Parallel Computing on Semidefinite Programs

Steven J. Benson
Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL, 60439

Revised: April 22, 2003

Abstract

This paper demonstrates how interior-point methods can use multiple processors efficiently to solve large semidefinite programs that arise in VLSI design, control theory, and graph coloring. Previous implementations of these methods have been restricted to a single processor. By computing and solving the Schur complement matrix in parallel, multiple processors enable the faster solution of medium and large problems. The dual-scaling algorithm for semidefinite programming was adapted to a distributed-memory environment and used to solve medium and large problems than faster than could previously be solved by interior-point algorithms. Three criteria that influence the parallel scalability of the solver are identified. Numerical results show that on problems of appropriate size and structure, the implementation of an interior-point method exhibits good scalability on parallel architectures.

Key words. Semidefinite programming, numerical optimization, parallel computing, high-performance computing.

1 Introduction

Semidefinite programming (SDP) has been an actively studied area of numerical optimization. One reason for the high level of interest is that applications of this class of optimization problem have been found in fields as diverse as structural design, control theory, and combinatorial optimization. A second reason is that although interior-point methods adopted from linear programming have proven reliable on small and medium-sized semidefinite programs, the computational and storage demands of these methods have exhausted the resources of most computers and limited the size of problems that can be solved. Much of the research in this field focuses on solving medium-scale problems more quickly and on solving large-scale problems by any means possible.

The positive semidefinite program in standard form is

$$(SDP) \qquad \inf\ C \bullet X \qquad \text{subject to}\ \ A_i \bullet X = b_i,\ i = 1, \ldots, m, \quad X \in K$$

where $C, A_i \in \Re^{n \times n}$ are given symmetric matrices, $b \in \Re^m$ is a given vector, and $K = K_1 \otimes K_2 \otimes \cdots \otimes K_r$ is the cone where the variable X resides. Furthermore, K_j, $j = 1, \ldots, r$ is the set of $n_j \times n_j$ symmetric positive semidefinite matrices such that $n_j > 0$ and $\sum_{j=1}^{r} n_j = n$. The notation $X \succeq 0$ means that X is positive semidefinite. We use the notation $A \succ (\succeq) B$ to denote that $A - B$ is positive (semi)definite. The operation $C \bullet X = \text{tr}\ C^T X = \sum_{jk} C_{jk} X_{jk}$. We assume the matrices A_i are linearly independent. Matrices X that satisfy the constraints are called feasible, while the others are called infeasible.

The dual of (SDP) can be written:

$$(DSP) \qquad \sup\ b^T y \qquad \text{subject to}\ \sum_{i=1}^{m} y_i A_i + S = C,\ S \in K.$$

Variables (y, S) that satisfy the constraints are called feasible. It is well known that if both SDP or DSP have feasible points such that the variable matrices are positive definite, the optimal objective values of these two problems are equal.

Various approaches have been tried to solve these positive semidefinite programs. These approaches include primal-dual interior-point methods (see Todd [23] for a survey) and a dual-scaling interior-point method of Benson, Ye, and Zhang [7]. Other approaches include the partial Lagrangian approach of Helmberg and Rendl [17] that uses a spectral bundle method to solve the nondifferentiable convex program, a penalty approach by Kočvara and Stingl [19], low-rank factorizations of Burer and Monteiro [10], and transformation to a constrained nonlinear program proposed by Burer and Monteiro [9] and Burer, Monteiro, and Zhang [11]. A discussion and comparison of these methods can be found in [24].

Some of these methods are particularly well suited for large-scale problems [21]. In particular, the spectral bundle method and low rank factorizations have solved solved some large instances of SDP. However, these methods lack polynomial convergence in theory and sometimes exhibit slow convergence in practice. Toh and Kojima [25] solved some very large semidefinite programs with a primal-dual method. The use of an iterative linear solver enabled Toh and Kojima to compute a step direction without storing the Schur complement matrix in memory. The examples used to test their implementation will also be used in this work to demonstrate the success of the dual-scaling method in parallel. Others who have used iterative solvers and preconditioners for SDP include Choi and Ye [12] and Lin and Saigal [20]. None of the methods

mentioned above have considered the use of parallel processors when implementing methods to solve semidefinite programs. (After the initial submission of this paper, Yamashita, Fujisawa, and Kojima[28] presented a parallel implementation of a primal-dual interior point method.)

This paper addresses issues that arise in the parallel implementation and performance of interior-point methods for positive semidefinite programming. Section 2 briefly describes the dual-scaling algorithm. Section 3 shows how the algorithm can be implemented in parallel using either an iterative or direct linear solver. Section 4 presents numerical results that show the implementation exhibit good parallel efficiency on a set of large instances of SDP. Section 5 presents results on a modification of the implementation that reduces the memory requirements by solving a linear system without storing the entire matrix. The discussion focuses on the dual-scaling algorithm, but many of the issues also apply to primal-dual methods.

