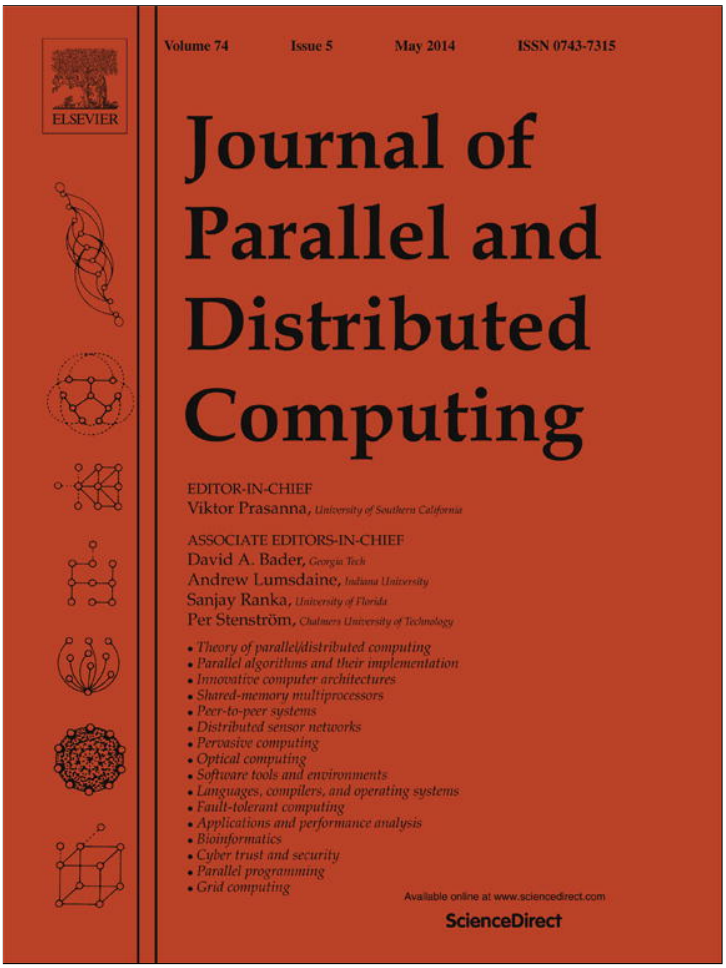
Alonso, P.; Dolz Zaragozá, MF.; Vidal Maciá, AM. (2014). Block pivoting implementation of a symmetric Toeplitz solver. Journal of Parallel and Distributed Computing. 74(5):2392-2399. doi:10.1016/j.jpdc.2014.02.003.



J. Parallel Distrib. Comput. 74 (2014) 2392–2399

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

Block pivoting implementation of a symmetric Toeplitz solver☆

Pedro Alonso [a,*], Manuel F. Dolz [b], Antonio M. Vidal [a]

[a] Dpto. de Sistemas Informáticos y Computación, Universitat Politècnica de València, Cno. Vera s/n, 46022 Valencia, Spain
[b] Department of Informatics, University of Hamburg, Bundesstraße 45a, 20146 Hamburg, Germany

HIGHLIGHTS

- We improve the solution of symmetric Toeplitz linear systems in multicore systems.
- We transform the Toeplitz matrix into a Cauchy-like one to obtain some benefits.
- The problem is partitioned into two half-sized independent problems.
- We use partial local pivoting to improve the accuracy of the solution.
- We propose a special scheme to store data in memory that accelerates the algorithm.

ARTICLE INFO

Article history:
Received 7 December 2011
Received in revised form
18 January 2014
Accepted 6 February 2014
Available online 15 February 2014

Keywords:
Symmetric Toeplitz matrices
Linear systems
Pivoting
Displacement structure
Multicores

ABSTRACT

Toeplitz matrices are characterized by a special structure that can be exploited in order to obtain fast linear system solvers. These solvers are difficult to parallelize due to their low computational cost and their closely coupled data operations. We propose to transform the Toeplitz system matrix into a Cauchy-like matrix since the latter can be divided into two independent matrices of half the size of the system matrix and each one of these smaller arising matrices can be factorized efficiently in multicore computers. We use OpenMP and store data in memory by blocks in consecutive positions yielding a simple and efficient algorithm. In addition, by exploiting the fact that diagonal pivoting does not destroy the special structure of Cauchy-like matrices, we introduce a local diagonal pivoting technique which improves the accuracy of the solution and the stability of the algorithm.

1. Introduction

The linear system of equations that we work with in this paper is defined as

$$Tx = b, \tag{1}$$

where $T \in \mathbb{R}^{n\times n}$ is a real symmetric Toeplitz matrix, and $b, x \in \mathbb{R}^n$ are the independent right-hand side and the solution vectors, respectively. The elements of a symmetric Toeplitz matrix are $T_{i,j} = t_{|i-j|}$, with $t^T = \begin{pmatrix} t_0 & t_1 & \dots & t_{n-1} \end{pmatrix}^T$.

Toeplitz matrices appear in many areas of science and engineering. Signal Processing is one of these fields where Toeplitz matrices can be found in topics like filtering, linear prediction, etc. Signal processing interpretations of Toeplitz matrices can be found, e.g., in [28,22,30]. In particular, Toeplitz matrices appear in the solution of inverse filtering problems and equalization of multichannel acoustic systems where matrices can be very large, according to the filter length [16,17]. In some cases, the number of filters are also large, proportional to the number of sources (loudspeakers) like it is the case in [26,8], where 96 loudspeakers are used to position a sound signal in a 3D space of a room. The least squares problems which arise in that problems conduce to the solution of linear systems with Toeplitz matrices, sometimes non-symmetric or symmetric indefinite. Efficient solvers, other than the traditional Levinson-type ones [18], represent a good alternative for these cases.

Many (*fast*) algorithms that exploit the special structure of Toeplitz matrices have been developed over recent years [23]. These algorithms reduce the $O(n^3)$ flops required to solve a dense linear system by at least one order of magnitude lower if any type of structure is taken into account. In general, fast algorithms can be classified into Levinson-type algorithms (which perform an

☆ This work was partially supported by the Spanish Ministerio de Ciencia e Innovación (Project TIN2008-06570-C04-02 and TEC2009-13741), Vicerrectorado de Investigación de la Universidad Politécnica de Valencia through PAID-05-10 (ref. 2705), and Generalitat Valenciana through project PROMETEO/2009/2013.

* Corresponding author.
E-mail addresses: palonso@dsic.upv.es (P. Alonso), dolzm@icc.uji.es (M.F. Dolz), avidal@dsic.upv.es (A.M. Vidal).

http://dx.doi.org/10.1016/j.jpdc.2014.02.003
0743-7315/

implicit computation of the inverse of the system matrix), and Schur-type algorithms (which perform a factorization of the system matrix) [15]. Levinson-type algorithms have a memory complexity of $O(n)$ data, but they are very rich in dot products with closely coupled operations. Therefore, a good speed-up in a parallel implementation is difficult to achieve. Schur-type algorithms usually need $O(n^2)$ data in memory and are easier to parallelize; however, they also have closely coupled operations [1].

Fast ($O(n^2)$) and superfast ($O(n \log^2 n)$) algorithms can be unstable or might return inaccurate solutions for indefinite Toeplitz matrices [10]. The work in [14] proposed moving the symmetric Toeplitz matrix to a symmetric Cauchy-like matrix to solve the linear system. This translation allows pivoting techniques to be incorporated in the algorithms since pivoting does not destroy the structure of Cauchy-like matrices, which is contrary to what happens with Toeplitz matrices.

