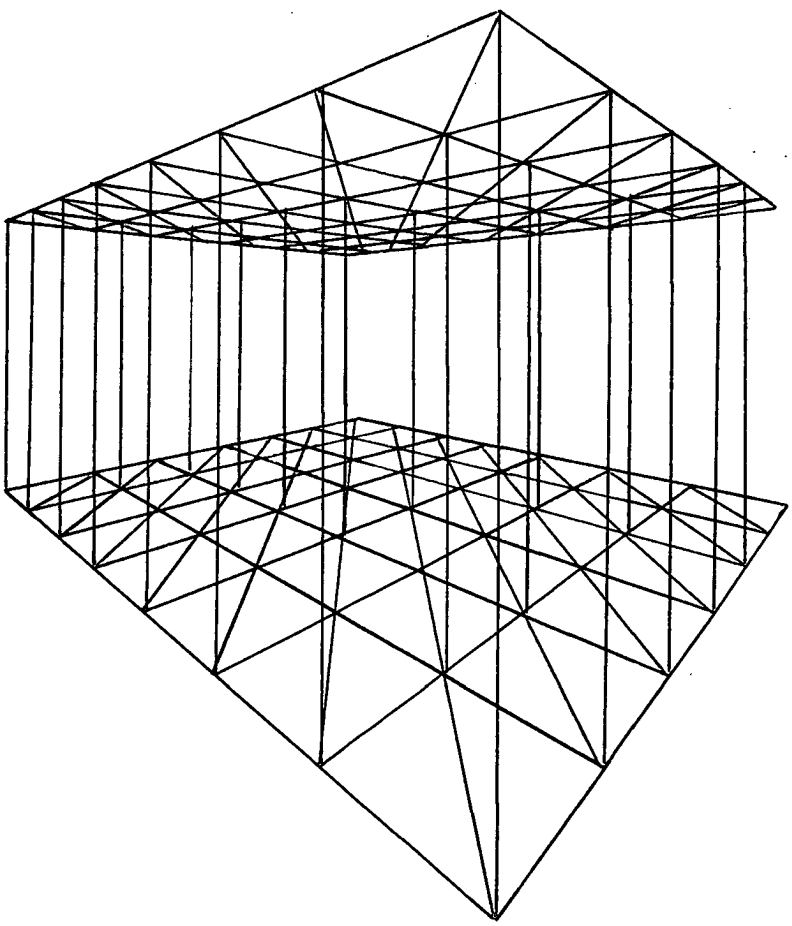
PARALLEL ALGORITHMS FOR SOLVING SYSTEMS OF LINEAR EQUATIONS AND THEIR MAPPING ON SYSTOLIC ARRAYS



K. JAINANDUNSING

PARALLEL ALGORITHMS FOR SOLVING SYSTEMS OF LINEAR EQUATIONS AND THEIR MAPPING ON SYSTOLIC ARRAYS

PARALLEL ALGORITHMS FOR SOLVING SYSTEMS OF LINEAR EQUATIONS AND THEIR MAPPING ON SYSTOLIC ARRAYS

Proefschrift

ter verkrijging van de graad van doctor aan de Technische Universiteit Delft, op gezag van de Rector Magnificus, prof.drs. P.A. Schenck, in het openbaar te verdedigen ten overstaan van een commissie aangewezen door het College van Dekanen op dinsdag 3 januari 1989 te 14.00 uur

door

K. JAINANDUNSING

geboren te Paramaribo
electrotechnisch ingenieur

Dit proefschrift is goedgekeurd door de promotor
prof.dr.ir. P. Dewilde

- To my parents -

CONTENTS

PREFACE

The research presented in this thesis has three points of focus. The first is the development of algorithms for solving systems of linear equations with a maximum degree of parallelism and pipelining. The second is the development of partitioning strategies for the design of systolic arrays with the number of processor elements independent of the size of the problem. And the last is the synthesis of a concrete systolic array, with a fixed number of processor elements, for the algorithms developed.

The thesis has the following outline. Chapter 1 is introductory. It is divided in two main sections. Section 1.1 starts by summarizing the conventional direct methods for solving systems of linear equations. It is shown that a systolic implementation of feed forward direct methods such as Faddeev's execute almost twice as fast than a systolic implementation of direct methods with backsubstitution. The backsubstitution is identified as the bottleneck in such systolic implementations. Unfortunately, Faddeev's feed forward direct method needs pivoting to be numerically stable, while pivoting is hard to implement on a systolic array. The problems discussed in this section form the motivation for the development of the feed forward direct methods of Chapter 2. Section 1.2 starts by summarizing the basic concepts of mapping a regular recurrent algorithm on a full size systolic array. That is, an array with the number of processor elements proportional to the size of the problem. Next, it continues with the introduction of partitioning strategies for full size systolic arrays. Application of these strategies results in reduced size systolic arrays. That is, systolic arrays with a fixed number of processor elements, independent of the size of the problem.

Chapter 2 presents a class of feed forward direct methods for solving non singular systems of linear equations. These methods obtain the solution $\mathbf{x}$ of $A\mathbf{x} = \mathbf{b}$, $A \in \mathbf{R}^{N \times N}$, $\mathbf{b} \in \mathbf{R}^{N \times 1}$, through combination of an LU, LQ or LL^t factorization of the matrix A and

an updating or downdating of the Cholesky factorization of the matrix $LL^t + \mathbf{b}\mathbf{b}^t$ or $LL^t - \mathbf{b}\mathbf{b}^t$, respectively, or an LU factorization of the matrix $[L \ -\mathbf{b}]^t$. The matrix L is the lower triangular factor in either of the three factorizations of the matrix A. Section 2.1 explains how the solution $\mathbf{x}$ is computed from the up- or downdating of the Cholesky factorization or from the LU factorization of the matrix $[L \ -\mathbf{b}]^t$. Section 2.2 shows how an LU, QR or LL^t factorization of the coefficient matrix is combined with the methods in Section 2.1, so that a class of feed forward direct methods is obtained. These methods compute the solution of a non singular system of linear equations by a single factorization. Thus the backsubstitution bottleneck is avoided in systolic implementations. The class contains a feed forward direct method which uses only orthogonal transformations as elementary operations. Hence, the method is stable for the general class of non singular systems of linear equations, unlike Faddeev's feed forward direct method (without (partial) pivoting). Section 2.3 presents a numerical analysis of the feed forward methods of Section 2.2. Section 2.4 shows that the method which combines an LL^t factorization of the coefficient matrix with a downdating of a Cholesky factorization is in fact a semi-direct method. Section 2.5 explains how the methods of Section 2.2 can be generalized to perform computations of the kind $CA^{-1}B + D$, where the matrices A, B, C and D are of proper dimensions and value. This generalization is similar to the extension of Faddeev's feed forward direct method.

Chapter 3 presents the LPGP and LSGP partitioning strategies to partition a full size systolic array into a reduced size systolic array. In an LPGP (local-parallel-global-pipelined) partitioning of a full size array the array is tessellated into congruent tiles of, say, p processor elements each. The computations of different tiles are executed in pipeline on the reduced size array, which has p processor elements and the same interconnection topology as the full size array. In an LSGP (local-sequential-global-parallel) partitioning of a full size systolic array the array is again tessellated into congruent tiles of p processor elements each. But, in this case the processor elements in a tile are replaced by a single processor element in the reduced size array. This processor element executes the tasks of the p processor elements in sequence. In general the reduced size systolic array has a different interconnection topology than the full size array. Section 3.1 presents the

tessellation of a full size systolic array. Section 3.2 presents the LPGP partitioning strategy and Section 3.3 presents the LSGP partitioning strategy.

Chapter 4 presents the design of a reduced size systolic array for two of the feed forward direct methods of Section 2.2. The purpose of this chapter is to illustrate the design of a practical reduced size systolic array for these two methods. Section 4.1 introduces the design constraints. Section 4.2 summarizes the two methods in terms of orthogonal and hyperbolic Householder transformations. Section 4.3 illustrates the design of full size systolic arrays for these methods. Section 4.4 illustrates the LPGP partitioning of the full size systolic arrays of Section 4.3 and their unification into a single reduced size array. Section 4.5 presents the architecture of a Householder processor element of the reduced size array. The architecture is that of an innerproduct-step processor, which serializes the computation of a Householder transformation of a vector.

Finally, Chapter 5 contains the conclusions of the thesis :

1. systolic implementations of the feed forward direct methods execute roughly twice as fast as systolic implementations of the direct methods with factorization and backsubstitution;

2. the class of feed forward direct methods contains a method which is numerically robust and stable without (partial) pivoting, for the complete set of non singular systems of linear equations;

3. it is possible to design systolic arrays for which the I/O bandwidth is closely matched to that of devices attached to the inputs and outputs of the systolic array or for which the number of processor elements is independent of the size of the problem that has to be executed on the array or both;

4. systolic arrays can be designed which execute more than one algorithm, with a minimum of control overhead.

I am greatly indebted to Dr.Ir. E.F. Deprettere for his professional advise and guidance in solving the problems I encountered during my Ph.D. research. Dr.Ir. E.F. Deprettere was my main source of stimuli in understanding these problems. I would also like to thank him for his sincere friendship in the past 4 years and his patience, while reading the preliminary drafts of the thesis.

Delft, K. Jainandunsing
October 1988

SUMMARY

SYSTEMS OF LINEAR EQUATIONS

Conventional direct methods for solving systems of linear equations consist of either a factorization of the coefficient matrix followed by a backsubstitution [Golu1983], or of a factorization of an augmented matrix [Fadd1959]. None of the two classes provide good solutions for implementations on systolic arrays. Methods of the first class are sequential in nature. I.e., the factorization of the coefficient matrix has to be completed before the backsubstitution can be done. This obstructs the acceleration of such direct methods when implemented on a systolic array.

A method of the second class is due to Faddeev [Fadd1959]. It computes the solution from a single LU factorization of an augmented matrix. This method, and feed forward direct methods in general, do not require a backsubstitution step. Therefore, highly parallel implementations on systolic arrays are possible. However, the LU factorization without (partial) pivoting is in general not numerically stable. Unfortunately, (partial) pivoting is hard to implement on an array where the length of interconnections between processors is independent of the size of the array, as is the case for systolic arrays. In this thesis feed forward direct methods are studied which rephrase the backsubstitution in terms of an updating or downdating of a Cholesky factorization, or in terms of an LU factorization. Combining this with an LU, LQ or LL^t factorization of the coefficient matrix A yields a whole class of feed forward direct methods, which do not suffer from the backsubstitution bottleneck. One of these algorithms uses only orthogonal transformations as elementary operations and is therefore numerically stable for the general class of non singular systems of linear equations. The feed forward direct methods that are presented here have simple and highly parallel systolic implementations.

SYSTOLIC IMPLEMENTATION

A systolic array is defined here (see also [Rao1985] or [Rao1988]) to be an array of synchronously operating processor elements which are interconnected in a regular mesh with interconnections of length independent of the size of the array. Their regular topology is a desirable feature for area-efficient VLSI implementations and the absence of interconnections of arbitrary length eliminates communication delays which depend on the size of the array. Due to the massive parallelism of systolic arrays, systolic implementations of computationally intensive algorithms are given much attention. Most matrix-based signal processing algorithms, including the ones presented in this thesis, can be expressed as regular recurrent algorithms (see for instance [Unge1958], [McCl1959], [Ahme1982], [Depr1982], [Fort1985], [Delo1986], [Jain1986a]). Such algorithms can be automatically mapped to full size systolic arrays ([Mold1983], [Rao1985], [Fris1986], [Rao1988]). However, since the number of processor elements in a full size systolic array increases whenever the size of the problem grows, it is important to know how to partition large problems so that they can be executed on a systolic array with a small number of processor elements [Hell1985], [Mold1986], [Hori1987], [Neli1988].

Two strategies are developed in this thesis to partition a full size systolic array into a reduced size systolic array. The LPGP (local-parallel-global-pipelined) partitioning strategy tessellates the full size systolic array into congruent tiles of, say, p processor elements each. The computations of different tiles are executed in pipeline on a reduced size systolic array of p processor elements which has the same interconnection topology as the full size systolic array. The LSGP (local-sequential-global-parallel) partitioning strategy also tessellates the full size array into congruent tiles of, say, p processor elements each. The processor elements in a tile are replaced by a single processor element which executes the tasks of these processor elements in sequence. In this case the topology of the reduced size array is different from that of the full size systolic array. The LPGP partitioning keeps all memory for intermediate results outside the reduced size array. The LSGP partitioning causes local memory to increase, and reduces the I/O

bandwidth of the processor elements. Combining the two strategies results in array designs in which the number of processor elements is independent of the size of the problem and which have an I/O communication bandwidth which matches that of attached peripheral devices (such as disks, a host computer, etc.) as close as possible.

SYSTOLIC ARRAY DESIGN

The partitioning theory is applied to the design of a reduced size systolic array for two feed forward methods; 1) the method consisting of a combination of an LQ factorization of the coefficient matrix A and an updating of a Cholesky factorization and 2) the method consisting of an LL^t factorization of the coefficient matrix A and a downdating of a Cholesky factorization. This application illustrates the practical use of the partitioning theory for designing reduced size systolic arrays. The two methods are stated in terms of orthogonal and hyperbolic Householder transformations, respectively. A processor element of the reduced size array is implemented as an innerproduct-step processor, which serializes the computation of a Householder transformation of a vector.

1. INTRODUCTION

1.1 SOLVING SYSTEMS OF LINEAR EQUATIONS : DIRECT METHODS AND SYSTOLIC IMPLEMENTATIONS

Non singular systems of linear equations $A\,\mathbf{x} = \mathbf{b}$ can be solved by iterative, direct or semi-direct methods. The iterative methods, like the Jacobi and the Gauss-Seidel method [Golu1983], improve iteratively on an initial guess $\mathbf{x}_0$ of the solution vector. The direct methods, on the other hand, use matrix factorization (and backsubstitution) to obtain the solution directly. Semi-direct methods, like the conjugate gradient method [Golu1983], obtain the solution by computing the minimum of the quadratic functional $f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^t A\,\mathbf{x} - \mathbf{x}^t\mathbf{b}$, which is reached for $\mathbf{x} = A^{-1}\mathbf{b}$. Theoretically the conjugate gradient method computes the minimum in a finite number of steps. However, the method is iterative in practice, due to roundoff errors and an error criterion is needed to halt the solution process.

The iterative and semi-direct methods do not change the coefficient matrix during the iterations. If the matrix is sparse, no fill-ins are created and the data can be efficiently stored in the (main) memory of the computer, even if the matrix is very large. Therefore, these methods play an important role in the solution of very large and sparse systems of linear equations.

The direct methods do create fill-ins in a sparse coefficient matrix, since factors of a sparse matrix are generally not sparse. These methods are more suited for solving dense systems of linear equations and no error criterion is needed to terminate the solution process.

1.1.1 DIRECT METHODS

The direct methods for solving non-singular systems of linear equations, $A\,\mathbf{x} = \mathbf{b}$ ($A \in \mathbf{R}^{N\times N}$, $\mathbf{b} \in \mathbf{R}^{N}$), can be divided in two classes :

1. direct methods with backsubstitution;
2. feed forward direct methods.

We shall give a quick presentation of these two classes.

DIRECT METHODS WITH BACKSUBSTITUTION

These methods compute either an LU, QR or LL^t (Cholesky) factorization of the matrix A, followed by a backsubstitution step. Let us denote by X^t either the lower triangular factor in an LU factorization, the orthogonal factor Q, in a QR factorization, or the lower triangular factor L, in an LL^t factorization and let us denote by R_X the corresponding upper triangular factor for each of these cases. Then, we can summarize the direct methods with backsubstitution as follows :

$$A = X^t R_X, \tag{1.1.a}$$

$$\mathbf{y} = X^{-t}\mathbf{b}, \tag{1.1.b}$$

$$\mathbf{x} = R_X^{-1}\mathbf{y}. \tag{1.1.c}$$

Computation of the factorizations is done by using either embeddings of elementary 2×2 transformations or Householder transformations [Golu1983]. The discussion of factorizations with Householder transformations is deferred to Chapter 4. In case of an LU factorization the 2×2 transformations are of the form :

$$\begin{bmatrix} 1 & 0 \\ \alpha & 1 \end{bmatrix}, \ \alpha \in \mathbf{R}.$$

In case of a QR factorization they are of the form :

$$\begin{bmatrix} cos(\alpha) & -sin(\alpha) \\ sin(\alpha) & cos(\alpha) \end{bmatrix}, \ \alpha \in \mathbf{R}.$$

And, in case of an LL^t factorization they are of the form :

$$\begin{bmatrix} cosh(\alpha) & sinh(\alpha) \\ sinh(\alpha) & cosh(\alpha) \end{bmatrix}, \ \alpha \in \mathbf{R}.$$

Remark : The hyperbolic transformations are used in Schur-Cholesky LL^t factorization algorithms (see Section 2.2 and [Dewi1981], [Depr1982] and [Delo1984]). These algorithms are less known than the Cholesky factorization algorithm as found in [Golu1983]. For the Schur-Cholesky LL^t factorization algorithms it has been illustrated that they can be mapped onto highly parallel systolic arrays of processor elements (see for instance [Ahme1982] and [Delo1986]).

The three types of elementary transformations can be represented by the following parameterized form [Walt1971] :

$$\Theta(m;\alpha) = \begin{bmatrix} cos(m^{\frac{1}{2}}\alpha) & -m^{\frac{1}{2}}sin(m^{\frac{1}{2}}\alpha) \\ m^{-\frac{1}{2}}sin(m^{\frac{1}{2}}\alpha) & cos(m^{\frac{1}{2}}\alpha) \end{bmatrix}, \ m \in \{0,1,-1\}. \tag{1.2}$$

Case $m = 0$, $m = 1$ and $m = -1$ are known as the so-called linear (Gauss), orthogonal (Givens) and hyperbolic (Minkowsky) rotation, respectively. The matrix $\Theta(m;\alpha)$ is orthogonal with respect to the following signature matrix :

$$S(m) = \begin{bmatrix} 1 & 0 \\ 0 & m \end{bmatrix}. \tag{1.3}$$

That is,

$$\Theta^t(m;\alpha)S(m)\Theta(m;\alpha) = S(m) \tag{1.4.a}$$

and for $m = \pm 1$ we also have :

$$\Theta(m;\alpha)S(m)\Theta^t(m;\alpha) = S(m). \tag{1.4.b}$$

Property (1.4.a) is referred to as $S(m)$–*orthogonality* of $\Theta(m;\alpha)$, or just orthogonality in case $m = 1$.

