The Journal of Supercomputing, 32, 251–278, 2005
© 2005 Springer Science + Business Media, Inc. Manufactured in The Netherlands.

An Efficient Parallel Algorithm to Solve Block–Toeplitz Systems

P. ALONSO palonso@dsic.upv.es
Departamento de Sistemas Informáticos y Computación, Universidad Politécnica de Valencia

J. M. BADÍA badia@icc.uji.es
Departamento de Ingeniería y Ciencia de los Computadores, Universidad Jaume I, Castellón, Spain

A. M. VIDAL avidal@dsic.upv.es
Departamento de Sistemas Informáticos y Computación, Universidad Politécnica de Valencia

Abstract. In this paper, we present an efficient parallel algorithm to solve Toeplitz–block and block–Toeplitz systems in distributed memory multicomputers. This algorithm parallelizes the Generalized Schur Algorithm to obtain the semi-normal equations. Our parallel implementation reduces the communication cost and optimizes the memory access. The experimental analysis on a cluster of personal computers shows the scalability of the implementation. The algorithm is portable because it is based on standard tools and libraries, such as ScaLAPACK and MPI.

Keywords: Block Toeplitz matrices, Toeplitz Block matrices, Generalized Schur Algorithm, distributed memory architectures

1. Introduction

In this paper, we present a new parallel algorithm that solves the following system of linear equations

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

where $T \in \mathrm{R}^{n\times n}$ is a *block–Toeplitz* matrix defined as

$$T = \begin{pmatrix} B_0 & B_{-1} & \dots & B_{1-n/\nu} \\ B_1 & B_0 & & \vdots \\ & \ddots & \ddots & \\ B_{n/\nu-1} & \dots & & B_0 \end{pmatrix}, \tag{2}$$

with each $B_i \in \mathrm{R}^{\nu\times\nu}$, for $i = 1 - n/\nu, \dots, n/\nu - 1$, being a non-structured matrix and with $b, x \in \mathrm{R}^n$ being the independent and the unknown vectors, respectively.

The same proposed algorithm can be used with a slight modification to solve the linear system $\hat{T}\hat{x} = \hat{b}$, where $\hat{T}$ is a *Toeplitz–block* matrix defined as

$$\hat{T} = \begin{pmatrix} T_{0,0} & T_{0,1} & \dots & T_{0,\nu-1} \\ T_{1,0} & T_{1,1} & & \vdots \\ T_{\nu-1,0} & \dots & & T_{\nu-1,\nu-1} \end{pmatrix}, \tag{3}$$

where each $T_{ij} \in \mathrm{R}^{n/\nu \times n/\nu}$, for $i, j = 0, \dots, \nu - 1$, is a scalar Toeplitz matrix and $\hat{b}, \hat{x} \in \mathrm{R}^n$ are the independent and the unknown vectors, respectively. Systems (2) and (3) are related because matrix $\hat{T}$ can be obtained from matrix T by means of a permutation matrix P.

Many different methods can be used to solve (1). In this paper, the normal equations $T^T T x = T^T b$ are solved in order to obtain the solution of the linear system. This is not a recommended method for the solution of non-structured linear systems. Nevertheless, it can offer an accurate solution if T is a Toeplitz–like matrix in same cases. This is mainly due to the fact that algorithms used for the Toeplitz–like case do not build the product $T^T T$ explicitly.

The usual solution of a Toeplitz–block system (3) is carried out by transforming the system matrix into the corresponding block–Toeplitz form (2) using the permutation matrix P mentioned above. This method allows us to use BLAS3–type operations, and it offers very efficient results. However, this method is very difficult to parallelize in a distributed memory architecture. The dependency among small granularity operations forces very frequent and expensive communications.

The parallel algorithm proposed in this work directly solves a Toeplitz–block system (3) by means of the solution of the normal equations. If the system matrix has a block–Toeplitz structure (2), then the systems matrix is transformed into the corresponding block–Toeplitz structure. In the following sections, we will show that our parallel algorithm avoids most of the communications of the previous methods by exploiting the structure of the Toeplitz–block matrices. The communications of our algorithm are reduced to only one broadcast in each iteration. Furthermore, the communication cost has been reduced even more by means of packing techniques that reduce the number of such broadcasts.

Another important aspect concerning the execution time of the parallel algorithm proposed is the pattern used to access the data stored in the local memory of each processor. Our experimental analysis shows that different patterns to access memory have a great impact on the the execution time. This factor is specially important in the case of algorithms with an asymptotic cost of $O(n^2)$ operations. In order to reduce the memory access time, our parallel algorithm always tries to access matrix elements stored in consecutive memory positions

The experimental results obtained from a cluster of personal computers show a great reduction in the execution time when the number of processors is increased. Furthermore, the algorithm is backward–stable and provides a very accurate solution. In addition, the parallel algorithm is portable because it is implemented using free–source computation and communication libraries.

The rest of this paper is structured as follows. In the following section, we give a brief description of the preceding work on the solution of Toeplitz systems. The concept of *displacement structure* of Toeplitz–like matrices is reviewed in the following sections, focusing on the block–Toeplitz case. In Section 5, a well-known algorithm for solving the normal equations with a cost of $O(n^2)$ is shown. In Section 6, our proposed parallel algorithm is described in detail. The experimental results are shown in Section 7. Finally, some conclusions are presented in the last section.

2. State of the art

In 1979, Kailath, Kung y Morf introduced the concept of *displacement rank* and showed that Toeplitz matrices have a very low displacement rank. Moreover, Toeplitz-like matrices, including block–Toeplitz and Toeplitz–block matrices, also have a very low displacement rank. Matrices of this kind arise in many applications in physics and engineering [22].

Regarding the solution of the system (1) in the scalar case, that is when $\nu = 1$, many more references can be found than in the case of block–Toeplitz or Toeplitz–block systems.

There exist *fast algorithms* that exploit the structure of Toeplitz–like matrices to solve the system (1) with a cost of $O(n^2)$ operations, instead of the $O(n^3)$ operations of classical algorithms that do not exploit the special structure of Toeplitz–like matrices. Algorithms of this kind can be divided into *Levinson-type*, if they perform a decomposition of matrix T^{-1}, and *Schur-type*, if they perform the decomposition of T. However, Schur-type algorithms have become very popular because they better exploit matrix properties such as low rank displacement or band structure; they have better numerical properties when dealing with positive definite matrices [8] and they can be better parallelized [24].

Block Schur-type algorithms can be found in [25, 30, 34]. More recently, Thirumalai [32] developed several block algorithms that use BLAS3 operations in order to increase their performance.

Schur-type algorithms apply the Generalized Schur Algorithm to solve (1) with block–Toeplitz matrices (2). If we are dealing with a Toeplitz–block matrix, we must first permute it into a block Toeplitz form. Other algorithms [21] solve the block linear system by directly applying scalar operations.

Another different approach to solve system (1) is based on the Gokhberg–Semencul inversion formulas. The parameter of these formulas can be computed by solving an interpolation problem, as shown in [23, 33], or the solution of the system can be obtained with methods based on the properties of Sylvester matrices [10, 11, 12, 26]. Algorithms of this kind use a look-ahead technique and an iterative refinement in order to improve the accuracy of the solution. However, there exist cases in which this method does not work as well as in the scalar case, as mentioned in [33]. Other algorithms based on inversion formulas and rational interpolation can be found in [1, 18].

There are few algorithms that solve system (1) in parallel. A great number of these algorithms use systolic architectures [31], and they solve some specific problems arising in digital signal analysis. One important limitation of these algorithms is that they can only be applied to positive definite matrices. Some algorithms for more general parallel

platforms exist, but they can only be used with particular kinds of Toeplitz matrices, like tridiagonal or banded [15]. Other parallel algorithms have also been developed based on iterative methods [27]. There also exist some parallel Levinson-type algorithms to solve Toeplitz [19] and block–Toeplitz [14] equation systems on shared-memory multiprocessors.

However, there exist very few parallel algorithms for solving block–Toeplitz systems on distributed–memory multicomputers. We will describe some of them [16, 17] in the following sections.

3. Displacement structure

We define the displacement rank $\nabla_{F,A}$ of a matrix $M \in \mathrm{R}^{m \times n}$ with respect to two lower triangular matrices $F \in \mathrm{R}^{m \times m}$ and $A \in \mathrm{R}^{n \times n}$, called displacement matrices, as

$$\nabla_{F,A} = M - FMA^T. \tag{4}$$

We say that a matrix M is structured or that it has a low displacement rank if the rank r of $\nabla_{F,A}$ is very small compared with the rank of M ($r \ll n$). The diagonal entries of F and A satisfy $1 - f_i a_j \neq 0$ for all i and j, iff matrix $\nabla_{F,A}$ has a unique factorization, such as

$$\nabla_{F,A} = GB^T. \tag{5}$$

Matrices $G \in \mathrm{R}^{m \times r}$ and $B \in \mathrm{R}^{n \times r}$ are called *generators* of A.

In the symmetric case, the displacement Equation (4) of M has the following form:

$$\nabla_F = M - FMF^T = GJG^T, \tag{6}$$

with $J \in \mathrm{R}^{r \times r}$ being the *signature matrix*. The signature matrix is diagonal and all its entries are 1 or -1. The number of 1 and -1 values is equal to the number of positive and negative eigenvalues of ∇_F, respectively. Matrices G and J in (6) are called a *generator pair* or M.

