Open access • Journal Article • DOI:10.1007/S10589-007-9030-3

Implementation of a primal—dual method for SDP on a shared memory parallel architecture — Source link

Brian Borchers, Joseph Young

Institutions: New Mexico Institute of Mining and Technology, Rice University

Published on: 01 Jul 2007 - Computational Optimization and Applications (Kluwer Academic Publishers-Plenum Publishers)

Topics: Shared memory, Semidefinite programming and Interior point method

Implementation of a Primal-Dual Method for SDP on a Shared Memory Parallel Architecture

Brian Borchers* Joseph G. Young†

March 27, 2006

Abstract

Primal–dual interior point methods and the HKM method in particular have been implemented in a number of software packages for semidefinite programming. 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 storage requirements and required computational effort. In this paper we describe a parallel implementation of the primal-dual method on a shared memory system. Computational results are presented, including the solution of some large scale problems with over 50,000 constraints.

1 Introduction

A variety of methods for solving semidefinite programming problems have been implemented, including primal-dual interior point methods [5, 17, 16, 18, 19, 20], dual interior point methods [3], and augmented Lagrangian methods [7, 8, 14].

*Department of Mathematics, New Mexico Tech, 801 Leroy Place, Socorro, NM 87801, borchers@nmt.edu

†Computational and Applied Mathematics, Rice University, 6100 Main Street - MS 134, Houston, TX 77005

Of the 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[17]. 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[19].

There are two main differences between the codes. Some of the codes use the HKM search direction while others use the NT search direction. The other major difference between the codes is that some of the codes use an infeasible interior point approach while others use a self dual embedding approach. 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 interior–point 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 32–bit system can address only 4 gigabytes of RAM, which is inadequate for some of the larger problems solved in this paper. For example, the hamming_10_2 problem has 23,041 constraints, so the resulting Schur complement matrix has

23,041 rows and columns and requires 23.6 gigabytes of memory.

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. With 64–bit addressing, it would theoretically be possible to access over 10^{19} bytes of storage and handle a problem with over a billion constraints. In practice, 64–bit workstations often have 16 to 32 gigabytes of RAM while large servers may have 256 gigabytes or more of RAM. This allows for the solution of problems with tens of thousands of constraints, but not for the solution of problems with several hundred thousand or more constraints.

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[21]. 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. Recently, the authors of SDPA have produced a 64–bit version of their code that also takes advantage of shared memory[12].

2 Analysis

In this paper we consider semidefinite programming problems of the form

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

where

$$A(X) = \begin{bmatrix} \text{tr } (A_1 X) \\ \text{tr } (A_2 X) \\ \cdots \\ \text{tr } (A_m X) \end{bmatrix}. \tag{2}$$

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 (2) is

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

where

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

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[20].

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. 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 matrices, 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, $m \gg n$, 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 other operations on the X and Z matrix can be dominant. In cases where 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 of 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 the performance and storage requirements of primal–dual interior point methods for SDP scale with problem size. In the typical case of sparse constraint matrices, with $m \gg n$, 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 implemented on a shared memory system. This code is based on CSDP 5.0. 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].

Most 64–bit computers use a computational model in which integers are stored as 32–bit numbers while long integers and pointers to data structures are stored as 64–bit quantities. In converting the existing 32–bit code to 64–bit form it was necessary to search carefully for any places in the code where it was assumed that pointers were 32–bit quantities. For well written C code, such errors are not common and they can easily be fixed when found. A second issue was that 32–bit integers were used in some places as indices into large arrays that could exceed 2^{32} entries. These integer variables were retyped as long integer variables.

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. Although the C compilers that we used to compile this code were capable of automatically parallelizing loops, this automatic parallelization is often not as efficient as explicitly specifying the parallelization.

Our initial attempt to parallelize this code involved the use of automatic compiler parallelization. Tables 1 and 2 show the run times and parallel efficiencies for a small collection of test problems. For each problem, the time spent computing the elements of the Schur complement matrix, the Cholesky factorization of the Schur complement matrix, and other operations are given. Speedups were computed for each phase of of the computation and parallel efficiencies were obtained by dividing each speedup by the number of processors.

