Systolic Array for Solving Toeplitz Systems of Equations

J. Chun, V. Roychowdhury and T. Kailath †

Information Systems Laboratory
Stanford University
Stanford, CA 94305

Abstract

Many problems of geophysics, image processing and time series analysis involve the problem of solving Toeplitz systems of equations. We present a fast parallel $O(mn)$ algorithm that solves both square and over-determined Toeplitz systems of equations. The solution is obtained directly from the triangular factorization without using back-substitution. This avoids separate factorization and back-substitution sections, which complicate architectural implementation. This also enables us to eliminate intermediate memory to store the triangular factor. The parallel implementation is carried out in two steps. First, Regular Iterative Algorithms (RIAs) for solving Toeplitz system of equations are formulated systematically from the mathematical description of our algorithm. The advantage of having RIAs is that the process of mapping the algorithms on regular processor arrays can be done in a systematic manner.

1. Introduction.

Large Toeplitz systems of equations,

$$T\mathbf{x} = \mathbf{b}, \quad T \in \mathbf{R}^{m\times n}, \quad \mathbf{b} \in \mathbf{R}^{n\times 1}, \tag{1}$$

where T is either scalar or block Toeplitz, arise in various areas including geophysics and image processing applications.

First let us consider $n \times n$ square Toeplitz systems of equations. The square Toeplitz system of equations (1) can be solved in $O(n^2)$ operations by using either Levinson algorithm or Schur algorithm. Levinson algorithm computes the triangular factorization, $T^{-1} = UU^T$, as well as the solution, whereas Schur algorithm computes the triangular factorization, $T = LL^T$.

It has been noted [6][8] that Schur algorithm is more suitable for parallel implementation than Levinson algorithm that needs inner-product operations. Kung and Hu [8] implemented a systolic array based on Schur algorithm.

However, the algorithms that factorize $T = LL^T$ does not directly give the solution, and a straight-forward application would need forward elimination, $L\mathbf{y} = \mathbf{b}$, and back-substitution, $L^T\mathbf{x} = \mathbf{y}$. A slight modification of Schur algorithm (or Bareiss algorithm) can give $\mathbf{y} = L^{-1}\mathbf{b}$ simultaneously during the factorization of $T = LL^T$, but the back-substitution step cannot easily avoided. If one closely examines the computational dependency among the three steps, factorization $T = LL^T$, forward elimination and back-substitution, then one can easily see that back-substitution step cannot be pipelined with the factorization step and forward elimination step. Implication of this fact is that we need $O(n^2)$ first-in-last-out (FILO) storages [8] to store the Cholesky factor L until we can start the back-substitution. Therefore, this approach has a serious drawback for large size systems of equations.

For general square systems, $A\mathbf{x} = \mathbf{b}$, Delsome and Ipsen [4] showed how to avoid back-substitution using hyperbolic rotations. Nash [9] also presented an algorithm, based on Faddeeva's method, that does not need back-substitution. Similar idea was discovered independently by Deprettere and Jainandunsing [5]. The underlying idea of avoiding back-substitution is to compute (at least implicitly) a factorization of A^{-1}.

For square Toeplitz systems, Brent and Luk [2] obtained a scheme that can "regenerate" the upper triangular matrix L^T in such a way that the back-substitution step can be pipelined with the factorization step. Deprettere and Jainandunsing [5], and Delsome and Ipsen [4] also obtained algorithms based on Schur algorithm that can avoid back-substitution.

For rectangular Toeplitz systems, Bojanczyk, Brent and de Hoog [1] used a similar regeneration method together with their fast QR factorization algorithm to pipeline the back-substitution step.

† This work was supported in part by the National Science Foundation under Grant MIP-21315-A2, the U.S. Army Research Office under Contract DAAL03-86-K-0045, and the SDIO/IST, managed by the Army Research Office under Contract DAAL03-87-K-0033.

In this paper, we shall give a unified algorithm and an architecture that can solve both rectangular Toeplitz systems of equations and square Toeplitz systems without back-substitution. Our method is very straight-forward, and therefore, has much simpler data flow compared with the previous works [1],[4],[5].

Our algorithm can be easily implemented in parallel as will be shown, and needs $O(n)$ processors, $O(\alpha n)$ storages and $O(\alpha n)$ time. The systolic array design is done in a hierarchical fashion using the theory of Regular Iterative Algorithms (RIAs) [10]. First, the algorithm is converted to a Single Assignment Code (SAC) [11], where every variable is an indexed variable and is assigned a unique value during execution. The SAC is then systematically converted into an RIA by removing global dependencies. One can then apply formal mapping techniques to obtain various systolic arrays by projecting the dependence graph of the algorithm along different *iteration vectors.*

In Sec 2, we shall provide a brief background about the concept of *displacement.* The algorithm is explained in Sec 2. Step by step explanation of obtaining systolic arrays for our algorithm is given in Sec 3.