The idea of using Cauchy-like matrices has also been used to develop parallel algorithms. The Cauchy-like matrix obtained in this way has great sparsity that can be exploited to reduce the linear system to two smaller independent linear systems of half the size of the original one, reducing both time and memory to solve the problem. For example, this was used in [33] to propose a shared memory algorithm. It was also used in [9] leading to a multilevel parallel MPI-OpenMP algorithm. In that paper, the Toeplitz matrix was transformed into a Cauchy-like matrix and split into two independent matrices, each one assigned to an MPI process which could be mapped onto different nodes in a network, or onto the same node thus fixing the problem of the lack of compilers that support OpenMP nested parallelism. Then, each one of these matrices arising from the former partition were factorized concurrently to obtain their LDL^T decomposition, but the scalability of this last factorization step was poor for many cores due to the low memory–CPU throughput of the algorithm. Different *out of order* strategies consisting of producer–consumer task queues implemented with `pthreads` were studied in [4] to improve the speed-up of the algorithm. However, it was shown that the sequential order in the computation of the blocks in which the triangular matrix is partitioned is as fast as other more complicated *out of order* approaches.

In this paper, we use a similar approach based on the transformation of linear system (1) into a *Cauchy-like* linear system. The algorithm applied to factorize Cauchy-like matrices makes use of very regular memory access patterns allowing independent blocks of the resulting triangular factor to be computed concurrently. Block versions of $O(n^3)$ algorithms have traditionally been developed to exploit the hierarchical memory levels. In the case of multicore computers, one step beyond this has been proposed to improve results. This step consists of storing all the data belonging to the same block in consecutive memory locations. It was initially proposed and investigated in [19–21] where the layout is referred to as *Square Block Format*. This technique has been successfully applied to level 3 algorithms of BLAS in [11] where the storage format is called *Block Data Layout* (BDL). Based on a hierarchical organization of the data by blocks, new ideas have been proposed as alternatives to the current implementation of LAPACK in different ways [12,34]. A recent contribution has been proposed, in particular, for a similar problem: the LDL^T decomposition of symmetric indefinite dense matrices [6]. In order to improve the efficiency of the parallel triangularization of each one of the two submatrices, we propose using the BDL storage format. The derived algorithm is also easy to implement since it is based on simple OpenMP directives instead of complicated task queues of `pthreads`.

In order to improve accuracy of the solution of symmetric linear systems by matrix decomposition diagonal pivoting (Bunch–Kaufman) is used. In parallel execution pivoting can reduce performance due to the data interchanging. Some variants of diagonal pivoting have been proposed for the factorization of symmetric indefinite matrices that improve performance, thanks to a reduction in the number of matrix column interchanges. This proposition can be found, e.g., in [31], where it is proposed and studied an algorithm variant supported on *lookahead*-type techniques. Also, for Toeplitz matrices, lookahead techniques have widely studied with the aim at improving accuracy, further to ensure stability of Levinson- and Schur-type algorithms which can even break down for well-conditioned matrices [13,7]. But, in general, look-ahead algorithms for Toeplitz matrices are based on heuristics with variable results (depend on the given matrix) and they are difficult to apply in concurrent environments where we try to keep the sough-after performance of the multicore system. Thus, our option consists of using *local diagonal pivoting* in the algorithm. Although diagonal and local pivoting has limitations compared with full or partial pivoting, we show through some examples that the precision of the solution might be improved. Since symmetric Cauchy-like matrices are not destroyed by diagonal pivoting and pivoting is bound to diagonal blocks (*local pivoting*), the execution time is barely affected.

The next section presents an abridged mathematical description of the problem and shows the overall algorithm. Details about the implementation of the proposed algorithm can be found in Section 3. The block pivoting technique incorporated to the algorithm is described in Section 4. Experimental results are shown in Section 5. Some conclusions are presented in Section 6.

2. Mathematical background

The solution of system (1) can be carried out by solving the linear system

$$C\,\tilde{x} = \tilde{b}, \tag{2}$$

where $C \in \mathbb{R}^{n\times n}$ is known as a *Cauchy-like* matrix, and $\tilde{b}, \tilde{x} \in \mathbb{R}^n$. Matrix $C = [a_{ij}]$ is called *Cauchy-like* (also *generalized Cauchy*) if for certain n-tuples of complex numbers $c = (c_i)_0^{n-1}$ and $d = (d_i)_0^{n-1}$ the matrix

$$\nabla(c, d)C = \left[(c_i - d_j)a_{ij}\right]_0^{n-1},$$

has a rank r which is "small" compared with the order of C. In this paper we deal with real symmetric *Cauchy-like* matrices, i.e., the n-tuples are real and $c = d$ [23]. The normalized Discrete Sine Transformation (DST), represented by means of the symmetric and orthogonal matrix $\mathcal{S}$ as defined in [25], allows system (1) to be transformed into system (2) by performing $C = \mathcal{S}T\mathcal{S}$, $\tilde{x} = \mathcal{S}x$, $\tilde{b} = \mathcal{S}b$. Matrices T and C both belong to the class of *structured matrices* [24]. Structured matrices are characterized by having a "low" *displacement rank* r ($r \ll n$), which briefly means that information contained in the full matrix is implicitly contained in only n-size r vectors. This property can be exploited to derive $O(n^2)$ algorithms for their triangular factorization. In the case of *Cauchy-like* matrix C (2), the displacement rank r is 4.

Working in the *Cauchy-like* domain has an additional advantage since entries c_{ij} such that $i + j$ is odd are 0. Let $P \in \mathbb{R}^{n\times n}$ be the odd–even permutation that positions the odd entries of an array to the top and the even entries to the bottom, then the linear system (2) can be divided into the two independent linear systems $C_i\hat{x}_i = \hat{b}_i$, for $i = 1, 2$, with $C_1 \in \mathbb{R}^{n_1\times n_1}$ and $C_2 \in \mathbb{R}^{n_2\times n_2}$, since

$$PCP^T = \begin{pmatrix} C_1 & \\ & C_2 \end{pmatrix}, \qquad P\tilde{x} = \begin{pmatrix} \hat{x}_1 \\ \hat{x}_2 \end{pmatrix} \quad \text{and} \quad P\tilde{b} = \begin{pmatrix} \hat{b}_1 \\ \hat{b}_2 \end{pmatrix}, \tag{3}$$

where $n_1 = \lceil n/2 \rceil$ and $n_2 = \lfloor n/2 \rfloor$.

Matrices C_1 and C_2 are also *Cauchy-like* (though they do not have zero entries as C) and have a displacement rank of 2, i.e.,

$$\begin{bmatrix} \Lambda_1 & \\ & \Lambda_2 \end{bmatrix}\begin{bmatrix} C_1 & \\ & C_2 \end{bmatrix} - \begin{bmatrix} C_1 & \\ & C_2 \end{bmatrix}\begin{bmatrix} \Lambda_1 & \\ & \Lambda_2 \end{bmatrix} = \begin{bmatrix} G_1 \\ G_2 \end{bmatrix} J \begin{bmatrix} G_1^T & G_2^T \end{bmatrix}^T \tag{4}$$

Algorithm 1 Algorithm for the solution of symmetric Toeplitz linear systems.

