CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2005; **17**:49–67
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe.883

Solving the block–Toeplitz least-squares problem in parallel

P. Alonso[1,*,†], J. M. Badía[2] and A. M. Vidal[1]

[1]*Departamento de Sistemas Informáticos y Computación, Universidad Politécnica de Valencia, 46071 Valencia, Spain*
[2]*Departamento de Ingenierá y Ciencia de los Computadores, Universidad Jaume I, 12080 Castellón, Spain*

SUMMARY

In this paper we present two versions of a parallel algorithm to solve the block–Toeplitz least-squares problem on distributed-memory architectures. We derive a parallel algorithm based on the seminormal equations arising from the triangular decomposition of the product $T^{\mathrm{T}}T$. Our parallel algorithm exploits the displacement structure of the Toeplitz-like matrices using the Generalized Schur Algorithm to obtain the solution in $O(mn)$ flops instead of $O(mn^2)$ flops of the algorithms for non-structured matrices. The strong regularity of the previous product of matrices and an appropriate computation of the hyperbolic rotations improve the stability of the algorithms. We have reduced the communication cost of previous versions, and have also reduced the memory access cost by appropriately arranging the elements of the matrices. Furthermore, the second version of the algorithm has a very low spatial cost, because it does not store the triangular factor of the decomposition. The experimental results show a good scalability of the parallel algorithm on two different clusters of personal computers. Copyright © 2005 John Wiley & Sons, Ltd.

KEY WORDS: least-squares problem; block–Toeplitz matrices; Toeplitz–block matrices; displacement structure; Generalized Schur Algorithm; parallel algorithms; clusters of personal computers

1. INTRODUCTION

The parallel algorithm presented in this paper solves the least-squares problem

$$\min_x \|Tx - b\| \tag{1}$$

where $b \in \mathbb{R}^m$ is the independent term, $x \in \mathbb{R}^n$ is the solution vector and $T \in \mathbb{R}^{m\times n}$ is a block–Toeplitz matrix defined as $T = [T_{ij}] = [T_{i-j}]$ for $i = 0, \ldots, m/\mu - 1$ and $j = 0, \ldots, n/\nu - 1$, with $T_{ij} \in \mathbb{R}^{\mu\times\nu}$.

*Correspondence to: Pedro Alonso, Departamento de Sistemas Informáticos y Computación, Universidad Politécnica de Valencia, 46071 Valencia, Spain.
†E-mail: palonso@dsic.upv.es

Contract/grant sponsor: Ministerio de Educación, Cultura y Deporte, Spain; contract/grant number: CICYT TIC 2000-1683-C03-03

Received 29 October 2002
Revised 28 October 2003
Accepted 14 January 2004

Problem (1) arises in many applications such as the inverse filtering signal processing. Sometimes we need to obtain a real-time solution of this problem, or the matrix T is very large. In these cases it is interesting to have efficient parallel algorithms to solve the least-squares problem.

Our parallel algorithm starts with the normal equations $T^{\mathrm{T}}Tx = T^{\mathrm{T}}b$ and obtains the seminormal equations $LL^{\mathrm{T}}x = T^{\mathrm{T}}b$, where L is a lower triangular matrix. From these equations, and solving several triangular systems, we can obtain the solution x of (1).

The concept of displacement structure was first introduced in [1] for matrices with low *displacement rank*. Matrices with this kind of structure include scalar Toeplitz matrices ($T_{ij} \in \mathbb{R}$), but also block–Toeplitz matrices, or the product $T^{\mathrm{T}}T$. Henceforth, we call this kind of matrices *structured* or *Toeplitz-like*.

There are algorithms that profit from this kind of structure to solve linear systems of equations in $O(n^2)$ flops, instead of the $O(n^3)$ flops taken by the algorithms for general dense matrices [2]. These kinds of algorithms are called fast algorithms and can be divided in two main classes. The *Levinson* algorithm [3] and its modifications [4–9] are based on an inverse triangular factorization of T, while *Schur-type* algorithms [10] are based on a direct triangular factorization of T. Several versions of the Schur Algorithm for scalar matrices can be found in [11,12], while block versions can be found in [11,13–15]. The Schur Algorithm can also be used to solve the normal equations, and hence the least-squares problem in (1).

Levinson-type algorithms have a linear spatial cost while Schur-type algorithms have a quadratic spatial cost. The first class of algorithms are also 30% faster than the second class [16]. However, Schur-type algorithms better exploit the numerical properties of positive–definite matrices [17] and they are more parallelizable [18].

The main part of parallel algorithms for Toeplitz-like matrices is devoted to the solution of linear systems of equations. Some systolic algorithms for solving this problem can be found in [19,20], but they can be applied only to positive–definite matrices. Another systolic algorithm [21], is based on the *Bareiss* method [11]. This method has also been implemented on shared-memory computers [18,19] using a bulk synchronous parallel (BSP) model. An algorithm for solving Toeplitz systems, also for shared-memory computers, can be found in [22], while [23] is devoted to the block case. Finally, there are also some parallel iterative algorithms [24], some of which can only be applied to symmetric [25], tridiagonal or band matrices.

Literature related to the solution of the block–Toeplitz least-squares problem is scarce, especially for distributed-memory multiprocessors, but many methods for scalar Toeplitz matrices can be generalized to the block case. Fast algorithms [15,22–25] for solving problem (1) are usually poorly parallelizable.

The method that we have used for solving the least-squares problem is based on the Cholesky factorization $M = LL^{\mathrm{T}}$ and is known as the Generalized Schur Algorithm. A block version of this algorithm can be found in [30,31]. This version performs the factorization of a product of block–Toeplitz matrices $T^{\mathrm{T}}T$ using BLAS3 routines. However, it is very difficult to parallelize this method on distributed-memory computers. The very small granularity of the parallelism and the dependency among the operations produce a high communication cost.

Our parallel algorithm also performs the Cholesky factorization of $T^{\mathrm{T}}T$, but it avoids all the point-to-point communications. Moreover, we have grouped the broadcast-type communications to reduce the communication cost even further. We have implemented two parallel versions based on the same algorithm. The second version avoids storing the Cholesky factor L, greatly reducing the spatial cost, but it theoretically doubles its computation cost. However, the second version is more efficient and so, depending on the block sizes and on the characteristics of the parallel architecture, it can be even faster than the first version.

The structure of this paper is as follows. The displacement structure is presented in Section 2. Section 3 includes our parallel algorithms. Section 4 shows the experimental analysis, and finally some conclusions are discussed in Section 5.

2. DISPLACEMENT STRUCTURE

Given a symmetric matrix $M \in \mathbb{R}^{n\times n}$, we define its displacement rank ∇_F with respect to a lower triangular matrix $F \in \mathbb{R}^{n\times n}$ as

$$\nabla_F = M - FMF^{\mathrm{T}} = GJG^{\mathrm{T}} \tag{2}$$

where $J \in \mathbb{R}^{r\times r}$ is the *signature* matrix and $G \in \mathbb{R}^{n\times r}$ is the *generator* [32]. The signature matrix is a diagonal matrix with as many entries 1 and -1 as the number of positive and negative eigenvalues of ∇_F, respectively.

We say that a matrix is *structured* or has *displacement structure*, if ∇_F has low rank with respect to M ($r \ll n$). This property can be extended to non-symmetric matrices.

2.1. Factorization of $T^{\mathrm{T}}T$ by the Generalized Schur Algorithm

If we want to obtain the triangular factorization $T^{\mathrm{T}}T$ we can apply the Generalized Schur Algorithm to the generator G in (2), where

$$F = Z_n^{\nu} = \begin{pmatrix} 0 & & & \\ I_\nu & 0 & & \\ & \ddots & \ddots & \\ & & I_\nu & 0 \end{pmatrix}, \quad Z_n = \begin{pmatrix} 0 & & & \\ 1 & 0 & & \\ & \ddots & \ddots & \\ & & 1 & 0 \end{pmatrix} \tag{3}$$

and I_ν is the identity matrix of order ν.