2. The Algorithm.

After a brief introduction of the concept of *displacement* [7], we shall apply the generalized Schur algorithm [3] to solve strongly nonsingular Toeplitz (i.e., all leading principal submatrices are nonsingular) systems as well as full column rank over-determined Toeplitz systems. Proofs as well as other interesting results can be found in [3].

Generalized Displacement.

Let $A \in \mathbf{R}^{m\times n}$ be a given matrix, and let $F^f = \bigoplus_{i=1}^{M} Z_{m_i} \in \mathbf{R}^{m\times m}$ and $F^b = \bigoplus_{i=1}^{N} Z_{n_i} \in \mathbf{R}^{n\times n}$, where $\oplus$ denotes the concatenated direct sum ($A \oplus B = \mathrm{diag}[A, B]$), and Z_n denotes the $n \times n$ shift-down matrix (1's along the first sub-diagonal, and 0's elsewhere). The rank-α matrix

$$\nabla_{(F^f, F^b)} A \equiv A - F^f A F^{bT} \tag{2}$$

is called the *displacement* of A with respect to *displacement operator* $\{F^f, F^b\}$. Any matrix pair, $\{X, Y\}$ such that

$$\nabla_{(F^f, F^b)} A = XY^T, \quad X \equiv [\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_\alpha], \quad Y \equiv [\mathbf{y}_1, \mathbf{y}_2, \ldots, \mathbf{y}_\alpha]$$

is called a *generator* of A (with respect to $\{F^f, F^b\}$). The number α is called the *length* of the generator (with respect to $\{F^f, F^b\}$). A generator of A with the minimal possible length is called a *minimal* generator. The length of the minimal generator of A (i.e., $\mathrm{rank}(\nabla_{(F^f, F^b)} A)$) is called the *displacement rank* of A (with respect to $\{F^f, F^b\}$).

Note that the displacement of a symmetric matrix A can be written as $\nabla_{(F^f, F^f)} A = X \Sigma X^T$ where Σ is a diagonal matrix with 1 or -1 along the diagonal, and therefore, has a *symmetric generator,* $\{X, X\Sigma\}$.

Generalized Schur Algorithm.

Given a generator of a matrix $M \in \mathbf{R}^{m\times n}$ with respect to $\{F^f, F^b\}$,

$$M = \begin{bmatrix} A & B \\ C & D \end{bmatrix}, \quad A \in \mathbf{R}^{r\times r}, \quad D \in \mathbf{R}^{(m-r)\times(n-r)},$$

where we assume that A is strongly nonsingular (i.e., all leading principal minors of A are nonsingular), the r-step generalized Schur algorithm computes the matrices L, U as well as a generator of $M^{(r)}$ with respect to $\{\bar{F}^f, \bar{F}^b\}$ (where $\bar{F}$ denotes the matrix obtained after deleting the first r rows and columns of F) of the following *partial triangular factorization,*

$$M = LU + \begin{bmatrix} O & O \\ O & M^{(r)} \end{bmatrix}, \quad L = \begin{bmatrix} L_1 \\ W \end{bmatrix}, \quad U = [U_1, V],$$

where $L_1, U_1^T \in \mathbf{R}^{r\times r}$ are lower triangular matrices. Generalized Schur algorithm needs $O(\alpha mn)$ floating point operations (flops), where α denotes the displacement rank of M. Notice that

$$M^{(r)} = D - CA^{-1}B.$$

The matrix $M^{(r)}$ is called the *Schur complement of A in M*. Before describing the Schur algorithm, let us define the

matrices called *spinors*. A spinor $S_{(j|i)} \in \mathbf{R}^{\alpha\times\alpha}$ is defined as the identity matrix except for the following 4 entries;

$$[S_{(j|i)}]_{i,i} = c, \quad [S_{(j|i)}]_{i,j} = s_2, \quad [S_{(j|i)}]_{j,i} = -s_1, \quad [S_{(j|i)}]_{j,j} = c, \quad c^2 + s_1 s_2 = 1,$$

where $[A]_{i,j}$ denotes the (i, j)th element of the matrix A.

Let c, s_1 and s_2 be chosen as

$$c = \left[\frac{[X]_{1,i}[Y]_{1,i}}{[X]_{1,i}[Y]_{1,i} + [X]_{1,j}[Y]_{1,j}} \right]^{1/2}, \quad s_2 = -c\frac{[X]_{1,j}}{[X]_{1,i}}, \quad s_1 = -c\frac{[Y]_{1,j}}{[Y]_{1,i}},$$

and define X' and Y' by $X' \equiv XS_{(j|i)}$, $Y' \equiv YS_{(j|i)}^{-T}$. Then it is easy to check that $[X']_{1,j} = [Y']_{1,j} = 0$, and $X'Y'^T = XY^T$. We shall call the elements $[X]_{1,i}$ and $[Y]_{1,i}$ *pivoting elements*. Therefore, by repeating this process with an *appropriate* annihilation ordering and pivoting element, we can *annihilate* all elements in the first row of X and Y except the pivoting element. When the given generator is symmetric then $S_{(j|i)}$ will reduce to Givens rotations $G_{(j|i)}$ or hyperbolic rotations $H_{(j|i)}$, and only need to transform X.