2 Dual-Scaling Algorithm

Fundamental to interior-point methods is the concept of the central path. This path consists of feasible points X and (y, S) such that $XS = \hat{\mu}I$ for $\hat{\mu} \geq 0$. Feasible points such that $XS = 0$ constitute a solution to the semidefinite program. Following conventional notations, let

$$\mathcal{A}X = \begin{bmatrix} A_1 \bullet X & \cdots & A_m \bullet X \end{bmatrix}^T \quad \text{and} \quad \mathcal{A}^T y = \sum_{i=1}^{m} A_i y_i.$$

Given a dual point (y, S) such that $\mathcal{A}^T y + S - C = R$ (R is the residual matrix), $S \succ 0$, a feasible matrix X, and a barrier parameter $\hat{\mu} > 0$, each iteration of the dual-scaling algorithm linearizes the equations

$$\mathcal{A}X = b, \qquad \mathcal{A}^T y + S = C, \qquad \hat{\mu}S^{-1} = X,$$

and solves the Schur complement of

$$\mathcal{A}(\Delta X) = b - \mathcal{A}X, \qquad \mathcal{A}^T(\Delta y) + \Delta S = -R, \qquad \hat{\mu}S^{-1}\Delta S S^{-1} + \Delta X = \hat{\mu}S^{-1} - X,$$

given by

$$\begin{pmatrix} S^{-1}A_1S^{-1} \bullet A_1 & \cdots & S^{-1}A_1S^{-1} \bullet A_m \\ \vdots & \ddots & \vdots \\ S^{-1}A_mS^{-1} \bullet A_1 & \cdots & S^{-1}A_mS^{-1} \bullet A_m \end{pmatrix} \Delta y = \frac{1}{\hat{\mu}} b - \mathcal{A}(S^{-1}) - \mathcal{A}(S^{-1}RS^{-1}). \tag{1}$$

The solution Δy is used to compute $\Delta S = -\mathcal{A}^T \Delta y - R$ and

$$X(S, \hat{\mu}) = \hat{\mu}S^{-1} - \hat{\mu}S^{-1}\Delta S S^{-1}$$

that satisfies the constraints $\mathcal{A}X(S, \hat{\mu}) = b$. Since the X variable does not appear in (1), it does not have to be given to begin the algorithm and it does not have to be computed at each iteration. For notational convenience, we denote M to be matrix on the left-hand side of (1). A more detailed explanation and derivation of the algorithm can be found in [7] and [29].

Computing the matrix M in (1) and solving the equations are the two most computationally expensive parts of the algorithm. For arbitrary matrices A_i and C, (1) can be computed by using $O(n^3m + n^2m^2)$ operations and solved by using $O(m^3)$ operations, although sparsity in the data may reduce the cost of the former part. The remaining operations include computing ΔS, the step length, and S, which are relatively inexpensive, and the factorization of S. This algorithm does not require the computation of $X(S, \mu)$ at each iteration. Primal-dual methods use a symmetric linearization of the equation $XS = \hat{\mu}I$ and form a Schur complement system with a similar form and complexity as (1).

With a feasible dual starting point and appropriate choices for $\hat{\mu}$ and step length, convergence results in [5, 7, 29] show that either the new dual point (y, S) or the new primal point X is feasible and reduces the Tanabe-Todd-Ye primal-dual potential function

$$\Psi(X, S) = \rho \ln(X \bullet S) - \ln \det X - \ln \det S$$

enough to achieve linear convergence.

3 PDSDP

The DSDP4 [5, 6] software package served as the basis for a parallel implementation of the dual-scaling algorithm. The parallel implementation of the dual-scaling method, called PDSDP, reads the problem data and stores it on each processor in an MPI [16] communicator. With the data, each processor computes and factors its own copy of the dual matrix S. The matrix and vectors in (1) are distributed over the processors. The matrix M and linear solver associated with it in DSDP4 were replaced by distributed matrix structures and parallel linear solvers. A parallel implementation of both the preconditioned conjugate gradient method and the Cholesky factorization was used. Collectively, each of the processors in the communicator computes (1) and solves the equations in parallel using one of these two linear solvers. After replacing these data structures, only a few changes in the software were needed to solve semidefinite programs in parallel. Figure 1 shows what data structures in PDSDP were allocated on each processor in a communicator of size 4.

The iterative solver came from PETSc [2][3][4]. Among the tools in PETSc are distributed vectors, distributed matrix structures, and the preconditioned conjugate gradient method for linear equations. For dense matrices, the preconditioner is simply the diagonal of the matrix. Scalability in the solver is achieved by applying the matrix-vector product, vectors sums, and vector inner products in parallel over the distributed vector and matrix objects. The vector Δy in Figure 1 has a darker shade because the solution to the linear system is computed collectively by all processors used by the linear solver. PETSc assumes a distributed-memory parallel environment and uses MPI for all message passing between processors.

