An Efficient and Stable Parallel Solution for Non-symmetric Toeplitz Linear Systems*

Pedro Alonso[1], José M. Badía[2], and Antonio M. Vidal[1]

[1] Universidad Politécnica de Valencia,
cno. Vera s/n, 46022 Valencia, Spain
{palonso, avidal}@dsic.upv.es

[2] Universitat Jaume I, Campus de Riu Sec,
12071 Castellón de la Plana, Spain
badia@icc.uji.es

Abstract. In this paper, we parallelize a new algorithm for solving non–symmetric Toeplitz linear systems. This algorithm embeds the Toeplitz matrix in a larger structured matrix, then transforms it into an embedded Cauchy–like matrix by means of trigonometric modifications. Finally, the algorithm applies a modified QR transformation to triangularize the augmented matrix. The algorithm combines efficiency and stability. It has been implemented using standard tools and libraries, thereby producing a portable code. An extensive experimental analysis has been performed on a cluster of personal computers. Experimental results show that we can obtain efficiencies that are similar to other fast parallel algorithms, while obtaining more accurate results with only one iterative refinement step in the solution.

Keywords: Non–symmetric Toeplitz, Linear Systems, Displacement Structure, Cauchy–like matrices, Parallel Algorithms, Clusters.

1 Introduction

In this paper, we present a new parallel algorithm for the solution of Toeplitz linear systems such as

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

is presented, where the Toeplitz matrix $T \in \mathrm{R}^{n\times n}$ has the form $T = (t_{ij}) = (t_{i-j})_{i,j=0}^{n-1}$, and where $b, x \in \mathrm{R}^n$ are the independent and the solution vector, respectively.

Many *fast* algorithms that solve this problem can be found in the literature; that is, algorithms whose computational cost is $O(n^2)$. Almost all these algorithms produce poor results unless strongly regular matrices are used; that is, matrices whose leading submatrices are all well–conditioned. Several methods

* Supported by Spanish projects CICYT TIC 2000-1683-C03-03 2003-08238-C02-02.

M. Daydé et al. (Eds.): VECPAR 2004, LNCS 3402, pp. 685–698, 2005.
© Springer-Verlag Berlin Heidelberg 2005

have been proposed to improve the solution of (1), including the use of *look-ahead* or refinement techniques [1, 2].

It is difficult to obtain efficient parallel versions of *fast* algorithms, because they have a reduced computational cost and they also have many dependencies among fine–grain operations. These dependencies produce many communications, which are a critical factor in obtaining efficient parallel algorithms, especially on distributed memory computers. This problem could partially explain the small number of parallel algorithms that exist for dealing with Toeplitz linear systems. For instance, parallel algorithms that use systolic arrays to solve Toeplitz systems can be found in [3, 4]. Other parallel algorithms deal only with positive definite matrices (which are strongly regular) [5], or with symmetric matrices [6]. There also exist parallel algorithms for shared memory multiprocessors [7, 8, 9] and several parallel algorithms have recently been proposed for distributed architectures [10].

If we apply some of the techniques to improve the stability of the sequential algorithms that solve (1), it is more difficult to obtain efficient parallel versions. In this work, we combine the advantages of two important results from [2, 10] in order to derive a parallel algorithm that exploits the special structure of the Toeplitz matrices and is both efficient and stable.

Another of our main goals is to offer an efficient parallel algorithm for general purpose architectures, i.e. clusters of personal computers. The algorithm presented in this paper is portable because it is based on the libraries LAPACK [11] and ScaLAPACK [12] that are sequential and parallel, respectively.

In [10], we proposed two parallel algorithms for the solution of Toeplitz linear systems. The first algorithm, called PAQR, performs a QR decomposition of T. This decomposition is *m*odified by a correcting factor that increases the orthogonality of the factor Q, so that the algorithm is backward stable and the solution is more accurate. The main drawback of this algorithm is that it produces poor speed–up's for more than two processors. In the second algorithm, called PALU, the Toeplitz matrix is transformed into a Cauchy–like matrix by means of discrete trigonometric transforms. This algorithm obtains better speed–up's than the PAQR algorithm, but produces results that are not very accurate. In this paper, we present a new parallel algorithm that combines the efficiency of the PALU algorithm with the stability and precision of the PAQR algorithm.

The structure of this paper is the following. In Section 2, we present the sequential algorithm. In Section 3 we show how to perform the triangular decomposition of a Cauchy–like matrix. In Section 4, we describe the parallel algorithm. The experimental results are shown in Section 5, and our conclusions are presented in the last section.

2 The Sequential Algorithm

The algorithm proposed and parallelized in this paper performs a *modified* QR decomposition in order to solve the Toeplitz system (1). This *modified* QR decomposition is based on the idea set out in [2] and applied in the PAQR parallel

algorithm mentioned in the previous section. The *modified* QR decomposition can be described as follows.

An augmented matrix M is constructed with the following decomposition

$$M = \begin{pmatrix} T^T T & T^T \\ T & 0 \end{pmatrix} = \begin{pmatrix} R^T & 0 \\ Q & \Delta \end{pmatrix} \begin{pmatrix} R & Q^T \\ 0 & -\Delta^T \end{pmatrix} , \tag{2}$$

where R^T and Δ are lower triangular matrices and Q is a square matrix. Using exact arithmetic, the factors Q and R in (2) form the QR decomposition of T ($T = QR$); however, the computed factor Q may actually lose its orthogonality. By introducing the correcting factor Δ, the product $(\Delta^{-1}Q)$ is almost orthogonal. We will refer to the decomposition $T = \Delta(\Delta^{-1}Q)R$ as *modified* QR decomposition. This factorization is then used to obtain x in the linear system (1) by using $x = R^{-1}(Q^T\Delta^{-T})\Delta^{-1}b$.