The solutions of either square or rectangular Toeplitz systems of equations are the Schur complements in certain Toeplitz-block matrices. After obtaining generators of the Toeplitz-block matrices, we shall apply the algorithm to obtain generators of solutions. From the generators of solutions, we can read out the solutions.

Square Toeplitz System.

Let us first consider a square symmetric Toeplitz system of equations,

$$T\mathbf{x} = \mathbf{b}, \quad T = (t_{i-j}) \in \mathbf{R}^{n\times n}, \quad t_0 = 1, \quad \mathbf{b} \in \mathbf{R}^{n\times 1}. \tag{3}$$

We further assume that T is strongly nonsingular. If we define the following matrix M_1,

$$M_1 = \begin{bmatrix} T & -\mathbf{b} \\ I_{n\times n} & 0 \end{bmatrix} \in \mathbf{R}^{(2n)\times(n+1)}, \quad I_{n\times n} = n \times n \text{ identity matrix}, \tag{4}$$

then the solution $\mathbf{x} = T^{-1}\mathbf{b}$ is the Schur complement of T in M_1. To obtain a generator of the solution $T^{-1}\mathbf{b}$ using the generalized Schur algorithm, we first need to find the displacement of M_1 with respect to suitable displacement operators. For the matrix M_1, it is easy to see that the following choice of the displacement operators,

$$F^f = Z_n \oplus Z_n, \quad F^b = Z_n \oplus 0, \quad \text{(Note that } Z_1 = 0\text{)}$$

gives the smallest length $\alpha = 3$, therefore, the least number of flops.

With these displacement operators, a minimal generator $\{X, Y\}$ can be easily seen to be

$$X = \begin{bmatrix} 1 & t_1 & \cdot & t_{n-1} & 1 & 0 & \cdot & 0 \\ 0 & t_1 & \cdot & t_{n-1} & 1 & 0 & \cdot & 0 \\ -b_1 & -b_2 & \cdot & -b_n & 0 & 0 & \cdot & 0 \end{bmatrix}^T \in \mathbf{R}^{2n\times 3}, \quad Y = \begin{bmatrix} 1 & t_1 & \cdot & t_{n-1} & 0 \\ 0 & -t_1 & \cdot & -t_{n-1} & 0 \\ 0 & 0 & \cdot & 0 & 1 \end{bmatrix}^T \in \mathbf{R}^{(n+1)\times 3}. \tag{5}$$

Although the generator in (5) is not symmetric, we shall only need to transform X because the upper part of the 1st (2nd) column of Y will remain to be same (negative) to the upper part of the 1st (2nd) column of X, and the 3rd column of Y will not change.

Algorithm (Solving symmetric Toeplitz systems without back-substitution)
Input: X in (5).
Output: $\mathbf{x} = T^{-1}\mathbf{b}$
Procedure A
begin
for $k := 1$ **to** n **do begin**
if $[X]_{1,1}^2 > [X]_{1,2}^2$ **then** $pvt := 1$; NEXT := {2, 3};
else $pvt := 2$; NEXT := {1, 3};
for each $j \in$ NEXT, **begin**

Determine $S_{(j|pvt)}$ to annihilate $[X]_{1,j}$;
$X := XS_{(j|pvt)}$
end;
Shift down upper $n-k+1$ and lower n elements of $\mathbf{x}_{pvt}$ each by one positions;
Remove the annihilated null row from X;
end
return (the last column of X)
end

In the above algorithm, the spinors, $S_{(1|pvt)}$, $S_{(2|pvt)}$ will reduce to hyperbolic rotations $H_{(1|pvt)}$, $H_{(2|pvt)}$, and the spinor $S_{(3|pvt)}$ will reduces to *elimination matrix* $E_{(3|pvt)}$, i.e., the identity matrix except the element $[E_{(3|pvt)}]_{pvt,3}$.

For nonsymmetric strongly nonsingular Toeplitz matrices, we shall need to transform Y also.

Over-determined Toeplitz System.

