Preconditioning Dense Linear Systems from Large-Scale Semidefinite Programming Problems

Chih-Jen Lin[1]

Abstract

In this paper we show semidefinite programming (SDP) could be a resource for large dense systems. The preconditioned conjugate gradient method is used to solve fully dense linear systems during each iteration of an optimization algorithm. An incomplete Cholesky factorization is proposed, and some numerical results are obtained. These results generate better lower bounds for the quadratic assignment problem, an application of SDP.

1 Introduction

Semidefinite programming (SDP) is a new topic in optimization and has many potential applications. Considerable attention is being paid to generating algorithms based on the interior point strategy, and many algorithms for linear programming have been extended to solve this kind of problem. An excellent survey is in a paper by Vandenberghe and Boyd [16]. To date, computational issues of solving large-scale SDP problems, using interior point methods, have not been investigated extensively. In particular, the dense linear systems in each optimization iteration are computational bottlenecks; this is the main difference with interior point methods for linear programming.

In section 2, we discuss how the dense systems arise. We present several special properties of our applications that allow more effective use of iterative methods. Although recently there have been several successful studies on solving large dense systems, (e.g., the survey by Edelman [3, 4]), not many papers have addressed the issue of preconditioning. In section 3, we propose a version of incomplete Cholesky factorization (ICF) for dense systems. Section 4 presents our computational results. Experiments show that the number of conjugate gradient iterations is reduced about 30% on the average, when compared with the diagonal preconditioner. Better lower bounds for the quadratic assignment problem, an SDP application, are reported. Finally, in section 5, some concluding remarks are presented.

2 Dense Systems in Semidefinite Programming

A standard form of the semidefinite programming problem is as follows:

$$\begin{array}{lrcll} \text{minimize} & C \bullet X & & & \\ & A_i \bullet X & = & b_i & \text{for every } i = 1, \cdots, m, \\ & X \succeq 0 & & & \end{array} \qquad (2.1)$$

[1]Department of Industrial and Operations Engineering, University of Michigan, Ann Arbor, MI 48109.
Work supported in part by the National Science Foundation grant CCR-9321550. Part of this work was done while the author visited the Mathematics and Computer Science Division, Argonne National Laboratory, supported by U.S. Department of Energy, under Contract W-31-109-Eng-38.

where A_i for $i = 1, \cdots, m$ and C are $n \times n$ symmetric matrices, $A \bullet B = \mathrm{trace}(A^T B)$, $X \succeq 0$ means that X is a symmetric and positive semidefinite matrix, and $X \succ 0$ means that it is a symmetric and positive definite matrix. The dual of problem (2.1) is as follows.

$$\begin{array}{lcccc} \text{maximize} & \sum_{i=1}^m b_i y_i & & & \\ & \sum_{i=1}^m A_i y_i & + & S & = & C \\ & & & S \succeq 0 & & \end{array} \tag{2.2}$$

Currently, the only effective way to solve problem (2.1) or (2.2) is through interior point methods. One of commonly used interior point methods is an infeasible start primal dual algorithm. It starts from an infeasible solution, and, in each iteration, a predictor system (2.3) and a corrector system (2.4) are solved:

$$\begin{array}{ccccccc} A_i \bullet \Delta X_k & & & & = & -(A_i \bullet X_k - b_i) & i = 1, \cdots, m \\ & \sum_{i=1}^m A_i \Delta y_i^k & + & \Delta S_k & = & -(\sum_{i=1}^m A_i y_i^k + S_k - C) & \\ \Delta X_k S_k & & + & X_k \Delta S_k & = & t_k I - X_k S_k & \end{array} \tag{2.3}$$

$$\begin{array}{ccccccc} A_i \bullet \Delta \bar{X}_k & & & & = & 0 & \text{for all } i = 1, \cdots, m \\ & \sum_{i=1}^m A_i \Delta \bar{y}_i^k & + & \Delta \bar{S}_k & = & 0 & \\ \Delta \bar{X}_k S_k & & + & X_k \Delta \bar{S}_k & = & \bar{t}_k I - \bar{X}_k \bar{S}_k & \end{array} \tag{2.4}$$

The next iteration is updated by using the information of the current iteration, the predictor direction, and the corrector direction.

