Gigaflops in Linear Programming

Irvin J. Lustig[1]   Edward Rothberg[2]

May, 1995

Abstract

This paper describes the parallelization of an "industrial strength" linear programming package. Our parallel version of CPLEX barrier, running on a Silicon Graphics Power Challenge shared-memory multiprocessor, provides dramatic performance improvements over sequential methods on a wide range of practical, realistic linear programming problems. The resulting software/hardware combination can provide sustained performance of as much as 2 Gflops.

[1]Director of Numerical Optimization, CPLEX Optimization, Inc., 930 Tahoe Boulevard, Building 802-279, Incline Village, NV 89451. Email: *irv@dizzy.cplex.com* Surface mail correspondence to this author should be sent to: 99 Braeburn Drive, Princeton, NJ 08540.

[2]Silicon Graphics, Inc., 2011 N. Shoreline Blvd., Mountain View, CA 94043. Email: *rothberg@sgi.com*

Introduction

Progress in linear programming algorithms and hardware technology has been dramatic over the past ten years. Problems that once took days to solve can now be solved in minutes. Linear programs with tens of thousands of constraints and hundreds of thousands of variables can typically now be solved in a few hours on a reasonably priced desktop workstation, using either the simplex method or a barrier method. With the advent of reasonably priced parallel computers, it is natural to investigate whether parallel computation can be used to improve the solution times of linear programs. In this paper, we describe the implementation and present computational results for an "industrial strength" linear programming solver on a parallel computer. The solver is not specialized towards a specific problem class. On some problems, the solver achieves a sustained computation rate of 2 gigaflops (i.e., 2 billion floating point operations per second).

The linear programming solver is implemented as an extension to the CPLEX[1] Barrier algorithm on the Silicon Graphics Power Challenge[2] shared-memory multiprocessor. The shared-memory programming model simplifies the task of designing and implementing parallel algorithms. The algorithms we employ are not necessarily novel nor scalable. Rather, they demonstrate the results of combining good hardware and software technology to provide a system that can solve large-scale problems in previously unachievable times.

The CPLEX Barrier algorithm is a primal-dual predictor-corrector algorithm based on logarithmic barrier functions. It is similar to, but different from, a class of interior point methods known as "infeasible interior point methods," which were introduced by Lustig [11] and shown to be equivalent to a damped Newton method by Lustig, Marsten and Shanno [14]. The predictor-corrector algorithm was introduced by Mehrotra [18]. A summary of the computational state of the art of these methods is given by Lustig, Marsten and Shanno [17]. A number of authors ([5], [8], [12], [22]) have considered the application of parallel computation to interior point methods, but usually to specially structured problems. Parallel implementations based on the dual affine interior point algorithm [1] to solve general linear programs were implemented by Bisseling, Doup and Loyens [3] on a network of 400 transputers and by Karypsis, Gupta and Kumar [7] on a 256 processor nCUBE2. These latter implementations concentrated on gaining efficiencies in the parallelization of the Cholesky factorization. The primary measure of progress in both cases was the speedup obtained. Our goal was to produce a *fast* implementation on a commercially available parallel computer that would solve any linear program, independent of the underlying problem structure. Hence, we chose specific parts of the CPLEX Barrier algorithm that were amenable to straightforward parallelization on a shared memory platform.

Section 1 gives an overview of the computational algorithm inside of CPLEX Barrier. In Section 2, the specific parts of the CPLEX Barrier algorithm that were implemented in parallel are discussed. Section 3 presents computational results for the parallel method.

[1]CPLEX is a trademark of CPLEX Optimization, Inc.
[2]Power Challenge is a trademark of Silicon Graphics, Inc.

