Mathematical Programming manuscript No.
(will be inserted by the editor)

Brian Borchers · **Joseph Young**

How Far Can We Go With Primal–Dual Interior Point Methods for SDP?

Received: date / Accepted: date

Abstract Primal–dual interior point methods and the HKM method in particular have been implemented in a number of software packages for semidefinite pro-

Brian Borchers

Department of Mathematics

New Mexico Tech

801 Leroy Place

Socorro, NM 87801

Tel.: 505-835-5813

Fax: 505-835-5366

E-mail: borchers@nmt.edu

Joseph Young

Department of Combinatorics & Optimization

Faculty of Mathematics

University of Waterloo

Waterloo, Ontario, Canada

N2L 3G1

gramming. These methods have performed well in practice on small to medium sized SDP's. However, primal–dual codes have had some trouble in solving larger problems because of the method's storage requirements. In this paper we analyze the storage requirements of the HKM method and describe a 64-bit parallel implementation of the method that has been used to solve some large scale problems that have not previously been solved by primal–dual methods.

Keywords Semidefinite Programming · Interior Point Methods

1 Introduction

A variety of methods for solving semidefinite programming problems have been implemented, including primal-dual interior point methods [5, 16, 15, 17–19], dual interior point methods [3], and augmented Lagrangian methods [7, 8, 13].

Of these widely used software packages, CSDP, SeDuMi, SDPA, and SDPT3 all implement primal–dual interior point methods. CSDP uses the HKM direction with a predictor–corrector scheme in an infeasible interior point algorithm[5]. SeDuMi uses the NT search direction with a predictor–corrector scheme and uses the self dual embedding technique[16]. Version 6.0 of SDPA uses the HKM direction within an infeasible interior point algorithm. SDPT3 uses either the HKM or NT direction with a predictor–corrector scheme in an infeasible interior point algorithm[18].

The main differences between the codes are in the search directions used and in whether an infeasible interior point method or the self dual embedding is used. Although these choices can have a significant effect on the speed and accuracy of the solutions obtained, they have little effect on the storage requirements of

the algorithms. Since storage limitations are often more important than CPU time limitations in solving large SDP's by primal–dual interior point methods, we will focus primarily on storage issues. Although the discussion in this paper is based on the implementation of the HKM method in CSDP, the results on the asymptotic storage requirements are applicable to all of the codes listed above.

The algorithms used by all of the primal–dual codes require the creation and Cholesky factorization of a large, dense, Schur complement matrix. This matrix is of size m by m where m is the number of linear equality constraints. The primal–dual codes have been developed and used mostly on desktop PC's, which until recently have been limited to 32 bit addressing. A computer with 32 bit addressing can address only 4 gigabytes of RAM. Since a 22,000 by 22,000 matrix of double precision floating point numbers requires 3.9 gigabytes of RAM, it has not been possible to use these codes to solve larger problems.

There are two general approaches to overcoming this limitation. The first is to use a computer with 64 bit addressing and more than 4 gigabytes of RAM. To our knowledge, this paper is the first to present computational results for a 64 bit implementation of a primal–dual method for SDP.

Another approach to dealing with the storage limitation is to distribute the Schur complement matrix over several computers within a cluster. This approach has been used in a parallel version of SDPA[20]. It has also been used in the dual interior point code PDSDP[2]. One problem with this approach is that other data structures used by the algorithm may also become too large to handle with 32 bit addressing. For example, neither SDPARA nor PDSDP can solve some of the

larger problems that have been solved by the version of CSDP described in this paper.

2 Analysis

In this paper we consider semidefinite programming problems of the form

$$\begin{aligned} \max\ & \mathrm{tr}\,(CX) \\ & A(X) = a \\ & X \succeq 0 \end{aligned} \tag{1}$$

where

$$A(X) = \begin{bmatrix} \mathrm{tr}\,(A_1X) \\ \mathrm{tr}\,(A_2X) \\ \cdots \\ \mathrm{tr}\,(A_mX) \end{bmatrix}. \tag{2}$$

Here $X \succeq 0$ means that X is positive semidefinite. All of the matrices A_i, X, and C are assumed to be of size n by n and symmetric. In practice, the X and Z matrices often have block diagonal structure with diagonal blocks of size n_1, n_2, ..., n_k.

