Fast Cholesky factorization for interior point methods of linear programming[1]

CSABA MÉSZÁROS

Computer and Automation Research Institute,
Hungarian Academy of Sciences
Department of Operations Research and Decision Systems
H-1518 Budapest, P.O. Box 63. Hungary
E-mail: h4225mes@huella.bitnet

Abstract

Every iteration of an interior point method of large scale linear programming requires computing at least one orthogonal projection. In the practice, Cholesky decomposition seems to be most efficient and sufficiently stable method. We studied the *'column oriented'* or *'left looking'* sparse variant of the Cholesky decomposition, which is a very popular method in the large scale optimization. We show some techniques such as using supernodes and loop unrolling for improving the speed of computation. We show numerical results on a wide variety of large scale, real-life linear programming problems.

Introduction

The logarithmic barrier methods are very powerful tools for solving large scale linear programming problems. The different computational characteristics to the competitive simplex-based algorithms make efficient implementation techniques possible. For the general barrier algorithm we consider the following linear programming problem :

$$
\text{(P)} \qquad \begin{aligned} \text{minimize} \quad & c^T x \\ \text{subject to} \quad & Ax = b \\ & x \geq 0, \end{aligned}
$$

[1] This research was supported by Hungarian Research Fund OTKA 2587

Typeset by $\mathcal{A}\mathcal{M}\mathcal{S}$-TEX where $x = (x_1, ..., x_n)^T$, $A \in \mathcal{R}^{m \times n}$, $c = (c_1, ..., c_n)^T$ and $b = (b_1, ..., b_m)^T$. The barrier algorithms generate a sequence of primal and/or dual solutions, which converge to the optimum. Readers interested in the full development of the logarithmic barrier algorithms are referred to the survey by Gonzaga [1]. The fundamental task of the logarithmic barrier methods is the computation of the following weighted least squares problems :

$$\min_u \|D(\alpha - A^T u)\|_2, \tag{1.1}$$

where D is $n \times n$ diagonal matrix, $\alpha = (\alpha_1, ...\alpha_n)$, and $u = (u_1, ..., u_m)$. The augmented system corresponding to (1.1) is

$$\begin{bmatrix} D^{-2} & A^T \\ A & 0 \end{bmatrix} \begin{bmatrix} v \\ u \end{bmatrix} = \begin{bmatrix} \alpha \\ 0 \end{bmatrix}. \tag{1.2}$$

In every iteration the diagonal matrix D and the vector α on the right hand side of (1.2) is changed, corresponding to the rule of the applied algorithm. The solutions u and v are used for the determination of the next iterate. In the practice the sparse Cholesky decomposition of AD^2A^T is used for solving (1.2). As the first step, a "good" pivot order is determined to minimize the number of the nonzeroes in the Cholesky factors of AD^2A^T [2]. When applying the algorithm the sparsity pattern of AD^2A^T and the sparsity pattern of its Cholesky factors are not changed, thus we compute only the numerical factorization into the given sparsity pattern in every iteration. For computing the Cholesky-factorization, there are two different families of algorithms, the left looking and the right looking methods. Lustig, Marsten and Shanno studied the interaction of algorithms and architectures [3],[4], one conclusion of theirs is that the left looking method is superior to the right looking one on SUN Sparkstastion machines. In this paper we extend the well-know left looking algorithm by using supernodes and loop unrolling techniques. We show comparisons with the original algorithm on a wide variety of large scale, real-life linear programming problems on our SUN Sparc-10 workstation.

The left looking algorithm

In this section we give a brief description of the left looking sparse Cholesky factorization. To this end, we will denote matrix $M = AD^2A^T$ and consider the factorization of M as $M = L\Delta L^T$. We needed to store only the nonzero entries of matrix L, so our formulae for computing the column j of L (denoted by L_j) and Δ_{jj} are:

$$\Delta_{jj} = m_{jj} - \sum_{k=1}^{j-1} l_{jk}^2,$$

$$L_j = \frac{1}{\Delta_{jj}} \left(M_j - \sum_{k=1, l_{jk} \neq 0}^{j-1} (\Delta_{kk} l_{jk}) L_k \right).$$