The matrix M is a *structured* matrix, which means that a *fast* algorithm can be used to compute the triangular factorization shown in (2). One of these algorithms is the Generalized Schur Algorithm. In [10], we proposed a parallel version of the Generalized Schur Algorithm, but its scalability for more than two processors is not very good. This fact is mainly due to the form of the so–called *displacement matrices* used. To improve its efficiency, we apply the same transformation technique used in the PALU algorithm to M in order to work with diagonal displacement matrices. This kind of displacement matrix allows us to avoid a large number of the communications during the application of the parallel algorithm.

Structured matrices are characterized by the *displacement rank* property [13]. We start from the following displacement representation of matrix M (2) with respect to a displacement matrix F,

$$\nabla_{M,F} = FM - MF = GHG^T \ , \quad F = Z_0 \oplus Z_1 \ , \tag{3}$$

where $Z_\epsilon \in \mathrm{R}^{n\times n}$ is a zero matrix whose first superdiagonal and subdiagonal are equal to one and whose first and last diagonal entries are equal to ϵ. The rank of $\nabla_{M,F}$ (3) is 8. Matrices $G \in \mathrm{R}^{2n\times 8}$ and $H \in \mathrm{R}^{8\times 8}$ are called a *generator pair.*

First, we define

$$\hat{\mathcal{S}} = \sin\frac{ij\pi}{n+1} \ , i,j = 1,\ldots,n, \quad \text{and} \quad \mathcal{S} = \sqrt{\frac{2}{n+1}}\,\hat{\mathcal{S}} \ ,$$

as the unnormalized and the normalized Discrete Sine Transforms (DST), respectively. Matrix $\mathcal{S}$ is symmetric and orthogonal. Matrix Z_0 can be diagonalized by means of the DST $\mathcal{S}$; that is, $\mathcal{S}Z_0\mathcal{S} = \Lambda_0$, where Λ is diagonal.

In the same way, we define

$$\hat{\mathcal{C}} = \xi_j \cos\frac{(2i+1)j\pi}{2n} \ , i,j = 0,\ldots,n-1, \quad \text{and} \quad \mathcal{C} = \sqrt{\frac{2}{n}}\,\hat{\mathcal{C}} \ ,$$

where $\xi_j = 1/\sqrt{2}$ for $j = 0$, and $\xi_j = 1$ otherwise, as the unnormalized and the normalized Discrete Cosine Transforms (DCT), respectively. Matrix $\mathcal{C}$ is non–symmetric and orthogonal. Matrix Z_1 can be diagonalized by means of the DCT $\mathcal{C}$; that is, $\mathcal{C}^T Z_1 \mathcal{C} = \Lambda_1$, where Λ_1 is diagonal.

There exist four distinct versions of DST and DCT which are numbered as I, II, III and IV [14, 15]. Transforms DST and DCT, as defined in this paper, correspond to DST-I and DCT-II, respectively. We also exploit *superfast* algorithms that perform the transformation of a vector by DST or DCT in $O(n \log n)$ operations.

Now, we construct the following transformation

$$S = \begin{pmatrix} \mathcal{S} & 0 \\ 0 & \mathcal{C} \end{pmatrix} . \tag{4}$$

Pre– and post–multiplying equation (3) by S leads us to a displacement representation of M with diagonal displacement matrices.

$$\begin{aligned} S^T(FM - MF)S &= S^T(GHG^T)S , \\ (S^TFS)(S^TMS) - (S^TMS)(S^TFS) &= (S^TG)H(S^TG)^T , \\ \Lambda\hat{C} - \hat{C}\Lambda &= \hat{G}H\hat{G}^T , \end{aligned} \tag{5}$$

where $\Lambda = \begin{pmatrix} \Lambda_0 \oplus \Lambda_1 \end{pmatrix}$, and $\hat{C}$ has the form

$$\hat{C} = \begin{pmatrix} C^TC & C^T \\ C & 0 \end{pmatrix} = \begin{pmatrix} \mathcal{S} & 0 \\ 0 & \mathcal{C}^T \end{pmatrix} \begin{pmatrix} T^TT & T^T \\ T & 0 \end{pmatrix} \begin{pmatrix} \mathcal{S} & 0 \\ 0 & \mathcal{C} \end{pmatrix} ,$$

with $C = \mathcal{C}^T T \mathcal{S}$. It is easy to see that, by means of DST and DCT, we obtain a Cauchy–like linear system from the Toeplitz linear system (1)

$$(\mathcal{C}^T T \mathcal{S})(\mathcal{S}x) = (\mathcal{C}^T b) \ \rightarrow \ C\hat{x} = \hat{b} . \tag{6}$$

Matrices C and $\hat{C}$ are called *Cauchy–like* matrices and are also structured. The advantage of using *Cauchy–like* matrices is that the diagonal form of the displacement matrix Λ allows us to avoid a great number of communications.

