A PARALLEL 4SID ALGORITHM

Bogdan DUMITRESCU, Boris JORA, Corneliu POPEEA
Alexandru ŞERBĂNESCU

Department of Automatic Control and Computers
"Politehnica" University of Bucharest
313, Spl. Independenţei, 77206 Bucharest, Romania
e-mail: bogdan@lucky.schur.pub.ro

Abstract: In this paper present a parallel implementation, based on ScaLAPACK, of a direct 4SID method. As the computational stages of the method are solved mainly by direct calls to ScaLAPACK routines, we concentrate on the specific difficulties of the implementation, e.g. the redistribution of data between some stages. A structured matrix multiplication is implemented efficiently by a dedicated algorithm. We report experimental results that show good behavior for up to 16 processors.

1. INTRODUCTION

High performance computing (HPC) is based not only on powerful parallel computers, but also on efficient software that tends to standardize some of the programming work. ScaLAPACK is the most important parallel tool for linear algebra problems and is built on top of the most efficient libraries for sequential computation and communication. With ScaLAPACK it is possible to tackle difficult problems of large size, solving them efficiently on a parallel computer, with relatively small programming effort.

Identification lies at the basis of control and new algorithms and approaches continue to appear. A new impulse in this field was given by the apparition of 4SID (State-Space Subspace-based System IDentification) methods, proposed in [21, 24], using numerical tools of high accuracy like the singular value decomposition (SVD). We mention the 4SID methods have been extended to stochastic systems [18, 19, 20, 15], time-variable systems [23], infinite dimensional systems [13], close-loop identification [11], real-time identification [9]; moreover, methods similar to 4SID have been used in several applications in signal processing, see e.g. [10, 12, 16]. Seen in the beginning as difficult to use, due to the large size of the problems to be solved, 4SID methods may now benefit from the power of HPC. The main contribution of our paper is the parallel implementation of a 4SID method, using ScaLAPACK routines, and also specialized algorithms when this was the only effi-

cient choice.

A short overview of this paper is as follows. In section 2, we shortly review the computation environment used for our algorithm: parallel architecture, communication models, matrix distribution on processors and the standard libraries used for computation and communication. Section 3 presents a (sequential) direct 4SID method. The parallel version of the method, which is the principal contribution of this paper, is presented in detail in section 4. Finally, section 5 is dedicated to experimental results that show the good parallel performance of our algorithm.

2. THE PARALLEL MATRIX COMPUTATION ENVIRONMENT

We aim to present in this section the most important features of the parallel hardware and software environment used for matrix computation. This environment has practically become a standard in the latest decade; since its use in applications of automatic control is only at the beginning, we consider necessary to give here a brief description.

2.1. *The parallel arhitecture*

The model of the target parallel computer is MIMD (Multiple Instruction Multiple Data) with distributed memory; each processor executes its own flow of instructions, on local data stored in a local memory. The processors are connected by communication channels and the parallel computer may be viewed as a graph with processors as vertices and channels as edges. The most used topology at this moment is the 2D or 3D grid, but several other fixed or configurable topologies also exist.

The current tendency is that the operating system provide software communication facilities for processor-to-processor communication, regardless the physical links between processors. Consequently, and also due to the efficiency of the corresponding algorithms, the most used topology in parallel matrix computation is the *virtual 2D grid*. We denote $p = p_r p_c$ the number of processors, where p_r and p_c are the number of rows and columns of the grid, respectively. Each processor P_{ij} (on row i, column j) of the grid has an address, made up of a single number, e.g. $ip_c + j$.

The programming style is SPMD (Single Program Multiple Data); all the processors execute the same program, on local data; the execution is generally asynchronous; although the program is unique, the processors may execute personalized instructions, conditioned by their address.

The organization of the local memory of a processor is of high importance. The most effective architecture is now hierarchical, in which the memory is structured on (at least) two levels: a small fast memory "near" the processor, and a slow and large one which stores the data. The fast memory is used for holding temporary data; as the processors are very fast, if data from this memory are reused without being stored on the slow memory, then the execution of a program is significantly faster. We will see the programming implications of this architecture in section 2.4.

2.2. *Matrix mapping*

In a distributed memory computer, the data are mapped to processors following three principles: (a) each piece of data is given to a single processor; (b) each processor receives roughly the same amount of data; (c) the computation is load balanced, i.e. each processor has roughly the same amount of computation to perform. With (a) and (b), the use of local memory is optimized, allowing to larger problems to be solved; however, requirement (c) does not follow automatically.

In matrix computation, the best mapping, satisfying always (a) and (b), and almost always (c), is the *block cyclic* mapping. Let A be a matrix of size $m \times n$; the matrix is split in blocks of size $m_b \times n_b$, where usually $m_b \ll m$, $n_b \ll n$; certainly, the blocks corresponding to the last rows or columns may have smaller sizes, if m is not a multiple of m_b or n is not a multiple of n_b. Let us denote A_{IJ} the blocks of A, where $0 \leq I < n_r$, $n_r = \lceil m/m_b \rceil$, and

$0 \leq J < n_c$, $n_c = \lceil n/n_b \rceil$; we consider the indices starting at 0.

To describe the mapping, we refer only to the rows of the processor grid and the block rows of the matrix; the situation on columns is similar. A processor on row k of the grid receives blocks A_{IJ} with I such that $I \bmod p_r = k$, i.e. $I = \ell p_r + k$, $\ell = 0 : \lceil (n_r - k)/p_r \rceil$. So, the matrix is partitioned in groups of $p_r \times p_c$ blocks, starting from the upper left corner; each block of such a group is mapped to a processor of the grid, in the most natural way; the block and the processor have the same position in the group and the grid, respectively. The mapping can be generalized by allowing any processor to hold the block A_{00}, the cyclicity being preserved.