1 Overview of the CPLEX Barrier Algorithm

The CPLEX Barrier algorithm is a primal–dual predictor–corrector algorithm for linear programming derived by applying the logarithmic barrier function to the primal linear program

$$\begin{array}{lrcl} \text{minimize} & c^T x & & \\ \text{subject to} & Ax & = & b, \\ & x & \geq & 0, \end{array} \tag{1}$$

and the corresponding dual problem

$$\begin{array}{lrcl} \text{maximize} & b^T y & & \\ \text{subject to} & A^T y + z & = & c, \\ & z & \geq & 0, \end{array} \tag{2}$$

where $A \in \Re^{m \times n}$, $b \in \Re^m$, and $c \in \Re^n$, are given. The implementation of CPLEX Barrier is similar to (but in many ways different from) the implementation of OB1, described in [17] and [16]. For ease of understanding the parallel barrier algorithm, a summary of the algorithm is presented. For clarity, the details of handling problems with upper bounds and free variables are omitted. In the statement of the algorithm, let $e = (1, 1, \ldots, 1)^T \in \Re^n$. The X and Z matrices are diagonal matrices with x^k and z^k on their diagonals, respectively.

After a problem has been instantiated via the CPLEX `loadprob()` function, a call to the function `baropt()` will cause the CPLEX Barrier algorithm to be applied to the linear program. Initialization of the algorithm consists of the following steps:

1. Apply CPLEX Presolve to the original problem.

2. Compute an ordering of the rows of A to minimize fill in the Cholesky factorization of AA^T.

3. Compute a symbolic factorization of AA^T.

4. Compute an initial iterate (x^0, y^0, z^0), by computing an estimate $\bar{x}^0 = A^T(AA^T)^{-1}b$, and adjusting all x_j that are below a threshold value to a new value at or above the threshold value. Compute an initial estimate y^0 by a cheap heuristic, set $\bar{z}^0 = c - A^T y^0$ and adjust z^0 in a method similar to that used for x^0.

On each iteration of the CPLEX Barrier algorithm, the following steps are performed

5. Compute $r_P = b - Ax^k$ and $r_D = c - z - A^T y$.

6. Check for convergence, using the norms of r_P and r_D and the inner product $x^T z$.

7. Form $A\Theta A^T$, where $\Theta = XZ^{-1}$ is a diagonal matrix.

8. Compute the Cholesky factor $A\Theta A^T = LDL^T$, where L is lower triangular and D is diagonal.

9. Compute the predictor directions $\Delta_p = (\Delta\hat{x}, \Delta\hat{y}, \Delta\hat{z})$, as
   (a) $\Delta\hat{y} = (A\Theta A^T)^{-1}[r_P + A\Theta(r_D - XZe)]$.
   (b) $\Delta\hat{x} = \Theta[A^T\Delta\hat{y} + XZe - r_D]$.
   (c) $\Delta\hat{z} = -Ze - \Theta^{-1}\Delta\hat{x}$.

10. Do a ratio test to compute $\hat{\alpha}_P$ and $\hat{\alpha}_D$, by computing

$$\hat{\alpha}_P = \min_{1\le j\le n}\left\{\frac{x_j}{-\Delta\hat{x}_j} : \Delta\hat{x}_j < 0\right\} \tag{3}$$

$$\hat{\alpha}_D = \min_{1\le j\le n}\left\{\frac{z_j}{-\Delta\hat{z}_j} : \Delta\hat{z}_j < 0\right\} \tag{4}$$

11. Compute the barrier "parameter" μ based on (x^k, y^k, z^k), Δ_p, $\hat{\alpha}_P$ and $\hat{\alpha}_D$.

12. Compute the search direction $\Delta = (\Delta x, \Delta y, \Delta z)$, as
   (a) $\Delta\hat{y} = (A\Theta A^T)^{-1}[r_P + A\Theta(r_D + \mu e - XZe - \Delta X\Delta Ze)]$, where ΔX and ΔZ are diagonal matrices with the components of $\Delta\hat{x}$ and $\Delta\hat{z}$ on the diagonals, respectively.
   (b) $\Delta\hat{x} = \Theta[A^T\Delta\hat{y} - \mu e + XZe + \Delta X\Delta Ze - r_D]$.
   (c) $\Delta z = \mu X^{-1}e - Ze - \Theta^{-1}\Delta X\Delta Ze - \Theta^{-1}\Delta\hat{x}$.

13. Do a ratio test to compute α_P and α_D, by computing

$$\alpha_P = \rho \min_{1\le j\le n}\left\{\frac{x_j}{-\Delta x_j} : \Delta x_j < 0\right\}, \tag{5}$$

$$\alpha_D = \rho \min_{1\le j\le n}\left\{\frac{z_j}{-\Delta z_j} : \Delta z_j < 0\right\}, \tag{6}$$

   where $\rho = 0.99995$.