The dual of this SDP is

$$\begin{aligned} \min\ & a^Ty \\ & A^T(y) - C = Z \\ & Z \succeq 0 \end{aligned} \tag{3}$$

where

$$A^T(y) = \sum_{i=1}^{m} y_iA_i. \tag{4}$$

The available software packages for semidefinite programming all solve slight variations of this primal–dual pair. For example, the primal–dual pair used in SDPA interchanges the primal and dual problems[19].

In analyzing the computational complexity of primal–dual methods, we will focus on the time per iteration of the algorithms. In practice, the number of iterations required grows very slowly with the size of the problem, and variations in problem structure seem to be more significant than problem size in determining the number of iterations required.

The algorithms used by the various primal–dual codes all involve the construction and Cholesky factorization of a symmetric and positive definite Schur complement matrix of size m by m in each iteration of the algorithm.

For the HKM method, the Schur complement matrix, O, is given by

$$O = \left[A(Z^{-1}A_1X), A(Z^{-1}A_2X, \ldots, A(Z^{-1}A_mX)\right]. \tag{5}$$

For dense X, Z, and A_j, the m products $Z^{-1}A_jX$ can be computed in $O(mn^3)$ time. Given $Z^{-1}A_jX$, computing $A(Z^{-1}A_jX)$ requires $O(mn^2)$ time. Thus in the worst case, for fully dense constraint matrices, the construction of the Schur complement matrix takes $O(mn^3 + m^2n^2)$ time.

In practice the constraint matrices are often extremely sparse. This sparsity can be exploited in the construction of the Schur complement matrix [11]. For sparse constraint matrices with $O(1)$ entries, $Z^{-1}A_jX$ can be computed in $O(n^2)$ time. Computing all m products $Z^{-1}A_jX$ takes $O(mn^2)$ time. Once the products have been computed, the $A()$ operations can be computed in $O(m^2)$ additional time.

The resulting Schur complement matrix is typically fully dense. Computing the Cholesky factorization of the dense Schur complement matrix takes $O(m^3)$ time.

In addition to the construction and factorization of the Schur complement matrix, the algorithms also require a number of operations on the X and Z matri-

ces, such as matrix multiplications, Cholesky factorization, and computation of eigenvalues. These operations require $O(n^3)$ time. When the matrices have block diagonal structure, this becomes $O(n_1^3 + \ldots + n_k^3)$.

The overall computational complexity of iterations of the primal-dual algorithm is dominated by different operations depending on the particular structure of the problem. For many problems, $m >> n$, and constraint matrices are sparse. In this case, the $O(m^3)$ operation of factoring the Schur complement matrix becomes dominant. On the other hand, when n is large compared to m, and the problem does not have many small blocks, the $O(n^3)$ time for operations on the X and Z matrix can be dominant. In other cases, when m and n are similar in size, and particularly when there are dense constraints, the construction of the Schur complement matrix can become the bottleneck.

Storage requirements are at least as important as the computational complexity. In practice, the size of the largest problems that can be solved often depends more on available storage than on available CPU time. In the worst case, storage for problem data including C, a, and the constraint matrices can require $O(mn^2)$ storage. However, in practice most constraints are sparse, with $O(1)$ entries per constraint, so that the constraint matrices take $O(m)$ storage. The C matrix, which often is dense, requires $O(n^2)$ storage in the worst case.

