ECOLE POLYTECHNIQUE

CENTRE DE MATHÉMATIQUES APPLIQUÉES
UMR CNRS 7641

91128 PALAISEAU CEDEX (FRANCE). Tél: 01 69 33 41 50. Fax: 01 69 33 30 11

http://www.cmap.polytechnique.fr/

A Cholesky algorithm for some complex symmetric systems.

Natacha Béreux

R.I. N^0 515 *October 2003*

A Cholesky algorithm for some complex symmetric systems.

Natacha Béreux

28th October 2003

Abstract

Complex symmetric systems do not in general admit a Cholesky factorization without pivoting, as would be the case for hermitian systems. Nevertheless, for some complex symmetric systems, as those coming from the discretization of boundary integral formulations, pivoting can be avoided. We present a Cholesky factorization algorithm for such complex symmetric systems. We propose a LAPACK-style implementation and compare this algorithm to the symmetric LDL^T factorization with Bunch-Kauffman pivoting (LAPACK `ZSYSV` solver), which is the standard method for complex symmetric systems. Numerical experimental comparisons of this approach with `ZSYSV` solver show considerable time performance improvement, with the same accuracy. Finally, a parallel SCALAPACK-style implementation is given and numerically evaluated.

1 Introduction

Our aim in this note is to develop, implement and evaluate a LAPACK-style direct solver for the linear system of equations:

$$A\,X = B, \tag{1}$$

where A is a symmetric ($A^T = A$) matrix of size $n \times n$ with complex entries, with the property that no pivoting is required during its factorization.
Complex symmetric systems arise naturally in many applications, for instance when boundary integral equations are discretized, and are thus of a wide importance. But, they do not in general admit simple methods like Cholesky factorization without pivoting, as would be the case for A complex hermitian.

The standard method for solving (1) can be found in LAPACK [1]. It relies on a symmetric factorization of (a permuted version of) A:

$$P\,A \;=\; L\,D\,L^T$$

where L is complex, lower triangular and D symmetric block-diagonal with blocks of size 1 or 2, and P the permutation matrix is obtained by Bunch Kauffman diagonal pivoting method.
Nevertheless, recent advances have been achieved to exhibit classes of complex matrices for which it is known to be safe not to pivot. N. Higham identified in [4], the class of complex symmetric positive definite (CSPD) matrices, which arise in calculations with Padé approximations to the exponential. We refer to [8], [6], [5], [3] for additional references. Though some BIE-based applications still use a pivoting (Bunch-Kauffman) algorithm [9], it is also empirically known (though not yet fully justified) that a Cholesky decomposition can be used, when the matrix comes from the discretization of boundary integral equation formulations.
We propose here a modified version of Cholesky algorithm for complex symmetric matrices (when no pivoting is required), provide a LAPACK-style implementation, and evaluate it.
In the first section, we recall the traditional Cholesky algorithm, for the symmetric factorization of a real, symmetric, definite positive matrix, as implemented in LAPACK. We then show how to formally extend the algorithm to complex symmetric matrices, and deduce an implementation of this algorithm based on LAPACK routines for real symmetric matrices.
We compare both the accuracy and the efficiency of the standard direct method for complex symmetric systems, based on a symmetric factorization with Bunch-Kauffman diagonal pivoting, with the method based on a complex symmetric Cholesky factorization.
These numerical experiments use matrices arising from the numerical analysis of piezo-electric surface acoustic wave (SAW) components, by a coupled integral equation/ finite element formulation [7]. We show a very significant increase of timing performances, with the same accuracy as the standard method.
We also propose a parallel, ScaLAPACK-style implementation of our algorithm, deduced from ScaLAPACK routines for real positive definite matrices, in the same way as we previously deduced a serial implementation based on LAPACK routines, and we give some numerical experiments.

2 Cholesky Algorithm for a real, symmetric definite positive matrix

This section is devoted to a description of Cholesky factorization of A, a real symmetric positive definite matrix (see *e.g* [3] p.142).

2.1 Basic version of Cholesky algorithm

There exists several variants of the algorithm, all requiring $n^3/3$ operations and all deduced from the equality

$$A = L\,L^T, \tag{2}$$

but featuring different orders of loops.
We recall here the gaxpy version implemented by LAPACK routine `spotf2` (see [3] p.144 for a discussion of the outer product version).