14. Update the iterate as

$$\begin{aligned} x^{k+1} &= x^k + \alpha_P\Delta x^k, \\ y^{k+1} &= y^k + \alpha_D\Delta y^k, \\ z^{k+1} &= z^k + \alpha_D\Delta z^k. \end{aligned}$$

In the initialization phase, the ordering is computed via an implementation of the multiple minimum degree procedure as described by Liu [9]. This step can be computationally expensive for large scale problems. The symbolic factorization computes the locations of the nonzeros in the Cholesky factorization of AA^T, since it will be the same for each iteration of the barrier algorithm. This procedure also identifies the supernodes in the factorization (a supernode is a set of columns in the sparse factor matrix L with identical non-zero structure).

In order to increase the performance of the Cholesky factorization, a supernode amalgamation procedure similar to that described by Ashcraft and Grimes [2] was implemented. This procedure introduces extra nonzeros into the symbolic factorization in order to increase the number and width of the supernodes. The computational cost of the amalgamation procedure is minimal.

On each iteration, the computationally intensive procedures are:

1. Forming the matrix $A\Theta A^T$.
2. Computing the Cholesky factorization $A\Theta A^T = LDL^T$.
3. Solving for $\Delta\hat{y}$ and Δy, using the LDL^T factorization.
4. The two ratio tests, one for the predictor and one for the corrector.
5. The computation of the matrix products Av and A^Tv for different vectors v. Note that Av and A^Tv are each computed three times per barrier iteration.

Other operations are inner products and vector additions that contribute a small amount compared to the above operations.

Note that when the initial iterate (x^0, y^0, z^0) is computed, steps (1) and (2) above are done, followed by one solve that uses LDL^T and two matrix–vector products (one each of Av and A^Tv).

We chose to implement the 5 procedures listed above in parallel, since our experience on large scale problems indicated that these procedures would benefit the most from parallelization. The specifics of the algorithms used are discussed in the next section.

2 Parallelization of CPLEX Barrier

To determine how parallelization could benefit the CPLEX Barrier algorithm, we profiled the single processor performance on a number of problems. As expected, we found that dependent upon problem structure, either the computation of the Cholesky factorization $A\Theta A^T = LDL^T$ or the formation of the matrix $A\Theta A^T$ are the most time consuming operations. For problems with a large number of columns compared to rows (for example, 1000 rows and 500000 columns), the computation of $A\Theta A^T$ is dominant. For problems with highly dense Cholesky factorizations, the factorization time is dominant. However, in most instances, each of the other procedures listed in the previous section also consumes significant fractions of the computation time. Hence, it is beneficial to parallelize them all, rather than to concentrate on only the Cholesky factorization, as has been done by other researchers. This section gives high-level overviews of the approaches we take to parallelize the various procedures.

Before describing the actual computation, let us first describe the environment in which the parallel method is developed and executed. The SGI Power Challenge contains up to 18 90 MHz R8000 processors. The R8000 processor is rated at 396 SPECfp92 and 360 Mflops peak. An 18 processor Power Challenge system therefore has a peak speed of 6.5 Gflops.

The Power Challenge is a fully symmetric multiprocessor, and provides a shared-memory programming model. The processors communicate with memory and with each other through a 1.2 GByte per second shared bus. To reduce the load on the shared bus, each processor has a 4 MByte cache. We express the parallelism in CPLEX Barrier using the SGI Power C programming language [21]. The three parallel C constructs we use are *#pragma parallel* (a parallel region), *#pragma pfor* (a parallel for loop), and *#pragma critical* (a critical section).

2.1 Cholesky factorization

Our primary goals when performing the Cholesky factorization of $A\Theta A^T$ are:

- Make effective use of cache memory. This improves the performance of the individual processors and avoids saturation of the shared bus.
- Share data at a coarse grain. This reduces the costs associated with data sharing (contention in critical sections, false sharing, etc.).
- Make no assumptions about the non-zero structure of $A\Theta A^T$. Recall that our approach does not assume anything about the structure of the problem.