The Schur complement matrix is typically fully dense for SDP problems and requires $O(m^2)$ storage. This is in contrast to primal–dual methods for linear programming, where the Schur complement matrix is typically quite sparse. The X matrix is typically fully dense and requires $O(n_1^2 + \ldots + n_k^2)$ storage. The dual matrix Z may be either sparse or dense, and requires $O(n_1^2 + \ldots n_k^2)$ storage in

the worst case. There are typically several block diagonal work matrices used by the algorithm. For example, the storage requirements for CSDP include a total 11 matrices of size and block diagonal structure of X. The approximate storage requirements, ignoring lower order terms, for CSDP are

$$\text{Storage (Bytes)} = 8(m^2 + 11(n_1^2 + \ldots + n_k^2)). \tag{6}$$

The results on computational complexity and storage requirements summarized in this section shed useful light on the question of how far we can go with primal–dual interior point methods for SDP. In the typical case of sparse constraint matrices, with $m >> n$, running time will grow as $O(m^3)$, and storage required will grow as $O(m^2)$. This growth is relatively tame, so that as computers become more powerful, we should be able to make progress in solving larger problems.

3 A Parallel Version of CSDP

In this section, we describe a 64-bit parallel version of CSDP that has been developed to solve large SDP instances. This code is based on CSDP 4.9, with minor modifications. The code is written in ANSI C with additional OpenMP directives for parallel processing [9]. We also assume that parallelized implementations of BLAS and LAPACK are available [4, 1]. The code is available under both the GNU Public License (GPL) and the Common Public License (CPL). Hans Mittelmann at Arizona State University has also made the code available through NEOS [10].

CSDP makes extensive use of routines from the BLAS and LAPACK libraries to implement matrix multiplication, Cholesky factorization, and other linear algebra operations. Since most vendors already provide highly optimized parallel

implementations of these libraries, there was no need for us to reimplement the linear algebra libraries.

Outside of the BLAS and LAPACK routines, the major computationally intensive part of the code involves the creation of the Schur complement matrix. The serial routine for the creation of the Schur complement matrix from CSDP 4.9 was rewritten in parallel form using OpenMP directives.

The software was developed and tested on both a four processor Sunfire V480 server at Arizona State University and on an IBM p690 system with 1.3 GHz processors at the National Center for Supercomputer Applications (NCSA). The results reported here are based on computations performed at NCSA.

Since for most problems the "hot spot" in which the code spends most of its time is the Cholesky factorization of the Schur complement matrix, we began by performing tests on the parallel efficiency of the LAPACK routine DPOTRF which computes the Cholesky factorization of a matrix. Table 1 shows the wall clock times and parallel efficiencies for matrices of size $n = 5,000$ up to $n = 15,000$ and from one to sixteen processors. The parallel efficiencies are quite high for four processors, but drop off rapidly for eight or sixteen processors, especially on the smaller problems.

A collection of test problems was selected from the DIMACS library of mixed semidefinite-quadratic-linear programs, the SDPLIB collection of semidefinite programming problems, and from problems that have been solved in other papers [2, 6, 12, 14, 20].

Table 2 shows wall clock times and parallel efficiencies for the solution of some of the test problems using one to sixteen processors. In this table, m is the

	1	2	4	8	16
5000	15.6	7.7	4.2	3.4	1.7
10000	118.6	65.5	30.8	22.8	10.7
15000	427.3	210.6	106.2	64.8	31.9

	1	2	4	8	16
5000	100%	101%	93%	57%	57%
10000	100%	91%	96%	65%	69%
15000	100%	101%	101%	82%	84%

Table 1 Wall clock times (in seconds) and parallel efficiencies for DPOTRF on matrices of size $n = 5,000$ up to $n = 15,000$, with one to sixteen processors.

number of constraints, and nmax is the size of the largest block in the X matrix. Because of round–off errors, there are sometimes differences in the number of iterations required by the algorithm. For example, on problem CH4, most runs required 34 iterations, but with four processors, only 32 iterations were required. This results in an anomalous parallel efficiency of over 100%. Results for the smallest problem, maxG60, are also somewhat anomalous in that the parallel efficiencies are much lower than for the other problems. Because this problem was relatively small, and because this problem has $m = n$, some of the operations in the code that had not been parallelized became bottlenecks. Overall, the parallel efficiencies are quite good up to four processors but drop off somewhat for eight and sixteen processors.