Require: Toeplitz vector t of size n and independent vector b.
1: Compute the 8 generator vectors, 4 per each block matrix C_1 and C_2.
2: Transform b into $\hat{b}_1$ and $\hat{b}_2$.
3: Compute the LDL^T decomposition of C_1 and C_2 using Algorithm 2.
4: Solve the two triangular systems for each *Cauchy-like* matrix C_i, $i = 1, 2$.
5: Transform $\hat{x}_1$ and $\hat{x}_2$ into x.

where $G_1 \in \mathbb{R}^{n_1 \times 2}$ and $G_2 \in \mathbb{R}^{n_2 \times 2}$, which are called *generators*, are computed as $\begin{pmatrix} G_1^T & G_2^T \end{pmatrix}^T = \sqrt{2} P \mathcal{J} (u \quad e_1)$, being $u = (0 \quad t_2 \quad t_3 \quad \cdots \quad t_{n-2} \quad t_{n-1} \quad 0)^T$ and e_1 the first column of the identity matrix of order n. Signature matrix J has the form

$$J = \begin{pmatrix} & 1 & & \\ -1 & & & \\ & & & 1 \\ & & -1 & \end{pmatrix}.$$

Let $F \in \mathbb{R}^{n \times n}$ be the matrix with $F_{ij} = 1$ if $|i - j| = 1$ and 0 otherwise, then *displacement matrices* $\Lambda_1 \in \mathbb{R}^{n_1 \times n_1}$ and $\Lambda_2 \in \mathbb{R}^{n_2 \times n_2}$ are computed as $\begin{pmatrix} \Lambda_1 & \oplus & \Lambda_2 \end{pmatrix} = P \mathcal{J} F \mathcal{J} P$.

The algorithm for the solution of the symmetric Toeplitz linear system (1) (Algorithm 1) starts with the computation of a total of 8 vectors, 4 for each submatrix C_i, $i = 1, 2$. Those are the two columns of G_i ($g1$ and $g2$), the diagonal of Λ_i (λ) and the diagonal of C_i (c), for $i = 1, 2$. For the sake of convenience, we denote all of them by *generator vectors* or simply *generators*. Next, the triangular factorization of matrices C_i is carried out. The algorithm used to perform the triangular factorization LDL^T of each one of submatrices C_i, with L unit lower triangular and D diagonal, works on the entries of these 4 generators (Algorithm 2). For the factorization of a matrix of order m, the cost of this algorithm is $\frac{13}{2}m^2 - \frac{39}{2}m$. In step 4, Algorithm 1 computes the solution of two triangular systems with the matrices obtained in the previous step, counting for $\frac{n^2}{4}$ flops. A more detailed description of the backgrounds of the computations of all of these operations can be found in [9].

Algorithm 2 Algorithm for the LDL^T decomposition of a *Cauchy-like* matrix of displacement rank 2.

Require: Generator vectors $g1, g2, \lambda$ and c describing a *Cauchy-like* matrix of the form C_i (3), for $i = 1, 2$.
1: **for** $k = 0, \ldots, n-1$ **do**
2: $d = c_k$
3: $L_{kk} = d$
4: **for** $i = k+1, \ldots, n-1$ **do**
5: $l = (-g2_i g1_k + g1_i g2_k)/(d(\lambda_i - \lambda_k))$
6: $g1_i = g1_i - g1_k l$
7: $g2_i = g2_i - g2_k l$
8: $c_i = c_i - d l^2$
9: $L_{ik} = l$
10: **end for**
11: **end for**

3. Implementation

Step 1 of Algorithm 1 is parallelized using OpenMP sections. The diagonal c of Cauchy-like matrix C (2) is computed with a fast algorithm that allows it to be obtained without explicitly forming C. Vector λ has analytically known entries that just depend on the size of the problem. Details of all of these operations can be found in [9]. We used a Fortran 90 module to apply the DST [2] which, based on the problem size, automatically chooses the best routine between `dfftpack` [32] and Intel MKL [27]. A total of four operations can be performed concurrently for the computation of the generator vectors. Since step 2 is independent from step 1, it is also carried out in a different parallel section. Having computed the first two steps of Algorithm 1, both triangular factorizations are computed (step 3). Since they are completely independent, two OpenMP sections are used. Actually, step 4 can also be performed concurrently at this level of parallelism, so it is carried out in the same section of its corresponding LDL^T factorization. We denote this splitting as *first level of parallelism*. Finally, step 5 of Algorithm 1 turns the solution obtained into the Toeplitz field by means of (3).

Algorithm 3 *Block Data Layout (BDL)* Algorithm for the LDL^T decomposition of a *Cauchy-like* matrix of displacement rank 2.

Require: Generators vectors $g1, g2, \lambda$ and c describing a *Cauchy-like* matrix of the form C_i (3), for $i = 1, 2$.
1: **for** $K = 0, \ldots, M-1$ **do**
2: Compute LD$_{KK}$
3: **for** $I = K+1, \ldots, M-1$ **do**
4: Compute LD$_{IK}$
5: **end for**
6: **end for**

Algorithm 4 Algorithm for the computation of a $m \times \nu$ block of the LDL^T decomposition of a *Cauchy-like* matrix of displacement rank 2.

Require: Arrays $g1, g2, \lambda$ and c of size m, and arrays $\bar{g1}, \bar{g2}, \bar{\lambda}$ and d of size ν which represent parts of generator $\tilde{G}$ and the diagonals of $\tilde{\Lambda}$ and $\tilde{C}$ of (6).
1: **for** $k = 0, \ldots, \nu-1$ **do**
2: **for** $i = 1, \ldots, m-1$ **do**
3: $l = (-g2_i \bar{g1}_k + g1_i \bar{g2}_k)/(d_k(\lambda_i - \bar{\lambda}_k))$
4: $g1_i = g1_i - \bar{g1}_k l$
5: $g2_i = g2_i - \bar{g2}_k l$
6: $c_i = c_i - d_k l^2$
7: $L_{ik} = l$
8: **end for**
9: **end for**

The main contribution to the efficiency of the algorithm is the parallelization of the LDL^T factorization of each matrix C_i (3) that we call *second level of parallelism*. Hence, we use OpenMP nested parallelism to combine both levels of parallelism. It can be said that, in general, it is difficult to achieve a good speed-up with algorithms of cost $O(n^2)$ mainly due to both the irregular memory access patterns that result in many cache misses and their intrinsic low throughput per data accessed in memory. In order to improve the efficiency of the Toeplitz solver, we rewrote Algorithm 2 so that it worked by blocks. Hence, we modified the code in Algorithm 2 to develop a *BDL version*, i.e., a blocked algorithm that uses the BDL storage format to store factor L (Algorithm 3). Let LDL^T be the factorization of C_i, and let LD be a block-matrix with lower triangular blocks on the diagonal and rectangular blocks below diagonal, then Algorithm 3 lets to obtain matrix LD containing the lower triangular factor L and the diagonal entries of D in its diagonal stored in a BDL format. The algorithm works on the $M = \lceil n/\nu \rceil$ blocks, for a given block size ν, in which matrix LD is partitioned. The computation of the lower triangular blocks of the diagonal in step 2, i.e. LD$_{KK}$, is carried out by means of Algorithm 2. Upon completion the computation of LD$_{KK}$, all blocks in column K down block LD$_{KK}$, i.e. blocks LD$_{IK}$, $I > K$, are computed (steps 3–5). This computation is carried out efficiently thanks to Algorithm 4, which is derived from Algorithm 2. To obtain block LD$_{IK}$, $I > K$, of size $m \times \nu$, being $m \leq \nu$ (blocks LD$_{(M-1)K}$ could have less rows than ν), Algorithm 4 is called with the following arguments in the same order they appear in the entry of the routine: $g1(i_I : j_I)$, $g2(i_I : j_I)$, $\lambda(i_I : j_I)$, $c(i_I : j_I)$, $g1(i_K : j_K)$, $g2(i_K : j_K)$, $\lambda(i_K : j_K)$, $diag(\text{LD}(i_K : j_K))$, where $i_I = I\nu$, $j_I = I\nu + m - 1$, $i_K = K\nu$ and $j_K = (K+1)\nu - 1$.