A *QR* or *LU* factorization is obtained by premultiplication of the matrix A by embeddings of the 2×2 orthogonal or linear rotations (m=1 or 0, respectively) of the following form :

$$\Theta_{ij}(m) = \begin{array}{c} \\ \\ \\ i \\ \\ \\ j \\ \\ \\ \end{array} \begin{bmatrix} 1 & & & & & & & \\ & \ddots & & & & & & \\ & & 1 & & & & & \\ & & & \cos(m^{1/2}\alpha_{ij}) & & -m^{1/2}\sin(m^{1/2}\alpha_{ij}) & & \\ & & & & 1 & & & \\ & & & & & \ddots & & \\ & & & m^{-1/2}\sin(m^{1/2}\alpha_{ij}) & & \cos(m^{1/2}\alpha_{ij}) & & \\ & & & & & & 1 & \\ & & & & & & & \ddots \\ & & & & & & & & 1 \end{bmatrix} \begin{array}{c} \\ \\ \\ i \\ \\ \\ j \\ \\ \\ \end{array}$$

$$\begin{array}{ccccc} & & i & & j \end{array}$$

(1.5)

For the appropriate choice of α_{ij}, premultiplication by $\Theta_{ij}(m)$ eliminates the entry a_{ij} of A. The product $\overleftarrow{\prod_{i,j}}\Theta_{ij}(0)$, where the pairs (i,j) are such that all elements in the strictly lower triangular part of the matrix A are eliminated, is a lower triangular matrix which is the inverse of the lower triangular factor L in the *LU* factorization of the matrix A. The product $\overleftarrow{\prod_{i,j}}\Theta_{ij}(1)$, where the pairs (i,j) are the same as in the case of the *LU*

factorization, is an orthogonal matrix which is the inverse of the orthogonal factor Q in the QR factorization of the matrix A. The LU factorization of indefinite matrices is numerically unstable without (partial) pivoting [Golu1983]. A small entry on the diagonal of the matrix A will blow up certain coefficients α_{ij} in (1.5). This is impossible for the QR factorization, since the absolute value of the sine and cosine function is bounded by 1.

The use of hyperbolic rotations is restricted to the case where an LL^t factorization of the matrix A is computed. That is, in case A is symmetric positive definite. We defer the discussion of computing Cholesky factors by means of hyperbolic rotations to Chapter 2.

FEED FORWARD DIRECT METHODS

A not too well known feed forward direct method is due to Faddeev [Fadd1959]. This method avoids an explicit backsubstitution step and obtains the solution from a single LU factorization of an augmented matrix. In fact, it was this method that motivated the search for the algorithms that will be considered in Chapter 2. Faddeev's method is stated in the following proposition.

Proposition 1.1 :

Let $A \in \mathbb{R}^{N\times N}$ *be a non singular matrix and* $\mathbf{b} \in \mathbb{R}^N$. *Let* $\mathbb{L} \in \mathbb{R}^{2N\times 2N}$ *be the inverse of the lower triangular factor of the LU factorization of the augmented matrix* $[A^t \mid -I_N]^t$ *and let* $[R^t \mid O]^t$ *be the corresponding upper triangular factor,* $R \in \mathbb{R}^{N\times N}$. *Then, the solution* $\mathbf{x}$ *of the system of linear equations* $A\mathbf{x} = \mathbf{b}$ *satisfies the following equation :*

$$\mathbb{L}\begin{bmatrix} A & \mathbf{b} \\ -I_N & 0 \end{bmatrix} = \begin{bmatrix} R & * \\ 0 & \mathbf{x} \end{bmatrix}. \tag{1.6.a}$$

Proof :

Partition the $2N \times 2N$ matrix $\mathbb{L}$ as follows :

$$\mathbb{L} = \begin{bmatrix} L_{11} & L_{12} \\ L_{21} & L_{22} \end{bmatrix}, \text{ with } L_{11}, L_{12}, L_{21}, L_{22} \in \mathbb{R}^{N \times N}. \tag{1.6.b}$$

Substituting (1.6.b) in (1.6.a) gives :

$$L_{21} = L_{22} A^{-1}. \tag{1.7}$$

Thus,

$$L_{21}\mathbf{b} = L_{22} A^{-1}\mathbf{b} = L_{22}\mathbf{x}. \tag{1.8}$$

And by observing that $L_{22} = I_N$, it follows that :

$$L_{21}\mathbf{b} = \mathbf{x} \tag{1.9}$$

and (1.6.a) follows.

□

Faddeev's method is not only capable of solving $A\mathbf{x} = \mathbf{b}$, but it can also compute the Schur complement of a matrix. This is easily understood by applying Faddeev's method to a block matrix :

$$\mathbb{L} \begin{bmatrix} A & B \\ C & D \end{bmatrix} = \begin{bmatrix} R & * \\ O & E \end{bmatrix}, \tag{1.10}$$

with appropriate $\mathbb{L}$. Using (1.6.b) in this equation gives :

$$E = (D - CA^{-1}B), \tag{1.11}$$

which is the Schur complement of the augmented matrix $\begin{bmatrix} A & B \\ C & D \end{bmatrix}$. Choosing the dimensions and values of the matrices A, B, C and D appropriately we can :

1. solve for multiple right hand sides, i.e., $AX = B$, where $B \in \mathbf{R}^{N \times M}$;

2. compute innerproduct accumulations, i.e., $d - \mathbf{c}^t\mathbf{b}$;

3. compute matrix-vector product accumulations, i.e., $\mathbf{d} - \mathbf{c}^t B$;

4. compute matrix-matrix product accumulations, i.e., $D - CB$.

The LU factorization in Faddeev's method is numerically stable only when the matrix A is positive definite. In case the matrix A is indefinite, pivoting strategies must be used to guarantee numerical stability. In [Nash1988] it is shown how the numerical stability of Faddeev's method can be improved, by using Givens rotations to compute the upper triangularization of the matrix A, after which linear rotations are applied to eliminate the matrix C, using the diagonal elements of the upper triangular factor of the matrix A as pivots. This method is not completely stable, since the linear rotations may still explode if the matrix A has a large condition number. A different approach is given in Chapter 2 of this thesis, see also [Jain1986b], where a numerically robust feed forward direct method is described which uses only Givens rotations.

1.1.2 SYSTOLIC ARRAYS

DEFINITION

Systolic arrays are defined to be arrays of synchronously operating processor elements, connected in a regular mesh with interconnections of length independent of the size of the array. This definition is the same one given in [Rao1985]. The first introduction of systolic arrays dates back to 1958 [Unge1958], [McCl1958]. With the maturing of VLSI technology they were re-introduced and their potential for compact integration was recognized [Kung1979a], [Leis1981]. Since then, many authors have illustrated how different numerical (see for instance [Kung1979b], [Kung1982], [McWh1983], [Kung1988], [Chua1985], [Schr1985], [Delo1986], [Nava1986], [Jain1986b] and [Krek1988]) as well as combinatorial algorithms, such as sorting and transitive closure

algorithms (see for instance [Lipt1986], and [Kung1988]), can be executed on systolic arrays.

SYSTOLIC ARRAYS FOR SOLVING SYSTEMS OF LINEAR EQUATIONS

Systolic arrays for the LU and QR factorization of matrices have been presented by different authors [Gent1981], [Leis1981], [Ahme1982]. Linear arrays for the back-substitution were presented in [Leis1981], [Gent1981]. In Figure 1.1(a) a systolic array is shown for the QR factorization of a 4×4 matrix $A = [a_{ij}] = QR$, with $R = [r_{ij}]$. At the same time the array also computes $\mathbf{y} = Q^t\mathbf{b}$, as requested in (1.1.b), with $\mathbf{y} = [y_1 \cdots y_4]^t$ and $\mathbf{b} = [b_1 \cdots b_4]^t$. Since the processor elements of the array operate synchronously, the input data at the inputs at the left of the top two processor elements must be delayed one and two time steps, denoted by "-" and "- -", respectively. The squares denote processor elements, which are assumed to have zero processing delay. An interconnection with a delay is denoted by an arrow through a rectangle. An unnumbered rectangle denotes a single delay unit. Otherwise, the number denotes the number of delay units associated with the interconnection.

Each processor element computes the angle needed to zero out an entry in the computation of the upper triangular factor of the coefficient matrix, and applies rotations over this angle on the rest of its input data. The angles α are stored locally at the processor elements, while the computation of the upper triangular matrix factor progresses upwards in the array. The entries of the upper triangular factor R and the vector $\mathbf{y}$, leaving at the top of the array, must be stored and re-ordered for backsubstitution on the bidirectional linear array in Figure 1.1(b).

The entries y_i of the vector $\mathbf{y} = Q^t\mathbf{b}$ are propagating to the right, while the computed entries x_i of the solution vector $\mathbf{x}$ propagate to the left. Notice that the arrays in Figure 1.1(a) and 1.1(b) are completely different. Not only in terms of processor elements but also in terms of data flow. Also notice that the backsubstitution array starts its computations with y_4 and r_{44}. These are the last elements produced by the factorization array of Figure 1.1(a), so that there is no concurrency between the computations in the

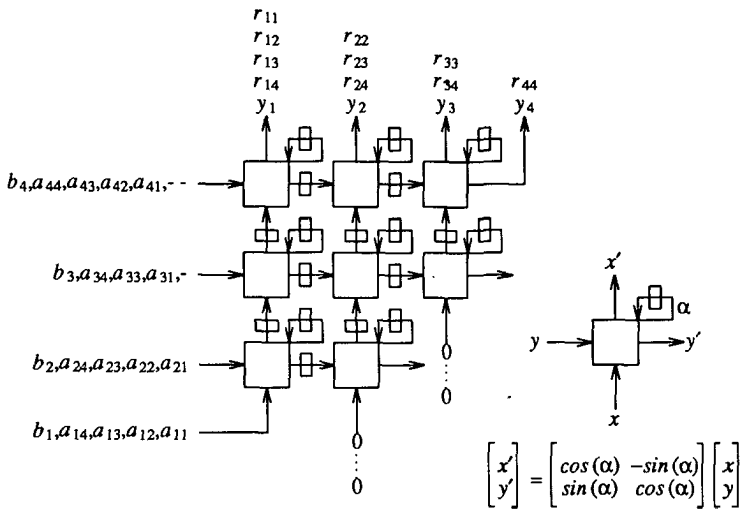


Figure 1.1. (a) Systolic array for the *QR* factorization.

two arrays. This lack of concurrency is a serious bottleneck in systolic implementations of the direct methods with backsubstitution.

A systolic array for Faddeev's method is shown in Figure 1.2 for an example where $A \in \mathbf{R}^{4\times4}$. Each processor element computes a coefficient α needed to zero out an element in the computation of the *LU* factorization of the matrix $[A^t \mid -I]^t$. Once this coefficient is computed, the processor element applies linear rotations with this coefficient on the rest of its input data.

Comparing the combination of the factorization and backsubstitution array to the array for Faddeev's method, the simplicity of the last one is striking. But, more important, as shall be illustrated next by a calculation of the number of time steps needed to compute the solution of $A\,\mathbf{x} = \mathbf{b}$ by the arrays for both methods, the backsubstitution is a bottleneck in a systolic implementation. From a generalization of Figure 1.1(a) it follows that the *QR* factorization of an $N\times N$ matrix takes $3N-2$ time steps to compute the upper triangular factor and the transformation of the right hand side vector. And from a generalization of Figure 1.1(b) it follows that the backsubstitution requires $4N-3$ time steps

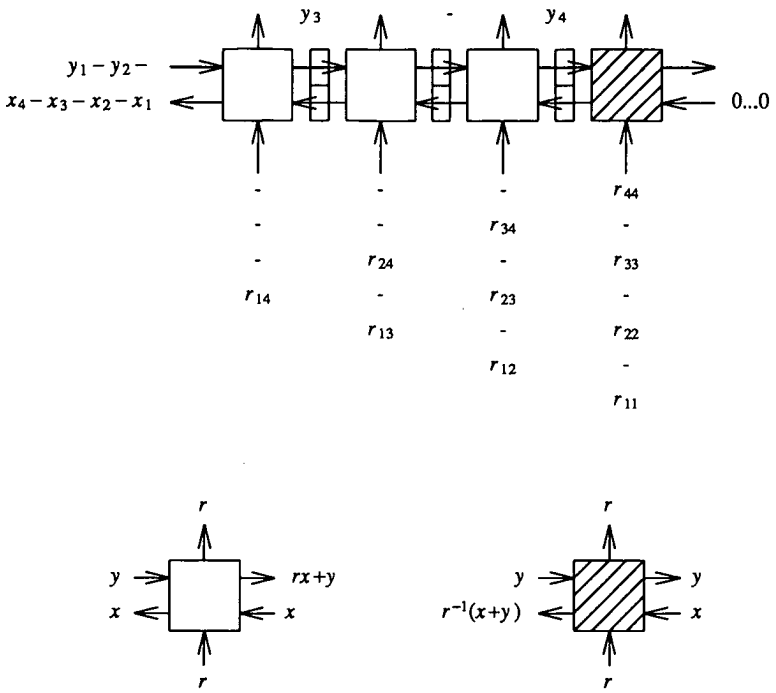


Figure 1.1. (b) Systolic array for the backsubstitution.

to be completed on the linear array ($N-1$ steps to load the array with the elements of the transformed right hand side, $2N-1$ time steps to compute all elements of the solution vector and $N-1$ time steps to shift the last computed element of the solution vector out of the array). Due to lack of parallelism or pipelining between the computations of the QR factorization and the backsubstitution, the total number of time steps to compute the solution is the addition of the number of time steps for each of them. Hence, the total number of time steps is $7N-5$ in this case. On the other hand, from a generalization of the array in Figure 1.2, we find that only $3N$ time steps are required to compute the solution in this case. This is roughly equal to the number of time steps required for the QR factorization. And Faddeev's array computes the solution roughly twice as fast as the combination of the QR factorization and backsubstitution array. Unfortunately, the Faddeev array is only useful for solving positive definite systems of linear equations, because

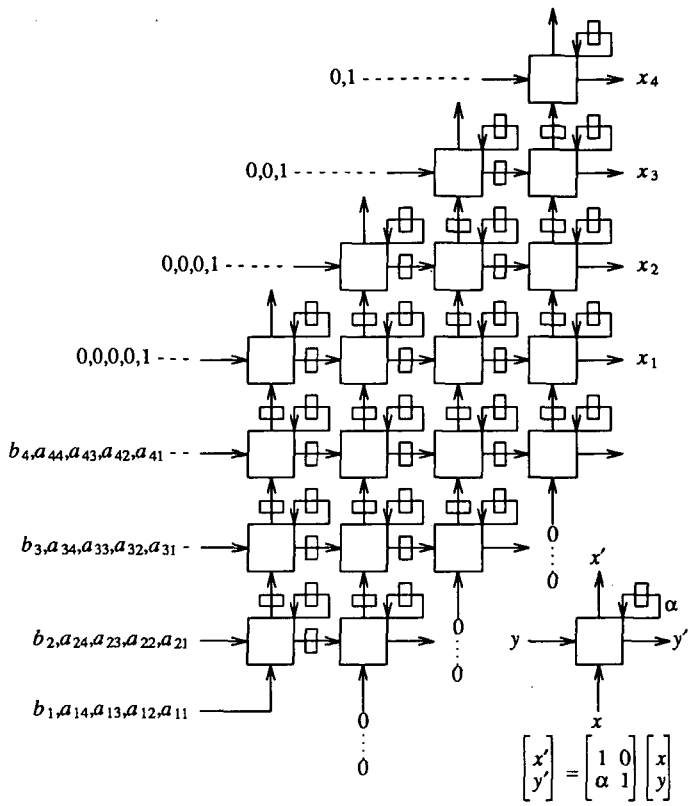


Figure 1.2. The Faddeev array for solving systems of linear equations.

pivoting, which is not straight forwardly done on such arrays due to lack of global communication paths, is not needed in such cases[1]. In Chapter 2 we derive a class of feed

1. In [Royc1988] it was shown that one can devise LU factorization algorithms with partial pivoting, which can be mapped to systolic arrays. However, the sacrifice being made in these algorithms is that the rows of the upper triangular factor are not naturally ordered at the outputs of the array. Thus, the output at which an entry of the solution vector appears will be depending on the coefficient matrix A. This property implies additional overhead for locating the entries of the solution vector at the outputs of the array in practical implementations.

forward direct methods of which some do not suffer from such pivoting problems. It will appear that the methods of this class can be mapped onto a triangular array, similar to the one in Figure 1.1(a), which is smaller than the array for Faddeev's method.

1.2 MAPPING REGULAR RECURRENT ALGORITHMS TO SYSTOLIC ARRAYS

Matrix operations, such as matrix-matrix multiplication and matrix factorization, have a highly regular computational structure. These operations are expressed in the form of multi-dimensional recurrence equations (which will be referred to as *regular recurrent algorithms*) [Kung1979b], [Mead1980]. The dependencies among the computations in these algorithms are, or can be forced to be regular. This regularity simplifies the analysis of the algorithms in the context of parallel execution of their computations [Karp1967] and it facilitates simple mappings of the algorithms on systolic arrays [Mold1983], [Rao1985], [Fris1986], [Rao1986]. The techniques used in this thesis for scheduling and mapping of the algorithms of Chapter 2 on systolic arrays are to be found in [Rao1985]. For convenience of the reader we shall give a brief review of these techniques in Section 1.2.1.

The synthesis techniques for systolic arrays, presented in [Rao1985] and [Fris1986], yield systolic arrays which are scaled to the size of the problem. Such arrays are called full size systolic arrays (FSA's). However, in practice we cannot go on adding or deleting processor elements in a systolic array when the size of the problem changes. Instead, the systolic array must be embedded in an environment which handles fluctuations of the problem size. Different strategies can be followed to partition a large problem on a small size array. Most notable are the strategies described in [Mold1986], [Hori1987], [Neli1988].

Unfortunately, the partitioning solutions described there have certain shortcomings. The partitioning strategy described in [Mold1986] divides the problem in parts which are

solved in sequence on the small size array. Each processor element of the array is assigned a sequence of computations, for each part. However, although a processor element may have finished its computations for a part, it has yet to wait until the rest of the processor elements of the array have completed their computations for the part, before it can start with the computations of the next part. This implies inefficient use of computational resources.

The partitioning strategy described in [Neli1988] clusters groups of, say, p processor elements on a single processor. The p processor elements in a group are scheduled in sequence and may therefore, be replaced by a single processor element. The analysis of this strategy was carried out only for 1-D groups of processor elements. Independently, in [Hori1987] the 1-D results were generalized for the clustering of 2-D groups of processor elements. However, questions were left open such as how the groups are positioned relatively to each other and under which conditions such 2-D groups do exist. In Section 1.2.2 the two types of partitioning strategies are reviewed and in Chapter 3 it is shown how to eliminate the shortcomings of the strategies described in [Mold1986], [Hori1987], [Neli1988].

1.2.1 SYNTHESIZING SYSTOLIC ARRAYS FROM REGULAR RECURRENT ALGORITHMS

REGULAR RECURRENT ALGORITHMS AND DEPENDENCY GRAPHS

A regular recurrent algorithm has the following definition[2].

2. In [Mold1983], [Rajo1987], [Bu1988], [Dong1988], [Royc1988] and [Wong1988] systematic procedures are presented for the derivation of such algorithms from code written in a von Neumann high level programming language.