Toeplitz matrices are structured matrices, that is, they have a very low displacement rank. For example, if $T \in \mathrm{R}^{n \times n}$ is a Toeplitz symmetric matrix,

$$T = \begin{pmatrix} t_0 & t_{-1} & \dots & t_{1-n} \\ t_1 & t_0 & & \\ & \ddots & \ddots & \\ t_{n-1} & \dots & & t_0 \end{pmatrix},$$

then

$$\nabla_Z = T - ZTZ^T = GJG^T, \tag{7}$$

where the displacement matrix $Z \in \mathrm{R}^{n\times n}$ has the following form

$$Z = \begin{pmatrix} 0 & & & \\ 1 & 0 & & \\ & \ddots & \ddots & \\ & & 1 & 0 \end{pmatrix}, \tag{8}$$

and the generator pair of (7) is

$$G = \frac{1}{\sqrt{t_0}} \begin{pmatrix} t_0 & 0 \\ t_1 & t_1 \\ \vdots & \vdots \\ t_{n-1} & t_{n-1} \end{pmatrix} \quad J = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}.$$

It can be seen that the displacement rank of T is 2, independently of its size.

The Generalized Schur Algorithm (GSA) obtains a fast triangular factorization of matrices with displacement structure. The triangular decomposition of structured matrices is *fast* because the GSA operates on the $n \times r$ entries of the generator instead of on the $n \times n$ entries of the whole matrix. In each of the n iterations of the algorithm, the generator or the generator pair is transformed to the *proper form* using a unitary transformation. A generator in proper form has only one nonzero entry in its first nonzero row. The proper form is obtained by means of a unitary transformation Θ such that

$$G\Theta\Theta^{-T}B^T = (G\Theta)(B\Theta^{-1})^T = G_p B_p^T, \quad \Theta\Theta^{-T} = I,$$

if the displacement matrix is non-symmetric, and

$$G\Theta J\Theta^T G^T = (G\Theta)J(G\Theta)^T = G_p J G_p^T, \quad \Theta J \Theta^T = J,$$

if the displacement matrix is symmetric, where p denotes the generator in proper form.

The GSA exploits the property that the successive Schur complements of a structured matrix are also structured matrices. Therefore, the generator of a Schur complement can be obtained from the previous one by using the same transformation process. This can be proved as follows. Consider a structured matrix $M \in \mathrm{R}^{n\times n}$ with the form (4), with $m_{1,1} = 1$ and with $u, v \in \mathrm{R}^n$ being the first column and row, respectively. Then, we define the matrix

$$S = M - uv^T = \begin{pmatrix} 0 & 0 \\ 0 & C \end{pmatrix}, \tag{9}$$

where $C \in \mathrm{R}^{(n-1)\times(n-1)}$ is the Schur complement of M with respect to $m_{1,1}$.

The displacement of matrix S can be obtained as

$$
\begin{aligned}
S - FSA^T &= M - uv^T - F(M - uv^T)A^T \\
&= M - FMA^T - uv^T + Fuv^TA^T \\
&= GB^T - uv^T + Fuv^TA^T \\
&= [G \quad -u \quad Fu][B \quad v \quad Av]^T.
\end{aligned}
$$

If the generators are in the proper form, then

$$
\begin{aligned}
S - FSA^T &= [G \quad -u \quad Fu][B \quad v \quad Av]^T \\
&= [Fu \quad G_{1:n,2:r}][Av \quad B_{1:n,2:r}]^T \\
&= G'B'^T. \qquad (10)
\end{aligned}
$$

The Schur algorithm computes the LU decomposition of matrix M, with L and $U \in \mathrm{R}^{n\times n}$, where L is a unit lower triangular matrix and U is an upper triangular matrix. This decomposition is computed from the generator pair GB^T (5). The first column of L and the first row of U are the same as the first columns of G and B, respectively.

By applying (10) to GB^T, we obtain the generators $G'B'^T$ for the displacement of matrix S (9) with respect to $m_{1,1}$, and so we obtain the displacement of the Schur complement C of M. The second column of L and the second row of U are the same as the first columns of G' and B' respectively. We obtain the LU decomposition of M by repeating this process as shown in Algorithm 1.

Algorithm 1 (*GSA*). *Given a generator pair G and B for the displacement of the structured matrix M with respect to matrices F and A* (4), *the following algorithm returns the triangular factors L and U of M.*

for $i = 1, \ldots, n$
1. *Choose and apply a unitary transformation that transforms generators G and B to the proper form.*
2. *The first column of G is the ith column of L, while the first column of B is the ith row of U.*
3. *Assign $FG_{1:n,1}$ to the first row of G and $AB_{1:n,1}$ to the first column of B.*

end for

Algorithm 1 can be further generalized by allowing us to choose the elements of the first row of the generators to be zeroed in order to transform the generator to the proper form [21]. It is also possible to apply a certain number of iterations $p < n$ of the algorithm in order to obtain a partial decomposition of M and the Schur complement generator of M with respect to the principal sub-matrix of size p.

4. Displacement structure for block matrices

Block–Toeplitz matrices (2) and Toeplitz–block matrices (3) are structured (also called Toeplitz-like) matrices.

Given a block–Toeplitz matrix T, its displacement representation can be defined as

$$T - FTF^T = \begin{pmatrix} B_0 & B_{-1} & \dots & B_{1-n/\nu} \\ B_1 & 0 & \dots & 0 \\ \vdots & \vdots & & \vdots \\ B_{n/\nu-1} & 0 & \dots & 0 \end{pmatrix}, \tag{11}$$

where F is the following displacement matrix of size n

$$F = Z^\nu = \begin{pmatrix} 0 & & & \\ I_\nu & 0 & & \\ & \ddots & \ddots & \\ & & I_\nu & 0 \end{pmatrix}, \tag{12}$$

with Z being the displacement matrix defined in (8), ν the block size and I_ν the identity matrix of size ν. That is, the displacement matrix F is a zero matrix of size n with ones in the $(\nu + 1)$th sub-diagonal.

The form of the displacement generators of T can be obtained analytically [32]. Schur-type algorithms can be used to compute the lower triangular factor L of $T = LL^T$ in the symmetric positive definite case, or the lower triangular factor L and the diagonal factor D of $T = LDL^T$ in the non-definite symmetric case, or the lower triangular factor L and the upper triangular factor U of $T = LU$ in the non-symmetric case.

Matrices T (2) and $\hat{T}$ (3) are related by a permutation matrix $P \in \mathrm{R}^{n\times n}$ whose elements are

$$p_{i,j} = \begin{cases} 1 & \text{if } j = ((i-1) \bmod \nu)\nu + ((i-1) \operatorname{div} \nu) + 1 \\ 0 & \text{in other case} \end{cases}, \quad \forall\, i, j = 1, \dots, n,$$

where div is the quotient of the integer division and *mod* is the remainder. It is easy to see that $\hat{T} = PTP^T$ and that $PP^T = P^TP = I$, with I being the identity matrix. A displacement representation of $\hat{T}$ can be obtained from (11) using the previous relation,

$$P(T - FTF^T)P^T = PTP^T - (PFP^T)(PTP^T)(PFP^T)^T = \hat{T} - \hat{F}\hat{T}\hat{F}^T,$$

where $\hat{F} = PFP^T$ for the matrix F defined in (12).

The displacement matrix $\hat{F}$ can be written as

$$\hat{F} = PFP^T = PZ^\nu P^T = Z_{n/\nu} \oplus \cdots \oplus Z_{n/\nu},$$

that is, matrix $\hat{F}$ is block diagonal and has the form

$$\hat{F} = \begin{pmatrix} Z_{n/\nu} & & \\ & \ddots & \\ & & Z_{n/\nu} \end{pmatrix}, \tag{13}$$

with each block of the diagonal being the displacement matrix Z (8) of size n/ν.

5. Solution of the normal equations

In order to solve the linear system (1), we consider the normal equation system

$$T^T T x = T^T b. \tag{14}$$

associated to the least squares problem

$$\min_x \|Tx - b\|.$$

The triangular Cholesky decomposition of $T^T T = R^T R$ leads us to the semi-normal equations

$$R^T R x = T^T b, \tag{15}$$

where R is the upper triangular factor of the QR decomposition of T.

In many cases, solving (1) by means of the solution of the normal equations is not recommended because the condition number $\kappa(T^T T)$ can be as large as $\kappa(T)^2$ [20, Section 5.3]. However, this method can be used in the Toeplitz case because, as has been proved in [6] using the results in [8] and [7], the computed triangular factor $\tilde{R}$ satisfies

$$\|\tilde{R}^T \tilde{R} - T^T T\| = O(\varepsilon \|T^T T\|),$$

therefore, the error bound does not depend on $\kappa(T^T T)$.

Furthermore, it is also shown in [6] that

$$\frac{\|T\tilde{x} - b\|}{\|T\|\|x\|} = O(\varepsilon\kappa(T)),$$

with $\tilde{x}$ being the computed solution after solving the matrix product and the two triangular systems in (15).

Moreover, factor R in (15) can be computed using the GSA with a cost of $O(n^2)$, and the semi-normal Equations (15) can be solved with $O(n^2)$ additional operations. The GSA can

be used to perform the triangular decomposition of $T^T T$ because $T^T T$ is structured. The rank of the displacement representation of $T^T T$ for a scalar matrix T is 4.

With respect to the stability of the GSA, an algorithm that solves a well conditioned equation system is said to be *weakly stable* if $\|\tilde{x} - x\|/\|x\|$ is small [6, 9]. The GSA applied to the normal equations is weakly stable. This result can be sufficient in many practical applications.