The threaded algorithm corresponding to Algorithm 3 computes each column of *LD* in the outer loop sequentially. On the contrary, the square blocks of the column in the inner loop are

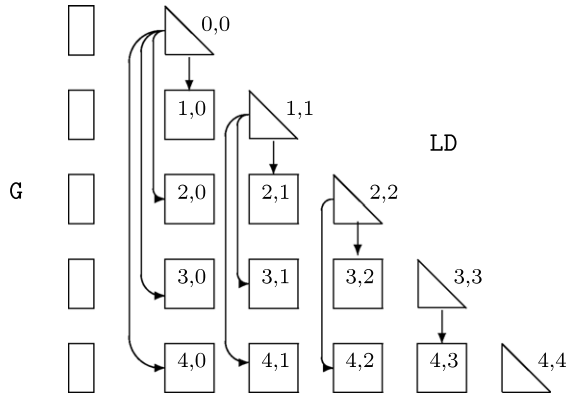


Fig. 1. Example of 5 × 5 block-matrix LD flow layout of Algorithm 3.

computed concurrently by means of a parallel-for OpenMP primitive. The dependency flow of tasks shown in Fig. 1 represents the parallel implementation of the blocked algorithm, where each task is denoted by the coordinates of LD block computed by the task. All the blocks on the same column down the diagonal block are computed in parallel after the computation of the diagonal block in data-level parallelism. Task-level parallelism has also been explored in [4] without significant improvements for this algorithm. Fig. 1 also shows the generator vectors (G) partitioned in rectangular blocks.

The BDL storing format has an additional advantage in memory saving. Using Algorithm 2 to factorize C requires a space of $n \times n$ entries to store L in column major order although only the lower part would be referenced. On the contrary, the BDL storage scheme only allocates memory for blocks in the lower triangular part of the matrix. Although a small change in Algorithm 2 could allow to save the factor in packed form, the low performance achieved by the BLAS routine for the triangular system solution discourages the use of this option.

4. Block diagonal pivoting

For the sake of simplicity we denote by C each of the two matrices C_i (3), $i = 1, 2$, in this section. A displacement representation for matrix C can be

$$\Lambda C - C\Lambda = GHG^T, \tag{5}$$

where Λ is a diagonal matrix such that $\lambda_i \neq \lambda_j$, $\forall i, j$. This displacement equation is obtained by equating (4). Each member of (5) has a rank of 2. Actually, a Cauchy-like matrix can be defined as the symmetric matrix C that keeps the rank of 2 of (5) for a given matrix Λ. Matrix G denotes each one of the two generators G_i, $i = 1, 2$, mentioned in Section 2. Matrix H is a 2×2 signature matrix.

Let P a permutation matrix (the identity matrix with any reordering of its rows), then we perform the transformation $P(\cdot)P^T$ to (5) so

$$\begin{aligned} P(\Lambda C - C\Lambda)P^T &= (P\Lambda P^T)(PCP^T) - (PCP^T)(P\Lambda P^T) \\ &= \tilde{\Lambda}\tilde{C} - \tilde{C}\tilde{\Lambda}^T = (PG)H(PG)^T = \tilde{G}H\tilde{G}^T, \end{aligned} \tag{6}$$

given that P is orthogonal. It is easy to show that matrix $\tilde{\Lambda}$ is diagonal, i.e., matrix Λ with the diagonal entries reordered according with P. In addition, the transformation used keeps the rank of 2 of the equation. Thus, $\tilde{C}$ is a also Cauchy-like matrix like C. In other words, it can be said that pivoting does not destroy the displacement structure of Cauchy-like matrices.

The factorization of C with pivoting consists of obtaining the LDL^T of a modified version of C. This modified version is obtained by applying a sequence of permutation matrices $\{P_k\}_{k=0}^{n-1}$ such that $P_{n-1} \cdots P_1 P_0 C P_0^T P_1^T \cdots P_{n-1}^T = LDL^T$.

Each permutation P_k of the sequence is calculated so that the diagonal element c_k, which is the first element of the diagonal of the Schur complement of C regarding $C_{0:k-1,0:k-1}$ (the Schur complement if $k = 0$ is C), is interchanged with c_j, being $c_j = \max_{k \le m \le n-1} |c_m|$. This diagonal pivoting is the same to that named *symmetric pivoting* in [15] providing the algorithm with the same stability properties and limitations. Algorithm 2 can be easily modified to incorporate pivoting at it is shown in Algorithm 5; in fact, only the addition of steps 2 and 3 in the latter makes them different.

Algorithm 5 Algorithm for the LDL^T decomposition of a *Cauchy-like* matrix of displacement rank 2 with pivoting.

Require: Columns $g1$ and $g2$ of $\tilde{G}$, arrays λ and c containing the diagonal of $\tilde{\Lambda}$ and $\tilde{C}$ (6), respectively.
1: **for** $k = 0, \ldots, n-1$ **do**
2: Search for j so that $c_j = \max_{k \le m \le n-1} |c_m|$.
3: Swap the pairs $(c_k, c_j), (g1_k, g1_j), (g2_k, g2_j), (\lambda_k, \lambda_j), (L_{k,0:k-1}, L_{m,0:k-1})$
4: $d = c_k$
5: $L_{kk} = d$
6: **for** $i = k+1, \ldots, n-1$ **do**
7: $l = (-g2_i g1_k + g1_i g2_k)/(d(\lambda_i - \lambda_k))$
8: $g1_i = g1_i - g1_k l$
9: $g2_i = g2_i - g2_k l$
10: $c_i = c_i - dl^2$
11: $L_{ik} = l$
12: **end for**
13: **end for**

In order to keep tasks independent so that they can be computed concurrently by separate cores, we propose to perform local pivoting. With local pivoting, the pivoting is limited to the computation of each diagonal block keeping the swaps local to the actual diagonal block not affecting the rest of the blocks. Lets see how the local block pivoting algorithm works with a simple example on a 2×2 blocks matrix.

Suppose that we make a 2×2 block partition of C and let $P = P_1 \oplus P_2$ be a diagonal block permutation matrix, the block diagonal pivoting algorithm obtains the following block triangular decomposition

$$\begin{aligned} PCP^T &= \begin{pmatrix} P_1 & \\ & P_2 \end{pmatrix} \begin{pmatrix} C_1 & C_3^T \\ C_3 & C_2 \end{pmatrix} \begin{pmatrix} P_1^T & \\ & P_2^T \end{pmatrix} \\ &= \begin{pmatrix} L_1 & \\ L_3 & L_2 \end{pmatrix} \begin{pmatrix} D_1 & \\ & D_2 \end{pmatrix} \begin{pmatrix} L_1^T & L_3^T \\ & L_2^T \end{pmatrix}, \end{aligned} \tag{7}$$

where L_1, L_2 are unit lower triangular, L_3 is rectangular and D_1, D_2 are diagonal.

