1)}$	25	$m^{(3,1)}$	0
	$nz^{(1,1)}$	1641358	$nz^{(2,1)}$	10372	$nz^{(3,1)}$	0
2nd block	$m^{(1,2)}$	0	$m^{(2,2)}$	0	$m^{(3,2)}$	1596
	$nz^{(1,2)}$	0	$nz^{(2,2)}$	0	$nz^{(3,2)}$	6050

summarized in Table 3.27, $T_{inc} = t_f/N \times 1.85 \times 10^9$. When the appropriate formulas are selected (in the case of SYM-ON), the computation for COMPUTE requires $t_f/N \times 4.93 \times 10^7$ second, and its real time is $596.0/48/N$ second (48 is the iteration number of PD-IPM). Hence T_{inc} becomes approximately $458/N$ second. On the other hand, the communication time is bounded as $t_v \leq 5.7 \times 10^{-7}$ owing to Myrinet capacity; thus $T_{sym} \leq 70/N$. Therefore, SYM-ON is faster than SYM-OFF as the numerical results indicated.

Next, we consider Max Cut Problem case. In Max Cut Problem, the number of non-zero elements involved in each input matrix is 1, that is, $nz(i, 1) = 1$ for $i = 1, 2, \ldots, m$. (Note that the number of

diagonal block matrix is 1, that is $h = 1$, in Max Cut Problem or Theta Function Problem.) Hence, only $\mathcal{F}_3$ is selected for all rows of $\boldsymbol{B}$. Then, the assignment of non-appropriate formulas do not happen. Therefore, $T_{inc} = 0$ since $m^{(1,l)} = m^{(2,l)} = nz^{(1,l)} = nz^{(2,l)} = 0$. In this case, SYM-OFF is faster than SYM-ON.

Finally, we pick up an interesting case, Theta Function Problem. In Theta Function Problem, only one input matrix is the identity matrix and the number of non-zero elements in the other input matrices are 2. In this case, two formulas $\mathcal{F}_2$ and $\mathcal{F}_3$ are used, but T_{inc} becomes 0 since $m^{(2,1)} = 1, m^{(3,1)} = m-1, nz^{(1,1)} = 0, nz^{(2,1)} = n_1, nz^{(3,l)} = 2(m-1)$. Hence, SYM-OFF is faster than SYM-ON. This result is consistent in the results of Table 3.25 and 3.26.

In SDPARA implementation, we adopt Automatic Symmetrization whose strategy is

$$\begin{cases} \text{If } \mathcal{F}_1 \text{ or } \mathcal{F}_2 \text{ are selected more than once :} & \text{SYM-ON is used} \\ \text{Otherwise :} & \text{SYM-OFF is used.} \end{cases}$$

In other words, the Automatic Symmetrization uses SYM-OFF when $T_{inc} = 0$. The strategy is simpler than the one we have discussed above based on the comparison between the computation time T_{inc} and the communication cost T_{sym}. However, numerical results on many SDPs indicate that the simple strategy can be considered as a well approximation to the one discussed above. Therefore, we have verified that the Automatic Symmetrization implemented in SDPARA is effective.

Until now, we concentrate on ELEMENTS component. In SDPARA, we have another parallel component, CHOLESKY. To enhance the performance of the parallel Cholesky Factorization, we redistribute the Schur complement matrix $\boldsymbol{B}$ from the row-wise distribution to TD-BCD (two-dimensional block-cyclic distribution). Therefore, it will be a subject matter that how much fast the parallel Cholesky Factorization on the row-wise distribution without the redistribution is as compared to the parallel Cholesky Factorization on TD-BCD with the redistribution.

Here, we assume a belt size in the row-wise distribution is one, because the slimest belt attains the shortest time for ELEMENTS component as verified in section 3.4.1. Therefore, the uth processor stores rows of $\boldsymbol{B}$ defined by $\mathcal{P}_u = \{i : 1 \leq i \leq m, (i-1)\%N = u\}$. An algorithmic framework of the parallel Cholesky Factorization on the row-wise distribution to factorize $m \times m$ matrix $\boldsymbol{B}$ into a lower triangular matrix is summarized as follow.

The parallel Cholesky Factorization on Row-wise Distribution on the uth processor

```
For i = 1, 2, ..., m
    If i ∈ P_u
        B_ii ← √B_ii
        Broadcast B_ii to all processors
    End (If)
    For j = i+1, i+2, ..., m
        If j ∈ P_u
            B_ji ← B_ji / B_ii
            Broadcast B_ji to all processors
        End(If)
    End (For)
    For j = i+1, i+2, ..., m
        If j ∈ P_u
            For k = i+1, i+2, ..., m
                B_jk ← B_jk − B_ji × B_ki
            End (For)
        End (If)
    End (For)
End (For)
```

Based on the above algorithmic framework, the order of the computation time of the parallel Cholesky Factorization on the row-wise distribution T_{ROW} can be estimated as

$$T_{ROW} = m^3 t_f/(3N) + (m + m^2/(2N)) \times t_v \log_2 N + 2mt_m, \tag{3.8}$$

where $\log_2 N$ implies that B_{ii} and B_{ji} are broadcasted. To broadcast from the 1st processor to all processors, 1st processor sends to the 2nd and the 3rd processor at first. Then the received ith processor sends to the $2i$th and the $(2i+1)$th processor in general. Hence, all processors receive the data from the 1st processor with in $t_v \log_2 N$ second.

On the other hand, the order of computation time of the parallel Cholesky Factorization on TD-BCD T_{TD} and the communication time for the redistribution from the row-wise distribution T_{red} are

$$T_{TD} = m^3 t_f/(3N) + (2 + 1/2 \log_2 N) m^2 t_v/\sqrt{N} + (4 + \log_2 N)/(mb) t_m, \quad (3.9)$$

$$T_{red} = m^2 t_v/N, \quad (3.10)$$

where mb is a block size of TD-BCD. The order of T_{TD} in (3.9) is different from that in section 3.2.3, because (3.9) is better suited for the comparison with the row-wise distribution. The difference comes from the one with respect to the assumptions of data transfer type. We cite the above order of T_{TD} from [8].

The comparison between the order of T_{ROW} and that of $T_{TD} + T_{red}$ implies the row-wise distribution looks faster than TD-BCD. However, the apparent result is inconsistent in numerical results on Presto III.Table 3.28 and 3.29 are numerical results on control11 and theta6, respectively. The problem control11 has $m = 1596$ equality constraints and requires 48 iterations, while theta6 has $m = 4375$ equality constraints and requires 18 iterations.

Table 3.28: Performance of CHOLESKY for control11

the number of processors	1	2	4	8	16	32	64
T_{ROW}	1173.3	418.5	247.9	161.4	132.7	105.3	94.5
T_{TD}	65.5	29.7	18.3	14.7	8.8	7.9	5.0
T_{red}	10.4	15.5	11.4	7.5	4.4	2.8	1.1

Table 3.29: Performance of CHOLESKY for theta6

the number of processors	1	2	4	8	16	32	64
T_{ROW}	12381.7	2826.5	1613.4	973.8	623.9	458.3	372.3
T_{TD}	547.3	164.8	92.6	62.5	33.5	25.2	14.5
T_{red}	31.0	40.4	33.1	21.5	11.5	6.3	4.4

The inconsistency stems from an implicit assumption that t_f, t_v and t_m are the same regardless of the row-wise or TD-BCD. For example, if we compute the Cholesky Factorization on a single processor, then the computation cost is approximately $m^3 t_f/3$. However, the computation time of non-block-oriented Cholesky Factorization which is a base of the parallel Cholesky Factorization on the row-wise and that of block-oriented Cholesky Factorization (see section 3.2.3) which is a base of the parallel Cholesky Factorization on TD-BCD are clearly different as indicated by Table 3.30. (The denominators in time are the number of iterations.)

Table 3.30: Difference of computation time between non-block-oriented and block-oriented

		computation time (second)
control (m=1596)	non-block	671 / 48
	block	37 / 48
theta (m=4375)	non-block	5561 /18
	block	253 / 18

We estimate t_f from Table 3.30 and estimate t_v and t_m from the formulas (3.8),(3.9),(3.10), and Table 3.28,3.29. The estimated values are summarized in Table 3.31. (The value t_v in T_{red} is the upper bound when the number of processors is changed.) Then, the most significant differences come from t_f and t_v. The t_f and t_v of T_{ROW} are considerably larger than those of T_{TD}. It may possible to consider that the parallel

Table 3.31: Estimated computation cost and communication cost on CHOLESKY

		t_f	t_v	t_m
control11	T_{ROW}	1.2×10^{-8}	9.0×10^{-7}	1×10^{-5}
	T_{TD}	1.7×10^{-9}	4.7×10^{-8}	1×10^{-5}
	T_{red}		6.2×10^{-7}	
theta6	T_{ROW}	8.7×10^{-9}	2.3×10^{-6}	1×10^{-4}
	T_{TD}	3.9×10^{-10}	3.3×10^{-8}	1×10^{-5}
	T_{red}		8.1×10^{-7}	

Cholesky Factorization can exploit memory cache more effectively and transfer data with longer length to attain a sufficient bandwidth owing to the block-oriented Cholesky Factorization than the parallel Cholesky Factorization on the row-wise distribution based on the non-block-oriented Cholesky Factorization. Hence, the parallel Cholesky Factorization on TD-BCD with the redistribution from the row-wise distribution is actually reasonable to be embedded in SDPARA.

Through the above discussions, we verified that the parallel implementation of SDPARA is superior to the other alternative approaches.

Chapter 4

Parallel Implementation with the Completion Method

In the previous chapter, we have pointed out the bottlenecks of PD-IPM and describe how to overcome them with advantage of parallel processing. In addition, we have shown the high performance of SDPARA [86] through numerical experiments, in particular, for quantum chemistry.

However, we still have another opportunity to solve considerably large SDPs. What we need to remember is that SDPARA could not reap any benefits from parallel processing in the numerical experiments for Max Cut Problem. This fact leads us a further approach for SDPs; an incorporation of the completion method.

We start this chapter from an examination of additional exploiting an structural sparsity, then introduce a concept of the completion method in section 4.1. We also present some distinguishing and important theoretical groundworks. Then, we carry forward the main subject in section 4.2; how to incorporate the completion method into the framework of parallel processing, especially from viewpoints of memory reduction and computation efficiency. Then, section 4.3 and section 4.4 show numerical results and prove a significant development based on the completion method At the end of this chapter, we provide a theoretical validity of the parallel implementation in section 4.5.

4.1 Incorporation of the Completion Method

4.1.1 Drawbacks of SDPARA and Introduction of the Completion Method

As we have seen, the bottlenecks in PD-IPM for general SDPs are the evaluation of the Schur complement matrix (ELEMENTS) and its Cholesky Factorization (CHOLESKY). Replacing the two bottlenecks by their parallel implementation is the most fundamental feature of SDPARA. As a result, SDPARA shows the high performance and the excellent scalability with the aid of strong parallel processing. In particular, SDPARA attains a considerable reduction in computation time for extremely large SDPs arisen from the quantum chemistry.

Nevertheless, we should not underestimate the fact that the performance is very poor for Max Cut Problem. SDPARA can not obtain any scalability on the problem. The time consumed by ELEMENTS and CHOLESKY components is far small compared to the total time. Therefore, the parallel implementation of SDPARA does not work effectively. The most computation time on Max Cut Problem is consumed to compute $d\boldsymbol{X}$ (the primal component of the search direction) and to deal with dense matrices, which are components called PMATRIX and DENSE, respectively, as we have categorized in section 3.2.1.

The characteristic of Max Cut Problem reveals a drawback of SDPARA. From the formulation described in section 3.2.1, the number m of equality constraints of the problem is equal to the size n of matrices . Thus, if we want to solve Max Cut Problem with more equality constraints, the required memory space to store variable matrices, $\boldsymbol{X}$ and $\boldsymbol{Y}$, grows in proportion to m^2. Another drawback of SDPARA is that SDPARA restores the matrices $\boldsymbol{X}$ and $\boldsymbol{Y}$ in the full-dense style on each processor, even if all the input data matrices $\boldsymbol{C}, \boldsymbol{A}_1, \boldsymbol{A}_2, \ldots, \boldsymbol{A}_m$ have some sparsity structures. To make matters worse, all the computation with respect to these matrices are done on each processor. In other words, PMATRIX and DENSE components can not obtain any advantage of parallel processing. Hence, almost all the performance of SDPARA is vanished,

when we try to solve SDPs that the number of equality constraints is not sufficiently larger than the size of matrices.

To overcome the above drawbacks of SDPARA, let us focus on a special sparsity which belongs to all the input data matrices. We start from an easy sample SDPs. First, let the size n of matrix be equal to the number of equality constraints in which SDPARA has trouble. Then, each input data matrix $\boldsymbol{A}_k (k = 1, 2, \ldots, n = m)$ has non-zero elements in the only position $\{(k,k),(k,n),(n,k)\}$. In addition, the positions of the non-zero elements in the coefficient matrix $\boldsymbol{C}$ are also $\{(i,i),(i,n),(n,i)\}$ where $i = 1, 2, \ldots, n$.

Terminologies in the field of graph theory play essential roles to explain the completion method. In order to relate non-zero elements of matrix to a graph, we introduce *an assigned graph.*

Definition 4.1.1. *Given an $n \times n$ symmetric matrix $\boldsymbol{X}$, a graph $G(V,E)$ is said to be an assignment graph of $\boldsymbol{X}$ if the vertex set is $V = \{1, 2, \ldots, n\}$ and the edge set is $E = \{(i,j) \in V \times V : X_{ij} \neq 0\}$. And we call such edge set E an assignment edge set of $\boldsymbol{X}$. Conversely, if all positions of non-zero elements of $\boldsymbol{X}$ is covered by a edge set of a given $G(V,E)$, we say $\boldsymbol{X}$ is assigned by E, thus by G .*

Then, let us consider the sparsity in the dual standard SDPs. The dual variable matrix $\boldsymbol{Y}$ directly inherits the special sparsity from the dual constraints,

$$\boldsymbol{Y} = \boldsymbol{C} - \Sigma_{k=1}^{m} \boldsymbol{A}_k z_k.$$

It means that $\boldsymbol{Y}$ in the simple example is assigned by an edge set in $\{(i,i),(i,n),(n,i) : i = 1, 2, \ldots, n\}$. We define *an aggregate sparsity pattern* as a composition of assigned edges of all the input data matrices. For convenience, let $\boldsymbol{A}_0 = \boldsymbol{C}$.

Definition 4.1.2. *We define an aggregate sparsity pattern $\overline{E}$ as*

$$\overline{E} = \cup_{k=0}^{m} \{(i,j) : (i,j) \text{ element of } \boldsymbol{A}_k \text{ is nonzero}\}.$$

We call a matrix assigned by $\overline{E}$ an aggregate sparsity matrix.

Hence, it may be natural that we exploit the aggregate sparsity in the dual variables. The direct inheritance of the sparsity in the dual SDP is origin of D-IPM adopted by PDSDP as described in section 3.6.1. However, we should retain the excellent stability of PD-IPM, instead using D-IPM.

Here, we look back on the primal standard SDPs and pay attention to the primal variable matrix $\boldsymbol{X}$. Since $\boldsymbol{X}$ is usually dense, SDPARA suffers from PMATRIX and DENSE. In the primal SDP, we minimize the objective function $\boldsymbol{C} \bullet \boldsymbol{X}$ over equality constraints $\boldsymbol{A}_k \bullet \boldsymbol{X} = b_k$ with positive semidefiniteness of $\boldsymbol{X}$. Therefore, the elements of $\boldsymbol{X}$ required for the inner-product are only the elements whose positions are correspond to non-zero elements of the input data matrices. In the simple example, $\{(k,k),(k,n),(n,k)\}$ elements of $\boldsymbol{X}$ are used to the take inner-product with $\boldsymbol{A}_k$. Therefore, $\{(i,i),(i,n),(n,i) : i = 1, 2, \ldots, n\}$ elements of $\boldsymbol{X}$ assigned by the aggregate sparsity pattern are enough to compute the inner-products with all the input matrices. We derive the following aggregate sparsity matrix $\overline{\boldsymbol{X}}$ from $\boldsymbol{X}$,

$$\overline{\boldsymbol{X}} = \begin{pmatrix} X_{11} & & & & X_{1n} \\ & X_{22} & & & X_{2n} \\ & & \ddots & & \vdots \\ & & & X_{n-1,n-1} & X_{n-1,n} \\ X_{1n} & X_{2n} & \ldots & X_{n-1,n} & X_{nn} \end{pmatrix}.$$

Note that $\overline{\boldsymbol{X}}$ is sufficient for inner-products in the primal SDP. Furthermore, if we hold the positive semidefiniteness of $\overline{\boldsymbol{X}}$ in some sense, we can construct $\boldsymbol{X}$ keeping the non-zero elements of $\overline{\boldsymbol{X}}$ as follows.

$$\boldsymbol{X} = \begin{pmatrix} X_{11} & \frac{X_{1n}X_{2n}}{X_{nn}} & \ldots & \frac{X_{1n}X_{n-1,n}}{X_{nn}} & X_{1n} \\ \frac{X_{1n}X_{2n}}{X_{nn}} & X_{22} & \ldots & \frac{X_{2n}X_{n-1,n}}{X_{nn}} & X_{2n} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ \frac{X_{1n}X_{n-1,n}}{X_{nn}} & \frac{X_{2n}X_{n-1,n}}{X_{nn}} & \ldots & X_{n-1,n-1} & X_{n-1,n} \\ X_{1n} & X_{2n} & \ldots & X_{n-1,n} & X_{nn} \end{pmatrix}.$$

We call $\boldsymbol{X}$ *a positive semidefinite completion matrix* of $\overline{\boldsymbol{X}}$.