The method we use is a panel left-looking factorization algorithm [20]. The method performs the factorization one panel at a time, where a panel is a subset of a supernode. By loading an entire panel into cache, the method utilizes the cache quite effectively. Since panels can be quite large (up to 64 columns in our implementation, depending on the size and structure of the problem), all data sharing happens at a coarse grain. Since the panel left-looking method dynamically distributes panels among processors, the method is effective across a variety of different matrix structures, from highly dense to extremely sparse.

2.2 Forward elimination and back substitution

Parallel solution of sparse triangular systems is a difficult problem. Little floating-point work is performed in each triangular solve, and the computation is highly recursive, so there is very little parallelism to be exploited. Unfortunately, the triangular solves consume a large enough fraction of total runtime that they would become a performance bottleneck if they were not parallelized. Our goals for the parallel triangular solve are more modest than those for the factorization:

- Keep parallelization overhead to a minimum. It is quite easy to create a parallel triangular solve method that spends more time looking for opportunities to perform work in parallel than it does performing the actual work.
- Share data at as coarse of a grain as possible. The triangular solve computation performs so little computation that data sharing costs must be kept to a minimum.

We use a panel approach to the triangular solve [19], combined with extensive preprocessing to identify those locations in the triangular solve where parallelism can be exploited. The panel approach keeps the grain of the solve reasonably coarse. By performing preprocessing before the actual solve, we avoid spending time looking for parallelism whose grain is too fine to effectively exploit.

2.3 Forming $A\Theta A^T$

In CPLEX Barrier, the lower triangle of $A\Theta A^T$ is formed one column at a time. Note that column j of $A\Theta A^T$ is given by

$$\sum_{k=1}^{n} A_{jk}\Theta_k A_{*k},$$

where A_{*k} is the k-th column of A and Θ_k is the k-th diagonal of Θ. If a row representation of A is maintained, a list of columns k for which $A_{jk} \neq 0$ is easily constructed. Note that the columns in $A\Theta A^T$ do not depend on one another, so the computation is entirely parallel.

To perform the computation, one work vector w of length m is maintained per processor. The values in column j are computed by adding $A_{jk}\Theta_k A_{*k}$ into w for each column k where $A_{jk} \neq 0$. The values in w are then gathered directly into the storage for column j in the factor matrix L. To balance the load in the parallel computation, the columns of $A\Theta A^T$ are dynamically distributed among processors (using `gss` loop scheduling in Power C [21]).

2.4 Computing Av and A^Tv

In each iteration of the barrier algorithm, the matrix vector products Av and A^Tv are computed three times, for different values of v. This computation can be expensive for large scale problems with large numbers of nonzeros. To compute A^Tv in parallel, no extra storage is required. This is because the values v^TA_j can independently be computed for each column j. Since the amount of computation for each column j depends on the density of the column, we chose to use `gss` scheduling here as well.

To compute Av, a similar strategy is used. However, to compute Av in parallel, a row-wise data structure is used, and the products A_iv are computed for each row i in parallel. Once again, `gss` scheduling was used to attempt to balance the computational load across processors.

2.5 The ratio test

For problems with a large number of variables, it was found that the ratio test could be time consuming. This is because the cost of a divide instruction greatly exceeds that of other floating point instructions. For each loop in the ratio test procedure, each processor was handed an equal number of columns to process. That processor found the minimum across it's particular set of columns. This was followed by a single processor finding the global minimum across the processors.

Table 1: Problem sizes

Problem	Rows	Columns	Nonzeros in A	Nonzeros in L
pilot	1441	3652	43167	215744
pilot87	2030	4883	73152	462546
dfl001	6071	12230	35632	1351576
AIRLINE	2256	11684	47181	1588036
A15	6330	22275	94950	9482335
GISMONDI	18262	23211	136324	45694837
FLEET12	21971	68136	273539	4899060
PDS-20	33874	105728	230200	6545930
SS2203	18321	47002	155101	5307183

Table 2: Runtimes for selected NETLIB problems (seconds)

	Runtime			
Problem	P=1	P=2	P=4	P=8
pilot	16.4	11.7	8.5	7.5
pilot87	54.4	36.7	25.8	20.1
dfl001	241.7	155.1	101.0	69.0

3 Computational Results

The computational results presented in this section were obtained on a Silicon Graphics Power Challenge with 16 R8000 processors and 4 GBytes of memory. All runs were made on a dedicated machine to minimize variability in the timings. We have chosen a few problems from the standard NETLIB test set assembled by Gay [6], as well as problems submitted to us by customers. The times for the CPLEX Presolve, the multiple minimum degree ordering, and other preprocessing operations are included in all reported runtimes. Table 1 gives the problem statistics for the problems reported here. Both the number of nonzeros in the constraint matrix A, as well as the number of symbolic nonzeros in the Cholesky factor L are reported.