The generator G has dimension $n \times 2(\mu+\nu)$ and can be obtained analytically [33] as follows

$$G = \begin{pmatrix} S_0 & 0 & 0 & 0 \\ S_1 & T_{-1}^{\mathrm{T}} & S_1 & T_{m/\mu-1}^{\mathrm{T}} \\ S_2 & T_{-2}^{\mathrm{T}} & S_2 & T_{m/\mu-2}^{\mathrm{T}} \\ \vdots & \vdots & \vdots & \vdots \\ S_{n/\nu-1} & T_{1-n/\nu}^{\mathrm{T}} & S_{n/\nu-1} & T_{m/\mu-(n/\nu-1)}^{\mathrm{T}} \end{pmatrix} \tag{4}$$

with

$$S = T^{\mathrm{T}}UR^{-1}, \quad U = \begin{pmatrix} T_0 \\ T_1 \\ \vdots \\ T_{m/\mu-1} \end{pmatrix} \quad \text{and} \quad R = \mathrm{qr}(U) \tag{5}$$

where $\mathrm{qr}(U)$ is the upper $\nu \times \nu$ submatrix of the triangular factor R of the QR decomposition of U.

Given the following partition of the generator

$$G = \begin{pmatrix} G_1 \\ G_2 \\ G_3 \\ \vdots \\ G_{n/\nu} \end{pmatrix} \tag{6}$$

where $G_k \in \mathbb{R}^{\nu \times 2(\mu+\nu)}$ for $k = 1, \ldots, n/\nu$, the Generalized Schur Algorithm operates on the generator entries and can be stated as follows.

Algorithm 2.1 (Generalized Schur Algorithm). *Given the generator G in (6) for the displacement of $T^{\mathrm{T}}T$ with respect to F, the following algorithm computes the lower triangular factor L such that $T^{\mathrm{T}}T = LL^{\mathrm{T}}$.*

```
Given $L = [\ ]$.
for $k = 1, \ldots, n/\nu$
    1. Choose a $J$-unitary transformation $\Theta$ ($\Theta J \Theta^{\mathrm{T}} = J$) such that $G_k\Theta = (\bar{G}_k \quad 0)$ where $\bar{G}_k$ is a
       lower triangular factor of order $\nu$.
    2. Apply transformation $\Theta$ to the generator, $G \leftarrow G\Theta$.
    3. Update triangular factor, $L \leftarrow [L \quad G_{:,1:\nu}]$.
    4. Update the generator, $G \leftarrow [FG_{:,1:\nu} \quad G_{:,\nu+1:2\mu+\nu}]$.
end for
```

If we apply the previous algorithm, after the iteration k, row blocks $G_i, i = 1, \ldots, k$, have been zeroed. In addition, product $FG_{:,1:\nu}$ of step 4 is the result of shifting down ν rows of $G_{:,1:\nu}$.

2.2. Stability of the method

Algorithm 2.1 is stable if the structured matrix is strongly regular; that is, if all its leading submatrices are non-singular. The accuracy of the results depends on the condition number of these leading submatrices. $T^{\mathrm{T}}T$ is positive definite and so strongly regular. Therefore, the Generalized Schur Algorithm is a stable method to perform the factorization $T^{\mathrm{T}}T = LL^{\mathrm{T}}$ in this case.

The use of the normal equations is not recommended when solving the general least-squares problem, $\min_x \|Ax - b\|$, because the norm and the condition number of $A^{\mathrm{T}}A$ can be very large ($\kappa(A^{\mathrm{T}}A) \approx \kappa(A)^2$ [2, Section 5.3]). However, we can apply this method with *Toeplitz-like* matrices because the product $T^{\mathrm{T}}T$ is never explicitly built.

From [17,34,35] it is known that the computed triangular factor $\tilde{L}$ satisfies

$$\|\tilde{L}\tilde{L}^{\mathrm{T}} - T^{\mathrm{T}}T\| = O(\varepsilon\|T^{\mathrm{T}}T\|)$$

that is, the error bound does not depend on $\kappa(T^{\mathrm{T}}T)$. The main result in [34] shows that

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

where $\tilde{x}$ is the computed solution after solving the two triangular systems included in the seminormal equations $\tilde{L}\tilde{L}^{\mathrm{T}}x = T^{\mathrm{T}}b$.

The Generalized Schur Algorithm is *weakly stable* when applied to the normal equations. That is $\|\tilde{x}-x\|/\|x\|$ is small when the system is well-conditioned [34,36]. This result is good enough in many applications.

3. PARALLEL ALGORITHM

A direct parallelization of Algorithm 2.1 can be obtained by distributing each block G_k in (6), $k = 1, \dots, n/\nu$, to a different processor [30,31]. This method allows the computation and application of each J-unitary transformation Θ (steps 1 and 2) with efficient BLAS3 routines instead of BLAS2 routines.

However, this method requires two kinds of communications on each iteration k:

- a broadcast of the parameters of the J-unitary transformation Θ to the rest of the processors (steps 1 and 2); and
- a point-to-point communication among adjacent processors to perform the shifts in step 4.

Although this algorithm is efficient in one processor [37], the communication cost limits the performance with more processors, mainly because of the point-to-point messages. We have implemented two parallel versions of this algorithm using different block distributions [33]. The experimental results on distributed multicomputers show the negative effect of the communications on the performance of the algorithms.

3.1. Avoiding point-to-point communications

In the parallel algorithm presented in this paper we have avoided the point-to-point communication during the shift step. We transform the block–Toeplitz matrix T into a *Toeplitz–block* matrix $\hat{T}$; that is, a block matrix with scalar Toeplitz blocks. Matrices T and $\hat{T}$ are related through two permutation matrices $Q \in \mathbb{R}^{m\times m}$ and $P \in \mathbb{R}^{n\times n}$, so that $\hat{T} = QTP^{\mathrm{T}}$. The structure of $\hat{T}$ is given by $\hat{T} = [T_{ij}]$, for $i = 1, \dots, \mu$ and $j = 1, \dots, \nu$, where $T_{ij} \in \mathbb{R}^{(m/\mu)\times(n/\nu)}$ are scalar Toeplitz matrices.

We can obtain the displacement of $M = \hat{T}^{\mathrm{T}}\hat{T}$ from that of $M = T^{\mathrm{T}}T$ (2) as follows:

$$\begin{aligned}
T^{\mathrm{T}}T - FT^{\mathrm{T}}TF^{\mathrm{T}} &= GJG^{\mathrm{T}} \\
P(T^{\mathrm{T}}T - FT^{\mathrm{T}}TF^{\mathrm{T}})P^{\mathrm{T}} &= P(GJG^{\mathrm{T}})P^{\mathrm{T}} \\
PT^{\mathrm{T}}Q^{\mathrm{T}}QTP^{\mathrm{T}} - PFP^{\mathrm{T}}PT^{\mathrm{T}}Q^{\mathrm{T}}QTP^{\mathrm{T}}PF^{\mathrm{T}}P^{\mathrm{T}} &= PGJG^{\mathrm{T}}P^{\mathrm{T}} \\
(QTP^{\mathrm{T}})^{\mathrm{T}}(QTP^{\mathrm{T}}) - (PFP^{\mathrm{T}})(QTP^{\mathrm{T}})^{\mathrm{T}}(QTP^{\mathrm{T}})(PFP^{\mathrm{T}})^{\mathrm{T}} &= (PG)J(PG)^{\mathrm{T}} \\
\hat{T}^{\mathrm{T}}\hat{T} - \hat{F}\hat{T}^{\mathrm{T}}\hat{T}\hat{F}^{\mathrm{T}} &= \hat{G}J\hat{G}^{\mathrm{T}}
\end{aligned} \tag{7}$$

given that $Q^{\mathrm{T}}Q = QQ^{\mathrm{T}} = I_m$ and $P^{\mathrm{T}}P = PP^{\mathrm{T}} = I_n$.