Definition 1.1 : *a regular recurrent algorithm is defined to be a 7–tuple* $\{I^n, V, C, F_V, F_C, s, D\}$, *where :*

1. I^n *is the index set of the algorithm, which is a lexicographically ordered collection of tuples* $(i_1,...,i_n)$, *where the* i_1 *to* i_n *assume values in (a sub set of)* $\mathbf{Z}$. *A tuple* $(i_1, \ldots, i_n)$ *is called an index point and to each index point there is associated an index vector* $\mathbf{i} = [i_1 \cdots i_n]^t$.

2. *V is a set of indexed variables that are defined at every point in the index set. A variable a at index point* $\mathbf{i}$ *will be denoted by* $a(\mathbf{i})$ *or* $a_{\mathbf{i}}$.

3. *C is a set of control variables, which are defined with the same denotational conventions as the variables in V.*

4. F_V *is a set of functional relations among the variables of V, restricted to be such that, if* $a(\mathbf{i})$ *is computed using* $a(\mathbf{i} - \mathbf{d})$, *then* $\mathbf{d}$ *is a constant vector, called displacement vector, independent of* $\mathbf{i}$ *and the extent of the index space (if* $(\mathbf{i} - \mathbf{d})$ *falls outside* I^n, *then* $a(\mathbf{i} - \mathbf{d})$ *is an input variable of the algorithm).*

5. F_C *is a set of functions, which define functional relations between the variables of C in a way similar to* F_V.

6. *s is a selector which selects functions from* F_V, *depending on the value of the variables in C.*

7. *D is the set of displacement vectors in the algorithm.*

Definition 1.2 : *The index set* I^n *and the set of displacement vectors D define a dependency graph* $G=\{I^n, D\}$ *of the algorithm, where* I^n *is the set of vertices and D is the set of directed edges.*

The algorithm is said to be regular since the dependencies among the variables are constant. Let $I^3 = \{(i_1,i_2,i_3) \mid 1 \le i_1 \le N,\ l_1(i_1) \le i_2 \le u_1(i_1),\ l_2(i_1,i_2) \le i_3 \le u_2(i_1,i_2)\}$ be a lexicographically ordered index set and $D = \{\mathbf{d}_1, \ldots, \mathbf{d}_w\}$. Let $C = \{a_1,...,a_p\}$,

$V = \{a_{p+1},...,a_w\}$, $F_V = \{F_1,...,F_q\}$, where $F_i(a_{p+1}(\mathbf{i})...a_w(\mathbf{i})) = (a_{p+1}(\mathbf{i}+\mathbf{d}_1)...a_w(\mathbf{i}+\mathbf{d}_w))$. Let $F_C = \{F_{q+1}\}$, where $F_{q+1}(a_1(\mathbf{i})...a_p(\mathbf{i})) = (a_1(\mathbf{i}+\mathbf{d}_1)...a_p(\mathbf{i}+\mathbf{d}_p))$. And let s be a selector function from V_C to F_V. The regular recurrent algorithm defined by the above is given in Figure 1.3. The selector function is implemented by the **case** statement and the statement "**initializations**" refers to the set of input variables of the algorithm. The algorithm is said to be in *input standard* form, because all variables in the domains of the functions F_i, $i=1,...,q+1$, are at the same index point.

MAPPING REGULAR RECURRENT ALGORITHMS ON FULL SIZE SYSTOLIC ARRAYS

From the dependency graph of the algorithm we can determine a *schedule* for the computations at the index points. One possible schedule S is the lexicographical order of the index points. This schedule corresponds to the sequential order in which the algorithm in Figure 1.3 is defined. Of all possible schedules we restrict ourselves to *linear schedules* [Rao1985]. This means that we identify an ordered set of parallel hyperplanes in the index space, which contain only index points in which computations can be scheduled simultaneously. Hyperplane k is characterized by the equation $\mathbf{s}^t\mathbf{i} = \rho_k$, where ρ_k is a constant and $\mathbf{s}$ is the normal of the plane. In order to make this vector unique, we choose it to be a vector which is such that there is at least one coprime pair of entries. This vector is referred to as the *schedule* vector and the hyperplane is referred to as the *schedule* plane. The sequence $\{\rho_1, \cdots, \rho_n\}$, where $\rho_{i+1} > \rho_i$, is a new sequence of schedule events for the computations of the algorithm. The fact that all points in a schedule plane are independent (i.e., they can be scheduled in parallel) means that there are no displacement vectors in a schedule plane. Hence, the innerproducts of the schedule vector $\mathbf{s}$ and any displacement vector in the dependency graph is non-zero. Moreover, in order to have causality it is further required that these innerproducts are positive. This leads to the inequality :

$$\mathbf{s}^t[\mathbf{d}_1 \ \cdots \ \mathbf{d}_w] \geq [1 \ \cdots \ 1]. \tag{1.12}$$

So far the index points in the dependency graph have been partitioned into sets of points which are scheduled in parallel. By mapping index points, scheduled at different

initializations;
for $i_1 = 1$ **to** N
  for $i_2 = l_1(i_1)$ **to** $u_1(i_1)$
    for $i_3 = l_2(i_1, i_2)$ **to** $u_2(i_1, i_2)$
      case $[a_1(\mathbf{i}) \cdots a_p(\mathbf{i})]$
        $[cv_{11} \cdots cv_{1p}] : (a_{p+1}(\mathbf{i}+\mathbf{d}_{p+1}) \cdots a_w(\mathbf{i}+\mathbf{d}_w)) = F_1(a_{p+1}(\mathbf{i}) \cdots a_w(\mathbf{i}));$
        .
        .
        .
        $[cv_{q1} \cdots cv_{qp}] : (a_{p+1}(\mathbf{i}+\mathbf{d}_{p+1}) \cdots a_w(\mathbf{i}+\mathbf{d}_w)) = F_q(a_{p+1}(\mathbf{i}) \cdots a_w(\mathbf{i}));$
      endcase
      $(a_1(\mathbf{i}+\mathbf{d}_1) \cdots a_p(\mathbf{i}+\mathbf{d}_p)) = F_{q+1}(a_1(\mathbf{i}) \cdots a_p(\mathbf{i}));$
    endfor
  endfor
endfor

Figure 1.3. A regular recurrent algorithm with three nested loops.

time steps ..., ρ_i, ρ_{i+1}, ..., to a single processor element we can minimize the number of processor elements needed for the execution of the algorithm. Let the integer vector $\mathbf{t}$ define a line $\mathbf{i} + v\mathbf{t}$, $\mathbf{i} \in \mathrm{I}^n$ and $v \in \mathbf{Z}$, on which all index points are not simultaneously scheduled. The vector $\mathbf{t}$ is made unique requiring that it has at least one coprime pair of entries. Then, with an $(n-1)\times n$ transformation matrix T, $T\mathbf{t} = 0$, we can map all such index points $\mathbf{i} + v\mathbf{t}$ to a single processor element at location $\bar{\mathbf{i}} = T(\mathbf{i} + v\mathbf{t}) = T\mathbf{i}$. This processor element executes the computations for all the points $(\mathbf{i} + v\mathbf{t}) \in \mathrm{I}^n$, for $v \in \mathbf{Z}$. We refer to $\mathbf{t}$ as the *projection vector*. The collection $\mathrm{I}^{n-1} = \{\bar{\mathbf{i}} \mid \bar{\mathbf{i}} = T(\mathbf{i} + v\mathbf{t}), \mathbf{i} \in \mathrm{I}^n, v \in \mathbf{Z}\}$ is referred to as the *processor space* and is envisualized as the set of processor elements in the full size array (FSA). A processor element $\bar{\mathbf{i}} = T(\mathbf{i} + v\mathbf{t})$ is scheduled at events $\{\mathbf{s}^t(\mathbf{i} + v\mathbf{t}) \mid v \in \mathbf{Z}\}$. The set of interconnections in the FSA is given by $\bar{\mathrm{D}} = \{\bar{\mathbf{d}} \mid \bar{\mathbf{d}} = T\mathbf{d}, \mathbf{d} \in \mathrm{D}\}$ and the number of delay units along an interconnection $\bar{\mathbf{d}} = T\mathbf{d}$ is given by $\alpha = \mathbf{s}^t\mathbf{d}$. (Note that if $\mathbf{s}^t\mathbf{t} = 0$ we would map index points, at which computations are scheduled simultaneously, on the same processor element. This is not allowed

and therefore, it is required that $\mathbf{s}^t\mathbf{t} \neq 0$).

Summarizing, a procedure for synthesizing full size systolic arrays from regular recurrent algorithms is as follows :

1. Find a solution $\mathbf{s}$ of the inequality :

$$\mathbf{s}^t[\mathbf{d}_1 \cdots \mathbf{d}_w] \geq [1 \cdots 1].$$

2. Select a direction of projection $\mathbf{t}$, such that $\mathbf{s}^t\mathbf{t} \neq 0$.

3. Transform the index set of the algorithm according to the rule :

$$\begin{bmatrix} T \\ \mathbf{s}^t \end{bmatrix} \mathbf{i} = \begin{bmatrix} \bar{\mathbf{i}} \\ \rho_k \end{bmatrix}. \tag{1.13}$$

4. Transform the dependencies of the dependency graph according to the rule :

$$\begin{bmatrix} T \\ \mathbf{s}^t \end{bmatrix} (\mathbf{d}_1 \cdots \mathbf{d}_w) = \left[\begin{bmatrix} \bar{\mathbf{d}}_1 \\ \alpha_1 \end{bmatrix} \cdots \begin{bmatrix} \bar{\mathbf{d}}_w \\ \alpha_w \end{bmatrix} \right]. \tag{1.14}$$

5. Identify the set of processor elements of the FSA as the image $\mathrm{I}^{n-1} = \{\bar{\mathbf{i}}\}$ of the index set I^n under the map T. Identify the set of the interconnections between the processor elements as the image $\bar{\mathrm{D}} = \{\bar{\mathbf{d}}_j\}$ of the set D under transformation T.

6. Identify the schedule sequence $S(\bar{\mathbf{i}})$ of a processor element $\bar{\mathbf{i}} = T(\mathbf{i} + v\,\mathbf{t})$, $v \in \mathbb{Z}$, as the sequence $S(\bar{\mathbf{i}}) = \{\rho_v \mid \rho_v = \mathbf{s}^t(\mathbf{i} + v\,\mathbf{t}),\ (\mathbf{i} + v\,\mathbf{t}) \in \mathrm{I}^n \text{ and } v \in \mathbb{Z}\}$, which is sorted on the values ρ_v.

7. Identify the delay along an interconnection $\bar{\mathbf{d}}_j = T\,\mathbf{d}_j$, as $\alpha_j = \mathbf{s}^t\,\mathbf{d}_j$.

In this procedure the order of steps 1) and 2) may be reversed. Steps 1) and 2) define a *space-time* partitioning of the algorithm, since the transformation $T(\mathbf{i} + v\,\mathbf{t})$ specifies the spatial coordinates of the processor elements in the FSA and $\mathbf{s}^t(\mathbf{i} + v\,\mathbf{t})$ specifies the time coordinates at which processor elements are scheduled. Steps 1) to 7) have been implemented in SYSTARS [Omtz1987], [Omtz1988], a CAD tool for designing systolic

arrays from regular recurrent algorithms.

To illustrate the above procedure, we give an example of designing an FSA for the matrix-matrix multiplication $FX = \tilde{X}$, where $F = [f_{ij}]$, $X = [x_{ij}]$ and $\tilde{X} = [\tilde{x}_{ij}]$ are $N \times N$ matrices. The matrix-matrix multiplication is written as a regular recurrent algorithm as shown in Figure 1.4.

The index set I^3 of this algorithm is the subset $I^3 = \{(i,j,k) \mid 1 \leq i,j,k \leq N\}$ and the set of displacement vectors is $D = \{[0\ 1\ 0]^t, [1\ 0\ 0]^t, [0\ 0\ 1]^t\}$. The dependency graph $DG = \{I^3, D\}$ is depicted in Figure 1.5, for $N = 4$. At a vertex (i,j,k) the variables $f_{i,j+1,k}$, $x_{i+1,j,k}$ and $\tilde{x}_{i,j,k+1}$ are computed and communicated to nodes at relative position $[0\ 1\ 0]^t$, $[1\ 0\ 0]^t$ and $[0\ 0\ 1]^t$, respectively.

Let us make the choice $\mathbf{s} = [1\ 1\ 1]^t$ for the schedule vector and $\mathbf{t} = [0\ 0\ 1]^t$ for the projection vector. Then, we may choose for the transformation T :

$$T = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}.$$

By applying the transformation $[T^t \mid \mathbf{s}]^t$ to the index vectors of I^3 and the displacement vectors of D we obtain a 4×4 systolic array for matrix-matrix multiplication as shown in Figure 1.6. The orientation of the interconnections and the number of delays per interconnection are found from (1.14).

At the inputs at the bottom of the array we find the columns of the matrix X and at the inputs at the left we find the rows of the matrix F. The elements of the matrix $\tilde{X}$ are computed iteratively according to Figure 1.4 and stored in the delays of the loops. Note that variations in the projection vector $\mathbf{t}$ cause variations in the topology of the systolic array. Similarly, variations in the schedule vector $\mathbf{s}$ cause variations in the number of delay units along the interconnections of the systolic array.

```
f_{i,0,k} = f_{ik}; /* initialization */
x_{0,j,k} = x_{kj}; /* initialization */
x̃_{i,j,0} = 0; /* initialization */
for i = 1 to N
 for j = 1 to N
  for k = 1 to N
   f_{i,j+1,k} = f_{i,j,k};
   x_{i+1,j,k} = x_{i,j,k};
   x̃_{i,j,k+1} = x̃_{i,j,k} + f_{i,j,k} x_{i,j,k};
  endfor
 endfor
endfor
```

Figure 1.4. Regular recurrent algorithm for matrix-matrix multiplication.

1.2.2 MAPPING LARGE PROBLEMS TO SMALL SYSTOLIC ARRAYS

A systolic array designed by the procedure outlined in the previous section, has a size which is proportional to the "accidental" size of the problem. Execution of the problem with a different size requires a re-scaling of the array. Here we face a difficulty with ever growing sizes of the problem. In practical situations there is a limit to the up-scaling of a systolic array, due to technological and economic reasons. Hence, it is important to look for partitioning strategies which can solve this scaling problem, so that once the array has been designed it does not have to be re-scaled for a particular problem size.

The partitioning strategies introduced here, act directly on the full size array. The number of processor elements of this array is directly proportional to the size of the problem at hand[3]. The FSA is reduced to a desirable size, yielding a so-called reduced size

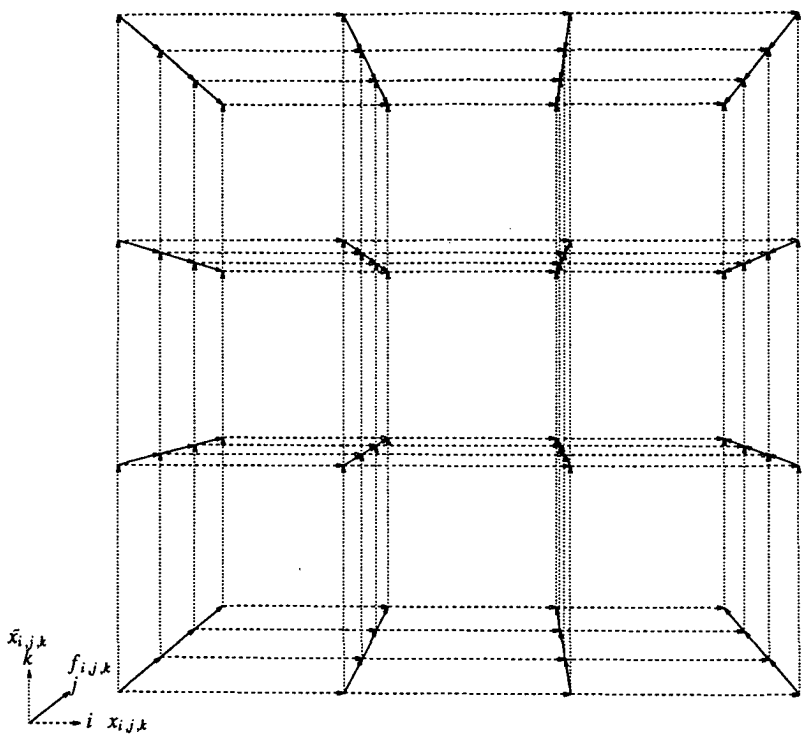


Figure 1.5. Dependency graph of matrix-matrix multiplication for $N = 4$.

array (RSA). It correctly executes the problem partitions and is again systolic.

Common to partitioning strategies is the tessellation of the FSA in tiles of, say, p processor elements each. The way in which the computations in these tiles are scheduled will determine the RSA and how the problem partitions are scheduled on the RSA. The

3. Partitioning of the problem itself [Hell1985] is equivalent to the partitioning of the full size array, since problem and array are equivalent with respect to the dependency graph. This equivalence is due to the fact that the transformation matrix in (1.13) is non-singular.

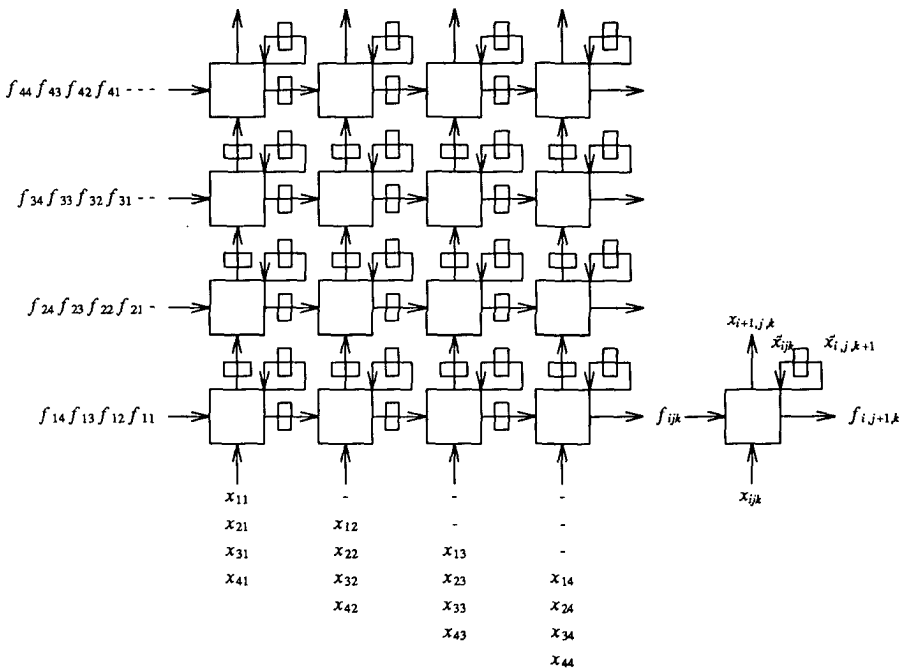


Figure 1.6. A full size systolic array for 4×4 matrix-matrix multiplication.

various ways in which an FSA can be partitioned can be traced to either of the following three cases (or combinations of these) of scheduling the computations of the tiles on a (yet to be designed) RSA.