The rows of the PETSc matrix are distributed over the processors as shown in Figure 1. There is no interleaving of rows among processors, and there is no distinction between the lower and upper triangular portions of matrices to identify symmetry. Since the matrix M is symmetric, only half of the nondiagonal elements need be explicitly computed, but these elements need to be assembled into two places in the matrix. Figure 1 also shows which elements of M are computed by each processor: each processor computes the shaded elements of its rows of the matrix M. This pattern was chosen to evenly distribute the number of elements of M computed on each row and each processor. These elements are inserted in the appropriate places in the matrix, a process that may require sending this information to another processor. In PDSDP,

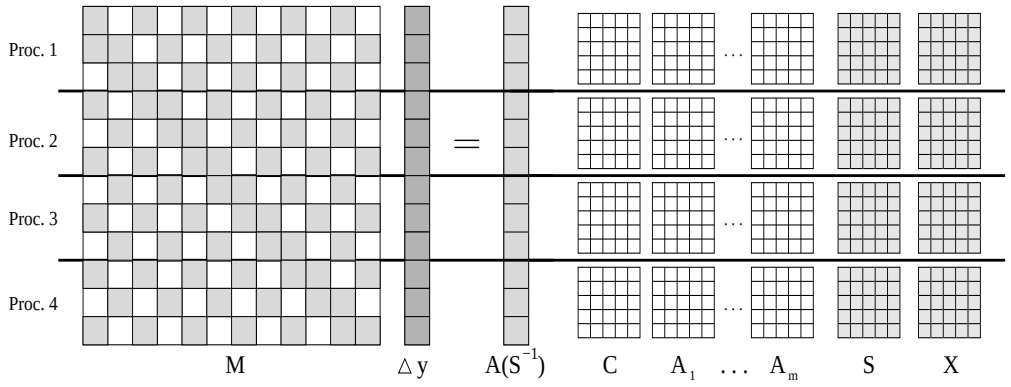


Figure 1: Distribution of data structures and computations over multiple processors

each processor computes the designated elements of one row in the matrix, sends relevant information to other processors, and assembles the data into the matrix. This procedure repeats until the entire matrix has been assembled.

Since each element of M has the form $M_{i,j} = (S^{-1}A_iS^{-1}) \bullet A_j$, all elements of $M_{i,j}$ $(j = 1, \ldots, m)$ in the ith row (or column) share a common matrix $S^{-1}A_iS^{-1}$. This matrix product can be a significant cost, especially when repeated for each A_i. In some applications, this cost exceeds the cost of computing the inner product of it with the data matrices A_i. The vectors $\mathcal{A}S^{-1}$ and $\mathcal{A}(S^{-1}RS^{-1})$ are also distributed over the processors, and ith element of these vectors is a byproduct of computing $(S^{-1}A_iS^{-1}) \bullet A_i$. PDSDP distributes this cost over all the processors, and it is very efficient and scalable when the rows are well distributed over the processor and the work to compute each row of the matrix is nearly identical.

These conditions are satisfied for the semidefinite relaxation of the maximum cut problem [15], which can be stated as follows:

$$\text{maximize } \; C \bullet X \qquad \text{subject to } \; X_{i,i} = 1, \; i = 1, \ldots, n, \qquad X \succeq 0.$$

In this problem, each constraint matrix A_i has one nonzero element. This structure significantly reduces the effort in computing M and distributes the computations among the processors evenly.

These conditions are not as well satisfied for the Lovász θ problem [18]. Given an undirected graph $G = (V, E)$, the Lovász number is the optimal objective value to the semidefinite program

$$\text{maximize } \; 1 \bullet X \qquad \text{subject to } \; X_{i,j} = X_{j,i} = 0 \;\; \forall (v_i, v_j) \notin \mathrm{E}, \;\; trace(X) = 1, \;\; X \succeq 0.$$

In this application, one constraint matrix is the identity matrix, which has n nonzeros and full rank. The other constraints have only two nonzeros. The row of M corresponding to the identity constraint matrix requires more floating point operations than do the other rows, hence reducing the scalability of computing M. While one can customize the calculation of M for each application to improve load balancing, this procedure applies in a more general context.

Another class of applications of SDP arise from control theory [13, 27]. One application in this class considers matrices $A, B, C \in \Re^{r \times r}$, possibly unsymmetric, and asks for a symmetric matrix $P \in \Re^{r \times r}$ and diagonal matrix $D \in \Re^{r \times r}$ that solves the following problem:

$$\begin{array}{ll} \text{maximize} & t \\ \text{subject to} & \left[\begin{array}{cc} -PA - A^T P - C^T D C - tI & -PB \\ -B^T P & D - tI \end{array} \right] \succeq 0, \qquad P - I \succeq 0. \end{array}$$

To put this problem into standard form (DSP), we set the left-hand side of the two inequality constraints equal to the positive semidefinite variable matrices S_1 and S_2 and let the vector $y \in \Re^{r(r+1)/2+r+1}$ represent the variables $[P_{1,1}, \ldots, P_{1,r},\ P_{2,2}, \ldots, P_{2,r},\ \ldots, P_{r,r},\ D_{1,1}, \ldots, D_{r,r},\ t]^T$. In this formulation, there are two blocks in the constraint and variable matrices such that $n_1 = 2r$ and $n_2 = r$.

In the first block of these problems, the constraint matrices corresponding to $D_{i,i}$ have $r^2 + 1$ nonzeros and a rank of $r + 1$; constraint matrices corresponding to $P_{i,j}$, $i \neq j$, have $8r - 4$ nonzeros and rank 4; constraint matrices corresponding to $P_{i,i}$ have $4r - 1$ nonzeros and rank two; and the constraint matrix corresponding to t has $2r$ nonzeros and rank $2r$. In the second block, constraint matrices corresponding to $P_{i,i}$ have one nonzero and rank one; constraint matrices corresponding to $P_{i,j}$, $i \neq j$, have two nonzeros and rank two; and the other constraint matrices have no nonzeros. Balancing constraints of such varied form is very difficult. Although constraints of each type could be distributed over each processor in an equitable manner, no attempt was made to alter the ordering used in the SDPA files that contain these examples [8].