Now let us consider the least squares problem,

$$T\mathbf{x} = \mathbf{b}, \qquad T = (t_{i-j}) \in \mathbf{R}^{m\times n}, \quad \mathbf{b} \in \mathbf{R}^{m\times 1}, \quad m > n, \tag{6}$$

where T is a Toeplitz matrix with a full column rank. If we define the matrix M_2 by

$$M_2 = \begin{bmatrix} -I_{m\times m} & T & -\mathbf{b} \\ T^T & O & 0 \\ O & I_{n\times n} & 0 \end{bmatrix} \in \mathbf{R}^{(m+2n)\times(m+n+1)}$$

then it is easy to check that the least squares solution to (6) is the Schur complement of $\begin{bmatrix} -I_{m\times m} & T \\ T^T & O \end{bmatrix}$ in M_2. With the following choice of displacement operators,

$$F^f = Z_m \oplus Z_n \oplus \mathbf{Z}_n, \quad F^b = Z_m \oplus Z_n \oplus 0,$$

we shall have rank-5 displacement, $\nabla_{(F^f,F^b)}M_2 \equiv XY^T$,

$$X = \begin{bmatrix} -1 & 0 & \cdot & 0 & t_0 & t_{-1} & \cdot & t_{-n+1} & 0^T \\ 0 & 0 & \cdot & 0 & t_0 & t_{-1} & \cdot & t_{-n+1} & 0^T \\ 0 & t_1 & \cdot & t_{m-1} & 0.5 & 0 & \cdot & 0 & \mathbf{e}_1^T \\ 0 & t_1 & \cdot & t_{m-1} & -0.5 & 0 & \cdot & 0 & \mathbf{e}_1^T \\ b_1 & b_2 & \cdot & b_m & 0 & 0 & \cdot & 0 & 0^T \end{bmatrix}^T, \quad Y = \begin{bmatrix} 1 & 0 & \cdot & 0 & -t_0 & -t_{-1} & \cdot & -t_{-n+1} & 0 \\ 0 & 0 & \cdot & 0 & t_0 & t_{-1} & \cdot & t_{-n+1} & 0 \\ 0 & t_1 & \cdot & t_{m-1} & 0.5 & 0 & \cdot & 0 & 0 \\ 0 & -t_1 & \cdot & -t_{m-1} & 0.5 & 0 & \cdot & 0 & 0 \\ 0 & 0 & \cdot & 0 & 0 & 0 & \cdot & 0 & 1 \end{bmatrix}^T, \tag{7}$$

where $\mathbf{e}_1^T = [\,1, 0, \cdots, 0\,] \in \mathbf{R}^{1\times n}$.

Algorithm (Solving Toeplitz least squares problem without back-substitution)
Input: X in (7).
Output: $\mathbf{x} = (T^TT)^{-1}T^T\mathbf{b}$
Procedure B
begin
for $k := 1$ **to** $m + n$ **do begin**
if $k \le m$ **then**
pvt := 1; FIRST = {4}; NEXT = {2,3,5};
else
pvt := 3; FIRST = {2}; NEXT = {1,4,5};
for each $j \in$ *FIRST* **and then for each** $j \in$ *NEXT*
Determine $S_{(j|pvt)}$ to annihilate $[X]_{1,j}$;
$X := XS_{(j|pvt)}$;

```
        end
        Shift down the three parts each by one position;
        Remove the annihilated null row from X;
    end
    return (the last column of X)
end
```

For the first m iterations, $S_{(4|1)}$, $S_{(5|1)}$ and $S_{(3|1)}$ will reduce to $G_{(4|1)}$, $E_{(5|1)}$ and $H_{(3|1)}$, respectively. Notice that $S_{(2|1)}$ is the identity matrix, and therefore, does not need to be applied. Also it turns out that $H_{(3|1)}$ and $G_{(4|1)}$ can be combined as a single matrix multiplication to further reduce the operation counts.

For the next n iterations, $S_{(2|3)}$, $S_{(1|3)}$, $S_{(4|3)}$ and $S_{(5|3)}$ will reduce to $G_{(2|3)}$, $H_{(1|3)}$, $H_{(4|3)}$ and $E_{(5|3)}$, respectively.

3. Systolic Arrays for solving Square Toeplitz Systems.

In this section we shall systematically derive systolic arrays for solving square Toeplitz systems of equations. Similar synthesis procedure can also be used to design systolic arrays solving over-determined Toeplitz systems of equations; however, we will not discuss the details in this paper. The first step in the design procedure is to obtain a Regular Iterative Algorithm (RIA) for the given algorithm. One can then utilize the formal procedures developed by Rao *et. al.* [10] to implement the given RIA on regular processor arrays. It has been shown that systolic algorithms (defined as algorithms implementable on systolic arrays) form a precisely defined subclass of RIAs. Moreover, the formal mapping techniques enable the designer to synthesize several systolic arrays by changing certain parameters in the design procedure. A detailed discussion of RIAs and their properties can be found in [10], [11] and we shall present a brief description of this class of structured algorithms. In general an RIA has the following features:

1. Each variable in the RIA is identified by a label and an *index vector* $I=[i, j, k]^T$. The range of the index vector is known as the *index space*.
2. It is in single assignment form i.e., every variable is assigned a unique value during the course of execution of the algorithm.
3. The main feature of the algorithm is the regularity of the dependences among the variables with respect to the index points. That is, if $x(I)$ is computed using the value of $y(I-d)$ then the *index displacement vector* d, corresponding to this direct dependence, is the same regardless of the index point I. As a consequence of this regularity, the dependence graph of an RIA has an iterative structure, which can be clearly demonstrated by drawing the dependence graph within the index space.