The GSA can fail if the matrix to decompose is not strongly regular, that is, if any of its leading sub-matrices is singular. If the leading sub-matrices are ill-conditioned, the algorithm does not fail, but it obtains inaccurate results. Nevertheless, the product $T^T T$ is a symmetric positive definite matrix so it is strongly regular.

Applying the GSA to $T^T T$ instead of T is more expensive because we have 4 columns in the generator instead of the 2 columns of the generator for the displacement representation of T; however, we do not need to apply any *look-ahead* or any other technique to improve the stability of the algorithm and the accuracy of the results.

In addition, we can always increase the accuracy of the solution by applying some iterative refinement step such as the following. Let $\tilde{x}_1$ be the computed solution after solving the system (15) by using an approximated decomposition

$$T^T T + \delta T^T T = \tilde{R}^T \tilde{R}.$$

The iterative refinement consists of computing the residual vector

$$r_1 = T^T b - T^T T x_1, \tag{16}$$

and solving the linear system

$$\tilde{R}^T \tilde{R}\, \Delta x_1 = r_1, \tag{17}$$

in order to obtain the correction factor Δx_1. The precision of the computed solution of the linear system (1) increases by applying

$$x_2 = x_1 + \Delta x_1. \tag{18}$$

This process can be repeated as many times as necessary to improve the accuracy of the results.

A block version of the GSA can be applied to solve the normal equations with block–Toeplitz matrices (2). In this case, the displacement of the product $T^T T$ has the following form

$$T^T T - F T^T T F^T = G J G^T, \tag{19}$$

where F is the displacement matrix of size n defined in (12), G is the generator, which in this case is a matrix of size $n \times 4\nu$, and where $J = I_{2\nu} \oplus -I_{2\nu}$ is the signature matrix.

The generator G has the following form

$$G = \begin{pmatrix} S_0 & 0 & 0 & 0 \\ S_1 & T_{-1}^T & S_1 & T_{n/\nu-1}^T \\ S_2 & T_{-2}^T & S_2 & T_{n/\nu-2}^T \\ \vdots & \vdots & \vdots & \vdots \\ S_{n/\nu-1} & T_{1-n/\nu}^T & S_{n/\nu-1} & T_1^T \end{pmatrix}, \tag{20}$$

where

$$S = T^T U R^{-1}, \quad U = \begin{pmatrix} T_0 \\ T_1 \\ \vdots \\ T_{n/\nu-1} \end{pmatrix}, \quad R = \mathrm{qr}(U), \tag{21}$$

with $\mathrm{qr}(U)$ being the sub-matrix $R_{1:\nu,1:\nu}$ of the factor R in the QR decomposition of U. R denotes the upper triangular factor of the Cholesky decomposition of $U^T U$ such that $U^T U = R^T R$.

6. The parallel algorithm

The first approaches to solve system (1) in parallel by applying the normal equations are based on the triangular decomposition of $T^T T$. If we are dealing with a Toeplitz–block matrix $\hat{T}$, this matrix is first permuted into a block–Toeplitz form T using P (13).

In [16, 17, 32], S. Thirumalai presents an efficient block algorithm to perform the previous decomposition. This algorithm uses a method that is similar to the one in LAPACK [2] to compute and apply Householder transformations [3, 28] following a notation called WY. The algorithm obtains good performance by applying BLAS3 operations.

Following a different approach, Kailath and Chun [21], propose a GSA to solve (1) with block–Toeplitz or Toeplitz–block matrices without having to perform any permutation. Their algorithm is based on scalar operations, which the authors consider to be advantageous for systolic or DSP architectures.

The parallel algorithm of S. Thirumalai consists of an iterative process with two basic steps: (1) the transformation of the generators to the proper form and (2) the displacement of a column. The algorithm can be described as follows. Let us begin with a generator

partitioned into two columns of blocks such as

$$G^0 = \begin{pmatrix} G_{0,0} & G_{0,1} \\ G_{1,0} & G_{1,1} \\ G_{2,0} & G_{2,1} \\ \vdots & \vdots \\ G_{n/\nu,0} & G_{n/\nu,1} \end{pmatrix},$$

where the super-index shows the iteration number. The first step of the iteration transforms this generator to the proper form by zeroing block $G_{0,1}$,

$$\tilde{G}^0 = \begin{pmatrix} \tilde{G}_{0,0} & 0 \\ \tilde{G}_{1,0} & \tilde{G}_{1,1} \\ \tilde{G}_{2,0} & \tilde{G}_{2,1} \\ \vdots & \vdots \\ \tilde{G}_{n/\nu,0} & \tilde{G}_{n/\nu,1} \end{pmatrix}. \tag{22}$$

The second step of the iteration down-shifts the first block column by applying a displacement matrix F (12), and obtains the following reduced generator

$$G^1 = \begin{pmatrix} 0 & 0 \\ \tilde{G}_{0,0} & \tilde{G}_{1,1} \\ \tilde{G}_{1,0} & \tilde{G}_{2,1} \\ \vdots & \vdots \\ \tilde{G}_{n/\nu-1,0} & \tilde{G}_{n/\nu,1} \end{pmatrix}. \tag{23}$$

If we distribute the generator by block rows on an array of processors, we have to perform two kinds of communications on each iteration:

– A broadcast of the parameters of the unitary transformation that zeroes block $G_{i,1}$.
– A parallel point-to-point communication among adjacent processors to perform the shift.

The sequential version of the method is very efficient, as shown in [17, 32]. However, the parallel implementation has a large communication cost. The effect of the communications increases if we take into account the small computational cost of this method when applied to structured matrices. The behaviour of this algorithm has been experimentally confirmed by its authors on distributed memory multicomputers [16].

6.1. Reducing the communication cost

One of the main improvements of our parallel algorithm over the previous one is the elimination of the point-to-point communications during the displacement step. Our parallel algorithm works on the generator $\hat{G}$ of the displacement representation of $\hat{T}^T\hat{T}$,

$$\hat{T}^T\hat{T} - \hat{F}\hat{T}^T\hat{T}\hat{F}^T = \hat{G}J\hat{G}^T, \tag{24}$$

where $\hat{T} = PTP^T$, $\hat{G} = PG$ y $\hat{F} = PFP^T$. The main advantage of dealing with a Toeplitz–block instead of working with the block–Toeplitz representation is that the displacement matrix $\hat{F}$ has the block diagonal form shown in (13). Therefore, the shift (or product $\hat{F}G_{1:n,0}$) in each iteration of the algorithm can be performed independently on each block of n/ν rows. If we distribute a different block or group of blocks on each processor, we avoid the communications among adjacent processors during the displacement steps of the algorithm. Hence, the only communications necessary in the resulting algorithm are the broadcasts of the transformation parameters in each iteration.

However, if we use this technique, we cannot apply the transformations in a blocked way, as proposed in [17]. Instead, we have to perform a displacement after transforming each row of the generator to the proper form. In this sense, we use a method that is similar to the one proposed in [21], although we always take the first column of the generator as the pivot column.

Let us detail the main steps of each iteration of our parallel triangularization process. As an example, we use a small Toeplitz–block matrix $\hat{T} \in \mathrm{R}^{12\times 12}$ with $\nu \times \nu = 3 \times 3$ blocks of size $n/\nu \times n/\nu = 4 \times 4$. The generator of the displacement of $\hat{T}^T\hat{T}$ is $\hat{G} \in \mathrm{R}^{12\times 12}$. We distribute the blocks of this generator cyclically by blocks of $n/\nu = 4$ rows in an array of 3 processors P_0, P_1 and P_2, as shown in Figure 1, where x denotes each element of the generator. We have added a subindex to the elements of the first column to show the displacement step.

During the first step of the first iteration, the algorithm computes the unitary transformation Θ that zeroes all the entries of the first row of the generator except for the first one. This step is locally performed by P_0. The parameters of Θ are broadcast to the rest of the processors. The result of this first step is shown in Figure 2, where $\tilde{x}$ represents modified entries.

During the second step of the first iteration, the first column of the generator is shifted by applying $\hat{F}G_{1:n,1}$. As $\hat{F}$ is a block diagonal matrix (13), this shift is independently performed on each block, and thus, it can be performed locally because each block belongs completely to one processor (Figure 3). It can be observed that all the elements of the first column remain in their own processor, as shown by the subindexes.

In Figure 4, we have extracted the elements of the first column from the first block in the previous example. All the entries of the block remain in the same processor during the entire the iteration.

	x_1	x	x	x	x	x	x	x	x	x	x	x
P_0	x_2	x	x	x	x	x	x	x	x	x	x	x
	x_3	x	x	x	x	x	x	x	x	x	x	x
	x_4	x	x	x	x	x	x	x	x	x	x	x
	x_5	x	x	x	x	x	x	x	x	x	x	x
P_1	x_6	x	x	x	x	x	x	x	x	x	x	x
	x_7	x	x	x	x	x	x	x	x	x	x	x
	x_8	x	x	x	x	x	x	x	x	x	x	x
	x_9	x	x	x	x	x	x	x	x	x	x	x
P_2	x_{10}	x	x	x	x	x	x	x	x	x	x	x
	x_{11}	x	x	x	x	x	x	x	x	x	x	x
	x_{12}	x	x	x	x	x	x	x	x	x	x	x

Figure 1. Initial state of the Generator.