Equation (7) for the displacement of $\hat{T}^{\mathrm{T}}\hat{T}$ shows that $\hat{G} = PG$ and $\hat{F} = PFP^{\mathrm{T}} = Z_{n/\nu} \oplus \cdots \oplus Z_{n/\nu}$, where $Z_{n/\nu}$ is the shift matrix defined in (3), but of order n/ν.

The main advantage of our approximation is that the displacement matrix $\hat{F}$ is block diagonal. Therefore,

$$\hat{F}\hat{G} = \hat{F}\begin{pmatrix}\hat{G}_1\\ \hat{G}_2\\ \vdots\\ \hat{G}_\nu\end{pmatrix} = \begin{pmatrix}Z_{n/\nu}\hat{G}_1\\ Z_{n/\nu}\hat{G}_2\\ \vdots\\ Z_{n/\nu}\hat{G}_\nu\end{pmatrix} \tag{8}$$

with $\hat{G}_k \in \mathbb{R}^{(n/\nu)\times 2(\mu+\nu)}$, $k = 1, \ldots, \nu$. That means that many of the operations of the algorithm can be performed independently on each block of n/ν rows of $\hat{G}$. Each product $Z_{n/\nu}\hat{G}_k$ involves shifting down one position the first column of each block $\hat{G}_k$, but this can be performed locally on each processor. Therefore, we avoid the point-to-point communication of the shifting in each iteration of the algorithm.

3.2. Implementation of the J-unitary transformations

Our parallel algorithm applies the transformations in the generator row by row. Therefore, we cannot exploit the efficient block routines included in BLAS3, as in [31]. However, we apply the hyperbolic rotations in a more stable way. It is shown in [38] that if the rotations are applied in a direct form as in [31], the Generalized Schur Algorithm can be unstable. However, if rotations are applied in a factored form which is well known to provide numerical stability in the context of Cholesky downdating, then the algorithm is stable.

Given a vector $g \in \mathbb{R}^{2(\mu+\nu)}$, the J-unitary rotation $\Theta \in \mathbb{R}^{2(\mu+\nu)\times 2(\mu+\nu)}$ has the form

$$\Theta = \begin{pmatrix}\Theta_1 & 0\\ 0 & I\end{pmatrix}\begin{pmatrix}I & 0\\ 0 & \Theta_2\end{pmatrix}\Theta_3 \tag{9}$$

where $\Theta_1 \in \mathbb{R}^{(\mu+\nu)\times(\mu+\nu)}$ is a Householder reflection that zeroes the first $(\mu+\nu)$ elements of g except the first,

$$(g_1' \quad 0 \quad \ldots \quad 0) \leftarrow (g_1 \quad g_2 \quad \ldots \quad g_{(\mu+\nu)})\Theta_1$$

$\Theta_2 \in \mathbb{R}^{(\mu+\nu)\times(\mu+\nu)}$ is a Householder reflection that zeroes the last $(\mu+\nu)$ element of g except the first,

$$(g_{(\mu+\nu+1)}' \quad 0 \quad \ldots \quad 0) \leftarrow (g_{(\mu+\nu+1)} \quad g_{(\mu+\nu+2)} \quad \ldots \quad g_{2(\mu+\nu)})\Theta_2$$

and $\Theta_3 \in \mathbb{R}^{2(\mu+\nu)\times 2(\mu+\nu)}$ is an elementary hyperbolic rotation that zeroes element $g_{(\mu+\nu+1)}'$ using g_1' as a pivot. This hyperbolic rotation is computed and applied using the orthogonal-diagonal (OD) method in [39]. It can be seen that $\Theta J \Theta^{\mathrm{T}} = J$ for J defined as in (2).

3.3. Data distribution

In our parallel algorithm we have used the tools provided by the ScaLAPACK library [40] to perform the data distribution. This library distributes the elements of the matrices in a logical two-dimensional mesh using a block cyclic scheme.

However, we employ a one-dimensional topology with a column of p processors. Blocks $\hat{G}_i$ in (8) are cyclically distributed among the processors P_k, $k = 0, \ldots, p-1$, so that the row block $\hat{G}_i$ is stored in processor $P_{(i-1) \bmod p}$. The Generalized Schur Algorithm mainly works with a generator containing a very small number of columns. The parallelism of the algorithm resides in the operations with different groups of rows and not in the operations with the elements on each row. Therefore, we distribute blocks of rows of the generator in a column of processors, but we do not distribute the elements of each row in a second dimension of processors. We could use a two-dimensional topology and perform the different transformations on each row in each iteration in parallel, but the communication cost involved would exceed the advantages of the parallelism due to the very small granularity of the computations.

3.4. First version of the parallel algorithm

The code for the first version of the parallel algorithm is the following.

Algorithm 3.1 (Parallel algorithm 1). *Given a matrix T and vector b defined in (1), this algorithm computes its solution vector x.*

On each processor P_k, for $k = 0, \ldots, p-1$:

1. *parallel computation of the generator for the displacement $\hat{T}^{\mathrm{T}}\hat{T}$;*
2. *triangular factorization $\hat{T}^{\mathrm{T}}\hat{T} = LL^{\mathrm{T}}$ using Algorithm 3.2;*
3. *parallel solution of the seminormal equations, $x \leftarrow \hat{T}^{\mathrm{T}}b$, $x \leftarrow L^{-1}x$, $x \leftarrow L^{-T}x$, and permutation of the solution vector, $x \leftarrow Px$; where P in (7) is the permutation matrix that relates T and $\hat{T}$.*

The computation of the generator $\hat{G}$ in step 1 is performed without communications [33]. We only need to replicate an operation of $O(m\nu^2)$ flops on each processor. The theoretical cost of computing the generator in parallel is $(mn\nu + n\nu^2)/p + 2\nu^2 m - (2\nu^3)/3$ flops.

The triangular factorization in step 2, is the core of the algorithm and the most costly part. We show this step in Algorithm 3.2.

Algorithm 3.2 (Parallel Generalized Schur Algorithm—Version 1). *Given the generator $\hat{G}$ of the displacement of $\hat{T}^{\mathrm{T}}\hat{T}$ with respect to matrix $\hat{F}$ cyclically distributed over p processors, the following algorithm computes the lower triangular factor L in $\hat{T}^{\mathrm{T}}\hat{T} = LL^{\mathrm{T}}$ using the same distribution as $\hat{G}$.*

On each processor P_k, for $k = 0, \ldots, p-1$:

```
for $i = 1, \ldots, n$
    Let $\vec{g}_i \in \mathbb{R}^{1\times 2(\mu+\nu)}$ be the $i$ row of $\hat{G}$, then,
    if $\vec{g}_i \in P_k$
        1. Choose a $J$-unitary transformation $\Theta$ (9) that zeroes all the elements of $\vec{g}_i$ except the first.
        2. $l_{ii} \leftarrow \vec{g}_{i1}$.
        3. Broadcast transformation $\Theta$ to the rest of processors.
    else
        Receive the transformation $\Theta$.
    end if
    for $j = i+1, \ldots, n$
```

```
        Let $\vec{g}_j$ and $\vec{g}_{j-1}$ be the rows $j$ and $(j-1)$ of $\hat{G}$ respectively, then,
        if $\vec{g}_j \in P_k$
            1. Apply transformation $\Theta$ to row $\vec{g}_j$.
            2. $l_{ji} \leftarrow \vec{g}_{j1}$.
            if $\vec{g}_{j-1} \in P_k$
                $\vec{g}_{j1} \leftarrow l_{j-1i}$.
            else
                $\vec{g}_{j1} \leftarrow 0$.
            end if
        end if
    end for
end for
```

The computation cost of Algorithm 3.2 is given by

$$\frac{n^2(4(\mu+\nu)+3)}{p} - \frac{(4(\mu+\nu)+3)n}{p} + 6n(\mu+\nu) + 7n = O\left(\frac{n^2(\mu+\nu)}{p}\right) \text{ flops} \tag{10}$$

Regarding the communication cost, the algorithm only includes a broadcast on each of the n iterations. The cost of sending a message of size n is modeled by $\beta + n\tau$, where β is the latency and τ is the cost of transferring one element. Then, the total cost of Algorithm 3.2 is

$$O\left(\frac{n^2(\mu+\nu)}{p}\right) t_{\text{flop}} + n(\beta + (2(\mu+\nu)+4)\tau) \tag{11}$$

where $2(\mu+\nu)+4$ is the number of values to represent a J-unitary transformation Θ.