Solving both systems (2.3) and (2.4) is the main computational burden of interior point methods. We use the following three derived equations to solve them:

$$(\bar{A}(S_k^{-1} \otimes X_k)\bar{A}^T)\Delta y_k = h_1 - \sum_{i=1}^m A_i \bullet ((h_3 - X_k h_2) S_k^{-1}) \tag{2.5}$$

$$\Delta S_k = -\sum_{i=1}^m \Delta y_i^k A_i + h_2 \tag{2.6}$$

$$\Delta X_k = (-h_3 - X_k \Delta S_k) S_k^{-1}, \tag{2.7}$$

where h_1, h_2 and h_3 are the right-hand side of (2.3) and (2.4), $\otimes$ is the Kronecker product, $\bar{A} = \begin{bmatrix} \mathrm{vec}(A_1)^T \\ \vdots \\ \mathrm{vec}(A_m)^T \end{bmatrix}$, $(\bar{A}(S_k^{-1} \otimes X_k)\bar{A}^T)$ is an m by m matrix, and

$$(\bar{A}(S_k^{-1} \otimes X_k)\bar{A}^T)_{i,j} = A_i \bullet (X_k A_j S_k^{-1}), \quad i, j = 1, \ldots, m. \tag{2.8}$$

Note that $\mathrm{vec}(A_i)$ denotes the vector formed from the columns of A_i.

From (2.8), $(\bar{A}(S_k^{-1} \otimes X_k)\bar{A}^T)$ is a fully dense matrix, as are X_k and S_k. While calculation of (2.6) and (2.7) is computationally cheap, obtaining $(\bar{A}(S_k^{-1} \otimes X_k)\bar{A}^T)$ and solving (2.5) are more difficult.

If A_j are dense, $X_k A_j S_k^{-1}$ takes $O(n^3)$ operations and hence $O(mn^3 + m^2 n^2)$ operations are required to formulate the matrix, which may be more expensive than $O(m^3)$ required for solving (2.5) by Cholesky factorizations. This situation is very different from the case of interior point methods for linear programming and makes (SDP) problems difficult to solve. However, for several applications, mainly in combinatorial optimization, most A_i are extremely sparse. That is, only a constant number of nonzeroes are in each A_i. By using this property, formulating $(\bar{A}(S_k^{-1} \otimes X_k)\bar{A}^T)$ requires only $O(m^2)$ operations. It is easy to see this result because for each i, j, $A_i \bullet (X_k A_j S_k^{-1})$ requires only a constant number of multiplications and additions. Therefore, the computational bottleneck is the effective solution of the fully dense system (2.5); this is where iterative methods will be used.

For difficult combinatorial problems, good lower bounds improve the effectiveness of procedures used to solve the original problems. Recently, SDP has been a popular method to obtain lower bounds. The basic idea is to reformulate combinatorial problems to be similar to SDP standard forms (2.1) or (2.2). An SDP relaxation is then solved, thereby obtaining a lower bound of the original problem. This procedure allows us to solve the dense system (2.5) with less accuracy. The higher level of accuracy is not necessary for the following reasons: Because the algorithm starts at an infeasible point, both predictor and corrector systems help the iterations approach feasibility. That is,

$$A_i \bullet X_{k+1} - b = \alpha(A_i \bullet X_k - b_i), i = 1, \ldots, m, \text{ and} \tag{2.9}$$

$$A^T y^{k+1} + S_{k+1} - C = \alpha(A^T y^k + S_k - C), \tag{2.10}$$

where $0 < \alpha < 1$.