Algorithm 2.1 (Gaxpy version of Cholesky algorithm). *This algorithm computes a lower triangular $L \in \mathbb{R}^{n\times n}$ such that*

$$A = L\,L^T,$$

and L overwrites the lower triangle of A:
for $j = 1 : n$
! Compute A(j,j)
1. $A_{j,j} \leftarrow A_{j,j} - A_{1:j-1,j}\,A_{1:j-1,j}^T$
! Test for non-positive-definiteness and compute L(j,j)
if $A_{j,j} <= 0$ ***then***
stop
else
2. $A_{j,j} \leftarrow \sqrt{A_{j,j}}$
endif
! Compute elements j+1:n of column j
if $(j < n)$ ***then***
3. $A_{j+1:n,j} \leftarrow A_{j+1:n,j} - A_{j+1:n,1:j-1}\,A_{j,1:j-1}^T$
4. $A_{j+1:n,j} \leftarrow A_{j+1:n,j}/A_{j,j}$
endif
end

Remark 1. The name "gaxpy" version comes from the richness of the algorithm in scaled "gaxpy" operations ([3] p.5). A "gaxpy" operation is a

generalized saxpy, that is a matrix-vector operation of the form:

$$y \leftarrow Ax + y$$

2.2 Block version of Cholesky algorithm

A block version of Cholesky algorithm operates on on blocks, or submatrices of the original matrix and is therefore rich in matrix-matrix operations. Since these operations (level 3 BLAS) are faster, block algorithms are usually more efficient.
The original matrix A is partitioned in blocks of size nb. The block version of algorithm 2.1, as implemented in LAPACK routine `spotrf` reads:

Algorithm 2.2 (Block version of Cholesky algorithm).
for $j = 1 : (n/nb + 1)$
s=(j-1).nb+1 ! start of block to factorize
e= min(j.nb,n) ! end of block to factorize
u=e+1 ! start position for update
! Update the current diagonal block
if $j > 1$ ***then***
1. $A_{s:e,s:e} \leftarrow A_{s:e,s:e} - A_{s:e,1:s-1}\, A^T_{s:e,1:s-1}$
endif
! Factorize the current diagonal block
2. $A_{s:e,s:e} \leftarrow G$, *with* $GG^T = A_{s:e,s:e}$
if $1 < j <= n/nb$ ***then***
! Update the current block column
3. $A_{u:n,s:e} \leftarrow A_{u:n,s:e} - A_{u:n,1:s-1}\, A^T_{s:e,1:s-1}$
endif
if $j <= n/nb$ ***then***
4. $A_{u:n,s:e} \leftarrow A_{u:n,s:e}/A^T_{s:e,s:e}$
endif
end

Remark 2. At step 2, the current block is factorized by algorithm 2.1.

This algorithm is also called block *left-looking* version of Cholesky algorithm.
In this version A is factorized one block column at a time. For each j, step 3 is a rank $s-1$ update that accesses the previously factorized block columns, hence the name.

3 Cholesky algorithm for complex symmetric matrices

In this section, we consider a symmetric matrix A with complex entries. We suppose that this matrix has the property that pivoting can be avoided during its factorization. A Cholesky method can then be used to factorize A in a stable way.

3.1 Derivation of the generalized Cholesky algorithms:

The complex extension of algorithm (2.2) can be described in the following manner:

Algorithm 3.1 (Complex block version of Cholesky algorithm).
for $j = 1 : (n/nb + 1)$
s=(j-1).nb+1 ! start of block to factorize
e= min(j.nb,n) ! end of block to factorize
u=e+1 ! start position for update
! Update the current diagonal block
if $j > 1$ ***then***
1. $A_{s:e,s:e} \leftarrow A_{s:e,s:e} - A_{s:e,1:s-1}\, A^T_{s:e,1:s-1}$
endif
! Factorize the current diagonal block
2. $A_{s:e,s:e} \leftarrow G$, *with* $GG^T = A_{s:e,s:e}$
if $1 < j <= n/nb$ ***then***
! Update the current block column
3. $A_{u:n,s:e} \leftarrow A_{u:n,s:e} - A_{u:n,1:s-1}\, A^T_{s:e,1:s-1}$
endif
if $j <= n/nb$ ***then***
4. $A_{u:n,s:e} \leftarrow A_{u:n,s:e}/A^T_{s:e,s:e}$
endif
end

Note that the algorithm remains essentially the same as (2.2). The notation A^T means "transpose", as in the real case and not conjugate transpose (as would be the case for a hermitian matrix).
Step 2 is achieved through a call to the level-2 following algorithm

Algorithm 3.2 (Complex gaxpy version of Cholesky algorithm).