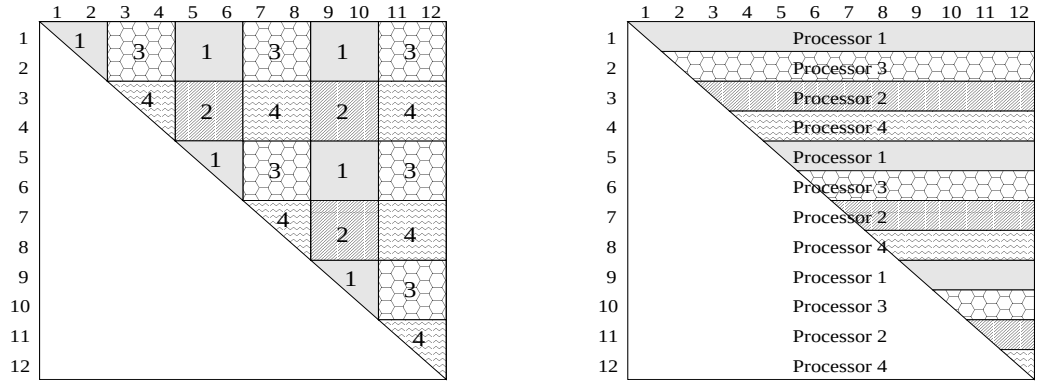
The three step directions associated with the right-hand sides of (1) are computed separately by using the conjugate gradient method, a diagonal preconditioner, and a relative convergence tolerance of 10^{-10}. A topic for continued research is whether an adaptive convergence criterion for this algorithm may be effective.

The parallel Cholesky solver is from PLAPACK [1][26]. As shown in Figure 2(b), this implementation operates on only half of the matrix and partitions it over the processors using a two-dimensional block cyclic structure. The blocking parameter in PLAPACK determines how many rows and columns are in each block. Larger block sizes can be faster and reduce the overhead of passing messages, but smaller block sizes balance the work among the processors more equitably. In Figure 2(b), the blocking parameter is 2, but PDSDP used a blocking parameter of 32 after experimenting with several choices.

When using the PLAPACK Cholesky solver, the elements of M were computed and assembled into the matrix using a pattern consistent with its block cyclic structure. For efficiency reasons discussed earlier, the elements of one row are computed by the same processor. Figure 2(a) shows a 12×12 matrix and labels the rows that each processor computed. The elements of each row are then inserted into the matrix, a process that may require passing messages. To reduce the size and complexity of these messages, these elements are computed by one of the processors on which the row resides. In Figure 2(a), rows 1, 2, 5, 6, 9, and 10 reside on processors 1 and 3. The elements of these rows will also be computed by processors 1 and 3. These six rows are assigned to the two processors in a cyclic manner to improve the load distribution. The remaining rows are assigned to processors 2 and 4 in a similar fashion. The elements $A_i \bullet S^{-1}$ and $A_i \bullet (S^{-1} R S^{-1})$ in the vectors on the right-hand side of (1) were computed by the same processor that computed the diagonal element $M_{i,i}$.

One advantage of the row cyclic structure used in PLAPACK over the noncyclic structure is that it can improve the load distribution an the processors. There is a natural tendency on the part of application developers to cluster related sets of variables and constraints. In the problem from control theory, for

example, the data files ordered the constraints corresponding to $P_{i,j}$ together and constraints corresponding to $D_{i,i}$ together. To balance the load among the processors, each group of constraints should also be well balanced among the processors. The row cyclic distribution accomplished this task automatically.



(a) Two-dimensional block cyclic distribution of Schur complement matrix M

(b) Processors compute rows of M in a row cyclic structure

Figure 2: Parallel distribution using PLAPACK on a 12×12 matrix and four processors

After (1) is solved by using one of the two parallel solvers, the step directions Δy are broadcast to the other processors in the communicator, so each processor has access to each element in vector. With these solutions, each processor calculates a suitable step-length, updates the dual matrix S, factors it, and computes an appropriate barrier parameter $\hat{\mu}$ for the next iteration. These operations are performed serially on each processor for several reasons. First, as shown in the following sections, these operations do not consume a large portion of the overall computation time. The Lovász θ and control problems, for instance, $n \ll m$ and the cost of computing S and factoring it is relatively inexpensive compared to the cost of computing M and solving (1). In the case of the maximum cut problem, sparsity in the dual matrix may drastically reduce the time needed to factor it. A second reason for not distributing these computations among the processors is that the factored form of S is needed on each processor to compute M. The cost of broadcasting a distributed factorization of S to each processor would undermine much of the benefit of parallelizing this operation. Nonetheless, the cost of these operations, relative to the other parts of the algorithm, significantly affects on the overall scalability of PDSDP.