1. All computations in a tile of the FSA are executed by a single processor element of the RSA. Thus the RSA has as many processor elements as there are tiles in the FSA. In general the processor elements and the topology of the RSA are different from those of the FSA.

2. The computations of all tiles in the FSA are executed in pipeline on an RSA. The RSA has the same number of processor elements as a tile and the same interconnections as the FSA. Pipelining among the computations of a single tile, if existing in the FSA, as well as the sequence of computations in a tile is preserved in the

RSA.

3. The same as the previous case, except that there is no pipelining among the computations of different tiles.

We shall briefly discuss these three cases in sequence.

Case 1 :
As an example of the first case we partition the 4×4 matrix-matrix multiplication example. Choosing schedule vector $\mathbf{s} = [1\ 2\ 4]^t$, we obtain a pattern of activity as shown by the snap shots in Figure 1.7. The self-loops are omitted for the sake of clarity and the active processor elements are hatched. In this pattern we can distinguish tiles of 4 processor elements each. In a tile one and only one processor element is scheduled at any time.

Hence, the 4 processor elements in a tile may be replaced by a single one, a so-called cluster processor. In this way the full size array of 16 processor elements is reduced to an array of only 5 processor elements. This method of partitioning is referred to as the *local-sequential-global-parallel* (LSGP) partitioning strategy. The name refers to the fact that the processor elements in a tile are scheduled sequentially, while processor elements in different tiles may be scheduled in parallel. A tile is referred to as *cluster*. The processor elements in a cluster are called virtual processor elements. The processor elements in the RSA are the cluster processors.

The cluster processors have selectors which select the correct input (a feed back from the processor element itself, or an output of another cluster processor) at the right time. These selectors require control information which is derived from the sequencing of activities in a tile. The cluster processors also have state registers which store the intermediate results.

Observe that there is always one and only one processor element active at a schedule time step when shifting the pattern of tiles across the FSA. For instance, we can shift the pattern over a distance of one column of processor elements to the right and still there is

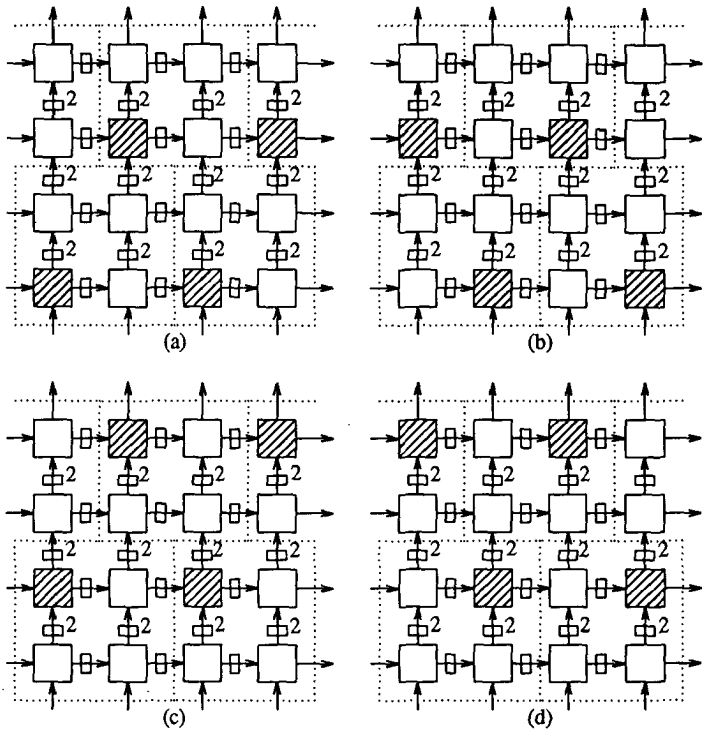


Figure 1.7. Snapshots of the tiles in the FSA for schedule vector $\mathbf{s}^t = [1\ 2\ 4]^t$.

one and only one processor element active in a tile, at a schedule time step. Depending on the shape of the FSA the pattern of tiles may be shifted in a position where the FSA is covered by a minimum of tiles. For instance, the positioning of the pattern of tiles in Figure 1.7 requires 5 tiles to cover the complete FSA. Shifting the pattern over a distance of one row up needs 8 tiles to cover the complete FSA. And since the number of tiles is equal to the number of cluster processors in the RSA, it follows that the positioning in Figure 1.7 results in only 5 cluster processors, compared to the 8 processors of the second positioning.

The LSGP partitioning strategy was considered in [Neli1988] for 1-D clusters, and in [Hori1987] for 2-D clusters. The analysis given in [Hori1987] does only find a set of

processor elements which may be clustered, but leaves open questions such as how the clusters are positioned relatively to each other and under what conditions which type of 2-D clusters can be proven to exist. As can be seen from Figure 1.7 ($s^t = [1\ 2\ 4]$) and Figure 1.8 ($s^t = [2\ 3\ 6]$) the clusters can form different patterns, due to different relative positions. Moreover, the relative positions seem to be predetermined by the choice of schedule vector. For example, the relative positions of clusters in Figure 1.8 are not possible for schedule vector $s^t = [1\ 2\ 4]$ in Figure 1.7. In Chapter 3 an analysis is given to show under what circumstances which 2-D (and 1-D) clusters with certain relative positions exist.

Case 2 :

As an illustration of the second case of partitioning we tessellate the matrix-matrix multiplication array of Figure 1.6 in 4 tiles, as shown in Figure 1.9.

In this partitioning strategy the RSA is identical to a single tile of processor elements in the FSA. The computations of a tile are scheduled on the RSA in the same way as they are scheduled in the tile itself, whereas the computations of different tiles are pipelined on the RSA. This partitioning method is referred to as the *local-parallel-global-pipelined* (LPGP) partitioning strategy. The name refers to the fact that the processor elements in a tile are scheduled in parallel, while the tiles are scheduled in pipeline on the RSA.

Let us denote (in clockwise direction, starting at the bottom left tile) the tiles in Figure 1.9 by (1,1), (1,2), (2,2) and (2,1), respectively. Now, since all computations inside tile (1,1) precede the computations in the other tiles, the computations for this tile are the first ones to be scheduled on the RSA. The initialization data for the computations of tile (2,1) are collected in the buffers connected to the horizontally oriented outputs of the RSA (see Figure 1.10). Similarly, the initialization data for the computations of tile (1,2) are collected in the buffers connected to the vertically oriented outputs of the RSA.

After the last computation of the bottom left processor element of tile (1,1) has been scheduled on the bottom left processor element of the RSA, the computations in either tile (1,2) or tile (2,1) may be scheduled on this processor element. In this way the

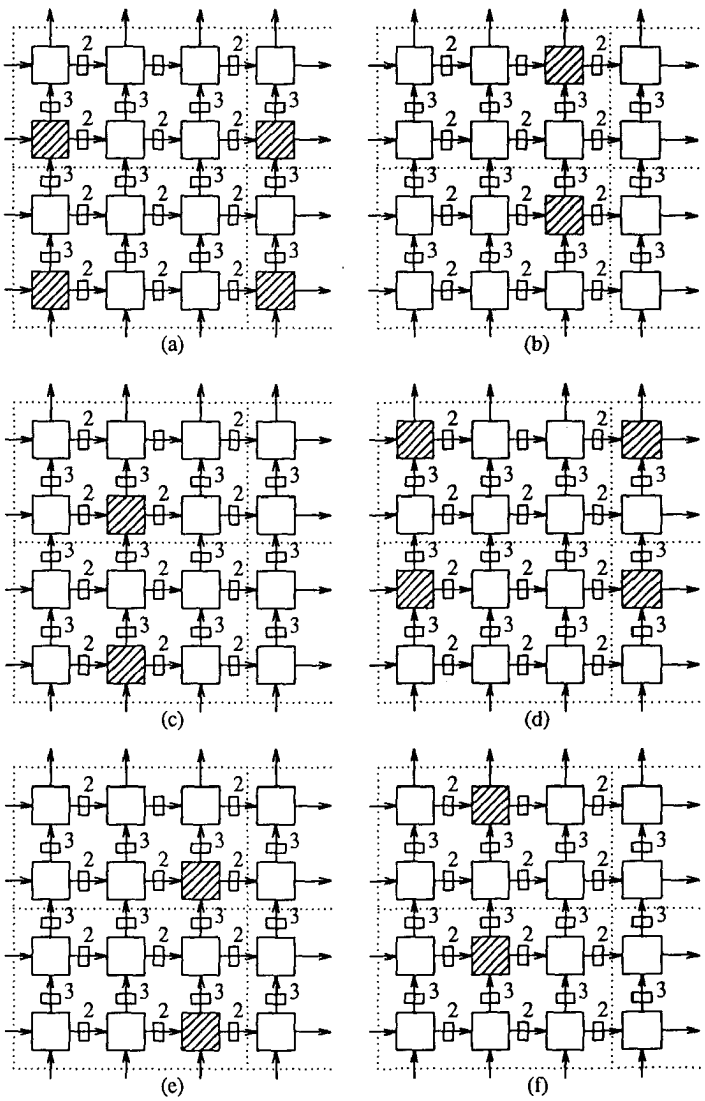


Figure 1.8. Same as Figure 1.7, but for schedule vector $\mathbf{s}^t = [2\ 3\ 6]$.

computations of different tiles are pipelined on the RSA. We may arbitrarily select the next tile, for instance tile (2,1). Now the data collected in the buffers at the vertically

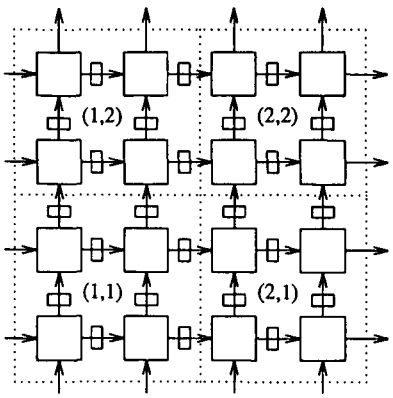


Figure 1.9. Tessellation of the matrix-matrix multiplication array.

oriented outputs of the RSA are part of the initializations of the computations of tile (2,2). After the last computation of the bottom left processor element of tile (2,1) has been scheduled on the RSA, the computations of tile (1,2) are scheduled. The data which is collected in the buffers at the horizontally oriented outputs of the RSA are the rest of the initializations for the computations of tile (2,2). Finally, the computations of tile (2,2) are scheduled on the RSA, after the last computation of the bottom left processor element of tile (1,2) has been scheduled.

Case 3 :

The third case is identical to the second one, except that the computations of the next tile are scheduled only after the RSA has finished all the computations of the previous tile. That is, the computations in different tiles are executed sequentially on the RSA. The disadvantage of this case, compared to the previous one, is obvious. This partitioning method is referred to as the *local-parallel-global-sequential* (LPGS) partitioning strategy. The name refers to the fact that the processor elements in a tile are scheduled in parallel, while the tiles themselves are scheduled sequentially on the RSA. Partitioning strategies with such schedules are found in [Fort1985], [Hell1985] [Mold1986].

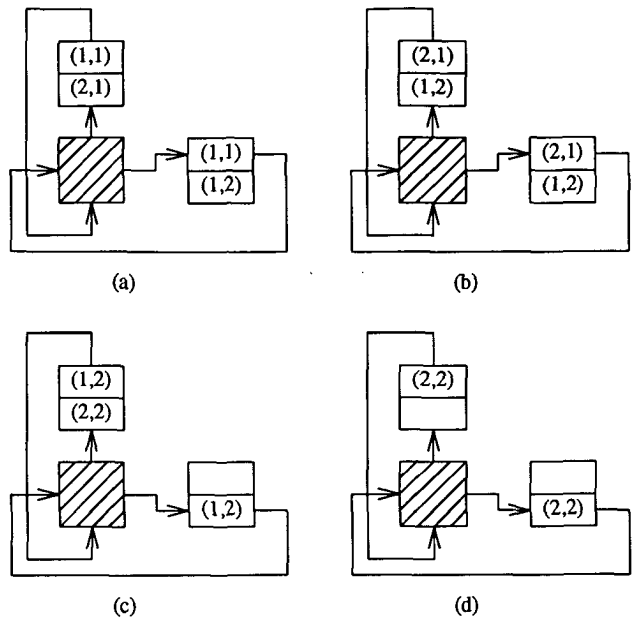


Figure 1.10. Snapshots for the contents of the buffers of the RSA.

The LSGP and the LPGP strategies are two partitioning methods which offer important advantages in the design of practical systolic arrays of fixed dimensions. These advantages are the following. In general the processor elements operate at a much higher I/O bandwidth than busses which supply data to and receive data from the processor elements. If clusters are large enough in an LSGP partitioning, there will be a considerable number of processor elements in each cluster, which do not communicate with processor elements outside the cluster. Since the LSGP partitioning has the effect of serializing the computations in the clusters, the I/O bandwidth of the cluster processors is reduced by a certain factor. Careful choices of cluster sizes may lead to a close match of the I/O bandwidths of the cluster processors and the busses, be it at the expense of increased local memory.

In the LPGP partitioning strategy the I/O bandwidth of the processor elements is unaffected by the partitioning. Moreover all memory to hold the initialization data for the

computations of the tiles is kept outside the RSA itself. This means that the number of processor elements of the RSA and the amount of local memory of a processor element is independent of the size of the instance of the problem. Thus, a combination of the two partitioning strategies may well serve the purpose of solving such important design issues such as I/O bandwidth matching and problem size independency. In Chapter 3 of this thesis a detailed analysis is given of both partitioning strategies. Additional problems, that appear at the boundaries of an FSA when tessellated, are also discussed[4] there.

4. These are problems caused by an incomplete number of processor elements in tiles that may extend across the boundaries of a tessellated FSA. See e.g., Figure 1.7 and Figure 1.8.

2. A CLASS OF HIGHLY STRUCTURED ALGORITHMS FOR SOLVING SYSTEMS OF LINEAR EQUATIONS

In this chapter a class of feed forward methods is presented for solving non singular systems of linear equations $A\mathbf{x} = \mathbf{b}$, $A \in \mathbb{R}^{N\times N}$ and $\mathbf{b} \in \mathbf{R}^N$. The presentation has the following outline. First, in Section 2.1 it will be assumed that the lower triangular factor L and the inverse X^{-1} of the remainder X in the factorization $A = LX$ are given. It is then explained how the solution $\mathbf{x}$ is obtained from the augmented matrix :

$$\begin{bmatrix} L^t & X^{-t} & 0 \\ -\mathbf{b}^t & 0^t & 1 \end{bmatrix},$$

by applying a series of either linear, orthogonal or hyperbolic rotations to this matrix.

Next, in Section 2.2 it is shown how the factors L^t and X^{-t} are properly generated and combined with the result of Section 2.1, to give simple feed forward methods for solving $A\mathbf{x} = \mathbf{b}$. In Section 2.3 the numerical stability of the methods is discussed and in Section 2.4 it is shown that one of the methods is in fact a semi-direct method, in contrast to the rest, which are all direct methods. Finally, in Section 2.5 it is shown how the methods of Section 2.2 can be generalized to perform computations of the kind $CA^{-1}B + D$. This generalization is similar to the generalization of Faddeev's method [Fadd1959].

2.1 FEED FORWARD COMPUTATION OF $\mathbf{x} = A^{-1}\mathbf{b}$

Let $A \in \mathbf{R}^{N\times N}$ be a non-singular matrix and let $\mathbf{b} \in \mathbf{R}^N$ be a column vector. The solution of the equation :

$$A\mathbf{x} = \mathbf{b} \tag{2.1}$$

may be expressed in terms of a given lower triangular factorization :

$$A = LX \tag{2.2.a}$$

where L is a lower triangular matrix and the matrix X is one of the following :

1. $X = U$ is upper triangular (LU factorization of A);
2. $X = Q$ is orthogonal ($QQ^t = Q^tQ = I$) (LQ factorization of A);
3. $X = L^t$ is the upper triangular Cholesky factor of the matrix A in case it is symmetric positive definite (LL^t factorization of A).

Hence, denoting

$$L\mathbf{y} = \mathbf{b}, \tag{2.2.b}$$

Equation (2.1) becomes

$$X\mathbf{x} = \mathbf{y}. \tag{2.2.c}$$

Equations (2.2.b) and (2.2.c) can be rewritten as follows :

$$[\mathbf{y}^t \quad 1] \begin{bmatrix} L^t \\ -\mathbf{b}^t \end{bmatrix} = 0, \tag{2.3.a}$$

$$[\mathbf{y}^t \quad 1] \begin{bmatrix} X^{-t} \\ 0 \end{bmatrix} = \mathbf{x}^t. \tag{2.3.b}$$

The computation of $\mathbf{x}$ in (2.3.b) is based on the following observation. The matrix $[L \mid -\mathbf{b}]^t$ in (2.3.a) is a rectangular $(N+1)\times N$ matrix, so that it can be reduced to an upper triangular form, with zero last row, by applying a proper sequence of rotations to the matrix. Then, since the last row, say $\mathbf{r}$, of this product of rotations satisfies $\mathbf{r} \begin{bmatrix} L^t \\ -\mathbf{b}^t \end{bmatrix} = 0$, it must be proportional to $[\mathbf{y}^t \quad 1]$, given that A is non-singular. Therefore, when this row is

substituted for $[\mathbf{y}^t \ \ 1]$ in (2.3.b), the result will be a vector proportional to the solution vector $\mathbf{x}^t$.