The algorithm starts by computing $P_1 C_1 P_1^T = L_1 D_1 L_1^T$ by means of Algorithm 5, which also returns the permutation matrix P_1. After this step we have

$$\begin{pmatrix} P_1 & \\ & I \end{pmatrix} \begin{pmatrix} C_1 & C_3^T \\ C_3 & C_2 \end{pmatrix} \begin{pmatrix} P_1^T & \\ & I \end{pmatrix} = \begin{pmatrix} L_1 D_1 L_1^T & P_1 C_3^T \\ C_3 P_1^T & C_2 \end{pmatrix}. \tag{8}$$

Now, the algorithm computes a rectangular factor M such that $C_3 P_1^T = M D_1 L_1^T$. The computation of this factor is carried out by means of Algorithm 4 as explained in Section 3. Using factor M we have that expression (8) is equal to

$$\begin{pmatrix} L_1 D_1 L_1^T & L_1 D_1 M^T \\ M D_1 L_1^T & M D_1 M^T + \tilde{C}_2 \end{pmatrix} = \begin{pmatrix} L_1 \\ M \end{pmatrix} D_1 \begin{pmatrix} L_1^T & M^T \end{pmatrix} + \begin{pmatrix} 0 & 0 \\ 0 & \tilde{C}_2 \end{pmatrix}. \tag{9}$$

Matrix $\tilde{C}_2$ is the Schur complement of matrix (8) regarding block $P_1 C_1 P_1^T$, i.e., $\tilde{C}_2 = C_2 - M D_1 M^T$. This Schur complement is not

explicitly formed, it is implicitly known by means of its generators which are also returned by Algorithm 4 when computes M.

The algorithm continues by computing $P_2\tilde{C}_2P_2^T = L_2D_2L_2^T$ through Algorithm 5. Lets see the form of the second term in (9) when permutation P_2 is applied,

$$\begin{pmatrix} I & \\ & P_2 \end{pmatrix} \begin{pmatrix} L_1D_1L_1^T & L_1D_1M^T \\ MD_1L_1^T & MD_1M^T + \tilde{C}_2 \end{pmatrix} \begin{pmatrix} I & \\ & P_2^T \end{pmatrix} = \begin{pmatrix} L_1D_1L_1^T & L_1D_1M^TP_2^T \\ P_2MD_1L_1^T & P_2MD_1M^TP_2^T + P_2\tilde{C}_2P_2^T \end{pmatrix}.$$

This matrix can also be expressed as

$$\begin{pmatrix} L_1 & \\ P_2M & L_2 \end{pmatrix} \begin{pmatrix} D_1 & \\ & D_2 \end{pmatrix} \begin{pmatrix} L_1^T & M^TP_2^T \\ & L_2^T \end{pmatrix},$$

showing the decomposition (7), where $P_2M = L_3$. The product P_2M is not explicitly formed since, as will be shown later, it can be avoided in later stages, as we are interested in solving the linear system. Extending this reasoning to partitions with more than two blocks can be derived easily.

We continue with the same 2×2 block matrix example to show the solution of the Cauchy-like system $Cx = b$ when pivoting is used, i.e.,

$$PCP^TPx = Pb, \tag{10}$$

where P defined in (7). System (10) is equivalent to

$$\begin{pmatrix} L_1 & \\ P_2M & L_2 \end{pmatrix} \begin{pmatrix} D_1 & \\ & D_2 \end{pmatrix} \begin{pmatrix} L_1^T & M^TP_2^T \\ & L_2^T \end{pmatrix} \begin{pmatrix} P_1x_1 \\ P_2x_2 \end{pmatrix} = \begin{pmatrix} P_1b_1 \\ P_2b_2 \end{pmatrix}. \tag{11}$$

Then, the solution to system (10) is obtained by solving

$$\begin{pmatrix} L_1 & \\ P_2M & L_2 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = \begin{pmatrix} P_1b_1 \\ P_2b_2 \end{pmatrix}, \tag{12}$$

where

$$\begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = \begin{pmatrix} D_1 & \\ & D_2 \end{pmatrix} \begin{pmatrix} L_1^T & M^TP_2^T \\ & L_2^T \end{pmatrix} \begin{pmatrix} P_1x_1 \\ P_2x_2 \end{pmatrix},$$

and this is performed by solving, first, the lower triangular system $L_1y_1 = P_1b_1$ to obtain y_1 and, second, solving

$$P_2My_1 + L_2y_2 = P_2b_2 \rightarrow L_2y_2 = P_2b_2 - P_2My_1 \rightarrow L_2y_2 = P_2(b_2 - My_1),$$

to obtain y_2.

Having the solution to the lower triangular system (12), the solution to (10) continues by solving

$$\begin{pmatrix} D_1 & \\ & D_2 \end{pmatrix} \begin{pmatrix} z_1 \\ z_2 \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \end{pmatrix},$$

where

$$\begin{pmatrix} z_1 \\ z_2 \end{pmatrix} = \begin{pmatrix} L_1^T & M^TP_2^T \\ & L_2^T \end{pmatrix} \begin{pmatrix} P_1x_1 \\ P_2x_2 \end{pmatrix}, \tag{13}$$

by solving $D_1z_1 = y_1$ and $D_2z_2 = y_2$.

The solution of (13) is performed similarly to the solution of (12). Let

$$\begin{pmatrix} w_1 \\ w_2 \end{pmatrix} = \begin{pmatrix} P_1x_1 \\ P_2x_2 \end{pmatrix},$$

then we solve $L_2^Tw_2 = z_2$ to obtain w_2 and $x_2 = P_2^Tw_2$. Now, looking the upper equation of (13), $L_1^Tw_1 + M^TP_2^Tw_2 = z_1$, we see that w_1 can be obtained by solving $L_1^Tw_1 = z_1 - M^Tx_2$ and, therefore, $x_1 = P^Tw_1$. Algorithm 6 summarizes the described process for the solution of the triangular system for the general case of more than 2 blocks. Diagonal of matrices in lines 6 and 15 is implicitly 1 so it is not referenced. These triangular system solutions are carried out by the BLAS routine `dtrsv`. The matrix–vector products are carried out with routine `dgemv`. The division in line 9 is a point-wise division between pair-wise arrays elements.

Algorithm 6 Algorithm for the solution of a system $LDL^Tx = b$ with the form (11).

Require: Matrix LD returned by Algorithm 3, a set of permutation matrices P_I and a right hand side vector b partitioned in N blocks b_I, where $I = 0, \ldots, N-1$.

{Unit lower triangular system }
1: **for** $J = 0$ **to** $M-1$ **do**
2: **for** $I = 0$ **to** $J-1$ **do**
3: $b_J \leftarrow b_J - \mathrm{LD}_{JI}y_I$
4: **end for**
5: $b_J \leftarrow P_Jb_J$
6: Solve $\mathrm{LD}_{JJ}y_J = b_J$
7: **end for**
{Diagonal system }
8: **for** $J = 0, \ldots, M-1$ **do**
9: $z_J = y_J / \mathrm{diag}(\mathrm{LD}_{JJ})$
10: **end for**
{Unit upper triangular system }
11: **for** $J = N-1$ **downto** 0 **do**
12: **for** $I = N-1$ **downto** $J+1$ **do**
13: $z_J \leftarrow z_J - \mathrm{LD}_{JI}^Tx_I$
14: **end for**
15: Solve $\mathrm{LD}_{JJ}^Tw_J = z_J$
16: $x_J = P_J^Tw_J$
17: **end for**

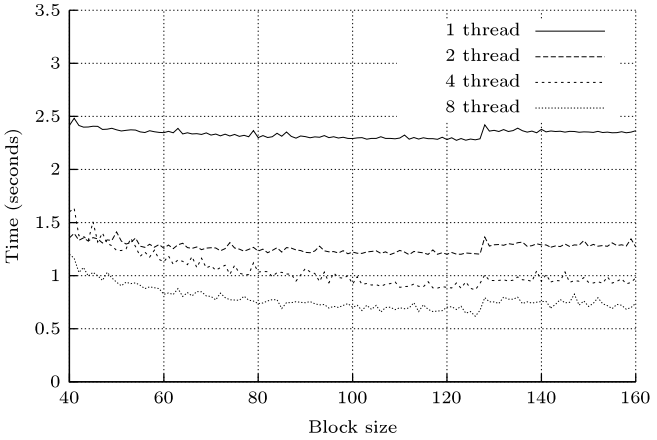


Fig. 2. Time of system solution (Algorithm 1) regarding block size ($n = 20\,000$).