4 Numerical Results

The performance of the PDSDP software package was tested on eleven medium and large problems from three applications. The three applications — maximum cut relaxation, Lovász θ, and control theory — were

chosen because large instances of them can be found in the standard test suites and they have different structures that affect the performance of the solver. These applications and their structure were described in the preceding section. The problems `maxG51`, `maxG55`, and `maxG60` are the three largest instances of the maximum cut problem in the SDPLIB test suite [8] and are formulated from graphs with 1000, 5000, and 7000 vertices, respectively. The problems `control10` and `control11` are the largest two control problems in the SDPLIB test suite. Two of the six instances of the Lovász θ problem are from the SDPLIB test suite, while the others were generated by Toh and Kojima [25].

The data in Tables 1 and 2 show the seconds required to solve the test problems on 1, 2, 4, 8, 16, and 32 processors. For each problem, the first row shows the seconds required to compute (1) and assemble the matrix, given a dual matrix S in factored form. The second row in Table 1 indicates the seconds spent solving (1) using the conjugate gradient method, and the second row in Table 2 indicates the time spent factoring the matrix M and solving (1) directly. The third row indicates the total seconds required by PDSDP to solve the problem. Tables 1 and 2 also list the dimensions of each problem and number of optimization iterations required to solve it. Problem instances that created memory requirements exceeding the resources of the machine are denoted by `mm`. The dual solutions are feasible and $C \bullet X - b^T y < 10^{-6} \times (|b^T y| + 1)$. Although not part of the termination criteria, $\|\mathcal{A}X - b\| \leq 10^{-8}$ in most solutions. The software has been compiled and tested on a cluster of 2.4 GHz Pentium Xeon processors. Each processor has at least 1 GB of RAM, and they are connected by a Myrinet 2000 network. The operating system is Linux, and the code was compiled using the GNU Version 2.96. In this section, the parallel efficiency on p processors is defined to be the ratio of the wall clock time using a single processor and p times the wall clock time using p processors.

The data in Tables 1 and 2 indicate that the direct solver was much more effective than the iterative solver on the control and θ problems. The time required to solve the matrices using the direct method was often an order of magnitude less that the time needed by the iterative solver. In the case of `theta6`, PDSPD with the iterative solver needed $4,944$ seconds on one processor and 303 seconds on 32 processors. Using a direct solver, PDSDP needed only 455 and 51 seconds to find a solution. The efficiency of iterative solvers often depends on good conditioning in the matrix. As noted by many others [12, 22, 25], the matrices are not well conditioned in later iterations and require better preconditioners.

Nonetheless, the performance of PDSDP with the iterative solver was competitive on the maximum cut problems. In fact, when more than one processor was used, the iterative solver was faster than the direct solver. On one processor, when only moderate precision was required, the iterative solver also performed better. Figure 3 shows the wall clock time of PDSDP using a single processor to solve `maxG51`. The vertical bars indicate the duality gap at selected iterations. If the application requires a solution whose objective is within only one of the solution, the iterative solver is a good choice. In the context of a branch-and-bound algorithm where the edges of a graph have integer values, this precision may be sufficient. On the control and θ problems, the initial iterations of PDSDP were also cheaper using the iterative solver, but the direct solver began to perform better very early in the optimization process.

The parallel efficiency of both linear solvers was influenced most significantly by the number of constraints, m, in the problem. The dense structure of M can be found in each problem, and as expected, the linear solver scaled better on larger problems than on smaller problems. On 32 processors, the parallel efficiency of the Cholesky ranged from 6.1% on `maxG11` to 58% on `theta62`, while the parallel efficiency of the iterative solver ranged from 17% on `maxG11` to 86% on `theta8`.

A second factor that affected the parallel scalability of PDSDP was the structure of the constraint

Table 1: Performance and scalability of PDSDP when computing the elements of M, solving M using the conjugate gradient method, and solving the SDP

Problem			Iter.		Number of Processors and Seconds					
Name	n	m			1	2	4	8	16	32
maxG51	1000	1000	26	Elements	19.3	12.7	6.86	3.48	2.32	1.22
				CG	29.2	14.0	12.6	8.54	6.31	5.44
				Total	53.7	31.7	24.6	16.9	13.6	11.6
maxG55	5000	5000	27	Elements	2275	1200	610	309	157	107
				CG	2568	1127	558	289	206	140
				Total	5062	2539	1378	809	572	455
maxG60	7000	7000	30	Elements	4745	2514	1280	644	327	169
				CG	6028	2271	1166	586	367	281
				Total	11270	5277	2921	1715	1203	954
control10	150	1326	73	Elements	1143	720	414	222	118	58.6
				CG	758	396	278	199	144	124
				Total	1988	1202	778	451	350	215
control11	165	1596	79	Elements	1913	1283	737	381	207	111
				CG	1402	779	440	323	232	188
				Total	3451	2194	1310	838	568	429
theta4	200	1949	19	Elements	20.5	19.9	13.1	6.86	4.36	3.26
				CG	480	189	105	87.0	60.1	50.5
				Total	503	211	120	95.7	66.3	56.2
theta42	200	5986	18	Elements	182	156	104	59.1	30.8	20.9
				CG	9443	4132	2370	1103	623	484
				Total	9638	4297	2482	1170	659	511
theta6	300	4375	21	Elements	142	131	76.4	42.4	25.5	17.4
				CG	4791	2201	943	490	376	278
				Total	4944	2341	1028	549	408	303
theta62	300	23390	19	Elements			527	319	160	94.0
				CG			14091	9766	4686	2275
				Total	mm	mm	14650	10110	4866	2386
theta8	400	7904	20	Elements	508	424	250	140	79	52
				CG	27082	9433	5363	2613	1171	985
				Total	27620	99390	5632	2770	1266	1053
theta82	400	23872	20	Elements					13861	318
				CG					25425	10792
				Total	mm	mm	mm	mm	39360	11170