Usually the latency β is much larger than τ, so sending many small messages reduces the efficiency of the algorithm. In our algorithm we could try to reduce the communication cost by grouping the transformations sent by each message. For example, one processor can compute a full block with n/ν transformations and broadcast them in only one message. The rest of the processors could then apply all of the received transformations to their rows. In this case, the resulting cost would be

$$O\left(\frac{n^2(\mu+\nu)}{p}\right) t_{\text{flop}} + \nu\left(\beta + \frac{n}{\nu}(2(\mu+\nu)+4)\tau\right)$$

The experimental results show that the best results are not obtained by sending individual transformations, nor by sending full blocks with n/ν transformations. The optimal number of transformations to group into one message has a value $1 \leq \eta \leq n/\nu$, that depends on the computer architecture and the size of the matrix.

Finally, step 3 in Algorithm 3.1 includes four basic operations. The matrix–vector product $\hat{T}^{\mathrm{T}}b = PT^{\mathrm{T}}b$ can be performed fully in parallel without communications, because all of the processors own the first row and column of blocks of T and the independent vector b. The result of this product is stored following the same cyclic distribution applied to the generator $\hat{G}$ and the triangular factor L. The two triangular systems are solved by using PBLAS routines that are part of the ScaLAPACK [40] parallel library. The last operation, the permutation, is performed in processor P_0. The total cost of the third step of Algorithm 3.1 is of $(mn + 2n^2)/p$ flops.

3.5. Second version of the parallel algorithm

The spatial cost of Algorithm 3.1 is $O(n^2/p)$, because it stores the triangular factor L. This could limit the maximum size of the problems that can be solved. In this section we describe a parallel algorithm that solves the least-squares problem without having to store the triangular factor L. This algorithm theoretically doubles the computational cost of Algorithm 3.1, but the experiments will show that the difference is not so big, and that in some cases this second version of the algorithm is even faster than the first version.

We embed the matrix $\hat{T}^{\mathrm{T}}\hat{T}$ in a larger matrix

$$\hat{M} = \begin{pmatrix} \hat{T}^{\mathrm{T}}\hat{T} & I \\ I & 0 \end{pmatrix} \tag{12}$$

and we define the following partition of $\hat{M}$

$$\hat{M} = \begin{pmatrix} \hat{T}^{\mathrm{T}}\hat{T} & I \\ I & 0 \end{pmatrix} = \begin{pmatrix} L \\ L^{-T} \end{pmatrix} \begin{pmatrix} L^{\mathrm{T}} & L^{-1} \end{pmatrix} - \begin{pmatrix} 0 & 0 \\ 0 & (\hat{T}^{\mathrm{T}}\hat{T})^{-1} \end{pmatrix} \tag{13}$$

where L is the lower triangular factor of the seminormal equations, such that $\hat{T}^{\mathrm{T}}\hat{T} = LL^{\mathrm{T}}$.

Matrix $\hat{M}$ has a displacement structure given by

$$\nabla_K = \hat{M} - \hat{K}\hat{M}\hat{K}^{\mathrm{T}} = \hat{\bar{G}}\bar{J}\hat{\bar{G}}^{\mathrm{T}} \tag{14}$$

where

$$\hat{K} = \hat{F} \oplus \hat{F} = \begin{pmatrix} \hat{F} & 0 \\ 0 & \hat{F} \end{pmatrix}$$

The generator $\hat{\bar{G}} \in \mathbb{R}^{2n\times 2(\mu+\nu)}$ can be analytically obtained [33] taking into account that $\hat{T} = QTP^{\mathrm{T}}$, $\hat{M} = \bar{P}M\bar{P}^{\mathrm{T}}$, $\bar{P} = P \oplus P$,

$$M = \begin{pmatrix} T^{\mathrm{T}}T & I \\ I & 0 \end{pmatrix}$$

where

$$M - KMK^{\mathrm{T}} = \bar{G}\bar{J}\bar{G}^{\mathrm{T}}$$

is the displacement of M with respect to $K = F \oplus F$. Following the idea in (7), and taking $\hat{K} = \bar{P}K\bar{P}^{\mathrm{T}}$, we can see that $\hat{\bar{G}} = \bar{P}\bar{G}$.

If we apply n iterations of the Generalized Schur Algorithm to the generator $\hat{G}$, we obtain the matrices L and L^{-T} in the factorization (13). From these matrices we can obtain the solution x of the least-squares problem (1) by solving the following triangular systems:

$$\begin{aligned} Ly &= w \\ z &= L^{-T}y \end{aligned} \tag{15}$$

where $w = \hat{T}^{\mathrm{T}}b$ and $x = Pz$.

It is not necessary to store all of the factors L and L^{-T}. The Generalized Schur Algorithm computes column i of both factors in iteration i. Given this column, it is possible to update (15).

First, consider the following partition

$$Ly = w \rightarrow \begin{pmatrix} L_{11} & \\ L_{21} & \begin{bmatrix} l_{ii} & \\ l_i & L_{22} \end{bmatrix} \end{pmatrix} \begin{pmatrix} y_1 \\ \begin{bmatrix} y_{ii} \\ y_2 \end{bmatrix} \end{pmatrix} = \begin{pmatrix} w_1 \\ \begin{bmatrix} w_{ii} \\ w_2 \end{bmatrix} \end{pmatrix}$$

The factors $L_{11} \in \mathbb{R}^{(i-1)\times(i-1)}$ and $L_{21} \in \mathbb{R}^{(n-i+1)\times(i-1)}$ contain the $(i-1)$ columns of L computed in the first $(i-1)$ iterations. From the previous equation we can see that $L_{11}y_1 = w_1$, where $y_1, w_1 \in \mathbb{R}^{(i-1)}$ and $L_{21}y_1 = (w_{ii} \quad w_2{}^{\mathrm{T}})^{\mathrm{T}}$ where $w_2 \in \mathbb{R}^{(n-i)}$ and w_{ii} is a scalar. On iteration i, y_1, w_{ii} and w_2 are known, and it is not necessary to store factors L_{11} and L_{21}, computed in previous iterations. On that iteration, the Generalized Schur Algorithm computes the vector $(l_{ii} \quad l_i^{\mathrm{T}})^{\mathrm{T}}$ where $l_i \in \mathbb{R}^{(n-i)}$ and l_{ii} is a scalar. From these values, the algorithm computes $y_{ii} = w_{ii}/l_{ii}$ and updates w_2, $w_2 \leftarrow w_2 - y_{ii}l_i$. It is apparent that it is not necessary to store the column of L computed in iteration i to perform the next iterations.