2.3. *Communication routines*

The communication model is based on messages sent between two processors. Two libraries implementing communication primitives became popular in the latest years: MPI (Message Passing Interface) [7, 17] and PVM (Parallel Virtual Machine) [8].

These libraries were written for general purposes. For matrix computation applications, the standard BLACS (Basic Linear Algebra Communication Subprograms) [6] was proposed in 1995, offering communication routines oriented on transmitting matrices or blocks of matrices. BLACS is built on top of MPI or PVM and assumes a block cyclic mapping of the matrices.

There are two types of communication routines in BLACS:

- point-to-point communication, where a processor sends a submatrix and another receives it;
- global communication, where all (or several) processors are involved; the implemented operations are broadcast (one sends to all), global sum (or minimum, maximum) computation, where all processors cooperate to compute the sum of blocks mapped in their local memories.

The global communication routines may be called by a group of processors, providing they are organized in a virtual grid.

2.4. *Sequential computation libraries*

The hierarchical organization of local memories led to the apparition of routines working on blocks of matrices; intensive computation on a block of matrix allows its storing in the fast memory, thus speeding up computation. Two important libraries for sequential matrix computation are now widely used.

Level 3 BLAS (Basic Linear Algebra Subroutines) [5] contains routines for matrix multiplication (with general, symmetric, and triangular operands) and for solving triangular systems with multiple right hand term.

LAPACK (Linear Algebra PACKage) [1] is a much more general library, dedicated to the fundamental problems of linear algebra: linear systems (with matrices with several structures: general, symmetric, symmetric positive definite, band), least squares linear problems, eigenvalues and eigenvectors, singular values, orthogonal bases for linear subspaces, etc. There are also distinct routines for the main factorizations used for solving the above problems, e.g. LU, QR, LQ, SVD.

LAPACK routines are based on algorithms at block level (opposite to element level algorithms, as in LINPACK or EISPACK). For block operations, BLAS 3 routines are called. A good implementation of BLAS 3 is sufficient to ensure good performance of LAPACK.

2.5. *Parallel computation libraries*

Similar to the sequential case, there are two main matrix computation libraries for MIMD computers with distributed memory.

PBLAS (Parallel BLAS) [3] contains parallel versions of the BLAS 3 routines. To execute an operation, all processors must call the same routine for matrix operands that are block cyclically mapped. For communication, PBLAS uses (internally) BLACS.

ScaLAPACK (Scalable LAPACK) [2] is the state-of-the-art library in parallel matrix com-

putation and contains parallel versions of almost all the routines of LAPACK. ScaLAPACK uses PBLAS for basic distributed operations and LAPACK for local computation.

Writing a program in ScaLAPACK is facilitated by the ease of modifying, to this purpose, a sequential LAPACK program. The names of the routines are changed by adding an initial 'P' and the arguments are also modified in a systematic way. The main concern is the proper distribution of the data onto the processor grid. ScaLAPACK contains at this moment some particularities that may harden the task of matrix mapping. However, ScaLAPACK offers a routine for matrix redistribution which may be very useful to cope with mapping problems, as we will show in section 4; for more information on redistribution algorithms, see [4, 14].

3. THE 4SID IDENTIFICATION METHOD

In this section, we outline a (sequential) direct 4SID method which determines the state-space model matrices A, B, C, D of a discrete-time dynamic linear system with m inputs and l outputs, using input-output data $u(k)$, $y(k)$, $k = 0 : N-1$. This and other similar methods for state-space identification can be found in [19, 22, 24], therefore we present only the information necessary for the understanding of the parallel version of the method.

The input-output data are organized as follows

$$\begin{cases} U = \left[\begin{array}{cccc} u^0 & u^1 & \cdots & u^{t-1} \end{array}\right] \in \mathbb{R}^{\alpha m \times t} \\ Y = \left[\begin{array}{cccc} y^0 & y^1 & \cdots & y^{t-1} \end{array}\right] \in \mathbb{R}^{\alpha l \times t} \end{cases}, \quad (1)$$

where t is the number of (time) windows and the input and output vectors $u(k) \in \mathbb{R}^m$ and $y(k) \in \mathbb{R}^l$ from a window are concatenated as

$$\begin{aligned} u^k &= \left[\begin{array}{c} u(k) \\ u(k+1) \\ \vdots \\ u(k+\alpha-1) \end{array}\right] \in \mathbb{R}^{\alpha m}, \\ y^k &= \left[\begin{array}{c} y(k) \\ y(k+1) \\ \vdots \\ y(k+\alpha-1) \end{array}\right] \in \mathbb{R}^{\alpha l}. \end{aligned} \quad (2)$$

The length of a window is α.

The basic functional relation on which the method is built reads

$$Y = Q_\alpha X + T_\alpha U, \quad (3)$$

where

$$\begin{aligned} Q_\alpha &= \left[\begin{array}{c} C \\ CA \\ \vdots \\ CA^{\alpha-1} \end{array}\right], \\ T_\alpha &= \left[\begin{array}{cccc} D & \cdots & 0 & 0 \\ CB & \ddots & 0 & 0 \\ \vdots & \ddots & \ddots & \vdots \\ CA^{\alpha-2}B & \cdots & CB & D \end{array}\right] \in \mathbb{R}^{\alpha l \times \alpha m} \end{aligned} \quad (4)$$

are the observability matrix and the causal block Toeplitz matrix of the discrete linear system, respectively. The matrix

$$X = \left[\begin{array}{cccc} x(0) & x(1) & \cdots & x(t-1) \end{array}\right] \in \mathbb{R}^{n \times t} \quad (5)$$

contains the state vectors of the system.

The identification problem is: determine four matrices A, B, C, D of appropriate dimensions satisfying (3), knowing the input-output data U and Y built as in (2), (1). The remarkable structure (4) of the matrices Q_α, T_α is intensively used.