$pivmax = abs(\sqrt{A_{11}})$
for $j = 1 : n$
! Update A(j,j)
1. $A_{j:j,j} \leftarrow A_{j,j} - A_{j,1:j-1}\, A_{j,1:j-1}^T$
! Compute L(j,j)
2. $A_{j,j} \leftarrow \sqrt{A_{j,j}}$
! Test
3. ***if*** $abs(A_{j,j})/pivmax <= tol$ ***then***
stop
else
$pivmax = max(pivmax, abs(A_{j,j})$
! Update A(j+1:n,j)
4. $A_{j+1:n,j} \leftarrow A_{j+1:n,j} - A_{j+1:n,1:j-1}\, A_{j+1,1:j-1}^T$
5. $A_{j+1:n,j} \leftarrow A_{j+1:n,j}/A_{j,j}$
endif end

Remark 3. The test step of algorithm 2.1 is modified: the square root is complex and exists for every value of $A(j, j)$. One tests instead that the diagonal term is not too small (with a suitable value of tolerance parameter *tol*).

3.2 Implementation details

The implementation heavily relies on LAPACK implementation of algorithms 2.1 and 2.2 for real matrices, in routines `DPOTF2` `DPOTRF` and `DPOTRS` and `DPOSV`. The complex equivalent of these routines are:

- `zlltf2` (based on `DPOTF2`): computes the Cholesky factorization of a complex symmetric matrix A. The factorization has the form $A = U^T\, U$ or $A = L\, L^T$, where U is an upper triangular matrix and L is lower triangular, depending of the value of `UPLO` argument. This is the unblocked version of the algorithm, calling Level 2 BLAS.

- `zlltf`(based on `DPOTRF`): is the blocked version of `zlltf2`, calling Level 3 BLAS.

- `zllts` (based on `DPOTRS`): solves a system of linear equations $A\, X = B$ with a symmetric matrix A using the Cholesky factorization $A = U^T\, U$ or $A = L\, L^T$ computed by `zlltf`.

- `zlltsv` (based on `DPOSV`): is the associated driver routine.

4 Performance results

We report in this section experimental results comparing the double precision driver routine `ZSYSV` from LAPACK and our Cholesky-based solver `zlltsv`. The test were carried out on a bi-processor Intel Pentium 4 XEON, with 2 CPU at 2.00GHz, and a cache size of 512 KB, under Linux. In each case, we employed the Intel-supplied BLAS in the Math Kernel Library (5.2 Service Pack 1) and Intel fortran compiler ifc (version 7.1.015).
We generated 5 complex symmetric matrices of size 1810, 3221, 7659, 9847, 10941 with the BEM-based code `transd` ([7]).

4.1 Accuracy results

A random exact solution x_{exact} was generated, then multiplied by the matrix A to generate a right hand side b. Then, the linear system:

$$A\,x \;=\; b$$

was solved using Bunch Kauffman factorization or Cholesky factorization, yielding two computed solution respectively denoted by x_{BK} and x_{llt}.
We compute the distance of both computed solutions to the exact solution in 2-norm, *i.e.*

$$||x_{exact} - x_{llt}||_2 \;=\; \left(\sum_{i=1}^{n} |x_{exact}(i) - x_{llt}(i)|^2 \right)^{1/2}$$

and

$$||x_{exact} - x_{BK}||_2 \;=\; \left(\sum_{i=1}^{n} |x_{exact}(i) - x_{BK}(i)|^2 \right)^{1/2}$$

and normalize it by $||x_{exact}||_2$ to get the forward relative error in 2-norm for both methods (see [5] ch.1, or [1] ch.4 for definitions of backward and forward errors). The results are shown in Table 1.
We computed the size of the residual $A\,x - b$ for both methods, in 2-norm, to get the absolute backward error.
We normalize the residual by $||b||_2$, and obtain this way a relative backward error, shown in Table 2. Tables 2 and 1 show that the same accuracy can be expected for both methods, when applied to a matrix coming from a boundary integral equation problem.