In the control10 problem, the computation of the elements of the Schur complement matrix dominates the total solution time. Unfortunately, this scales very poorly with the number of processors, so that overall performance scales poorly. In the maxG51 problem, other operations dominate the total solution time. The time to perform these operations scales poorly with the number of processors. In the theta6 problem, the time to compute the Cholesky factorization of the Schur complement matrix is somewhat time consuming but not completely dominant. Although the Cholesky factorization scales well with the number of processors, the computation of elements of the Schur complement matrix and other operations do not scale well, and the overall performance of the code on these problems is poor.

It is clear from these initial results that although the Cholesky factorization scales well, it is necessary to improve the parallelization of the computation of the elements of the Schur complement matrix and other operations. For this reason, the serial routine for the creation of the Schur complement matrix from CSDP was rewritten in explicitly parallel form using OpenMP directives. In this version of the code, individual processors are assigned to work on horizontal strips of the Schur complement matrix, with each processor computing a block of 16 rows of the matrix before moving on to another block. Similar modifications were made to other routines in CSDP.

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.

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, 13, 15, 21].

Tables 3 and 4 show run times and parallel efficiencies for the solution of the test problems using one to sixteen processors. In these tables, m is the number of constraints, and $n_{\max}$ is the size of the largest block in the X matrix. Run times are given in seconds.

The table of parallel efficiencies shows some superlinear speedup anomalies. Superlinear speedups can occur because the system makes more cache memory available as additional processors are used. Also, there are sometimes differences in the number of iterations required by the algorithm. For example, on problem CH4, most runs required 30 iterations, but with one processor, 32 iterations were required. This results in anomalous parallel efficiencies of over 100%.

In these results, we see that the performance of both the Cholesky factorization and the construction of the Schur complement matrix scale well with the number of processors. However, the performance of other operations does not scale as well, and in cases where these operations require a significant amount of time, this effects the overall performance of the code. For example, on the maxG51, maxG55, and maxG60 problems, other operations dominate the running time, and scale poorly with the number of processors. In problems where the computation of the elements of Schur complement matrix and the Cholesky factorization take most of the running time, the performance of CSDP scales

very well with the number of processors.

It is difficult to directly compare run times for CSDP with run times for SDPARA and PDSDP, since the codes have been run on systems with different processors. The processors used by CSDP are 1.3 GHz Power 4 processors, while SDPARA was run on a cluster of machines with 1.6 GHz Athlon processors, and PDSDP was run on a cluster of machines with 2.4 GHz Pentium Xeon processors. However, it is reasonable to compare the parallel efficiencies of the codes.

In order to calculate parallel efficiency, we must know the solution time for a problem using only one processor. Unfortunately, SDPARA was able to solve only four problems using a single processor[21]. Table 5 shows the parallel efficiencies for SDPARA using between 1 and 64 processors on these four problems. In general, the computation of the elements of the Schur complement matrix scales very well, while the Cholesky factorization and other computations scale poorly. On the control10 and control11 problems, SDPARA has somewhat better parallel efficiency than CSDP with 16 processors. On the theta5 and theta6 problems, CSDP has better parallel efficiency than SDPARA with 16 processors. In SDPARA, the performance of the Cholesky factorization scales poorly, while the performance of the computation of the elements of the Schur complement matrix scales well.

Similarly, we computed parallel efficiencies for the problems solved with 1 to 32 processors by PDSDP[2]. These efficiencies are shown in Table 6. The parallel efficiency of over 400% in the elements computation on problem theta62 with two processors is an unexplained anomaly. Overall the parallel efficiency with 16 processors is better for CSDP than for PDSDP on all but two of the problems. Again, we see that in PDSDP the Cholesky factorization scales poorly, while the computation of the elements of the Schur complement matrix scales well.