The only hypothesis for solving the problem is that the input data windows are persistent, i.e. the matrix U is epic (rank$U = \alpha m$), and t is sufficiently large, more precisely $t \geq \alpha m + n$. The method obtains an observable state space model, so the observability matrix Q_α is monic (rank$Q_\alpha = n$), where n is the order of the desired model.

The idea of the method is to begin by determining n and Q_α, thus eliminating the contribution of the second term from din (3). To this purpose, we compute the right orthogonal factorization of the epic matrix U, i.e.

$$U\bar{V}^T = [\ L\ \ 0\ ] \tag{6}$$

where the matrix

$$\bar{V}^T = [\ \bar{V}_1^T\ \ \bar{V}_2^T\ ] \tag{7}$$

is orthogonal and L is lower triangualar and invertible. From (6) and (7), one can see that the block $\bar{V}_2^T$ represents an orthogonal basis for the subspace $\operatorname{Ker} U$, hence $U\bar{V}_2^T = 0$.

Multiplying (3) at right by $\bar{V}_2^T$, we obtain

$$Y\bar{V}_2^T = Q_\alpha(X\bar{V}_2^T). \tag{8}$$

In order to determine the monic matrix Q_α from (8), we compute the singular value decomposition (SVD) of the matrix $Y\bar{V}_2^T$, and truncate the singular values below a threshold ε, conveniently chosen. Let n be the number of significant singular values of $Y\bar{V}_2^T$. With obvious notations, after the truncation we obtain

$$\begin{aligned} Y\bar{V}_2^T &= [\ U_1^T\ \ U_2^T\ ] \begin{bmatrix} \Sigma_1 & \\ & 0 \end{bmatrix} \begin{bmatrix} V_1 \\ V_2 \end{bmatrix} \\ &= U_1^T \Sigma_1 V_1, \end{aligned} \tag{9}$$

where the diagonal block Σ_1 of size n contains the first n singular values of $Y\bar{V}_2^T$, and U_1^T has orthogonal columns. Consequently, we can take

$$Q_\alpha = U_1^T. \tag{10}$$

Therefore, n is the order of the state space model and U_1^T represents an orthogonal basis of the subspace $\mathcal{Y}_\alpha = \operatorname{Im} Q_\alpha$. The matrices A and C result from the definition of Q_α, i.e.

$$C = Q_\alpha(1:l,:), \tag{11}$$

and A is the least squares solution to the matrix equation

$$Q_{\alpha-1}A = Q_{2:\alpha}. \tag{12}$$

Next, in order to compute the matrix B and D, we use again (9); since U_2^T is the orthogonal complement of U_1^T, we have $U_2 Q_\alpha = 0$, see (10). Multiplying (3) at the left with U_2 and at the right with $\bar{V}_1^T$, we obtain the relation

$$U_2 Y\bar{V}_1^T = U_2 T_\alpha (U\bar{V}_1^T), \tag{13}$$

from where X disappeared. As, by construction, $U\bar{V}_1^T = L$ is lower triangular and invertible, the previous relation can be written simply as

$$U_2 T_\alpha = Y_2, \tag{14}$$

where we define

$$\begin{aligned} U_2 &= [M_1\ M_2\ \ldots\ M_\alpha], \\ (U_2 Y\bar{V}_1^T)L^{-1} = Y_2 &= [\ N_1\ \ N_2\ \ldots\ \ N_\alpha\ ], \end{aligned} \tag{15}$$

where $U_2 \in \mathbb{R}^{(\alpha l - n)\times \alpha l}$, $Y_2 \in \mathbb{R}^{(\alpha l - n)\times \alpha m}$.

Taking advantage of the block Toeplitz structure of the matrix T_α (where A and C are now known), the relation (14) can be written, after some processing we skip here for brevity, in the form

$$\begin{bmatrix} M_1 \cdots M_{\alpha-1} M_\alpha \\ M_2 \cdots\ M_\alpha \quad 0 \\ \vdots\ \ \vdots\ \ \vdots\ \ \vdots \\ M_\alpha \cdots\ 0 \quad 0 \end{bmatrix} \begin{bmatrix} I_l & 0 \\ 0 & Q_{\alpha-1} \end{bmatrix} \begin{bmatrix} D \\ B \end{bmatrix} = \begin{bmatrix} N_1 \\ N_2 \\ \vdots \\ N_\alpha \end{bmatrix} \tag{16}$$

from which B and D can be found by solving a least squares problem.

To summarize, the direct 4SID procedure for building the state space model (A, B, C, D), using the relation (3), is the following.

Algorithm 1 (**4SIDD**) (Input: the size of a window α, the number of windows t, the input-output data of the linear system, organized in the matrices U, Y, with the structure (2), (1), and a tolerance ε used to neglect the small singular values. Output: the order n and a state space model (A, B, C, D).)

1. Compute the right orthogonal triangularization of the matrix U^T, i.e.

$$\bar{V}U^T = \begin{bmatrix} L^T \\ 0 \end{bmatrix},$$

where, following (6) and (7), the matrix

$$\bar{V} = \begin{bmatrix} \bar{V}_1 \\ \bar{V}_2 \end{bmatrix}$$

is a sequence of elementary reflectors and L^T is upper triangular and invertible.

2. Build the matrix $Z_2 = Y\bar{V}_2^T$ and compute the truncated SVD (9), choosing the order n such that $\sigma_n > \varepsilon > \sigma_{n+1}$. Only the blocks U_1 and U_2 from (9) are stored for future use.
3. $Q_\alpha = U_1^T$.
4. $C = Q_\alpha(1:l, :)$.
5. Compute the least squares solution A of the linear matrix equation (12).
6. Compute the matrix $Y_2 = (U_2 Y \bar{V}_1^T)L^{-1}$ and partition U_2 and Y_2 according to (15).
7. Find the least squares solution (B, D) of the linear matrix equation (16).