N	ZLLTSV solver no pivoting	ZSYSV solver Bunch-Kauffman pivoting
1810	4.217D-013	2.728D-013
3221	1.684D-013	1.830D-013
7659	1.759D-012	2.031D-012
9847	1.354D-012	1.182D-012
10941	7.187D-013	1.533D-012

Table 1: Relative forward error, in 2-norm.

N	ZLLTSV solver no pivoting	ZSYSV solver Bunch-Kauffman pivoting
1810	2.079D-013	1.649D-015
3221	3.876D-014	1.080D-015
7659	1.673D-014	1.704D-015
9847	4.347D-015	1.006D-014
10941	1.856D-014	4.073D-015

Table 2: Relative backward error, in 2-norm.

4.2 Timing results

The runtimes presented in Figures 1 and 2 confirm that when pivoting can be avoided, the solver is faster. The computation time is divided by 2 on our examples.

5 Parallel, Scalapack-style implementation

SCALAPACK library [2] provides a set of routines to perform the Cholesky factorization of a distributed real symmetric dense matrix : PDPOTF2, PDPOTRF, PDPOTRS and PDPOSV. These routines are the parallel equivalents of DPOTF2, DPOTRF, DPOTRS and DPOSV.
We deduced from these SCALAPACK routines a parallel implementation of Algorithm 3.1, consisting of routines pzlltf2, pzlltf, pzllts and pzlltsv, and tested it on matrices of size $N = 10941$ to .
Numerical experiments were carried out on a Linux cluster, with ten nodes, using a high-performance Scalable Coherent Interface (SCI) network. Each node is a bi-processor Pentium 4 Xeon at 2GHz.

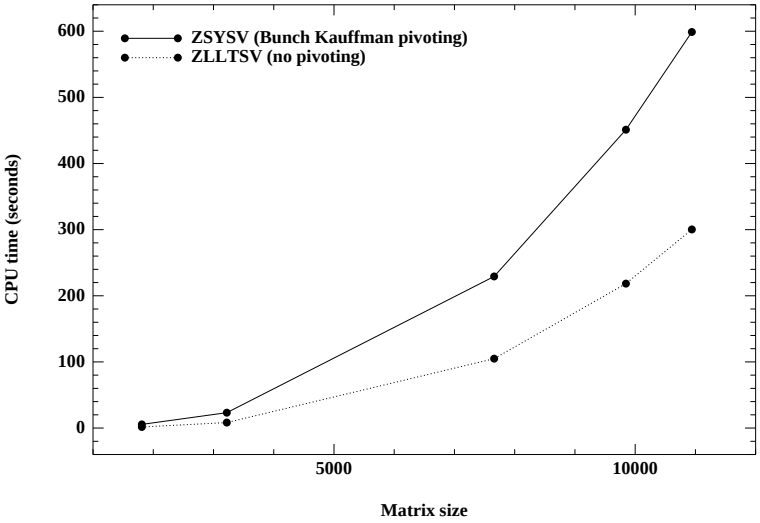

Figure 1: Timing result

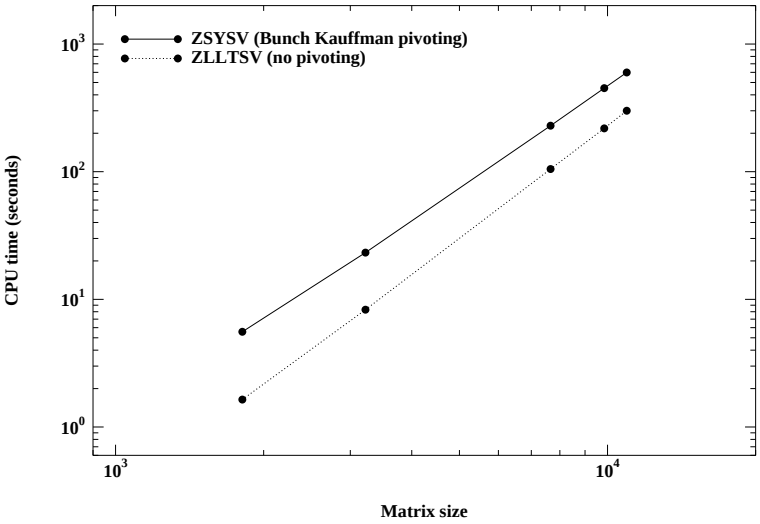

Figure 2: Timing result, logarithmic scale

Number of nodes	Maximum runtime	Minimum runtime
4	132 (50)	106 (222)
6	98 (50)	80 (216)
8	80 (50)	69 (178)
9	70 (50)	58 (230)