	$\tilde{x}_1$	0	0	0	0	0	0	0	0	0	0	0
P_0	$\tilde{x}_2$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$
	$\tilde{x}_3$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$
	$\tilde{x}_4$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$
	$\tilde{x}_5$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$
P_1	$\tilde{x}_6$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$
	$\tilde{x}_7$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$
	$\tilde{x}_8$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$
	$\tilde{x}_9$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$
P_2	$\tilde{x}_{10}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$
	$\tilde{x}_{11}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$
	$\tilde{x}_{12}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$

Figure 2. Generator after computing and applying Θ to zero all the entries of the first row except for the first one.

		0	0	0	0	0	0	0	0	0	0	0	0
P_0	$\downarrow$	$\tilde{x}_1$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$
		$\tilde{x}_2$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$
		$\tilde{x}_3$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$
		0	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$
P_1	$\downarrow$	$\tilde{x}_5$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$
		$\tilde{x}_6$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$
		$\tilde{x}_7$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$
		0	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$
P_2	$\downarrow$	$\tilde{x}_9$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$
		$\tilde{x}_{10}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$
		$\tilde{x}_{11}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$

Figure 3. Generator after the shift step.

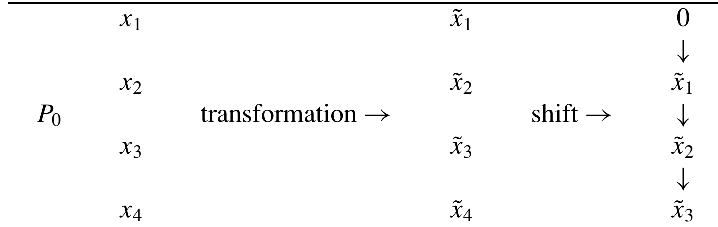

Figure 4. Main steps of the first iteration on processor P_0.

6.2. *Implementation of the unitary transformations*

Although our algorithm does not exploit the blocked application of the transformations shown in [17], the hyperbolic transformations used on each iteration produce more accurate results. In [29], it is shown that if the hyperbolic transformations are applied directly as in [17], the GSA may be unstable, while if they are applied in a factorized way, the algorithm is stable.

More specifically, our algorithm uses the transformation method presented in [13]. The hyperbolic transformation $\Theta \in \mathrm{R}^{4\nu\times 4\nu}$ consists of two Householder reflections and an elementary hyperbolic rotation. If $g = (g_1 \ldots g_{2\nu}\ g_{2\nu+1} \ldots g_{4\nu})$ is the first non-zero row of the generator, then the hyperbolic transformation has the following form

$$\Theta = \begin{pmatrix} \Theta_1 & 0 \\ 0 & I \end{pmatrix} \begin{pmatrix} I & 0 \\ 0 & \Theta_2 \end{pmatrix} \Theta_3, \tag{25}$$

where $\Theta_1 \in \mathrm{R}^{2\nu\times 2\nu}$ is a Householder reflection that zeroes the first 2ν element of g, except for the first one,

$$(g_1'\ 0 \cdots 0) \leftarrow (g_1 g_2 \cdots g_{2\nu})\Theta_1,$$

$\Theta_2 \in \mathrm{R}^{2\nu\times 2\nu}$ is a Householder reflection that zeroes the last 2ν elements of g except for the first one,

$$(g_{2\nu+1}' 0 \cdots 0) \leftarrow (g_{2\nu+1} g_{2\nu+2} \cdots g_{4\nu})\Theta_2,$$

and $\Theta_3 \in \mathrm{R}^{4\nu\times 4\nu}$ is an elementary hyperbolic rotation that zeroes the element $g_{2\nu+1}'$ of g using g_1' as pivot element. The hyperbolic rotation is computed and applied using the OD method described in [13]. It can be easily seen that $\Theta J \Theta^T = J$, with $J = I_{2\nu} \oplus -I_{2\nu}$.

6.3. *Implementation of the parallel algorithm*

The parallel algorithm presented in this paper uses a message-passing model on a distributed-memory multicomputer of a logical array of p processors. The algorithm solves system (1), where T is a block–Toeplitz matrix (2). It can be divided into three main steps:

Algorithm 2 (*Parallel Algorithm for the system solution*). *This algorithm solves the system $Tx = b$ in parallel, where T is a block–Toeplitz matrix of the form* (2), *b is the independent vector, and x is the solution vector.*

1. *Parallel computation of the generator for the displacement of $\hat{T}^T\hat{T}$, where $\hat{T}$ is a Toeplitz–block matrix* (3).
2. *Cholesky triangularization of the product $\hat{T}^T\hat{T} = LL^T$ using the Parallel GSA* (*Algorithm* 3).
3. *Parallel solution of the system by means of*

$$x \leftarrow \hat{T}^T b, \tag{26}$$

$$x \leftarrow L^{-1}x, \tag{27}$$

$$x \leftarrow L^{-T}x, \tag{28}$$

$$x \leftarrow Px, \tag{29}$$

where P is the permutation matrix that transforms $\hat{T}$ to T.

The first step of Algorithm 2, the parallel computation of the generator $\hat{G}$, begins with the computation of the generator G (20) and then permutes it using P. Each processor has all the data of the problem, $T_{1-n/\nu}, \ldots, T_0, \ldots, T_{n/\nu-1}$, including b, so each processor independently computes blocks $\hat{S}_i$, $i = 0, \ldots, \nu - 1$ of matrix $\hat{S}$ that belongs to it,

$$\hat{S} = PS = PT^TUR^{-1} = \begin{pmatrix} \hat{S}_0 \\ \vdots \\ \hat{S}_{\nu-1} \end{pmatrix}. \tag{30}$$

Now, we need to compute the R factor of the QR factorization of U (21). Factor R can be computed in one processor and broadcast to the rest in order to perform the product UR^{-1}. However, we have chosen to replicate the QR factorization of U on all the processors so that the communication is avoided. The computational cost of computing R is lower than the communication cost of broadcasting R, especially in networks with a high latency. For the computation of $\hat{S}$ (30), each computed row of S is placed in the correct memory location of the processor according to the permutation matrix P, avoiding the interchange of rows in memory after the computation of S.

The second step of our parallel algorithm is the most costly step. It is described in Algorithm 3.

Algorithm 3 (*Parallel GSA to decompose the normal equations*). *Given the generator $\hat{G} \in R^{n\times 4\nu}$ of the displacement of matrix $\hat{T}^T\hat{T}$* (3) *with respect to $\hat{F}$* (24), *which is cyclically distributed by blocks on an array of p processors with a block size of $n/\nu \times 4\nu$, the following algorithm returns the lower triangular factor L distributed in ν blocks of n/ν rows so that $\hat{T}^T\hat{T} = LL^T$.*

Each processor P_k, *with* $k = 0, \ldots, p-1$, *performs the following steps*:

for $i = 1, \ldots, n$
  if $G_{i,1:4\nu} \in P_k$
    1. *Choose a unitary transformation* Θ_i (25)
      that zeroes all the elements of $G_{i,1:4\nu}$ *except for the first one.*
    2. $L_{i,i} \leftarrow G_{i,1}$.
    3. *Broadcast the transformation to the rest of processors.*
  else
    1. *Receive the transformation* Θ_i.
  end if
  for $j = i+1, \ldots, n$
    if $G_{j,1:4\nu} \in P_k$
      1. *Apply transformation* Θ_i *to* $G_{j,1:4\nu}$.
      2. $L_{j,i} \leftarrow G_{j,1}$.
      if $G_{j-1,1:4\nu} \in P_k$
        $G_{j,1} \leftarrow L_{j-1,i}$.
      else
        $G_{j,1} \leftarrow 0$.
      end if
    end if
  end for
end for

Algorithm 3 summarizes the steps shown for the iteration example in Figures 1, 2 and 3 taking into account that, in the ith iteration, the first column of $\hat{G}$ is the ith column of L. Algorithm 3 is a parallel version of Algorithm 1 for the symmetric case.

We would like to emphasize that only one broadcast is performed in each iteration. The shift of the first column of the generator is performed independently on each block, and therefore locally in each processor.

The theoretical computational cost of Algorithm 3 is $O(\frac{n^2\nu}{p})$ floating point operations.

For the theoretical analysis of the communication time, let us represent the cost involved in sending a message of size n as $\beta + n\alpha$, where β represents the latency time to start a message transmission and α represents the time needed to send a floating point real number. The theoretical communication cost of Algorithm 3 is

$$n(\beta + (4\nu + 4)\alpha),$$

where $4\nu + 4$ is the number of values to represent each unitary transformation Θ.

This communication cost shows the relative weight of performing n broadcasts. In distributed-memory multicomputers, it is very inefficient to send a large number of small messages. In order to alleviate this problem, we have reduced the number of broadcasts by increasing the size of the messages. We have modified Algorithm 3 so that each processor computes and applies n/ν transformations and broadcasts them in the same message. The rest of the processors receive the message and apply n/ν consecutive transformations. This

modification of the algorithm gives a communication cost of

$$\nu\left(\beta + \frac{n}{\nu}(4\nu + 4)\alpha\right).$$

The choice of the best number of transformations to be packed in each message has very important effects on the performance of the parallel algorithm. We determined experimentally that packing all the transformations of a block does not produce the best performance. In the implemented parallel algorithm, we have chosen a value $\eta \in [1, n\nu]$ for the number of transformations to be packed in each message. This value has been determined experimentally and depends on the characteristics of the architecture.

Let us detail the development of the communications following the example in Section 6.1, and with $\eta = 2$. During the first iteration, the algorithm computes a unitary transformation Θ_1 that zeroes all of the elements in the first row of the generator except the first one, and then applies it to the rest of the block. This step is performed in processor P_0 (Figure 5). Then, the algorithm shifts the first column of the block one position down (Figure 6). The next iteration ($i = 2$) begins with the computation of the unitary transformation Θ_2 that zeroes all the entries of the second row except the first one, and then applies this transformation to the rest of the lower rows in the block (Figure 7). By x_i' we represent entries modified by Θ_2. After applying the second transformation, the first column of the block is again shifted one position down (Figure 8).