In the sparsity exploiting implementation we store the nonzeroes in the packed form. Let $c(j)$ denote the number of the nonzeroes in the column j of L, then we store the nonzero values and their row indices in the two parallel working arrays $nz(1,j), nz(2,j), ..., nz(c(j),j)$ and $indx(1,j), indx(2,j), ..., indx(c(j),j)$. Because M is symmetric, and the sparsity pattern of the off lower diagonal part of M is a subset of the sparsity pattern of L, at the begin of the factorization we store the matrix M in the place of L in the $nz(.,.)$ and $indx(.,.)$ arrays. For computing a column of L, we use a double precision work vector $wr(.)$ to accumulate the transformations of the previous columns, and the integer work vector $p(.)$ to hold the actual row pointers of the completed columns of L. The pseudo code of this algorithm can be described as follows:

```
do j = 1, m
    p(j) = 1
    Δ(j) = M(j, j)
    wr(indx(i, j)) = nz(i, j)   for i = 1, ..., c(j)
    do k = 1, j − 1
        if (c(k) ≥ p(k)) and (indx(p(k), k) = j) then
            Δ(j) = Δ(j) − Δ(k) ∗ nz(p(k), k) ∗ nz(p(k), k)
            do i = p(k) + 1, c(k)
                wr(indx(i, k)) = wr(indx(i, k)) − Δ(k) ∗ nz(i, k) ∗ nz(p(k), k)
            enddo
            p(k) = p(k) + 1
        endif
    enddo
    nz(i, j) = wr(indx(i, j))   for i = 1, ..., c(j)
enddo
```

Steps 2-14 are the inner loop for computing the columns L_j. In step 4. we extract the target column in the working array, step 6 checks the condition $L_{jk} \neq 0$. Steps 7-9 are the transformations with the previous columns which fulfil the condition. In step 14. we compress the working vector to the final position of L_j. An alternative way for computing sparse Cholesky factorization is the so-called right looking algorithm, which was extensively and very successfully studied by Rothberg and Gupta [5] from our point of view. The first improvement of this method is the exploitation of the dense window. In the practice the last columns of L are often totally dense. This last partition of the columns of L (the so-called dense window) can be handled separately, dense matrix operations can be used, i.e. no indirect addressing is needed during the transformations. The details of this idea can be found in the book of Duff, Erisman and Reid [6].

The supernodal algorithm

Originally, the idea of the supernodal factorization is coming from parallel and vector computation areas [7]. Their efficiency on the shared-memory multiprocessors is discussed by Esmond and Peyton [8], the exploitation of the cache memory on high-performance workstations is studied by Rothberg and Gupta by right looking factorization [6]. We examine efficiency by the left looking factorization, which is a superior method to the right looking method, on our SUN Spark-10 machine [3]. We can consider the supernodes as the generalization of the dense window : The supernode is a partition of the continuous columns, which share the same nonzero structure. In other words, apart from the free rows a supernode consists of any, totally dense columns. In our experiments we use another type of the supernode too. By this supernode, the nonzero structure of the columns are identical under the diagonal block only.

Type 1 supernode Type 2 supernode

$$\begin{bmatrix} * & & \\ * & * & \\ * & * & * \\ * & * & * \\ & & \\ * & * & * \\ * & * & * \end{bmatrix} \qquad \begin{bmatrix} * & & \\ & * & \\ & & * \\ * & * & * \\ & & \\ * & * & * \\ * & * & * \end{bmatrix}$$

By the transformations with the supernodes likewise to the dense window method we can use dense matrix-vector transformations and can save indirect addressing and thereby memory references. For the computation we use one new integer working vector. The working vector *snhead*(.) holds the last column of its supernode partition for each column, or 0 for the non-supernodal columns. Our new pseudo code is as follows:

```
 1. do j = 1, m
 2.     p(j) = 1
 3.     Δ(j) = M(j, j)
 4.     wr(indx(i, j)) = nz(i, j)   for i = 1, ..., c(j)
 5.     k = 1
 6.     if (c(k) ≥ p(k)) and (indx(p(k), k) = j) then
 7.         if(snhead(k) ≠ 0) then
 8.             l = min(snhead(k), j − 1)
 9.             nz(i − p(k), j) = wr(ind(i, k))   for i = p(k) + 1, .., c(k)
10.             call dense(k, l, j, p, c, nz, Δ)
11.             wr(ind(i, k)) = nz(i − p(k), j)   for i = p(k) + 1, .., c(k)
12.             p(i) = p(i) + 1   for i = k, .., l
13.             k = snhead(k)
14.         else
15.             Δ(j) = Δ(j) − Δ(k) ∗ nz(p(k), k) ∗ nz(p(k), k)
16.             do i = p(k) + 1, c(k)
17.                 nz(indx(i, k)) = nz(indx(i, k)) − Δ(k) ∗ nz(i, k) ∗ nz(p(k), k)
18.             enddo
19.             p(k) = p(k) + 1
20.         endif
21.     endif
22.     k = k + 1
23.     if (k ≤ j − 1) goto 6.
24.     nz(i, j) = wr(indx(i, j))   for i = 1, ..., c(j)
25. enddo
```