Now, consider the following partition

$$z = L^{-T}y \rightarrow \begin{pmatrix} \begin{bmatrix} z_1 \\ z_{ii} \end{bmatrix} \\ z_2 \end{pmatrix} = \begin{pmatrix} \begin{bmatrix} L_{11}^{-T} & l'_i \\ & l'_{ii} \end{bmatrix} & L_{12} \\ & L_{22}^{-T} \end{pmatrix} \begin{pmatrix} \begin{bmatrix} y_1 \\ y_{ii} \end{bmatrix} \\ y_2 \end{pmatrix}$$

where $L_{12} \in \mathbb{R}^{i\times(n-i)}$. From this partition, it can be seen that $z_1 = L_{11}^{-T}y_1$, where $z_1 \in \mathbb{R}^{(i-1)}$ is a vector computed in the $(i-1)$ previous iterations. In iteration i the Generalized Schur Algorithm computes column i of L^{-T} and the vector $(l'^{\mathrm{T}}_i \quad l'_{ii})^{\mathrm{T}}$. Using this data, the algorithm computes $z_{ii} = l'_{ii}y_{ii}$ and updates z_1, $z_1 \leftarrow z_1 + y_{ii}l'_i$.

This version of the algorithm uses the same data distribution as version 1. However, in this case we distribute the $2n$ rows of the generators $\hat{\hat{G}}$ in 2ν blocks with n/ν rows. Vectors w and z in (15) are distributed so that vector $(w^{\mathrm{T}} \quad z^{\mathrm{T}})^{\mathrm{T}}$ is divided into 2ν blocks with n/ν elements.

The following algorithm is a version of the Generalized Schur Algorithm (Algorithm 3.2) for the generator of the displacement of matrix $\hat{M}$ (14). The algorithm has been modified to update (15) on each iteration.

Algorithm 3.3 (Parallel Generalized Schur Algorithm—Version 2). *Given the generator of the displacement of the matrix $\hat{M}$ (14), $\hat{\hat{G}} \in \mathbb{R}^{2n\times 2(\mu\times\nu)}$, with respect to the matrix $\hat{K}$, given a partition of this generator in 2ν blocks of size $n/\nu \times 2(\mu+\nu)$ cyclically distributed over p processors, and given the vector $w = \hat{T}^{\mathrm{T}}b$, the following algorithm computes the solution of the seminormal equations $LL^{\mathrm{T}}z = w$.*

On each processor P_k, for $k = 0, \ldots, p-1$:

```
for $i = 1, \ldots, n$
  Let $\vec{g}_i \in \mathbb{R}^{1\times 2(\mu+\nu)}$ be the row $i$ of $\hat{\hat{G}}$, then,
  if $\vec{g}_i \in P_k$
    1. Choose a $J$-unitary transformation $\Theta$ (9) that zeroes all the elements of $\vec{g}_i$ except the first.
    2. $w_i \leftarrow w_i/\vec{g}_{i1}$.
    3. Broadcast the transformation $\Theta$ and $w_i$ to the rest of the processors.
    else
        Receive the transformation $\Theta$ and $w_i$.
    end if
    for $j = i+1, \ldots, 2n$
        Let $\vec{g}_j$ and $\vec{g}_{j-1}$ be the rows $j$ and $(j-1)$ of $\hat{\hat{G}}$ respectively, then,
        if $\vec{g}_j \in P_k$ and $\vec{g}_j \neq \vec{0}$
            1. Apply the transformation $\Theta$ to the row $\vec{g}_j$.
            2. if $j \leq n$
                   $w_j \leftarrow w_j - \vec{g}_{i1} w_i$.
               else
                   $w_j \leftarrow w_j + \vec{g}_{i1} w_i$.
               end if
            3. if $\vec{g}_{j-1} \in P_k$
                   $\vec{g}_{j1} \leftarrow l_{j-1i}$.
               else
                   $\vec{g}_{j1} \leftarrow 0$.
               end if
        end if
    end for
end for
```

The cost of Algorithm 3.3 is of

$$\frac{2n^2(4(\mu+\nu)+3)}{p} + \frac{2n^2}{p} + 6n(\mu+\nu) + 7n = O\left(\frac{2n^2(\mu+\nu)}{p}\right) \text{ flops} \tag{16}$$

In Algorithm 3.3 when a J-unitary transformation Θ is computed n rows are updated, because there are only n rows of the generator different from $\vec{0}$ between j and $2n$. In contrast, in Algorithm 3.2 the number of rows to update in each iteration decreases. This is the main reason for the different computational cost of both algorithms. However, in the second version of the algorithm the workload is balanced if we use ν processors or an integer divisor of this value. In this case the n rows of the generator to update are always equally distributed among the processors. On the other hand, in the first version of the algorithm the workload becomes unbalanced when the number of rows of the generator decreases.

The communication cost is the same in both versions of the algorithm. In the second version we have also grouped η transformations in each message in order to reduce its number, and so reduce the communication cost.

The complete second version of the parallel algorithm can be seen in Algorithm 3.4.

Algorithm 3.4 (Parallel Algorithm 2). *Given the matrix T and the vector b, the following algorithm computes the solution vector x of the problem in (1)*

On each processor P_k, $k = 0, \ldots, p-1$:

1. *Compute in parallel the generator $\hat{\hat{G}}$ of the augmented matrix of displacement $\hat{M}$ (14) and compute $w = \hat{T}^{\mathrm{T}} b$, where $\hat{T} = QTP^{\mathrm{T}}$.*
2. *Compute the vector z using Algorithm 3.3.*
3. *In P_0 solve $x = Pz$.*

The cost of step 1 of Algorithm 3.4 is the same as the cost of step 1 of Algorithm 3.1 plus the cost of the matrix vector product $\hat{T}^{\mathrm{T}}b$. The cost of solving the triangular systems in step 3 of Algorithm 3.1 is now included in the cost of step 2.

4. EXPERIMENTAL RESULTS

The target platform for the main part of the experimental study is a personal computer cluster connected through a Myrinet network [41]. The cluster consists of 32 PCs, based on 300 MHz Intel Pentium II processors, with 128 Mbytes of SDRAM and 512 Kbytes of cache each. The interconnection network consists of two Myrinet 16-port SAN crossbar switches.

A Myricom network card has been incorporated into each PC in order to connect it with one of the switches by means of a bidirectional link, with a bandwidth of 1.28 Gbps. Thus, a bisection bandwidth of 20.48 Gbps on each switch of 16 ports is achieved. The connection between both switches allows this added bandwidth to be scaled.

Each switch is a crossbar and together they allow the definition of any kind of topology by manually or automatically configuring a set of paths between the different ports. Communications are carried out by using a cut-through protocol with low latency and flow control.

There exist specific implementations for Myrinet networks of some message-passing environments. In our case a MPICH/GM environment is used where MPICH [42] is a standard implementation from Argonne National Laboratory and GM is the Myrinet message-passing library, which implemented the zero-copy protocol and offers small latencies and high bandwidths.

In Table I we present the time taken by each of the three main steps of Algorithm 3.2 with matrices of different sizes. We also include the percentage of time taken by each of these steps.

It can be seen that step 1 (the computation of the generator) is most affected by the increment in the number of rows of the matrix, m. This parameter has a slight influence in the other two steps of the algorithm. We can also see how the main part of the time of the algorithm is devoted to step 2 (the factorization of the matrix).

Table II shows the times taken by steps 1 and 2 of the algorithm with a fixed size matrix. We vary the block size (μ and ν) and the number of processors. We can see that the time spent by the computation of the generator (step 1) basically depends on the number of columns of each block (ν), and it is slightly influenced by the number of rows of each block (μ). The reduction of the speed-up is mainly due to the replication of part of the computations in each processor, because this step does not involve any communication. The number of columns of the generator $\hat{G}$ is given by $2(\mu + \nu)$, and so the number of rows and columns of each block equally affect the cost of step 2.