Definition 4.1.3. *A matrix $\boldsymbol{X}$ is said to be a completion matrix of $\overline{\boldsymbol{X}}$ if $X_{ij} = \overline{X}_{ij}$ for any (i, j) in the assignment edge set of $\overline{\boldsymbol{X}}$. In addition, if $\boldsymbol{X}$ is positive semidefinite, $\boldsymbol{X}$ is called a positive semidefinite completion matrix of $\overline{\boldsymbol{X}}$.*

Consequently, we can execute PD-IPM with $\overline{\boldsymbol{X}}$ instead of $\boldsymbol{X}$, and after the computation of PD-IPM, we complete $\boldsymbol{X}$ from $\overline{\boldsymbol{X}}$. It is apparent that a benefit owing to the structure of $\overline{\boldsymbol{X}}$ is significant, because the required memory space of $\overline{\boldsymbol{X}}$ is only $2n + 1$ while that of the fully dense matrix $\boldsymbol{X}$ is n^2. For the larger n, the reduction will become more drastic.

In general, however, we should not forget that we can not progress PD-IPM iterations with only the non-zero elements assigned by the aggregate sparsity pattern $\overline{E}$. The reason is that the algorithmic framework of PD-IPM involves the Cholesky Factorization of variable matrices to obtain the step length to keep a condition that we can make a positive semidefinite completion matrix.

The matrix $\overline{\boldsymbol{X}}$ may not be positive semidefinite and even if $\overline{\boldsymbol{X}}$ is positive semidefinite, the assignment edge set $\overline{E}$ may not cover all elements arisen from fill-in due to the Cholesky Factorization. A concept of *clique-PSD* which will be described in section 4.1.2 enables us to retain positive semidefiniteness of $\overline{\boldsymbol{X}}$ in some enough sense to complete $\boldsymbol{X}$.

Now, suppose that $\overline{\boldsymbol{X}}$ is positive definite for a moment to view the fill-in more precisely. For instance, let us consider the Cholesky Factorization on the following sample positive semidefinite matrix $\overline{\boldsymbol{X}}$.

$$\overline{\boldsymbol{X}} = \begin{pmatrix} 5 & 3 & 1 & \\ 3 & 6 & & 2 \\ 1 & & 7 & 4 \\ & 2 & 4 & 8 \end{pmatrix}.$$

The result of the Cholesky Factorization into the lower triangular matrix is

$$\overline{\boldsymbol{L}} = \begin{pmatrix} 2.23607 & & & \\ 1.34164 & 2.04939 & & \\ 0.44721 & -0.29277 & 2.59119 & \\ & 0.97590 & 1.65395 & 2.07655 \end{pmatrix},$$

where $\overline{\boldsymbol{X}} = \overline{\boldsymbol{L}}\,\overline{\boldsymbol{L}}^T$. Note that $(3, 2)$ element of $\overline{\boldsymbol{L}}$ is non-zero, whose position has zero in $\overline{\boldsymbol{X}}$. We call this phenomenon *fill-in*. Therefore, we prepare $\widehat{\boldsymbol{X}}$ whose assignment graph $\widehat{G}(V, \widehat{E})$ cover not only non-zero elements of $\overline{\boldsymbol{X}}$ but also elements emerged from fill-in. To express such assignment edge set $\widehat{E}$, we define *an extended sparsity pattern*.

Definition 4.1.4. *Given an aggregate sparsity pattern $\overline{E}$, we call $\widehat{E}$ an extended sparsity pattern if $\widehat{E}$ covers all non-zero elements of the matrix $\overline{\boldsymbol{X}}$ assigned by $\overline{E}$ and all fill-in elements arisen from the Cholesky Factorization of $\overline{\boldsymbol{X}}$.*

Theoretically speaking, $\widehat{E}$ is determined by the chordal graph, as we will describe in the next subsection 4.1.2. Thus, the memory space for $\widehat{\boldsymbol{X}}$ is minimum to contain all non-zero elements required for the evaluation of primal feasibility, the result of the Cholesky Factorization for the step length and the completion to construct $\boldsymbol{X}$.

However, we have to notice that the evaluation of the Schur complement matrix $\boldsymbol{B}$ whose elements are of form $B_{ij} = (\boldsymbol{X}\boldsymbol{A}_i\boldsymbol{Y}^{-1}) \bullet \boldsymbol{A}_k$ demands the dense matrices $\boldsymbol{X}$ and $\boldsymbol{Y}^{-1}$, even if $\widehat{\boldsymbol{X}}$ and $\boldsymbol{Y}$ posses considerably sparsity. The completion method ensures that if we choose $\boldsymbol{X}$ as a completion matrix whose determinant is maximum in all the completion matrices from $\widehat{\boldsymbol{X}}$, the sparse Cholesky Factorization matrices is available for $\boldsymbol{X}$ and $\boldsymbol{Y}^{-1}$, that is $\boldsymbol{X} = \boldsymbol{M}^{-T}\boldsymbol{M}^{-1}$ and $\boldsymbol{Y} = \boldsymbol{N}\boldsymbol{N}^T$. The matrices $\boldsymbol{M}$ and $\boldsymbol{N}$ are assigned by the extended sparsity pattern $\widehat{E}$. Therefore, the sparse Cholesky Factorization enables us to evaluate $\boldsymbol{B}$ without directly storing dense matrices, $\boldsymbol{X}$ and $\boldsymbol{Y}^{-1}$.

Here, we have reached a framework into which we introduce the sparsity for the primal formulation in general scheme. First, we collect all the indices of non-zero elements of the input data matrices and construct an aggregate sparsity pattern $\overline{E}$. Then, we extend $\overline{E}$ to obtain an extended sparsity pattern $\widehat{E}$ which is characterized by the chordal graph. Let $\boldsymbol{M}$ and $\boldsymbol{N}$ be matrices assigned by $\widehat{E}$. Thus $\boldsymbol{M}$ and $\boldsymbol{N}$ have enough non-zero storage for the primal variable matrix $\boldsymbol{X}$ and the dual variable matrix $\boldsymbol{Y}$, respectively. As a result, PD-IPM can be done with information only sparse matrices $\boldsymbol{M}$ and $\boldsymbol{N}$. In other words, $\boldsymbol{M}$

and $\boldsymbol{N}$ are sufficient for the evaluation of the Schur complement matrix $\boldsymbol{B}$, the computation of the step length and feasibility check for both primal and dual. After all the iteration of PD-IPM, we construct the optimal solution $\boldsymbol{X} = \boldsymbol{M}^{-T}\boldsymbol{M}^{-1}$ and $\boldsymbol{Y} = \boldsymbol{N}\boldsymbol{N}^{T}$.

Based on the framework of PD-IPM with the completion method, SDPA-C was proposed in [24, 59]. We examine three software, SDPA, SDPA-C and SDPARA on an SDP arisen from the relaxation for Max Clique mentioned in section 4.3 to investigate a relation between an effectiveness of the completion method (SDPA-C) and an efficiency of parallel processing (SDPARA on 64 processors). The number of equality constraints of SDP is $m = 1891$ and the number of matrix size is $n = 1000$. Table 4.1 shows the time required by each components of PD-IPM and memory space for the Schur Complement matrix $\boldsymbol{B}$ and the sum total of $n \times n$ matrices.

Table 4.1: Comparison between SDPA,SDPA-C and SDPARA

	SDPA	SDPA-C	SDPARA
ELEMENTS	82.0s	662.8s	7.7s
CHOLESKY	25.3s	34.1s	2.9s
PMATRIX	69.4s	32.6s	69.0s
DENSE	125.7s	2.6s	126.1s
Total Computation time	308s	733s	221s
Memory Space for $\boldsymbol{B}$	27MB	27MB	1MB
Memory Space for $n \times n$ matrices	237MB	8MB	237MB
Total Memory Space	279MB	39MB	265MB

Table 4.1 makes it clear that SDPA-C successfully reduces of the computation time for PMATRIX and DENSE and the memory space for $n \times n$ matrices. It is quite natural because SDPA-C directly reaps the benefits of the completion method. However, SDPA-C requires considerably longer time to evaluate the Schur complement matrix (ELEMENTS) than SDPA, since the memory storages of $\boldsymbol{M}$ and $\boldsymbol{N}$ make it impossible to incorporate into the exploitation of the sparsity described in section 2.3.2 (proposed in [22]).

On the other hand, SDPARA achieves the excellent performance on ELEMENTS and CHOLESKY as shown in section 3.5, while it can not attain any reductions for PMATRIX, DENSE, and the memory space for $n \times n$ matrices. Because the number m of equality constraints is at most twice of the size n of matrices, the non-parallel components PMATRIX and DENSE counterbalance the performance of the parallel processing.

To obtain higher results, we want the benefits from the completion method and the parallel processing simultaneously. This idea leads us to a combination of them. Based on various knowledges of the parallelization for SDPARA, we have implemented SDPARA-C in incorporation of the completion method into SDPARA. It should be emphasized is that not only ELEMENTS and CHOLESKY but also PMATRIX components are parallelized in SDPARA-C. We will mention schemes for the parallelization, after we briefly summarize the theoretical groundwork of the completion method. The numerical experiments reported in section 4.3 and 4.4 will prove the idea that we combine the completion method and the parallel processing attains a significant performance.

4.1.2 Theoretical Groundwork for Positive Semidefinite Matrix Completion

In [24, 59], Fukuda, *et al.* and Nakata, *et al.* brought to light the close link between the chordal graph and the positive semidefinite matrix completion. Shortly, we pick up the important ingredients for the completion method from the theoretical viewpoints. See [24, 59] and their references for more details.

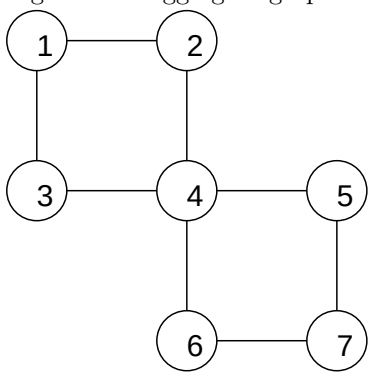
Two substantial materials we focus on here are one that the extended sparsity pattern $\widehat{E}$ covers all the positions of fill-in arisen from the sparse Cholesky Factorization and one that the matrix $\widehat{\boldsymbol{X}}$ assigned by $\widehat{E}$ enables us to complete the positive semidefinite matrix $\boldsymbol{X}$ as an optimal solution. The two materials play primitive roles for us to exploit the sparsity in the primal variable and to attain a significant reduction of computation cost and memory space.

As we have seen, it is apparent that the aggregate sparsity pattern $\overline{E}$ provides enough elements for $\overline{\boldsymbol{X}}$ to be checked the primal feasibility. We derive the aggregate graph $\overline{G}(V, \overline{E})$, whose edge set is the aggregate

sparsity pattern $\overline{E}$. In other words, an edge $\overline{e} = (i, j)$ exists in $\overline{G}$ if (i, j) elements of $\overline{\boldsymbol{X}}$ is non-zero. The following sample 7×7 matrix $\overline{\boldsymbol{X}}$ in (4.1) generates the aggregate graph $\overline{G}$ in Figure 4.1; '+' in $\overline{\boldsymbol{X}}$ stands for non-zero elements. Note that since $\overline{\boldsymbol{X}}$ is symmetric, we ignore the edges $(i, j) \in \overline{E}$ for $i \geq j$ in the Figure 4.1. In addition, we can assume that $\{(i, i) : i \in V\} \subset \overline{E}$.

$$\overline{\boldsymbol{X}} = \begin{pmatrix} + & + & + & & & & \\ + & + & & + & & & \\ + & & + & + & & & \\ & + & + & + & + & + & \\ & & & + & + & & + \\ & & & + & & + & + \\ & & & & + & + & + \end{pmatrix} \tag{4.1}$$

Figure 4.1: Aggregate graph $\overline{G}$



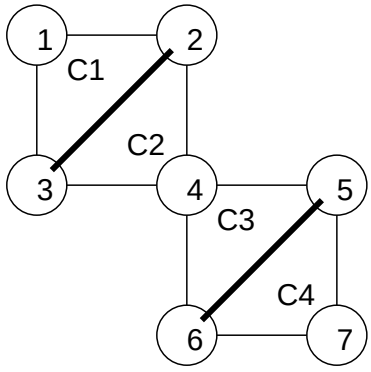
We introduce *an induced graph* to investigate characteristics of a given graph.

Definition 4.1.5. *A graph $G'(V', E')$ is said to be an induced graph of $G(V, E)$ if $V' \subset V$ and $E' = \{(i, j) \in E : i, j \in V'\}$.*

Applying the symbolic Cholesky Factorization, we obtain the matrix $\widehat{\boldsymbol{X}}$ in (4.2), where '*' stands for fill-in elements. Then, we add some edges to $\overline{G}$ to reflect the fill-in, and we build an assigned graph $\widehat{G}$ for $\widehat{\boldsymbol{X}}$ in Figure 4.2. The terminology *chordal* represents a meaningful characteristic of $\widehat{G}$.

$$\widehat{\boldsymbol{X}} = \begin{pmatrix} + & + & + & & & & \\ + & + & * & + & & & \\ + & * & + & + & & & \\ & + & + & + & + & + & \\ & & & + & + & * & + \\ & & & + & * & + & + \\ & & & & + & + & + \end{pmatrix} \tag{4.2}$$

Figure 4.2: Chordal graph $\widehat{G}$



Definition 4.1.6. *A graph $\widehat{G}(V, \widehat{E})$ is defined to be chordal if the length of each cycle in any induced graphs of $\widehat{G}$ is at most three.*

For the chordal graph, a lot of matters has already been discussed [9, 30, 49]. To make a relation of fill-in and the chordal graph more meaningful, we had better understand how to decrease the number of fill-in. Let $G(V, E)$ be a given graph. We say the induced graph $C \subset G$ is a *clique* if C is complete. Next, a vertex $v \in V$ is said to be *simplical* if the graph induced by $\{v\} \cup \{u : (u, v) \in E\}$ is a clique. Now, we suppose that G is a chordal graph. From the characteristic of the chordal graph, we can find a simplical vertex v. Moreover, even after we eliminate the vertex v from G, the left graph induced by $V \backslash \{v\}$ is still the chordal graph. Hence we can eliminate all vertices in a sequential order $\{v_1, v_2, \ldots, v_n\}$ and we generate the sequential induced chordal graphs. We call the order *Perfect Elimination Ordering* in G. In the chordal graph of Figure 4.2, a perfect elimination ordering is, for example, $1, 7, 2, 6, 3, 5, 4$. The perfect elimination is not alway unique for a general chordal graph.

We can summarize a strong connection between the chordal graph and perfect elimination ordering in the following theorem:

Theorem 4.1.7 (Fulkerson and Gross [26]) : *A graph is chordal if and only if it has a perfect elimination ordering.*

In the next step, we focus from the viewpoints of linear algebra. Let $\boldsymbol{Z}$ be an $n \times n$ positive definite matrix to be factorized and $\boldsymbol{P}$ be an $n \times n$ permutation matrix. Note that the $\boldsymbol{PZP}^T$ is always positive semidefinite. Since there may be a difference between the number of fill-in of the Cholesky Factorization to $\boldsymbol{Z}$ and that of $\boldsymbol{PZP}^T$, there is a possibility that an adequate choice of the permutation matrix $\boldsymbol{P}$ may reduce fill-in. For example, the reduction is obvious if $\boldsymbol{Z}$ is left-upper arrow matrix. It is desired that there exists a permutation matrix $\boldsymbol{P}$ which does not produce any fill-in through the Cholesky Factorization on $\boldsymbol{PZP}^T$. If such a permutation exists with respect to $\boldsymbol{Z}$, we call the orderings of rows and columns determined by the permutation matrix $\boldsymbol{P}$ *Perfect Elimination Ordering* on matrix $\boldsymbol{Z}$.

It is apparent from Figure 4.1 and 4.2 that a positive semidefinite matrix $\widehat{\boldsymbol{X}}$ has a perfect elimination ordering if and only if the assigned graph $\widehat{G}(V, \widehat{E})$ has a perfect elimination ordering, since the fill-in of $\widehat{\boldsymbol{X}}$ can be assigned by $\widehat{E}$. Therefore, $\widehat{\boldsymbol{X}}$ characterized by the chordal graph has enough memory space to apply the sparse Cholesky Factorization.

Furthermore, a set of maximal cliques $\{C_1, C_2, \ldots, C_r\}$ which comprises of the chordal graph $\widehat{G}(V, \widehat{E})$ is very useful to detect whether we can complete a positive semidefinite matrix $\boldsymbol{X}$ from the matrix $\widehat{\boldsymbol{X}}$ assigned by $\widehat{E}$. We say a *maximal* clique for a clique which is not contained by any other clique as its induced graph. In addition, we extract sub-matrix $\widehat{\boldsymbol{X}}_{C_k}$ whose rows and columns determined by the vertex set of the maximal clique $C_k (k = 1, 2, \ldots, r)$ from the matrix $\widehat{\boldsymbol{X}}$. We call $\widehat{\boldsymbol{X}}$ is *clique-PSD* if all sub-matrices $\widehat{\boldsymbol{X}}_{C_k} (k = 1, 2, \ldots, r)$ are positive semidefinite.

The next theorem plays another central role in a combination of PD-IPM with the completion method.