The new steps 7-13 are the supernodal transformations. We use the subroutine *dense* to compute dense matrix-vector transformations, the columns $k, ..., l$ are applied to column j. The dense computation can be further exploited by loop unrolling technique. In the typical inner loop of the factorization we add a multiple of a column into another (e.g. step 17). Let a be the target vector, b the source vector and c the multiplicator. If we assume that c is kept in a register, then the steps of the computer by one transformation of the $a \leftarrow a + cb$ can be written as follows:

1. reading $a(i)$ from the memory
2. reading $b(i)$ from the memory
3. computing $a(i) + cb(i)$
4. storing the result in the memory

In this case we do one arithmetical operation (step 3) and three memory references (steps 1,2 and 4). During factorization we do multiple column modifications on a single column, and can unroll the loop over the column transformations. Let a be the target vector, b, c, d, e, f and g the source vectors, and $h(1), ..., h(6)$ scalar multiplicators. By the 6-way loop unrolling we compute the following transformation:

$$a \leftarrow a + h(1)b + h(2)c + h(3)d + h(4)e + h(5)f + h(6)g.$$

To do this transformation eight memory references and six arithmetical transformations were needed in the inner loop of the transformation, which is by ten-memory-reference less than the equivalent transformations in the original way.

Computational results

For studying the efficiency of the supernodal and loop unrolling techniques, we used NETLIB test problems [9], and any other large-scale real life LP problems. From the NETLIB test set we omit the small problems. All problems were solved without pre-processing, on a SUN SPARKstation with 64MB of memory.

In the first experiment we compare the time for computing one factorization required by the standard left looking factorization, dense window technique, supernodal without loop unrolling and with 2-, 4-, and 6-way loop unrollings. Table 1 collects the results. Its first two columns contain the name of the problems, and the expected column length of the Cholesky factors. The execution times are given in seconds in columns 3-8. We denote the times required by the standard left looking method by *T(ll)*, the execution times by the dense window techniques by *T(dw)*. The columns denoted by *T(sn1)*, *T(sn2)*, *T(sn4)* and *T(sn6)* contain the times required by the simple supernodal, 2- 4- and 6-way loop unrolling supernodal methods. We use 6 test problems with different characteristics. Problems *aircraft* and *fit2p* have very sparse factorisation, the expected number of the nonzeroes in each column is under 4. Problems *80bau3b* and *25fv47* are 'average' sparse problems, and the *rat7a* and *dfl001* are typically dense ones. By the extreme sparse problems, the discussed techniques having a very little influence to the factorisation times. By the 'average' problems the dense window method unequivocally superior to the standard left looking method, but the overhead of the handling the supernodes recompensed only with the 2-way loop unrolling. The 4-way loop unrolling gives a better execution time, but the further effect of the 6-way loop unrolling is not measurable. By the dense problems, the superiority of the simple supernodal method to the dense window method is unambiguous, and the computation time monotonically decreases with the degree of the loop unrolling.

The efficiency of the 6-way loop unrolling supernodal method to the standard method is compared on a larger set of the test problems. The first four column of table 2 contain the name of the problems, the numbers of the rows, columns and nonzero elements in the original constraint matrix. Column 4. holds the number of the different supernode partitions, column 6. the numbers of the supernodal columns. The last two columns contain the time required by the standard left looking Cholesky factorization and by supernodal loop unrolling technique in seconds. The first part of table 2 contains the results on the NETLIB problems, the second part the other ones.

Table 1. Comparison of the different methods

Name	Col.Len.	T(ll)	T(dw)	T(sn1)	T(sn2)	T(sn4)	T(sn6)
aircraft	1.8	0.13	0.14	0.14	0.13	0.13	0.12
fit2p	4.2	0.36	0.36	0.37	0.37	0.36	0.35
80bau3b	20.5	0.63	0.58	0.66	0.60	0.54	0.53
25fv47	29.7	0.56	0.52	0.55	0.48	0.42	0.41
rat7a	317.9	266.0	252.8	213.6	178.2	164.0	151.1
dfl001	421.4	461.2	370.7	232.0	202.3	185.6	171.8

Conclusion

The supernodal method with loop unrolling techniques is a very powerful method in the framework of the left looking factorization. It often works more than two times faster than the standard algorithm, especially in the computationally 'hard', dense problems. The experiments show little disadvantageous behavior in the case only if the supernodes are very close, i.e. if the ratio of the number of the supernodal columns and that of the supernode partitions is small. This ratio can be controlled by the creation of the supernode partitions for suitably big supernodes only.