P_0	$\tilde{x}_1$ 0 0 0 0 0 0 0 0 0 0 0
	$\tilde{x}_2$ $\tilde{x}$ $\tilde{x}$ $\tilde{x}$ $\tilde{x}$ $\tilde{x}$ $\tilde{x}$ $\tilde{x}$ $\tilde{x}$ $\tilde{x}$ $\tilde{x}$ $\tilde{x}$
	$\tilde{x}_3$ $\tilde{x}$ $\tilde{x}$ $\tilde{x}$ $\tilde{x}$ $\tilde{x}$ $\tilde{x}$ $\tilde{x}$ $\tilde{x}$ $\tilde{x}$ $\tilde{x}$ $\tilde{x}$
	$\tilde{x}_4$ $\tilde{x}$ $\tilde{x}$ $\tilde{x}$ $\tilde{x}$ $\tilde{x}$ $\tilde{x}$ $\tilde{x}$ $\tilde{x}$ $\tilde{x}$ $\tilde{x}$ $\tilde{x}$

Figure 5. Example of the triangularization process with $\eta = 2$. After computing and applying Θ_1.

P_0		0 0 0 0 0 0 0 0 0 0 0 0
	$\downarrow$	$\tilde{x}_1$ $\tilde{x}$ $\tilde{x}$ $\tilde{x}$ $\tilde{x}$ $\tilde{x}$ $\tilde{x}$ $\tilde{x}$ $\tilde{x}$ $\tilde{x}$ $\tilde{x}$ $\tilde{x}$
		$\tilde{x}_2$ $\tilde{x}$ $\tilde{x}$ $\tilde{x}$ $\tilde{x}$ $\tilde{x}$ $\tilde{x}$ $\tilde{x}$ $\tilde{x}$ $\tilde{x}$ $\tilde{x}$ $\tilde{x}$
		$\tilde{x}_3$ $\tilde{x}$ $\tilde{x}$ $\tilde{x}$ $\tilde{x}$ $\tilde{x}$ $\tilde{x}$ $\tilde{x}$ $\tilde{x}$ $\tilde{x}$ $\tilde{x}$ $\tilde{x}$

Figure 6. Example of the triangularization process with $\eta = 2$. After shifting the first column of the actual block.

P_0	0 0 0 0 0 0 0 0 0 0 0 0
	$\tilde{x}_1'$ 0 0 0 0 0 0 0 0 0 0 0
	$\tilde{x}_2'$ $\tilde{x}'$ $\tilde{x}'$ $\tilde{x}'$ $\tilde{x}'$ $\tilde{x}'$ $\tilde{x}'$ $\tilde{x}'$ $\tilde{x}'$ $\tilde{x}'$ $\tilde{x}'$ $\tilde{x}'$
	$\tilde{x}_3'$ $\tilde{x}'$ $\tilde{x}'$ $\tilde{x}'$ $\tilde{x}'$ $\tilde{x}'$ $\tilde{x}'$ $\tilde{x}'$ $\tilde{x}'$ $\tilde{x}'$ $\tilde{x}'$ $\tilde{x}'$

Figure 7. Example of the triangularization process with $\eta = 2$. After computing and applying Θ_2.

		0	0	0	0	0	0	0	0	0	0	0	0
		0	0	0	0	0	0	0	0	0	0	0	0
P_0	$\downarrow$	$\tilde{x}'_1$	$\tilde{x}'$	$\tilde{x}'$	$\tilde{x}'$	$\tilde{x}'$	$\tilde{x}'$	$\tilde{x}'$	$\tilde{x}'$	$\tilde{x}'$	$\tilde{x}'$	$\tilde{x}'$	$\tilde{x}'$
		$\tilde{x}'_2$	$\tilde{x}'$	$\tilde{x}'$	$\tilde{x}'$	$\tilde{x}'$	$\tilde{x}'$	$\tilde{x}'$	$\tilde{x}'$	$\tilde{x}'$	$\tilde{x}'$	$\tilde{x}'$	$\tilde{x}'$

Figure 8. Example of the triangularization process with $\eta = 2$. After shifting the first column of the actual block.

	0	0	0	0	0	0	0	0	0	0	0	0
P_0	0	0	0	0	0	0	0	0	0	0	0	0
	$\tilde{x}'_1$	$\tilde{x}'$	$\tilde{x}'$	$\tilde{x}'$	$\tilde{x}'$	$\tilde{x}'$	$\tilde{x}'$	$\tilde{x}'$	$\tilde{x}'$	$\tilde{x}'$	$\tilde{x}'$	$\tilde{x}'$
	$\tilde{x}'_2$	$\tilde{x}'$	$\tilde{x}'$	$\tilde{x}'$	$\tilde{x}'$	$\tilde{x}'$	$\tilde{x}'$	$\tilde{x}'$	$\tilde{x}'$	$\tilde{x}'$	$\tilde{x}'$	$\tilde{x}'$
	0	$\tilde{x}'$	$\tilde{x}'$	$\tilde{x}'$	$\tilde{x}'$	$\tilde{x}'$	$\tilde{x}'$	$\tilde{x}'$	$\tilde{x}'$	$\tilde{x}'$	$\tilde{x}'$	$\tilde{x}'$
P_1	y_1	$\tilde{x}'$	$\tilde{x}'$	$\tilde{x}'$	$\tilde{x}'$	$\tilde{x}'$	$\tilde{x}'$	$\tilde{x}'$	$\tilde{x}'$	$\tilde{x}'$	$\tilde{x}'$	$\tilde{x}'$
	$\tilde{x}'_5$	$\tilde{x}'$	$\tilde{x}'$	$\tilde{x}'$	$\tilde{x}'$	$\tilde{x}'$	$\tilde{x}'$	$\tilde{x}'$	$\tilde{x}'$	$\tilde{x}'$	$\tilde{x}'$	$\tilde{x}'$
	$\tilde{x}'_6$	$\tilde{x}'$	$\tilde{x}'$	$\tilde{x}'$	$\tilde{x}'$	$\tilde{x}'$	$\tilde{x}'$	$\tilde{x}'$	$\tilde{x}'$	$\tilde{x}'$	$\tilde{x}'$	$\tilde{x}'$
	0	$\tilde{x}'$	$\tilde{x}'$	$\tilde{x}'$	$\tilde{x}'$	$\tilde{x}'$	$\tilde{x}'$	$\tilde{x}'$	$\tilde{x}'$	$\tilde{x}'$	$\tilde{x}'$	$\tilde{x}'$
P_2	y_2	$\tilde{x}'$	$\tilde{x}'$	$\tilde{x}'$	$\tilde{x}'$	$\tilde{x}'$	$\tilde{x}'$	$\tilde{x}'$	$\tilde{x}'$	$\tilde{x}'$	$\tilde{x}'$	$\tilde{x}'$
	$\tilde{x}'_9$	$\tilde{x}'$	$\tilde{x}'$	$\tilde{x}'$	$\tilde{x}'$	$\tilde{x}'$	$\tilde{x}'$	$\tilde{x}'$	$\tilde{x}'$	$\tilde{x}'$	$\tilde{x}'$	$\tilde{x}'$
	$\tilde{x}'_{10}$	$\tilde{x}'$	$\tilde{x}'$	$\tilde{x}'$	$\tilde{x}'$	$\tilde{x}'$	$\tilde{x}'$	$\tilde{x}'$	$\tilde{x}'$	$\tilde{x}'$	$\tilde{x}'$	$\tilde{x}'$

Figure 9. Example of triangularization process with $\eta = 2$. After updating rows 5 to 12 of $\hat{G}$ by processors P_1 and P_2.

The previous two iterations have been locally performed in P_0. Now, since $\eta = 2$, the parameters of transformations Θ_1 and Θ_2 are packed and broadcasted to the rest of the processors in the same message. The rest of the processors receive Θ_1 and Θ_2 and update their local rows, shifting the first column of each block one position down after each transformation. The form of the generator $\hat{G}$ after the first two iterations of the parallel algorithm is shown in Figure 9. Entries y_1 and y_2 represent new values produced by the algorithm, i.e., let $(\, x_5 \; x \; \cdots \; x \,)$ be the first row of P_1 before any computation; this row is modified as follows

$$\begin{aligned}
(\tilde{x}_5 \; \tilde{x} \; \cdots \; \tilde{x}) &\leftarrow (x_5 \; x \; \cdots \; x)\Theta_1 \\
(0 \; \tilde{x} \; \cdots \; \tilde{x}) &\leftarrow (\tilde{x}_5 \; \tilde{x} \; \cdots \; \tilde{x}) \text{ (shift)} \\
(y_1 \; \tilde{x}' \; \cdots \; \tilde{x}') &\leftarrow (0 \; \tilde{x} \; \cdots \; \tilde{x})\Theta_2.
\end{aligned}$$

The theoretical communication cost of this version of the parallel algorithm is given by

$$\eta\left(\beta + \frac{n}{\eta}(4\nu + 4)\alpha\right),$$

with $\eta \in [1, n/\nu]$. This version significantly reduces the communication cost of the parallel algorithm although it complicates the implementation.