Problem	m	$n_{\max}$	Phase	1	2	4	8	16
control10	1326	100	Elements	167.1	200.0	182.2	159.4	174.3
			Cholesky	10.0	6.0	4.1	2.4	1.1
			Other	20.6	28.3	27.0	23.9	27.1
			Total	197.7	234.3	213.3	185.7	202.5
maxG51	1000	1000	Elements	2.8	3.6	3.7	3.1	2.9
			Cholesky	2.0	1.2	0.7	0.4	0.2
			Other	134.9	98.2	65.8	43.5	30.6
			Total	139.7	103.0	70.2	47.0	33.7
theta6	4375	300	Elements	73.9	82.6	103.6	108.4	103.4
			Cholesky	174.8	84.9	51.2	24.9	14.3
			Other	18.9	17.9	22.1	24.6	20.3
			Total	267.6	185.4	176.9	157.9	138.0

Table 1: Run times (in seconds) for the solution of selected SDP problems using CSDP 5.0 with automatic compiler parallelization.

Problem	m	$n_{\max}$	Phase	1	2	4	8	16
control10	1326	100	Elements	100	42	23	13	6
			Cholesky	100	83	61	52	57
			Other	100	36	19	11	5
			Total	100	42	23	13	6
maxG51	1000	1000	Elements	100	39	19	11	6
			Cholesky	100	83	71	63	63
			Other	100	69	51	39	28
			Total	100	68	50	37	26
theta6	4375	300	Elements	100	45	18	9	4
			Cholesky	100	103	85	88	76
			Other	100	53	21	10	6
			Total	100	72	38	21	12

Table 2: Percent parallel efficiencies for selected SDP problems using CSDP 5.0 with automatic compiler parallelization.

Problem	m	$n_{\max}$	Phase	1	2	4	8	16
CH4	24503	324	Elements	11299.5	3824.2	2595.3	1214.4	603.4
			Cholesky	81276.0	32213.1	18276.0	9000.4	4333.4
			Other	3328.5	1395.6	1085.3	690.6	756.9
			Total	95904.0	37432.9	21956.6	10905.4	5693.7
control10	1326	100	Elements	172.5	106.7	41.7	16.9	12.0
			Cholesky	11.0	6.5	3.3	1.9	0.9
			Other	23.0	22.9	16.5	11.3	10.5
			Total	206.5	136.1	61.5	30.1	23.4
control11	1596	110	Elements	251.3	154.7	64.7	32.9	21.1
			Cholesky	19.4	11.7	5.5	2.6	1.4
			Other	29.6	30.7	20.9	19.5	20.5
			Total	300.3	197.1	31.1	55.0	43.0
fap09	15225	174	Elements	8999.7	3764.6	2053.5	1011.8	494.2
			Cholesky	39341.3	18083.3	9242.0	4561.0	2202.4
			Other	2489.1	1144.5	631.7	361.4	216.9
			Total	50830.1	22992.4	11927.2	5934.2	2913.5
hamming_8_3_4	16129	256	Elements	879.0	579.5	231.6	120.1	58.4
			Cholesky	7538.1	4251.6	1935.8	973.6	479.6
			Other	430.6	334.3	138.3	78.4	44.3
			Total	8847.7	5165.4	2305.7	1172.1	582.3
hamming_10_2	23041	1024	Elements	2629.3	1401.0	683.5	286.2	148.4
			Cholesky	34083.0	17596.0	8704.3	3921.8	2070.7
			Other	1693.9	1100.5	696.2	380.5	289.4
			Total	38406.2	20097.5	10084.0	4588.5	2508.5
LiF	15313	256	Elements	3283.8	1536.2	665.9	367.0	456.9
			Cholesky	14166.5	6635.1	3207.6	1727.7	1995.4
			Other	929.5	507.8	278.5	213.5	698.0
			Total	18379.8	8679.1	4152.0	2308.2	3150.3
maxG51	1000	1000	Elements	1.1	0.7	0.3	0.2	0.1
			Cholesky	2.3	1.1	0.7	0.3	0.2
			Other	143.2	78.6	50.7	35.4	26.4
			Total	146.6	80.4	51.7	35.9	26.7
maxG55	5000	5000	Elements	64.5	32.8	21.5	12.7	5.5
			Cholesky	275.5	128.6	71.3	37.8	16.5
			Other	7802.2	4943.1	4143.6	4577.2	2475.4
			Total	8142.2	5104.5	4236.4	4627.7	2497.4
maxG60	7000	7000	Elements	158.9	94.9	49.7	29.3	14.1
			Cholesky	785.4	397.5	199.3	108.2	50.5
			Other	21123.1	14622.3	10366.3	8993.6	6704.0
			Total	22067.4	15114.7	10615.3	9131.1	6768.6
theta4	1949	200	Elements	7.0	4.1	1.5	0.8	0.4
			Cholesky	19.3	8.9	4.2	2.5	1.0
			Other	5.6	3.7	2.1	1.8	1.8
			Total	31.9	16.7	7.8	5.1	3.2
theta5	3028	250	Elements	23.9	12.2	5.4	2.6	1.2
			Cholesky	65.1	31.6	15.5	7.9	3.7
			Other	11.5	7.3	4.6	3.3	2.6
			Total	100.5	51.1	25.5	13.8	7.5
theta6	4375	300	Elements	36.6	23.5	12.7	6.7	3.2
			Cholesky	171.4	89.2	45.2	23.1	11.2
			Other	18.7	14.0	8.8	6.0	5.6
			Total	226.7	126.7	66.7	35.8	20.0
theta8	7905	400	Elements	294.5	98.1	72.7	23.9	13.9
			Cholesky	1142.6	500.7	284.7	118.9	60.3
			Other	139.6	54.7	41.2	16.8	12.3
			Total	1576.7	653.5	398.6	159.6	86.5
theta42	5986	200	Elements	133.0	62.4	23.1	12.2	8.7
			Cholesky	472.1	238.0	100.1	506.	29.6
			Other	53.1	28.2	11.5	6.9	6.7
			Total	658.2	328.6	134.7	69.7	45.0
theta62	13390	300	Elements	639.5	363.8	174.9	87.7	49.0
			Cholesky	4923.9	2641.8	1266.0	622.9	335.8
			Other	311.5	192.1	100.6	50.6	33.9
			Total	5874.9	3197.7	1541.5	761.2	418.7
theta82	23872	400	Elements	2521.4	1234.8	673.9	323.9	156.0
			Cholesky	30368.0	14577.8	7503.6	3750.4	1829.3
			Other	1239.1	653.8	360.5	184.4	101.8
			Total	34128.5	16466.4	8538.0	4258.7	2087.1