Theorem 4.1.8 (Theorem 2.5 in [24]) : *Suppose that $\widehat{G}(V, \widehat{E})$, an assignment graph for $\widehat{\boldsymbol{X}}$, is a chordal graph. Then, we can complete a positive semidefinite matrix $\boldsymbol{X}$ from $\widehat{\boldsymbol{X}}$ if and only if $\widehat{\boldsymbol{X}}$ is clique-PSD.*

In Figure 4.2, the chordal graph can be decomposed into four maximal cliques,

$$\begin{cases} C_1(V_1, E_1) & \text{where} \quad V_1 = \{1,2,3\}, \quad E_1 = \{(1,2),(1,3),(2,3)\} \\ C_2(V_2, E_2) & \text{where} \quad V_2 = \{2,3,4\}, \quad E_2 = \{(2,3),(2,4),(3,4)\} \\ C_3(V_3, E_3) & \text{where} \quad V_3 = \{4,5,6\}, \quad E_1 = \{(4,5),(4,6),(5,6)\} \\ C_4(V_4, E_4) & \text{where} \quad V_4 = \{5,6,7\}, \quad E_4 = \{(5,6),(5,7),(6,7)\} \end{cases}$$

Thus, we can complete from $\widehat{\boldsymbol{X}}$ if clique-PSD as below is satisfied.

$$\widehat{\boldsymbol{X}}_{C_1} = \begin{pmatrix} \widehat{X}_{11} & \widehat{X}_{12} & \widehat{X}_{13} \\ \widehat{X}_{12} & \widehat{X}_{22} & \widehat{X}_{23} \\ \widehat{X}_{13} & \widehat{X}_{23} & \widehat{X}_{33} \end{pmatrix} \succeq \boldsymbol{O}, \quad \widehat{\boldsymbol{X}}_{C_2} = \begin{pmatrix} \widehat{X}_{22} & \widehat{X}_{23} & \widehat{X}_{24} \\ \widehat{X}_{23} & \widehat{X}_{33} & \widehat{X}_{34} \\ \widehat{X}_{24} & \widehat{X}_{34} & \widehat{X}_{44} \end{pmatrix} \succeq \boldsymbol{O},$$

$$\widehat{\boldsymbol{X}}_{C_3} = \begin{pmatrix} \widehat{X}_{44} & \widehat{X}_{45} & \widehat{X}_{46} \\ \widehat{X}_{45} & \widehat{X}_{55} & \widehat{X}_{56} \\ \widehat{X}_{46} & \widehat{X}_{56} & \widehat{X}_{66} \end{pmatrix} \succeq \boldsymbol{O}, \quad \widehat{\boldsymbol{X}}_{C_4} = \begin{pmatrix} \widehat{X}_{55} & \widehat{X}_{56} & \widehat{X}_{57} \\ \widehat{X}_{56} & \widehat{X}_{66} & \widehat{X}_{67} \\ \widehat{X}_{57} & \widehat{X}_{67} & \widehat{X}_{77} \end{pmatrix} \succeq \boldsymbol{O}.$$

Hence, the retain of clique-PSD through all the iterations of PD-IPM guarantees that we acquire a result matrix $\boldsymbol{X}$ as a completion matrix of $\widehat{\boldsymbol{X}}$. Note that the retain is not so difficult, since it is almost the same as the case where we consider all the sub-matrices $\widehat{\boldsymbol{X}}_{C_k}$ $(k = 1, 2, \ldots, r)$ as one block diagonal matrix. Although we may generate multiple matrices when we complete the matrix $\widehat{\boldsymbol{X}}$, however, an actual computation generates a distinguished complete matrix $\boldsymbol{X}$ with the following characteristic.

Theorem 4.1.9 (Grone, Johnson, Sá and Wolkowicz [31]) : *Assume that we can complete the matrix $\widehat{\boldsymbol{X}}$ to be positive semidefinite. Then, there exists a unique completion matrix $\boldsymbol{X}$ such that*

$$\det(\boldsymbol{X}) = \max\{\det(\widetilde{\boldsymbol{X}}) : \widetilde{\boldsymbol{X}} \text{ is a completion matrix of } \widehat{\boldsymbol{X}}\}$$

At the end of this subsection, we summarize the incorporation of the completion method into PD-IPM. We progress PD-IPM with $\widehat{\boldsymbol{X}}$ instead of dense matrix $\boldsymbol{X}$. The matrix $\widehat{\boldsymbol{X}}$ assigned by the extended sparsity pattern $\widehat{E}$ is considerably sparse, however, $\widehat{E}$ assures enough memory space for all the fill-in arisen from the sparse Cholesky Factorization and for check the primal feasibility. In addition, the retain of clique-PSD through all the iteration enables us to complete $\boldsymbol{X}$ from $\widehat{\boldsymbol{X}}$. After the terminal condition of PD-IPM is satisfied, we construct a completion matrix of $\boldsymbol{X}$ that maximizes the determinant. Finally we output $\boldsymbol{X}$ as a primal optimal solution.

4.2 Primal-Dual Interior-Point Methods with the Completion Method and its Parallelization

In this section, we focus on schemes to incorporate the completion method into parallel computation. Hence, we assume that we have already understood the parallel implementation of SDPARA in some depth.

We begin from a fundamental change for the algorithmic framework; we do not use Mehrotra Type Predictor-Corrector PD-IPM adopted by SDPARA. Then we investigate an effect of the completion method on the evaluation of the Schur complement matrix and its parallel implementation. Furthermore, we point out another component of PD-IPM, the computation of the primal variable matrix of the search direction, also can be replaced with its parallel implementation.

Based on these schemes, we have implemented SDPARA-C (SemiDefinite Programming Algorithm paRAllel version with the Completion method). In subsequent sections, we will show numerical results of SDPARA-C.

4.2.1 Adoption of Simple Primal-Dual Interior-Point Methods

As we have mentioned in section 2.3.3, SDPA and its descendant SDPARA employ Mehrotra Type Predictor-Corrector (MT-PC) PD-IPM as their algorithmic frameworks. In MT-PC PD-IPM, we can re-use the Schur complement matrix to obtain the predictor search direction in the computation of the corrector search direction. Since the evaluation of the Schur complement matrix and its Cholesky Factorization (ELEMENTS and CHOLESKY) require most of computation cost for general SDPs, not only the reduction of computation cost but also the almost second-order approximation by the corrector search direction provide us sufficient reasons to employ MT-PC PD-IPM.

However, since SDPs we are trying to solve with the advantage of the completion method in this chapter hold a special structural sparsity, a large portion of computation cost is consumed by the primal component of the search direction (PMATRIX) and computation for $n \times n$ matrices (DENSE). It will be legitimate from viewpoints of a convergence rate that the corrector search direction with the re-use of the Schur complement matrix is inferior to the search direction obtained from the Schur complement matrix which is evaluated for each search direction. In MT-PC, the inferior point is covered by the significant reduction of computation cost due to the re-use. However, as the occupying portions of ELEMENTS and CHOLESKY decrease, the benefit of MT-PC also vanishes. It means that the completion method may compete with MT-PC.

The other perspective of the MT-PC PD-IPM is the matrix $\boldsymbol{R}$ to obtain the right hand side of the Schur complement equation, $r_i = p_i - \boldsymbol{A}_i \bullet ((\boldsymbol{R} - \boldsymbol{X}\boldsymbol{D})\boldsymbol{Y}^{-1}$. The matrix $\boldsymbol{R}$ emerges also in the course of PMATRIX. In MT-PC framework, we need to prepare two versions of $\boldsymbol{R}$, for the predictor and corrector search direction,

$$\boldsymbol{R}_p = \mu_p \boldsymbol{I} - \boldsymbol{X}\boldsymbol{Y}, \quad \boldsymbol{R}_c = \mu_c \boldsymbol{I} - \boldsymbol{X}\boldsymbol{Y} - d\boldsymbol{X}_p d\boldsymbol{Y}_p,$$

respectively. The above definition indicates that the matrix for corrector $\boldsymbol{R}_c$ requires multiplication of the primal and dual matrices of the predictor search direction. Since the matrices $d\boldsymbol{X}_p$ and $d\boldsymbol{Y}_p$ are $n \times n$ matrices, in the completion method, it is preferable to avoid the multiplication to decrease the weight of DENSE component if possible. When we do not compute the corrector search direction, we can consider only one version of $\boldsymbol{R}$,

$$\boldsymbol{R}_p = \mu_p \boldsymbol{I} - \boldsymbol{X}\boldsymbol{Y}.$$

Substituting the definition into the computation of PMATRIX,

$$\begin{aligned}\widehat{d\boldsymbol{X}} &= (\boldsymbol{R} - \boldsymbol{X}d\boldsymbol{Y})\boldsymbol{Y}^{-1} \\ &= (\mu\boldsymbol{I} - \boldsymbol{X}\boldsymbol{Y} - \boldsymbol{X}d\boldsymbol{Y})\boldsymbol{Y}^{-1} \\ &= \mu\boldsymbol{Y}^{-1} - \boldsymbol{X} - \boldsymbol{X}d\boldsymbol{Y}\boldsymbol{Y}^{-1},\end{aligned}$$

enable us to preserve almost sparsity of $\boldsymbol{X} = \boldsymbol{M}^{-T}\boldsymbol{M}^{-1}$ and $\boldsymbol{Y} = \boldsymbol{N}\boldsymbol{N}^{T}$.

From the above observations, SDPA-C and its parallel implementation SDPARA-C do not adopt MT-PC PD-IPM. In other words, they adopt the simple PD-IPM. (Here, *simple* stands for 'without MT-PC'.) In exchange for discarding MT-PC, SDPA-C and SDPARA-C guarantee the only first-order convergence.On the other hand, PD-IPM often work out fine with the special structural sparse SDPs, for example, when we sometime know priori the existence of the interior feasible point and the boundness of objective value. Thus, the first-order convergence has enough likelihood to obtain a sufficient accuracy for an optimal solution. Note that in PD-IPM framework, we acquire only an approximate solution, not an optimal solution itself.

In the simple PD-IPM, SDPA-C solves SDPs effectively due to the incorporation of the completion method. Nevertheless, the computation for PMATRIX is still heavy for a single processors. In the following subsection, we consider the parallel implementation to replace PMATRIX.

4.2.2 Schur Complement Matrix and its Cholesky Factorization

In the implementation of SDPARA, the exploitation of the sparsity for the evaluation of the Schur complement matrix $\boldsymbol{B}$ (ELEMENTS) proposed in [22] plays an essential role to reduce the significant computation cost. However, the exploitation directly accesses the elements of dense matrices $\boldsymbol{X}$ and $\boldsymbol{Y}^{-1}$ which we do not hold in the framework of the completion method. Therefore, we use the formulation proposed in [59] with the assistance of the sparsity matrix $\boldsymbol{M}$ and $\boldsymbol{N}$ which satisfy $\boldsymbol{X} = \boldsymbol{M}^{-T}\boldsymbol{M}^{-1}$ and $\boldsymbol{Y} = \boldsymbol{N}\boldsymbol{N}^{T}$. To introduce the formulation, we need to note that for any symmetric matrices $\boldsymbol{A}$ and $\boldsymbol{B}$,

$$\boldsymbol{A} \bullet \boldsymbol{B} = \mathrm{Tr}(\boldsymbol{A}\boldsymbol{B}) = \Sigma_{k=1}^{n}[\boldsymbol{A}\boldsymbol{B}]_{kk} = \Sigma_{k=1}^{n} e_k^T \boldsymbol{A}\boldsymbol{B} e_k = \Sigma_{k=1}^{n}(\boldsymbol{A}e_k)^T[\boldsymbol{B}]_{*k},$$

where Tr stands for the trace of a matrix, e_k is the unit vector whose only the kth component is one, $[\boldsymbol{X}]_{kk}$ is the (k,k) element of $\boldsymbol{X}$ and $[\boldsymbol{X}]_{*k}$ denotes the kth column vector of $\boldsymbol{X}$.

Here, we compose the formula to compute B_{ij}, elements of the Schur complement matrix $\boldsymbol{B}$, and $r_i (i = 1, 2, \ldots, m)$, the right hand side of the Schur complement equation, acquiring the benefits of sparsity of $\boldsymbol{M}$ and $\boldsymbol{N}$. We start the composition from the Schur complement to obtain the HRVW/KSH/M search direction described in section 2.3.1. Furthermore, we fix $\boldsymbol{R} = \mu\boldsymbol{I} - \boldsymbol{X}\boldsymbol{Y}$, since we do not adopt the MT-PC PD-IPM but the simple PD-IPM.

$$\begin{aligned}B_{ij} &= (\boldsymbol{X}\boldsymbol{A}_i\boldsymbol{Y}^{-1}) \bullet \boldsymbol{A}_j \\ &= \mathrm{Tr}(\boldsymbol{X}\boldsymbol{A}_j\boldsymbol{Y}^{-1}\boldsymbol{A}_i) \\ &= \Sigma_{k=1}^{n} e_k^T(\boldsymbol{X}\boldsymbol{A}_j\boldsymbol{Y}^{-1}\boldsymbol{A}_i)e_k \\ &= \Sigma_{k=1}^{n}(\boldsymbol{X}e_k)^T\boldsymbol{A}_j\boldsymbol{Y}^{-1}[\boldsymbol{A}_i]_{*k} \\ &= \Sigma_{k=1}^{n}(\boldsymbol{M}^{-T}\boldsymbol{M}^{-1}e_k)^T\boldsymbol{A}_j(\boldsymbol{N}^{-T}\boldsymbol{N}^{-1}[\boldsymbol{A}_i]_{*k}) \qquad (4.3) \\ r_i &= p_i - \boldsymbol{A}_i \bullet ((\boldsymbol{R} - \boldsymbol{X}\boldsymbol{D})\boldsymbol{Y}^{-1}) \\ &= p_i - \boldsymbol{A}_i \bullet ((\mu\boldsymbol{I} - \boldsymbol{X}\boldsymbol{Y} - \boldsymbol{X}\boldsymbol{D})\boldsymbol{Y}^{-1}) \\ &= (p_i + \boldsymbol{A}_i \bullet \boldsymbol{X}) - \mu\boldsymbol{A}_i \bullet \boldsymbol{Y}^{-1} + \boldsymbol{A}_i \bullet (\boldsymbol{X}\boldsymbol{D}\boldsymbol{Y}^{-1}) \\ &= b_i - \mu\mathrm{Tr}(\boldsymbol{Y}^{-1}\boldsymbol{A}_i) + \mathrm{Tr}(\boldsymbol{X}\boldsymbol{D}\boldsymbol{Y}^{-1}\boldsymbol{A}_i) \\ &= b_i - \mu\Sigma_{k=1}^{n} e_k^T(\boldsymbol{Y}^{-1}[\boldsymbol{A}_i]_{*k}) + \Sigma_{k=1}^{n}(\boldsymbol{X}e_k)^T\boldsymbol{D}\boldsymbol{Y}^{-1}[\boldsymbol{A}_i]_{*k} \\ &= b_i - \mu\Sigma_{k=1}^{n} e_k^T(\boldsymbol{N}^{-T}\boldsymbol{N}^{-1}[\boldsymbol{A}_i]_{*k}) + \Sigma_{k=1}^{n}(\boldsymbol{M}^{-T}\boldsymbol{M}^{-1}e_k)^T\boldsymbol{D}(\boldsymbol{N}^{-T}\boldsymbol{N}^{-1}[\boldsymbol{A}_i]_{*k}) \\ & \qquad (4.4)\end{aligned}$$

It should be emphasized that all the computations involving $\boldsymbol{M}$ and $\boldsymbol{N}$ are multiplications between their inverse and vectors. Calculating $\boldsymbol{M}^{-1}\boldsymbol{v}$ for some vector $\boldsymbol{v}$ is equivalent to solve the system $\boldsymbol{M}\boldsymbol{w} = \boldsymbol{v}$ as follow, because $\boldsymbol{M}$ and $\boldsymbol{N}$ are the lower triangular matrices.

For $i = 1, 2, \ldots, n$
$\quad w_i = (v_i - \Sigma_{j=1}^{i-1} M_{ij} w_j)/M_{ii}$
end(For)

Hence, it is apparent that the sparsity of $\boldsymbol{M}$ decreases the computation of the summention. The computation $\boldsymbol{M}^{-T}\boldsymbol{v}$ are also done in the same way. In addition, the bilinear inner-product in the form $\boldsymbol{v}^T\boldsymbol{A}\boldsymbol{w}$ in the last step of B_{ij} and of r_i can be computed effectively, since $\boldsymbol{A}_i (i = 1, 2, \ldots, m)$ and $\boldsymbol{D}$ are assigned by the aggregate sparsity pattern which is a subgraph of the extended sparsity pattern. We need to remember that the completion method adequately acts if the extended sparsity pattern of the SDP holds a sufficient sparsity. Therefore, we can consider that the above formulas actively incorporate into the sparsity of the SDP.

The other feature of the above formulations is that we can skip in the summention with respect to k if the kth vector of $\boldsymbol{A}_i$ is zero vector. For example, since all $\boldsymbol{A}_i (i = 1, 2, \ldots, m)$ in Max Cut Problem described in section 3.2.1 has only one non-zero elements, the effect based on the skip is remarkable.

From the viewpoint of parallel processing, the formula of the Schur complement matrix $\boldsymbol{B}$ in (4.3) reserves the row-wise computation. As we have mentioned in section 3.2.2, we assign the computation of the ith row of $\boldsymbol{B}$, where $i \in \mathcal{P}_u$, to the uth processor over row-wise memory distribution on distributed memory. Furthermore, $\mathcal{P}_u$ is comprised of some belts in rotation with size of belt sb,

$$\mathcal{P}_u = \cup\{i \in \mathcal{B}_w : \ w\%N = u\}, \quad \text{where } \mathcal{B}_w = \{i : (w-1) \times sb \leq i \leq \min\{m, w \times sb\}\}.$$

The concept of the row-wise distribution is basically quite useful even when we incorporate into the completion method, since the above formula for B_{ij} can be done independently and individually by the same implementation as SDPARA.

The most significant difference we are facing is that the skip in the summention of k due to the considerably sparsity of $\boldsymbol{A}_i$ may break the appropriate load-balance attained by SDPARA. Let us consider the case we apply the above formula to SDPs arisen from Theta Function Problem described in section 3.2.1. Only one input data matrix, for example $\boldsymbol{A}_1$, is an identity matrix and each other input data matrix $\boldsymbol{A}_i (i = 2, 3, \ldots, m)$ has only two non-zero elements. In other words, we can not skip any column vector of $\boldsymbol{A}_1$, while only two column vectors of $\boldsymbol{A}_i (i = 2, 3, \ldots, m)$ are taken into the computation. Since $\boldsymbol{M}$ and $\boldsymbol{N}$ are identical for all rows in $\boldsymbol{B}$ and the computation of their inverse does not depend on the sparsity of the vector to be multiplied, the processor assigned to the row with respect to $\boldsymbol{A}_1$ requires quite larger amount of computation cost than other processors.