However, even if the solution of (2.5) is not very accurate, this does not affect the dual feasibility because ΔS_k and $\Delta \bar{S}_k$ are directly calculated as $-\sum_{i=1}^m A_i \Delta y_i^k - (\sum_{i=1}^m A_i y_i^k + S_k - C)$ and $-\sum_{i=1}^m A_i \Delta \bar{y}_i^k$, respectively. Therefore, (2.10) is always numerically correct. On the other hand, primal feasibility will be affected. However, since we are trying to get lower bounds for combinatorial applications, it is sufficient that the algorithm returns a dual solution that is almost dual feasible. The dual objective value of that iteration is a lower bound of SDP relaxation and hence a lower bound of the original application. This observation allows a further reduction in the effort of using iterative methods for solving (2.5).

As pointed out by Edelman in his survey [3] and some case studies [5, 13], a modern approach for solving dense linear systems is to use a preconditioned iterative method that accesses the matrix only by matrix-vector multiplication and look for fast approximate methods for computing the matrix-vector product. Because of the huge amount of computer memory required to store the dense matrices, for very large problems this approach is the only plausible solution. In our application, since most A_i are extremely sparse, the matrix-vector product $(\bar{A}(S_k^{-1} \otimes X_k)\bar{A}^T)p$ has the same computational order as $(\bar{A}((S_k^{-1} \otimes I)((I \otimes X_k)(\bar{A}^T p))))$. Therefore, the matrix-vector product is not a problem. Since the matrix

$(\bar{A}(S_k^{-1} \otimes X_k)\bar{A}^T)$ is positive definite, the conjugate gradient method is a natural choice for solving the system (2.5).

3 Incomplete Cholesky Factorization for Dense Systems

A good preconditioner can dramatically reduce the number of conjugate gradient iterations as well as total computational time. In general, a preconditioner is a matrix M such that $M^{-1}(\bar{A}(S_k^{-1} \otimes X_k)\bar{A}^T)$ is close to the identity matrix; usually M is factorized in an LL^T form. In each conjugate gradient iteration, two back substitutions for solving $(LL^T)z = r$ are executed. Since $(\bar{A}(S_k^{-1} \otimes X_k)\bar{A}^T)$ is positive definite, an easy preconditioner is the diagonal elements of the original matrix. The diagonal preconditioner was first used on SDP problems by Zhao, Karish, Rendl, and Wolkowicz [17]. Note that the computational complexity of finding the diagonal preconditioner is the same as doing only one matrix-vector multiplication $(\bar{A}((S_k^{-1} \otimes I)((I \otimes X_k)(\bar{A}^T p))))$. However, from our experience, in final stage of optimization procedure, $(\bar{A}(S_k^{-1} \otimes X_k)\bar{A}^T)$ is more ill-conditioned and many more CG iterations are required. Therefore, it is advantageous to use more complicated preconditioners. In addition, more iterations could be saved, since the same preconditioner is used for predictor and corrector systems.

However, most existing preconditioners are designed for sparse matrices or some special applications. Furthermore, since we do not store the dense matrices, some traditional preconditioners cannot be used. For example, if the SSOR preconditioner is used, L is a fully dense lower triangular matrix, and we could not afford to store it. Hence, in each CG iteration, in order to calculate $L^{-1}r$, columns of L (which depends on the same columns of $(\bar{A}(S_k^{-1} \otimes X_k)\bar{A}^T)$) will be generated and back substituted sequentially. This approach takes more time than $(\bar{A}(S_k^{-1} \otimes X_k)\bar{A}^T)p$, and $L^{-T}(L^{-1}r)$ has to be done by a similar procedure as well. Thus, L should be sparse and be generated without requiring the whole matrix $(\bar{A}(S_k^{-1} \otimes X_k)\bar{A}^T)$.

After testing several preconditioners, we found it is possible to modify a version of the Lin and Moré incomplete Cholesky factorization (ICF) [8]. Their basic idea is from Saad [14, 15] and Jones and Plassmann [6]; the procedure is in Algorithm 3.1. Note that in this algorithm, the (i, j) component of $(\bar{A}(S_k^{-1} \otimes X_k)\bar{A}^T)$ is represented as `a(i,j)`.