We apply these ideas in the following algorithm to solve (1) in three steps.

1. The computation of generator G (3) and $\hat{G} = S^TG$, where S is defined in (4).
2. The computation of the triangular decomposition of matrix $\hat{C}$,

$$\hat{C} = \begin{pmatrix} C^TC & C^T \\ C & 0 \end{pmatrix} = \begin{pmatrix} \hat{R}^T & 0 \\ \hat{Q} & \hat{\Delta} \end{pmatrix} \begin{pmatrix} \hat{R} & \hat{Q}^T \\ 0 & -\hat{\Delta}^T \end{pmatrix} , \tag{7}$$

   The embedded factors $\hat{Q}$, $\hat{R}$ and $\hat{\Delta}$ represent the modified QR decomposition of matrix C, $C = \hat{\Delta}(\hat{\Delta}^{-1}\hat{Q})\hat{R}$.
3. The computation of the solution of the system (6)

$$\hat{x} = \hat{R}^{-1}(\hat{Q}^T\hat{\Delta}^{-T})\hat{\Delta}^{-1}\hat{b} , \tag{8}$$

   and the solution of the Toeplitz system (1), $x = \mathcal{S}\hat{x}$.

The form of generator pair (G, H) (3) can be derived analytically as follows: Given the following partitions of T and $T^T T$,

$$T = \begin{pmatrix} t_0 & v^T \\ u & T_0 \end{pmatrix} = \begin{pmatrix} T_0 & \bar{v} \\ \bar{u}^T & t_0 \end{pmatrix} , \quad T^T T = \begin{pmatrix} s_0 & w_0^T \\ w_0 & S_0 \end{pmatrix} = \begin{pmatrix} S_1 & w_1 \\ w_1^T & s_1 \end{pmatrix} ,$$

where a vector $\bar{x}$ represents the vector $x = \begin{pmatrix} x_0 \, x_1 \cdots x_{n-2} \, x_{n-1} \end{pmatrix}^T$ with the elements placed in the reverse order $\bar{x} = \begin{pmatrix} x_{n-1} \, x_{n-2} \cdots x_1 \, x_0 \end{pmatrix}^T$, and given the following partitions for the resulting vectors u, v, w_0 and w_1, in the above partitions,

$$u = \begin{pmatrix} t_1 \\ u' \end{pmatrix} = \begin{pmatrix} u'' \\ t_{n-1} \end{pmatrix} , \quad v = \begin{pmatrix} t_{-1} \\ v' \end{pmatrix} = \begin{pmatrix} v'' \\ t_{-n+1} \end{pmatrix} ,$$

$$w_0 = \begin{pmatrix} \alpha \\ w_0' \end{pmatrix} , \quad w_1 = \begin{pmatrix} w_1' \\ \beta \end{pmatrix} ,$$

we obtain the following generator pair

$$G = \begin{pmatrix} 0 & 0 & 0 & t_{-1} & 1 & 0 & t_{n-1} & 0 \\ w_0' & w_1' & \bar{u}' & v' & \hat{0} & \hat{0} & \bar{u}'' & v'' \\ 0 & 0 & t_1 & 0 & 0 & 1 & 0 & t_{-n+1} \\ t_0 + t_1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ u' & \bar{v}' & \hat{0} & \hat{0} & \hat{0} & \hat{0} & \hat{0} & \hat{0} \\ 0 & t_0 + t_{-1} & 1 & 0 & 0 & 0 & 1 & 0 \end{pmatrix} , \quad H = \begin{pmatrix} & I_4 \\ -I_4 & \end{pmatrix} ,$$

where I_4 is the identity of order 4. A demonstration of the form of the generator can be found in [16]. The computation of $\hat{G}$ involves $O(n \log(n))$ operations.

Once the generator has been computed, a *fast* triangular factorization ($(O(n^2)$ operations) of matrix $\hat{C}$ (7) can be performed by means of the process explained in the following section.

3 Triangular Decomposition of Symmetric Cauchy–Like Matrices

For the discussion of this section, let us start with the following displacement representation of a symmetric Cauchy–like matrix $C \in \mathrm{R}^{n \times n}$,

$$\Lambda C - C \Lambda = G H G^T , \tag{9}$$

where Λ is diagonal and where (G, H) is the generator pair. For the sake of convenience, we have used the same letters as in the previous section in this representation of a general Cauchy–like matrix.

Generally, the displacement representation (9) arises from other displacement representations, like the displacement representation of a symmetric Toeplitz matrix or another symmetric structured matrix. Matrix C is not explicitly formed in order to reduce the computational cost; matrix C is implicitly known by means of matrices G, H and Λ.

From equation (9), it is clear that any column of C, $C_{:,j}$, $j = 0, \ldots, n-1$, of C can be obtained by solving the Sylvester equation

$$\Lambda C_{:,j} - C_{:,j}\lambda_{j,j} = GHG_{j,:}^T ,$$

and that the $(i,j)^{th}$ element of C can be computed as

$$C_{i,j} = \frac{G_{i,:}HG_{j,:}^T}{\lambda_{i,i} - \lambda_{j,j}} , \qquad (10)$$

for all i, j with $i \neq j$, that is, for all the off–diagonal elements of C. We can use this assumption because all the elements of the diagonal matrix Λ (5) used in our case are different. With regard to the diagonal elements of C, assume that they have been computed a priori.