Table 3: Run times (in seconds) for the solution of selected SDP problems.

Problem	m	$n_{\max}$	Phase	1	2	4	8	16
CH4	24503	324	Elements	100	148	109	116	117
			Cholesky	100	126	111	113	117
			Other	100	119	77	60	27
			Total	100	128	109	110	105
control10	1326	100	Elements	100	81	103	128	90
			Cholesky	100	85	83	72	76
			Other	100	50	35	25	14
			Total	100	76	84	86	55
control11	1596	110	Elements	100	81	97	95	74
			Cholesky	100	83	88	93	87
			Other	100	48	35	19	9
			Total	100	76	82	68	44
fap09	15225	174	Elements	100	120	110	111	114
			Cholesky	100	109	106	108	112
			Other	100	109	99	86	72
			Total	100	111	107	107	109
hamming_8_3_4	16129	256	Elements	100	76	95	91	94
			Cholesky	100	89	97	97	98
			Other	100	64	78	69	61
			Total	100	86	96	94	95
hamming_10_2	23041	1024	Elements	100	94	96	115	111
			Cholesky	100	97	98	109	103
			Other	100	77	61	56	37
			Total	100	96	95	105	96
LiF	15313	256	Elements	100	107	123	112	45
			Cholesky	100	107	110	102	44
			Other	100	92	83	54	8
			Total	100	106	111	100	36
maxG51	1000	1000	Elements	100	79	92	69	69
			Cholesky	100	105	82	96	72
			Other	100	91	71	51	34
			Total	100	91	71	51	34
maxG55	5000	5000	Elements	100	98	75	63	73
			Cholesky	100	107	97	91	104
			Other	100	79	47	21	20
			Total	100	80	48	22	20
maxG60	7000	7000	Elements	100	84	80	68	70
			Cholesky	100	99	99	91	97
			Other	100	72	51	29	20
			Total	100	73	52	30	20
theta4	1949	200	Elements	100	85	117	109	109
			Cholesky	100	108	115	97	121
			Other	100	76	67	39	19
			Total	100	96	102	78	62
theta5	3028	250	Elements	100	98	111	115	124
			Cholesky	100	103	105	103	110
			Other	100	79	62	44	28
			Total	100	98	99	91	84
theta6	4375	300	Elements	100	78	72	68	71
			Cholesky	100	96	95	93	96
			Other	100	67	53	39	21
			Total	100	89	85	79	71
theta8	7905	400	Elements	100	150	101	154	132
			Cholesky	100	114	100	120	118
			Other	100	128	85	104	71
			Total	100	121	99	123	114
theta42	5986	200	Elements	100	107	144	136	96
			Cholesky	100	99	118	117	100
			Other	100	94	115	96	50
			Total	100	100	122	118	91
theta62	13390	300	Elements	100	88	91	91	82
			Cholesky	100	93	97	99	92
			Other	100	81	77	77	57
			Total	100	92	95	96	88
theta82	23872	400	Elements	100	102	94	97	101
			Cholesky	100	104	101	101	104
			Other	100	95	86	84	76
			Total	100	104	100	100	102