Thus, let the sequence of rotations applied to $[L \mid -\mathbf{b}]^t$ be denoted by the $(N+1)\times(N+1)$ matrix $\Theta(m)$, with $m \in \{0,1,-1\}$:

$$\Theta(m) = \prod_{i=1}^{\overleftarrow{N}} \Theta_i(m), \tag{2.4.a}$$

with :

$$\Theta_i(m) = \begin{bmatrix} I_{i-1} & & & \\ & cos\,(m^{1/2}\alpha_i) & & -m^{1/2} sin\,(m^{1/2}\alpha_i) \\ & & I_{N-i} & \\ & m^{-1/2} sin\,(m^{1/2}\alpha_i) & & cos\,(m^{1/2}\alpha_i) \end{bmatrix} \tag{2.4.b}$$

and such that :

$$\Theta(m) \begin{bmatrix} L^t \\ -\mathbf{b}^t \end{bmatrix} = \begin{bmatrix} R\,(m) \\ 0^t \end{bmatrix}, \tag{2.5}$$

where $R\,(m)$ is upper triangular. That is, the angles α_i are chosen such that an upper triangularization of the matrix $[L \mid -\mathbf{b}]^t$ is obtained in the recursion :

$$\begin{bmatrix} L_i^t \\ -\mathbf{b}_i^t \end{bmatrix} = \prod_{j=1}^{\overleftarrow{i}} \Theta_j(m) \begin{bmatrix} L^t \\ \mathbf{b}^t \end{bmatrix}, \ i=1,2,...,N, \tag{2.6.a}$$

where :

$$\mathbf{b}_i^t = [0 \underset{i}{\cdots} 0 \ b_1^{(i)} \ \cdots \ b_{N-i}^{(i)}]. \tag{2.6.b}$$

The matrix $\Theta(m)$ is $\Sigma(m)$-orthogonal, i.e.,

$$\Theta^t(m)\Sigma(m)\Theta(m) = \Sigma(m), \tag{2.7}$$

with :

$$\Sigma(m) = \begin{bmatrix} I_N & \\ & m \end{bmatrix}. \tag{2.8}$$

For $m = \pm 1$, we also have that :

$$\Theta(m)\Sigma(m)\Theta^t(m) = \Sigma(m). \tag{2.9}$$

As yet we have to prove that the angles α_i exist for given lower triangular matrix $L = [l_{ij}]$ and vector $\mathbf{b} = [b_i]$ in either of the cases $m \in \{0,1,-1\}$. For $m = 0$, $\alpha_i = -\frac{b_1^{(i)}}{l_{ii}}$ (allowing pivoting in case of zero pivots). Using linear rotations, $m = 0$, amounts to an LU factorization in (2.5), with $U = [R^t(0) \mid 0]^t$. For $m = 1$, $tan(\alpha_i) = -\frac{b_1^{(i)}}{l_{ii}}$ always exists, since the function $tan(\alpha_i)$ has image $\mathbf{R}$. Using orthogonal rotations, $m = 1$, amounts to a QR factorization in (2.5), with $R = [R^t(1) \mid 0]^t$.

The use of hyperbolic rotations, $m = -1$, is restricted to special cases. Namely, to those cases where the elements of the matrix L and the vector $\mathbf{b}$ are such that $|tanh(\alpha_i)| = |\frac{b_1^{(i)}}{l_{ii}}| < 1$. Only then do the hyperbolic rotations exist. The following proposition states for which case $\Theta(-1)$ in (2.5) exists.

Proposition 2.1 :

Let $L \in \mathbf{R}^{N \times N}$ be a lower triangular matrix and $\mathbf{b} \in \mathbf{R}^N$ a vector, such that :

$$LL^t - \mathbf{b}\mathbf{b}^t > 0. \tag{2.10}$$

Then, there exists a matrix $\Theta(-1)$ as defined in (2.4.a) and (2.4.b) for $m = -1$, such that :

$$\Theta(-1)\begin{bmatrix} L^t \\ -\mathbf{b}^t \end{bmatrix} = \begin{bmatrix} R(-1) \\ 0^t \end{bmatrix}, \tag{2.11}$$

where $R(-1)$ is upper triangular.

Proof :

For a proof we refer to Appendix A.

□

The case $m = -1$ may seem superfluous since it puts a restriction on the vector $\mathbf{b}$ (see (2.10)). But, as we shall see in Section 2.4, this case is important for computation of maximum entropy approximations of the solution vector.

The proportionality constant between the last row of the matrix $\Theta(m)$ and $[\mathbf{y}^t \ \ 1]$ can be found from the properties of the matrix $\Theta(m)$, as follows. Write for the $(N+1)\times(N+1)$ matrix $\Theta(m)$:

$$\Theta(m) = \begin{bmatrix} \Theta_{11}(m) & \theta_{12}(m) \\ \theta_{21}^t(m) & k(m) \end{bmatrix}, \quad \Theta_{11}(m) \in \mathbb{R}^{N\times N}, \tag{2.12}$$

where $k(m)$ is the proportionality constant. For $m = 0$ it follows from (2.4.a) and (2.4.b) that $k(0) = 1$. Hence, $\theta_{21}(0) = \mathbf{y}$ and, consequently :

$$\Theta(0) \begin{bmatrix} L^t & X^{-t} \\ -\mathbf{b}^t & 0^t \end{bmatrix} = \begin{bmatrix} R(0) & * \\ 0^t & \mathbf{x}^t \end{bmatrix}. \tag{2.13}$$

This equation expresses an LU factorization of the matrix $[L \mid -\mathbf{b}]^t$. For $m = 1$ or $m = -1$ we substitute (2.12) in (2.9). This gives :

$$mk(m) + \theta_{21}^t(m)\theta_{21}(m) = m.$$

Hence, with $\mathbf{y} = k^{-1}(m)\theta_{21}(m)$ it follows that :

$$k(m) = (1 + m\mathbf{y}^t\mathbf{y})^{-1/2}. \tag{2.14}$$

Thus :

$$\Theta(m) \begin{bmatrix} L^t & X^{-t} & 0 \\ -\mathbf{b}^t & 0^t & 1 \end{bmatrix} = \begin{bmatrix} R(m) & * & * \\ 0^t & k(m)\mathbf{x}^t & k(m) \end{bmatrix}, \quad m \in \{1, -1\}. \tag{2.15}$$

Using the $\Sigma(m)$-orthogonality of $\Theta(m)$, we get from (2.15) $LL^t + m\mathbf{b}\mathbf{b}^t = R^t(m)R(m)$, so that (2.15) expresses an updating and a downdating of a Cholesky factorization for $m = 1$ and $m = -1$, respectively.

A numerically important property of the case $m = -1$, for $LL^t - \mathbf{b}\mathbf{b}^t > 0$, is stated in the following theorem.

Theorem 2.1 :

Let $L \in \mathbf{R}^{N\times N}$ *be a lower triangular matrix and* $\mathbf{b} \in \mathbf{R}^N$, *such that the matrix :*

$$\begin{bmatrix} 1 & -\mathbf{b}^t \\ -\mathbf{b} & LL^t \end{bmatrix},$$

is positive definite with all its main diagonal entries equal to 1 *(this is accomplished by a trivial normalization). Then, the magnitude of all entries of the matrices* L_i *and the vectors* $\mathbf{b}_i$ *in the recursion (2.6.a)-(2.6.b), for* $m = -1$, *is bounded by* 1.

Proof :

The proof is deferred to Section 2.2.3.

□

2.2 THE CLASS OF FEED FORWARD ALGORITHMS

In this section it is shown how the results in (2.13) and (2.15) are combined with the lower triangular factorization of the matrix A, to obtain feed forward methods for solving $A\mathbf{x} = \mathbf{b}$.

2.2.1 LU FACTORIZATION

Let $A = LU$ be an LU factorization of the matrix A. Then, an LU factorization of the augmented matrix $[A^t \mid I_N]$ will give :

$$\overline{U}[A^t \mid I_N] = [L^t \mid U^{-t}], \tag{2.16}$$

where $\overline{U} = U^{-t}$. L^t and U^{-t} are the matrix factors appearing in (2.3.a) and (2.3.b). I.e., $X^{-t} = U^{-t}$ in (2.3.b). Next, we show how either of the possible factorizations in (2.13) and (2.15) are combined with the LU factorization in (2.16), such that a feed forward method is obtained for solving $A\mathbf{x} = \mathbf{b}$. Put :

$$U_{ij} = \begin{bmatrix} I_{j-1} & & & & \\ & 1 & & & \\ & & I_{i-j} & & \\ & \alpha_{ij} & & 1 & \\ & & & & I_{N-i} \end{bmatrix}, \quad i,j=1,\ldots,N-1. \tag{2.17}$$

The entries of the first column of the strictly lower triangular part of the matrix $[A^t \mid I_N]$ are eliminated by premultiplication with $\overleftarrow{\prod_{i=1}^{N-1}} U_{i1}$, with appropriate α_{i1}. Hence, the first row of the resulting matrix will be the first row of the matrix $[L^t \mid U^{-t}]$. Next the first rotation $\Theta_1(m)$, in the product $\Theta(m)$ in (2.4.a) is applied to this row and the vector $[-\mathbf{b}^t \mid 0^t]$, such that the first entry of $-\mathbf{b}^t$ is eliminated. Secondly, we eliminate the elements of the second column of the strictly lower triangular part of the matrix $(\overleftarrow{\prod_{i=1}^{N-1}} U_{i1})[A^t \mid I_N]$, by premultiplication with $\overleftarrow{\prod_{i=2}^{N-1}} U_{i2}$, with appropriate α_{i2}. Thus, obtaining the second row of the matrix $[L^t \mid U^{-t}]$. Next, the second rotation $\Theta_2(m)$, in the product $\Theta(m)$ in (2.4.a) is applied to this row and the vector that resulted from the application of the rotation $\Theta_1(m)$ on $[-\mathbf{b}^t \mid 0^t]$. $\Theta_2(m)$ is chosen such that the second element of this vector is eliminated. The above procedure is repeated until all the elements of the vector $-\mathbf{b}^t$ are eliminated against the pivots of the matrix L^t.

Thus, we get the following feed forward computational scheme for $A\mathbf{x} = \mathbf{b}$:

$$\Theta_N(m)(\Theta_{N-1}(m)U_{N-1,N-1}) \cdots (\Theta_2(m)\overleftarrow{\prod_{i=2}^{N-1}}U_{i2})(\Theta_1(m)\overleftarrow{\prod_{i=1}^{N-1}}U_{i1})\begin{bmatrix} A^t & I_N & 0 \\ -\mathbf{b}^t & 0^t & 1 \end{bmatrix} =$$

$$\begin{bmatrix} R(m) & * & * \\ 0^t & k(m)\mathbf{x}^t & k(m) \end{bmatrix} \tag{2.18}$$

($m \in \{0,1,-1\}$). Note, that allowing $m = -1$ is constrained to the condition that $LL^t - \mathbf{b}\mathbf{b}^t > 0$, where L is the lower triangular factor of the LU factorization of the matrix A.

Observe that (2.18) expresses an LU factorization of the matrix $[A \mid -\mathbf{b}]^t$ in case we use linear rotations for the $\Theta_i(m)$ ($m = 0$). I.e., :

$$\Theta_N(0)(\Theta_{N-1}(0)U_{N-1,N-1}) \cdots (\Theta_2(0)\overleftarrow{\prod_{i=2}^{N-1}}U_{i2})(\Theta_1(0)\overleftarrow{\prod_{i=1}^{N-1}}U_{i1})\begin{bmatrix} A^t \\ -\mathbf{b}^t \end{bmatrix} = \begin{bmatrix} R(0) \\ 0^t \end{bmatrix}. \tag{2.19}$$

Since the LU factorization of a matrix is unique, it follows that different orderings of linear rotations can be applied to obtain the result in (2.18). For instance, the following order of linear rotations (indicated by combinations (i,j) of row i and row j) gives the same result as in (2.18) :

$(N,N+1)$, $(N-1,N)$, ..., $(1,2)$ followed by $(N,N+1)$, $(N-1,N)$, ..., $(2,3)$, etc.

A possible regular, systolic processor array which uses the scheme of linear rotations in (2.19) is shown in Figure 2.1 for $N = 4$. This array is identical to the one in Figure 1.1(a), except that now the processor elements perform linear rotations and there is an extra row of linear rotors for the elimination of the elements of the vector $\mathbf{b} = [b_1\ b_2\ b_3\ b_4]^t$.

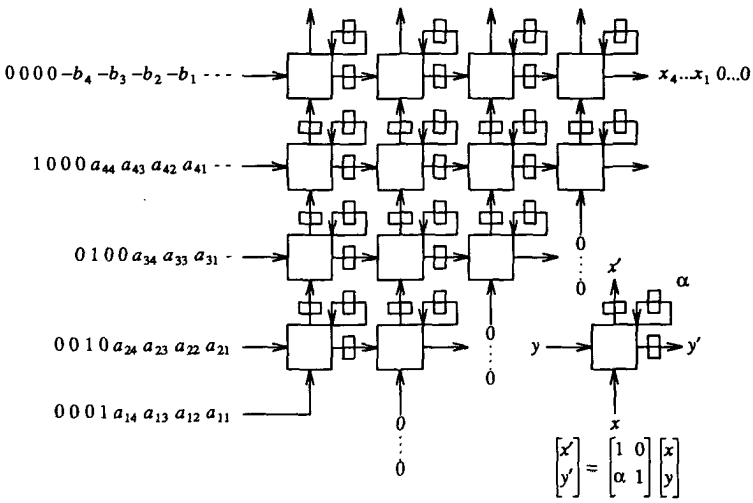


Figure 2.1. Systolic array for the feed forward *LU* method.

2.2.2 QR FACTORIZATION

Let $A = LQ$ be the LQ factorization of the matrix A. Then, a QR factorization of the matrix $[A^t \mid I_N]$ will give :

$$\overline{Q}[A^t \mid I_N] = [L^t \mid Q], \tag{2.20}$$

where $\overline{Q} = Q$. L^t and Q are the matrix factors appearing in (2.3.a) and (2.3.b). I.e., $X^{-t} = Q$ in (2.3.b). The factorizations in (2.13) and (2.15) are combined with the factorization in (2.20) in the same way as was explained for the LU factorization of $[A^t \mid I_N]$, but using circular rotations for the QR factorization of $[A^t \mid I_N]$. That is, putting :

$$Q_{ij} = \begin{bmatrix} I_{j-1} & & & & \\ & cos(\alpha_{ij}) & & -sin(\alpha_{ij}) & \\ & & I_{i-j} & & \\ & sin(\alpha_{ij}) & & cos(\alpha_{ij}) & \\ & & & & I_{N-i} \end{bmatrix}, \tag{2.21}$$

we get the following feed forward computational scheme for $A\mathbf{x} = \mathbf{b}$:

$$\Theta_N(m)(\Theta_{N-1}(m)Q_{N-1,N-1}) \cdots (\Theta_2(m)\overleftarrow{\prod_{i=2}^{N-1}}Q_{i2})(\Theta_1(m)\overleftarrow{\prod_{i=1}^{N-1}}Q_{i1})\begin{bmatrix} A^t & I_N & 0 \\ -\mathbf{b}^t & 0^t & 1 \end{bmatrix} =$$

$$\begin{bmatrix} R(m) & * & * \\ 0^t & k(m)\mathbf{x}^t & k(m) \end{bmatrix} \tag{2.22}$$

($m \in \{0,1,-1\}$). Note, that allowing $m = -1$ is constrained to the condition that $LL^t - \mathbf{b}\mathbf{b}^t > 0$, where L is the lower triangular factor of the LQ factorization of the matrix A.

Observe that (2.22) expresses a QR factorization of the matrix $[A \mid -\mathbf{b}]^t$ in case we use orthogonal rotations for the $\Theta_i(m)$ ($m = 1$). I.e., :

$$\Theta_N(1)(\Theta_{N-1}(1)Q_{N-1,N-1}) \cdots (\Theta_2(1)\overleftarrow{\prod_{i=2}^{N-1}}Q_{i2})(\Theta_1(1)\overleftarrow{\prod_{i=1}^{N-1}}Q_{i1})\begin{bmatrix} A^t \\ -\mathbf{b}^t \end{bmatrix} = \begin{bmatrix} R(1) \\ 0^t \end{bmatrix}. \tag{2.23}$$

Since the QR factorization of a matrix is unique, it follows that different orderings of orthogonal rotations may be found by which the result in (2.23) is obtained. For instance, the following order of orthogonal rotations (indicated by combinations (i,j) of row i and row j) gives the same result as in (2.22) :

$(N,N+1)$, $(N-1,N)$, ..., $(1,2)$ followed by $(N,N+1)$, $(N-1,N)$, ..., $(2,3)$, etc.

A regular, systolic processor array which uses the scheme of rotations in (2.23) is

identical to the one in Figure 2.1, except that the processor elements perform Givens rotations.

2.2.3 SCHUR-CHOLESKY FACTORIZATION

If the coefficient matrix A is a symmetric positive definite matrix, a fast Cholesky factorization of the matrix is possible with the so-called *Generalized Schur* algorithm [Depr1982], [Delo1984]. This algorithm is a generalization of the Schur algorithm for the Cholesky factorization of a symmetric and positive definite Toeplitz or close to Toeplitz matrix [Dewi1978], [Lev1984]. First we shall present the generalized Schur algorithm and its properties, after which it is shown how to combine this algorithm with the factorizations in (2.13) and (2.15), in order to obtain feed forward algorithms.

The Generalized Schur algorithm

Theorem 2.2 :

Let $A = [a_{ij}] \in \mathbf{R}^{N\times N}$ *be a symmetric positive definite matrix, normalized such that we can write*

$$A = R_L + I_N + R_L^t, \tag{2.24}$$

where $R_L = [a_{ij}]$, $i > j$, *is the strictly lower triangular part of* A. *Put :*

$$U = R_L + I_N \tag{2.25.a}$$

$$Y = R_L. \tag{2.25.b}$$

Then, there exists a matrix product $\Phi \in \mathbf{R}^{2N\times 2N}$:

$$\Phi = \prod_{j=1}^{\overleftarrow{N-1}} \prod_{i=j}^{\overleftarrow{N-1}} \Phi_{ij}, \tag{2.26.a}$$

of embedded plane hyperbolic rotations :

$$\Phi_{ij} = \begin{bmatrix} I_i & & & & \\ & cosh(\alpha_{ij}) & & sinh(\alpha_{ij}) & \\ & & I_{N-j-1} & & \\ & sinh(\alpha_{ij}) & & cosh(\alpha_{ij}) & \\ & & & & I_{N+j-i-1} \end{bmatrix}, \tag{2.26.b}$$

such that the Cholesky factor L^t of the matrix A satisfies :

$$\begin{bmatrix} L^t \\ O \end{bmatrix} = \Phi \begin{bmatrix} U^t \\ Y^t \end{bmatrix}. \tag{2.27}$$

Proof :

We refer to Appendix B for the proof.

□

The hyperbolic rotations Φ_{kk}, $\Phi_{k+1,k}$, ..., $\Phi_{N-1,k}$ in the planes $(k+1,N+1)$, $(k+2,N+2)$, ..., $(N, 2N-k)$, respectively, eliminate the elements on the $(N-k+1)st$ diagonal of the matrix :

$$\prod_{j=1}^{\overleftarrow{k-1}} \prod_{i=j}^{\overleftarrow{N-1}} \Phi_{ij} \begin{bmatrix} U^t \\ Y^t \end{bmatrix}.$$

The matrix Φ is J-orthogonal, i.e. :

$$\Phi^t J \Phi = \Phi J \Phi^t = J, \tag{2.28}$$

with :

$$J = I_N \oplus -I_N. \tag{2.29}$$

The product form in (2.26.a) implies a *diagonal*-recursive form for the computation of the Cholesky factor of the matrix A :

$$U_0 = U, \quad Y_0 = Y$$

$$\begin{bmatrix} U_j^t \\ Y_j^t \end{bmatrix} = \Phi_j \begin{bmatrix} U_{j-1}^t \\ Y_{j-1}^t \end{bmatrix}, \quad j = 1,...,N-1 \tag{2.30}$$

$$U_{N-1} = L, \quad Y_{N-1} = 0,$$

where

$$\Phi_j = \prod_{i=j}^{\overleftarrow{N-1}} \Phi_{ij}. \tag{2.31}$$

From Theorem 2.2 the factor L^t is obtained as required in (2.3.a). It remains to show how the factor $X^{-t} = L^{-1}$, is obtained for use in (2.3.b). For this purpose we need the following corollary as it follows from Theorem 2.2.