Given the first column of C, we use the following partition of C and Λ,

$$C = \begin{pmatrix} d \; c^T \\ c \; C_1 \end{pmatrix} , \qquad \Lambda = \begin{pmatrix} \lambda \; 0 \\ 0 \; \hat{\Lambda} \end{pmatrix} ,$$

with $C_1, \hat{\Lambda} \in \mathrm{R}^{(n-1)\times(n-1)}$, $c \in \mathrm{R}^{n-1}$ and $d, \lambda \in \mathrm{R}$, to define the following matrix X,

$$X = \begin{pmatrix} 1 & 0 \\ l & I_{n-1} \end{pmatrix} , \qquad X^{-1} = \begin{pmatrix} 1 & 0 \\ -l & I_{n-1} \end{pmatrix} ,$$

where $l = c/d$. By applying $X^{-1}(.)X^{-T}$ to equation (9) we have,

$$\begin{aligned} X^{-1}(\Lambda C - C\Lambda)X^{-T} &= \\ (X^{-1}\Lambda X)(X^{-1}CX^{-T}) - (X^{-1}CX^{-T})(X^T\Lambda X^{-T}) &= \\ \begin{pmatrix} \lambda & 0 \\ \hat{\Lambda}l - \lambda l & \hat{\Lambda} \end{pmatrix} \begin{pmatrix} d & 0 \\ 0 & C_{sc} \end{pmatrix} - \begin{pmatrix} d & 0 \\ 0 & C_{sc} \end{pmatrix} \begin{pmatrix} \lambda & l^T\hat{\Lambda} - \lambda l^T \\ 0 & \hat{\Lambda} \end{pmatrix} &= \\ (X^{-1}G)H(X^{-1}G)^T , \end{aligned}$$

where $C_{sc} = C_1 - (cc^T)/d$ is the Schur complement of C with respect to d. At this point, we have the first column of L, $\begin{pmatrix} 1 \; l^T \end{pmatrix}^T$, and the first diagonal entry, d, of the LDL^T decomposition of $C = LDL^T$, with L being a lower unit triangular factor and D diagonal.

Equating the $(2,2)$ position in the above equation, we have the displacement representation of the Schur complement of C with respect to its first element,

$$\hat{\Lambda}C_{sc} - C_{sc}\hat{\Lambda} = G_1HG_1^T ,$$

where G_1 is the portion of $X^{-1}G$ from the second row down. The process can now be repeated on the displacement equation of the Schur complement C_{sc} to get the second column of L and the second diagonal element of D. After n steps, the LDL^T factorization of C is obtained.

Now, we show how to apply the LDL^T decomposition to matrix $\hat{C} = S^T M S$ (7). The LDL^T factorization of $\hat{C}$ can be represented as

$$\hat{C} = \begin{pmatrix} C^T C & C^T \\ C & 0 \end{pmatrix} = \begin{pmatrix} \hat{\bar{R}}^T & 0 \\ \hat{\bar{Q}} & \hat{\bar{\Delta}} \end{pmatrix} \begin{pmatrix} \hat{\bar{D}}_1 & 0 \\ 0 & -\hat{\bar{D}}_2 \end{pmatrix} \begin{pmatrix} \hat{\bar{R}} & \hat{\bar{Q}}^T \\ 0 & \hat{\bar{\Delta}}^T \end{pmatrix} , \qquad (11)$$

where $\hat{\bar{R}}^T$ and $\hat{\bar{\Delta}}$ are lower unit triangular matrices, and $\hat{\bar{D}}_1$ and $\hat{\bar{D}}_2$ are diagonal.

From (7) and (11), we have $\hat{R} = \hat{\bar{D}}_1^{\frac{1}{2}} \hat{\bar{R}}$, $\hat{Q} = \hat{\bar{Q}} \hat{\bar{D}}_1^{\frac{1}{2}}$ and $\hat{\Delta} = \hat{\bar{\Delta}} \hat{\bar{D}}_2^{\frac{1}{2}}$, and the solution of system (1) is $x = \mathcal{S} \hat{\bar{R}}^{-1} \hat{\bar{Q}}^T \hat{\bar{\Delta}}^{-T} \hat{\bar{D}}_2^{-1} \hat{\bar{\Delta}}^{-1} \hat{b}$.

Diagonal entries of Cauchy–like matrix $C^T C$ cannot be computed by means of (10), so we need an algorithm to obtain them without explicitly computing all the Cauchy–like matrix entries by means of discrete transforms. An algorithm for the computation of these entries exists and has a computational cost of $O(n \log n)$ operations. An algorithm for the computation of the entries of $\mathcal{O}^T T^T T \mathcal{O}$, where $\mathcal{O}$ is any of the existing orthogonal trigonometric transforms, can be found in [15]. A more specific algorithm can be found in [16] for the case in which $\mathcal{O} = \mathcal{S}$.

4 The Parallel Algorithm

Most of the cost of the parallel algorithm is incurred during the second step, that is, during the triangular decomposition of the Cauchy–like matrix $\hat{C}$. The triangularization process deals with the $2n \times 8$ entries of the generator of $\hat{C}$. Usually, $2n \gg 8$ and the operations performed in the triangularization process can be carried out independently on each row of the generator. In order to get the maximum efficiency in the global parallel algorithm, we have chosen the best data distribution for the triangular factorization step.