Table 3: Maximum and minimum runtimes with respect to blocksize

5.1 Accuracy

We compared the accuracy of `pzlltsv`, with that of the serial solver `zlltsv`. We computed the solution of

$$A\,x \;=\; b$$

by `zlltsv` and by `pzlltsv`, yielding two computed solutions x_{seq} and x_{par}. The relative 2-norm of the difference is:

$$\frac{||x_{seq} - x_{par}||_2}{||x_{seq}||_2} \;=\; 9.17e-14.$$

for a problem of size $N = 10941$. So the accuracy of the parallel solver is the same as the accuracy of the sequential solver.

5.2 Timing results

5.2.1 Blocksize

The global matrix is partitioned in small blocks of size *blocksize* and dealt out to the processes, following the block-cyclic distribution scheme. The performance of the solver depends on this parameter: Table 3 shows min and max runtimes (in seconds) and corresponding blocksizes (between parentheses), for different number of nodes, for a problem of size 10941: In all the following measurements, the blocksize is equal to 220.

5.2.2 Speedup and efficiency

The *speed-up* on p nodes, S_p is defined as the ratio

$$S_p \;=\; \frac{T_1}{T_p}, \tag{3}$$

where T_1 is the time required to execute the algorithm on one node, and T_p the time required to execute the algorithm on p nodes.

Number of nodes	Runtime(seconds)	Speedup	Efficiency
$1 = 1 \times 1$	424.66	1	1
$2 = 2 \times 1$	193.37	2.19	1.09
$4 = 2 \times 2$	106.88	3.97	0.99
$6 = 2 \times 3$	79.41	5.35	0.89
$8 = 2 \times 4$	69.20	6.13	0.77
$9 = 3 \times 3$	58.22	7.29	0.81
$10 = 5 \times 2$	61.16	6.94	0.69

Table 4: Speedup and efficiency for $N = 10941$

Remark 4. There exists another definition for the speedup (see [2] p. 95):

$$S_p = \frac{T_1^{seq}}{T_p}, \tag{4}$$

where T_1^{seq} is the time required by the best possible algorithm on one node. This definition takes into account the slowness of parallel algorithm on one node, due to additional operations that may not be needed on a single node (communications, synchronizations ...). With this definition, we would compare the time on p nodes with the time required by the serial routine `zlltsv`, that is $T^{seq} = 300.29$ seconds for the matrix of size 10941.
We prefer to use the first definition (3) since it normalizes the speedup to 1 on one node.

The *efficiency* is simply the speed-up divided by the number of nodes:

$$E_p = \frac{S_p}{p} \tag{5}$$

Table 4 shows the runtimes (in seconds) for different number of nodes, and the corresponding speedup and efficiency. The best efficiency is obtained for 2 nodes.

5.2.3 Timing results for different matriz sizes

We report on Figure 3 timing results, with respect to the number of nodes used, for different matrix sizes. On Figure 4, we represented the time required to solve problems of different sizes on 10 nodes.