Notice that although the direct dependences among the variables in an RIA are required to be constant, the actual computations carried out to evaluate these variables can depend on the index point. In addition to that, functional relations among the variables can involve conditional branches, as will be seen in later sections. This scheme allows RIAs to have inhomogeneities in the dependence graph.

With this brief introduction a formal definition of an RIA can be presented.

Definition A Regular iterative algorithm is defined as follows:

1. Let $\mathbf{I}$ be an Index-Space which is the set of all lattice points enclosed within a specified region in a S-dimensional Euclidean space, $I \in \mathbf{I} \subset \mathbf{Z}^S$.
2. bold X is a set of V variables $x_i \in \mathbf{X}$ that are defined at every point in the index space, where the variable x_i defined at the index-point I will be denoted as $x_i(I)$ and takes on an unique value in any particular instance of the algorithm, and
3. $\mathbf{F}$ is a set of functional relations among the variables, restricted to be such that if $x_i(I)$ is computed using $x_j(I-d_{ji})$, then
   (a). d_{ji} is a constant vector independent of I and the extent of the index space, and
   (b). for every J contained in the index-space, $x_i(J)$ is directly dependent on $x_j(J-d_{ji})$ (if $(J-d_{ji})$ falls outside the index-space, then $x_j(J-d_{ji})$ is an external input to the algorithm).

Most algorithms are not presented to the designer in the nice regular form of an RIA. The key to the formulation of RIAs is to identify an index space such that all the variables can be expressed as indexed variables within the index space. The output of this step is often referred to as a Single Assignment Code (SAC). In a single assignment code every variable is distinct and takes on a unique value during the execution. The next step is to eliminate global dependencies in the dependence graph of the SAC. A formal procedure for doing so has recently been developed; however, for our purposes the global dependencies are quite straightforward to remove and one can do so without a formal framework.

Now, we shall show how the algorithm for solving square Toeplitz systems can be first converted into a SAC and then into an RIA. Let $x_1(i,j)$, $x_2(i,j)$ and $x_3(i,j)$ denote the elements of the first, second and third columns of the matrix X after the ith update. Now, at the $(i+1)$th step first thing is to determine the pivoting column. This can be done by defining a boolean variable $c(i,i)$ as follows:

$$c(i,i) = [x_1^2(i,i) \geq x_2^2(i,i)]$$

That is, $c(i,i)=1$ if column 1 is chosen as the pivoting column and $c(i,i)=0$ if column 2 is chosen as the pivotal column. In this paper, $\overline{c(i,i)}$ is defined as the binary complement of the boolean variable $c(i,i)$ and we shall consider multiplication of any variable by a boolean variable as multiplication by 0 or 1. Next, let us define $p_1(i,j)$, $p_2(i,j)$, $p_3(i,j)$ as the updated versions of the pivoting column during the $(i+1)$st stage. Hence, to begin with

$$p_1(i,j) = x_1(i,j) \times c(i,i) + x_2(i,j) \times \overline{c(i,i)}.$$

That is, the pivotal column is set to column 1 if $c(i,i)=1$ (i.e., if the column 1 is chosen as the pivotal column) else it is set to column 2. Now, either $p_1(i,i)$ is used to annihilate $x_2(i,i)$ (when $c(i,i)=1$) or $p_1(i,i)$ is used to annihilate $x_1(i,i)$ (when $c(i,i)=0$). A common parameter $m_1(i,i)$ can be defined as follows:

$$m_1(i,i) = \frac{x_1(i,i) \times \overline{c(i,i)} + x_2(i,i) \times c(i,i)}{p_1(i,i)}$$

Thus, $m_1(i,i)=x_2(i,i)/x_1(i,i)$ if $c(i,i)=1$ else $m_1(i,i)=x_2(i,i)/p_1(i,i)$. This parameter is equivalent to the rotation parameters c, s_1 or s_2 in the previous section. Hence, the updating of the pivoting column and the first or the second column can be obtained as follows:

$$p_2(i,j) = \frac{1}{(1\pm m_1^2(i,i))^{1/2}}[p_1(i,j) + m_1(i,i)(x_1(i,j)\times\overline{c(i,i)} + x_2(i,j)c(i,i))]$$

$$y_1(i,j) = \overline{c(i,i)} \times \frac{1}{(1\pm m_1^2(i,i))^{1/2}}[\pm m_1(i,i)p_1(i,j) + x_1(i,j)]$$