The generator $\hat{G}$ of the displacement representation of $\hat{C}$,

$$\Lambda \hat{C} - \hat{C} \Lambda = \hat{G} H \hat{G}^T ,$$

is partitioned into $2n/\nu$ blocks of size $\nu \times 8$, and they are cyclically distributed onto an array of p processors, denoted by P_k, for $k = 0, \ldots, p-1$, in such a way that block $\hat{G}_i$, $i = 0, \ldots, 2n/\nu - 1$, belongs to processor $P_{i \bmod p}$ (Fig. 1). For simplicity, in the presentation we assume that $2n \bmod \nu = 0$, although we do not have this restriction in the implemented algorithm. This distribution is performed and managed by ScaLAPACK tools on a one–dimensional mesh of p processors.

The lower unit triangular factor,

$$L = \begin{pmatrix} \hat{\bar{R}}^T & 0 \\ \hat{\bar{Q}} & \hat{\bar{\Delta}} \end{pmatrix} , \qquad (12)$$

obtained by the triangular factorization algorithm, is partitioned into square blocks of size $\nu \times \nu$ forming a two–dimensional mesh of $(2n/\nu) \times (2n/\nu)$ blocks.

P_0	$\hat{G}_0$	L_{00}
P_1	$\hat{G}_1$	L_{10} L_{11}
P_2	$\hat{G}_2$	L_{20} L_{21} L_{22}
P_0	$\hat{G}_3$	L_{30} L_{31} L_{32} L_{33}
P_1	$\hat{G}_4$	L_{40} L_{41} L_{42} L_{43} L_{44}
$\vdots$	$\vdots$	$\vdots$ $\vdots$ $\vdots$ $\vdots$ $\vdots$ $\ddots$

Fig. 1. Example of data distribution for $p = 3$

Let $L_{i,j}$ be the $(i,j)^{th}$ block of L; then, blocks $L_{i,j}$, $j = 0, \ldots, 2n/\nu - 1$, belong to processor $P_{i \bmod p}$ (Fig. 1). The diagonal elements of matrix D (11) are stored in the diagonal entries of L (all diagonal entries of L are equal to one).

The block size ν influences the efficiency of the parallel algorithm. Large values of ν produce a low number of large messages, but the workload is unbalanced among the processors, while small values of ν produce a higher number of smaller messages, but also a better workload balance. Furthermore, the effect of the block size depends on the hardware platform used, and so the block size must be chosen by experimental tuning.

Once we know the data distribution among the processors, we briefly describe the parallel version of the algorithm. The generator is computed in the first step. The computation of the generator involves a Toeplitz matrix–vector product and the translation to the Cauchy–like form by means of the trigonometric transforms. The trigonometric transforms are applied with subroutines from the library BIHAR [17, 18]. Currently, there are no efficient parallel implementations of these routines for distributed memory architectures so the trigonometric transforms will be applied sequentially in this first step of the parallel algorithm. This first step has a low weight in the total execution time if the problem size plus one is not a prime number, as will be shown in Section 5.

The block triangular factorization can be described as an iterative process of $2n/\nu$ steps. When the iteration k starts (Fig. 2), blocks $\hat{G}_i$ such that $i < k$ have already been zeroed, and blocks $L_{i,j}$, such that $j < k$ for all i, have also been computed. Then, during the iteration k, the processor that contains block

$$
\begin{array}{c|ccccc}
\hat{\mathbf{0}} & L_{00} & & & & \\
\vdots & \vdots & \ddots & & & \\
\hat{\mathbf{0}} & L_{(k-1)0} & \ldots & L_{(k-1)(k-1)} & & \\
\hat{G}_k & L_{k0} & \ldots & L_{k(k-1)} & L_{kk} & \\
\hat{G}_{k+1} & L_{(k+1)0} & \ldots & L_{(k+1)(k-1)} & L_{(k+1)k} & \\
\hat{G}_{k+2} & L_{(k+2)0} & \ldots & L_{(k+2)(k-1)} & L_{(k+2)k} & \\
\vdots & \vdots & \vdots & \vdots & \vdots &
\end{array}
$$

Fig. 2. Iteration example

$\hat{G}_k$ computes $L_{k,k}$, zeroes $\hat{G}_k$ and broadcasts the suitable information to the rest of the processors. Then, the processors that contain blocks of $\hat{G}$ and L with indexes that are greater than k, compute blocks $L_{i,k}$ and update blocks $\hat{G}_i$, for $i = k+1, \ldots, 2n/\nu - 1$. Blocks $L_{i,j}$, with $j > k$ for all i, have not yet been referenced at this iteration.

Finally, the third step of the algorithm is carried out by means of calls to PBLAS routines in order to solve three triangular systems and one matrix–vector product in parallel. The parallel algorithm is summarized in Algorithm 1.

Algorithm 1. (Algorithm QRMC). *Given a non–symmetric Toeplitz matrix $T \in R^{n \times n}$ and an independent vector $b \in R^n$, this algorithm returns the solution vector $x \in R^n$ of the linear system $Tx = b$ using a* column *array of p processors.*

For all P_i, $i = 0, \ldots, p-1$, do

1. *Each processor computes the full generator matrix $\hat{G}$ (5) and stores its own row blocks in order to form the distributed generator (Fig. 1).*
2. *All processors compute the triangular factor L (12) in parallel using the parallel triangular decomposition algorithm explained above. Factor L will be distributed as shown in Fig. 1.*
3. *All processors compute $\hat{x}$ by means of (8) using parallel routines of PBLAS, and P_0 computes the solution $x = \mathcal{S}x$ of the linear system.*

In order to improve the precision of the results, an iterative refinement can be applied by repeating the third step of the algorithm.