To avoid the above worse load-balance, the row with respect to $\boldsymbol{A}_i$ which has many non-zero columns should be computed by multiple processors. Let $\overline{\mathcal{B}} \subset \{1, 2, \ldots, m\}$ be a index set of such rows. Suppose that N processors are available for us now. Then we hash the columns of $\boldsymbol{A}_i$ ($i \in \overline{\mathcal{B}}$) into N disjoint sets $\mathcal{Q}_1^i, \mathcal{Q}_2^i, \ldots, \mathcal{Q}_N^i$ such that

$$\cup_{u=1}^{n} \mathcal{Q}_u^i = \{k : 1 \leq k \leq n, \ [\boldsymbol{A}_i]_{*k} \neq \boldsymbol{0}\} \ \text{ and } \ \mathcal{Q}_u^i \cap \mathcal{Q}_v^i = \phi \ \text{ for } u \neq v.$$

However, the division poses the problem of the memory space on distributed memory space. If we divide each row of memory space according to $\mathcal{Q}_1^i, \mathcal{Q}_2^i, \ldots, \mathcal{Q}_N^i$, the memory space for the entire matrix $\boldsymbol{B}$ may be hashed, and a lot of communication will be required. Therefore, we should keep the row-wise distribution of memory space based on $\mathcal{P}_u (u = 1, 2, \ldots, N)$. Hence, to compute the ith row, we prepare on each processor an temporary vector $\boldsymbol{b}_u$ and an temporary value s_u for elements of the ith row of $\boldsymbol{B}$ and the right hand side of the Schur complement equation $\boldsymbol{r}$, respectively. Here, $[\boldsymbol{b}_u]_j$ denotes the jth component of $\boldsymbol{b}_u$.

$$\begin{aligned}
[\boldsymbol{b}_u]_j &= \Sigma_{k \in \mathcal{Q}_u^i} (\boldsymbol{M}^{-T}\boldsymbol{M}^{-1}e_k)^T \boldsymbol{A}_j (\boldsymbol{N}^{-T}\boldsymbol{N}^{-1}[\boldsymbol{A}_i]_{*k}) \\
s_u &= -\mu \Sigma_{k \in \mathcal{Q}_u^i} e_k^T (\boldsymbol{N}^{-T}\boldsymbol{N}^{-1}[\boldsymbol{A}_i]_{*k}) + \Sigma_{k \in \mathcal{Q}_u^i} (\boldsymbol{M}^{-T}\boldsymbol{M}^{-1}e_k)^T \boldsymbol{D} (\boldsymbol{N}^{-T}\boldsymbol{N}^{-1}[\boldsymbol{A}_i]_{*k})
\end{aligned}$$

To acquire B_{ij} and r_i, we need to collect the results of $\boldsymbol{b}_u$ and s_u of all processors. Hence, we send the result from the uth vector to the vth processor, which holds original memory space for the ith row ($i \in \mathcal{P}_v$), and compute the summention of $\boldsymbol{b}_u$ and s_u to obtain the ith row of $\boldsymbol{B}$ and the ith element of $\boldsymbol{r}$ on the vth processor. Finally, we store the evaluated them into distributed memory space of the vth processor.

After the computation for rows in $\overline{\mathcal{B}}$, we evaluate the rest rows in the same way as SDPARA. Since the rest rows come from the relatively sparse matrix, the serious trouble with respect to load-balance do not occur in general. In this case, no communication is required between multiple processors, since the processor holds memory space of elements which are computed by the processor itself.

We call a combined distribution of a row-wise distribution for $\mathcal{B}$ and a hashed distribution for $\overline{\mathcal{B}}$ *a hashed row-wise distribution*. On the contrary, we call a row-wise distribution adopted by SDPARA as *a simple row-wise distribution*.

Consequently, the algorithmic framework of the evaluation of the Schur complement matrix $\boldsymbol{B}$ and the right hand side $\boldsymbol{r}$ on each processor can be expressed as follow.

Evaluation of the Schur complement matrix $\boldsymbol{B}$ and the right hand side $\boldsymbol{r}$ on the uth processor in the style of a hashed row-wise distribution

Set $\boldsymbol{B} = \boldsymbol{O}$ and $\boldsymbol{r} = \boldsymbol{b}$
For $i \in \overline{\mathcal{B}}$
  Decide processor v such that $i \in \mathcal{P}_v$
  Set $\boldsymbol{b}_u = \boldsymbol{0}$ and $s_u = 0$.
  For $k \in \mathcal{Q}_u^i$
    Compute $\boldsymbol{g} = \boldsymbol{M}^{-T}\boldsymbol{M}^{-1}e_k$, $\boldsymbol{h} = \boldsymbol{N}^{-T}\boldsymbol{N}^{-1}[\boldsymbol{A}_i]_{*k}$
    For $j \in 1, 2, \ldots, m$
      Compute $\boldsymbol{g}^T\boldsymbol{A}_j\boldsymbol{h}$ and add to $[\boldsymbol{b}_u]_j$
    end(For:j)
    Compute $-\mu e_k^T\boldsymbol{h} + \boldsymbol{g}^T\boldsymbol{D}\boldsymbol{h}$ and add to s_u
  end(For:k)
  Send $\boldsymbol{b}_u$ and s_u to the vth processor
  If $u = v$
    Compute $\Sigma_{u=1}^N \boldsymbol{b}_u$ and store into the ith row of $\boldsymbol{B}$
    Compute $\Sigma_{u=1}^N s_u$ and add to r_i
  end(If)
end(For:i)
For $i \in \mathcal{P}_u \setminus \overline{\mathcal{B}}$
  For $k \in \{k : 1 \leq k \leq n, [\boldsymbol{A}_i]_{*k}$is not zero vector$\}$
    Compute $\boldsymbol{g} = \boldsymbol{M}^{-T}\boldsymbol{M}^{-1}e_k$, $\boldsymbol{h} = \boldsymbol{N}^{-T}\boldsymbol{N}^{-1}[\boldsymbol{A}_i]_{*k}$
    For $j \in 1, 2, \ldots, m$
      Compute $\boldsymbol{g}^T\boldsymbol{A}_j\boldsymbol{h}$ and add to B_{ij}
    end(For:j)
    compute $-\mu e_k^T\boldsymbol{h} + \boldsymbol{g}^T\boldsymbol{D}\boldsymbol{h}$ and add to r_i
  end(For:k)
end(For:i)

In actual implementation, a kind of estimation is required to determine which rows are included by $\overline{\mathcal{B}}$, in other words, which rows had better be computed by multiple processors. SDPARA-C adopts the following simple threadhold to determine whether the ith row to be put in $\overline{\mathcal{B}}$ or not.

$$nz(\boldsymbol{A}_i) \geq \sqrt{n},$$

where $nz(\boldsymbol{A}_i)$ is the number of non-zero elements of $\boldsymbol{A}_i$. The $nz(\boldsymbol{A}_i)$ is slightly different from the number of non-zero columns of $\boldsymbol{A}_i$ what we want. However, the threshold works well as numerical results will show in the following section.

Once we construct $\boldsymbol{B}$ and $\boldsymbol{r}$, we solve the Schur complement equation $\boldsymbol{B}\ d\boldsymbol{z} = \boldsymbol{r}$ to obtain $d\boldsymbol{z}$. In the course, we adopt the parallel Cholesky Factorization. The computation to obtain $d\boldsymbol{z}$ is identical to the process on two-dimensional block-cyclic distributed memory which we have mentioned in section 3.2.3.

4.2.3 Parallel Computation for Primal Variable Matrix of the Search Direction

After the acquisition of $d\boldsymbol{z}$ as the solution of the Schur complement equation, each processor receives it on their own memory space. Then, we compute the dual variable matrix of the search direction,

$$d\boldsymbol{Y} \quad = \quad \boldsymbol{C} - \Sigma_{k=1}^{m} \boldsymbol{A}_k d\boldsymbol{z}_k.$$

Since the sparsity of the input data matrices is inherited directly into the dual variable matrix, $d\boldsymbol{Y}$ can be assigned by the aggregate sparsity pattern $\overline{E}$. The computation to obtain $d\boldsymbol{Y}$ is so cheap that we compute it on each processor without any high communication between multiple processors.

On the other hand, the primal variable $d\boldsymbol{X}$, the PMATRIX component of PD-IPM, is more complicated. The matrix $d\boldsymbol{X}$ is essentially computed through the multiplications between matrices and symmetrization. Although the PMATRIX is not replaced by a parallel implementation in SDPARA, we need to focus on the component because the component consumes relatively much portion of computation cost, in particular, if the completion method is incorporated. Therefore, we investigate an implementation to apply parallel processing to PMATRIX.

PMATRIX is done by the computations $\widehat{d\boldsymbol{X}} = (\boldsymbol{R} - \boldsymbol{X}d\boldsymbol{Y})\boldsymbol{Y}^{-1}$ and $d\boldsymbol{X} = (\widehat{d\boldsymbol{X}} + \widehat{d\boldsymbol{X}}^T)/2$. Here, we use an auxiliary matrix $\widetilde{d\boldsymbol{X}}$ instead of $\widehat{d\boldsymbol{X}}$, because we use $\widehat{d\boldsymbol{X}}$ to indicate the matrix assigned by the extended sparsity pattern $\widehat{E}$ characterized by the chordal graph. Hence, a relation between the three matrices $d\boldsymbol{X}, \widetilde{d\boldsymbol{X}}$ and $\widehat{d\boldsymbol{X}}$ as follow.

First, from the $d\boldsymbol{Y}$ assigned by the aggregate sparsity pattern, we compute $\widetilde{d\boldsymbol{X}} = (\boldsymbol{R} - \boldsymbol{X}d\boldsymbol{Y})\boldsymbol{Y}^{-1}$. Then $\widehat{d\boldsymbol{X}}$ is composed of the elements assigned by $\widehat{E}$ of the symmetrized matrix $(\widetilde{d\boldsymbol{X}} + \widetilde{d\boldsymbol{X}}^T)/2$. The point is that we can equate $d\boldsymbol{X}$ with $\widehat{d\boldsymbol{X}}$, because both matrices requires to be assigned by $\widehat{E}$. Note that discarding some non-zero elements in the course of construction for $\widehat{d\boldsymbol{X}}$ enforces us a non-linear search, substituting linear search $\boldsymbol{X} + \alpha_p d\boldsymbol{X}$. However, the non-linear search is enough for a sequence in PD-IPM to converge to an optimal solution [59].

Since we store only $\boldsymbol{M}$ and $\boldsymbol{Y}$ on each processor instead of $\boldsymbol{X}$ and $\boldsymbol{Y}$ as we have mentioned in the above section 4.1.2, we prefer to utilize the information of $\boldsymbol{M}$ and $\boldsymbol{N}$. The column-wise computation for $\widetilde{d\boldsymbol{X}}$ proposed in [59] will be a practical from viewpoints of not only the completion method but also parallel processing.

$$\begin{aligned}
[\widetilde{d\boldsymbol{X}}]_{*k} &= [(\boldsymbol{R} - \boldsymbol{X}d\boldsymbol{Y})\boldsymbol{Y}^{-1}]_{*k} \\
&= [(\mu\boldsymbol{I} - \boldsymbol{X}\boldsymbol{Y} - \boldsymbol{X}d\boldsymbol{Y})\boldsymbol{Y}^{-1}]_{*k} \\
&= [-\boldsymbol{X} + \mu\boldsymbol{Y}^{-1} - \boldsymbol{X}d\boldsymbol{Y}\boldsymbol{Y}^{-1}]_{*k} \\
&= [-\boldsymbol{X}]_{*k} + \mu\boldsymbol{Y}^{-1}e_k - \boldsymbol{X}d\boldsymbol{Y}\boldsymbol{Y}^{-1}e_k \\
&= [-\boldsymbol{X}]_{*k} + \mu\boldsymbol{N}^{-T}\boldsymbol{N}^{-1}e_k - \boldsymbol{M}^{-T}\boldsymbol{M}^{-1}d\boldsymbol{Y}\boldsymbol{N}^{-T}\boldsymbol{N}^{-1}e_k
\end{aligned}$$

Remember that the multiplication of the inverse of $\boldsymbol{M}$ and $\boldsymbol{N}$ with vector is considerably cheap, because $\boldsymbol{M}$ and $\boldsymbol{N}$ are the sparse triangular lower matrices. In addition, $d\boldsymbol{Y}$ is assigned by the aggregate sparsity pattern, then the total computation cost of the above formula can be reasonably suppressed.

The column-wise computation provides us a natural scheme to apply parallel processing to PMATRIX. Let $\mathcal{R}_u$ be the column set computed by the uth processor such that

$$\cup_{u=1}^{N} \mathcal{R}_u = \{k : 1 \leq k \leq n\} \text{ and } \mathcal{R}_u \cap \mathcal{R}_v = \phi \text{ for } u \neq v.$$

Since the computation cost for each column is almost the same, $\mathcal{R}_u$ can be determined in a simple manner even regarding load-balance.

$$\begin{array}{lcl}
\mathcal{R}_u & = & \{k : (u-1) \times \lfloor n/N \rfloor + 1 \leq k \leq u \times \lfloor n/N \rfloor\} \quad (u = 1, 2, \ldots, N-1) \\
\mathcal{R}_N & = & \{k : (N-1) \times \lfloor n/N \rfloor + 1 \leq k \leq n\}
\end{array},$$

where $\lfloor x \rfloor$ is the largest integer that does not exceed x. We call the distribution $\{\mathcal{R}_u : u = 1, 2, \ldots, N\}$ *a serial-columns distribution*. As a result, the computation for $\widetilde{d\boldsymbol{X}}$ on the uth processor is described as follow.

The Computation of Intermediate matrix $\widetilde{d\boldsymbol{X}}$ on the uth processor

Set $\widetilde{d\boldsymbol{X}} = -\boldsymbol{X}$.
For $k \in \mathcal{R}_u$
  Compute $\mu \boldsymbol{N}^{-T}\boldsymbol{N}^{-1}e_k - \boldsymbol{M}^{-T}\boldsymbol{M}^{-1}d\boldsymbol{Y}\boldsymbol{N}^{-T}\boldsymbol{N}^{-1}e_k$
  and add to $[\widetilde{d\boldsymbol{X}}]_{*k}$
end(For)
Broadcast all columns in $\mathcal{R}_u$ to all other processors
Receive the rest columns from all other processors

In the last two step of the above framework, we require communication to store the entire matrix $\widetilde{d\boldsymbol{X}}$ on all processors. The matrix $\widehat{d\boldsymbol{X}}$, however, is assigned by the extended sparsity pattern $\widehat{E}$. Therefore, the elements to be transmitted are limited by $\widehat{E}$ and the amount of the communication in PMATRIX will not be so siginicant. In addition, the serial-columns distribution $\{\mathcal{R}_u : u = 1, 2, \ldots, N\}$ requires smaller communication time than a stereotypic cyclic-columns distribution.

The application of parallel processing to compute the primal component will produce a meaningful reduction of computation cost. The effect will be shown in numerical results.

4.2.4 SDPARA-C (SemiDefinite Programming Algorithm paRAllel Version with the Completion Method)

We have implemented SDPARA-C (SemiDefinite Programming Algorithm paRAllel version with the Completion method) based on the above-mentioned schemes to apply parallel processing to the evaluation of the Schur complement matrix (ELEMENTS) and the computation of the primal variable matrix of the search direction (PMATRIX) with advantage of the completion method. A main objective of SDPARA-C is clearly to merge the merits of SDPA-C and SDPARA.

For the implementation of SDPARA-C, the knowledge of the communication acquired through SDPARA is essential. We have integrated two libraries, MPI (Message Passing Interface) and ScaLAPACK (Scalable Linear Algebra PACKage), into SDPARA-C.

Here, we want to focus on the data storage on each processor which we have not mentioned so far. In the same way as SDPARA, all the input data from an input file, $\boldsymbol{A}_1, \boldsymbol{A}_2, \ldots, \boldsymbol{A}_m$, $\boldsymbol{C}, b_1, b_2, \ldots, b_m$ are stored on each processor. In addition, the Schur complement matrix $\boldsymbol{B}$ is stored twice, the row-wise distribution for the evaluation of elements and two-dimensional block-cyclic distribution for the parallel Cholesky Factorization.

The point we have to consider involved in the completion method is the storage of the variable matrices, $\boldsymbol{X}$ and $\boldsymbol{Y}$, and the component of search direction $d\boldsymbol{X}$ and $d\boldsymbol{Y}$. As we have mentioned, we store $\boldsymbol{M}$ and $\boldsymbol{N}$ assigned by the extended sparsity pattern $\widehat{E}$ instead of $\boldsymbol{X}$ and $\boldsymbol{Y}$. Generally speaking, since $\boldsymbol{X}$ and $\boldsymbol{Y}^{-1}$ are considered to be fully dense matrices, we can not hold them on each processors. However, the numbers of non-zero elements involved in the matrices $\boldsymbol{M}, \boldsymbol{N}, d\boldsymbol{X}, d\boldsymbol{Y}$ are limited by the edge number of $\widehat{E}$. Therefore, we can maintain the four matrices on each processor. The storage on each processor results in no communication to evaluate the rows of the Schur complement matrix $\boldsymbol{B}$ that are not members of $\overline{\mathcal{B}}$.

In short, the information divided into distributed memory space on each processor is only the Schur complement matrix $\boldsymbol{B}$. The computation of the primal variable matrix of the search direction (PMATRIX) requires communication between multiple processors, however, we do not divide the memory space of $\widetilde{d\boldsymbol{X}}$ and $d\boldsymbol{X}$ on multiple processors to cut the ineconomical communication.

It should be mentioned that SDPARA-C itself do not construct the aggregate sparsity pattern $\overline{E}$ and the extended sparsity pattern $\widehat{E}$. Instead, we utilize an additional highly technical program to generate such information before we apply SDPARA-C. Since the generation requires file input/output, it should not be applied parallel processing. Thus, if we measure the time including the generation, it makes difficult to measure an immediate effect brought by parallel processing. Hence, we assume in the following numerical results that the information of $\overline{E}$ and $\widehat{E}$ are supplied through the given input file with the input data matrices.

As we have described in the first paragraph of this subsection, merging the merit of SDPA-C and SDPARA is the main objective of SDPARA-C. Here, we show a portion of numerical results, which clearly proves the objective is accomplished. Table 4.2 presents Table 4.1 in section 4.1.1 with the additional results of SDPARA-C on 64 processors.