Table 3 shows the results obtained using four processors on a somewhat larger collection of test problems. Here the number of constraints, m, varies from 7000 up to 56321, while the size of the largest block in X varies from 174 up to 8113. The wall clock time is reported in minutes and seconds. For each solution, the

Problem	m	nmax	1	2	4	8	16
CH4	24503	324	1609:36	787:29	434:40	185:24	122:38
fap09	15225	174	836:46	445:37	224:40	126:20	83:12
hamming_8_3_4	16129	256	148:21	82:45	52:26	24:33	16:14
hamming_10_2	23041	1024	671:02	318:30	201:47	93:41	63:21
maxG60	7000	7000	529:58	339:26	233:13	167:18	124:47
theta82	23872	400	640:24	282:33	174:46	94:42	57:40

Problem	m	nmax	1	2	4	8	16
CH4	24503	324	100%	98%	108%	92%	82%
fap09	15225	174	100%	94%	93%	83%	63%
hamming_8_3_4	16129	256	100%	90%	71%	76%	57%
hamming_10_2	23041	1024	100%	105%	83%	90%	66%
maxG60	7000	7000	100%	78%	57%	40%	27%
theta82	23872	400	100%	113%	92%	85%	69%

Table 2 Wall clock times (in minutes and seconds) and parallel efficiencies for the solution of selected SDP problems.

largest of the six DIMACS errors is reported[14]. Finally the storage required, in gigabytes, as reported by the operating system, is given for each solution.

For the fap and hamming families, m is significantly larger than n, and the constraint matrices are sparse, so that we would expect the running time to grow as $O(m^3)$. This relationship is roughly correct for the fap09 and fap12 problems. However, between hamming_8_3_4 and the other hamming problems, the running time grows somewhat slower than m^3. This is perhaps due to greater parallel efficiencies for the larger problems.

The DIMACS error measures show that all of these problems were solved to high accuracy.

Problem	m	nmax	Time	Error	Storage
CH4.1A1.STO6G.noncore.pqg	24503	324	434:40	2.3e-09	4.5G
fap09	15225	174	224:40	5.8e-09	1.8G
fap12	26462	369	944:25	1.2e-08	5.3G
hamming_8_3_4	16129	256	52:26	7.2e-08	2.0G
hamming_9_5_6	53761	512	1385:51	3.1e-07	21.7G
hamming_10_2	23041	1024	806:20	9.8e-08	4.1G
hamming_11_2	56321	2048	2083:46	7.1e-09	24.2G
ice_2.0	8113	8113	624:43	9.8e-09	7.9G
LiF.1Sigma.STO6G.pqg	15313	256	92:19	1.1e-09	1.8G
maxG60	7000	7000	233:13	1.2e-09	5.9G
p_auss2	9115	9115	1933:06	8.0e-08	9.9G
theta62	13390	300	34:57	1.6e-09	1.4G
theta82	23872	400	660:32	1.6e-09	4.3G

Table 3 Results with four processors.

4 Conclusions

Analysis of the complexity of the primal–dual interior point methods for SDP show that the storage required should grow quadratically in m and n, while for problems with sparse constraints, the growth in running time should be cubic in m and n.

We have described a 64 bit code running in parallel on shared memory system that has been used to solve semidefinite programming problems with over 50,000 constraints. This code obtained parallel efficiencies of 57% to 100% with four processors and 27% to 82% with 16 processors. As 64 bit processing, paral-

lel processors, and systems with large memory become common, the solution of SDP's of this size by primal–dual codes will become easier.

However, for the foreseeable future, primal–dual codes, even running on supercomputers, will not be able to solve much larger SDP's with many hundreds of thousands of constraints. Thus there is a continued need for research into methods for SDP that do not require the $O(m^2)$ storage used by the primal–dual methods.

Acknowledgements This work was partially supported by the National Computational Science Alliance under grant DMS040023 and utilized the IBM p690 system at NCSA. Hans Mittelmann at Arizona State University was also very helpful in allowing us to use the Sunfire server.