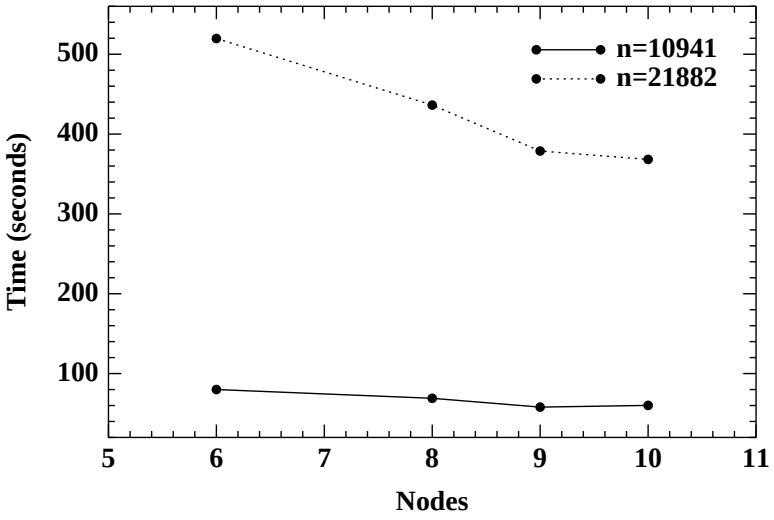

Figure 3: Timing results with respect to the number of nodes

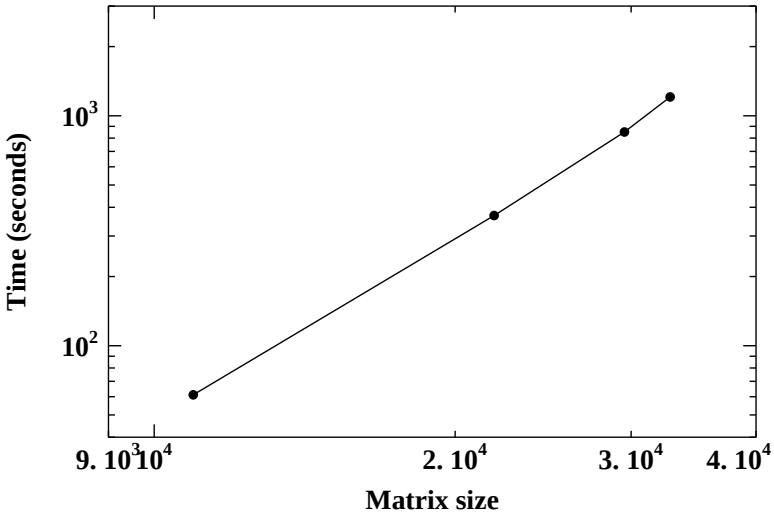

Figure 4: Timing results with respect to the matrix size, on 10 nodes, logarithmic scale

6 Conclusion

Numerical experiments suggest that Cholesky factorization should be used whenever possible, in particular for linear systems coming from the discretization of boundary integral equations. It namely achieves the same accuracy as the traditional LDL^T factorization with Bunch Kauffman diagonal pivoting, and is faster.
Thanks to the availability of LAPACK and ScaLAPACK sources, and thanks to their ease of use, we were able to obtain a serial and a parallel implementation of Algorithm 3.1 with a small amount of work. Moreover, this implementation benefits from the performances of LAPACK algorithms on vector machines and of ScaLAPACK design on distributed machines.

References

[1] E. Anderson, Z. Bai, C.Bischof, J. Demmel, J. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK Users' Guide*. SIAM, third edition, 1999.

[2] L. Blackford, J. Choi, A. Ceary, E. D'Azevedo, J. Demmel, I. Dhilon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. Whaley. *ScaLAPACK Users' Guide*. SIAM, 1997.

[3] G. H. Golub and C. F. V. Loan. *Matrix computations*. John Hopkins University Press, third edition, 1996.

[4] N. J.Higham. Factorizing complex symmetric matrices with positive definite real and imaginary parts. *Mathematics of computation*, 67(224):1591–1599, 1998.

[5] N. J.Higham. *Accuracy and stability of Numerical Algorithms*. SIAM, second edition, 2002.

[6] R. Mathias. Matrices with positive definite Hermitian part: Inequalities and linear systems. *SIAM Journal on Matrix Analysis and Applications*, 13(2):640–654, 1992.

[7] J. Ribbe. *On the coupling of integral equations and finite elements/Fourier modes for the simulation of piezoelectric surface acoustic wave components*. Thèse de doctorat, École Polytechnique, France, 2002.

[8] S. M. Serbin. On factoring a class of complex symmetric matrices without pivoting. *Mathematics of Computation*, 35(152):1231–1234, 1980.

[9] P. E. Strazdins. A dense complex symmetric indefinite solver for the Fujitsu AP3000. Technical Report TR-CS-99-01, Canberra 0200 ACT, Australia, 1999.