$$y_2(i,j) = c(i,i) \times \frac{1}{(1\pm m_1^2(i,i))^{1/2}}[\pm m_1(i,i)p_1(i,j) + x_2(i,j)]$$

where "±" can be chosen appropriately to define either an orthogonal or a hyperbolic rotation. One can easily verify that if $c(i,i)=1$ then $p_1(i,i)$ is being used to annihilate $x_2(i,i)$ and $y(i,j)$ is set to 0. Similarly, if $c(i,i)=0$ then $p_2(i,j)$ represents the updated pivoting column and $y_1(i,j)$ represent the updated first column. Next, we have to use $p_2(i,i)$ to annihilate $x_3(i,i)$. The elements of the updated pivoting column and the third column can be given by

$$p_3(i,j) = \frac{1}{(1\pm m_2^2(i,i))^{1/2}}[p_2(i,j)\pm m_2(i,i)x_3(i,j)]$$

$$x_3(i+1+j) = \frac{1}{(1\pm m_2^2(i,i))^{1/2}}[\pm m_2(i,i)p_2(i,j) + x_3(i,j)]$$

where $m_2(i,i)=x_3(i,i)/p_2(i,i)$, and operations ± are chosen appropriately to define the intended rotation. Note that if column 1 is chosen as the pivoting column then the elements $p_3(i,j)$ have to be shifted down by one place (in two parts) and made the new first column and the updated second column would be the variables $y(i,j)$. The situation is just reversed if column 2 is chosen as the pivoting column instead of column 1. These assignment procedure can be coded in terms of the variables defined so far as follows:

$$x_1(i+1,j) = \begin{cases} c(i,i) \times p_3(i,j-1) + \overline{c(i,i)}\, y_1(i,j) & \text{if } j \neq n+1 \\ \overline{c(i,i)}\, y_1(i,j) & \text{if } j = n+1 \end{cases}$$

$$x_2(i+1,j) = \begin{cases} \overline{c(i,i)} \times p_3(i,j-1) + c(i,i)\, y_2(i,j) & \text{if } j \neq n+1 \\ c(i,i)\, y_2(i,j) & \text{if } j = n+1 \end{cases}$$

The special case of $j=n+1$ is necessary to ensure that the two parts (the top part is from $j=1$ to $j=n$ and the bottom part is from $j=n+1$ to $j=2n$) of the pivoting column are shifted down separately.

Overwriting of variables has been avoided and all the statements are in the single assignment form. However, the statements are not in the RIA form; for example in the definition of $p_2(i,j)$, we can observe that $p_2(i,j)$ is dependent on $m_1(i,i)$ and $c(i,i)$. The displacement vector is a function of the index point. However, such statements can be easily localized by defining propagating variables that propagate the values of $m_1(i,i)$ and $c(i,i)$ to the index points (i,j). One way of defining the propagating variables is

$$c(i,j) = \begin{cases} (x_1^2(i,j) \geq x_2^2(i,j)) & \text{if } j = i \\ c(i,j-1) & \text{if } j > i \end{cases}$$

$$m_1(i,j) = \begin{cases} \dfrac{x_1(i,j) \times \overline{c(i,j)} + x_2(i,j) \times c(i,j)}{p_1(i,j)} & \text{if } j = i \\ m_1(i,j-1) & \text{if } j > i \end{cases}$$

The above statements ensure that $c(i,i)$ and $m_1(i,i)$ are defined as before and are propagated to index points (i,j). Similar definition can be given for $m_2(i,i)$ and then all the statements can be written in the RIA form. The iteration unit of the resultant RIA can be represented as:

for all tuples (i,j) where $1 \leq i \leq n$ and $i \leq j \leq 2n$ **do**

$$c(i,j) = \begin{cases} (x_1^2(i,j) \geq x_2^2(i,j)) & \text{if } j = i \\ c(i,j-1) & \text{if } j > i \end{cases}$$

$$p_1(i,j) = x_1(i,j) \times c(i,j) + x_2(i,j) \times \overline{c(i,j)}$$

$$m_1(i,j) = \begin{cases} \dfrac{x_1(i,j) \times \overline{c(i,j)} + x_2(i,j) \times c(i,j)}{p_1(i,j)} & \text{if } j = i \\ m_1(i,j-1) & \text{if } j > i \end{cases}$$

$$p_2(i,j) = \frac{1}{(1 \pm m_1^2(i,j))^{1/2}} [p_1(i,j) + m_1(i,j)(x_1(i,j) \times \overline{c(i,j)} + x_2(i,j) c(i,j)]$$

$$y_1 = \overline{c(i,j)} \times \frac{1}{(1 \pm m_1^2(i,j))^{1/2}} [-m_1(i,j) p_1(i,j) + x_1(i,j)]$$