Corollary 2.1 :

The J-orthogonal matrix Φ in (2.26.a) is of the global form :

$$\Phi = \begin{bmatrix} L^{-1} & 0 \\ 0 & R^{-1} \end{bmatrix} \begin{bmatrix} U & -Y \\ -Y^t & U^t \end{bmatrix}, \tag{2.32}$$

where

$$A = LL^t = RR^t \tag{2.33}$$

(L is lower triangular and R is upper triangular).

Proof :

Let Φ be partitioned in 4 $N \times N$ sub matrices :

$$\Phi = \begin{bmatrix} \Phi_{11} & \Phi_{12} \\ \Phi_{21} & \Phi_{22} \end{bmatrix}.$$

Then, it follows from (2.27) that :

$$\Phi_{11} U^t + \Phi_{12} Y^t = L^t \tag{2.34}$$

and

$$\Phi_{21} U^t + \Phi_{22} Y^t = 0. \tag{2.35}$$

With :

$$S = U^{-1} Y = Y U^{-1}. \tag{2.36}$$

(as follows directly from (2.25.a) and (2.25.b)), Equation (2.35) becomes :

$$\Phi_{21} = -\Phi_{22} S^t. \tag{2.37}$$

Since Φ is J-orthogonal, it follows that :

$$\Phi_{12} = \Phi_{11} \Phi_{21}^t \Phi_{22}^{-t} = -\Phi_{11} S. \tag{2.38}$$

Substituting this in (2.34) gives :

$$\Phi_{11} U^{-1} (U U^t - Y Y^t) U^{-t} = L^t U^{-t},$$

with :

$$U U^t - Y Y^t = A = L L^t, \tag{2.39}$$

as follows from (2.25.a) and (2.25.b). Thus,

$$\Phi_{11} = L^{-1} U \tag{2.40}$$

and, consequently :

$$\Phi_{12} = -L^{-1}Y. \tag{2.41}$$

From the J-orthogonality of Φ it follows that :

$$\Phi_{22}\Phi_{22}^t - \Phi_{21}\Phi_{21}^t = I_N.$$

Using (2.35) and (2.34), this becomes :

$$(\Phi_{22}U^{-t})A\,(\Phi_{22}U^{-t})^t = I_N. \tag{2.42}$$

But, from (2.26.a) and (2.26.b) it follows that Φ_{22} is upper triangular and hence, $\Phi_{22}U^{-t}$ is upper triangular. From the uniqueness of the *upper-lower* triangular factorization of a square matrix it follows that :

$$\Phi_{22}U^{-t} = R^{-1}. \tag{2.43}$$

□

From the global form of the matrix Φ it immediately follows that the factor L^{-1} is found from :

$$\begin{bmatrix} L^{-1} \\ R^{-1} \end{bmatrix} = \Phi \begin{bmatrix} I_N \\ I_N \end{bmatrix}. \tag{2.44}$$

Remark : *Using (2.25.a) and (2.25.b), a simple calculation shows that the upper triangular factor R is obtained from :*

$$\begin{bmatrix} O \\ R^t \end{bmatrix} = \Phi \begin{bmatrix} Y \\ U \end{bmatrix},$$

where Φ is the same matrix as in (2.27).

Feed forward direct methods with Schur-Cholesky factorization

In the preceding it was shown how to obtain the Cholesky factor L^t and its inverse L^{-1}, as required in (2.3.a) and (2.3.b). Next it will be shown how either of the possible factorizations in (2.13) and (2.15) are combined with the generalized Schur algorithm,

such that a feed forward algorithm is obtained.

The first row of $[U^t \mid I_N]$ (U is as given in (2.25.a) is the first row of the matrix $[L^t \mid L^{-1}]$ (L is the Cholesky factor of the matrix A). Application of $\Theta_1(m)$ to this row and the vector $[-\mathbf{b}^t \mid 0^t]$ eliminates the first element of $-\mathbf{b}^t$ for a proper choice of the angle α_1. Next, the rotations Φ_{11}, Φ_{21}, ..., $\Phi_{N-1,1}$ are applied to the matrix $[U\ Y]^t$. The result being that the *Nth* lower subdiagonal of this matrix is eliminated and the second row of the matrix :

$$\prod_{i=1}^{\overleftarrow{N-1}} \Phi_{i1} \begin{bmatrix} U^t & I_N \\ Y^t & I_N \end{bmatrix}$$

is equal to the second row of $[L^t \mid L^{-1}]$. Now a rotation $\Theta_2(m)$ is applied to this row and the vector that resulted from the application of $\Theta_1(m)$ to $[-\mathbf{b}^t \mid 0^t]$. This procedure is repeated until all elements of the vector $-\mathbf{b}^t$ are eliminated. With the following embedding of the matrices $\Theta_i(m)$ in (2.4.b) :

$$[\Theta_i(m)] = \begin{bmatrix} I_{i-1} & & & \\ & cos\,(m^{1/2}\alpha_i) & & -m^{1/2} sin\,(m^{1/2}\alpha_i) \\ & & I_{2N-i} & \\ & m^{-1/2} sin\,(m^{1/2}\alpha_i) & & cos\,(m^{1/2}\alpha_i) \end{bmatrix}, \tag{2.45}$$

we get the following feed forward computational scheme for $A\mathbf{x} = \mathbf{b}$:

$$[\Theta_N(m)]([\Theta_{N-1}(m)](\Phi_{N-1,N-1} \oplus 1)) \cdots ([\Theta_2(m)] \overleftarrow{\prod_{i=2}^{N-1}} (\Phi_{i2} \oplus 1))([\Theta_1(m)] \overleftarrow{\prod_{i=1}^{N-1}} (\Phi_{i1} \oplus 1))$$

$$\begin{bmatrix} U^t & I_N & 0 \\ Y^t & I_N & 0 \\ -\mathbf{b}^t & 0^t & 1 \end{bmatrix} = \begin{bmatrix} R(m) & * & * \\ O & * & * \\ 0^t & k(m)\mathbf{x}^t & k(m) \end{bmatrix} \tag{2.46}$$

($m \in \{0,1,-1\}$). Note that, allowing $m=-1$ is constrained to the condition $LL^t - \mathbf{b}\mathbf{b}^t > 0$. In this case (2.46) expresses the Cholesky factorization of the positive definite matrix $A - \mathbf{b}\mathbf{b}^t$. Indeed, since the product of hyperbolic rotations in (2.46) is orthogonal with respect to the signature matrix $\bar{J} = (I_N \oplus -I_{N+1})$, we deduce that $A - \mathbf{b}\mathbf{b}^t = R^t(-1)R(-1)$. And since the Cholesky factorization of a symmetric positive definite matrix is unique it follows that $R^t(-1)$ is the Cholesky factor of the matrix $A - \mathbf{b}\mathbf{b}^t$. A systolic processor array for the execution of the above scheme of rotations is given in Figure 2.2 for N=4 and $m=-1$.

The processor elements on the first diagonal of the lower 3×3 triangular part that process the entries of the matrix A, eliminate the *Nth* lower subdiagonal of the matrix $[U\ Y\ -\mathbf{b}]^t$. The processor elements in the *ith* ($i>1$) diagonal of this lower 3x3 triangular part eliminate the $(N-i+1)st$ lower subdiagonal of the matrix that results after $i-1$ steps in (2.46). The processor element at the top of the *ith* column of this lower 3x3 triangular part produces the $(i+1)th$ row of L^t and L^{-1}. These rows are used in the upper row of 4 rotors, which compute $k(-1)\mathbf{x}^t$ and $k(-1)$.

Lemma 2.1 :

Let the following be given. $A \in \mathbb{R}^{N\times N}$ *is a symmetric positive definite matrix with its diagonal entries equal to* 1. $\mathbf{b} \in \mathbb{R}^N$ *is such that* $A - \mathbf{b}\mathbf{b}^t > 0$. $B \in \mathbb{R}^{(N+1)\times(N+1)}$ *is the positive definite matrix :*

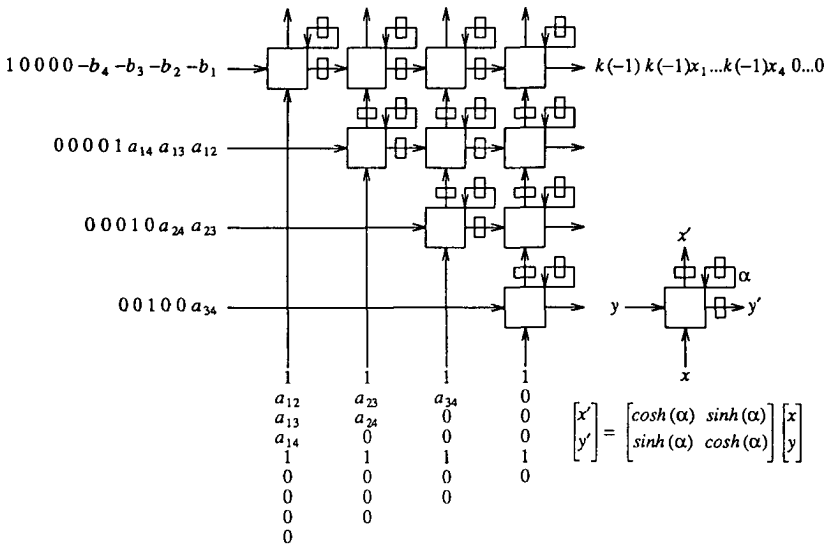


Figure 2.2. Systolic array for the feed forward Cholesky method.

$$B = \begin{bmatrix} 1 & -\mathbf{b}^t \\ -\mathbf{b} & A \end{bmatrix}. \tag{2.47}$$

U_B^t is the upper triangular part of the matrix B :

$$U_B^t = \begin{bmatrix} 1 & -\mathbf{b}^t \\ 0 & U^t \end{bmatrix}, \tag{2.48.a}$$

and Y_B^t is the strictly upper triangular part of the matrix B :

$$Y_B^t = \begin{bmatrix} 0 & -\mathbf{b}^t \\ 0 & Y^t \end{bmatrix}, \tag{2.48.b}$$

where U^t and Y^t are the upper and strictly upper triangular part, respectively, of the matrix A. Let Φ be the sequence of hyperbolic rotations as in (2.26.a)-(2.26.b), with N substituted by N+1, that obtains the Cholesky factor of the matrix B by premultiplication of $[U_B \ \ Y_B]^t$ with Φ. Then, the solution $\mathbf{x}$ of $A\mathbf{x} = \mathbf{b}$ satisfies the equation :

$$\Phi \left[\begin{array}{ccc} 1 & -\mathbf{b}^t & \\ 0 & U^t & I_{N+1} \\ \hline 0 & -\mathbf{b}^t & \\ 0 & Y^t & I_{N+1} \end{array}\right] = \left[\begin{array}{ccc} R(-1) & & * \\ \hline 0^t & & k(-1) \quad k(-1)\mathbf{x}^t \\ O & & * \end{array}\right]. \tag{2.49}$$

Proof :

The proof follows immediately by noticing that the arrangement of the rows in $[U_B \; Y_B]^t$ and the choice of the sequence of rotations in Φ gives rise to the same angles and accumulation of product terms of the product sequence in (2.46).

□

Contractivity

The use of hyperbolic rotations deserves special care. When applying a hyperbolic plane rotation to a vector $[x_1 \; x_2]$:

$$[x_1 \; x_2] \begin{bmatrix} cosh(\alpha) & sinh(\alpha) \\ sinh(\alpha) & cosh(\alpha) \end{bmatrix} = [y_1 \; y_2],$$

the square of the length of the vector $[x_1 \; x_2]$ is preserved when measured with respect to the metric $\Sigma = 1 \oplus -1$. I.e., $x_1^2 - x_2^2 = y_1^2 - y_2^2$. However, the elements y_1 and y_2 may be very large in this equation, even when x_1 and x_2 happen to be small. Such behavior leads to numerical problems. Therefore, it is important that we guarantee in the generalized Schur algorithm that excessive element growth does not occur. This is indeed the case, as stated by the following theorem.

Theorem 2.3 :

Let $U_0 = U_B$ and $Y_0 = Y_B$ in recursion (2.30). Then, since the absolute value of the elements of U_B and Y_B is bounded by 1 in (2.48.a) and (2.48.b), respectively, the

absolute value of the elements of U_j and Y_j in (2.30) is also bounded by 1.

Proof :

We refer to Appendix C for the proof.

□

Referring back to Theorem 2.1 in Section 2.1, we observe that the recursion stated there is contained in the recursion on U'_B and Y'_B in (2.48.a) and (2.48.b). Hence, the proof of Theorem 2.1 follows directly from Lemma 2.1 and Theorem 2.3.

Economized Schur-Cholesky factorization

In case the coefficient matrix is a Toeplitz matrix, e.g., $A = [a_{ij} \mid a_{ij} = a_{i+1,j+1}] \in \mathbf{R}^{N\times N}$, or a close to Toeplitz, symmetric positive definite matrix, the amount of computations involved in the generalized Schur algorithm can be drastically reduced as is shown by the results in [Lev1984]. Let the matrix Z be the lower shift matrix :

$$Z = \begin{bmatrix} 0 & & & \\ 1 & \ddots & & \\ & \ddots & \ddots & \\ & & 1 & 0 \end{bmatrix} \tag{2.50}$$

and define the so-called box operator on the matrix A as follows [Kail1979] :

$$\square A := A - ZAZ^t. \tag{2.51}$$

Then, $\square A$ can always be decomposed as follows [Kail1979] :

$$\square A = GSG^t, \tag{2.52}$$

where the signature matrix S is of the form $S = I_p \oplus -I_q$, with $p + q = r \leq N$. The matrix $G^t \in \mathbf{R}^{r\times N}$ is such that there exists an S-orthogonal matrix P, such that :

$$(GP)^t = \left[\begin{array}{c|c|c} & \delta^t & \\ \hline 0 & + & \\ \hline . & | & \\ . & | & \Gamma^t \\ 0 & | & \end{array}\right], \tag{2.53}$$

where $\delta^t = [\delta_1 \ \cdots \ \delta_N]$ and $\Gamma = [\gamma_{ij}] \in \mathbb{R}^{(N-1)\times(r-1)}$.

Remark : *In case the matrix A is a Toeplitz matrix, we have* $S = 1 \oplus -1$ *and* G^t *is the following matrix (assuming that the matrix A is normalized such that* $a_{ii} = 1$*) [Dewi1978] :*

$$G^t = \begin{bmatrix} 1 & a_{12} & \cdots & a_{1N} \\ 0 & a_{12} & \cdots & a_{1,N-1} \end{bmatrix}. \tag{2.54}$$

In [Lev1984] it is explained how the rows of the Cholesky factor L^t of the matrix A are computed through elimination of the columns of the matrix Γ^t in $(GP)^t$, with S-orthogonal rotations. The entries of the first column of Γ^t can be eliminated by shifting Γ^t one column to the left and applying $(p-1)$ circular rotations in the planes (1,2), (1,3), ..., (1,p), such that the first $p-1$ entries of the first column of Γ^t are eliminated. Next, the remaining entries can be eliminated by applying $(r-p+1)$ hyperbolic rotations in the planes (1,p+1), (1,p+2), ..., (1,r). The first row of the Cholesky factor L^t of the matrix A is equal to the modification of the row δ^t after the sequence of rotations. Next, the new matrix $\bar{\Gamma}^t$ can be shifted one column to the left and again such a sequence of appropriate $(p-1)$ circular and $(r-p+1)$ hyperbolic rotations can be applied, such that the elements in the second column of $\bar{\Gamma}^t$ are eliminated. The second row of the Cholesky factor is now equal to the modification of the row vector δ^t, which results after this second sequence of rotations. The shift and rotate procedure is repeated until all columns of Γ^t have been eliminated. The inverse Cholesky factor L^{-1} of the matrix A is found by applying the rotations to the column vector $[1 \underbrace{0 \cdots 0}_{r-2} 1]^t$. The number of operations is reduced to $O(rN^2)$, compared to the $O(N^3)$ operations of the generalized Schur algorithm.

Combining these economized Schur-Cholesky factorizations with the possible factorizations in (2.13) and (2.15) results in efficient solvers as shown in Figure 2.3. The rectangles denote the rotors and the squares marked by "D" denote a unit delay. The entries of the factors L^t and L^{-1} appear at the taps after the delays. An example of such a solver can be found in [Jain1986a] for an application in speech coding. Note that the number of rotors in Figure 2.3 for the elimination of a column of Γ^t is equal to $r-1$, where r may range from 2 to N. The case $r=2$ corresponds to the Toeplitz case and $r=N$ leads to the full complexity of the generalized Schur algorithm.

2.3 NUMERICAL STABILITY OF THE ALGORITHMS

In the numerical stability analysis of the feed forward direct methods of Section 2.2 the central issue is the study of the propagation of roundoff errors due to finite precision. Detailed numerical stability analyses of the LU factorization with linear rotations, the QR factorization with circular rotations and the LL^t factorization with hyperbolic rotations can be found in, for instance, [Golu1983], [Gent1975] and [Bult1981], respectively. Since the methods of Section 2.2 can all be deduced to either of these three types of factorizations, a detailed analysis of the numerical properties of the methods shall not be given here. Instead, an intuitive explanation shall be given.