Finally, the third step of the parallel algorithm (Algorithm 2) consists of four basic operations. A matrix-vector product $\hat{T}^T b = PT^T b$ (26) that can be carried out in parallel without any communication, because all processors have the first row and column of blocks of T and the array b. Each processor computes and saves its local rows of $PT^T b$ following the same distribution as $\hat{G}$ and L. The two triangular systems (27) and (28) are solved in parallel using a parallel PBLAS routine included in the parallel linear algebra package ScaLAPACK [4]. The last matrix-vector product (29) is performed in P_0 after gathering the elements of vector x from the rest of the processors. The last product is necessary only if T is block–Toeplitz matrix and not in the Toeplitz–block case.

7. Experimental analysis

7.1. Experimental environment

All the experiments presented in this section were performed on a cluster of 32 Intel Pentium-II processors (300 MHz, 128 MBytes of RAM), connected with a Myrinet network [5]. Tests with a specific version of MPI for this platform (GM-MPICH) showed a latency time of 33 μs. and a bandwidth of 33 Mbytes/s.

All the algorithms were implemented in Fortran 77. Several mathematical libraries were used. First, ScaLAPACK parallel library was used to distribute the data and to solve triangular systems of equations in parallel. Optimized versions of the sequential BLAS and LAPACK libraries were used to perform basic local operations on each processor.

7.2. Topology and data distribution

The ScaLAPACK parallel library includes routines which allow us to easily and efficiently manage data that is distributed onto a logical bi-dimensional mesh of processors. Each processor within the mesh is identified by a pair of coordinates and the data is distributed using a block cyclic bi-dimensional scheme.

Our algorithm uses a unidimensional $p \times 1$ mesh of processors. The generator $\hat{G}$ is cyclically distributed by row blocks in that unidimensional mesh. There is not much concurrency in the updating of each row, so a bi-dimensional mesh is not suitable. The use of a bi-dimensional mesh forces us to interchange columns during the updating process. This greatly increases the communication cost without taking much advantage of the additional concurrency because the number of columns of the generator is small.

Let us now discuss the effect of the memory access pattern on the performance of the algorithm. Routines included in the ScaLAPACK library work on entries of matrices stored consecutively in memory by columns. This layout is very inefficient in some phases of our algorithm, specifically during the computation Θ_i and the updating of each row. Memory access has a great impact on the execution time; therefore, care must be taken with the arrangement of the elements in memory in order to optimize the use of fast cache memories.

In order to reduce the memory access time in our algorithm, we use two different logical topologies in steps 2 and 3 of Algorithm 2 (Figures 10 and 11). In step 1, we distribute $\hat{G}^T$

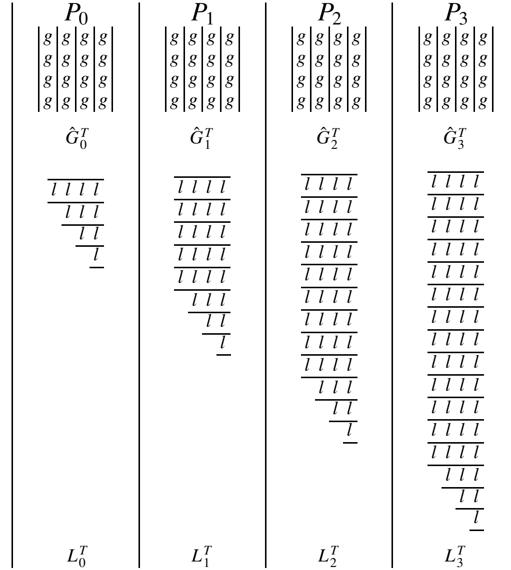

Figure 10. Distribution of $\hat{G}$ and L in steps 1 and 2 of Algorithm 2.

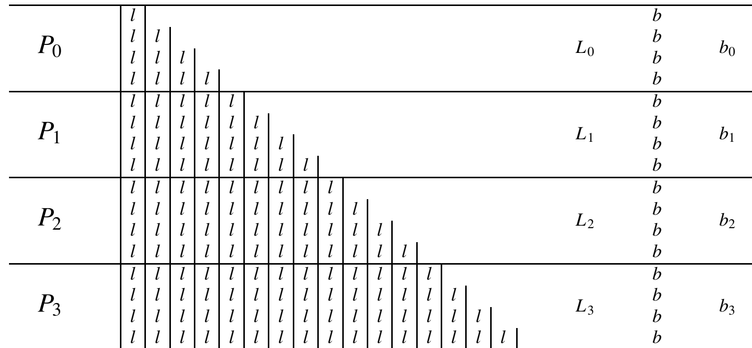

Figure 11. Distribution of the triangular factor L and vector b in step 3 of Algorithm 2.

in a *logical row* of processors instead of distributing $\hat{G}$ in a *logical column* as shown in the theoretical presentation in the previous sections. Thus, the row entries of the generator $\hat{G}$ are stored in consecutive memory positions. In Figure 10, all elements between lines are consecutively stored in memory. In each iteration of step 2 of Algorithm 2, a new column of the triangular factor L is computed. This column is stored in memory as if the topology were a *logical column* of processors instead of a *logical row* in order to allow access to adjacent elements in memory. With this modification, both the entries of $\hat{G}$ and the entries of L are stored in consecutive memory positions.

After step 2, we need to change the logical grid from a *logical row* of processors $(1 \times p)$ to a *logical column* $(p \times 1)$. This is necessary because PBLAS routines need to have L

stored in memory as shown in Figure 11. The change of topology is carried out by using the ScaLAPACK concept of *context*. The p processors involved in the computation belong to two different contexts at the same time. In one context, the processors are arranged in a row while, in the other context, the processors are arranged in a column. The change from one context to the other does not imply computation nor communication cost, but only a change in the coordinated notation of each processor within the logical grid. Note that the distribution of factor L is the same in both figures; the only difference is that the second figure represents L while the first one represents its transpose.

7.3. *Test matrices*

The experimental results shown in the following sections were obtained with KMS (Kac-Murdock-Szegö) scalar matrices and with randomly generated scalar matrices. KMS matrices are defined as

$$t_0 = \epsilon, \quad t_i = t_{-i} = \left(\frac{1}{2}\right)^i, \quad i = 1, 2, \ldots, n, \tag{31}$$

where t_i and t_{-i} are the values of the first row and column of a Toeplitz matrix $T \in \mathbf{R}^{n\times n}$, respectively. If ϵ is small (i.e. $\epsilon = 10^{-14}$), the leading sub-matrices of order $3m + 1$ ($m = 0, 1, \ldots$) are very ill-conditioned.

Scalar Toeplitz matrices can be seen as (2) and (3) matrices with an arbitrary block size ν. In the experiments, we took the same scalar matrices as block matrices with different block sizes. Matrices preserve conditioning and regularity despite the block size.

7.4. *Results of the sequential algorithm*

Our first experimental analysis shows the relative weight of each of the three main steps of Algorithm 2 in only one processor. Table 1 includes time in seconds of each step for a fixed matrix dimension using different block sizes. These times show that the most costly step is devoted to the factorization of the matrix (Algorithm 3), while the computation of the generator represents a small part of the global cost. The third step, depends only on the matrix size and not on the block size; this is why the run time does not vary with the block size.

7.5. *Parallelization of each step of the algorithm*

Table 2 shows the time (speedup) of steps 1 and 2 of Algorithm 2 with different block sizes. Computing the generator (step 1) is a highly parallel process that does not involve communications. The efficiency of this step is only reduced by the computation of factor R (30), which is replicated in each processor. The results for step 2 were taken with the best message size η in each case. Step 2 of the algorithm obtained very good results when we increased the block size, producing super-speedups in some cases. Super-speedups can

Table 1. Time in seconds of each step of Algorithm 2 in one processor with different block sizes ν ($n = 1080$)

ν	Step 1	Step 2	Step 3	Total
2	0.12 (11.2%)	0.88 (82.2%)	0.07 (6.5%)	1.07
4	0.14 (7.9%)	1.56 (88.1%)	0.07 (4.0%)	1.77
6	0.27 (9.2%)	2.58 (88.4%)	0.07 (2.4%)	2.92
8	0.45 (14.2%)	2.64 (83.5%)	0.07 (2.2%)	3.16
10	0.40 (12.2%)	2.82 (85.7%)	0.07 (2.1%)	3.29
15	0.59 (14.6%)	3.38 (83.7%)	0.07 (1.7%)	4.04
20	0.67 (13.6%)	4.19 (85.0%)	0.07 (1.4%)	4.93
30	1.08 (14.2%)	6.48 (84.9%)	0.07 (0.9%)	7.63

Table 2. Time in seconds of steps 1 and 2 of Algorithm 2 with different block sizes and processors

	Step 1				Step 2			
ν	1	2	4	8	1	2	4	8
10	0.40	0.20 (2.00)	0.13 (3.08)	0.09 (4.44)	2.84	1.51 (1.88)	0.89 (3.19)	0.60 (4.73)
15	0.59	0.32 (1.84)	0.17 (3.47)	0.09 (6.56)	3.49	1.78 (1.96)	1.00 (3.49)	0.62 (5.63)
20	0.66	0.34 (1.94)	0.18 (3.67)	0.12 (5.50)	4.09	1.94 (2.11)	1.07 (3.82)	0.66 (6.20)
30	1.06	0.53 (2.00)	0.31 (3.42)	0.18 (5.89)	6.33	2.60 (2.43)	1.33 (4.76)	0.79 (8.01)
40	1.59	0.66 (2.41)	0.37 (4.30)	0.23 (6.91)	8.16	3.58 (2.28)	1.61 (5.07)	0.93 (8.77)
45	1.89	0.77 (2.45)	0.45 (4.20)	0.28 (6.75)	9.36	4.14 (2.26)	1.73 (5.41)	1.02 (9.18)

be expected when we deal with low-cost algorithms where the memory access and the use of the cache have a large impact on the cost. If one processor is used to solve the problem, this processor has to access about $n^2/2$ elements in memory; if p processors are used to solve the problem, each one only has to access approximately $n^2/(2 \times p)$ elements, thereby reducing the probability of cache misses.