The superlinear speed-up in Table II is due to the memory management of the algorithm. It is easy to see that when all the elements of the generator can be stored in cache memory, the time of the algorithm [33] notably decreases. When the generator cannot be fully stored in cache with one processor, but does fit in cache in the multiprocessor case, the speed of the algorithm increases more than proportionally to the number of processors, and so we obtain superlinear speed-up.

The order in which the different elements of the matrices are accessed has a big influence on the efficiency of the sequential and parallel algorithms. In algorithms where the computational cost is of $O(n^2)$ the influence of the memory access time can be at least as important as the influence of the number of computations. Therefore, we have tried to optimize the arrangement of the elements in

Table I. Time in seconds of the three steps of Algorithm 3.2 with matrices of size $T \in \mathbb{R}^{m\times 720}$ on one processor. The size of the blocks is always 60×40.

m	Step 1	Step 2	Step 3	Total
720	0.35 (3.7%)	9.19 (96.0%)	0.03 (0.3%)	9.57
840	0.41 (4.2%)	9.31 (95.5%)	0.03 (0.3%)	9.75
960	0.96 (9.3%)	9.27 (90.3%)	0.04 (0.4%)	10.27
1080	1.29 (12.3%)	9.16 (87.3%)	0.04 (0.4%)	10.49
1200	1.61 (14.9%)	9.13 (84.7%)	0.04 (0.4%)	10.78
1320	2.59 (22.3%)	8.98 (77.3%)	0.05 (0.4%)	11.62
1440	2.17 (19.2%)	9.10 (80.4%)	0.05 (0.4%)	11.32

Table II. Time in seconds of steps 1 and 2 of Algorithm 3.1 using 1, 2, 4, 8 and 16 processors. Matrix T has a size of 1080×720, $\mu = 60, 120$ and $\nu = 60, 120$. The speed-ups are shown in brackets.

	ν	μ	1	2	4	8	16
Step 1	60	60	1.65	0.91 (1.81)	0.40 (4.13)	0.26 (6.35)	0.18 (9.17)
		120	1.58	0.85 (1.86)	0.41 (3.85)	0.28 (5.64)	0.18 (8.78)
	120	60	4.33	2.70 (1.60)	1.75 (2.47)	1.23 (3.52)	0.96 (4.51)
		120	4.29	2.73 (1.57)	1.74 (2.47)	1.23 (3.49)	0.94 (4.56)
Step 2	60	60	5.47	2.37 (2.31)	1.09 (5.02)	0.60 (9.12)	0.38 (14.4)
		120	11.7	5.37 (2.18)	2.47 (4.73)	1.06 (11.0)	0.57 (20.5)
	120	60	11.7	5.24 (2.24)	2.42 (4.85)	1.23 (9.54)	0.62 (18.9)
		120	14.4	6.90 (2.09)	2.93 (4.93)	1.48 (9.76)	0.71 (20.3)

memory in order to minimize the memory access time for all the algorithms. In other words, we have tried to arrange the elements in such a way that consecutive operations access to adjacent elements in order to reuse the same cache line.

Matrices in ScaLAPACK are stored in a Fortran way, that is, by columns. In order to optimize the memory access in the second step of the algorithm we have distributed $\hat{G}^{\mathrm{T}}$ in a logical row of processors. However, in each iteration of this step (Algorithm 3.2) we have stored the computed column of L as if we had a column of processors. Using this data distribution, access is always to elements that are contiguous in memory in both matrices. When we have applied this idea, we have obtained improvements of more than 50% in the execution time [33].

The previous data distribution forces us to change the logical mesh of processors between steps 2 and 3 of Algorithm 3.1. This change can easily be performed by using two different ScaLAPACK contexts, one with a processor row topology and one with a processor column topology.

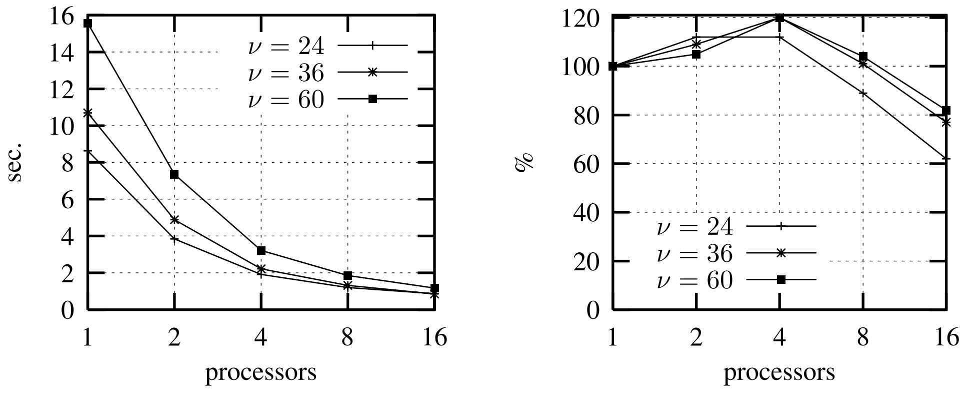

Figure 1. Time and efficiency of Algorithm 3.1 with a matrix $T \in \mathbb{R}^{1440\times 1080}$, $\mu = 48$ and $\nu = 24, 36, 60$.

Another parameter that affects the performance of the algorithm is the number of transformations grouped into each message, η. This value can vary from 1 to n/ν. The experimental results show that the worst results are obtained when η is near to either of these limits, but varies only slightly when we group an intermediate number of transformations. The optimal value of η is around $(2n)/(3(\mu + \nu))$. Furthermore, this behavior is independent of the number of processors.

Figure 1 shows the time and efficiency of Algorithm 3.1 for different values of ν. We can see a constant decrease of the time when we increase the number of processors, allowing efficiencies greater than 60% even with 16 processors. In addition, the efficiency of the parallel algorithm increases with larger block sizes, especially when we increase the number of columns of the block.

We have performed a similar experimental analysis with Algorithm 3.4 and the main conclusions are quite the same. The computation time of the algorithm is mainly spent in step 2 (>90%), which in this case includes the solution of the triangular systems. The influence of cache memory also produces superlinear speed-up. On the other hand, the influence of the parameter η is slightly less than in Algorithm 3.1, but the optimal size of the messages is very similar.

The results in Figure 2 allow us to compare both versions of the parallel algorithm. First, it can be seen that the sequential time of the second version is double that of the first version, confirming the theoretical analysis of previous sections. However, the time of the second version decreases faster than that of the first version, and so the second version obtains better efficiencies. This behavior causes that the time of the second version to be almost the same as that of the first version when we use many processors. Let us note that an efficiency of around 90% is obtained with 24 processors. This is a very good result with a low-cost algorithm and shows the small influence of the communication cost.

On the other hand, we wanted to test the effect of the communication and computation speed of the architecture on the behavior of the parallel algorithms. Therefore, we tested both parallel algorithms on another cluster of PCs with different characteristics. The second cluster is composed of 12 nodes (IBM xSeries 330 SMP). Each node contains two Intel Pentium III processors with a clock rate of

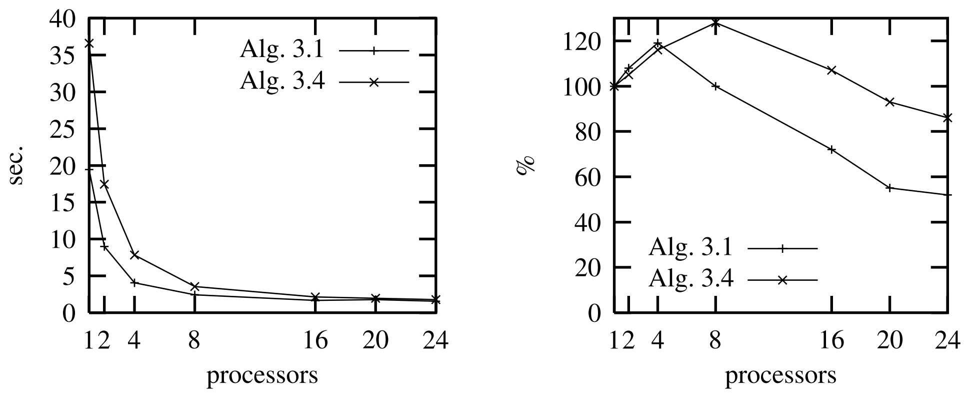

Figure 2. Time and efficiency of Algorithms 3.1 and 3.4 with a matrix $T \in \mathbb{R}^{1536\times 768}$ and blocks of size $\mu \times \nu = 96 \times 96$.