Despite its apparent complexity — mainly due to the complexity of the problem — the algorithm 4SIDD uses standard techniques of numerical computation (orthogonal triangularization, SVD etc.) and is suited to sequential or parallel implementation using standard libraries, such as LAPACK or ScaLAPACK.

4. PARALLEL ALGORITHM AND IMPLEMENTATION

We propose in this section a parallel version of the algorithm 4SIDD, using as much as possible the routines of the library ScaLAPACK. The algorithm 4SIDD can be naturally split into two parts:

- the steps 1-6 (and 7, partially), having a straightforward implementation by calls at ScaLAPACK routines;
- the construction, in step 7, of the matrix of the system (16), operation that could be performed in several ways; a difficulty related to this part is the unfavorable initial distribution of the matrices.

The two parts will be treated distinctly in the sequel.

4.1. *ScaLAPACK implementation details*

The input data of the algorithm 4SIDD are the matrices U and Y (of input-output data of the system to be identified), of size $\alpha m \times t$, and $\alpha l \times t$, respectively. Remind that $t > \alpha \max(m, l)$ and, practically, t has a value significantly larger than $\alpha \max(m, l)$. The matrices U and Y are block cyclically distributed over the processor grid; the blocks have size $m_b \times n_b$;

The steps 1-6 of the algorithm 4SIDD are implemented via calls to ScaLAPACK routines. For the sake of completeness, we list below the names of the routines and the steps where they are used.

PDGELQF orthogonal LQ factorization, in step 1, relation (6);

PDORMLQ application of the corresponding transformations, to compute $Y\bar{V}^T$, in step 2, the left hand side term of (8) and in step 6; these two transformations are condensed in a single function call, in order to increase the parallelism;

PDGESVD singular value decomposition, in step 2, relation (9));

PDGELS orthogonal QR factorization and least squares solution of overdetermined linear systems, in steps 5 and 7.

There are also two PBLAS routines that are used, namely **PDGEMM** (general matrix multiplication) and **PDTRSM** (triangular system solution), both in computing the matrix Y_2 from (15), in step 6.

Although specifying the arguments of the routines above is a simple (but somewhat tedious) task, some data alignment constraints require special attention. For many ScaLAPACK and PBLAS routines, the initial indices `IA` and `JA` of the submatrix of interest, have to correspond to the upper left corner of a block, i.e., for instance, `IA` $= km_b + 1$, for a positive integer k (the indices start at 1). This condition is often restrictive, as we notice for our algorithm. We will give three examples showing the impact of this alignment constraint;

of course, in order to meet the alignment requirements, the general solution is the redistribution of the matrix, a communication operation that is implemented by the routine **PDGEMR2D** and that could be costly.

1. The free term of the system (12) is the matrix denoted $Q_{2:\alpha}$ and represents the submatrix starting with row $l+1$ of the matrix U_1^T of the SVD (9). As an output of the routine PDGESVD, the factor U_1^T is aligned with its first element at a block beginning, which is very convenient, since U_1^T is necessary in step 5 of algorithm 4SIDD, as the matrix of the linear system (12). However, for $Q_{2:\alpha}$ to be aligned, l should be a multiple of m_b; this is a relatively tough constraint, since l has usually small values, while m_b is chosen sufficiently large in order to have efficient local BLAS operations (typically $m_b = 16, 32, \ldots$).

2. In the same SVD (9), the block U_2^T is used in the computation of the matrix Y_2 from (15). The block U_2 starts at column $n+1$ of the orthogonal factor from (9); opposite to the previous case, where the dimensions l and m_b are initially known, now n is found during the computation, function of the data and the tolerance ε. Thus, the redistribution of the matrix U_2^T must be performed when n is not a multiple of n_b.

A solution not requiring redistribution is to take instead of n the first multiple of n_b greater than n, let it be n'; this solution could be accepted only if $\sigma_n \approx \sigma_{n'}$ (obviously, we have $\sigma_n > \varepsilon > \sigma_{n'}$). Although a reduction of the execution time is expected, this idea should be applied carefully, since the dimension of the identified system is affected.

3. Another alignment constraint is related to the computation of the SVD of matrix $\bar{Y}_2 = Y\bar{V}_2^T$ from (8). Since $Y\bar{V}^T$ is computed in place in Y, the submatrix $\bar{Y}_2$ starts in column $\alpha m+1$. The routine PxGESVD requires that the first element of the argument (sub)matrix, in our case $\bar{Y}_2$, be placed on the diagonal of a local block. This condition can be fulfilled only if αm is a multiple of n_b, which is not very restrictive.

4.2. *Parallel computation of the matrix product (16)*

We will detail here the parallel algorithm for building the system (16), rewritten as

$$S \begin{bmatrix} D \\ B \end{bmatrix} = N, \tag{17}$$

where the matrices S and N are obvious notations. (Remind that the system itself is solved by a call to the routine PDEGELS from ScaLAPACK.)

In order to build N, only a redistribution of the matrix Y_2 from (15) is necessary. To obtain S is a more difficult task, since a matrix product must be computed, in which the operands are matrices with particular structures; an efficient algorithm should take advantage of these structures, opposite to the easy solution of calling directly the routine PDGEMM (thus ignoring the structure).

As a principle, we chose to design an algorithm which is efficient on a small number of processors; this is a less scalable solution. This approach is supported by the sizes of matrices for which parallel computation (particularly with ScaLAPACK) is efficient; a common recommendation is to choose the number of processors such that each processor have in its local memory at least one million matrix elements. For identification problems, such matrix dimensions are usually large, that is a small number of processors is usually needed.

This principle, and further reasons given below, led to the choice of the ring as virtual parallel architecture. A first reason was the shape of matrices S and N, which have a much more rows than columns. A natural distribution of these matrices is on rows, i.e. a local block contains all the columns of the matrices. Therefore, the processors ring may be seen as a *column* of processors, topology denoted $\mathcal{C}$. For ease of manipulation, we will also use a *row* ring, denoted $\mathcal{R}$; the initial processor grid, whose structure was preserved in the implementation of steps 1-6 of algorithm 4SIDD, is denoted $\mathcal{G}$.