With this example we have shown that permutation P only must be applied to one dimensional arrays showing that the product P_2M does not need to be explicitly formed.

5. Experimental results

The experimental results have been obtained on a computer which consists of two processors AMD Opteron™ Processor 6272 at 2.1 GHz. with 16 cores each. The processor module consists of two dies, each with four dual-core modules sharing a cache memory (L3) of 6 MB. The total amount of main memory is 32 GB. The compiler used is GNU `gcc` (Version 4.4.6).

The blocked algorithm is sensitive to the block size, so a preliminary study for the best size must be done. Fig. 2 shows the runtime behavior with regard to the block size for a matrix of size $n = 20\,000$. The time shown in Fig. 2 is for all the computation of Algorithm 1, that is, the overall computation for the solution of the linear system. The behavior of the entire algorithm according to this parameter allows to obtain the value for which the solution is obtained in the shortest time possible. The cache size is the most influential factor for the block size. We did more tests varying the problem side and increasing the number of threads used obtaining

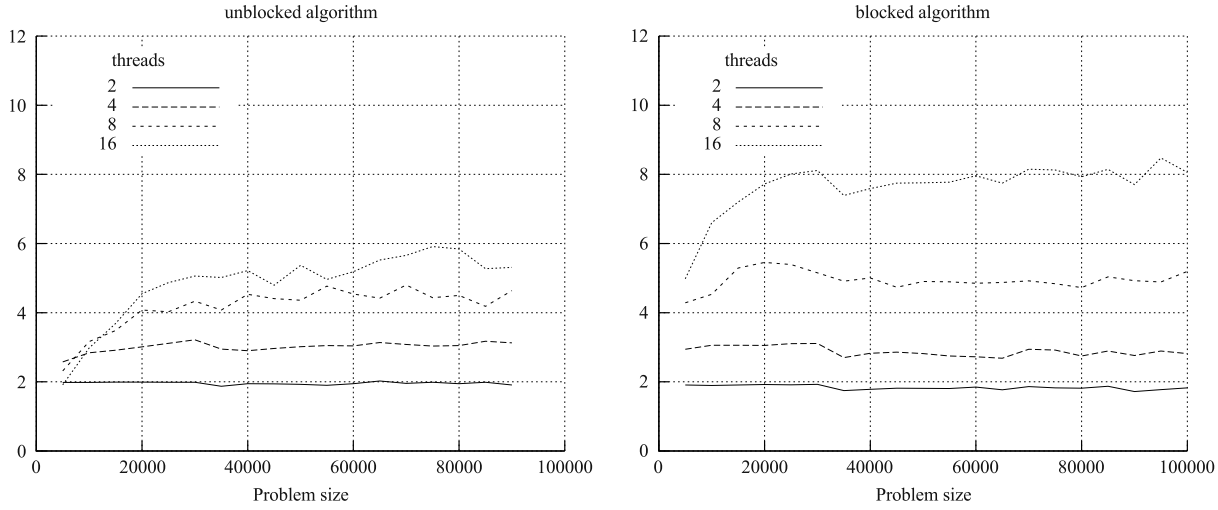


Fig. 3. Speed-up of the factorization of Cauchy-like matrix C (2). Left is for the unblocked algorithm and right is for the blocked algorithm.

Table 1
Time results in one core of the different parts of Algorithm 1.

n	Init.	Decomp.	Triang. Syst. Sol.	Obtaining x	Total
10 000	00.8E−2	0.5	0.1	0.2E−2	0.6
20 000	01.4E−2	1.8	0.4	0.3E−2	2.2
30 000	14.6E−2	4.1	0.9	4.6E−2	5.0
40 000	03.2E−2	7.9	1.7	0.7E−2	9.6
50 000	34.8E−2	12.5	2.6	11.2E−2	15.1
60 000	35.1E−2	18.3	3.7	11.2E−2	22.0

always the same conclusion, i.e., a fix value of 126 for the block size is suitable to run the application as fast as possible in most of the cases.

The analysis of the algorithm continues with the study of the sequential time consumed by the different parts of the algorithm. Table 1 shows in four columns the time spent by Algorithm 1 in each one of the four parts in which the algorithm divides all the computation:

- the initialization, which consists of steps 1 and 2 of Algorithm 1;
- the LDL^T decomposition (step 3 of Algorithm 1);
- the Triangular Systems Solution (step 4 of Algorithm 1); and
- obtaining the final solution x to system (1) (step 5 of Algorithm 1).

Clearly, the second and third stages are the most time-consuming parts of the process. The time used on these parts grows quadratically with the problem size with 5 being the approximate ratio between them. Thus, the parallelization should be focused on these two steps. The time used on the first and last parts of the algorithm are not directly correlated with the problem size. On the contrary, the computational cost of the DST is proportional to the size of the largest of the prime factors of the problem size. Either way, the Fortran 90 module mentioned in the implementation section selects the best routine to apply the transformation so this particular effect of the DST is lessened and the time is kept within a very small strip.

For the triangular factorization of the Cauchy-like matrix in (2), two levels of parallelism are used as described in the implementation section. The algorithm appeared in [9] also used both levels of parallelism, by partitioning the work into two MPI processes, each one in charge of factorizing one of the two rank-2 Cauchy-like matrices (C_1 and C_2, respectively) resulting from the splitting process shown in (3). Each one of these two matrices were factorized in turn by an OpenMP parallel loop. We denote here this version as *unblocked* algorithm, where the two MPI processes have been replaced by two OpenMP threads instead. The parallel loop has been suitably tuned to get the maximum performance by fixing the schedule policy to `static` and choosing the best chunk size (500 for the target machine).

Fig. 3 shows the results obtained in form of speed up and the evolution with the problem size. The figures compare the two versions only for the factorization of the Cauchy-like matrix C. Both algorithms speed up well when only 2 threads are used since this is the case when only the first level of parallelism is used. More than 2 threads mean that half the total of threads participates in the factorization of each one of the two rank-2 Cauchy-like matrices (second level of parallelism). For example, with 4 threads, the factorization of C_1 and C_2 is performed by 2 threads each. The performance of the unblocked version is limited by the memory data transfer. This is just what the new blocked version tries to avoid by using the BDL storage format, and it is what it can be observed when more than 4 threads are used. Furthermore, the BDL storage format allows tackle larger matrices since it is only needed to allocate memory to store the blocked lower triangular matrix. This result shows the scalability of the factorization method achieved by a suitable rearrangement of the data in memory.