Table 4: Percent parallel efficiencies for selected SDP problems.

Table 7 shows the results obtained using four processors on a somewhat larger collection of test problems using four processors. Here the number of constraints, m, varies from 1326 up to 56321, while the size of the largest block in X varies from 100 up to 8113. Run times are given in seconds. For each solution, the largest of the six DIMACS errors is reported[15]. The DIMACS error measures show that all of these problems were solved to high accuracy. Finally, the storage in gigabytes required, 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 fap and hamming problems.

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 a shared memory system. In comparison with primal-dual codes running on distributed memory systems, the scalability of the Cholesky factorization of the Schur complement matrix is improved substantially. Our code has been used to solve semidefinite programming problems with over 50,000 constraints. This code obtained parallel efficiencies of 48% to 122% with four processors and 20% to 114% with 16 processors.

As 64–bit processing, shared memory parallel processors, and systems with large memory become common, the solution of SDP's of this size by primal–dual

codes will become common.

Significant challenges remain. One important challenge is the effective parallelization of SDP solvers on supercomputers with hundreds or thousands of processors. Another important challenge is the solution of very large problems with hundreds of thousands of constraints. For the foreseeable future, primal–dual codes, even running on supercomputers, will not have enough memory to solve such very large SDP's. 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.

5 Acknowledgments

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 a Sunfire server at ASU. We would like to thank an anonymous referee who provided very helpful comments.

Problem	m	$n_{\max}$	Phase	1	2	4	8	16	32	64
control10	1326	100	Elements	100	102	101	104	109	106	103
			Cholesky	100	87	65	39	31	16	13
			Other	100	39	25	14	9	5	2
			Total	100	94	85	74	65	43	28
control11	1596	110	Elements	100	103	103	102	105	95	105
			Cholesky	100	93	73	44	34	19	16
			Other	100	34	23	15	8	6	2
			Total	100	94	88	76	64	50	34
theta5	3028	250	Elements	100	111	117	121	125	121	114
			Cholesky	100	120	98	68	59	35	32
			Other	100	40	25	15	9	6	3
			Total	100	97	77	54	40	27	17
theta6	4375	300	Elements	100	112	119	123	126	124	117
			Cholesky	100	129	112	83	73	48	38
			Other	100	20	26	17	12	7	4
			Total	100	88	89	67	55	37	25

Table 5: Percentage parallel efficiencies for problems solved by SDPARA. Times were taken from [21].