The incomplete factor L is calculated column by column. After the jth column is obtained, some of the largest (in magnitude) elements are stored back to L (see the last line of Algorithm 3.1). For sparse problems, either the `p` or `col_len + p` largest nonzeroes are selected, where `p` is a small positive number and `col_len` is the number of nonzeroes of jth column of the original matrix. The first ICF proposed by Meijerink and van der Vorst [11] kept the same sparsity pattern, so only elements at nonzero positions of the original matrix are kept. For the fully dense matrix, `col_len` is equal to m, and their approach could not be applied. In order to keep L sparse, we select the $5m^{\frac{1}{3}}$ largest elements. Hence, the number of nonzero elements of L is $O(m^{\frac{4}{3}})$.

```
for j = 1:n
    a(j,j) = sqrt(a(j,j))
    col_len = size(i≥j:a(i,j) ≠ 0)
    for k = 1:j-1 & a(j,k) ≠ 0
        for i = j+1:n & a(i,k) ≠ 0
            a(i,j) = a(i,j) - a(i,k)*a(j,k)
        end
    end
    for i = j+1:n & a(i,j) ≠ 0
        a(i,j) = a(i,j)/a(j,j)
        a(i,i) = a(i,i) - a(i,j)^2
    end
    Find largest col_len + p elements in a(:,j) and store.
end
```

Algorithm 3.1: Incomplete Cholesky factorization (ICF)

Another popular ICF method is the drop tolerance approach, (e.g., Munksgaard [12]). In this strategy nonzeroes are included in the incomplete factor when they are larger than some threshold parameter. Therefore, the memory requirements are unpredictable, a situation that is not appropriate for dense matrices.

Another difficulty with incomplete Cholesky factorization is that it may fail even if the original matrix is positive definite [7]. This problem is noticed during ICF when a negative diagonal element is encountered. One solution is to replace negative diagonal elements with positive numbers. Another approach is the shifted incomplete factorization of Manteuffel [10]. It adds αI to the original matrix, where I is an identity matrix. Eventually, if α is big enough, the matrix will be diagonal-dominant and ICF will succeed. In [8], these two approaches are discussed, and the shifted incomplete factorization appears more stable. Algorithm 3.2 thus is the actual procedure for calculating ICF of a matrix $(\bar{A}(S_k^{-1} \otimes X_k)\bar{A}^T)$ by using the shifted incomplete factorization.

In this algorithm, D is a scaling matrix used to further improve the condition of $(\bar{A}(S_k^{-1} \otimes X_k)\bar{A}^T)$. The justification of using scaling matrices in Algorithm 3.2 for unconstrained optimization is in Bouaricha, Moré, and Wu [1], where $D_{i,i} = \|(\bar{A}(S_k^{-1} \otimes X_k)\bar{A}^T)$'s ith column$\|_2$ is used (l_2 norm scaling). This l_2 norm scaling is not suitable in our case because $(\bar{A}(S_k^{-1} \otimes X_k)\bar{A}^T)$ is not stored and each column has to be calculated at least twice: once for the scaling matrix D and once during ICF for $\hat{B}_k$. Therefore, diagonal scaling, namely, $D_{i,i} = (\bar{A}(S_k^{-1} \otimes X_k)\bar{A}^T)_{i,i}$, is used.

Another issue of Algorithm 3.2 is the numbers α_k and μ. The parameter μ is selected to be 1. In [8], relations of scaling matrices D and the number of α_k are discussed.