Table 4.2: Comparison between SDPA,SDPA-C,SDPARA and SDPARA-C

	SDPA	SDPA-C	SDPARA	SDPARA-C
ELEMENTS	82.0s	662.8s	7.7s	10.5s
CHOLESKY	25.3s	34.1s	2.9s	4.0s
PMATRIX	69.4s	32.6s	69.0s	2.4s
DENSE	125.7s	2.6s	126.1s	2.3s
Total Computation time	308s	733s	221s	26s
Memory Space for $\boldsymbol{B}$	27MB	27MB	1MB	1MB
Memory Space for $n \times n$ matrices	237MB	8MB	237MB	8MB
Total Memory Space	279MB	39MB	265MB	41MB

Inheriting the characteristic from SDPARA, the computation time of SDPARA-C for the evaluation of the Schur complement matrix $\boldsymbol{B}$ (ELEMENTS) and its Cholesky Factorization (CHOLESKY) are successfully reduced. Moreover, the memory space for $\boldsymbol{B}$ is divided into all processors.

Additionally, the completion method enables us to compute the primal variable matrix of the search direction (PMATRIX) and linear algebra with respect to $\boldsymbol{X}, \boldsymbol{Y}, d\boldsymbol{X}, d\boldsymbol{Y}$ (DENSE) with very cheap costs and save a lot of memory space for $n \times n$ matrices. Furthermore, the parallel implementation of PMATRIX which does not belong to SDPARA decreases the significant computation time.

In consequence, SDPARA-C solves the SDP with the shortest computation time in the above four software. In the following full-scale numerical results, we investigate the performance of SDPARA-C on various SDPs. We also explore the effect of the sparsity of the input data.

4.3 Numerical Results

Our numerical experiments of SDPARA-C were done on PC-Cluster Presto III. The specs of Presto III have been described in section 3.3.

In this section, we focus on the performance of SDPARA-C. Comparisons with other software will be subjects in the subsequent section. First, we investigate computation time required by SDPARA-C to solve various SDPs. Then, we examine SDPs which have different sparsity, because the efficiency of the completion method is strongly dependent of the sparsity of the input data matrices; thus the performance of SDPARA-C may also depends on the sparsity.

4.3.1 Scalability of SDPARA-C

To investigate a fundamental performance of SDPARA-C, we have applied it to SDPs in Table 4.3. The problem cut-10-500 is a Max Cut Problem mentioned in section 3.2.1. Additionally, clique-10-200 and norm-10-990 are SDPs from Max Clique Problem and Min Norm Problem, respectively. Here, we describe the definition of these problems shortly. The rest four SDPs have been selected from SDPLIB [11].

Table 4.3: SDPs for numerical experiments

name	m	n		
cut-10-500	5000	5000	Max Cut Problem	with $P = 10, Q = 500$
clique-10-200	3791	2000	Max Clique Problem	with $P = 10, Q = 200$
norm-10-990	11	1000	Min Norm Problem	with $P = 10, Q = 990$
control10	1326	150	SDPLIB	
theta6	4375	300	SDPLIB	
maxG51	1000	1000	SDPLIB	
qpG11	1600	800	SDPLIB	

SDP relaxation for Max Clique Problem

Let $G(V, E)$ be an undirected graph with a vertex set $V = \{1, 2, \ldots, n\}$ and an edge set $E \subset \{(i, j) : i, j \in V\}$. We assume $C(V_c, E_c)$ is max clique in G, that is, C is complete and the vertex number of C is not smaller than that of any other clique in G. Let K be the cardinality of V_c. Therefore K is to be maximized. We introduce variables $x_1, x_2, \ldots, x_n$ to indicate $v_i \in V \ \ (x_i > 0)$ or $v_i \notin V \ \ (x_i = 0)$.

Then, finding max clique in G is equivalent to solve the following maximization problem.

$$\max\{\Sigma_{i=1}^{n}\Sigma_{j=1}^{n} x_i x_j : x_i x_j = 0((i, j) \notin E), \Sigma_{i=1}^{n} x_i^2 = 1\}.$$

The constraint $x_i x_j = 0$ $((i, j) \notin E)$ ensures that if there does not exist an edge between v_i and v_j, at most one of the vertices can be contained in the clique. On the other hand, $\Sigma_{i=1}^{n} x_i^2 = 1$ is added to bound the objective value from above. We have $x_i = 1/\sqrt{K}$ $(v_i \in V_c)$ and $x_i = 0$ $(v_i \notin V_c)$ when optimal is attained. Therefore the optimal objective value of the maximization problem is K.

Since Max Clique problem as well as Max Cut Problem is NP-complete problem, SDP relaxation method is essential to acquire an approximate optimal value in polynomial time. From Max Clique Problem, we obtain standard SDP.

$$\max\{\boldsymbol{E} \bullet \boldsymbol{X} : \boldsymbol{E}_{ij} \bullet \boldsymbol{X} = 0((i, j) \notin E),\ \boldsymbol{I} \bullet \boldsymbol{X} = 1,\ \boldsymbol{X} \succeq \boldsymbol{O}\},$$

where $\boldsymbol{E}$ is the $n \times n$ matrix whose all elements are one, $\boldsymbol{E}_{ij}$ is the $n \times n$ matrix whose only (i, j) and (j, i) elements are one and other elements are zero. Furthermore, we apply an transformation proposed in [24] to the above SDP to change its aggregate sparsity pattern to suit the completion method. We usually say SDP relaxation arisen from Max Clique Problem as Max Clique Problem.

Min Norm Problem

Given $\boldsymbol{F}_p \in \mathbb{R}^{q \times r}$ $(i = 0, 1, 2, \ldots, p)$, a subject of Min Norm Problem is to find a linear combination of the matrices with the minimum norm in all linear combinations. In a mathematical formulation, the norm minimization problem is described as

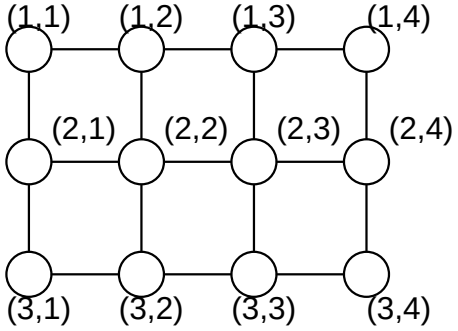
$$\min\{||\boldsymbol{F}_0 + \Sigma_{k=1}^{p} \boldsymbol{F}_i z_i|| : z_i \in \mathbb{R}\ (i = 1, 2, \ldots, p)\},$$

where the norm $||\boldsymbol{X}||$ is square root of a maximal eigenvalue of $\boldsymbol{X}^T\boldsymbol{X}$. The minimization problem is equivalent to the following dual SDP.

$$\max\left\{-z_{p+1} : \boldsymbol{Y} = \sum_{i=1}^{p} \begin{pmatrix} \boldsymbol{O} & \boldsymbol{F}_i^T \\ \boldsymbol{F}_i & \boldsymbol{O} \end{pmatrix} z_i + \begin{pmatrix} \boldsymbol{I} & \boldsymbol{O} \\ \boldsymbol{O} & \boldsymbol{I} \end{pmatrix} z_{p+1} + \begin{pmatrix} \boldsymbol{O} & \boldsymbol{F}_0^T \\ \boldsymbol{F}_0 & \boldsymbol{O} \end{pmatrix},\ \boldsymbol{Y} \succeq \boldsymbol{O}\right\}.$$

We construct the SDPs, cut-10-500, clique-10-200 and norm-10-990 in Table 4.3 based on lattice graphs. Here, let us take a look at a definition of lattice graph. Figure 4.3 shows a lattice graph with size 4×3. Vertices of lattice graph correspond to lattice nodes and edges connecting vertices comprises of the structure of lattice itself.

Figure 4.3: Lattice graph with size 4×3



In general, let $G(V, E)$ be a lattice graph with size $P \times Q$. Then V and E are precisely defined as

$$\begin{aligned} V &= \{(i,j) : 1 \le i \le P,\ 1 \le j \le Q\} \\ E &= \{(\ (i,j),\ (i,j+1)\ ) : 1 \le i \le P,\ 1 \le j \le Q-1\} \\ &\quad \cup \{(\ (i,j),\ (i+1,j)\ ) : 1 \le i \le P-1,\ 1 \le j \le Q\}. \end{aligned}$$

Therefore, Max Cut Problem based on a lattice graph with size $P \times Q$ has the number of equality constraints $m = P \times Q$ and the size of matrices $n = P \times Q$. In the same way, we have $m = 2P \times Q - P - Q + 1, n = P \times Q$ in Max Clique Problem and $n = P + Q$ in Min Norm Problem. Note that the number m of equality constraints in Min Norm Problem is independent from the structure of lattice graph, since $m = p + 1$ where p is the number of matrices involved in the linear combination.

The matrix assigned by the lattice graph has the same property of the coefficient matrix arisen from a differential equation. Hence, it is not difficult to construct an ordering which reduces the fill-in of the Cholesky Factorization. It means that we can estimate the sparsity of the extended sparsity pattern if the matrices assigned by the lattice graph. We define a sparsity ρ of an SDP as an average number of non-zero elements on each row of an extended sparsity pattern generated by its input data matrices. A short consideration leads us that the sparsity ρ is at most $2 \times \min\{P, Q\} + 1$. This property clearly fits the completion method we are focusing in this chapter.

So far, we take a look at the formulation of cut-10-500, clique-10-200, norm-10-990. Meanwhile, control10, theta6, maxG51 and qpG11 are picked up from SDPLIB [11]. They are Control Problem, Theta Function Problem, Max Cut Problem and Quadratic Partition Problem, respectively.

Table 4.4 shows numerical results of SDPARA-C with changing the number of processors. In the table, we skip the components of PD-IPM that can be computed in less than 20 seconds by even a single processor.

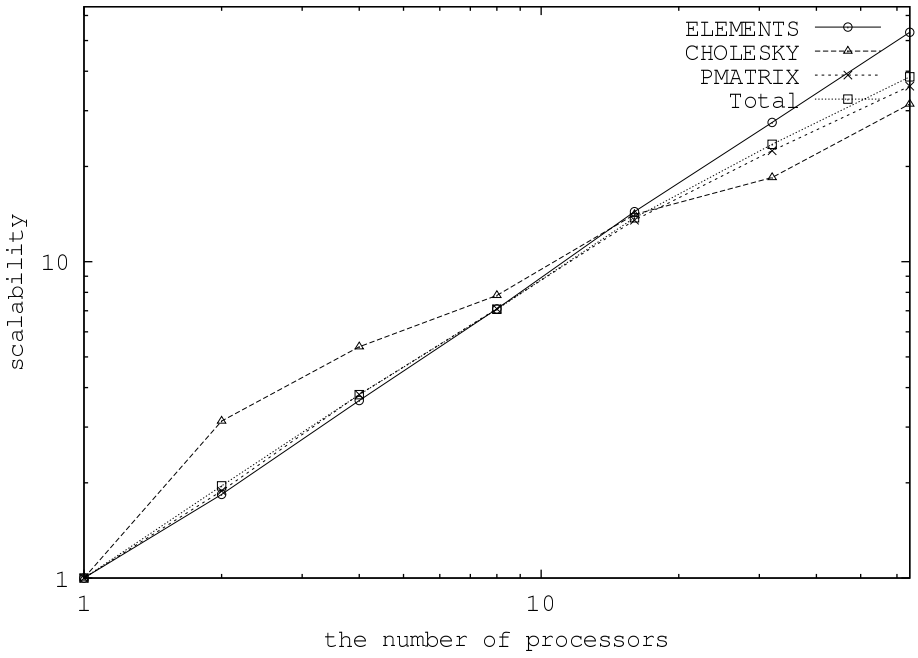
Table 4.4: Performance of SDPARA-C

the number of processors		1	2	4	8	16	32	64
cut-10-500	ELEMENTS	937.0	482.5	270.4	149.1	74.6	40.9	23.0
	CHOLESKY	825.1	253.4	142.0	90.0	49.7	35.5	19.9
	PMATRIX	459.5	234.7	120.4	60.6	30.9	15.9	9.2
	Total	2239.7	982.5	544.4	310.6	166.8	105.0	70.7
clique-10-200	ELEMENTS	2921.9	1589.1	802.8	412.6	203.6	106.6	55.6
	CHOLESKY	538.9	171.9	100.1	68.9	38.2	29.2	17.1
	PMATRIX	197.4	104.7	51.9	27.8	14.6	8.8	5.5
	Total	3670.1	1876.0	966.2	519.1	266.5	156.2	95.6
norm-10-990	ELEMENTS	29.4	15.0	8.2	3.8	2.1	1.9	1.6
	PMATRIX	28.8	14.9	7.7	3.4	2.2	1.7	2.2
	Total	66.0	37.5	23.4	14.1	14.9	13.3	20.4
control10	ELEMENTS	27341.8	14294.4	7444.1	4022.5	2276.6	1400.5	1001.9
	CHOLESKY	66.9	34.0	22.2	18.2	11.3	10.2	7.5
	Total	27437.3	14351.8	7488.2	4060.5	2308.7	1432.3	1035.1
theta6	ELEMENTS	1743.1	958.5	522.7	281.8	140.3	76.6	42.8
	CHOLESKY	898.7	275.0	155.9	105.1	57.9	43.8	25.3
	PMATRIX	51.9	27.1	14.8	9.2	7.4	6.1	10.8
	Total	2714.4	1278.9	711.5	412.3	222.3	145.3	101.9
maxG51	ELEMENTS	228.1	126.3	65.2	34.9	17.9	10.2	6.3
	PMATRIX	220.1	116.5	60.1	33.5	20.3	14.0	18.4
	DENSE	26.8	26.8	26.4	26.8	26.7	26.4	26.9
	Total	485.7	276.7	157.2	100.6	70.1	56.9	61.1
qpG11	ELEMENTS	42.1	24.0	12.8	6.9	3.6	2.1	1.4
	PMATRIX	24.8	13.1	6.7	3.5	1.9	1.2	0.9
	Total	73.5	42.8	24.9	15.5	10.9	9.9	12.68

First, we pay attention the scalability of each component of PD-IPM in clique-10-200. The scalability is depicted in Figure 4.4. It is clear that all parallelized components, ELEMENTS, CHOLESKY and

PMATRIX, attain excellent scalability. The surprising scalability in ELEMENTS is also shown in SDPARA-C. On the other hand, what we have to recognize on here is the scalability of PMATRIX which is not replaced by parallel implantation in SDPARA. Although PMATRIX offers a less scalability than ELEMENTS, the reduction of computation cost in PMATRIX strongly affect the scalability of the total computation time. If we do not apply parallel processing to PMATRIX, the scalability of the total computation time in 64 processors remains only 12.7, almost one-third of SDPARA-C.

Figure 4.4: Scalability of SDPARA-C for clique-10-200



Next, let us overlook all the problems reported in Table 4.4. SDPARA-C shows high scalability for, in particular, cut-10-500, clique-10-200, norm-10-990. The fact the problems are constructed on lattice graphs implies us that if the extended sparsity pattern is successfully assembled keeping the sparsity of the input data matrices, not only the completion method but also parallel processing work effectively.

For control10 and theta6, the scalability is not so worse. However, ELEMENTS requires much longer time compared to the case SDPARA. Since we have the factorized matrices $\boldsymbol{M}$ and $\boldsymbol{N}$ instead of $\boldsymbol{X}$ and $\boldsymbol{Y}$, we can not exploit the sparsity in the input data matrices proposed in [22]. It should not be underestimated that we need to consider scalability and total computation time simultaneously.

On the other hand, the scalability for maxG51 and qpG11 are not so good. More than 16 processors, it is difficult to attain some scalability for these problems. The issue of qpG11 is that the problem is too small to solve on 64 processors. In fact, SDPARA-C solves the problem with enough scalability up to 16 processors. It is natural that scalability becomes worse if we assign an excessive number of processors to small problems. However, the issue of maxG51 is essentially different from that of qpG11. The issue is the ratio of DENSE to Total computation time. Since the components is processed on each processors, the scalability is bounded if SDPARA-C requires most of computation cost on DENSE. However, the computation time for DENSE is much smaller than the case of SDPARA owing to the advantage of the completion method.

4.3.2 Effect of Sparsity

To investigate effect of the completion method in SDPARA-C, we solve Max Cut Problems by SDPARA-C on 64 processors changing the sparsity of the input data matrices. In Max Cut Problem, we can easily generate SDPs involving the same m and n and different ρ, where m is the number of equality constraints, n is the size of matrices $\boldsymbol{X}$ and $\boldsymbol{Y}$, and ρ is the sparsity, that is, an average number of non-zero elements on

each row of an extended sparsity pattern. Here, we fix $m = n = 1000$ and change ρ as $3, 10, 30, 100, 300, 1000$. (Note that $\rho = 1000$ means the extended sparsity matrix is full dense.) Table 4.5 shows the time in second required by each component of PD-IPM and memory space.

Table 4.5: Effect of sparsity for SDPARA-C

sparsity	3	10	30	100	300	1000
ELEMENTS	0.7	1.1	1.5	6.5	23.9	31.6
CHOLESKY	0.7	1.1	0.7	0.8	1.1	1.1
PMATRIX	0.6	0.9	3.0	18.6	99.5	108.7
DENSE	0.2	0.9	3.2	19.4	226.3	187.7
Total	9.4	12.7	15.8	53.7	364.2	344.3
Memory(MB)	31	33	48	152	644	471

As the increment of sparsity ρ, the time for PMATRIX and DENSE positively grow. Table 4.5 reconfirms us the fact that the two components are strongly affected by the completion method and the memory storage of $\boldsymbol{M}$ and $\boldsymbol{N}$. On the other hand, CHOLESKY component is constant for the sparsity, because the parallel Cholesky Factorization can be applied even without the completion method.

From the above numerical experiments, SDPARA-C solves sparse SDPs effectively on the small memory space. Conversely, when the sparsity if not enough, SDPARA-C suffers from longer computation time and requires larger memory space. In other words, the sparsity of the extended sparsity pattern deeply affects the performance of SDPARA-C.