A second reason is the size of the system (16), relatively small with respect to the dimen-

sions of the data matrices U and Y. Solving the system (16) has a small weight in the total number of operations, moreover if the matrices U_2 and Y_2 from (15) are truncated as suggested in the previous section.

Let us first discuss about building the matrix N. We notice that

$$Y_2^T = \begin{bmatrix} N_1^T \\ N_2^T \\ \vdots \\ N_\alpha^T \end{bmatrix}, \qquad N = \begin{bmatrix} N_1 \\ N_2 \\ \vdots \\ N_\alpha \end{bmatrix}, \tag{18}$$

which shows that N can be obtained by transposing Y_2, and then transposing — *locally*, if each block N_i belongs to a single processor — the blocks N_i. However, there is an alternative solution implying the same complexity of communication, but avoiding the double transposition.

We first redistribute the matrix Y_2 from the grid $\mathcal{G}$ onto the row ring $\mathcal{R}$, modifying also the block size; on the grid, the size was $m_b \times n_b$, theoretically independent of the input data; on the ring $\mathcal{R}$, we use blocks of size $(\alpha l - n) \times m$, such that a block N_i be mapped onto a single processor. (Of course, the distribution of Y_2 is perfectly balanced if α is a multiple of p.) The redistribution is implemented by a call to the routine PxGEMR2D.

If we look now at the processor ring as a column $\mathcal{C}$, we remark that each block N_i is on the correct processor; if $\alpha = p$, then the redistribution is finished; otherwise, the local blocks must be rearranged from the row major order into a column major order, operation which is entirely local. For instance, in the case $p = 4$, $\alpha = 8$, the processor number 0 should perform the operation

$$[\ N_1\ N_5\ ] \longrightarrow \begin{bmatrix} N_1 \\ N_5 \end{bmatrix}.$$

We describe now the computation of the matrix S from (17), split into three stages.

1. The matrix U_2 from (15) is redistributed into

$$M = \begin{bmatrix} M_1 \\ M_2 \\ \vdots \\ M_\alpha \end{bmatrix}, \qquad M_i \in \mathbb{R}^{(\alpha l - n) \times l}, \tag{19}$$

from the grid $\mathcal{G}$ onto the column ring $\mathcal{C}$, using the same technique as described above for passing from Y_2 to N; now, the size of the local blocks M_i is $(\alpha l - n) \times l$.

2. Next, the matrix $Q_{\alpha-1}$ is redistributed in the same way as M, i.e.

$$Q_{\alpha-1} = \begin{bmatrix} W_1 \\ W_2 \\ \vdots \\ W_{\alpha-1} \end{bmatrix}, \qquad W_i \in \mathbb{R}^{l \times n}. \tag{20}$$

$Q_{\alpha-1}$ is distributed on $\mathcal{C}$ such that each block W_i is local to a single processor.

3. Finally, the actual multiplication is performed. Using the definition (16), the matrix S can be written as

$$S = \begin{bmatrix} S_{11} & S_{12} \\ S_{21} & S_{22} \\ \vdots & \vdots \\ S_{\alpha 1} & S_{\alpha 2} \end{bmatrix}, \tag{21}$$

$$S_{i1} = M_i,\ S_{i2} = \textstyle\sum_{k=i+1}^{\alpha} M_k W_{k-i}.$$

The first block column of S is already available from step 2 of the algorithm 4SIDD. The blocks S_{i2} from the second column are computed via a convolution, as described below.

At the first step, the matrix M, i.e. the first block column of S, is shifted cyclically upwards with one block on the processor ring $\mathcal{C}$; after the shift, the pairs of blocks necessary to compute the block

$$S_{12} = M_2 W_1 + M_3 W_2 + \ldots + M_\alpha W_{\alpha-1}$$

are on the same processor: M_2 and W_1 on processor 0, M_3 and W_2 on processor 1 etc. Each processor computes the products $M_i W_{i-1}$ of its local blocks; then, S_{12} can be computed

via a global sum over the whole ring of processors (we remark that this operation is in no way constrained by the virtual ring topology, but may be executed more efficiently depending on the properties of the target computer). Due to the cyclic block mapping, together with S_{12}, the blocks $S_{p+1,2}$, $S_{2p+1,2}$ etc., can be also computed.

At the second step, the matrix M is again shifted cyclically upwards on the ring, thus allowing, as described above, the computation of the blocks S_{22}, $S_{p+2,2}$ etc.

After p steps, the matrix M is brought to its initial distribution and the second block column of S is entirely computed. The detailed algorithm is the following.

Algorithm 2 (**S**) (Input data: the matrices M from (19) and $Q_{\alpha-1}$ from (20), block cyclically distributed on a ring with p processors. Output: the matrix S from (21). The algorithm is written for processor i, whose local blocks are initially M_{i+1}, M_{p+i+1}, M_{2p+i+1} etc. and W_{i+1}, W_{p+i+1}, W_{2p+i+1} etc. The processors "up" and "down" with respect to i have the addresses $(i-1) \bmod p$, and $(i+1) \bmod p$, respectively.)

1. `For` $\ell = 1 : p$
   1. send "up" the (currently) local blocks of M
   2. receive from "down" the blocks $M_{\ell+i+1}$, $M_{p+\ell+i+1}$, $M_{2p+\ell+i+1}$ etc.
   3. `For` $j = 0 : \frac{\alpha-\ell}{p}$ (participate at the computation of $S_{\ell+jp,2}$)
      1. compute the local product
         $$T = M_{\ell+jp+i+1}W_{i+1} + M_{\ell+(j+1)p+i+1}W_{2i+1} + \dots$$
      2. participate at the global sum of the blocks T; the destination is processor $\ell - 1$ (the result is $S_{\ell+jp,2}$)

All the operations of this algorithm are performed by calls to library routines, guaranteeing a good efficiency in general conditions. The communication of matrix blocks is implemented through two BLACS routines, namely xGESD2D and xGERV2D. The global sum is computed by the BLACS routine xGSUM2D. The local products are computed with the BLAS routine xGEMM.