4 Computational Experiments

The computational experiments were done on a Sun UltraSPARC1-140 workstation with 128 MB RAM. Computationally-intensive parts such as diagonal preconditioning, the incom-

Choose $\mu > 0$.
Compute $\widehat{B} = D^{-1/2}(\bar{A}(S_k^{-1} \otimes X_k)\bar{A}^T)D^{-1/2}$
Set $\alpha_0 = 0$.
For $k = 0, 1, \dots,$
    Use Algorithm 3.1 on $\widehat{B}_k = \widehat{B} + \alpha_k I$; if successful exit.
    Set $\alpha_{k+1} = \max(2\alpha_k, \mu)$

Algorithm 3.2: Incomplete Cholesky factorization for general matrices

plete Cholesky factorization, and the matrix-vector multiplication $\bar{A}(S_k^{-1} \otimes X_k)\bar{A}^T p$ were written in FORTRAN and linked to MATLAB (version 5.0) drivers through C subroutines and **mex** scripts.

Our test problems are SDP relaxations of some quadratic assignment problems (QAP) from the library QAPLIB maintained by Burkard, Karisch, and Rendl [2]. In a QAP, $\mathcal{P}$ plants are assigned to $\mathcal{P}$ locations so as to minimize the flow-distance-cost. It is an NP-hard problem, and some instances with $\mathcal{P} \leq 20$ are not solved yet. Further details of test problems can be found in Lin and Saigal [9]. In our SDP formulation, $m = \mathcal{P}^3 + 2$ and $n = \mathcal{P}^2 + 1$. Therefore, both X_k and S_k take $O(n^2) = O(\mathcal{P}^4)$ computer memory. We stored $O(m^{\frac{1}{3}})$ nonzero elements in each column of the preconditioner in order to keep the same order of storage. We start the conjugate gradient method with the zero vector and stop the iteration when the relative residue

$$\frac{\|(\bar{A}(S_k^{-1} \otimes X_k)\bar{A}^T)\Delta y_k - h_1 + \sum_{i=1}^m A_i \bullet ((h_3 - X_k h_2)S_k^{-1})\|_2}{\| - h_1 + \sum_{i=1}^m A_i \bullet ((h_3 - X_k h_2)S_k^{-1})\|_2} \leq \epsilon, \tag{4.1}$$

where $\epsilon = 10^{-3}$ for the predictor step and $\epsilon = 5 \times 10^{-3}$ for the corrector step. The main reason for using a larger tolerance for the corrector step is that h_1 and h_2 are both zero and hence the denominator of (4.1) is smaller.

Before comparing the overall performance, first we compare numbers of CG iterations by using diagonal preconditioner and ICF on the same linear systems. In Table 4.1, the diagonal preconditioner is used to solve problems NUG12 and TAI12a (both $\mathcal{P} = 12$); for the same predictor and corrector systems, ICF is also used. In early iterations, because matrices are well conditioned and close to diagonal-dominant, the diagonal preconditioner performs better than ICF. In later iterations, matrices are more ill-conditioned, and ICF takes many fewer CG iterations. Hence, the total number of CG iterations is also less. From this experience, we realized it would be better to start ICF in the middle of the optimization algorithm. The rule implemented is that after the number of of CG iterations in one interior point iteration (either predictor or corrector) reaches more then $0.1\mathcal{P}^2$, incomplete Cholesky factorization is used.

Table 4.2 compares the diagonal preconditioner and incomplete Cholesky factorization for solving NUG12. The first column shows the outer iterations (interior point iterations). For the diagonal preconditioner, the number of CG iterations and the time of matrix-vector

Table 4.1: Comparisons of running the same linear systems

	NUG12				TAI12a			
	Predictor		Corrector		Predictor		Corrector	
Iter.	Diag.	ICF	Diag.	ICF	Diag.	ICF	Diag.	ICF
1	3	4	3	3	3	4	2	2
2	3	3	3	3	3	3	3	3
3	4	4	4	3	4	3	3	3
4	4	5	5	5	4	3	4	3
5	24	57	34	55	4	4	5	5
6	14	29	73	108	25	50	35	45
7	34	35	41	37	16	27	79	96
8	21	22	25	13	15	22	31	14
9	38	27	29	17	33	47	29	17
10	74	37	27	16	167	21	21	14
11	115	67	51	32	43	40	60	31
12	187	115	89	41	77	45	50	54
13					102	58	78	41
Total	521	405	384	333	496	327	400	328