4.4 Comparison with SDPARA and PDSDP

In the previous chapter, we proposed SDPARA. Then, we have proposed another parallel implementation SDPARA-C in this chapter. In this subsection, we compare the performance of the two software and PDSDP [4].

We start from the scalability on various SDPs. Then, we investigate the effect of the sparsity of SDPs and examine how large SDPs can be solved by each software. Finally, the performance of software for SDPs from SDPLIB [11] and 7th DIMACS implementation challenge problem library (semidefinite and related optimization problems) are reported.

4.4.1 Scalability for Various SDPs

In section 4.3.1, we have applied SDPARA-C to SDPs listed in Table 4.3. To compare the performance, we also apply SDPARA and PDSDP to the same SDPs.

Table 4.6 shows the computation time consumed by each software to solve the SDPs. In the table, 'M' indicates memory over. Since SDPARA can not exploit the sparsity in the input data matrices at any point in the variable matrices $\boldsymbol{X}$ and $\boldsymbol{Y}$, SDPARA generally requires more memory space than SDPARA-C and PDSDP.

In the same way as section 4.3.1, we start from scalability on cut-10-500 and control10 depicted in Figure 4.5. We can confirm that SDPARA-C attains the highest scalability in the three software. The reason is that almost all time-consuming components of PD-IPM are replaced by their parallel implementation in SDPARA-C. Therefore, SDPARA-C is adequately to parallel processing. The numerical results in Table 4.6 show the high scalability of SDPARA-C on the other problems.

However, SDPARA-C requires more computation time on control 10 than SDPARA, even though the scalability of SDPARA-C is higher than that of SDPARA. Since the numbers m of equality constraints of control10 and theta6 are relatively greater than the sizes of variable matrices $\boldsymbol{X}$ and $\boldsymbol{Y}$, the parallel implementation with respect to only the Schur complement matrix $\boldsymbol{B}$ is enough and the effect of PMATRIX is small. Additionally, SDPARA-C needs much computation cost for $\boldsymbol{B}$ because of the storage of $\boldsymbol{M}$ and $\boldsymbol{N}$ instead of $\boldsymbol{X}$ and $\boldsymbol{Y}$, even when we have developed the further parallelized implementation. Meanwhile, SDPARA directly exploits the sparsity for ELEMENTS through the method described in section

Table 4.6: Performance of SDPARA-C, SDPARA and PDSDP on multiple processors

the number of processors		1	2	4	8	16	32	64
cut-10-500	SDPARA-C	2239.7	982.5	544.4	310.6	166.8	105.0	70.7
	SDPARA	M	M	M	M	M	M	M
	PDSDP	2129.0	1894.6	1672.8	1604.9	1559.5	1652.0	1752.08
clique-10-200	SDPARA-C	3670.3	1876.0	966.0	519.1	266.5	156.2	95.6
	SDPARA	M	M	M	M	M	M	M
	PDSDP	5295.0	5222.3	4879.2	4764.1	4718.5	4743.2	4777.8
norm-10-990	SDPARA-C	66.0	37.5	23.4	14.1	14.9	13.7	20.4
	SDPARA	1372.6	893.4	566.4	440.3	414.3	407.2	431.1
	PDSDP	458.7	265.9	180.3	138.4	99.7	100.5	102.6
control10	SDPARA-C	27437.1	14351.5	7488.8	4061.2	2308.2	1432.8	1035.6
	SDPARA	429.1	233.0	128.3	75.2	42.7	302.	21.9
	PDSDP	2101.2	1913.6	727.8	468.3	210.7	213.9	207.0
theta6	SDPARA-C	2714.4	1278.9	711.5	412.3	222.3	145.3	101.9
	SDPARA	697.0	332.2	169.0	112.6	66.2	50.0	39.7
	PDSDP	555.5	463.6	320.8	288.7	250.8	324.3	370.4
maxG51	SDPARA-C	485.7	276.7	157.2	100.6	70.1	56.9	61.0
	SDPARA	174.9	177.5	176.0	175.8	174.5	178.0	178.7
	PDSDP	82.6	82.1	79.3	82.6	83.6	96.9	109.8
qpG11	SDPARA-C	73.5	42.8	24.5	15.5	10.9	9.9	12.68
	SDPARA	639.8	650.2	651.0	650.3	651.3	653.0	655.8
	PDSDP	43.1	43.1	40.8	41.8	43.3	51.9	60.1

2.3.2. Therefore, the performance of SDPARA on Control Problem is higher than SDPARA-C. Conversely, SDPARA-C is well-suited for SDPs with the condition $m \leq n$.

On the other hand, the performance of PDSDP shows similar characteristic to SDPARA-C. If $m << n$, PDSDP also solves SDPs effectively. Comparing SDPARA-C and PDSDP, SDPARA-C solves all the SDPs listed in Table 4.3 much faster than PDSDP, in particular, in the case more processors are participated, except only control10.

4.4.2 Effect of Size and Sparsity

We examine how large SDPs can be solved by each software. Table 4.7, 4.8 and 4.9 shows the time and the memory space to solve Max Cut Problems, Max Clique Problems and Min Norm Problems, respectively. These SDPs are solved on Presto III with 64 processors. We can generate the SDPs with various sizes keeping the sparsity ρ based on lattice graphs. More precisely, if we have a lattice graph with size $P \times Q$, the sparsity of the generated SDPs can be bounded as $\rho \leq 2\min(P, Q) + 1$. Meanwhile, the number of equality constraints is $m = P \times Q$ (Max Cut Problem), $m = 2P \times Q - P - Q + 1$ (Max Clique Problem), and the size of matrices is $n = P \times Q$ (Max Cut Problem, Max Clique Problem), $n = P \times Q$ (Norm Minimization Problem). Therefore, by fixing $P = 10$ and changing Q from 100 to 4000, we generate SDPs keeping the same sparsity in various sizes.

In the three tables, Table 4.7, 4.8 and 4.9, 'M' stands for memory over. SDPARA is not strong against the increment of n. Since SDPARA holds the variable matrices $\boldsymbol{X}$ and $\boldsymbol{Y}$ as full dense matrices on each processor, the exhaust of memory space soon becomes gross amount. The upper bound for SDPARA is at most $n = 1000$. The results of PDSDP are slightly better than SDPARA; PDSDP also can not exploit the sparsity so effectively.

In the three software, SDPARA-C solves the largest problems with $n = 40000$. Due to the completion method, SDPARA-C reserves a lot of memory space and attains adequate performance for the sparse and large SDPs.

In turn, we fix the size of matrices $n = m = 10000$ and change the sparsity ρ from 3 to 1000. We generate Max Cut Problems to examine the effect of sparsity in the same way as section 4.3.2. Table 4.10 shows the numerical results of the three software. As we have mentioned with respect to SDPARA-C, the

Figure 4.5: Scalability of SDPARA-C, SDPARA and PDSDP for cut-10-500 and control10

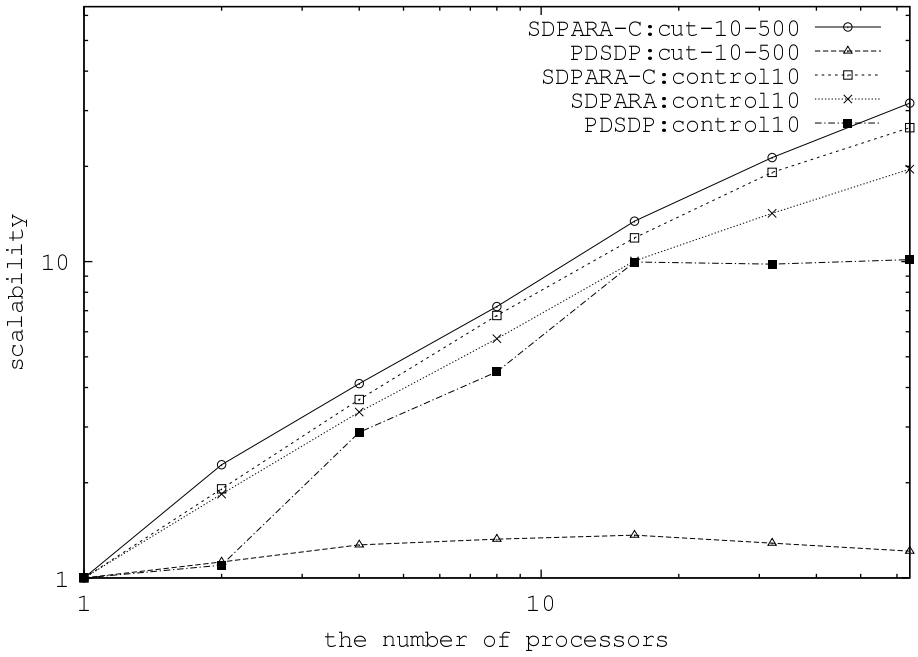


Table 4.7: Large-Scale Max Cut Problem

size of lattice	n	SDPARA-C time (s)	SDPARA-C memory (MB)	SDPARA time (s)	SDPARA memory (MB)	PDSDP time (s)	PDSDP memory (MB)
10×100	1000	10.2	35	164.3	262	54.7	36
10×200	2000	16.6	42		M	192.0	72
10×500	5000	69.3	70		M	1731.1	317
10×1000	10000	274.3	126		M		M
10×2000	20000	1328.2	276		M		M
10×4000	40000	7462.0	720		M		M

increment of the sparsity sharply means the increment of the computation time and the memory space. PDSDP has the same tendency as SDPARA-C, although the tendency is milder than SDPARA-C. On the other hand, the sparsity does not have any effect on the performance of SDPARA. Since SDPARA does not take the sparsity in the primal variables into consideration, SDPARA is robust no matter how dense the SDPs are.

The conclusion from the above numerical results is that we had better apply SDPARA-C to sparse SDPs and SDPARA to dense SDPs. In addition, PDSDP attains middle performance between SDPARA-C and SDPARA and is useful to limit the memory space in the small amount.

4.4.3 SDPs from SDPLIB and DIMACS

To evaluate the performance of software for the standard benchmark problems, we pick up 7 SDPs from SDPLIB and 4 SDPs from 7th DIMACS implementation challenge problem library (semidefinite and related optimization problems). The problems equalG11 and equalG51 are Max Cut Problems with the constraints that two divided partitions have the same number of vertices. The problems which start with 'torus' are also Max Cut Problems with the special structure defined by model of spin glasses. The reason why 'torus'

Table 4.8: Large-Scale Max Clique Problem

size of lattice	n	SDPARA-C time (s)	SDPARA-C memory (MB)	SDPARA time (s)	SDPARA memory (MB)	PDSDP time (s)	PDSDP memory (MB)
10×100	1000	28.1	41	225.6	265	684.8	50
10×200	2000	93.9	58		M	4776.0	119
10×500	5000	639.5	119		M		M
10×1000	10000	3033.2	259		M		M
10×2000	20000	15329.0	669		M		M

Table 4.9: Large-Scale Min Norm Problem

size of lattice	n	SDPARA-C time (s)	SDPARA-C memory (MB)	SDPARA time (s)	SDPARA memory (MB)	PDSDP time (s)	PDSDP memory (MB)
10×990	1000	16.6	40	417.2	262	107.9	35
10×1990	2000	32.3	54		M	653.0	63
10×4990	5000	96.9	97		M		M
10×9990	10000	409.5	164		M		M
10×19990	20000	1800.9	304		M		M
10×39990	40000	7706.0	583		M		M

SDPs are selected from DIMACS is that the extended sparsity patterns do not become so dense, with the aid of the transformation proposed in [24]. In Table 4.11, the time (in second) and the amount of memory space ('mem' in Mega Bytes) to solve the SDPs by SDPARA-C, SDPARA and PDSDP are shown. In the table, m is the number of equality constraints, n is the size of matrices and ρ is sparsity. 'M' stands for memory over and 'I' indicates that PDSDP can not reach an optimal solution in a predefined iteration number.

SDPARA-C can solve effectively the large SDPs, maxG32, thetaG51, torusg3-15 and toruspm3-15-50 which SDPARA can not solve because of memory over. The size n of matrix size of these problems is over 2000, and SDPARA is required to hold full dense variable matrices with size n on each processor. On the other hand, SDPARA-C hold the only elements assigned by the extended sparsity pattern, and reduces a lot of amount of memory space for the variable matrices.

Furthermore, SDPARA-C reaches the optimal solution which PDSDP can not reach optimal solution. Generally speaking, PD-IPM adopted by SDPARA-C is considered to be more stable than D-IPM adopted by PDSDP. The result reflects the difference in the stability.

Table 4.10: Effect of sparsity on SDPARA-C, SDPARA and PDSDP

ρ	SDPARA-C time (s)	SDPARA-C memory (MB)	SDPARA time (s)	SDPARA memory (MB)	PDSDP time (s)	PDSDP memory (MB)
3	9.4	31	156.0	262	44.6	36
10	12.7	33	175.7	262	61.6	36
30	15.8	48	175.1	262	61.3	36
100	53.7	152	182.5	263	126.5	38
300	364.2	644	168.4	263	291.6	41
1000	344.3	471	160.2	263	276.5	42

Table 4.11: Performance for SDPLIB and DIMACS

				SDPARA-C		SDPARA		PDSDP	
	m	n	ρ	time	mem	time	mem	time	mem
thetaG11	2401	801	23	22.7	15	130.3	182	I	
maxG32	2000	2000	32	31.8	51		M	229.6	73
equalG11	801	801	35	17.2	40	141.3	177	57.3	32
qpG51	2000	1000	67	654.8	139	416.4	287	500.5	65
control11	1596	165	74	2017.6	84	29.9	67	307.2	42
thetaG51	6910	1001	137	107.9	107		M	I	
equalG51	1001	1001	534	528.5	482	230.1	263	400.3	41
torusg3-8	512	512	78	14.7	51	45.4	88	26.0	27
toruspm3-8-50	512	512	78	14.8	51	34.7	88	25.7	27
torusg3-15	3375	3375	212	575.0	463		M	1958.9	165
toruspm3-15-50	3375	3375	212	563.3	463		M	1841.9	165

From Table 4.11, we confirm again that SDPARA-C solves very effectively if SDP holds sufficient sparsity. In particular, the performance of SDPARA-C is prominent compared to other two software on large sparse problem, such as torusg3-15 and toruspm3-15-50.

4.5 Theoretical Validity of Parallel Implementation in SDPARA-C

In this section, we discuss a theoretical validity of the parallel implementation of SDPARA-C. In SDPARA-C, we have replaced three components, ELEMENTS (section 4.2.2), CHOLESKY and PMATRIX (section 4.2.3) with their parallel implementation. Among them, the parallel implementation of CHOLESKY is the same as the one used in SDPARA. Hence, we examine the two rest components in this section. To advance discussions, we use the three values t_f, t_v and t_m in Table 3.19 again.

For ELEMENTS component, we proposed the hashed row-wise distribution as a substitute for the simple row-wise distribution of SDPARA. Here, let us observe an effect of the hashed row-wise distribution comparing with the simple row-wise distribution.

To evaluate the Schur complement matrix $\boldsymbol{B}$, we employ the following formula based on the completion method.

$$B_{ij} \quad = \quad \Sigma_{k=1}^{m}(\boldsymbol{M}^{-T}\boldsymbol{M}^{-1}e_k)^T\boldsymbol{A}_j(\boldsymbol{N}^{-T}\boldsymbol{N}^{-1}[\boldsymbol{A}_i]_{*k})$$

Let $\mathcal{B} = \{1, 2, \ldots, m\}$ be the set of row indices of $\boldsymbol{B}$, $\overline{\mathcal{B}} \subset \mathcal{B}$ be a set of row indices to be hashed and $\overline{m} = |\overline{\mathcal{B}}|$. We assume $\overline{m} << N << m$ where N is the number of available processors. This assumption often comes into effect in large-scale sparse SDPs arisen from combinatorial optimization. We use $nz(ex)$ to denote the number of non-zero elements involved in the extended sparsity and let $\overline{n}(i) = |\{k : 1 \leq k \leq n, [\boldsymbol{A}_i]_{*k} \neq \boldsymbol{0}\}|$. Then the computation cost for the ith row is

$$t_r(i) = \overline{n}(i) \times 4 \times nz(ex) \times 2 \times \Sigma_{j=1}^{n} nz(j) \times t_f.$$

In addition, let $t_r(\overline{\mathcal{B}}) = \Sigma_{i \in \overline{\mathcal{B}}} t_r(i), \quad t_r(\mathcal{B} - \overline{\mathcal{B}}) = \Sigma_{i \in \mathcal{B} - \overline{\mathcal{B}}} \quad t_r(i)$.