The graphics in Fig. 4 show the total time for the solution of the Toeplitz linear system (1). The sequential algorithm is very fast so there is no significant speed-up with small problem sizes. In particular, the low throughput of the parallel implementation of the triangular systems solution in step 4 of Algorithm 1 somehow burdens the total solution. However, savings in time are larger with the increase in the problem size. The algorithm in the experiment also uses local diagonal pivoting and thread affinity. (Both parameters are optional in our application.) We tested that no large differences exist for different thread-core binding combinations, obtaining only slightly better results by mapping each of the two subproblems (3) on each processor. Just to eliminate variable thread placement during the computation, which allows reusing memory resources (caches), implies some improvement in performance. Since Algorithm 2 is used to factorize two matrices of order $n/2$, the total cost of step 3 of Algorithm 1 is $\frac{13}{4}n^2 - \frac{39}{2}n$. This operations are carried on 4 n-arrays to produce a large set of $n^2/2$. Thus the most time-consuming operation is storing data into memory, fact that limits the performance of the algorithm in terms of speedup when the number of threads is large. Given the characteristics of the algorithm, we value that the parallel application exploits the total amount of cores available in this hardware to produce the result in the minimum possible time.

Table 2 shows some features of Algorithm 1 in comparison with other methods. We used the Levinson algorithm implemented in routine `tsld` of the Netlib Library [29]. For the comparison, we

Table 2
Time results and accuracy of different Toeplitz solvers.

	$n = 10\,001$					$n = 30\,000$		
		Random		KMS(10E−14)		Random		
	Time	For. err.	Back. err.	For. err.	Back. err.	Time	For. err.	Back. err.
Algorithm 1	0.22	2.0E−7	1.4E−12	1.3E−10	4.2E−14	1.30	5.9E−6	1.5E−11
Algorithm 1 + piv	0.22	8.6E−9	2.7E−14	1.3E−10	4.2E−14	1.30	9.3E−8	3.6E−14
Levinson	0.34	2.5E−9	9.2E−12	2.6E+6	8.6E−3	2.20	9.2E−7	5.0E−10
dsysv	12.5	1.4E−14	8.7E−15	1.4E−14	8.7E−15	216	3.1E−11	5.3E−15

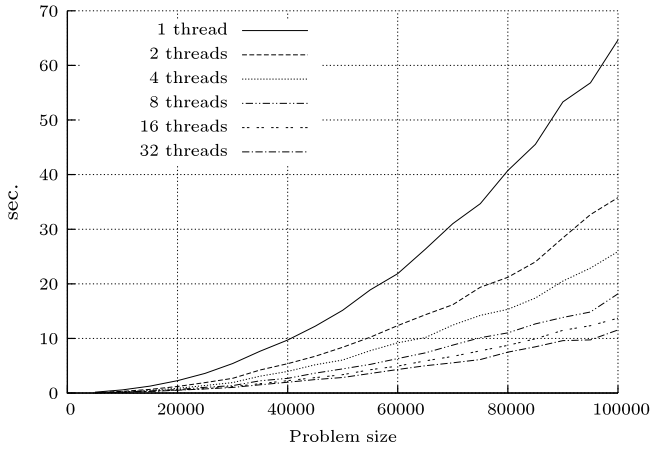


Fig. 4. Time for the solution of the symmetric Toeplitz linear system (1).

also picked the Lapack [5] routine `dsysv`, which solves symmetric linear systems with pivoting of general matrices. In particular we used the Intel MKL implementation of this routine. For randomly generated matrices of sizes $n = 10\,001$ and $n = 30\,000$, the time obtained by Algorithm 1 using 32 threads was smaller that the Levinson algorithm. Obviously, the Lapack routine spent more time computing the linear system since it is a $O(n^3)$ flops algorithm that does not take into account the Toeplitz structure of the matrix. We also note that the use of our local diagonal pivoting technique did not have a significant impact on the execution time.

We studied forward and backward errors defined as

$$\frac{\|x - \hat{x}\|}{\|x\|}, \qquad \frac{\|b - T\hat{x}\|}{\|T\| \cdot \|b\|},$$

respectively, where $\hat{x}$ is the computed solution and x has 1 in all its entries. The case for $n = 10\,001$ shows that the most accurate solution was obtained when the structure of the matrix was not taken into account. The second most accurate result was attained with Algorithm 1 with pivoting followed by the Levinson algorithm. It is well known that the Levinson algorithm offers an inaccurate solution and might even break down if the Toeplitz matrix is not *strongly regular* (a matrix is *strongly regular* if all its leading principal minors are well-conditioned). To show this, we used a Kac–Murdock–Szego (KMS(α)) matrix defined as $t_0 = \alpha$ and $t_i = 0.5^i$, for $i = 1, \ldots, n-1$. A KMS(α) matrix has all its principal minors of order m ill-conditioned if $m - 1$ is a multiple of 3 and $\alpha = 10^{-14}$. The Cauchy-like transformation method, however, is insensitive to this problem.

6. Conclusions

In this paper, we have addressed the solution of symmetric Toeplitz linear systems in multicore systems. By transforming the system matrix into a Cauchy-like matrix we built a blocked algorithm in which all blocks in a column can be concurrently computed. The special structure of the matrix was exploited to have a *fast* algorithm. We used the *Block Data Layout* scheme to manage data in memory, thus obtaining a good speed-up with the usual number of cores available in current machines. The resulting parallel algorithm is easy to implement with OpenMP. There exist techniques allowing algorithms based on Levinson and Schur-type methods to deal with non *strongly regular* matrices, e.g., *look ahead* algorithms. However, the algorithm that factorizes the Cauchy-like matrix is not affected by this problem making these techniques unnecessary. Furthermore, since pivoting does not destroy the structure of Cauchy-like matrices, we have proposed incorporating local diagonal pivoting, which does not impair the execution time and improves the accuracy of the solution. This algorithm is available in `StructPack` [3], which is a package for the solution of structured matrix problems.

The designed algorithm does not leverage efficiently all available cores. This inefficiency comes from the time spent on writing the resulting factors of the triangular decompositions into memory, which leads to a CPU–memory bus saturation. In addition, the solution of the triangular systems is a low cost operation compared to the decomposition, yet this operation is hardly parallelizable and negatively affects the efficiency of the whole algorithm as the number of cores increases. A future concept to explore consists of designing a variant of the algorithm in which the first lower triangular system solution is merged into the triangular decomposition so that both operations would be realized in a single step. With this, we aim at increasing the number of operations performed per data stored in memory, and only one triangular system solution would be computed a posteriori, i.e. once the triangular factor is stored in memory. As current systems have an increasing number of cores, these aggregated resources can be used to perform most of the new operations involved in the simultaneous solution of the triangular system of equations and the triangular factorization.