Table 2: Performance and scalability of PDSDP when computing the elements of M, solving M using a Cholesky factorization, and solving the SDP

Problem			Iter.		Number of Processors and Seconds					
Name	n	m			1	2	4	8	16	32
maxG51	1000	1000	28	Elements	22.2	11.2	7.1	3.8	2.2	1.4
				Cholesky	10.9	7.4	5.4	4.8	4.6	5.2
				Total	37.7	23.1	16.9	13.1	11.2	11.6
maxG55	5000	5000	28	Elements	2562	1253	923	464	221	124
				Cholesky	623	387	219	140	87.2	65.9
				Total	3607	2011	1515	983	681	565
maxG60	7000	7000	30	Elements	5051	2499	1965	971	621	365
				Cholesky	1864	1022	563	338	214	148
				Total	7609	4237	3253	2140	1529	1226
control10	150	1326	63	Elements	1026	521	313	186	95.6	52.5
				Cholesky	47.6	31.5	20.2	17.5	16.4	16.5
				Total	1140	601	376	235	144	99.6
control11	165	1596	72	Elements	1924	973	585	339	163	94.0
				Cholesky	89.3	58.2	34.6	29.5	24.5	24.8
				Total	2139	1116	682	419	230	158
theta4	200	1949	19	Elements	20.3	10.2	8.38	5.01	3.41	2.28
				Cholesky	39.1	25.1	14.8	10.8	8.66	8.48
				Total	60.6	36.5	24.3	17.1	13.3	12.0
theta42	200	5986	19	Elements	124	62.2	63.4	35.8	21.4	12.7
				Cholesky	738	426	239	145	89.9	63.5
				Total	865	491	307	185	114	79.4
theta6	300	4375	20	Elements	121	59.4	51.5	29.2	18.3	11.3
				Cholesky	328	197	113	72.6	45.9	35.0
				Total	455	261	170	106	70.6	51.4
theta62	300	13390	20	Elements	3142	347	366	200	116	62.7
				Cholesky	7829	4058	2104	1178	664	422
				Total	11018	4425	2489	1397	798	503
theta8	400	7904	21	Elements	432	225	216	110	66.1	37.5
				Cholesky	1756	994	561	318	190	133
				Total	2204	1236	795	444	271	186
theta82	400	23872	20	Elements			1224	679	399	209
				Cholesky			11383	5767	3178	1810
				Total	mm	mm	12655	6493	3614	2066

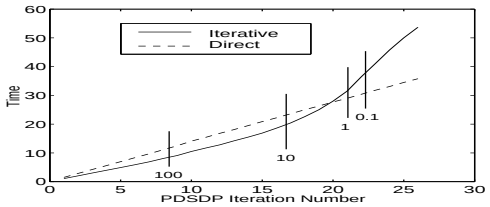


Figure 3: Wall clock time at each iteration of PDSDP to solve `maxG51` using one processor. The times for both versions are shown, and the vertical bars indicate the duality gap at selected iterations.

matrices. Since the constraint matrices in the maximum cut and θ problems have a simple structure, the elements of M in these applications are very cheap to compute. This cost was distributed over the processors, but inserting these elements into M incurred the overhead of passing messages between processors. This overhead had less impact on the control problems because the constraint matrices in this application are more dense. The increased cost of computing each element of M reduced the percentage of time spent passing messages. Using the data in Table 2, the computation and assembly of M for `control11`, which has only 1526 constraints in standard form, had a parallel efficiency of 64% on 32 processors. This efficiency was much higher than the 28% achieved on `theta4`, which is a larger SDP. The parallel efficiency on `maxG55` was also 64%, despite having three times more constraints than the `control11` and a structure that allows for excellent load-balancing. Similar efficiency numbers concerning the compuation and assembly of the PETSc matrix can be found in Table 1.

A third issue that affected the scalability of the solver was the ratio of n and m. The computation and factorization of S was computed in serial by each processor. When $n \ll m$, the cost of these operations was very low. On the control and theta problems, the factorizations used only 1% of the overall time PDSDP needed to solve the problem on one processor. On the maximum cut problem, however, this percentage increased to as much as 15% and severely limited the overall scalability of PDSDP. Using the data in Table 2, the overall parallel scalability of PDSDP `maxG55` was 20%. The parallel scalability of `theta6`, which has fewer constraints and worse load balancing, the overall parallel efficiency was 28%. The overall parallel scalability of PDSDP on `control11` was over 42% despite being much smaller in size. The overall efficiencies were even better on fewer processors and would improve for larger instances of these applications.