3.1 Results for Selected NETLIB problems

All but a few of the problems in the NETLIB suite are extremely small by today's standards. The only ones that are potentially interesting from the point of view of reducing runtimes are `pilot`, `pilot87`, and `dfl001`. Table 2 illustrates results on these three problems.

For historical perspective, it is worth examining the performance of different codes on different architectures on the NETLIB problem `pilot`. Table 3 summarizes these results, which have been obtained from the sources indicated. Sources indicated as "New" are times

Table 3: Historical results for `pilot`

Machine	Code	Algorithm	Time (secs)	Source
DEC VAXStation II	MINOS 5.1	Pri. Simplex	79939.9	[10], Tab. 3
SGI 4D/70	MINOS 5.3	Pri. Simplex	4024.8	[14], Tab. 4
SGI 4D/70	OB1 1.0	Pri.-Du. Barr.	6742.6	[14], Tab. 2
CRAY Y/MP	OB1 1.0	Pri.-Du. Barr.	58.6	[13], Tab. 1
SGI 4D/70	OB1 3.0	Pri.-Du. Barr.	3813.5	[16], Tab. 3
SGI 4D/70	OB1 3.0	Pri.-Du. Pred-Corr	1784.4	[16], Tab. 3
Sun Sparcstation 2	CPLEX 2.0	Pri. Simplex	768.3	[4] Tab. V
IBM RS/6000 Model 530	OB1 5.0	Pri.-Du. Pred-Corr	146.2	[15] Tab. 1
SGI R4000 (100 Mhz)	CPLEX 2.1	Pri. Simplex	385.0	New
SGI R4000 (100 Mhz)	CPLEX 2.1	Du. Simplex	206.9	New
SGI R4000 (100 Mhz)	CPLEX 3.0	Pri. Simplex	323.0	New
SGI R4000 (100 Mhz)	CPLEX 3.0	Du. Simplex	177.1	New
SGI R4000 (100 Mhz)	CPLEX 3.0	Barrier	180.3	New
SGI Pwr Chal. R8000	CPLEX 3.0	Du. Simplex	40.0	New
SGI Pwr Chal. R8000	CPLEX 3.0	Barrier	16.4	New
SGI Pwr Chal. (2 proc.)	Par. CPLEX 3.0	Barrier	11.7	New
SGI Pwr Chal. (4 proc.)	Par. CPLEX 3.0	Barrier	8.5	New
SGI Pwr Chal. (8 proc.)	Par. CPLEX 3.0	Barrier	7.5	New

obtained by runs by the authors. All times, except those for CPLEX Parallel Barrier, are CPU times. The dramatic improvement in solution time leads one to ask the following question: If Dantzig had this type of computational power when the `pilot` models were developed and analyzed in the late 1970's and early 1980's, what kind of further analysis of the model would have been possible?

3.2 Results for Other Problems

CPLEX Optimization, Inc. has collected over 1000 problems from customers for benchmarking. We have selected a few of these proprietary problems to illustrate the computational performance of CPLEX Barrier. To protect the interest of these customers, only problem sizes for these problems are given. The names of the problems have been changed as well, unless the problem corresponds to a problem reported in another paper. Note that the problem PDS-20 is part of the Kennington test set on NETLIB, but that we include it here because of its large size. Table 4 show parallel runtimes and sustained performance rates for these problems on 1, 4, 8, and 16 processors. Note that sustained performance is computed by dividing the total number of floating-point operations performed in solving the problem by the total runtime, which includes the time to perform pre-processing. The

Table 4: Results on customer problems (runtimes in seconds and performance in sustained Mflops)

	P=1		P=4		P=8		P=16	
	Time	Perf.	Time	Perf.	Time	Perf.	Time	Perf.
AIRLINE	234	240	75	745	49	1135	36	1577
A15	2656	234	735	846	466	1334	336	1853
GISMONDI	12297	224	3286	838	1887	1460	1341	2055
FLEET12	1648	195	598	537	411	781	341	942
PDS-20	2805	204	905	631	605	944	487	1172
SS2203	1613	212	540	634	359	954	294	1162

Table 5: Runtime breakdown for problem PILOT (in seconds).