Table III. Characteristics of the two clusters of PCs used in the experiments.

	Myrinet	*Gigabit*
β	62 μs	122 μs
τ	0.021 μs	0.030 μs
Mflops	196.46	645.16
t_f	5.09×10^{-3}	1.55×10^{-3}

866 MHz and 512 MBytes of memory. The interconnection network is Gigabit Ethernet composed of two backbone switches with eight full-duplex 1000BASE-SX (SC-type) ports (TigerSwitch 1000 SMC 8) with 16 Gbps of bandwidth where each node has an SX Ethernet Gigabit NIC for xSeries. We denote the first parallel architecture as *Myrinet* and the second one as *Gigabit*.

Table III shows the characteristics of both clusters. We represent the communication speed with the parameters β (latency) and τ (transfer rate). We represent the computation speed of the processors using the performance obtained with the matrix–matrix product (DGEMM) included in a BLAS3 kernel optimized for each architecture. The table also includes the time required to perform a flop obtained from the results of the previous routine. We can see that the Myrinet network is faster than the Gigabit network, but the second cluster has processors more than three times faster than the first.

If we compare Figures 2 and 3, we can see the different behavior of both clusters with the same matrices. As was expected, the algorithms take less time in the second architecture. However, the behavior of the parallel algorithms is worse on the Gigabit cluster. This is due to the larger ratio of communication to computation cost in the second architecture. In this case, the negative effect of the communications on the efficiency is greater.

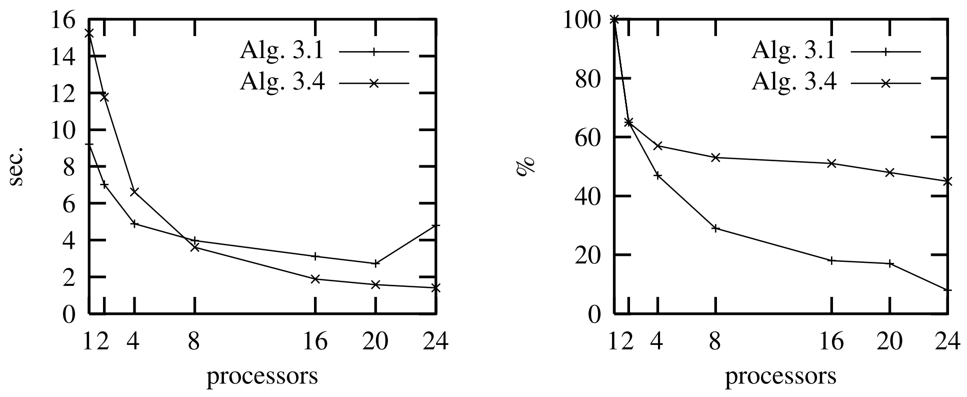

Figure 3. Time and efficiency of Algorithms 3.1 and 3.4 with a matrix $T \in \mathbb{R}^{1536\times 768}$ and a block size of $\mu \times \nu = 96 \times 96$ obtained on the *Gigabit* machine.

Table IV. Memory use (in Mbytes) of both parallel algorithms with matrices $T \in \mathbb{R}^{10000\times 8000}$ and a block size of $(\mu \times \nu) = (20 \times 20)$.

	Processors				
	1	2	4	8	16
Algorithm 3.1	497	251	127	78	53
Algorithm 3.4	14	9	6	5	5

However, we can see that in the second architecture Algorithm 3.4 is even faster than Algorithm 3.1 when we use more than eight processors. This fact is due to the better load balancing of the second version of the algorithm. The weight of the time used for communications in Algorithm 3.4 is smaller than in Algorithm 3.1 regarding the computation time. Therefore, although, theoretically, the second version of the parallel algorithm doubles the cost of the first, in some circumstances the second version can be faster. Specifically, the second version of the algorithm overcomes the first when we use a large enough number of processors, when the relation n/ν is small and when the computer architecture has a large ratio of communication to computation.

There is another important factor that differs on both parallel algorithms: the storage cost. The second version of the algorithm does not store the triangular factor L, and so its storage cost is smaller than the storage cost of the first version (see Table IV). Therefore, the second version can be used with much larger matrices.

Finally, we analyse the precision of the results obtained with both parallel algorithms. In Table V the relative errors of both algorithms with matrices whose elements are generated randomly are shown.

Table V. Relative error of results of Algorithms 3.1 and 3.4 when solving problem (1) with $T \in \mathbb{R}^{1440\times 1080}$ and different block sizes.

	Algorithm 3.1			
$\mu \times \nu$	refs = 0	refs = 1	refs = 2	Algorithm 3.4
4×4	9.15×10^{-15}	1.53×10^{-15}	1.13×10^{-15}	1.56×10^{-14}
12×12	2.34×10^{-14}	5.45×10^{-15}	5.44×10^{-15}	2.42×10^{-14}
20×12	2.36×10^{-14}	1.61×10^{-15}	1.08×10^{-15}	1.43×10^{-14}

The parameter used is

$$\frac{\|\tilde{x} - x\|_2}{\|x\|_2} \tag{17}$$

where $\tilde{x}$ is the computed solution, and x is the exact solution ($x_i = 1$, for all i).

We can see that both algorithms produce solutions with a similar relative error. However, the first version allows the application of several iterative refinement steps to improve the precision of the results.

5. CONCLUDING REMARKS

In this paper we presented two versions of a parallel algorithm for solving the least-squares problem with block–Toeplitz matrices on distributed-memory multiprocessors.

Both parallel algorithms have the following characteristics related to performance.

- They involve only broadcasts and avoid the point-to-point communications of previous versions.
- The cost of the broadcasts has been reduced by grouping the parameters of several iterations into one message. This is especially important with distributed-memory architectures with large latency cost.
- Our algorithms cannot profit from the efficiency of the BLAS3 routines, but we have increased the granularity of the computations by computing and applying groups of transformations.
- The memory access cost can be very important with low-cost algorithms. We have reduced this effect by trying to optimize the arrangement of the elements in memory for all of the algorithms.
- The effect of the cache memory can produce superlinear speed-up.
- We have used standard and efficient routines included in the libraries LAPACK and ScaLAPACK. This increased the efficiency of the algorithms and improves portability.

Regarding the precision of the results, we raise the following points.

- Our algorithms are based on the triangular factorization of the matrix $T^{\mathrm{T}}T$ using the Generalized Schur Algorithm. This matrix is positive-definite and so strongly regular. In this case, it is well known that the algorithm is stable.

- We have computed and applied the hyperbolic rotations in a factored form, which is well known to provide numerical stability.

Finally, the two parallel versions implemented have different characteristics.

- Algorithm 3.1 has half the theoretical cost of Algorithm 3.4. However, the second version produces better load balancing. The experimental results show that the second version is more efficient. Moreover, in some cases the second version can be even faster than the first.
- Algorithm 3.4 has a much smaller storage cost than Algorithm 3.1.
- The precision of the results is similar for both algorithms. However, Algorithm 3.1 allows the application of several iterative refinement steps.