In the manner of the simple row-wise distribution, the uth processor computes rows assigned by $\mathcal{P}_u$,

$$\mathcal{P}_u = \{i : 1 \leq i \leq m, i\%N = u\}.$$

Then the computation time for ELEMENTS, T_{simple}, is

$$T_{simple} \quad = \quad \max_u\{\Sigma_{i \in \overline{\mathcal{B}}, i\%N=u} t_r(i)\} + \max_u\{\Sigma_{i \in \mathcal{B}-\overline{\mathcal{B}}, i\%N=u} t_r(i)\}.$$

On the other hand, in the hashed row-wise distribution, we hash non-zero column vectors of $\boldsymbol{A}_i$ ($i \in \overline{\mathcal{B}}$) into sets of column indices, $\cup_{u=1}^{N} Q_u^i = \{k : 1 \leq k \leq n, [\boldsymbol{A}_i]_{*k} \neq \boldsymbol{0}\}$. Hence, for the ith row in $\overline{\mathcal{B}}$, the uth processor computes

$$B_{ij}^{(u)} \quad = \quad \Sigma_{k \in Q_u^i} (\boldsymbol{M}^{-T} \boldsymbol{M}^{-1} e_k)^T \boldsymbol{A}_j (\boldsymbol{N}^{-T} \boldsymbol{N}^{-1} [\boldsymbol{A}_i]_{*k}).$$

We assume that $\overline{n}(i)$ is sufficiently larger than N ($i \in \overline{B}$). This assumption is also natural for SDPs arisen from combinatorial optimization. Then, the computation cost of $B_{ij}^{(u)}$ becomes almost the same over all the processors. Since the rest rows in $\mathcal{B} - \overline{\mathcal{B}}$ are evaluated in the same manner as the simple row-wise distribution, the computation time of the hashed row-wise distribution, T_{hashed}, is

$$T_{hashed} \quad = \quad \Sigma_{i \in \overline{\mathcal{B}}} tr(i)/N + \max_u \{\Sigma_{i \in \mathcal{B} - \overline{\mathcal{B}}, i\%N=u} t_r(i)\}$$

However, the hashed row-wise distribution needs communication cost T_{acc} to accumulate the hashed rows over all processors to obtain $B_{ij} = \Sigma_{u=1}^{N} B_{ij}^{(u)}$ ($i \in \overline{\mathcal{B}},\ j = 1, 2, \ldots, m$). T_{acc} can be estimated as follow.

$$T_{acc} \quad = \quad \overline{m}(mt_v + t_m) \log_2 N.$$

Consequently, the hashed row-wise distribution evaluates the Schur complement matrix in a shorter time than the simple row-wise distribution if

$$\begin{aligned} T_{simple} - (T_{hashed} + T_{acc}) \quad &= \quad \max_u \{\Sigma_{i \in \overline{\mathcal{B}}, i\%N=u} t_r(i)\} - (\Sigma_{i \in \overline{\mathcal{B}}} tr(i)/N + \overline{m}(mt_v + t_m) \log_2 N) \\ &= \quad \left(\max_u \{\Sigma_{i \in \overline{\mathcal{B}}, i\%N=u} \overline{n}(i)\} - \Sigma_{i \in \overline{\mathcal{B}}} \overline{n}(i)/N\right) \times 8 \times nz(ex) \times \Sigma_{j=1}^{n} nz(j) \times t_f \\ &\quad -\overline{m}(mt_v + t_m) \log_2 N \end{aligned}$$

is positive.

In Max Clique Problem, only $\boldsymbol{A}_1$ is an identity matrix and the number of non-zero elements involved in each other input matrix ($\boldsymbol{A}_2, \boldsymbol{A}_3, \ldots, \boldsymbol{A}_m$) is two. In this case, $\overline{\mathcal{B}} = \{1\}$ and $\overline{n}(1) = 1$, hence,

$$T_{simple} - (T_{hashed} + T_{acc}) \quad = \quad (1 - 1/N) t_r(1) + (mt_v + t_m) \log_2 N.$$

Therefore, the computation cost for $\boldsymbol{A}_1$ becomes prominent when the number of processors increase.

From numerical experiments (Table 4.12 and 4.13) on Max Clique Problem ($m = 3971, n = 2000$, on lattice graph with size 10×200), we obtain the real values $t_r(1) = 573$, $t_r(\mathcal{B} - \overline{\mathcal{B}}) = 2420$, $T_{acc} \leq 14$ and the range of t_v is $1.6 \times 10^{-6} \leq t_v \leq 1.8 \times 10^{-4}$. (The communication time for the accumulation depends on N.) These value indicates that $T_{simple} - (T_{hashed} + T_{acc}) > 0$ when $N \geq 2$.

Table 4.12: Simple row-wise distribution for Max Clique Problem

the number of processors	1	2	4	8	16	32	64
$t_r(\mathcal{B})$ (second)	3039.7	1788.2	1210.0	927.4	769.6	689.8	645.8
Total (second)	3855.0	2200.1	1457.3	1088.9	855.5	755.9	698.0

Table 4.13: Hashed row-wise distribution for Max Clique Problem

the number of processors	1	2	4	8	16	32	64
$t_r(\mathcal{B})$ (second)	2993.0	1471.9	717.4	369.5	180.8	93.1	45.3
$t_r(1)/N$ (second)	573.6	295.9	152.7	76.8	37.6	19.2	9.7
T_{acc} (second)	0.0	11.4	13.5	5.6	2.2	1.1	0.6
Total (second)	3814.0	1840.4	967.3	518.6	272.9	162.6	100.0

The case of Max Clique Problem is an obvious example. However, the simple row-wise distribution always suffers from the ill-conditioned load-balance of $tr(\overline{\mathcal{B}})$ and requires more computation time than the communication time for the accumulation provided the assumptions $\overline{n}(i) >> N$ and $\overline{m} << N << m$ is valid. We often encounter the imbalance in the number of non-zero column vectors. In particular, the

minorities of input matrices involve a lot of non-zero column vectors in SDPs arisen from combinatorial optimization. Hence, it would be appear that the assumptions are reasonable. On the other hand, the hashed row-wise distribution hashes the computation of $tr(\overline{\mathcal{B}})$ into multiple processors and reduces the number of idle processors. Therefore, the hashed row-wise distribution is faster than the simple row-wise distribution, even though it requires the additional communication cost.

The last parallel component of SDPARA-C we discuss in this section is PMATRIX. To evaluate a primal search direction $\widetilde{d\boldsymbol{X}}$, we based on the column-wise formula proposed in [59],

$$[\widetilde{d\boldsymbol{X}}]_{*k} \quad = \quad [-\boldsymbol{X}]_{*k} + \mu \boldsymbol{N}^{-T}\boldsymbol{N}^{-1}e_k - \boldsymbol{M}^{-T}\boldsymbol{M}^{-1}d\boldsymbol{Y}\boldsymbol{N}^{-T}\boldsymbol{N}^{-1}e_k.$$

We divide the column vectors of $\widetilde{d\boldsymbol{X}}$ by a serial-column distribution.

$$\begin{array}{lcl} \mathcal{R}_u & = & \{k : (u-1) \times \lfloor n/N \rfloor + 1 \leq k \leq u \times \lfloor n/N \rfloor\} \quad (u = 1, 2, \ldots, N-1) \\ \mathcal{R}_N & = & \{k : (N-1) \times \lfloor n/N \rfloor + 1 \leq k \leq n\} \end{array}$$

Then algorithmic framework on the uth processor is summarized as follow.

Computation of $\widetilde{d\boldsymbol{X}}$ on the uth processor

Set $\widetilde{d\boldsymbol{X}} = -\boldsymbol{X}$.
For $k \in \mathcal{R}_u$
  Compute $\mu \boldsymbol{N}^{-T}\boldsymbol{N}^{-1}e_k - \boldsymbol{M}^{-T}\boldsymbol{M}^{-1}d\boldsymbol{Y}\boldsymbol{N}^{-T}\boldsymbol{N}^{-1}e_k$
  and add to $[\widetilde{d\boldsymbol{X}}]_{*k}$
end(For)
Broadcast all columns in $\mathcal{R}_u$ to all other processors
Receive the rest columns from all other processors

In the above framework, we broadcast only the elements assinged by the extended sparsity. First, we investigate how much communication cost is reduced as compared to the case when we broadcat full-elements of $\widetilde{d\boldsymbol{X}}$. Let ρ be the average number of non-zero elements in each column of the extended sparsity.

When we broadcast only the elements of the extended sparsity, we repeat broadcast of the length of $\rho \times n/N$ for N times. On the other hand, in the case of full-elements, we repeat broadcast of the length of $n \times n/N$ for N times. Since the length of each broadcast is different, we use two values $t_v(extended)$ and $t_v(full)$ for the communication time to broadcast one 'double' value, respectively. Then, the total communication time in the case of the extended sparsity $T_{extended}$ and that of full-elements T_{full} are estimated as follow.

$$\begin{array}{rcl} T_{extended} & = & (t_v(extended) \times \rho \times n/N \times \log_2 N + t_m) \times N \\ T_{full} & = & (t_v(full) \times n \times n/N \times \log_2 N + t_m) \times N \end{array}$$

Therefore,

$$T_{extended} \leq T_{full} \quad \Longleftrightarrow \quad \rho \times t_v(extended) \leq n \times t_v(full).$$

It is common $\rho < n/100$ for SDPs the completion method solves effectively. In addition, we obtained $t_v(extended) \leq 10 \times t_v(full)$ from the numerical experiments on such SDPs. Therefore, the broadcast with only elements assigned by the extended sparsity is faster than full-elements. The point regarding the communication cost is that we can attain enough bandwidth in the case of the broadcast with the length $\rho \times n/N$ even when we compare with the case of the length $n \times n/N$. These results are consistent on the numerical results on Presto III. In Table 4.14, we compare the broadcast time for $\widetilde{d\boldsymbol{X}}$ in the case of the extended sparsity and full-elements. The problems cut-10-500 and clique-10-200 are the same problems listed in Table 4.3 and cut-10-200 is a smaller Max Cut Problem on lattice graph with size 10×200. The unit size of time $T_{extended}$, T_{full}, $t_v(extended)$, and $t_v(full)$ in Table 4.14 is second.

Table 4.14: Communication time to broadcast $\widetilde{d\boldsymbol{X}}$

	ρ	n	$T_{extended}$	$t_v(extended)$	T_{full}	$t_v(full)$
cut-10-200	9	2000	0.017	1.51×10^{-7}	0.80	3.33×10^{-8}
cut-10-500	9	5000	0.026	9.39×10^{-8}	4.83	3.21×10^{-8}
clique-10-200	16	2000	0.022	1.12×10^{-7}	0.80	3.33×10^{-8}

Finally, we investigate an advantage of the serial-column distribution $\{\mathcal{R}_u : u = 1, 2, \ldots, N\}$ as compared to a stereotypic cyclic-column distribution $\{\mathcal{S}_u : u = 1, 2, \ldots, N\}$, where

$$\begin{array}{rcll} \mathcal{R}_u & = & \{k : (u-1) \times \lfloor n/N \rfloor + 1 \leq k \leq u \times \lfloor n/N \rfloor\} & (u = 1, 2, \ldots, N-1) \\ \mathcal{R}_N & = & \{k : (N-1) \times \lfloor n/N \rfloor + 1 \leq k \leq n\} & \end{array},$$

and

$$\mathcal{S}_u \quad = \quad \{i : 1 \leq i \leq n, i\%N = u\}.$$

We abbreviate the serial-column distribution and the cyclic-column distribution as SERIAL and CYCLIC, respectively. Since the computation cost for each column of $\widetilde{d\boldsymbol{X}}$ is almost the same, there is little difference between the computation cost of SERIAL and that of CYCLIC. Therefore, we concentrate on a difference of the communication cost. The significant difference between SERIAL and CYCLIC comes from the data storage of $[\widetilde{d\boldsymbol{X}}]$; $[\widetilde{d\boldsymbol{X}}]_{*,k}$ and $[\widetilde{d\boldsymbol{X}}]_{*,k+1}$ are continuous on memory space on each processor. Therefore, in the case of SERIAL, we can broadcast $\cup\{[\widetilde{d\boldsymbol{X}}]_{*,k} : k \in \mathcal{R}_u\}$ on the uth processor by only one broadcast operation.

To show the advantage of SERIAL, we estimate the shortest communication cost when we are supposed to adopt CYCLIC. We first examine a data conversion to make $\cup\{[\widetilde{d\boldsymbol{X}}]_{*,k} : k \in \mathcal{S}_u\}$ continuous. However, the communication after the conversion is the same as SERIAL. Therefore, CYCLIC is slower than SERIAL due to an overhead of the conversion. Hence, we have to consider the possibility of transfers on CYCLIC without conversing the data storage. We discuss two types of transfers on CYCLIC here.

1. Broadcast with a smaller length.

We repeat broadcast of $[\widetilde{d\boldsymbol{X}}]_{*,k}$ for $k = 1, 2, \ldots, n$. In this case, the total amount of communication is the same as SERIAL. However, we need n times broadcast of the length ρ in CYCLIC, while N times broadcast of the length $\rho \times n/N$ in SERIAL. Therefore, the bandwidth of SERIAL is higher than that of CYCLIC. As a result, the communication time of SERIAL is shorter than CYCLIC.

2. Accumulation over all processors.

We accumulate the result of $\widetilde{d\boldsymbol{X}}$ on each processor. Let $\widetilde{d\boldsymbol{X}}^{(u)}$ be the evaluated $\widetilde{d\boldsymbol{X}}$ on the uth processor. (Note that $[\widetilde{d\boldsymbol{X}}^{(u)}]_{*,k}$ is assigned the zero vector if $k \notin \mathcal{S}_u$.) Then we compute $\sum_{u=1}^{N} \widetilde{d\boldsymbol{X}}^{(u)}$ and broadcast the result of summention. Then each processor holds the entire matrix of $\widetilde{d\boldsymbol{X}}$.

In this case, we can send by the length $\rho \times n$ by one 'accumulation' operation. Let $t_v(acc)$ and $t_v(broad)$ be the communication time to send one 'double' value by 'accumulation' and 'broadcast' operations, respectively. Then, the communication time for SERIAL, T_{SERIAL}, and that for CYCLIC, T_{CYCLIC} are estimated as follow.

$$\begin{array}{rcl} T_{SERIAL} & = & (\rho \times n/N \times t_v(broad) \times \log_2 N + t_m) \times N, \\ T_{CYCLIC} & = & (\rho \times n) \times (t_v(acc) \times \log_2 N) + t_m. \end{array}$$

Therefore, the difference is

$$T_{CYCLIC} - T_{SERIAL} \quad = \quad (t_v(acc) - t_v(broad)) \times \rho \times n \times \log_2 N - (N-1) \times t_m.$$

From the numerical experiments of cut-10-200, cut-10-500, clique-10-200 on Presto III, we obtained the order of real values, $t_v(acc) = 10^{-5}$, $t_v(broad) = 10^{-8}$ and $t_m = 10^{-5}$. Considering situations $n >> N-1 \geq 2$ and $\rho \geq 1$ which are reasonable for a general SDPs, we can conclude $T_{CYCLIC} \geq T_{SERIAL}$. Consequently, we have verified that the communication cost for $\widetilde{d\boldsymbol{X}}$ is minimized when we broadcast only the elements assigned by the extended sparsity and adopt the serial-columns distribution.

Chapter 5

Conclusions and Future Directions

SDP has made continuous impacts by the strong PD-IPM on the broad range fields such as control theory, combinatorial optimization, financial engineering. Furthermore, the application regions are extended to data-mining and quantum chemistry. However, in particular in quantum chemistry and combinatorial optimization, SDPs become extremely large involving a lot of equality constraints and large-scale variable matrices and beyond the range of the existing software on a single processor. Therefore, we focus on the question *"how we solve larger SDPs in shorter time"* in this thesis.

To answer the question, we relied on parallel computation which have recently supplied massive computation resource. To apply parallel computation, at first we divided the computation of PD-IPM into the following four components and investigated the bottlenecks which should be replaced by their parallel implementation (section 3.2.1).

- ELEMENTS (the evaluation of the Schur complement matrix)
- CHOLESKY (the Cholesky Factorization of the Schur complement matrix)
- PMATRIX (the computation for the $d\boldsymbol{X}$)
- DENSE (the other computation for $n \times n$ matrices)

We verified that most of computation cost to solve general SDPs is occupied by ELEMENTS and CHOLESKY. We replaced these two components by their parallel implementation and developed the parallel software SDPARA.

- ELEMENTS is distributed on multiple processors in accordance with the simple but strong row-wise distribution (section 3.2.2).
- We redistribute the Schur complement matrix from the row-wise distribution to two-dimensional block-cyclic distribution to reduce the computation time of the parallel Cholesky Factorization for CHOLESKY (section 3.2.3).

To analyze the performance of SDPARA, we carried out the following preliminary numerical experiments.

- We examined how the best performance is obtained in the row-wise distribution and learned that the slimest belts attain the best load-balance (section 3.4.1).
- It was verified that the Cholesky Factorization is faster than Conjugate Gradient method when we solve the Schur complement equation on parallel computation. However, it should be mentioned that Conjugate Gradient method obtains higher scalability than the Cholesky Factorization (section 3.4.2).
- The numerical experiments to investigate the effect of the physical network environments were also conducted. The ELEMENTS component is almost independent of the capacity of the network, while the CHOLESKY component is strongly affected. The high speed network is required to obtain enough scalability (section 3.4.3).

Through the full-scale numerical experiments, SDPARA got the following results.

- SDPARA can solve control11 and theta6 from SDPLIB in the very short time which other existing software could not achieve. In particular, the excellent scalability of ELEMENTS on control11 is attained, because the row-wise distribution enables SDPARA to compute ELEMENTS without any communication between multiple processors (section 3.5.1).
- SDPARA also solves SDPs in quantum chemistry at a further rapid speed than ever. Furthermore, the scalability grows up when the size of SDP increases. (section 3.5.3).
- SDPARA is superior to the existing parallel software PDSDP on Control Problems, Theta Function Problems, and SDPs in quantum chemistry from the viewpoints of both computation time and scalability. However, PDSDP is superior on Max Cut Problems (section 3.6).

Especially, it can be said that SDPs in quantum chemistry will be put into practical use by the short computation time attained by SDPARA.

We have also discussed the theoretical validity of the selected approaches in SDPARA as compared to other alternative approaches and verified the superiority of SDPARA (section 3.7).

These numerical results and validity show that SDPARA have proposed a partial approach to answer the question how we solve larger SDPs in shorter time.

Nevertheless, parallel computation implemented in SDPARA does not always perform all SDPs effectively. SDPARA is not an eligible for SDPs involving large-scale variable matrices, since SDPARA retains $n \times n$ dense matrices on each processor and the computation cost occupied by PMATRIX looks conspicuous. In addition, PD-IPM has a drawback that the primal variable matrix $\boldsymbol{X}$ is fully-dense no matter how sparse the input data matrices are. Therefore, even if the number m of equality constraints is large, the advantage of SDPARA to reduce the computation cost of ELEMENTS and CHOLESKY is vanished.

To handle the sparsity of the input data matrices on the primal variable matrix, we incorporated the completion method into parallel computation. We implemented SDPARA-C which is a combination of SDPARA with the completion method.

SDPARA-C has remarkable features as below.