Table 2. Comparison of the standard and the 6-way loop unrolled supernodal method

Name	Rows	Cols.	Nonz.	S.nodes	S.cols.	T(ll)	T(sn6)
25fv47	821	1571	10400	92	493	0.56	0.41
80bau3b	2262	9301	20413	105	457	0.63	0.53
bnl2	2324	3489	13999	136	732	2.56	1.63
cycle	1903	2857	20720	214	1190	1.26	0.98
d2q06c	2171	5167	32417	152	1098	6.96	4.27
degen3	1503	1818	24646	144	727	3.21	2.63
dfl001	6071	12230	35632	69	2046	461.2	171.8
fit2d	25	10500	129018	1	23	0.59	0.65
fit2p	3000	13525	50284	1	24	0.36	0.35
ganges	1309	1681	6912	66	401	0.36	0.14
greenbea	2392	5302	30715	149	803	1.33	1.04
greenbeb	2392	5290	30676	150	805	1.26	0.99
maros	846	1408	9576	85	481	0.30	0.25
nesm	662	2748	13078	51	323	0.29	0.22
pilot	1441	3449	41092	76	882	9.63	6.03
pilot87	2030	4663	70682	101	1217	33.94	21.22
pilot-ja	940	1677	11821	64	482	1.47	0.99
pilot-we	722	2711	8862	36	250	0.19	0.15
pilotnov	975	1968	12186	70	490	1.31	0.90
scfxm3	990	1371	7777	120	530	0.09	0.08
sctap2	1090	1880	6714	100	323	0.15	0.11
sctap3	1480	2480	8874	159	493	0.17	0.14
ship08l	778	4283	12802	28	144	0.08	0.09
ship08s	778	2387	7114	44	208	0.04	0.05
ship12l	1151	5427	16170	58	202	0.10	0.12
ship12s	1151	2763	8178	61	298	0.05	0.06
stocfor3	16675	15695	64875	2105	6649	1.39	1.44
truss	1000	8806	27836	69	684	0.76	0.61
woodw	1098	8405	37474	115	632	0.81	0.66
aircraft	3754	7517	20267	1	5	0.13	0.12
complex	1023	1398	46453	11	396	4.62	2.90
cr42	905	1513	6614	5	18	0.06	0.06
ken11	14694	21349	49058	363	779	1.46	1.53
l30	2701	15380	51169	149	1926	5.02	3.06
model10	4400	15067	148620	241	2775	21.24	13.10
progas	1650	1300	8072	268	1109	0.39	0.31
rat1	3136	9408	88267	70	2785	115.6	67.00
rat5	3136	9408	137413	93	2973	103.7	59.89
rat7a	3136	9408	268908	19	2973	266.0	151.1
slptsk	2861	3347	72465	19	274	9.35	8.62
south31	18425	35223	93673	26	64	0.57	0.58

References

1. C.G. Gonzaga, *Path following methods for linear programming*, SIAM Rewiew **34 (2)** (1992), 167-227.
2. A. George and J.W.H Liu, *Computer Solution of Large Scale Positive Definite Systems*, Prentice Hall, Englewood Cliffs, NJ, 1981.
3. I.J. Lustig, R.E. Marsten and D.F. Shanno, *The interaction of algorithms and architectures for interior point method*, Advances in Optimization and Parallel Computing (P.M Pardalos, eds.), North-Holland, Amsterdam, The Netherlands, 1992, pp. 190-205.
4. I.J. Lustig, R.E. Marsten and D.F. Shanno, *Interior Point Methods for Linear Programming: Computational State of the Art*, ORSA Journal on Computing **6 (2)** (1994), 1-14.
5. E. Rothberg and A. Gupta, *Efficient Sparse Matrix Factorization on High-Performance Workstations-Exploiting the Memory Hiearchy*, ACM Transactions on Mathematical Software **17 (3)** (1991), 313-334.
6. I.S. Duff, A.M. Erisman and J.K. Reid, *Direct Methods for Sparse Matrices*, Calderon Press, Oxford, England, 1967.
7. C.C. Ashcraft, R.G. Grimes, J.G. Lewis, B.W. Peyton and H.D. Simon, *Recent progress in sparse matrix methods for large linear systems*, Int. J. Supercomput. Appl. **1 (4)** (1987), 10-30.
8. N.G. Esmond and B.W. Peyton, *A Supernodal Cholesky Factorization Algorithm for Shared-Memory Multiprocessors*, SIAM Journal on Scientific Computing **14 (2)** (1993), 761-769.
9. D.M. Gay, *Electronic mail distribution of linear programming test problems*, Mathematical Programming Society COAL Newsletter (1988).