The result of an operation f applied to a vector $\mathbf{u}$ in finite precision arithmetic will be denoted as $fl(f(\mathbf{u}))$. The relation between the finite precision and exact result is the following :

$$fl(f(\mathbf{u})) = (1+\eta)f(\mathbf{u}), \tag{2.55}$$

where η is bounded by the machine precision. Notice that (2.55) is a generalization of the case where f denotes either a scalar addition, subtraction, multiplication or division, as found in [Golu1983]. Let f_{ij}, $i,j=1,2$ be the functions that determine the ij-entries of a plane rotation Θ from a vector $\mathbf{u}$. Let the matrix $\overline{\Theta}$ denote the matrix that results when the entries are computed in finite precision arithmetic. Then, $\overline{\Theta}$ is represented as follows :

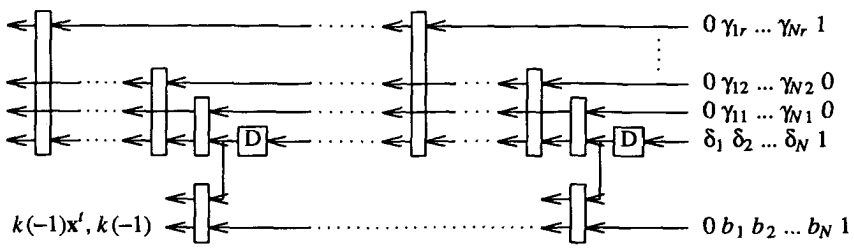


Figure 2.3. Solver with economized Cholesky factorization.

$$\overline{\Theta} = \begin{bmatrix} fl(f_{11}(\mathbf{u})) & fl(f_{12}(\mathbf{u})) \\ fl(f_{21}(\mathbf{u})) & fl(f_{22}(\mathbf{u})) \end{bmatrix} = \begin{bmatrix} (1+\eta_{11})f_{11}(\mathbf{u}) & (1+\eta_{12})f_{12}(\mathbf{u}) \\ (1+\eta_{21})f_{21}(\mathbf{u}) & (1+\eta_{22})f_{22}(\mathbf{u}) \end{bmatrix}, \tag{2.56}$$

where the η_{ij}, $i,j=1,2$, are bounded by the machine precision. Putting $\eta = max(|\eta_{11}|, |\eta_{12}|, |\eta_{21}|, |\eta_{22}|)$ we find the following upper bound for the Frobenius norm of the matrix $\overline{\Theta}$:

$$||\overline{\Theta}||_F \le (1+\eta)\left[\sum_{i=1}^{2}\sum_{j=1}^{2} f_{ij}^2(\mathbf{u})\right]^{1/2} = (1+\eta)||\Theta||_F. \tag{2.57}$$

Now, suppose $\overline{\Theta}_k = [\overline{\theta}_{ij}^{(k)}]$, $k=1,..,n$, are plane rotations computed in finite precision arithmetic. Then, applying $\overline{\Theta}_1$ in finite precision arithmetic to a vector $\mathbf{x} = [x_1 \ x_2]^t$ gives for some error numbers bounded by machine precision, ε_{ij} and γ_i, $i,j=1,2$:

$$fl(\overline{\Theta}_1\mathbf{x}) = \begin{bmatrix} [\overline{\theta}_{11}^{(1)}x_1(1+\varepsilon_{11}) + \overline{\theta}_{12}^{(1)}x_2(1+\varepsilon_{12})](1+\gamma_1) \\ [\overline{\theta}_{21}^{(1)}x_1(1+\varepsilon_{21}) + \overline{\theta}_{22}^{(1)}x_2(1+\varepsilon_{22})](1+\gamma_2) \end{bmatrix}$$

$$= \begin{bmatrix} (1+\gamma_1) & \\ & (1+\gamma_2) \end{bmatrix} \left(\begin{bmatrix} (1+\varepsilon_{11})\overline{\theta}_{11}^{(1)} & (1+\varepsilon_{12})\overline{\theta}_{12}^{(1)} \\ (1+\varepsilon_{21})\overline{\theta}_{21}^{(1)} & (1+\varepsilon_{22})\overline{\theta}_{22}^{(1)} \end{bmatrix} \mathbf{x} \right).$$

Putting $\eta_1 = max(|\varepsilon_{11}|, |\varepsilon_{12}|, |\varepsilon_{21}|, |\varepsilon_{22}|, |\gamma_1|, |\gamma_2|)$ and using the inequalities $||AB||_2 \le ||A||_2||B||_2$, $||A\mathbf{x}||_2 \le ||A||_2||\mathbf{x}||_2$, $||A||_F \ge ||A||_2$, we obtain the following bound for the l_2-norm of $fl(\overline{\Theta}_1\mathbf{x})$:

$$\| fl(\bar{\Theta}_1 \mathbf{x}) \|_2 \leq (1+\eta_1)^2 2^{1/2} \| \bar{\Theta}_1 \|_F \| \mathbf{x} \|_2.$$

Using the result in (2.57), with $\xi_1 = max(\eta, \eta_1)$, we obtain :

$$\| fl(\bar{\Theta}_1 \mathbf{x}) \|_2 \leq (1+\xi_1)^3 2^{1/2} \| \Theta_1 \|_F \| \mathbf{x} \|_2. \tag{2.58}$$

By induction we can proof that :

$$\| fl(\bar{\Theta}_n ... fl(\bar{\Theta}_1 \mathbf{x})...) \|_2 \leq 2^{\frac{n}{2}} (1+\xi_n)^{3n} \left[\prod_{i=1}^{n} \| \Theta_i \|_F \right] \| \mathbf{x} \|_2, \quad n=1,2,...$$

$$\xi_n = max(\eta_n, \xi_{n-1}, \eta), \quad \xi_0 = 0.$$

Indeed, assume that the induction hypothesis :

$$\| fl(\bar{\Theta}_{n-1} ... fl(\bar{\Theta}_1 \mathbf{x})...) \|_2 \leq 2^{\frac{n-1}{2}} (1+\xi_{n-1})^{3(n-1)} \left[\prod_{i=1}^{n-1} \| \Theta_i \|_F \right] \| \mathbf{x} \|_2$$

$$\xi_{n-1} = max(\eta_{n-1}, \xi_{n-2}, \eta)$$

is valid. Then,

$$fl(\bar{\Theta}_n ... fl(\bar{\Theta}_1 \mathbf{x})...) = \begin{bmatrix} (1+\gamma_1) & \\ & (1+\gamma_2) \end{bmatrix} \begin{bmatrix} (1+\varepsilon_{11})\theta_{11}^{(n)} & (1+\varepsilon_{12})\theta_{12}^{(n)} \\ (1+\varepsilon_{21})\theta_{21}^{(n)} & (1+\varepsilon_{22})\theta_{22}^{(n)} \end{bmatrix} fl(\bar{\Theta}_{n-1} ... fl(\bar{\Theta}_1 \mathbf{x})...).$$

Putting $\eta_n = \max(|\varepsilon_{11}|, |\varepsilon_{12}|, |\varepsilon_{21}|, |\varepsilon_{22}|, |\gamma_1|, |\gamma_2|)$ and using the inequalities, we obtain :

$$\| fl(\bar{\Theta}_n ... fl(\bar{\Theta}_1 \mathbf{x})...) \|_2 \leq 2^{1/2} (1+\eta_n)^2 \| \bar{\Theta}_n \|_F \| fl(\bar{\Theta}_{n-1} ... fl(\bar{\Theta}_1 \mathbf{x})...) \|_2$$

$$\leq 2^{\frac{n}{2}} (1+\xi_{n-1})^{3(n-1)} (1+\eta_n)^2 \| \bar{\Theta}_n \|_F \left[\prod_{i=1}^{n-1} \| \Theta_i \|_F \right] \| \mathbf{x} \|_2.$$

Using the result in (2.57), with $\xi_n = max(\eta_n, \xi_{n-1}, \eta)$, we obtain :

$$\| fl(\bar{\Theta}_n ... fl(\bar{\Theta}_1 \mathbf{x})...) \|_2 \leq 2^{\frac{n}{2}} (1+\xi_n)^{3n} \left[\prod_{i=1}^{n} \| \Theta_i \|_F \right] \| \mathbf{x} \|_2$$

$$\leq 2^{\frac{n}{2}} (1+3n\xi_n) \left[\prod_{i=1}^{n} \| \Theta_i \|_F \right] \| \mathbf{x} \|_2 + O(\xi_n^2)$$

$$\leq 2^{\frac{n}{2}} \left[\prod_{i=1}^{n} \| \Theta_i \|_F \right] \| \mathbf{x} \|_2 + e_n,$$

where :

$$e_n \leq 2^{\frac{n}{2}} 3n\xi_n \left[\prod_{i=1}^{n} \| \Theta_i \|_F \right] \| \mathbf{x} \|_2 + O(\xi_n^2).$$

And from the inequality $\| A \|_F \leq N^{1/2} \| A \|_2 = N^{1/2} \lambda(A)$, where $\lambda(A)$ is the spectral radius of the matrix $A \in \mathbf{R}^{N \times N}$, we find :

$$e_n \leq 2^n 3n\xi_n \left[\prod_{i=1}^{n} \lambda(\Theta_i) \right] \| \mathbf{x} \|_2 + O(\xi_n^2). \tag{2.59}$$

This expression shows that the propagation of the roundoff error is amplified by the spectral radius of the rotations Θ_i. In case the Θ_i are orthogonal the spectral radii are equal to 1 and there is no amplification of roundoff errors due to the Θ_i. Hence, the feed forward direct method in (2.23) is numerically robust.

However, in case the Θ_i are either linear or hyperbolic rotations, the spectral radii are, respectively, $[(1 + \frac{1}{2}|\alpha_i|^2) + ((1 + \frac{1}{2}|\alpha_i|^2)^2 - 1)^{1/2}]$ and $(1 + |tanh(\alpha_i)|)(1 - \tanh^2(\alpha_i))^{-1/2}$. In case of linear rotations the $|\alpha_i|$ become large, if the diagonal elements of the matrix A are small relatively to the *off*-diagonal elements. In such cases the spectral radius of Θ_i is approximately equal to $|\alpha_i|^2$. And this already happens for relatively small α_i's, because of the quadratic term $(1 + \frac{1}{2}|\alpha_i|^2)^2$. On the other hand, if the matrix $A = [a_{ij}]$ is positive definite, then the $|\alpha_i| < 1$ due to the fact that $a_{ii} > |a_{ij}|$, and the amplification of roundoff errors is less.

From Section 2.2.3 we know that the hyperbolic rotations can only be used when the matrix B in (2.47) is positive definite. In that case $|tanh(\alpha_i)| < 1$ for all i. From (2.59) the amplification of the roundoff error due to the hyperbolic rotations is given by :

$$\prod_{i=1}^{n}(1 + tanh(\alpha_i))(1 - \tanh^2(\alpha_i))^{-1/2}.$$

From (2.26.a), (2.26.b) and (2.44) it follows that the *ith* diagonal entry of the inverse Cholesky factor L_B^{-1}, of the matrix $B \in \mathbf{R}^{(N+1)\times(N+1)}$ in (2.47) is equal to $\prod_{j=1}^{i-1}(1 - \tanh^2(\alpha_j))^{-1/2}$, for i=2,3,..., and equal to 1, for i=1. Hence, the determinant of B is equal to $\prod_{i=2}^{N+1}\prod_{j=1}^{i-2}(1 - \tanh^2(\alpha_j))$ and it follows that the closer to singularity the matrix is, the closer to ±1 one of the factors $tanh(\alpha_j)$ is. Therefore, the amplification of the round-off error increases as the matrix approaches singularity.

2.4 SEMI-DIRECT NATURE OF THE GENERALIZED SCHUR ALGORITHM

The generalized Schur algorithm of Section 2.2 can be shown to be a semi-direct method for solving positive definite systems of linear equations. (This was shown in [Neho1983] for the Levinson algorithm [Orfa1985]). Hence, we may prematurely abort the recursion in (2.30) and still obtain an approximation for the solution. Let $A \in \mathbf{R}^{N\times N}$ be a symmetric positive definite matrix with its diagonal entries equal to 1. And let $\mathbf{b} \in \mathbf{R}^N$ be such that $A - \mathbf{b}\mathbf{b}^t > 0$, and $\mathbf{p}_i$ is the *ith* row of the inverse of the Cholesky factor L of the matrix A. Then, the computations in (2.30), with $U_0 = U_B$ and $Y_0 = Y_B$, can also be stated in terms of the following recursion.

$x_0 = 0$
for i=0 **to** $N-1$ **do**

compute $\mathbf{p}_{i+1}$ *such that* $\mathbf{p}_{i+1}^t A \mathbf{p}_j = \delta_{i+1,j}$ $j = 1,...,i+1$ (2.60)

$\mathbf{r}_i = A\mathbf{x}_i - \mathbf{b}$ (2.61)

$\alpha_{i+1} = \mathbf{p}_{i+1}^t \mathbf{r}_i$ (2.62)

$\mathbf{x}_{i+1} = \mathbf{x}_i + \alpha_{i+1}\mathbf{p}_{i+1}$ (2.63)

endfor

□

This recursion is very similar to the conjugate gradient method [Golu1983]. The difference being that the vectors $\mathbf{p}_i$ are linear combinations of the Lanczos vectors of the matrix A in case of the conjugate gradient method. Notice that the above recursion can be terminated when $\mathbf{r}_i = 0$. This may happen before $i = N-1$, if the solution $\mathbf{x}$ lies in a subspace of the space spanned by the columns of the inverse of the Cholesky factor of A. For instance, if $\mathbf{x}$ lies in the space spanned by $\mathbf{p}_1$, then the residuals $\mathbf{r}_i$, i=1,...,$N-1$, are zero and the recursion can stop at i=1.

To show that the computations in recursion (2.60)-(2.63) are indeed those of recursion (2.30) with $U_0 = U_B$ and $Y_0 = Y_B$, the following approach is taken. First we show that α_{i+1} in (2.62) is really $tanh(\alpha_{i+1})$ in (2.4.b), for $m = -1$, by showing that recursion (2.61)-(2.63) can be embedded in recursion (2.6.a)-(2.6.b), for $m = -1$. And since we compute the $\mathbf{p}_{i+1}$, by recursion (2.30) it follows that the computations in (2.60)-(2.63) are similar to the computations in recursion (2.30) when $U_0 = U_B$ and $Y_0 = Y_B$. From this we will be able to conclude that recursion (2.30) expresses a semi-direct method for solving symmetric positive definite systems of linear equations.

Embedding of residual Equation (2.61) in a vector-matrix product gives :

$$[1 \mid \mathbf{x}_i^t] \begin{bmatrix} 1 & -\mathbf{b}^t \\ -\mathbf{b} & A \end{bmatrix} = [(1 - \mathbf{x}_i^t\mathbf{b}) \mid \mathbf{r}_i^t] \tag{2.64}$$

($B = \begin{bmatrix} 1 & -\mathbf{b}^t \\ -\mathbf{b} & A \end{bmatrix} = L_B L_B^t$, where L_B is the Cholesky factor of the matrix B). At the end of

recursion (2.60)-(2.63) we have :

$$[1 \mid \mathbf{x}^t]B = [(1 - \mathbf{x}^t\mathbf{b}) \mid 0...0] = [(1 - ||\mathbf{x}||_A^2) \mid 0...0], \tag{2.65}$$

where

$$||\mathbf{x}||_A^2 = \mathbf{x}^t A\,\mathbf{x}. \tag{2.66}$$

Since the matrix B is positive definite, it follows that $(1 - ||\mathbf{x}||_A^2) > 0$. Hence, dividing both sides of (2.65) by $(1 - ||\mathbf{x}||_A^2)^{1/2}$ is allowed and gives :

$$[(1 - ||\mathbf{x}||_A^2)^{-1/2} \mid \bar{\mathbf{x}}^t]B = [(1 - ||\mathbf{x}||_A^2)^{1/2} \mid 0...0], \tag{2.67.a}$$

and

$$\bar{\mathbf{x}}^t = (1 - ||\mathbf{x}||_A^2)^{-1/2}\mathbf{x}^t. \tag{2.67.b}$$

Equation (2.67.a) can be solved using a Levinson recursion on B [Orfa1985] as follows.

Normalize (2.64) through division by $(1 - \mathbf{x}_i^t\mathbf{b})^{1/2}$:

$$[(1 - \mathbf{x}_i^t\mathbf{b})^{-1/2} \mid \bar{\mathbf{x}}_i^t]B = [(1 - \mathbf{x}_i^t\mathbf{b})^{1/2} \mid \bar{\mathbf{r}}_i^t], \tag{2.68.a}$$

where

$$\bar{\mathbf{x}}_i^t = (1 - \mathbf{x}_i^t\mathbf{b})^{-1/2}\mathbf{x}_i^t \tag{2.68.b}$$

and

$$\bar{\mathbf{r}}_i^t = (1 - \mathbf{x}_i^t\mathbf{b})^{-1/2}\mathbf{r}_i^t. \tag{2.68.c}$$

For the moment assume that

$$[(1 - \mathbf{x}_i^t\mathbf{b})^{-1/2} \mid \bar{\mathbf{x}}_i^t] = [\mathbf{y}_i^t \mid \underbrace{0 \cdots 0}_{N-i}], \tag{2.69.a}$$

with

$$\mathbf{y}_i^t = [y_{i,0}\ \ y_{i,1}\ \ \cdots\ \ y_{i,i}] \tag{2.69.b}$$

and

$$y_{i,0} = (1 - \mathbf{x}_i^t\mathbf{b})^{-1/2}. \tag{2.69.c}$$

Also assume that

$$[\mathbf{y}_i^t \mid \underbrace{0 \cdots 0}_{N-i}]B = [(1 - \mathbf{x}_i^t\mathbf{b})^{1/2} \mid \underbrace{0 \cdots 0}_{i}\, \bar{r}_i(1) \cdots \bar{r}_i(N-i)]. \tag{2.70}$$

It will follow next, by induction, that both assumptions are true.

Surely (2.70) holds for $i=0$ (i.e., putting $x_0=0$). Now, let $\bar{\mathbf{p}}_{i+1}$ denote the $(i+1)st$ row of L^{-1}, without the zero elements of $\mathbf{p}_{i+1}$. Then, $\mathbf{y}_i$ is updated by the rule :

$$\mathbf{y}_{i+1}^t = ([\mathbf{y}_i^t \mid 0] + \rho_{i+1}[0 \mid \bar{\mathbf{p}}_{i+1}])(1 - \rho_{i+1}^2)^{-1/2}. \tag{2.71}$$

The value of ρ_{i+1} then follows from the residual equation :

$$\left[[\mathbf{y}_i^t\, 0 \mid \underbrace{0 \cdots 0}_{N-i-1}] + \rho_{i+1}[0\, \bar{\mathbf{p}}_{i+1} \mid \underbrace{0 \cdots 0}_{N-i-1}]\right]B =$$

$$[(1 - \mathbf{x}_i^t\mathbf{b})^{1/2} \mid \underbrace{0 \cdots 0}_{i}\, \bar{r}_i(1) \cdots \bar{r}_i(N-i)] +$$

$$\rho_{i+1}[\gamma_{i+1}\, \underbrace{0 \cdots 0}_{i}\, l_{i+1}(1) \cdots l_{i+1}(N-i)] =$$

$$[(1 - \mathbf{x}_{i+1}^t\mathbf{b})^{1/2} \mid \underbrace{0 \cdots 0}_{i+1}\, \bar{r}_{i+1}(1) \cdots \bar{r}_{i+1}(N-i-1)]. \tag{2.72}$$

The elements $l_{i+1}(k)$ are the non-zero elements of the $(i+1)st$ row of L^t. From (2.72) it then follows that :

$$\rho_{i+1} = -\frac{\bar{r}_i(1)}{l_{i+1}(1)}. \tag{2.73}$$

And from the positivity of the matrix B it follows that

$$|\rho_{i+1}| < 1. \tag{2.74}$$

A flow graph representation of the Levinson recursion in (2.72) is depicted in Figure 2.4, for an initial guess of $x_0 = 0$.