Problem	m	$n_{\max}$	Phase	1	2	4	8	16	32
control10	1326	100	Elements	100	98	82	69	67	61
			Cholesky	100	76	59	34	18	9
			Other	100	68	39	26	13	7
			Total	100	95	76	61	49	36
control11	1596	110	Elements	100	99	82	71	74	64
			Cholesky	100	77	65	38	23	11
			Other	100	74	50	31	18	10
			Total	100	96	78	64	58	42
maxG51	1000	1000	Elements	100	99	78	73	63	50
			Cholesky	100	74	50	28	15	7
			Other	100	51	26	13	7	3
			Total	100	82	56	36	21	10
maxG55	5000	5000	Elements	100	102	69	69	72	65
			Cholesky	100	80	71	56	45	30
			Other	100	57	28	14	7	4
			Total	100	90	60	46	33	20
maxG60	7000	7000	Elements	100	101	64	65	51	43
			Cholesky	100	91	83	69	54	39
			Other	100	48	24	10	6	3
			Total	100	90	58	44	31	19
theta4	1949	200	Elements	100	100	61	51	37	28
			Cholesky	100	78	66	45	28	14
			Other	100	50	27	12	6	3
			Total	100	83	62	44	28	16
theta6	4375	300	Elements	100	102	59	52	41	33
			Cholesky	100	83	73	56	45	29
			Other	100	65	27	18	6	4
			Total	100	87	67	54	40	28
theta8	7905	400	Elements	100	96	50	49	41	36
			Cholesky	100	88	78	69	58	41
			Other	100	47	22	13	7	3
			Total	100	89	69	62	51	37
theta42	5986	200	Elements	100	100	49	43	36	31
			Cholesky	100	87	77	64	51	36
			Other	100	54	16	9	7	3
			Total	100	88	70	58	47	34
theta62	13390	300	Elements	100	453	215	196	169	157
			Cholesky	100	96	93	83	74	58
			Other	100	118	62	31	16	8
			Total	100	124	111	99	86	68

Table 6: Percentage parallel efficiencies for problems solved by PDSDP. Times were taken from [2].

Problem	m	$n_{\max}$	Elements	Cholesky	Other	Total	Error	Storage
CH4.1A1.STO6G.noncore.pqg	24503	324	2595.3	18276.0	1085.3	21956.6	7.7e-09	4.54G
control10	1326	100	41.7	3.3	16.5	61.5	8.5e-08	0.03G
control11	1596	110	64.7	5.5	20.9	91.1	1.2e-07	0.03G
fap09	15225	174	2053.5	9242.0	631.7	11927.2	1.4e-08	2.93G
fap12	26462	369	4138.6	34915.1	4529.8	40583.5	4.4e-09	5.29G
hamming_8_3_4	16129	256	231.6	1935.8	138.3	2305.7	6.1e-07	1.98G
hamming_9_5_6	53761	512	2935.6	91769.6	2683.7	97388.9	1.3e-07	21.70G
hamming_10_2	23041	1024	683.5	8704.3	696.2	10084.0	8.3e-07	4.13G
hamming_11_2	56321	2048	4863.9	134235.7	3955.8	143055.4	1.1e-06	24.30G
ice_2.0	8113	8113	222.5	1144.7	97299.9	98667.1	5.4e-07	7.86G
LiF.1Sigma.STO6G.pqg	15313	256	665.9	3207.6	278.5	4152.0	3.3e-09	1.79G
maxG51	1000	1000	0.3	0.7	50.7	51.7	2.3e-09	0.12G
maxG55	5000	5000	21.5	71.3	4143.6	4236.4	1.1e-08	2.98G
maxG60	7000	7000	49.7	199.3	10366.3	10615.3	2.4e-08	5.85G
p_auss2	9115	9115	296.0	1751.4	246432.4	248479.8	1.0e-08	9.93G
theta4	1949	200	1.5	4.2	2.1	7.8	8.1e-09	0.04G
theta5	3028	250	5.4	15.5	4.6	25.5	2.8e-08	0.09G
theta6	4375	300	12.7	45.2	8.8	66.7	2.3e-07	0.17G
theta8	7905	400	72.7	284.7	41.2	398.6	2.4e-07	0.51G
theta42	5986	200	23.1	100.1	11.5	134.7	1.5e-07	0.29G
theta62	13390	300	174.9	1266.0	100.6	1541.5	1.6e-08	1.37G
theta82	23872	400	673.9	7503.6	360.5	8533.0	2.4e-08	4.31G

Table 7: Results with four processors.