7.6. Optimization of the communications

Figure 12 shows the run time of the parallel algorithm as a function of η. This parameter represents the number of unitary transformations packed in each message in the Schur triangularization. The optimal choice for this parameter is found to be in the range 1 and n/ν and, as we showed in the previous figure, it is independent of the number of processors. The optimal value depends on the relation between the size of the problem n and the block size ν. Our experiments show that this value can be approximated by

$$\eta \approx \frac{n}{3\nu},$$

for the target machine used in the experiments.

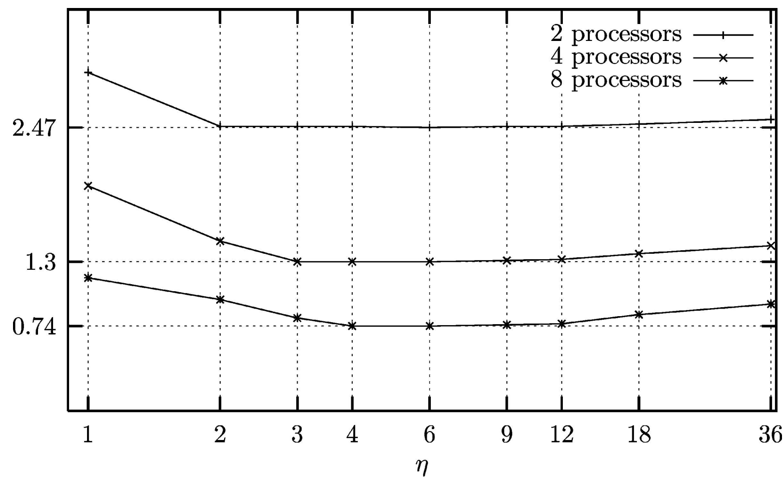

Figure 12. Time in seconds of step 2 of Algorithm 2 varying the number of transformations per message η. ($n = 1080$ and $\nu = 30$).

7.7. Results of the parallel algorithm

The maximum number of processors that can be used in our algorithm is given by the number of blocks of the permuted matrix (3). This value is ν, which is the block size of the original block–Toeplitz matrix (2).

Figure 13 shows the run time of the parallel algorithm as a function of the block size ν and the number of processors. The time spent by the algorithm greatly decreases when we increase the number of processors, and this speedup is larger for large block sizes. Although the number of columns of the generator grows with the block size, this additional cost is compensated by the possibility of using more processors. Figure 13 also shows that the minimum time of the algorithm for different values of ν is quite similar.

The experimental analysis shows that it might be interesting to use the parallel algorithm even with scalar Toeplitz matrices. The value 0.86 s. shown in Figure 13 represents the time spent by the sequential GSA to solve a scalar system of dimension $n = 1080$. Since a scalar Toeplitz matrix can also be seen as a block–Toeplitz matrix, if we choose the appropriate block size and the number of processors, the parallel algorithm for block–Toeplitz systems improves even the sequential algorithm for scalar matrices.

Table 3 shows the efficiency of the parallel algorithm with two different block sizes ν that are larger than the number of available processors ($p = 32$). The results are better with a large block size because the parallelism of the algorithm increases with the number of columns of the generator. Even with relatively small matrices ($n = 360$), the parallel algorithm obtains a good efficiency, taking into account the small computational cost ($O(n^2)$ operations). These results confirm that we have greatly reduced the effect of the communication cost of the parallel algorithm.

Table 3. Efficiency of the parallel algorithm with different problem and block sizes

n	360		720		1080		1440	
ν	36 (%)	40 (%)	36 (%)	40 (%)	36 (%)	40 (%)	36 (%)	40 (%)
2	98.6	96.2	123.6	125.7	123.5	122.9	112.8	111.2
4	80.7	81.5	110.9	116.5	121.2	122.0	119.4	120.3
8	59.2	62.5	90.1	95.5	99.6	104.7	98.4	109.5
16	34.1	39.1	61.8	68.2	74.4	77.5	71.4	82.5
32	17.1	19.5	36.0	39.8	45.4	47.6	42.1	52.1

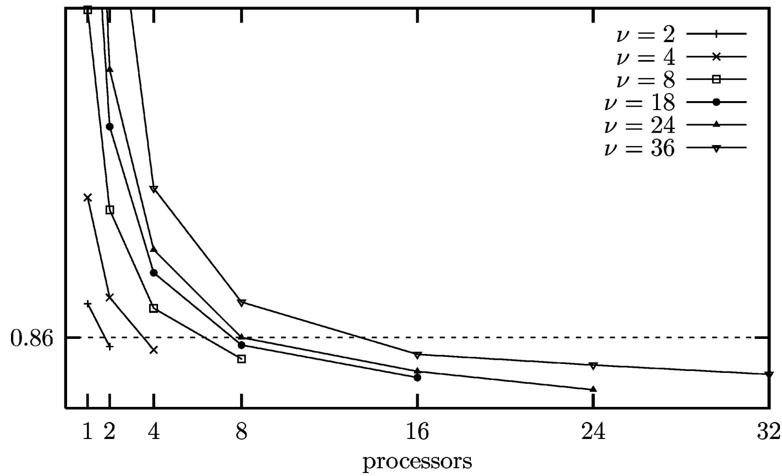

Figure 13. Time in seconds of the parallel algorithm for different block sizes ν. ($n = 1080$).

7.8. Precision of the results

The accuracy of the solution was measured using the relative error of the system solution given by

$$\frac{\|\tilde{x} - x\|}{\|x\|} \tag{32}$$

where vector x in (1) has all its elements equal to 1 and $\tilde{x}$ is the computed solution. Table 4 shows the value of ε for the KMS matrix, the condition number of the system matrix T, the condition number of the product $T^T T$ and the relative error with different block sizes. It follows that, when $\varepsilon = 10^{-14}$, we obtain worse results due to the condition number of $T^T T$, which is $\kappa(T^T T) \approx \kappa^2(T)$. However, the precision of the solution does not depend on the regularity of T, because the algorithm works on the product $T^T T$, and a positive definite

Table 4. Relative error with different block sizes for KMS matrices of dimension $n = 128$

	T_1	T_2	T_3
ε	10^{-14}	1.0	10^5
$\kappa(T)$	211.51	9.0	1.0
$\kappa(T^T T)$	0.45×10^5	80.8	1.0
$\nu = 1$	9.80×10^{-14}	5.69×10^{-15}	2.43×10^{-16}
$\nu = 2$	4.40×10^{-13}	8.37×10^{-15}	4.33×10^{-16}
$\nu = 4$	2.28×10^{-13}	7.56×10^{-15}	4.43×10^{-16}
$\nu = 8$	1.63×10^{-13}	5.34×10^{-15}	6.20×10^{-16}
$\nu = 16$	1.71×10^{-13}	6.13×10^{-15}	6.84×10^{-16}
$\nu = 32$	4.69×10^{-13}	8.02×10^{-15}	5.84×10^{-16}

Table 5. Relative error with different block sizes for KMS matrices of dimension $n = 128$ and one iteration of iterative refinement

	T_1	T_2	T_3
ε	10^{-14}	1.0	10^5
$\kappa(T)$	211.51	9.0	1.0
$\kappa(T^T T)$	0.45×10^5	80.8	1.0
$\nu = 1$	7.05×10^{-15}	5.51×10^{-16}	1.86×10^{-16}
$\nu = 2$	7.60×10^{-15}	4.23×10^{-16}	2.19×10^{-17}
$\nu = 4$	8.70×10^{-16}	3.18×10^{-16}	0.0
$\nu = 8$	1.10×10^{-15}	1.97×10^{-16}	0.0
$\nu = 16$	1.34×10^{-15}	1.45×10^{-16}	0.0
$\nu = 32$	1.94×10^{-15}	8.83×10^{-17}	0.0

matrix is strongly regular. Recall that classical *fast* algorithms are only stable with strongly regular matrices, at least if no additional technique is used to stabilize the algorithm.

Our parallel algorithm offers the possibility of improving the accuracy of the solution by applying some iterations of *iterative refinement*, as shown in Section 5. This post-process implies a cost that is equivalent to step 3 of Algorithm 2 (Table 1); that is, two matrix–vector products and the solution of two triangular systems. Table 5 shows the relative error in the same cases as in Table 4 after applying one iterative refinement step. The precision is improved with every class of matrix.

Table 6 shows the relative error with randomly generated block–Toeplitz matrices. The elements of these matrices are uniformly distributed among -1.0 and 1.0. The table also shows the condition number of matrix T and the matrix product $T^T T$. In all cases, $\kappa(T) \approx \kappa(T^T T)$ holds. Table 6 shows the precision of the solution after 0, 1 and 2 iterations of iterative refinement. It can be seen that the accuracy of the solution depends on the condition number of the matrix (or the product $T^T T$). However, by applying only one iterative refinement step, the precision of the result is always reduced to values that are very similar to the machine precision ($\approx 0.22 \times 10^{-15}$). Indeed, the application of more refinement steps could produce even worse results.