	Pre-process	Form $A\Theta A^T$	Factor	Tri. solve	Ratio test	Mat.-vect. mult.	Total
P=1	2.7	2.3	8.6	0.8	0.2	0.9	16.4
P=2	2.7	1.3	5.2	0.9	0.2	0.4	11.7
P=4	2.7	0.7	3.1	0.7	0.1	0.3	8.5
P=8	2.7	0.4	2.2	0.9	0.1	0.3	7.5

results demonstrate that extremely high computation rates are possible using our parallel linear programming method. Parallel CPLEX Barrier on the Power Challenge system sustains more than 200 Mflops on 1 processor, 500-800 Mflops on 4 processors, and in excess of 2 Gflops on 16 processors.

3.3 Analysis of Results

To illustrate where time is being spent in parallel CPLEX Barrier, Tables 5 and 6 and 7 show how runtimes are distributed among the various phases of the computation for three problems (on 1, 2, 4, 8, and 16 processors). As can be observed from these runtime breakdowns, the factorization, the formation of $A\Theta A^T$, and the ratio test speed up quite nicely. The triangular solve and matrix-vector multiply exhibit moderate speedups. In cases where parallel speedups are limited, the primary culprit is the sequential preprocessing. The majority of this preprocessing time is spent performing the multiple minimum degree ordering. Unfortunately, research into the parallelization of ordering algorithms is still in its infancy. Progress in this area will certainly lead to direct improvements in parallel CPLEX Barrier.

An obvious question is how the results described here for a shared-memory multiprocessor would compare against results for a distributed-memory machine. The main advantage of a shared-memory machine is ease of programming and ease of use. By making limited mod-

Table 6: Runtime breakdown for problem A15 (in seconds).

	Pre-process	Form $A\Theta A^T$	Factor	Tri. solve	Ratio test	Mat.-vect. mult.	Total
P=1	98	28	2455	68	1	4	2656
P=2	98	14	1144	54	1	2	1316
P=4	98	8	591	34	0	1	735
P=8	98	6	332	26	0	1	466
P=16	98	5	204	22	0	1	336

Table 7: Runtime breakdown for problem FLEET12 (in seconds).

	Pre-process	Form $A\Theta A^T$	Factor	Tri. solve	Ratio test	Mat.-vect. mult.	Total
P=1	117	48	1340	87	9	20	1648
P=2	117	25	709	65	6	11	962
P=4	117	15	387	36	3	8	598
P=8	117	11	216	29	2	6	411
P=16	117	10	149	29	1	5	341

ifications to a commercial linear programming package, we were able to produce a parallel library that gives excellent performance while providing users with the same interface they would see in a sequential version. In a distributed-memory environment, the modifications would have been much more extensive, since all relevant data structures (A, $A\Theta A^T$, etc.) would have to be broken up among the memories of the processors. One potential advantage of a distributed-memory machine is increased scalability. While a distributed-memory machine might be able to use more processors than a shared-memory machine like the Power Challenge, our data shows that the marginal benefit of adding processors beyond sixteen, even for very large problems, drops off dramatically. Too much of the computation is either serial or contains limited parallelism to effectively utilize a larger number of processors.

Our shared-memory implementation of CPLEX barrier provides a tremendous leap forward in linear programming performance. At this point, it will be up to modelers to provide real models that can take true advantage of this power.

References

[1] I. Adler, N. K. Karmarkar, M. G. C. Resende, and G. Veiga, *An implementation of Karmarkar's algorithm for linear programming*, Mathematical Programming, 44 (1989), pp. 297–335. Errata in *Mathematical Programming*, 50:415, 1991.