Analyzing the algorithm, we remark immediately that the computation is balanced. Naturally, when computing a block S_{i2}, some processors have to compute one more block product that the others; if $\alpha \gg p$, as usual for small number of processors, this unbalance has a small effect on the efficiency.

Looking now at the complexity of communication, we notice that the matrix M passes entirely through each processor, implying a volume of $(\alpha l - n)\alpha l$ transmitted matrix elements. To this, we should add the communication required by the computation of the global sums; denoting $\gamma = (\alpha l - n)\alpha n$ the number of elements in the second block column of S, this communication volume may vary between $\gamma \log p$ and γp, depending on the communication algorithm and the topology of the target computer. In both cases, if p is sufficiently small, the total communication volume is small with respect to the computation effort.

Finally, let us remark that the algorithm **S** can be easily generalized from a ring to a grid topology. We sketch here the main modifications to be done to this purpose. We suppose that the matrices M and $Q_{\alpha-1}$ have the same (block cyclic) distribution over the grid, and that each block M_i (or W_i) is distributed over a row of processors, in the grid. The matrix M is shifted cyclically upwards, as in algorithm **S**, on each column of the grid; hence, the communication volume is smaller than on ring. However, in order to compute the products T, the processors on a same row of the grid must cooperate, and the communication pattern is a complete exchange; this implies more communication than on ring. The global sum is performed on the columns of the grid, thus more efficiently than on ring.

All these considerations lead us to the assumption that, for a sufficiently large number of processors, e.g. in the tens, the (modified as above) algorithm **S** will be more efficient on grid than on ring. For a small number of processors (< 10), however, we expect the ring variant to be superior, although we have performed no experimental comparison with the

grid variant.

5. EXPERIMENTAL RESULTS

The parallel version of the algorithm 4SIDD, described in the previous section, was implemented in C, using calls to the libraries ScaLAPACK, PBLAS, LAPACK, BLAS and BLACS, in their double precision versions.

The program was tested on a 32 processors IBM SP1 computer located at LMC-INPG (Grenoble, France). Each processor has a peak performance of about 100 Mflops/s, and a top communication speed of about 30 Mbit/s; hence, the computation speed is relatively high with respect to communication speed; as a consequence, very good performance is obtained for large matrices. The processors are connected by two crossbar (programmable) switches and thus virtually any two processors may be directly connected; however, at a given moment, only a limited number of direct connections may be realized.

In our experiments, we monitored the effect of some variables which affect the parallel performance, i.e.

- the sizes of matrices U and Y, see (1), namely the number of time windows t, the length of a window α and the number of inputs m and outputs l of the considered system;
- the sizes of the block in the block cyclic distribution of the matrices U and Y on the processor grid, i.e. m_b and n_b. These block sizes are constant through the whole algorithm, excepting step 7;
- the number of processors p, and the configuration of the virtual grid, i.e. the number of rows p_r and columns p_c.

The performance of the parallel program was measured by the speed-up

$$c_p = \frac{T_s}{T_p}, \tag{22}$$

where T_s is the execution time of the (best) sequential program, in our case a LAPACK based implementation, and T_p is the parallel execution time. Another performance criterion, normalized with respect to the number of processors, is the efficiency

$$\varepsilon_p = \frac{T_s}{pT_p}. \tag{23}$$

The efficiency has (theoretically) subunitary values; the value 1 corresponds to the ideal case when the computation is perfectly distributed to the p processors and there are no overheads.

Before giving a summary of the experimental results, let us present some general conclusions that are suited also to our algorithm:

- increasing the size of the input matrices increases the efficiency;
- on the contrary, increasing the number of processors decreases the efficiency;

The authors of ScaLAPACK appreciate that a good efficiency could be obtained only when a processors holds in its memory at least one million elements of the matrix to be processed. For the IBM SP1, which has slow communication, even larger matrices could be necessary.

Other general appreciations are the following.

- The block sizes, m_b and n_b, usually have an optimum value which maximizes the efficiency, on a given computer, for a given matrix size. If the blocks are small, then the (local) BLAS routines have small performance, but the natural parallelism of the problem is high, due to the large number of blocks. If the blocks are large, then the parallelism is poor, but the local routines are efficient. The optimum is given by a compromise of these tendencies.
- The shape of the virtual grid, i.e. the ratio of p_r and p_c, has interesting effects; of course, we consider $p = p_r p_c$ constant. The ScaLAPACK routines have generally a better performance when $p_c > p_r$; this feature is caused by the nature of numerical algorithm for matrices 2D block cyclically distributed; in these algorithms, the highly sequential operations (representing bottlenecks), are performed by processors on the

same column; thus, grids with "small" columns are favored.

For PBLAS routines, especially matrix multiplication, square grids are the most suited. Moreover, PBLAS routines are less sensitive at the shape of the grid than ScaLAPACK routines.

We now present the most significant experimental results, on three data sets, ordered increasingly upon the sizes of the matrices U and Y (here n is the actual order of the system):

D1. $t = 1500, m = l = 10, \alpha = 40, n = 100$;

D2. $t = 2000, m = l = 10, \alpha = 50, n = 100$;

D3. $t = 3000, m = l = 10, \alpha = 80, n = 100$.

The best execution times T_p are presented in table 1, function of the number of processors T_p, together with the corresponding speed-ups c_p. A graph of the speed-ups for the three data sets is presented in figure 1. We notice that, naturally, the performances are improved when data matrices are larger. For the set **D3**, an anomaly can be observed: a speed-up greater than 2 for $p = 2$; however, there is a simple explanation of this fact. Due to the large sizes, the matrices cannot be stored in the local memory of one processor; the mechanism of virtual memory is therefore used, slowing down the computation; on two processors or more, the local data have smaller size, thus the local memory is sufficient.