References

1. Anderson, E., Bai, Z., Bischof, C., Blackford, L.S., Demmel, J., Dongarra, J.J., Croz, J.D., Hammarling, S., Greenbaum, A., McKenney, A., Sorensen, D.: LAPACK Users' guide (third ed.). Society for Industrial and Applied Mathematics, Philadelphia (1999)
2. Benson, S.J.: Parallel computing on semidefinite programs. Tech. Rep. ANL/MCS–P939–0302, Argonne National Laboratory (2003)
3. Benson, S.J., Ye, Y.: DSDP3: Dual–scaling algorithm for semidefinite programming. Tech. Rep. ANL/MCS–P851-1000, Argonne National Laboratory (2001)
4. Blackford, L.S., Demmel, J., Dongarra, J., Duff, I., Hammarling, S., Henry, G., Heoux, M., Kaufman, L., Lumsdaine, A., Petitet, A., Pozo, R., Remington, K., Whaley, R.C.: An updated set of basic linear algebra subprograms (BLAS). ACM Transactions on Mathematical Software **28**(2), 135–151 (2002)
5. Borchers, B.: CSDP, a C library for semidefinite programming. Optimization Methods & Software **11-2**(1-4), 613 – 623 (1999)
6. Borchers, B.: SDPLIB 1.2, a library of semidefinite programming test problems. Optimization Methods & Software **11-2**(1-4), 683 – 690 (1999)

7. Burer, S., Choi, C.: Computational enhancements in low–rank semidefinite programming (2004). University of Iowa
8. Burer, S., Monteiro, R.D.C.: A nonlinear programming algorithm for solving semidefinite programs via low-rank factorization. Mathematical Programming **95**(2), 329 – 357 (2003)
9. Chandra, R., Dagum, L., Kohr, D., Maydan, D., andf R. Menon, J.M.: Parallel Programming in OpenMP. Morgan Kaufmann, New York (2000)
10. Czyzyk, J., Mesnier, M.P., Mor, J.J.: The NEOS server. IEEE Comput. Sci. Eng. **5**(3), 68–75 (1998)
11. Fujisawa, K., Kojima, M., K.Nakata: Exploiting sparsity in primal–dual interior–point methods for semidefinite programming. Mathematical Programming **79**, 235–253 (1997)
12. Keuchel, J., Schnorr, C., Schellewald, C., Cremers, D.: Binary partitioning, perceptual grouping, and restoration with semidefinite programming. IEEE Transactions on Pattern Analysis and Machine Intelligence **25**(11), 1364 – 1379 (2003)
13. Kocvara, M., Stingl, M.: Pennon: A code for convex nonlinear and semidefinite programming. Optimization Methods & Software **18**(3), 317 – 333 (2003)
14. Mittelmann, H.D.: An independent benchmarking of SDP and SOCP solvers. Mathematical Programming **95**(2), 407 – 430 (2003)
15. Sturm, J.F.: Using sedumi 1.02, a matlab toolbox for optimization over symmetric cones. Optimization Methods & Software **11-2**(1-4), 625 – 653 (1999)
16. Sturm, J.F.: Implementation of interior point methods for mixed semidefinite and second order cone optimization problems. Optimization Methods & Software **17**(6), 1105 – 1154 (2002)
17. Toh, K.C., Todd, M.J., Tutuncu, R.H.: SDPT3 - a MATLAB software package for semidefinite programming, version 1.3. Optimization Methods & Software **11-2**(1-4), 545 – 581 (1999)
18. Tutuncu, R.H., Toh, K.C., Todd, M.J.: Solving semidefinite-quadratic-linear programs using SDPT3. Mathematical Programming **95**(2), 189 – 217 (2003)
19. Yamashita, M., Fujisawa, K., Kojima, M.: Implementation and evaluation of SDPA 6.0 (semidefinite programming algorithm 6.0). Optimization Methods & Software **18**(4), 491 – 505 (2003)

20. Yamashita, M., Fujisawa, K., Kojima, M.: SDPARA: Semidefinite programming algorithm parallel version. Parallel Computing **29**, 1053–1067 (2003)