The parallel linear solver not only efficiently reduced the time needed to solve medium and large instances

of SDP, but it also allowed the interior-point solver to compute solutions to larger problems. The biggest memory requirement of interior-point methods for semidefinite programs usually arises from (1), which can be distributed over multiple processors. The linear system in `theta82` can be distributed and solved on four processors, but not on a single processor because of excessive memory requirements.

5 Matrix-Free Conjugate Gradient Method

The time needed to solve large instances of SDP may frustrate some application developers, but it is the high memory requirements of interior-point methods that have actually restricted the size of problems that can be solved. Distributing M across processors allows larger problems to be solved, but even this technique may not be sufficient for problems with tens of thousands or hundred of thousands of constraints. Several techniques mentioned in the introduction have been developed to address large problems.

Since iterative linear solvers require only the product of the matrix with a vector, the entire matrix M does not have to be stored. Efficient routines that evaluate the matrix-vector product can be implemented without having to recompute the matrix at each iteration. For any vector $v \in \Re^m$, $Mv = \mathcal{A}(S^{-1}WS^{-1})$, where $W = \mathcal{A}^T v$. The cost of this product is up to an order of magnitude less than the cost of computing the entire matrix M. Choi and Ye have used this approach on maximum cut problems[12] and Toh and Kojima [25] have explored similar ideas for primal-dual methods in conjunction with a reduced space preconditioner.

In this implementation, the elements of the matrix-vector product are distributed over the processors. Each processor computes the product of the vector and a set of rows. Balancing the constraints across rows to distribute the work evenly is very important here because the routine is repeated at every conjugate gradient iteration. There can be thousands of these iterations in every iteration of PDSDP, so custom routines that compute this product were written for the maximum cut and θ problems. The implementation of these routines was similar to one described in [14], but they were customized for these problems and the data structures in PDSDP. The constraints in the maximum cut problem are identical, which made the task easy. The θ problems have only one odd constraint, and the cost of computing the product of the vector with the corresponding row exceeds the cost of its product with other rows. Since the number of constraints cannot always be divided evenly over the processors, this constraint was placed on one of the processors with the fewest number of constraints to partially compensate for the extra work. The simplicity of these constraints means that the cost of computing the matrix-vector product will not be much more expensive than multiplying a previously computed matrix by a vector.

Table 3 shows the performance of PDSDP using the matrix-free conjugate gradient method on several maximum cut and theta problems. The times indicate the number of seconds required during the entire optimization process. Solutions satisfied the feasibility constraints and $C \bullet X - b^T y < 10^{-6} \times (|b^T y| + 1)$.

Several observations can be made by contrasting the data in Tables 1 and 3. First, its performance on the maximum cut problem was very competitive with the version of PDSDP that stored the matrix. The difference in times is less than a factor of two and indicates that this technique may also be effective for larger instances of the maximum cut problem. The differences in the two versions on the θ problem are much more significant. The matrix-free version was slower by about an order of magnitude on these problems. These cost of these problems is dominated by the the iterative solver. While early iterations of the PDSDP on the maximum cut problems needed only a few conjugate gradient iterations, the θ problems required many

of these iterations very early in the process. Nakata, Fujisawa, and Kojima [22] improved the performance of the conjugate gradient method in primal-dual methods using preconditioners and inexact solutions, and until these techniques can be utilized in dual step direction, its use on these problems probably will not be competitive with other solvers.

Nonetheless, the scalability of PDSDP using this technique was better than its scalability using when storing the entire matrix. On `maxG60` for instance, the DPSDP used 11270 seconds to solve the problem on one processor by storing the matrix and 20420 seconds without storing it. On 32 processors, there was less difference; PDSDP needed 954 seconds when it stored the matrix and 1250 seconds when it did not store the matrix. The linear solver dominated the overall computation time, and good parallel efficiency of the matrix-vector product improved the scalability of the PDSDP. Future work on preconditioners and inexact solutions should also apply on parallel architectures.

Table 3: Performance and scalability of PDSDP when computing a step direction using the conjugate gradient method without the matrix M

Problem			Iter.	Number of Processors and Seconds					
Name	n	m		1	2	4	8	16	32
maxG55	5000	5000	28	10530	6106	3181	1737	971	602
maxG51	1000	1000	26	123	63.8	35.6	21.5	14.9	11.3
maxG60	7000	7000	30	20420	10460	5493	3025	1763	1240
theta4	200	1949	21	979	491	281	158	105	81.1
theta42	200	5986	18	45927	27925	11951	6608	4248	2508
theta6	300	4375	21	7853	3722	1995	1127	725	488

6 Conclusion

This paper demonstrates that multiple processors can be used to efficiently solve medium and large semidefinite programs. The computations needed to compute the linear systems associated with these problems can be distributed across the processors very efficiently. Three factors that affected the scalability of the solver are the number of constraints, the complexity of the constraints, and the ratio of the number of constraints to the size of the variable matrices. An Increase in any of these three criteria usually improves the scalability of the solver. Either iterative or direct methods can be used to solve the linear system in parallel. Although both methods scale very well, the direct method usually performs better and is recommended when the entire linear system can be stored on the processors. The iterative solver is appropriate for instances where the linear system is simply too large to store on the available processors, only low or moderate precision is needed in the solution, or certain classes of SDP like the maximum cut problem. Multiple processors significantly reduced the time needed to compute solutions, and in the case of very large applications, they enabled an interior-point algorithm to find solutions that cannot be found using a single processor.