$$y_2 = c(i,j) \times \frac{1}{1 \pm m_1^2(i,j))^{1/2}} [-m_1(i,j) p_1(i,j) + x_2(i,j)]$$

$$m_2(i,j) = \begin{cases} x_3(i,j)/p_2(i,j) & \text{if } j = i \\ m_2(i,j-1) & \text{if } j > i \end{cases}$$

$$p_3(i,j) = \frac{1}{(1 \pm m_2^2(i,j))^{1/2}} [p_2(i,j) + m_2(i,j) x_3(i,j)]$$

$$x_3(i+1,j) = \frac{1}{(1 \pm m_2^2(i,j))^{1/2}} [-m_2(i,j) p_2(i,j) + x_3(i,j)]$$

$$x_1(i+1,j) = \begin{cases} c(i,j) \times p_3(i,j-1) + \overline{c(i,j)} y_1(i,j) & \text{if } j \neq n+1 \\ \overline{c(i,j)} y_1(i,j) & \text{if } j = n+1 \end{cases}$$

$$x_2(i+1,j) = \begin{cases} \overline{c(i,j)} \times p_3(i,j-1) + c(i,j) y_2(i,j) & \text{if } j \neq n+1 \\ c(i,j) y_2(i,j) & \text{if } j = n+1 \end{cases}$$

end.

Though the above iteration unit seems complex, it has several desirable features that it shares with RIAs. First, all the computations have been explicitly enumerated without overwriting and secondly the dependence graph of the algorithm can be embedded in a regular fashion in an index space. Fig. 1 shows the coarse grained dependence graph whereas Fig. 2 and Fig. 3 show the fine grained dependence structure of the two different types of nodes in the dependence graph. The nodes along the diagonal marked by '*' are the ones that determine the pivoting column and the parameters for rotation at every step of the algorithm. The rest of the processors receive these parameters and carry out the operations; hence the *-nodes do more complex operations than the other nodes.

Systolic Arrays.

One can show that the I/O latency of the above algorithm is $O(n)$ and hence from the theory of RIA one is assured that the algorithm is a systolic algorithm. Various systolic arrays can be obtained by projecting the dependence graph of the algorithm in different directions. Thus, one needs to choose a projection direction (also referred to as the iteration vector) and all computations corresponding to index index points lying along this direction are executed by the same processor. The sequencing of the operations assigned to each processor is quite straight forward for our case and will not be discussed any more in this paper.

Fig. 4 shows two such arrays obtained by projecting the dependence graph along directions $[1, 0]^T$ and $[1, 1]^T$ respectively. In the first array, the *-nodes are mapped to different arrays. Hence, some processors have to perform the operations of the *-nodes as well as those of the other nodes at different times. This is avoided in the second array, where all the *-nodes have been mapped to the same processor. The direction of data traversal is also different in the two arrays; however, both the arrays will have the same number of processors (i.e., $2n$). The number of processors can be reduced by a factor of 2 by projecting the dependence graph along the direction [0, 1], in which case it will have only n processors.

REFERENCES

[1]. A. Bojanczyk, R. Brent and F. de Hoog, *Linearly connected arrays for Toeplitz least squares problems*, Technical report, Australian National University, (1985).

[2]. R. Brent and F. Luk, *A systolic array for the linear-time solution of Toeplitz systems of equations* Journal of VLSI and Comp. Sys., vol. 1, No. 1. 1, (1983) pp. 1-22.

[3]. J. Chun and T. Kailath *Block-Toeplitz and Toeplitz-block linear equations*, Stanford univ., Preprint, (1988)

[4]. J. Delosme and I. Ipsen, *Efficient parallel solution of linear systems with hyperbolic rotations*, Tech. Report No. 8501, Mar. (1985)

[5]. Ed.F. Deprettere and K. Jainandunsing, *On the design and the partitioning of dedicated arrays for solving sets of linear equations without back-substitution* Tech. Report, Department of Electrical Engineering, Delft University of Technology, (1986)

[6]. T. Kailath, *Signal processing in the VLSI era*, VLSI and Modern Signal Processing, S. Kung, H. Whitehouse and T. Kailath eds., Prentice-Hall, Englewood Cliffs, (1985).

[7]. T. Kailath, S. Kung and M. Morf, *Displacement ranks of matrices and linear equations*, J. Math. Anal. App'., 68 (1979) pp. 395-407. See also Bull. Amer. Math. Soc., 1 (1979), pp. 769-773.

[8]. S. Kung and Y. Hu, *A highly concurrent algorithm and pipelined architecture for solving Toeplitz systems*, IEEE Trans., ASSP, vol. 31, No. 1, Feb., (1983).

[9] J. Nash and S. Hansen, *Modified Faddeeva algorithm for concurrent execution of linear algebraic operations* IEEE Trans. Comput., vol. C-37, No. 2, (1988), pp. 129-137.