As announced, the efficiency decreases as the number of processors is increased, as shown by figure 2. The values of the efficiency could be considered satisfactory for a small number of processors, especially for the set **D3**.

To give an idea on the optimum block size, we present in table 2 two sequences of execution times for the set **D2**, for blocks of size 10×10 and 20×20. It can be noticed that for a small number of processors, the larger size gives the best results, which is explained by the good behavior of BLAS 3 routines; when the number of processors increases, the smaller block size is favored, as the degree of parallelism is higher.

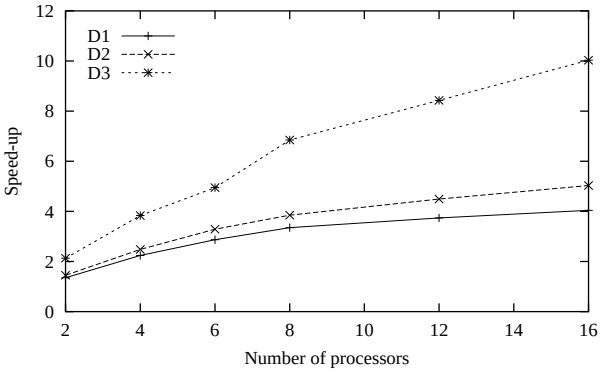

Figure 1: Speed-ups for the three experimental data sets.

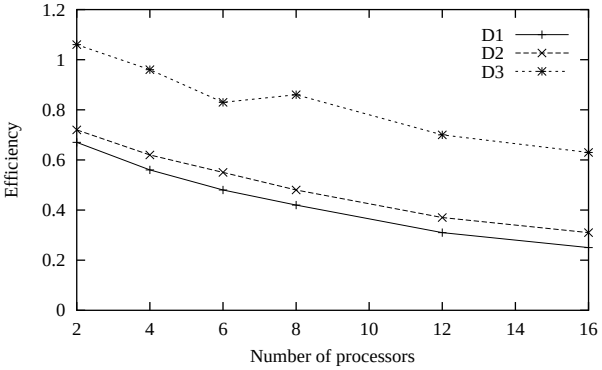

Figure 2: Efficiencies for the three experimental data sets.

A glimpse of the effect of the grid shape (the ratio of p_r and p_c) on the execution time is presented in table 3, for the data set **D2**, with $m_b = n_b = 10$. It can be noticed that more "horizontal" grids favor the efficiency. The most important differences appear for the case $p = 4$, when the grid 1×4 is significantly better than the grid 2×2. Some measurements on grids with $p_r > p_c$ showed that this shape is not efficient.

Since our program calls several ScaLAPACK and PBLAS routines, with matrices of several sizes, we considered necessary to study what routines (or program parts, generally) are the most time consuming and what is their scalability (i.e. the property of preserving a good efficiency as the number of processors increases). To this purpose, we split our pro-

Table 1: Execution times and speed-ups for the parallel 4SIDD algorithm.

	p	1	2	4	6	8	12	16
D1	T_p	54.6	40.6	24.4	19.0	16.3	14.6	13.5
	c_p		1.35	2.24	2.87	3.35	3.74	4.04
D2	T_p	104.7	72.2	42.3	31.8	27.2	23.3	20.8
	c_p		1.45	2.48	3.29	3.85	4.49	5.03
D3	T_p	656	308	171.5	132.6	95.8	77.8	65.4
	c_p		2.13	3.83	4.95	6.85	8.43	10.03

Table 2: Execution times for the set **D2**, for two block sizes.

	p	2	4	6	8
D2	$m_b = n_b = 10$	82.2	42.5	31.8	27.2
	$m_b = n_b = 20$	72.2	42.3	32.3	28.5

gram into three parts:

1. The orthogonal right triangularization of the matrix U (6) and the application of the corresponding transformations on Y from (8) and (15).
2. The SVD of $Y\bar{V}_2^T$ (9).
3. The other operations, i.e. steps 3-7 of algorithm 4SIDD.

This split is natural for the following reasons. The first two parts operate on large matrices, opposite to the third part, where relatively small matrices appear. Moreover, the SVD is the most costly operation (in terms of flops), thus it is interesting to study its weight on the total execution time.

Table 4 presents the weight of the execution times of each part of the program, for the three sets of data. The time required by the SVD represents slightly less than half the total time, and thus the efficiency of the program depends crucially on the efficiency of the ScaLAPACK routine PxGESVD. The first part, representing the orthogonal LQ factorization and the application of the transformations, is the most scalable of the three parts, its weight obviously decreasing as the number of processors increases. On the contrary, the efficiency of the last part decreases as p grows; this is due mainly to the relatively small sizes of the matrices used in this part, but also to the matrix multiplication algorithm **S**, chosen especially for performance on small p.

Concluding this section, let us appreciate that the experimental results show a convenient efficiency of our program, thus confirming our approach of using ScaLAPACK as a base and also the good performance of our new routines for matrix multiplication. The fact that sufficiently large matrices lead naturally to a good efficiency is confirmed, but we also showed that the performance can be improved by a careful choice of some parameters, such as the size of the blocks and the shape of the processor grid.

6. REFERENCES

[1] Anderson, E., Bai, Z., Bischof, C., Demmel, J., Dongarra, J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenney, A., Ostrouchov, S., and Sorensen, D. "LAPACK Users' Guide, Second Edition". SIAM, 1995.

[2] Blackford, L.S., Choi, J., Cleary, A., D'Azevedo, E., Demmel, J., Dhillon, I., Dongarra, J.J., Hammarling, S., Henry, G., Petitet, A., Stanley, K., Walker, D.W., and Whaley, R.C.