5 Experimental Results

All experimental analyse were carried out in a cluster with 12 nodes connected by a Gigabit Ethernet. Each node is a bi–processor board IBM xSeries 330 SMP composed of two Intel Pentium III at 866 MHz with 512 Mb of RAM and 256 Kb of cache memory.

The first analysis concerns the block size ν. The main conclusion of that analysis is that the worst results are obtained with small values of ν. The results improve as ν grows, arriving to the optimum when $\nu \approx 50$. Thus, the time spent by the parallel algorithm grows very slowly as we approach the maximum value of $\nu = 2n/p$ (only one block per processor). This performance behaviour is independent of both the number of processors and the matrix size. A full experimental analysis of the influence of the block size can be found in [16].

The second experimental analysis deals with the effect of the three main steps of the parallel algorithm on its total cost. Table 1 shows the time in seconds spent on each of these steps. First of all, it can be observed that the time used for the calculus of the generator tends to grow with the problem size, but it shows large changes for certain values of n. More specifically, the execution time of the trigonometric transform routines depends on the size of the prime factors of $(n+1)$. That is, for problem sizes $n = 2800, 3000, 4000$, $(n+1)$ is a prime number and the divide–and–conquer techniques used in trigonometric transform

Table 1. Execution time of each step of the parallel algorithm on one processor

n	primes of $(n+1)$	generator calculus	factorization step	system solution	total time
2000	$3 \times 23 \times 29$	0.32 (9%)	2.76 (81%)	0.32 (9%)	3.24
2200	$3^4 \times 71$	0.38 (9%)	3.47 (82%)	0.39 (9%)	4.43
2400	7^4	0.42 (8%)	4.52 (84%)	0.47 (9%)	5.34
2600	$3^2 \times 17^2$	0.48 (7%)	5.49 (84%)	0.55 (8%)	6.54
2800	2801	1.58 (17%)	6.81 (75%)	0.74 (8%)	9.04
3000	3001	1.79 (17%)	7.77 (75%)	0.84 (8%)	10.65
3200	$3 \times 11 \times 97$	0.67 (6%)	9.48 (86%)	0.82 (7%)	10.62
3400	19×179	0.76 (6%)	10.62 (86%)	0.94 (8%)	12.43
3600	13×277	0.85 (6%)	12.49 (87%)	1.05 (7%)	14.26
3800	$3 \times 7 \times 181$	0.89 (6%)	14.05 (87%)	1.17 (7%)	16.15
4000	4001	2.99 (15%)	15.96 (78%)	1.50 (7%)	20.24

routines cannot be exploited. However, with the exception of these cases, the weight (in parenthesis) of this step with respect to the total time is low. The most costly step is the second one, the "factorization step". The third step has a low impact on the total time. If we need to apply an iterative refinement step, the time for the "system solution" doubles the time shown in Table 1.

Our third experimental analysis is about the precision of the results obtained with the parallel algorithm. To perform this analysis we use the forward or relative error, defined as $\|x - \tilde{x}\|/\|x\|$, where x and $\tilde{x}$ are the exact and the computed solution respectively. In order to obtain the relative errors, we have computed the independent vector $b = Tx$ with a known solution x and with all entries equal to 1.

The matrix used for the experimental analysis of the precision of the solution and the stability of the algorithm is

$$T = T_0(\epsilon) + \epsilon T_1 \ ,$$

where $T_0(\epsilon) = \left(t_{|i-j|}\right)_{i,j=0}^{n-1}$ is a symmetric Toeplitz matrix with $t_i = (1/2)^i$, for all $i \neq 0$, and $t_0 = \epsilon$. When $\epsilon \ll 1$, all leading principal submatrices of size $3k+1$, $k = 0, \ldots, (n-1)/3$ are near–singular. We have used a value of $\epsilon = 10^{-14}$, thus, matrix T_0 is a non–strongly regular matrix. Classical *fast* algorithms like Levinson [19] and *superfast* algorithms like Bitmead–Anderson [20] fail or may produce poor results with non–strongly regular matrices. These problems occur even with well–conditioned matrices, at least for non–symmetric and symmetric indefinite matrices [21]. The only solution is to apply some additional look–ahead or refinement techniques in order to stabilize the algorithm and to improve the accuracy of the solution. Matrix T_1 is a randomly formed Toeplitz matrix with entries belonging to $[0, 1]$. We use matrix ϵT_1 to obtain a non–symmetric Toeplitz matrix T with features that are similar to T_0.