- Instead of fully-dense $\boldsymbol{X}$ and $\boldsymbol{Y}^{-1}$, SDPARA-C retains their inverse sparse Cholesky Factorization $\boldsymbol{M}$ and $\boldsymbol{N}$ ($\boldsymbol{X}^{-1} = \boldsymbol{M}\boldsymbol{M}^T, \boldsymbol{Y} = \boldsymbol{N}\boldsymbol{N}^T$). The retention successfully cuts a lot of memory space (section 4.1).
- For ELEMENTS component, SDPARA-C adopts the hashed row-wise distribution which hashes the rows which disturb the load-balance of the simple row-wise distribution into smaller units. Therefore, SDPARA-C does not lose its scalability on SDPs involving the special structure arisen from graph theory (section 4.2.2).
- SDPARA-C also adopts the column-wise distribution for PMATRIX, which enables us to reduce the computation time of PMATRIX bottlenecks. In addition, the distribution is implemented so that SDPARA-C can compute with the minimum amount of communication regarding PMATRIX (section 4.2.3).

The numerical experiments of SDPARA-C show us that

- SDPARA-C requires less computation cost and memory space to solve sparse SDPs. However, the performance of SDPARA-C is strongly affected by the sparsity of input data matrices. (section 4.3).
- SDPARA-C can obtain higher scalability than SDPARA or PDSDP. Furthermore SDPARA-C can handle extremely large SDPs arisen from combinatorial optimization which are out of the ranges of SDPARA and PDSDP (section 4.4).

The validity of parallel implementation in SDPARA-C regarding the hashed row-wise distribution and the transfer type of primal variable matrix have also presented (section 4.5).

In particular, the results how large SDPs can be carried out by SDPARA-C show that a combination of the completion method with parallel computation brings another approach to answer the question how we larger SDPs in shorter time to us.

Here, we point out some future directions from the viewpoint of parallel computation.

- The current implementation is derived from the simple row-wise distribution, although it attains enough performance. We had better equalize the load-balance for ELEMENTS based on a more sophisticated theoretical analysis. However, the issue will become NP-hard combinatorial problem if we treat it strictly. Therefore we have to seek a compromise between load-balance and computation cost. At the same time, the distribution of memory space and the amount of communication must be taken into consideration.

- SDPARA-C attains higher scalability than SDPARA. Nevertheless, if the sparsity of the input data matrices is not enough, SDPARA-C does not perform them efficiently since the effect of the completion method shrinks. Meanwhile, the performance of SDPARA can attain its performance even when all the input data matrices are dense. In short, SDPARA and SDPARA-C can complement each other. An automatic selection between two software according to the sparsity will bring significant benefits and should be implemented. However, the selection will require accurate estimations of parallel computation time of SDPARA and SDPARA-C. It may not be so easy task.

- Although we focused on only SDP in the thesis, it should be challenged that how to apply parallel computation to a combination of SDP and Second-Order Cone Programming which also can be solved effectively by PD-IPM. A research of parallel computation retaining the cheaper computation cost of Second-Order Cone Programming than that of SDP should be discussed.

Finally, we make mention of some researches on which we have strong theoretical interests as our future directions.

- An unsatisfying point of the current implementation of SDPARA and SDPARA-C is their numerical stability. To solve SDPs whose feasible regions are very narrow, for example Control Problem, we can not avoid a discussion of the numerical stability. In particular, since a condition number of the Schur complement matrix becomes extremely worse in a neighborhood of an optimal solution, we must bound it above by some schemes.

- A promising idea of efficiency we have now is a combination of CG method in the beginning iterations, the Cholesky Factorization in the middle iterations, and the preconditioned symmetric quasi-minimal residual method proposed by Toh [78] in the last iterations. However, the computation cost for the condition number to switch the methods on parallel computation is heavy, other threshold is required to be developed.

- In PD-IPM frameworks, the assumption that both primal and dual SDPs have interior feasible points is essential, since PD-IPM are based on the duality theorem which ensures the coincidence of the primal and dual optimal values. However, all SDPs do not have interior feasible points [72]. To deal with such SDPs, we have to adopt other methods. For example, approaches such as Quantifier Elimination (QE) [2] which do not assume the existence of interior feasible points should be discussed more widely. However, it also should be mentioned that computation cost of QE is so heavier than PD-IPM that QE is now far from practical use.

- It is common that SDPs arisen from data-mining have special structures about the number m of equality constraints and the size n of variable matrices. Since the field of data-mining is expected to grow quickly, it will be meaningful for many researches that we implement specific software to solve such special SDPs very fast. We may be able to apply some knowledge of SDPs from quantum chemistry to this type of large-scale SDPs.

Bibliography

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

[2] H. Anai, Quantifier Elimination for Real Algebraic Constraints in Industry, *Proceedings of the Sixth International Workshop on Computer Algebra in Scientific Computing*, CASC 2003, Institut für Informatik, Technische Universität München, Germany, October, 2003, 3–11.

[3] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Croz, A. Greenbaum, S. Hammarling, A. McKenney and D. Sorensen, *LAPACK Users' Guide Third*, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1999.

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

[5] S. J. Benson and Y. Ye, DSDP home page, [http://www.mcs.anl.gov/~benson/dsdp], 2002.

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

[7] A. Ben-Tal and A. Nemirovskii, *Lectures on Modern Convex Optimization Analysis, Algorithms, and Engineering Applications*, Society for Industrial and Applied Mathematics, Philadelphia, PA, 2001.

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

[9] J. R. S. Blair and B. Peyton, An introduction to chordal graphs and clique trees, in A. George, J. R. Gilbert and J. W. H.Liu (Eds.), *Graph Theory and Sparse Matrix Computation*, Springer-Verlag, New York, 1993, 1–29.

[10] B. Borchers, CSDP, A C Library for Semidefinite Programming, *Optimization Methods and Software*, (1999), **11 & 12**, 613–623.

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

[12] S. Boyd, L. E. Ghaoui, E. Feron and V. Balakrishnan, *Linear matrix inequalities in system and control theory*, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1994.

[13] S. Burer, Semidefinite programming in the space of partial positive semidefinite matrices, manuscript, Department of Management Sciences, University of Iowa, Iowa City, IA 52242-1000, USA, May (2002).

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

[15] 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, USA, 2000.

[16] J. Choi, J. J. Dongarra, L. S. Ostrouchov, A. P. Petitet D. W. Walker and R. C. Whaley, Design and implementation of the ScaLAPACK LU, QR, and Cholesky factorization routines, *Scientific Programming*, (1996), **5(3)**, 173–184.

[17] A. J. Coleman, Structure of Fermion Density Matrices, *Reviews of Modern Physics*, (1963), **35(3)**, 668–689.

[18] R. P. Feynman, R. B. Leighton and M. L. Sands, *The Feynman Lectures on Physics*, Addison-Wesly Publishing Company Inc, Reading, Massachusetts, 1965.

[19] M. J. Frish *et al.*, Gaussian 98, Revision A.9 Gaussian, Inc, Pittsburgh, PA, 1998.

[20] K. Fujisawa, M. Fukuda, M. Kojima and K. Nakata, Numerical Evaluation of SDPA (SemiDefinite Programming Algorithm), in H. Frenk, K. Roos, T. Terlaky and S. Zhang (Eds.), *High Performance Optimization*, Kluwer Academic Press, 2000, 267–301.

[21] K. Fujisawa, Y. Futakata, M. Kojima, K. Nakata and M. Yamashita, SDPA-M (Semidefinite Programming Algorithm in MATLAB) User's Manual, Technical Report B-359, Department of Mathematical and Computing Sciences, Tokyo Institute of Technology. Revised July 2003.

[22] K. Fujisawa, M. Kojima and K. Nakata, Exploiting Sparsity in Primal-Dual Interior-Point Methods for Semidefinite Programming, *Mathematical Programming*, (1997), **79**, 235–253.

[23] K. Fujisawa, M. Kojima, K. Nakata and M. Yamashita, SDPA (Semidefinite Programming Algorithm) User's Manual, Technical Report B-308, Department of Mathematical and Computing Sciences, Tokyo Institute of Technology. Revised July 2002.

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

[25] M. Fukuda, M. Kojima and M. Shida, Lagrangian Dual Interior-Point Methods for Semidefinite Programs, To appear in SIAM Journal on Optimization, March 2001. Revised October 2001.

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

[27] M. X. Goemans, Semidefinite programming in combinatorial optimization, *Mathematical Programming*, (1997), **79**, 143–161.

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

[29] G. H. Golub and C. F. Van Loan, *Matrix Computations*, The John Hopkins University Press, Maryland, 1983.

[30] M. C. Golumbic, *Algorithmic Graph Theory and Perfect Graphs*, Academic Press, New York, 1980.

[31] R. Grone, C. R. Johnson, E. M. Sá and H. Wolkowicz, Positive definite completion of partial hermitian matrices, *Linear Algebra and its Applications*, (1998), **82**, 291–315.

[32] C. Helmberg, SemiDefinite Programming Home Page, [http://www-user.tu-chemnitz.de/~helmberg/semidef.html]

[33] C. Helmberg, K. C. Kiwiel, A Spectral Bundle Method with Bounds (rev.Vers.SEP01), *Mathematical Programming*, (2002), **93**, 173–194.

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

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

[36] N. Karmarkar, A new polynomial-time algorithm for linear programming, *Combinatorica*, (1984) **4**, 375–395.

[37] N. Karmarkar, J. C. Lagarias, L. Slutsman and P. Wang, Power series variants of Karmarkar-type algorithms, *AT&T Technical Journal*, (1989), **68**, 20–36.

[38] L. G. Khachiyan, A polynomial algorithm in linear programming, *Soviet Mathematics Doklady*, (1979), **20**, 191–194.

[39] M. Kocvara and M. Stingl, PENNON - A Generalized Augmented Lagrangian Method for Semidefinite Programming, Research Report 286, Institute of Applied Mathematics, University of Erlangen, 2001

[40] M. Kocvara and M. Stingl, PENNON - A Code for Convex Nonlinear and Semidefinite Programming, Research Report 290, Institute of Applied Mathematics, University of Erlangen, 2002

[41] M. Kojima, S. Mizuno and A. Yoshise (1989), A Primal-Dual Interior Point Algorithm for Linear Programming, in N. Megiddo (Eds.), *Progress in Mathematical Programming: Interior Point and Related Methods*, Springer-Verlag, New York, 1989, 29–47.

[42] M. Kojima, S. Shindoh and S. Hara, Interior-point methods for the monotone semidefinite linear complementarity problems, *SIAM Journal on Optimization*, (1994), **7**, 86–125.

[43] M. Kojima and A. Takeda, Complexity analysis of successis convex relaxation methods for non convex sets, To appear in *Mathematics of Operations Research*, April 1999, Revised July 2000.

[44] M. Kojima and L. Tunçel, Cones of matrices and successive convex relaxations of nonconvex sets, *SIAM Journal on Optimization*, (2000), **10**, 750–778.

[45] M. Kojima and L. Tunçel, Discretization and Localization in Successive Convex Relaxation for Non-convex Quadratic Optimization Problems, *Mathematical Programming*, (2000), **89**, 79–111.

[46] K. Krishnan and J. Mitchell, Semi-infinite linear programming approaches to semidefinite programming (SDP) problems, in P. M. Pardalos and H. Wolkowicz (Eds.), *Novel Approaches to Hard Discrete Optimization, Fields Institute Communications Series*, American Math. Society, 2002.

[47] K. Krishnan and J. Mitchell, Properties of a Cutting Plane Method for Semidefinite Programming, Technical Report, Department of Computational & Applied Mathematics, Rice University, May 2003

[48] G. Lanckriet, N. Cristianini, P. Bartlett, L. El Ghaoui and M. Jordan, Learning the kernel matrix with semi-definite programming, in C. Sammut and A. Hoffmann (Eds.), *Proceedings of the 19th International Conference on Machine Learning*, Morgan Kaufmann, 2002.

[49] J. G. Lewis, B. W. Peyton and A. Pothen, A fast algorithm for reordering sparse matrices for parallel factorization, *SIAM Journal on Scientific and Statistical Computing,* (1989), **10**, 1146 – 1173.

[50] C. J. Lin and R. Saigal, On solving large-scale semidefinite programming problems - a case study of quadratic assignment problem, Dept. of Industrial and Operations Engineering, University Michigan, Ann Arbor, MI 481098, 1998.

[51] C. J. Lin and R. Saigal, An incomplete Cholesky factorization for dense symmetric positive definite matrices, *BIT*, (2000), **40** 536–558.

[52] K. London, J. Dongarra, S. Moore, P. Mucci, K. Seymour, and T. Spencer, PAPI:End-user Tools for Application Performance Analysis, Using Hardware Counters, Presented at International Conference on Parallel and Distributed Computing Systems, August 2001.

[53] H. M. Markowitz, *Mean-variance analysis in portfolio choice and capital markets*, Blackwell Publishers, Oxford, New York, 1987.

[54] J. E. Mayer, Electron Correlation, *Physics Review*, (1955), **100 (2)**, 1579–1586.

[55] S. Mehrotra, On the implementation of a primal-dual interior point method, *SIAM Journal on Optimization*, (1992), **2**, 575–601.

[56] S. Mizuno, M. J. Todd and Y. Ye, On adaptive-step primal-dual interior-point algorithms for linear programming method, *Mathematics of Operations Research*, (1995), **18**, 964–981.

[57] R. D. C. Monteiro, Primal-dual path following algorithms for semidefinite programming, *SIAM Journal on Optimization*, (1995), **7**, 663–678.

[58] Myricom Home Page, [http://www.myri.com/]

[59] 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*, Series B, (2003), **95**, 303–327.

[60] 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*, Tokeisuuri, (1998) **46(2)**, 297–316.

[61] M. Nakata, M. Ehara and H. Nakatsuji, Density matrix variational theory: Application to the potential energy surfaces and strongly correlated systems, *Journal of Chemical Physics*, (2002), **116**, 5432–5439.

[62] 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, *Journal of Chemical Physics*, (2001), **114**, 8282–8292.

[63] Yu. E. Nesterov and A. Nemirovskii, *Interior Point Polynomial Methods in Convex Programming: Theory and Applications*, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1994.

[64] Yu. E. Nesterov and M. J. Todd, Primal-Dual Interior-Point Methods for Self-Scaled Cones, *SIAM Journal on Optimization*, (1994), **8**, 324–364.

[65] PC Cluster Consortium Home Page, [http://www.pccluster.org/]

[66] A. Robert and van de Geijn, *Using PLAPACK: Scientific and Engineering Computation series*, The MIT Press, Cambridge, Massachusetts, 1997.

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

[68] A. Szabo and N. S. Ostlund, *Modern quantum chemistry : introduction to advanced electronic structure theory*, Dover Publications Inc, New York, 1996.

[69] A. Takeda, Y. Dai, M. Fukuda, and M. Kojima, Towards Implementations of Successive Convex Relaxation Methods for Nonconvex Quadratic Optimization Problems, in P. M. Pardalos (Eds.), *Approximation and Complexity in Numerical Optimization: Continuous and Discrete Problems*, Kluwer Academic Publisher, 2000, 489–510.

[70] K. Tanabe, Centered Newton Method for Mathematical Programming, in M. Iri and K. Yajima (Eds.), *System Modeling and Optimization*, Springer, New York, 1988, 197–206.

[71] TAO : Toolkit for Advanced Optimization Home-page, [http://www-fp.mcs.anl.gov/tao/]

[72] M. J. Todd, Semidefinite optimization, *Acta Numerica*, (2001), **10**, 515–560.

[73] M. J. Todd, K. C. Toh and R. H. Tütüncü, On the Nesterov-Todd direction in semidefinite programming, Technical Report, School of Operations Research and Industrial Engineering, Cornell University, Ithaca, NY 14853-3801, USA, 1996.

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

[75] M. J. Todd and Y. Ye, A centered projective algorithm for linear programming, *Mathematics of Operations Research*, (1990), **15**, 508–529.

[76] K. C. Toh, Some new search directions for primal-dual interior point methods in semidefinite programming, *SIAM Journal on Optimization*, (2000), **11**, 223–242.

[77] K. C. Toh, A note on the calculation of step-lengths in interior-point methods for semidefinite programming, *Computational Optimization and Applications*, (2002), **21**, 301–310.

[78] K. C. Toh, Solving large scale semidefinite programs via an iterative solver on the augmented systems, To appear in *SIAM Journal on Optimization.*

[79] K. C. Toh and M. Kojima, Solving some large scale semidefinite programs via the conjugate residual method, *SIAM Journal on Optimization* (2002), **21**, 669–691.

[80] TOP 500 Supercomputer sites, [http://www.top500.org/]

[81] A. M. Turing, On computable numbers, with an application to the Entscheidungs problem, *Proc.London Math.Soc., Ser.2*, (1936), **42** , 230–265, (1937), **43** , 544–546.

[82] L. Vandenberghe and S. Boyd, Positive-Definite Programming, in J. R. Birge and K. G. Murty (Eds.), *Mathematical Programming: State of the Art 1994*, U. of Michigan, 1994.

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

[84] S. J. Wright, *Primal-Dual Interior-Point Methods*, SIAM, Philadelphia, 1997.

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

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

[87] Y. Ye, *Interior Point Algorithms : Theory and Analysis,* Wiley-Interscience Series in Discrete Mathematics and Optimization. John Wiley & Sons, New York, 1997.

[88] M. Zhang, On extending some primal-dual interior-point algorithms from linear programming to semidefinite programming, *SIAM Journal on Optimization*, (1998), **8**, 365–386.

[89] S. L. Zhang, K. Nakata and M. Kojima, Incomplete orthogonalization preconditioners for solving large and dense linear systems which arise from Semidefinite Programming, *Applied Numerical Mathematics*, (2002), **41**, 235–245.

[90] Q. Zhao, S. E. Karish, F. Rendl and H. Wolkowicz, Semidefinite programming relaxations for the quadratic assignment problem, *Journal of Combinatorial Optimization*, (1998), **2**, 71–109.

[91] Z. Zhao, B. J. Braams, M. Fukuda, M. L. Overton, and J. K. Percus, The reduced density matrix method for electronic structure calculations and the role of three-index representability, *Journal of Chemical Physics*, (2004), **120**, 2095–2104.