Acknowledgment

The author thanks the anonymous referees and the associate editor for their comments. Their comments led to several important improvements in the solver and a more thorough discussion of the issues involved with adapting interior-point methods to parallel architectures.

This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract W-31-109-ENG-38.

References

[1] Philip Alpatov, Gregory Baker, H. Carter Edwards, John Gunnels, Greg Morrow, James Overfelt, and Robert van de Geijn. PLAPACK: Parallel linear algebra libraries design overview. In *Proceedings of Supercomputing'97 (CD-ROM)*, San Jose, CA, November 1997. ACM SIGARCH and IEEE. University of Texas, Austin.

[2] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. Efficient management of parallelism in object oriented numerical software libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhauser Press, 1997.

[3] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. PETSc 2.0 users manual. Technical Report ANL-95/11 - Revision 2.0.22, Argonne National Laboratory, 1998.

[4] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. PETSc home page. http://www.mcs.anl.gov/petsc, 1998.

[5] S. J. Benson and Y. Ye. DSDP4 – a software package implementing the dual-scaling algorithm for semidefinite programming. Technical Report ANL/MCS-TM-255, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL, June 2002.

[6] S. J. Benson, Y. Ye, and X. Zhang. Mixed linear and semidefinite programming for combinatorial optimization. *Optimization Methods and Software*, 10:515–544, 1999.

[7] S. J. Benson, Y. Ye, and X. Zhang. Solving large scale sparse semidefinite programs for combinatorial optimization. *SIAM Journal of Optimization*, 10:443–461, 2000.

[8] B. Borchers. SDPLIB 1.0 : A collection of semidefinite programming test problems. Technical report, Faculty of Mathematics, Institute of Mining and Technology, New Mexico Tech, Socorro, NM, USA, July 1998.

[9] S. Burer and R. D. C. Monteiro. An efficient algorithm for solving the MAXCUT SDP relaxation. Manuscript, School of Industrial and Systems Engineering, Georgia Institute of Technology, Atlanta, GA, December 1998.

[10] 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.

[11] S. Burer, R. D. C. Monteiro, and Y. Zhang. Solving a class of semidefinite programs via nonlinear programming. *Mathematical Programming*, 93(1):97–122, 2002.

[12] C. Choi and Y. Ye. Solving sparse semidefinite programs using the dual-scaling algorithm with an iterative solver. Working Paper, Department of Management Science, The University of Iowa, Iowa City, IA, 2000.

[13] 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, editors, *High Performance Optimization*, pages 267–301. Kluwer Academic Press, 1999.

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

[15] M. X. Goemans and D. P. Williamson. .878-approximation for MAX CUT and MAX 2SAT. In *Proc.* 26^{th} *ACM Symp. Theor. Computing*, pages 422–431, 1994.

[16] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. The MIT Press, Cambridge, MA, 1997.

[17] C. Helmberg and F. Rendl. A spectral bundle method for semidefinite programming. *SIAM Journal of Optimization*, 10:673–696, 2000.

[18] Jon Kleinberg and Michel X. Goemans. The Lovász theta function and a semidefinite programming relaxation of vertex cover. *SIAM Journal on Discrete Mathematics*, 11(2):196–204, May 1998.

[19] Michal Kočvara and Michael Stingl. A generalized augmented lagrangian method for semidefinite programming. http://www2.am.uni-erlangen.de/ ∼kocvara/pennon/, 2002.

[20] C.-J. Lin and R. Saigal. An incomplete Cholesky factorization for dense symmetric positive definite matrices. *BIT*, 40(3):536–558, 2000.

[21] Hans D. Mittelmann. DIMACS challenge benchmarks. http://plato.la.asu.edu/dimacs.html, 2000.

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

[23] M. J. Todd. On search directions in interior-point methods for semidefinite programming. *Optimization Methods and Software*, 11:1–46, 1999.

[24] M. J. Todd. Semidefinite optimization. In *Acta Numerica*, pages 515–560. Cambridge University Press, 2001.

[25] K-C. Toh and M. Kojima. Solving some large scale semidefinite programs via the conjugate residual method. *SIAM Journal of Optimization*, 12(3):669–691, 2002.

[26] Robert A. van de Geijn. *Using PLAPACK: Parallel Linear Algebra Package*. Scientific and Engineering Computing. MIT Press, Cambridge, MA, 1997.

[27] L. Vandenberghe and S. Boyd. Semidefinite programming. *SIAM Review*, 38(1):49–95, 1996.

[28] M. Yamashita, K. Fujisawa, and M. Kojima. SDPPARA: Semidefinite Programming PARAllel version. Technical Report Research Reports on Mathematical and Computing Sciences Series B : Operations Research B-384, Tokyo Institute of Technology, 2-12-1 Oh-Okayama, Meguro-ku, Tokyo, 152-8552, Japan, Oct 2002.

[29] Y. Ye. *Interior Point Algorithms: Theory and Analysis.* Wiley-Interscience Series in Discrete Mathematics and Optimization. John Wiley & Sons, New York, 1997.