Table 3: Execution times for the data set **D2**, for several sizes and shapes of the processor grid.

$p_r \times p_c$	2×2	1×4	2×4	1×8	4×4	2×8
T_p	50.6	42.5	29.2	27.2	22.4	20.8

Table 4: The weights, in percents, of the three parts of the program (100% is the total execution time).

Part		1	2 (SVD)	3
D1	$p = 4$	29.7	44.5	25.8
	$p = 8$	27.2	46.5	26.3
	$p = 16$	23.5	45.0	31.5
D2	$p = 4$	32.8	48.0	19.2
	$p = 8$	31.0	48.1	20.9
	$p = 16$	27.3	47.6	25.1
D3	$p = 4$	38.1	42.9	19.0
	$p = 8$	33.9	44.1	22.0
	$p = 16$	34.1	43.6	22.3

"ScaLAPACK Users' Guide". SIAM, 1997.

[3] Choi, J., Dongarra, J.J., Ostrouchov, S., Petitet, A., Walker, D. and Whaley, R.C. "LAPACK Working Note 100, A Proposal for a Set of Parallel Basic Linear Algebra Subprograms". Technical Report CS-95-292, University of Tennessee, Knoxville, 1995.

[4] Desprez, F., Dongarra, J.J., Petitet, A., Randriaramaro, C., and Robert, Y. "LAPACK Working Note 120, Scheduling Block-Cyclic Array Redistribution". Technical Report CS-97-349, University of Tennessee, Knoxville, 1997.

[5] Dongarra, J.J., Duff, I., Du Croz, J., and Hammarling, S. "A Set of Level-3 Basic Linear Algebra Subprograms". *ACM Trans.Math.Software*, vol.16, pp.**1–17, 18–28**, 1990.

[6] Dongarra, J.J., and Whaley, R.C. "LAPACK Working Note 94, A User's Guide to the BLACS v1.0". Technical Report CS-95-281, University of Tennessee, Knoxville, 1995.

[7] MPI Forum. "A message passing interface standard". *Int. J. Supercomputer Appl. and High Perf. Comp.*, vol. 8, 1994. (Special issue on MPI).

[8] Geist, A., Beguelin, A., Dongarra, J.J., Jiang, W., Mancheck, R., and Sunderam, V. "PVM: Parallel Virtual Machine. A User's Guide and Tutorial for Networked Parallel Computing". MIT Press, 1994.

[9] Jora, B., Popeea, C., and Popescu, D. 4SID Identification for Control. In *Proc. Int. Conf. Control Systems and Computer Science*, volume 1, pp.**132–137**, Bucharest, Romania, 1997.

[10] Liu, H., Xu, G., Tong, L., and Kailath, T. "Recent Developments in Blind Channel Equalization: from Cyclostationarity to Subspaces". *Signal Processing*, vol.50, pp.**83–99**, 1996.

[11] Ljung, L., and McKelvey, T. "Subspace Identification from Closed Loop Data". *Signal Processing*, vol.52, pp.**179–194**, 1996.

[12] MacInnes, C.S., and Vaccaro, R.J. "Tracking Directions-of-arrival with In-

variant Subspace Updating". *Signal Processing*, vol.50, pp.**137–150**, 1996.

[13] McKelvey, T., Akcay, T., and Ljung, L. "Subspace Based Identification of Infinite-Dimensional Multivariable Systems from Frequency-response Data". *Automatica*, vol.32, no.6, pp.**885–902**, 1996.

[14] Petitet, A., and Dongarra, J.J. "LAPACK Working Note 133, Algorithmic Redistribution Methods for Block-Cyclic Decompositions". Technical report, University of Tennessee, Knoxville, 1998.

[15] Picci, G., and Katayama, T. "Stochastic Realization with Exogenous Inputs and 'Subspace-Methods' Identification". *Signal Processing*, vol.52, pp.**145–160**, 1996.

[16] Porumbescu, A., Dobrescu, R., Jora, B., and Popeea, C. "Patient Specific Expert System for IDDM Control". In *Proc. Int. Conf. Control Systems and Computer Science*, vol.1, pp.**132–137**, Bucharest, Romania, 1997.

[17] Snir, M., Otto, S.W., Huss-Lederman, S., Walker, D.W., and Dongarra, J.J. "MPI: The Complete Reference". MIT Press, 1996.

[18] Van Overschee, P., and De Moor, B. "Subspace Algorithms for the Stochastic Identification Problem". *Automatica*, vol.29, no.3, pp.**649–660**, 1993.

[19] Van Overschee, P., and De Moor, B. "N4SID. Subspace Algorithms for the Identification of Combined Deterministic-Stochastic Systems". *Automatica*, vol.30, no.1, pp.**75–93**, 1994.

[20] Van Overschee, P., and De Moor, B. "Choice of State-Space Basis in Combined Deterministic-Stochastic Subspace Identification". *Automatica*, vol.31, no.12, pp.**1877–1883**, 1995.

[21] Verhaegen, M. "A Novel Non-iterative MIMO State-Space Model Identification Technique". In *Proc. 9-th IFAC/IFORS Symposium on Identification and System Parameter Estimation*, pp.**1453–1458**, Budapest, Hungary, 1991.

[22] Verhaegen, M. "Identification of the Deterministic Part of MIMO State Space Models Given in Innovations Form from Input-Output Data". *Automatica*, vol.30, no.1, pp.**61–74**, 1994.

[23] Verhaegen, M., and Yu, X. "A Class of Subspace Model Identification Algorithms to Identify Periodically and Arbitrarily Time-Varying Systems". *Automatica*, vol.31, no.2, pp.**201–216**, 1995.

[24] Viberg, M. "Subspace-based Methods for the Identification of Linear Time-Invariant Systems". *Automatica*, vol.31, no.12, pp.**1835–1851**, 1995.