Table 4.2: NUG12

	Diag. Preconditioner		ICF			
Iter.	#CG	$(\bar{A}(S_k^{-1} \otimes X_k)A^T)p$	#CG	ICF	$(LL^T)^{-1}r$	$(\bar{A}(S_k^{-1} \otimes X_k)\bar{A}^T)p$
1-5	87		87			
6	87	15.28	52	6.96	7.09	10.94
7	75	13.17	57	6.21	7.79	11.50
8	46	9.31	47	7.78	6.32	9.48
9	67	12.64	47	6.19	6.67	11.39
10	101	18.32	53	6.34	7.50	10.79
11	166	35.10	98	5.84	14.42	21.92
12	276	63.03	185	9.69	27.14	38.32
6-12	818	166.85	539	49.01	76.93	114.34

multiplication are reported. For ICF, the time of preconditioning (the "ICF" column) and of back-substitution in CG is also reported. For this problem, from the first to the fifth iteration, only the diagonal preconditioner is used. After that, in each outer iteration, the numbers of CG iterations are generally lower for ICF. However, the last row of the table shows the diagonal preconditioner still takes less total computational time. For small problems, time spent on calculating ICF is expensive, and hence the benefit of the decrease in CG iterations does not show up.

Table 4.3 presents the same comparison for solving problem TAI25a ($\mathcal{P} = 25$). It can be clearly seen that by using ICF, the total computational time for the preconditioning and CG iterations is less. This result shows us that nontrivial preconditioners such as ICF are useful for large-scale dense linear systems.

We observe consistent results on the decrease in CG iterations when using ICF. Tables

Table 4.3: TAI25a

Iter.	Diag. Preconditioner # CG	$(\bar{A}(S_k^{-1} \otimes X_k)\bar{A}^T)p$	ICF	ICF	$(LL^T)^{-1}r$	$(\bar{A}(S_k^{-1} \otimes X_k)\bar{A}^T)p$
1-7	276		276			
8	77	2037.45	56	574.63	191.99	1510.86
9	156	4102.70	88	526.18	305.38	2330.85
10	114	3055.39	96	554.80	348.02	2562.39
11	323	8564.22	150	550.76	547.73	4024.23
12	96	2561.04	88	515.65	340.77	2371.18
13	106	2794.45	69	514.50	261.67	1825.90
8-13	872	23115.25	547	3236.5	1995.6	14625.4

Table 4.4: NUG problems

$\mathcal{P}$	Diag.	ICF	Diag.	ICF	Ratio
12	905	626	818	539	66%
14	739	648	652	561	86%
16	935	863	848	776	92%
18	948	712	861	625	73%
20	1389	773	1282	666	52%
22	4093	2256	3767	1930	51%
25	940	895	702	657	94%

4.4 and 4.5 show the comparison of CG iterations that use two preconditioners. The second and third columns are the total number of CG iterations. In the ICF approach, the diagonal preconditioner is used during the early iterations. Without considering those CG iterations, the comparison is shown in the forth and fifth columns. Their ratios are shown in the last column. Except for NUG20 and NUG22, in which the interior point algorithm using ICF takes one and three fewer outer iterations, respectively, all other problem instances take the same number of outer iterations for both preconditioners. For ICF, the total number of CG iterations is always less and in some cases is dramatically reduced. Finally, using this new preconditioner, we are able to obtain the best lower bounds for some problems from QAPLIB. Results are shown in Table 4.6.

Note that when $\mathcal{P} = 30$, we are solving 27002 by 27002 fully dense systems, which are

Table 4.5: TAI problems (series A)

$\mathcal{P}$	Diag.	ICF	Diag.	ICF	Ratio
12	896	610	801	515	64%
15	717	593	612	488	80%
17	845	596	744	495	67%
20	778	642	459	323	70%
25	1148	823	872	547	63%

Table 4.6: New best lower bounds

Problem	Best Lower Bounds	New Results
TAI25a	1006749	1010025
KRA30a	76003	78151

already huge systems for general workstations.