[2] C. Ashcraft and R. Grimes, *The influence of relaxed supernode partitions on the multifrontal method*, ACM Transactions on Mathematical Software, 15 (1989), pp. 291–309.

[3] R. Bisseling, T. Doup, and L. Loyens, *A parallel interior point algorithm for linear programming on a network of transputers*, Annals of Operations Research, 43 (1993), pp. 51–86.

[4] R. E. Bixby, *Implementing the simplex method: The initial basis*, ORSA Journal on Computing, 4 (1992), pp. 267–284.

[5] J. Eckstein, R. Qi, V. I. Ragulin, and S. A. Zenios, *Data–parallel implementations of dense linear programming algorithms*, Technical Report TMC–230, Thinking Machines Corporation, Cambridge, MA 02142, USA, May 1992. Also available as Decision Sciences Department Report 92–05–06, The Wharton School, University of Pennsylvania, Philadelphia, PA 19101, USA.

[6] D. M. Gay, *Electronic mail distribution of linear programming test problems*, Mathematical Programming Society COAL Newsletter, (1988).

[7] G. Karypis, A. Gupta, and V. Kumar, *A parallel formulation of interior point algorithms*, in Proceedings Supercomputing '94, IEEE Computer Society Press, 1994, pp. 204–213.

[8] R. Levkovitz, J. A. Andersen, and G. Mitra, *The interior point method for LP on parallel computers*, in System Modelling and Optimization (Proceedings of the 15th IFIP Conference held at the University of Zurich 1991), P. Kall, ed., vol. 180 of Lecture Notes in Control and Information Sciences, Springer Verlag, Berlin, Germany, 1992, pp. 241–250.

[9] J. Liu, *Modification of the minimum-degree algorithm by multiple elimination*, ACM Transactions on Mathematical Software, 11 (1985), pp. 141–153.

[10] I. J. Lustig, *An analysis of an available set of linear programming test problems*, Computers and Operations Research, 16 (1989), pp. 173–184.

[11] ——, *Feasibility issues in a primal–dual interior point method for linear programming*, Mathematical Programming, 49 (1990/91), pp. 145–162.

[12] I. J. Lustig and G. Li, *An implementation of a parallel primal-dual interior point method for block-structured linear programs*, Computational Optimization and Applications, 1 (1992), pp. 141–161.

[13] I. J. Lustig, R. E. Marsten, and D. F. Shanno, *The primal–dual interior point method on the Cray supercomputer*, in Large–Scale Numerical Optimization, Papers from the Workshop held at Cornell University, Ithaca, NY, USA, October 1989, T. F. Coleman and Y. Li, eds., vol. 46 of SIAM Proceedings in Applied Mathematics, Society

of Industrial and Applied Mathematics (SIAM), Philadelphia, PA, USA, 1990, pp. 70–80.

[14] ——, *Computational experience with a primal–dual interior point method for linear programming*, Linear Algebra and Its Applications, 152 (1991), pp. 191–222.

[15] ——, *Interior method vs. simplex method : Beyond* netlib, COAL Newsletter, 19 (1991), pp. 41–44.

[16] ——, *On implementing Mehrotra's predictor–corrector interior point method for linear programming*, SIAM Journal on Optimization, 2 (1992), pp. 435–449.

[17] ——, *Interior point methods for linear programming: Computational state of the art*, ORSA Journal on Computing, 6 (1994), pp. 1–14.

[18] S. Mehrotra, *On the implementation of a primal–dual interior point method*, SIAM Journal on Optimization, 2 (1992), pp. 575–601.

[19] E. Rothberg, *Alternatives for solving sparse triangular systems of equations*, Parallel Computing, to appear (1995).

[20] E. Rothberg, A. Gupta, E. Ng, and B. Peyton, *Parallel sparse cholesky factorization algorithms for shared-memory multiprocessor systems*, in Proceedings of the Seventh IMACS International Conference on Computer Methods for Partial Differential Equations, 1992.

[21] Silicon Graphics, Inc., *Power C User's Guide.*

[22] A. Vannelli, *A parallel implementation of an interior point method for linear programming*, in Parallel Processing for Scientific Computing (Houston Texas, USA), 1991, Philadelphia, USA, 1992, SIAM, pp. 163–167.