[10]. S. Rao and T. Kailath, *Regular iterative algorithms and their implementations on processor arrays*, Proc. IEEE, vol. 76, No. 3, (1988) pp. 259-282.

[11]. V. Roychowdhury, L. Thiele, S. Rao and T. Kailath, *On the localization of algorithms for VLSI processor arrays*, submitted to IEEE Trans. on Computers (1988).

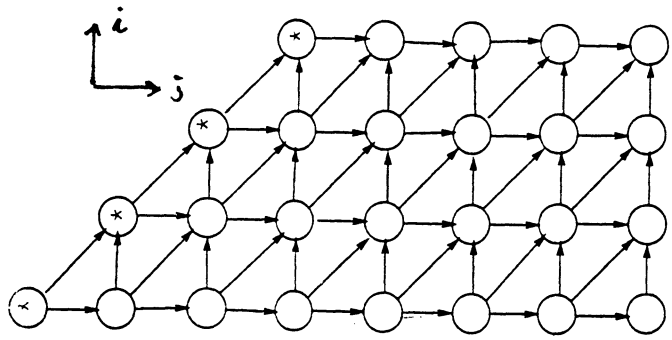

Fig 1. A coarse grained dependence graph.

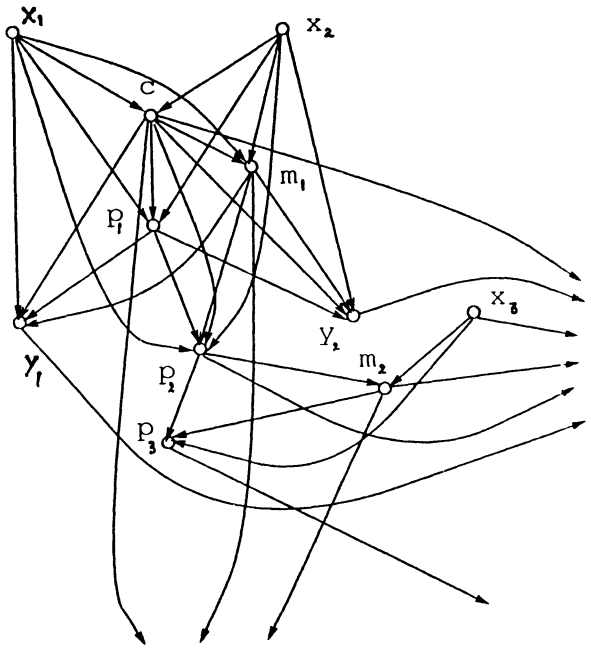

Fig 2. Fine grained structure of the * nodes.

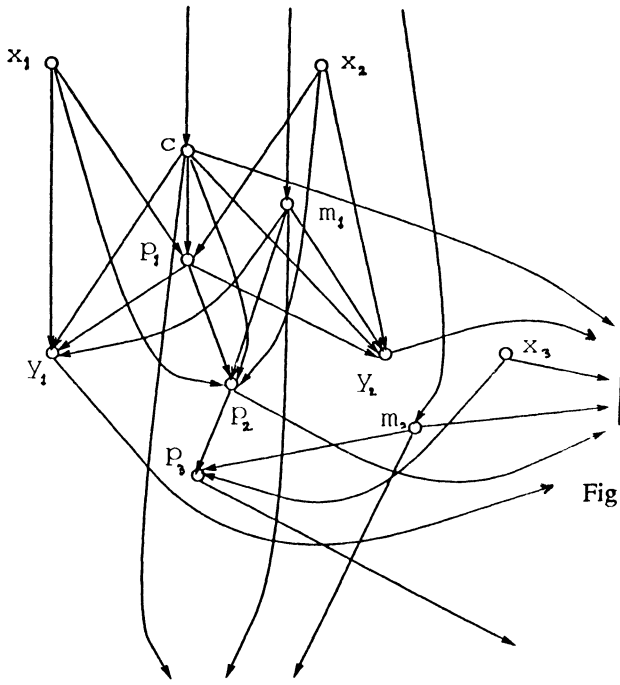

Fig 3. Fine grained structure of other nodes.

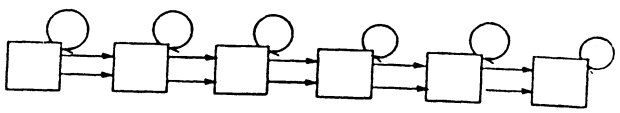

Fig 4.1. Processor array obtained by projecting along [1, 0]

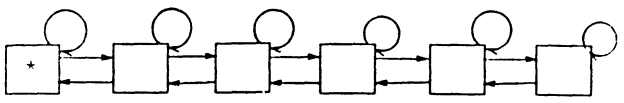

Fig 4.2. Processor array obtained by projecting along [1, 1]