5 Conclusions and Future Research

In this paper, we present an example of using nontrivial preconditioners to solve dense systems that arise in an optimization algorithm. Our numerical results show that when problems are huge, the proposed ICF decreases the total number of CG iterations as well as the computational time. For one SDP problem, a sequence of dense systems has to be solved, which is very time consuming. In the future, a further analysis of the proposed ICF to improve its performance will be conducted.

Acknowledgments

I thank Dr. Jorge Moré and Professor Romesh Saigal for their kind help and support. I also thank Stephanie Lindemann and Gail Pieper for their careful reading of the manuscript; their comments improved the presentation.

References

[1] A. Bouaricha, J. J. Moré, and Z. Wu. Newton's method for large-scale optimization. Preprint MCS-P635-0197, Argonne National Laboratory, Argonne, Illinois, 1997.

[2] R. Burkard, S. Karisch, and F. Rendl. QAPLIB - a quadratic assignment problem library. Technical report, Department of Mathematics, Graz University of Technology, Graz, Austria, 1996.

[3] A. Edelman. Large dense numerical linear algebra in 1993: The parallel computing influence. *International J. Supercomputer Appl.*, 7:113–128, 1993.

[4] A. Edelman. Large numerical linear algebra in 1994: The continuing influence of parallel computing. In *Proceedings of the 1994 Scalable High Performance Computing Conference*, pages 781–787. IEEE Computer Society Press, 1994.

[5] A. Greenbaum, L. Greengard, and G. B. McFadden. Laplace's equation and the Dirichlet-Neumann map in multiply connected domains. *J. Comp. Phys.*, 105(2):267–278, 1993.

[6] M. T. Jones and P. E. Plassmann. An improved incomplete Cholesky factorization. *ACM Trans. Math. Software*, 21:5–17, 1995.

[7] D. S. Kershaw. The incomplete Cholesky-conjugate gradient method for the iterative solution of system of linear equations. *J. Comp. Phys.*, 26:43–65, 1978.

[8] C.-J. Lin and J. J. Moré. Incomplete Cholesky factorizations with limited memory. Preprint MCS-P682-0897, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Illinois, 1997.

[9] C.-J. Lin and R. Saigal. On solving large scale semidefinite programming problems– a case study of quadratic assignment problem. In preparation, Department of Industrial and Operations Engineering, University of Michigan, Ann Arbor, MI 48109-2117, 1997.

[10] T. A. Manteuffel. An incomplete factorization technique for positive definite linear systems. *Math. Comp.*, 34:307–327, 1980.

[11] J. A. Meijerink and H. A. Van der Vorst. An iterative solution method for linear equations systems of which the coefficient matrix is a symmetric M-matrix. *Math. Comp.*, 31:148–162, 1977.

[12] N. Munksgaard. Solving sparse symmetric sets of linear equations by preconditioned conjugate gradients. *ACM Trans. Math. Software*, 6:206–219, 1980.

[13] J. Rahla. Solution of dense systems of linear equations in the discrete-dipole approximation. *SIAM J. Sci. Comput.*, 17:78–89, 1994.

[14] Y. Saad. SPARSKIT : a basic tool kit for sparse matrix computations. Technical report, Computer Science Department, University of Minnesota, Minneapolis, MN 55455, 1994.

[15] Y. Saad. *Iterative Methods for Sparse Linear Systems.* PWS Publishing Company, Boston, 1996.

[16] L. Vandenberghe and S. Boyd. Semidefinite programming. *SIAM Review*, 38:49–95, 1996.

[17] Q. Zhao, S. E. Karish, F. Rendl, and H. Wolkowicz. Semidefinite programming relaxations for the quadratic assignment problem. Technical report, University of Waterloo, Department of Combinatorics and Optimization, Waterloo, Ontario, N2L 3G1 Canada, 1996.