Table 6. Relative error with different block sizes and different numbers of refinement steps for randomly generated matrices of dimension $n = 128$

	$\kappa(T)$	$\kappa(T^T T)$	0	1	2
$\nu = 1$	104.57	1.09×10^4	0.75×10^{-13}	0.15×10^{-14}	0.84×10^{-15}
$\nu = 2$	118.57	1.41×10^4	0.23×10^{-12}	0.97×10^{-15}	0.99×10^{-15}
$\nu = 4$	85.95	7.39×10^3	0.28×10^{-10}	0.37×10^{-14}	0.62×10^{-14}
$\nu = 8$	161.69	2.61×10^4	0.31×10^{-12}	0.61×10^{-15}	0.60×10^{-15}
$\nu = 16$	3685.9	1.36×10^7	0.11×10^{-9}	0.97×10^{-15}	0.13×10^{-14}
$\nu = 32$	858.56	7.37×10^5	0.61×10^{-11}	0.25×10^{-14}	0.26×10^{-14}

8. Conclusions

Algorithms that exploit the displacement structure property of a matrix are called *fast* algorithms and have a cost of order $O(n^2)$. The parallelization of algorithm of this kind in multicomputers is difficult due to the influence of the communications. In the case of fast algorithms to solve Toeplitz systems, this problem is even worse due to the strong dependency among the successive steps of the algorithms.

In this paper, we present an efficient parallel algorithm to solve block–Toeplitz and Toeplitz–block linear systems of equations based on the GSA. Our algorithm works with the second type of matrices, while previous approaches work on the first type.

The two main contributions of our parallel algorithm consist of reducing the communication cost and using an appropriate memory access scheme. We have considerably reduced the communication cost by using a data distribution that allows us to perform the shift of the first column of the generator locally on each processor. Therefore, we avoid a lot of communications and reduce the number of messages to only one broadcast per iteration. Moreover, we have reduced the cost of the communications by means of a suitable packing of transformations that allows us to reduce the number of messages transmitted in the computation.

Our algorithm improves the memory access pattern by using the best logical topology in each step of the algorithm. This adaptation of the topology can be obtained by using different ScaLAPACK contexts. The use of different topologies allows the algorithm to access as many consecutive positions in memory as possible, and better exploits the speed of the cache memories.

The experimental analysis on a cluster of personal computers shows good efficiencies even with 32 processors. The accuracy of the solution is good and it is independent of the regularity of the matrix. It can also be improved by applying iterative refinement.

Furthermore, the parallel algorithm is portable because it is implemented using standard tools and libraries such as LAPACK and ScaLAPACK.

Acknowledgment

This work is supported by the Spanish grant CICYT TIC 2000-1683-C03-03.

References

1. V. M. Adukov. Generalized inversion of block Toeplitz matrices. *Linear Algebra and its Applications* 274(1–3):85–124, 1998.
2. 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*, 2nd ed. Philadelphia, SIAM, 1995.
3. C. Bischof and C. Van Loan. The WY representation for products of Householder matrices. *SIAM Journal on Scientific and Statistical Computing*, 8(1):S2–S13, 1987. Parallel processing for scientific computing (Norfolk, Va., 1985).
4. L. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. Whaley, *ScaLAPACK Users' Guide*, Philadelphia, SIAM, 1997.
5. N. Boden, D. Cohen, R. Felderman, A. Kulawik, C. Seitz, J. Seizovic, and W. Su. Myrinet. A Gigabit-per-Second Local-Area Network. *IEEE Micro*, 15:29–36, 1995.
6. A. Bojanczyk, R. P. Brent, and F. de Hoog, A weakly stable algorithm for general toeplitz systems. Technical Report TR-CS-93-15, Laboratory for Computer Science, Australian National University, Canberra, Australia. Revised June 1994.
7. A. W. Bojanczyk, R. P. Brent, and F. R. de Hoog. Stability analysis of a general Toeplitz systems solver. *Numerical Algorithms*, 10(3/4):225–244, 1995.
8. A. W. Bojanczyk, R. P. Brent, F. R. de Hoog, and D. R. Sweet. On the stability of the Bareiss and related Toeplitz factorization algorithms. *SIAM Journal on Matrix Analysis and Applications*, 16(1):40–57, 1995.
9. J. R. Bunch. The Weak and strong stability of algorithms in numerical linear algebra. *Linear Algebra and its Applications*, 88/89:49–66, 1987.
10. S. Cabay, A. R. Jones, and G. Labahn, Computation of numerical Padé-Hermite and simultaneous Padé systems. I. Near inversion of generalized Sylvester matrices. *SIAM Journal on Matrix Analysis and Applications*, 17(2):248–267, 1996.
11. S. Cabay, A. R. Jones, and G. Labahn. Computation of numerical Padé-Hermite and simultaneous Padé systems. II. A weakly stable algorithm. *SIAM Journal on Matrix Analysis and Applications*, 17(2):268–297, 1996.
12. S. Cabay, A. R. Jones, and G. Labahn, Algorithm 766: Experiments with a weakly stable algorithm for computing padé and simultaneous padé approximants. *ACM Transactions on Mathematical Software*, 23(1):91–110, 1997.
13. S. Chandrasekaran and A. H. Sayed. A fast stable solver for nonsymmetric toeplitz and quasi-toeplitz systems of linear equations. *SIAM Journal on Matrix Analysis and Applications*, 19(1):107–139, 1998.
14. E. de Doncker and J. Kapenga. Parallelization of Toeplitz solvers. In G. H. Golub and P. V. Dooren, eds. *Numerical Linear Algebra, Digital Signal Processing and Parallel Algorithms*, No. 70 in Computer and systems sciencies. Springer-Verlag, pp. 467–476, 1990.
15. D. J. Evans and G. Oka. Parallel solution of symmetric positive definite Toeplitz systems. *Parallel Algorithms and Applications*, 12(9):297–303, 1998.
16. K. Gallivan, S. Thirumalai, and P. V. Dooren. On solving block toeplitz systems using a block schur algorithm. In J. Chandra, ed. *Proceedings of the 23rd International Conference on Parallel Processing. Volume 3: Algorithms and Applications*. Boca Raton, FL, USA, pp. 274–281, 1994.
17. K. A. Gallivan, S. Thirumalai, P. V. Dooren, and V. Vermaut. High performance algorithms for Toeplitz and block toeplitz matrices. *Linear Algebra and its Applications*, 241/243(1–3):343–388, 1996. In *Proceedings of the Fourth Conference of the International Linear Algebra Society* (Rotterdam, 1994).
18. L. Gemignani. Schur complements of bezoutians and the inversion of block hankel and block toeplitz matrices. *Linear Algebra and its Applications*, 253(1–3):39–59, 1997.
19. I. Gohberg, I. Koltracht, A. Averbuch, and B. Shoham. Timing analysis of a parallel algorithm for Toeplitz matrices on a MIMD parallel machine. *Parallel Computing*, 17(4/5):563–577, 1991.
20. G. H. Golub and C. F. V. Loan. Matrix computations. *Johns Hopkins Studies in the Mathematical Sciences*. The Johns Hopkins University Press, Baltimore, MD, USA, 1996.
21. T. Kailath and J. Chun. Generalized displacement structure for block-toeplitz, toeplitz-block, and toeplitz-derived matrices. *SIAM Journal on Matrix Analysis and Applications*, 15(1):114–128.

22. T. Kailath and A. H. Sayed. Displacement structure: Theory and applications. *SIAM Review*, 37(3):297–386, 1995.
23. P. Kravanja and M. V. Barel. A fast block Hankel solver based on an inversion formula for block Loewner matrices. *Calcolo*, 33:147–164, 1996.
24. S. Y. Kung and Y. H. Hu. A highly concurrent algorithm and pipelined architecture for solving toeplitz systems. *IEEE Trans. Acoustics, Speech and Signal Processing*, ASSP-31(1):66, 1983.
25. S. Y. Kung, H. J. Whitehouse, and T. Kailath (eds.). *VLSI and Modem Signal Processing (Los Angeles, CA, November 1–3, 1982)*. Prentice-Hall, Englewood Cliffs, NJ, 1985.
26. G. Labahn, D. K. Choi, and S. Cabay. The inverses of block hankel and block toeplitz matrices. *SIAM Journal on Computing*, 19(1):98–123, 1990.
27. V. Y. Pan. Concurrent iterative algorithm for Toeplitz-like linear systems. *IEEE Transactions on Parallel and Distributed Systems*, 4(5):592–600, 1993.
28. R. Schreiber and C. Van Loan. A storage-efficient WY representation for products of Householder transformations. *SIAM Journal on Scientific and Statistical Computing*, 10(1):53–57, 1989.
29. M. Stewart and P. Van Dooren, Stability issues in the factorization of structured matrices. *SIAM Journal on Matrix Analysis and Applications*, 18(1):104–118, 1997.
30. D. R. Sweet. Fast Toeplitz orthogonalization. *Numerische Mathematik*, 43(1):1–21, 1984.
31. D. R. Sweet. The Use of Linear-time Systolic Algorithms for the solution of Toeplitz Problems. Technical Report JCU-CS-91/1, Department of Computer Science, James Cook University. Tue, 15:17:55 GMT, 1991.
32. 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.
33. M. Van Barel and A. Bultheel. A lookahead algorithm for the solution of block Toeplitz systems. *Linear Algebra and its Applications*, 266(1–3):291–335, 1997.
34. M. Wax and T. Kailath. Efficient inversion of toeplitz-block toeplitz matrix. *IEEE Trans. Acoustics, Speech and Signal Processing*, ASSP-31(5):1218, 1983.