We will compare the relative error of the QRMC algorithm with two other parallel algorithms. The first algorithm, called QRMS (QR Modified decomposi-

Table 2. Forward error

n	QRMS	LUC	QRMC
2000	5.72×10^{-13}	3.12×10^{-11}	2.78×10^{-10}
2300	1.74×10^{-12}	2.30×10^{-11}	4.62×10^{-10}
2600	6.76×10^{-13}	3.93×10^{-11}	1.71×10^{-09}
2900	4.25×10^{-13}	9.77×10^{-11}	1.05×10^{-09}
3200	2.02×10^{-12}	6.29×10^{-12}	6.20×10^{-10}
3500	3.10×10^{-13}	9.72×10^{-11}	1.88×10^{-09}
3800	1.11×10^{-12}	2.31×10^{-11}	2.87×10^{-10}

Table 3. Forward error with one iteration of iterative refinement

n	QRMS	LUC	QRMC
2000	6.07×10^{-16}	4.03×10^{-13}	5.28×10^{-16}
2300	5.19×10^{-16}	8.55×10^{-13}	4.89×10^{-16}
2600	6.15×10^{-16}	4.04×10^{-13}	3.36×10^{-16}
2900	6.67×10^{-16}	5.15×10^{-12}	3.41×10^{-16}
3200	6.67×10^{-16}	1.43×10^{-12}	5.92×10^{-16}
3500	7.03×10^{-16}	3.42×10^{-12}	2.69×10^{-16}
3800	5.16×10^{-16}	6.13×10^{-13}	6.72×10^{-16}

tion with the Generalized Schur Algorithm), corresponds to the PAQR algorithm mentioned in the introduction and in Section 2 of this paper. The second algorithm, called LUC (LU factorization of a Cauchy–like translation of the Toeplitz matrix), is described in [10]. Basically, this algorithm performs a transformation of the Toeplitz system into a Cauchy–like system and computes a LU factorization of the resulting matrix.

Table 2 shows that we did not obtain the expected precision in the solution, that is, a relative error close to the relative error obtained with the QRMS algorithm. This is due to the errors induced by the discrete trigonometric transforms. The transforms $\mathcal{S}$ and $\mathcal{C}$ have an undesirable impact on the computation of the generator and on the solution obtained by the parallel algorithm.

However, the results in Table 3 show that the solution can be greatly improved by means of only one step of iterative refinement. This refinement uses matrices $\hat{\bar{R}}^T$, $\hat{\bar{Q}}$, and $\hat{\bar{\Delta}}$ that form the L (12). A similar refinement technique can be applied to the solution obtained with the LUC algorithm, but the precision does not improve as much as with the QRMC algorithm. In [16], we performed a similar analysis using the backward error, and we arrived to similar conclusions.

Finally, we analyze the execution time of the QRMC algorithm with several processors and different problem sizes (Fig. 3). It can be seen that the time decreases as the number of processors increases.

Fig. 4 allows us to compare the execution time of the three algorithms for a fixed problem size. The difference in time is mainly due to the size of the generator used in each case. While LUC works on a generator of size $n \times 4$,

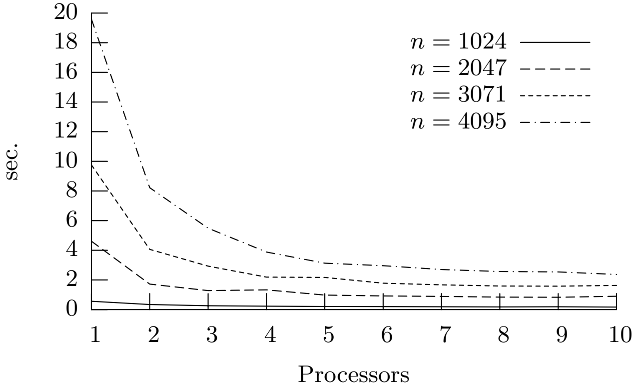

Fig. 3. Time in seconds of the QRMC algorithm

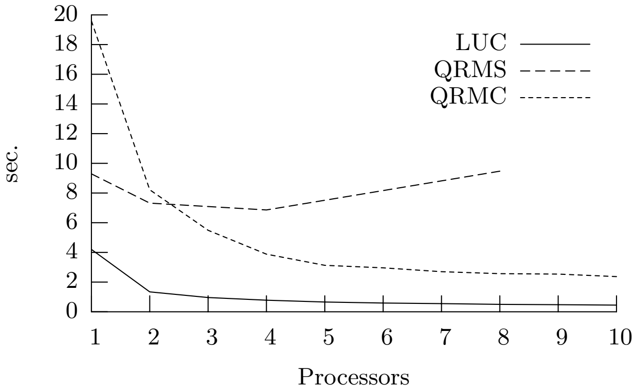

Fig. 4. Time in seconds of the three algorithms for $n = 4095$

QRMS works on a generator of size $2n\times 6$ and QRMC works on a generator of size $2n \times 8$. The embedding technique used by these last two algorithms to improve the stability and the precision of their results increases the number of rows from n to $2n$. The number of columns is related to the rank of the displacement representation. Also, the last step of the QRMS and QRMC algorithms involves the solution of three triangular systems and a matrix–vector product, while LUC only solves two triangular systems. In addition, the time shown in Fig. 4 for QRMC includes one step of the iterative refinement solution, which is not performed in the other two parallel algorithms.