References

[1] P. Alonso, J.M. Badía, A.M. Vidal, Parallel algorithms for the solution of Toeplitz systems of linear equations, Lecture Notes in Comput. Sci. 3019 (2004) 969–976.
[2] P. Alonso, M. Bernabéu, A.-M. Vidal-Maciá, An Adaptive Interface for the Efficient Computation of the Discrete Sine Transform, in: Lecture Notes in Computer Science, 4967, 2008, pp. 89–98.
[3] P. Alonso-Jordá, P. Martínez-Naredo, F.J. Martínez-Zaldívar, J. Ranilla-Pastor, A.M. Vidal-Maciá, StructPack: a package for the solution of structured matrix problems, 2013. http://www.inco2.upv.es/structpack.html.
[4] P. Alonso, A.M. Vidal, Cauchy-like system solution on multicore platforms, StructPack Working Note 2 (SPAWN 002), February 2014, http://www.inco2.upv.es/spawn002.pdf.
[5] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J.D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, D. Sorensen, LAPACK Users' Guide, second ed., SIAM, Philadelphia, 1999.
[6] M. Baboulin, D. Becker, J. Dongarra, A parallel tiled solver for dense symmetric indefinite systems on multicore architectures, in: 26th IEEE International Parallel and Distributed Processing Symposium, IPDPS 2012, Shanghai, China, May 21–25, 2012, 2012, pp. 14–24.
[7] M.V. Barel, A. Bultheel, A lookahead algorithm for the solution of block Toeplitz systems, Linear Algebra Appl. 266 (0) (1997) 291–335.
[8] J.A. Belloch, M. Ferrer, A. González, J. Lorente, A.M. Vidal, GPU-based WFS systems with mobile virtual sound sources and room compensation, in: Proceedings of the 52nd AES Conference on Sound Field Control-Engineering and Perception, 2013.
[9] M.O. Bernabeu, P. Alonso, A.M. Vidal, A multilevel parallel algorithm to solve symmetric Toeplitz linear systems, J. Supercomput. 44 (2008) 237–256.

[10] J.R. Bunch, Stability of methods for solving Toeplitz systems of equations, SIAM Journal on Scientific and Statistical Computing 6 (2) (1985) 349–364.
[11] A. Buttari, J. Langou, J. Kurzak, J. Dongarra, A class of parallel tiled linear algebra algorithms for multicore architectures, Parallel Comput. 35 (2009) 38–53.
[12] E. Elmroth, F. Gustavson, I. Jonsson, B. Kågström, Recursive blocked algorithms and hybrid data structures for dense matrix library software, SIAM Rev. 46 (1) (2004) 3–45.
[13] R.W. Freund, A look-ahead Bareiss algorithm for general Toeplitz matrices, Numer. Math. 68 (1) (1994) 35–69.
[14] I. Gohberg, T. Kailath, V. Olshevsky, Fast Gaussian elimination with partial pivoting for matrices with displacement structure, Math. Comp. 64 (212) (1995) 1557–1576.
[15] G.H. Golub, C.F.V. Loan, Matrix Computations, third ed., in: Johns Hopkins Studies in the Mathematical Sciences, The Johns Hopkins University Press, Baltimore, MD, USA, 1996.
[16] A. González, J.J. López, Time domain recursive deconvolution in sound reproduction, in: Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing, ICASSP'00, vol. 2, 2000, pp. 833–836.
[17] A. González, J.J. López, Software toolbox for 3D sound using loudspeakers, in: Proceedings of the AES 16th International Conference on Spatial Sound Reproduction, 1999.
[18] A. González, J.J. López, Practical and efficient implementation of the Levinson algorithm for multichannel sound reproduction, in: Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing, ICASSP'00, vol. 2, 2000, pp. 777–780.
[19] F.G. Gustavson, Recursion leads to automatic variable blocking for dense linear-algebra algorithms, IBM J. Res. Dev. 41 (6) (1997) 737–755.
[20] F. Gustavson, New generalized data structures for matrices lead to a variety of high performance dense linear algebra algorithms, in: J. Dongarra, K. Madsen, J. Wasniewski (Eds.), Applied Parallel Computing, in: Lecture Notes in Computer Science, vol. 3732, Springer, Berlin/Heidelberg, 2006, pp. 11–20.
[21] F. Gustavson, J. Gunnels, J. Sexton, Minimal Data Copy for Dense Linear Algebra Factorization, in: B. Kågström, E. Elmroth, J. Dongarra, J. Wasniewski (Eds.), Applied Parallel Computing. State of the Art in Scientific Computing, in: Lecture Notes in Computer Science, vol. 4699, Springer, Berlin/Heidelberg, 2007, pp. 540–549.
[22] S. Haykin, Adaptive Filter Theory, Prentice Hall, Upper Saddle River, NJ., 1996.
[23] T. Kailath, A.H. Sayed (Eds.), Fast Reliable Algorithms for Matrices with Structure, SIAM, Philadelphia, PA, 1999.
[24] T. Kailath, A.H. Sayed, Displacement Structure: Theory and Applications, SIAM Rev. 37 (3) (1995) 297–386.
[25] C.V. Loan, Computational Frameworks for the Fast Fourier Transform, SIAM Press, Philadelphia, 1992.
[26] J. Lorente, J.A. Belloch, M. Ferrer, A. González, M. de Diego, A.M. Vidal, Multichannel active noise control system over graphics processing units, in: Proceedings of Internoise, 2012.
[27] Intel®, Intel Math Kernel Library for the Linux* OS 10.1. User's Guide, 2008. http://developer.intel.com.
[28] T. Moon, W. Stirling, Mathematical Methods and Algorithms for Signal Processing, Prentice Hall, Upper Saddle River, NJ, 2000.
[29] Netlib Library, 1982. http://www.netlib.org/toeplitz/index.html.
[30] J. Proakis, Advanced Topics in Digital Signal Processing, Macmillan Publishing Company, New York, 1992.
[31] P.E. Strazdins, Accelerated methods for performing the LDLT decomposition, ANZIAM J. 42 (C) (2000) C1328–C1355.
[32] P. Swarztrauber, Vectorizing the FFT's, in: G. Rodigue (Ed.), Parallel Computations, 1982, pp. 51–83.
[33] S. Thirumalai, High performance algorithms to solve Toeplitz and block Toeplitz systems, Ph.D. Thesis, Graduate College of the University of Illinois at Urbana-Champaign, 1996.
[34] G. Quintana-Ortí, E.S. Quintana-Ortí, R.A. van de Geijn, F.G. Van Zee, E. Chan, Programming matrix algorithms-by-blocks for thread-level parallelism, ACM Trans. Math. Software 36 (3) (2009) 14:1–14:26.

Pedro Alonso was born in Valencia, Spain, in 1968. He received the engineering degree in computer science from the Universitat Politècnica de València, Spain, in 1994 and the Ph.D. degree from the same University in 2003. His dissertation was on the design of parallel algorithms for structured matrices with application in several fields of digital signal analysis. He is a professor in the Department of Computer Science and a member of the High Performance Networking and Computing Research Group of the Universitat Politècnica de València. His main areas of interest include numerical algorithms and parallel computing.

Manuel F. Dolz received the BSc degree in computer science from the Universitat Jaume I, Castellón, in 2008, and the MSc degree in Parallel and Distributed Computing from the Universitat Politècnica de València, in 2010. He is currently working toward the PhD degree in the Department of Computer Science Engineering at the Universitat Jaume I. His main research interests include power-aware computing techniques for clusters and multicore processors in the high performance computing domain.

Antonio M. Vidal received his M.S. degree in Physics from the "Universidad de Valencia", Spain, in 1972, and his Ph.D. degree in Computer Science from the "Universidad Politécnica de Valencia", Spain, in 1990. Since 1992 he has been with the Universidad Politécnica de Valencia, Spain, where he is currently a full professor in the Department of Computer Science. He is the coordinator of the project "High Performance Computing on Current Architectures for Problems of Multiple Signal Processing", currently developed by INCO2 Group and financed by the Generalitat Valenciana, in the frame of PROMETEO Program for research groups of excellence. His main areas of interest include parallel computing with applications in numerical linear algebra and signal processing.