REFERENCES

1. Kailath T, Kung SY, Morf M. Displacement ranks of matrices and linear equations. *Journal of Mathematical Analysis and Applications* 1979; **68**:395–407.
2. Golub GH, Van Loan CF. *Matrix Computations* (3rd edn) (*Johns Hopkins Studies in the Mathematical Sciences*). Johns Hopkins University Press: Baltimore, MD, 1996.
3. Levinson N. The Wiener RMS (Root Mean Square) error criterion in filter design and prediction. *Journal of Mathematics and Physics* 1946; **25**:261–278.
4. Durbin J. The fitting of time series models. *Review of the International Statistical Institute* 1960; **28**:233–243.
5. Trench WF. An algorithm for the inversion of finite Toeplitz matrices. *Journal of the Society for Industrial and Applied Mathematics* 1964; **12**(3):515–522.
6. Zohar S. Toeplitz matrix inversion: The algorithm of W. F. Trench. *Journal of the ACM* 1969; **16**(4):592–601.
7. Akaike H. block Toeplitz matrix inversion. *SIAM Journal on Applied Mathematics* 1973; **24**(2):234–241.
8. Watson GA. An algorithm for the inversion of block matrices of Toeplitz form. *Journal of the ACM* 1973; **20**(3):409–415.
9. Kailath T, Vieira A, Morf M. Inverses of Toeplitz operators, innovations, and orthogonal polynomials. *SIAM Review* 1978; **20**:106–119.
10. Schur J. Über Potenzreihen, die im Innern des Einkeitskreise beschänkt sind. *Journal für die reine und angewandte Mathematik* 1917; **147**:205–232.
11. Bareiss EH. Numerical solution of linear equations with Toeplitz and vector Toeplitz matrices. *Numerische Mathematik* 1969; **13**:404–424.
12. Rissanen J. Solution of linear equations with Hankel and Toeplitz matrices. *Numerische Mathematik* 1974; **22**(5):361–366.
13. Wax M, Kailath T. Efficient inversion of Toeplitz-block Toeplitz matrix. *IEEE Transactions on Acoustics, Speech and Signal Processing* 1983; **31**(5):1218.
14. Kung SY, Whitehouse HJ, Kailath T (eds.). *VLSI and Modem Signal Processing*, Los Angeles, CA, 1–3 November 1982. Prentice-Hall: Englewood Cliffs, NJ, 1985.
15. Sweet DR. Fast Toeplitz orthogonalization. *Numerische Mathematik* 1984; **43**(1):1–21.
16. Kailath T, Sayed AH (eds.). *Fast Reliable Algorithms for Matrices with Structure*. SIAM: Philadelphia, PA, 1999.
17. Bojanczyk AW, Brent RP, de Hoog FR, Sweet DR. On the stability of the Bareiss and related Toeplitz factorization algorithms. *SIAM Journal on Matrix Analysis and Applications* 1995; **16**(1):40–57.
18. Kung SY, Hu YH. A highly concurrent algorithm and pipelined architecture for solving Toeplitz systems. *IEEE Transactions on Acoustics, Speech and Signal Processing* 1983; **31**():66.
19. Brent RP, Luk FT. A systolic array for the linear-time solution of Toeplitz systems of equations. *Technical Report TR82-526*, Cornell University, Computer Science Department, 1982.
20. Sweet DR. 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, 1 January 1991.
21. Huang Y, McColl WF. A BSP Bareiss algorithm for Toeplitz systems. *Journal of Parallel and Distributed Computing* 1999; **56**:99–121.
22. Gohberg I, Koltracht I, Averbuch A, Shoham B. Timing analysis of a parallel algorithm for Toeplitz matrices on a MIMD parallel machine. *Parallel Computing* 1991; **17**(4–5):563–577.
23. de Doncker E, Kapenga J. Parallelization of Toeplitz solvers. *Numerical Linear Algebra, Digital Signal Processing and Parallel Algorithms* (*Computer and Systems Sciences*, vol. 70), Golub GH, Van Dooren P (eds.). Springer: Berlin, 1990; 467–476.

24. Pan VY. Concurrent iterative algorithm for Toeplitz-like linear systems. *IEEE Transactions on Parallel and Distributed Systems* 1993; **4**(5):592–600.
25. Evans DJ, Oka G. Parallel solution of symmetric positive definite Toeplitz systems. *Parallel Algorithms and Applications* 1998; **12**(9):297–303.
26. Bojanczyk AW, Brent RP, Van Dooren P, De Hoog FR. A note on downdating the Cholesky factorization. *SIAM Journal on Scientific and Statistical Computing* 1987; **8**(3):210–221.
27. Chun J, Kailath T, Lev-Ari H. Fast parallel algorithms for QR and triangular factorization. *SIAM Journal on Scientific and Statistical Computing* 1987; **8**(6):899–913.
28. Nagy JG. Fast inverse QR factorization for Toeplitz matrices. *SIAM Journal on Scientific Computing* 1993; **14**(5):1174–1193.
29. Cybenko G. Fast Toeplitz orthogonalization using inner products. *SIAM Journal on Scientific and Statistical Computing* 1987; **8**(5):734–740.
30. Thirumalai S. High performance algorithms to solve Toeplitz and block Toeplitz systems. *PhD Thesis*, Graduate College of the University of Illinois at Urbana-Champaign, IL, 1996.
31. Gallivan KA, Thirumalai S, Van Dooren P, Vermaut V. High performance algorithms for Toeplitz and block Toeplitz matrices. *Proceedings of the 4th Conference of the International Linear Algebra Society, Rotterdam, 1994. Linear Algebra and its Applications* 1996; **241/243**(1–3):343–388.
32. Kailath T, Sayed AH. Displacement structure: Theory and applications. *SIAM Review* 1995; **37**(3):297–386.
33. Alonso P, Badía JM, Vidal AM. Resolución de sistemas lineales y del problema de mínimos cuadrados tipo toeplitz por bloques en paralelo. *Technical Report DSIC-II/13/2002*, Departamento de Sistema Informáticos y Computación de la Universidad Politécnica de Valencia, 2002.
34. Bojanczyk A, Brent RP, de Hoog F. A weakly stable algorithm for general Toeplitz systems. *Technical Report TR-CS-93-15*, Laboratory for Computer Science, Australian National University, Canberra, Australia, 1993 (revised June 1994).
35. Bojanczyk AW, Brent RP, de Hoog FR. Stability analysis of a general Toeplitz systems solver. *Numerical Algorithms* 1995; **10**(3–4):225–244.
36. Bunch JR. The weak and strong stability of algorithms in numerical linear algebra. *Linear Algebra and its Applications* 1987; **88/89**:49–66.
37. Gallivan K, Thirumalai S, Van Dooren P. On solving block Toeplitz systems using a block Schur algorithm. *Proceedings of the 23rd International Conference on Parallel Processing. Volume 3: Algorithms and Applications*, Boca Raton, FL, August 1994 (*Lecture Notes in Computer Science*, vol. 1845), Chandra J (ed.). CRC Press: Boca Raton, FL, 1994; 274–281.
38. Stewart M, Van Dooren P. Stability issues in the factorization of structured matrices. *SIAM Journal on Matrix Analysis and Applications* 1997; **18**(1):104–118.
39. Chandrasekaran S, Sayed AH. A fast stable solver for nonsymmetric Toeplitz and quasi-Toeplitz systems of linear equations. *SIAM Journal on Matrix Analysis and Applications* 1998; **19**(1):107–139.
40. Blackford LS *et al. ScaLAPACK Users' Guide*. SIAM: Philadelphia, PA, 1997.
41. Boden NJ, Cohen D, Felderman RE, Kulawik AE, Seitz CL, Seizovic JN, Su WK. Myrinet. A gigabit-per-second local-area network. *IEEE Micro* 1995; **15**:29–36.
42. Snir M, Otto SW, Huss-Lederman S, Walker DW, Dongarra J. *MPI: The Complete Reference*. MIT Press: Cambridge, MA, 1996.