The LUC and QRMC algorithms have very similar behaviours. The execution time decreases very fast when few processors are used, and decreases more slowly when the number of processors used increases. In the QRMS algorithm, the time is only reduced when two processors, or even four processors are used for

large problems. For more processors, the time increases due to the effect of the communication cost. Furthermore, QRMS can only be executed on $p = 2^i$ processors, for any integer i. Our goal when developing the QRMC algorithm was to improve QRMS by means of the Cauchy translation and the results show that we have reduced the communication cost, obtaining an efficiency that is similar to the LUC algorithm, while keeping the precision of the results obtained with the QRMS algorithm. As an example, in the problem in Fig. 4, the relative error is 2.33×10^{-11} for LUC, 1.48×10^{-12} for QRMS, and 3.13×10^{-16} for QRMC using only one iteration of iterative refinement in this last case.

6 Conclusions

In this paper, we describe the parallelization of a new algorithm for solving non–symmetric Toeplitz systems. The parallel algorithm combines the efficiency of the LUC algorithm and the stability of the QRMS algorithms presented in [10].

This algorithm embeds the original Toeplitz matrix into an augmented matrix in order to improve its stability and the precision of the results. The application of only one step of iterative refinement to the solution produces very precise results. However, as can be observed in [10], this technique alone produces a non-scalable parallel algorithm due to the high communication cost.

In this paper, we solve this problem by applying trigonometric transforms that convert the original Toeplitz matrix into a Cauchy–like matrix. This transformation allows us to deal with diagonal displacement matrices, which greatly reduces the communication cost. The efficiency obtained with this method is similar to other less accurate parallel algorithms, while maintaining the accuracy of the solution.

An experimental analysis of the algorithm was performed in a cluster of personal computers. Standard tools and libraries, both sequential and parallel, were used. This has produced a code that is portable to different parallel architectures.

The experimental results show the precision and efficiency of the new parallel algorithm with few processors (< 10). The scalability of the parallel algorithm is quite good, considering the low cost of the sequential algorithm that we are parallelizing, $O(n^2)$. The analysis also shows the negative effect of some steps of the algorithm in certain cases. Future work should be done to address this problem.

Acknowledgments

The authors are indebted to Prof. Raymond Hon–fu Chan for his reference to several useful papers and Prof. Daniel Potts for his useful hints and explanations about the computation of diagonal entries of a symmetric Cauchy–like matrix.

References

1. R. W. Freund. A look-ahead Schur-type algorithm for solving general Toeplitz systems. *Zeitschrift für Angewandte Mathe. und Mechanik*, 74:T538–T541, 1994.

2. S. Chandrasekaran and Ali 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, January 1998.
3. 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.
4. 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, January 1 1991. Tue, 23 Apr 1996 15:17:55 GMT.
5. I. Ipsen. Systolic algorithms for the parallel solution of dense symmetric positive-definite Toeplitz systems. Technical Report YALEU/DCS/RR-539, Department of Computer Science , Yale University, May 1987.
6. D. J. Evans and G. Oka. Parallel solution of symmetric positive definite Toeplitz systems. *Parallel Algorithms and Applications*, 12(9):297–303, September 1998.
7. 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, July 1991.
8. K. Gallivan, S. Thirumalai, and P. Van Dooren. On solving block toeplitz systems using a block schur algorithm. In Jagdish Chandra, editor, *Proceedings of the 23rd International Conference on Parallel Processing. Volume 3: Algorithms and Applications*, pages 274–281, Boca Raton, FL, USA, August 1994. CRC Press.
9. 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.
10. Pedro Alonso, José M. Badía, and Antonio M. Vidal. Parallel algorithms for the solution of toeplitz systems of linear equations. In *Proceedeings of the Fifth International Conference On Parallel Processing and Applied Mathmatics*, Czestochowa, Poland, September 2003 (to appear in *Lecture Notes in Computer Science*).
11. E. Anderson et al. *LAPACK Users' Guide*. SIAM, Philadelphia, 1995.
12. L.S. Blackford et al. *ScaLAPACK Users' Guide*. SIAM, Philadelphia, 1997.
13. Thomas Kailath and Ali H. Sayed. Displacement structure: Theory and applications. *SIAM Review*, 37(3):297–386, September 1995.
14. Georg Heinig and Adam Bojanczyk. Transformation techniques for Toeplitz and Toeplitz-plus-Hankel matrices. I. transformations. *Linear Algebra and its Applications*, 254(1–3):193–226, March 1997.
15. Daniel Potts and Gabriele Steidl. Optimal trigonometric preconditioners for nonsymmetric Toeplitz systems. *Linear Algebra and its Applications*, 281(1–3):265–292, September 1998.
16. Pedro Alonso, José M. Badía, and Antonio M. Vidal. An efficient and stable parallel solution for non–symmetric Toeplitz linear systems. Technical Report II-DSIC-08/2004, Universidad Politécnica de Valencia, 2004.
17. P. N. Swarztrauber. *Vectorizing the FFT's*. Academic Press, New York, 1982.
18. P. N. Swarztrauber. FFT algorithms for vector computers. *Parallel Computing*, 1(1):45–63, August 1984.
19. N. Levinson. The Wiener RMS (Root Mean Square) error criterion in filter design and prediction. *Journal of Mathematics and Physics*, 25:261–278, 1946.
20. Robert R. Bitmead and Brian D. O. Anderson. Asymptotically fast solution of Toeplitz and related systems of linear equations. *Linear Algebra and its Applications*, 34:103–116, 1980.
21. James R. Bunch. Stability of methods for solving Toeplitz systems of equations. *SIAM Journal on Scientific and Statistical Computing*, 6(2):349–364, April 1985.