References

[1] E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. Demmel, Jack J. Dongarra, J. Du Croz, S. Hammarling, A. Greenbaum, A. McKenney, and D. Sorensen. *LAPACK Users' guide (third ed.)*. Society for Industrial and Applied Mathematics, Philadelphia, 1999.

[2] Steven J. Benson. Parallel computing on semidefinite programs. Technical Report ANL/MCS–P939–0302, Argonne National Laboratory, 2003.

[3] Steven J. Benson and Yinyu Ye. DSDP3: Dual–scaling algorithm for semidefinite programming. Technical Report ANL/MCS–P851-1000, Argonne National Laboratory, November 2001.

[4] L. S. Blackford, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heoux, L. Kaufman, A. Lumsdaine, A. Petitet, R. Pozo, K. Remington, and R. C. Whaley. An updated set of basic linear algebra subprograms (BLAS). *ACM Transactions on Mathematical Software*, 28(2):135–151, 2002.

[5] B. Borchers. CSDP, a C library for semidefinite programming. *Optimization Methods & Software*, 11-2(1-4):613 – 623, 1999.

[6] B. Borchers. SDPLIB 1.2, a library of semidefinite programming test problems. *Optimization Methods & Software*, 11-2(1-4):683 – 690, 1999.

[7] S. Burer and C. Choi. Computational enhancements in low–rank semidefinite programming. University of Iowa, July 2004.

[8] S. Burer and R. D. C. Monteiro. A nonlinear programming algorithm for solving semidefinite programs via low-rank factorization. *Mathematical Programming*, 95(2):329 – 357, 2003.

[9] R. Chandra, L. Dagum, D. Kohr, D. Maydan, and J. McDonald andf R. Menon. *Parallel Programming in OpenMP*. Morgan Kaufmann, New York, 2000.

[10] Joseph Czyzyk, Michael P. Mesnier, and Jorge J. Mor. The NEOS server. *IEEE Comput. Sci. Eng.*, 5(3):68–75, 1998.

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

[12] Mituhiro Fukuda, Bastiaan J. Braams, Maho Nakata, Michael L. Overton, Jerome K. Percus, Makoto Yamashita, and Zhengji Zhao. Large-scale semidefinite programs in electronic structure calculation. Technical Report B–413, Tokyo Institute of Technology, Department of Mathematical and Computing Sciences, February 2005.

[13] J. Keuchel, C. Schnorr, C. Schellewald, and D. Cremers. Binary partitioning, perceptual grouping, and restoration with semidefinite programming. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 25(11):1364 – 1379, November 2003.

[14] M. Kocvara and M. Stingl. Pennon: A code for convex nonlinear and semidefinite programming. *Optimization Methods & Software*, 18(3):317 – 333, 2003.

[15] H. D. Mittelmann. An independent benchmarking of SDP and SOCP solvers. *Mathematical Programming*, 95(2):407 – 430, 2003.

[16] J. F. Sturm. Using sedumi 1.02, a matlab toolbox for optimization over symmetric cones. *Optimization Methods & Software*, 11-2(1-4):625 – 653, 1999.

[17] J. F. Sturm. Implementation of interior point methods for mixed semidefinite and second order cone optimization problems. *Optimization Methods & Software*, 17(6):1105 – 1154, 2002.

[18] K. C. Toh, M. J. Todd, and R. H. Tutuncu. SDPT3 - a MATLAB software package for semidefinite programming, version 1.3. *Optimization Methods & Software*, 11-2(1-4):545 – 581, 1999.

[19] R. H. Tutuncu, K. C. Toh, and M. J. Todd. Solving semidefinite-quadratic-linear programs using SDPT3. *Mathematical Programming*, 95(2):189 – 217, 2003.

[20] M. Yamashita, K. Fujisawa, and M. Kojima. Implementation and evaluation of SDPA 6.0 (semidefinite programming algorithm 6.0). *Optimization Methods & Software*, 18(4):491 – 505, 2003.

[21] M. Yamashita, K. Fujisawa, and M. Kojima. SDPARA: Semidefinite programming algorithm parallel version. *Parallel Computing*, 29:1053–1067, 2003.