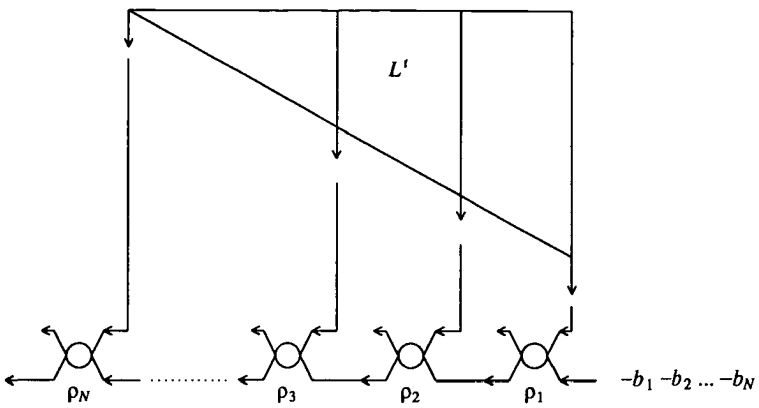
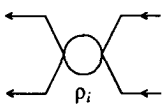


Figure 2.4. Graphical representation of the Levinson recursion in (2.72).

The symbol



denotes the 2×2 hyperbolic rotation with $\rho_i = tanh(\alpha_i)$ in (1.6) ($m = -1$). This sequence of hyperbolic rotations is exactly the same as the one in recursion (2.6.a)-(2.6.b). But since (2.64) is an embedding of the residual Equation (2.61) and (2.70) is equal to (2.64), it must be that the update rule for $\mathbf{x}_{i+1}$ in (2.63) is equivalent to (2.71), with $\rho_{i+1} = \alpha_{i+1}$, $\mathbf{x}_{i+1}^t = [y_{i,1} \cdots y_{i,i} \mid \underbrace{0 \cdots 0}_{N-i}] + \rho_{i+1}[\bar{\mathbf{p}}_{i+1} \mid \underbrace{0 \cdots 0}_{N-i-1}]$, $\mathbf{x}_i^t = [y_{i,1} \cdots y_{i,i} \mid \underbrace{0 \cdots 0}_{N-i}]$, $\mathbf{p}_{i+1} = [\bar{\mathbf{p}}_{i+1} \mid \underbrace{0 \cdots 0}_{N-i-1}]$ and $\mathbf{r}_i^t = [\underbrace{0 \cdots 0}_{i+1} \bar{r}_{i+1}(1) \cdots \bar{r}_{i+1}(N-i-1)]$. And because recursion (2.30) combines the computations of recursion (2.6.a)-(2.6.b) and the computations of the rows of L and L^{-1}, for $U_0 = U_B$ and $Y_0 = Y_B$, it follows that the computations in recursion (2.30) are equivalent to those of recursion (2.60)-(2.63). And we conclude that recursion (2.30) is in fact a semi-direct method.

As was noticed before, the recursion in (2.60)-(2.63) expresses the conjugate gradient method, when the $\mathbf{p}_i$ are chosen to be linear combinations of the Lanczos vectors of the

matrix A. If the Lanczos procedure is prematurely terminated the resulting tridiagonal matrix T has extreme eigenvalues which approximate the extreme eigenvalues of the matrix A and the obtained solution vector is an approximation to the exact solution vector [Golu1983].

In contrast, premature termination of the generalized Schur recursion results in a maximum entropy extension approximation of the inverse of the coefficient matrix A (see [Dewi1981] for the Toeplitz case and [Dewi1987] for the general case). In fact, the approximative inverse is banded, such that its inverse coincides with the matrix A within the band [Dym1981]. Hence, early termination of (2.60)-(2.63) results in a solution of a system of linear equations, which coincides in the band with the original set of linear equations.

2.5 GENERALIZATIONS OF THE FEED FORWARD METHODS

The algorithms in Section 2.2 can be straight forwardly generalized to handle more general matrix arithmetic. Straightforward generalizations can be given for the following computations :

1. computing an unknown matrix X from the matrix equation $AX = B$, where $A \in \mathbb{R}^{N\times N}, X \in \mathbb{R}^{N\times q}$ and $B \in \mathbb{R}^{N\times q}$;
2. computing an inner product/accumulation $\mathbf{b}^t\mathbf{c} + d$;
3. computing a vector-matrix product/accumulation $C\mathbf{b} + \mathbf{d}$;
4. computing a matrix-matrix product/accumulation $CA^{-1}B + D$.

We shall consider two cases. The first case presents a generalization based on an LU or LQ factorization of the matrix A. The second case presents a generalization based on an LL^t factorization of the matrix A.

CASE I

Suppose we have a non singular matrix $A \in \mathbf{R}^{N\times N}$ and matrices $B \in \mathbf{R}^{N\times q}$, $C \in \mathbf{R}^{r\times N}$ and $D \in \mathbf{R}^{r\times q}$. Let $\mathbf{b}_1$, ..., $\mathbf{b}_q$ and $\mathbf{d}_1$, ..., $\mathbf{d}_q$ denote the columns of the matrices B and D, respectively. Let $\mathbf{m}$ be a vector $\mathbf{m} = [m_1 \cdots m_q]^t$, $m_i \in \{0,1,-1\}$, $i=1,...,q$. We show that sequences of rotations as in (2.18) and (2.22) can be applied to the matrices :

$$W_1 = \begin{bmatrix} A^t & C^t \\ -\mathbf{b}_1^t & \mathbf{d}_1^t \end{bmatrix}, \ldots, W_q = \begin{bmatrix} A^t & C^t \\ -\mathbf{b}_q^t & \mathbf{d}_q^t \end{bmatrix},$$

such that we can combine the results into a matrix $E_{\mathbf{m}}$:

$$E_{\mathbf{m}} = diag\,(k_1(m_1) \cdots k_q(m_q))(B^t A^{-t} C^t + D^t). \tag{2.75}$$

The sequences of rotations in (2.18) and (2.22) can be re-ordered to $\Theta(m)\begin{bmatrix} X^{-1} & 0 \\ 0^t & 1 \end{bmatrix}$, where $\Theta(m) = \overleftarrow{\prod_{i=1}^{N}} \Theta_i(m)$ (see (2.4.a)-(2.4.b)) and

$$\begin{bmatrix} X^{-1} & 0 \\ 0^t & 1 \end{bmatrix} = \overleftarrow{\prod_{j=1}^{N-1}} \overleftarrow{\prod_{i=j}^{N-1}} U_{ij},$$

or

$$\begin{bmatrix} X^{-1} & 0 \\ 0^t & 1 \end{bmatrix} = \overleftarrow{\prod_{j=1}^{N-1}} \overleftarrow{\prod_{i=j}^{N-1}} Q_{ij},$$

respectively. U_{ij} and Q_{ij} are as given in (2.17) and (2.21), respectively. Let us denote by $\Theta(m_i)$ the matrix $\Theta(m)$ in the product $\Theta(m)\begin{bmatrix} X^{-1} & 0 \\ 0^t & 1 \end{bmatrix}$, when applied to W_i :

$$\Theta(m_i)\begin{bmatrix} X^{-1} & 0 \\ 0^t & 1 \end{bmatrix}\begin{bmatrix} A^t & C^t \\ -\mathbf{b}_i^t & \mathbf{d}_i^t \end{bmatrix} = \begin{bmatrix} R_i & * \\ 0^t & \mathbf{e}_i^t \end{bmatrix},$$

where $[R_i^t \mid 0]^t$, with $R_i \in \mathbf{R}^{N\times N}$, is the upper triangular factor of $[A \mid -\mathbf{b}_i]^t$. Writing the $(N+1)\times(N+1)$ matrix $\Theta(m_i)$ as :

$$\Theta(m_i) = \begin{bmatrix} \Theta_{11}(m_i) & \theta_{12}(m_i) \\ \theta_{21}^t(m_i) & k_i(m_i) \end{bmatrix}, \quad \Theta_{11}(m_i) \in \mathbf{R}^{N \times N},$$

we find :

$$\theta_{21}^t(m_i) = k_i(m_i)\mathbf{b}_i^t A^{-t} X,$$

and

$$\mathbf{e}_i^t = \theta_{21}^t(m_i)X^{-1}C^t + k_i(m_i)\mathbf{d}_i^t.$$

Hence, it follows that :

$$\mathbf{e}_i^t = k_i(m_i)(\mathbf{b}_i^t A^{-t} C^t + \mathbf{d}_i^t).$$

Thus, for $i = 1,...,q$ we can combine the results $\mathbf{e}_1^t$, ..., $\mathbf{e}_q^t$ as the rows of the matrix $E_{\mathbf{m}}$ in (2.75).

CASE II

A similar result can be obtained for a *Schur-Cholesky* factorization of the matrix A (in case A is symmetric positive definite). Let the following be given. $A \in \mathbf{R}^{N \times N}$ is a positive definite matrix, $B \in \mathbf{R}^{N \times q}$ is a matrix with columns $\mathbf{b}_1$, ..., $\mathbf{b}_q$, such that $A - \mathbf{b}_i\mathbf{b}_i^t > 0$, $i = 1,...,q$, $\overline{C} \in \mathbf{R}^{r \times 2N}$ is the following augmented matrix : $\overline{C} = [C \mid C]$ with $C \in \mathbf{R}^{r \times N}$, and $D \in \mathbf{R}^{r \times q}$ is a matrix with columns $\mathbf{d}_1$, ..., $\mathbf{d}_q$. Moreover, let $G = [U \; Y]$, where U and Y are the lower and strictly lower triangular part of the matrix A, respectively. We show that sequences of rotations as in (2.46) can be applied to the matrices :

$$W_1 = \begin{bmatrix} G^t & \overline{C}^t \\ -\mathbf{b}_1^t & \mathbf{d}_1^t \end{bmatrix}, \ldots, W_q = \begin{bmatrix} G^t & \overline{C}^t \\ -\mathbf{b}_q^t & \mathbf{d}_q^t \end{bmatrix},$$

such that we can combine the results into a matrix $E_{\mathbf{m}}$ as given in (2.75).

The sequence of rotations in (2.46) can be reordered to $[\Theta(m)]\begin{bmatrix} \Phi & 0 \\ 0^t & 1 \end{bmatrix}$, where

$[\Theta(m)] = \overleftarrow{\prod_{i=1}^{N}}[\Theta_i(m)]$ (see (2.45)) and Φ is as given in (2.26.a)-(2.26.b). Let us denote by $[\Theta(m_i)]$ the matrix $[\Theta(m)]$ in the product $[\Theta(m)]\begin{bmatrix}\Phi & 0\\ 0^t & 1\end{bmatrix}$, when applied to W_i :

$$[\Theta(m_i)]\begin{bmatrix}\Phi & 0\\ 0 & 1\end{bmatrix}\begin{bmatrix}G^t & \bar{C}^t\\ -\mathbf{b}_i^t & \mathbf{d}_i^t\end{bmatrix} = \begin{bmatrix}\bar{R}_i & *\\ 0^t & \mathbf{e}_i^t\end{bmatrix},$$

where $\bar{R}_i = [R_i \mid O]^t \in \mathbf{R}^{2N\times N}$ and $R_i \in \mathbf{R}^{N\times N}$ being the Cholesky factor of the matrix $A - \mathbf{b}_i\mathbf{b}_i^t$. Writing the $(2N+1)\times(2N+1)$ matrix $[\Theta(m_i)]$ as :

$$[\Theta(m_i)] = \begin{bmatrix}\Theta_{11}(m_i) & O & \theta_{12}(m_i)\\ O & I_N & O\\ \theta_{21}^t(m_i) & O & k_i(m_i)\end{bmatrix},\ \Theta_{11}(m_i) \in \mathbf{R}^{N\times N},$$

we find :

$$\theta_{21}^t(m_i)L^t = k_i(m_i)\mathbf{b}_i^t$$

and because :

$$\Phi = \begin{bmatrix}L^{-1} & O\\ O & R^{-1}\end{bmatrix}\begin{bmatrix}U & -Y\\ -Y^t & U^t\end{bmatrix}$$

(see Corollary 2.1) we get :

$$\mathbf{e}_i^t = \theta_{21}^t(m_i)L^{-1}C^t + k_i(m_i)\mathbf{d}_i^t.$$

Hence, it follows that :

$$\mathbf{e}_i^t = k_i(m_i)(\mathbf{b}_i^tA^{-1}C^t + \mathbf{d}_i).$$

Thus, for $i = 1,...,q$, we can combine the results $\mathbf{e}_1^t, ..., \mathbf{e}_q^t$ as the rows of the matrix $E_{\mathbf{m}}$ in (2.75). Hence, with the appropriate choice of the matrices A, B, C and D we can realize each of the four above listed computations.

An example of a systolic implementation for the computation of (2.75), for $\mathbf{m} = 0$, is shown in Figure 2.5.

The rows of processing elements in the rectangular part transmit the data coming from the triangular part unchanged, along the vertical interconnections to their upper neighbor. This has been indicated by the arrows across a processing element.

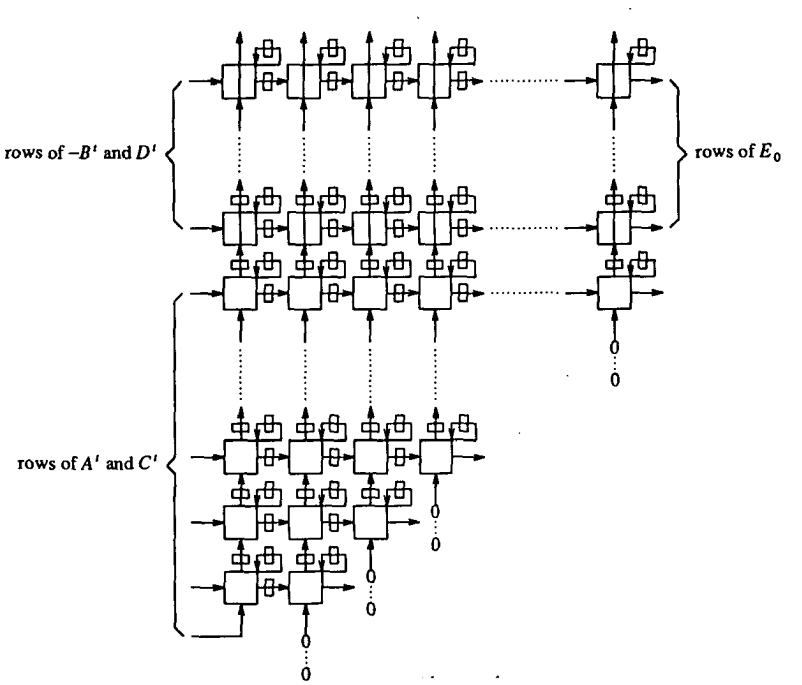


Figure 2.5. Systolic array for expressions $(CA^{-1}B + D)$.

3. MAPPING REGULAR RECURRENT ALGORITHMS TO FIXED SIZE SYSTOLIC ARRAYS

In this chapter the LPGP and the LSGP partitioning strategies are presented. The presentation will be restricted to the partitioning of 2-D systolic arrays. In Section 3.1 the tessellation of a systolic array into tiles of processor elements is characterized. This tessellation is common to the LPGP and the LSGP partitioning strategy. In Section 3.2 the LPGP partitioning strategy is presented. A full size systolic array is first tessellated after which a dependency graph for the tessellated FSA is derived. From this dependency graph the order is determined in which the computations of different tiles must be executed on a reduced size array. A model for the LPGP partitioned FSA is presented, which consists of the RSA, buffers to store intermediate data, a controller which issues the read and write addresses for the buffers and an interconnection network which routes the outputs of the RSA to the inputs. It is shown how the addresses are derived from the scheduling of the processor elements in the FSA and the dependency graph of the tessellated FSA.

In Section 3.3 the LSGP partitioning strategy is presented. The existence is proved of clusters which have the shape of either a parallelogram, trapezoid, or triangle and it is shown how the clusters are positioned relatively to each other in the FSA. The clusters and their positioning can be characterized by the tessellation of an FSA, as explained in Section 3.1. From the dependency graph of the tessellated FSA the control is derived for a cluster processor.

3.1 TESSELLATION OF FULL SIZE ARRAYS

A 2-D full size systolic array is regularly tessellated into congruent tiles of equal size. The tessellation is done by repeating a tile across the FSA, without overlapping. The repetition is done only by translations. Tessellation of an FSA is common to the LSGP and the LPGP partitioning strategy. With respect to its local frame of reference, a tile P is characterized by two linearly independent integer vectors $\mathbf{b}_1 = [b_{11}\ b_{21}]^t$ and $\mathbf{b}_2 = [b_{12}\ b_{22}]^t$ and an integer support region F :

$$P = \{\mathbf{x} \mid B\mathbf{n} = \mathbf{x},\ \mathbf{n} \in F\}, \tag{3.1}$$

where $B = [\mathbf{b}_1 \mid \mathbf{b}_2]$ and $F \subset \mathbb{Z}^2$ has finite cardinality. The integer support region F is such that P is homogeneous (see CONVENTIONS, SYMBOLS AND DEFINITIONS, Definition III) in the space spanned by $\mathbf{b}_1$ and $\mathbf{b}_2$. To each value of $\mathbf{n}$ in F corresponds a processor element in P.

Let us define an integer displacement matrix $U = [\mathbf{u}_1 \mid \mathbf{u}_2]$, where the columns $\mathbf{u}_1$ and $\mathbf{u}_2$ are such that translations of tile P, by linear combinations of these two vectors, uniquely assigns a processor element per value of $\mathbf{n}$ to a tile $P_{\mathbf{i}}$:

$$P_{\mathbf{i}} = \{\mathbf{x} \mid B\mathbf{n} + U\mathbf{i} + \mathbf{r} = \mathbf{x},\ \mathbf{n} \in F\}, \tag{3.2}$$

for some translation vector $\mathbf{i}$ and offset vector $\mathbf{r}$. For a given size of the FSA we have $\mathbf{i} \in H$, where H is the domain for the repetition of tile P, such that each processor element of the FSA is uniquely assigned to a tile. The offset vector $\mathbf{r}$ determines the point at which the repetition starts. Thus, we can characterize the tessellation of an FSA by the following 5–*tuple* T of parameters :

$$T = \{U, H, B, F, \mathbf{r}\}. \tag{3.3}$$

At this point we would like to remark that the choice of the matrix U is not unique, given B, F and $\mathbf{r}$. Similarly, the choice of $\mathbf{r}$ is not unique, given U, B and F. For examples of different tessellations of a 2-D FSA we refer to Figure 3.1.

In order to denote which processor elements belong to a tile, dotted squares have been drawn around the processor elements. In the sequel we shall use the convention that delays at an output of a processor element belong to the same tile as the processor element itself. In Figure 3.1(a) the following choices for the parameters of the tessellation T have been made :

$$\mathbf{r} = 0,$$

$$B = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

$$\mathrm{F} = \left\{ \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \begin{bmatrix} 1 \\ 1 \end{bmatrix} \right\},$$

$$U = \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix},$$

$$\mathrm{H} = \left\{ \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 2 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \begin{bmatrix} 0 \\ 2 \end{bmatrix}, \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \begin{bmatrix} 2 \\ 1 \end{bmatrix}, \begin{bmatrix} 1 \\ 2 \end{bmatrix}, \begin{bmatrix} 2 \\ 2 \end{bmatrix} \right\}.$$

In Figure 3.1(b) these are chosen to be :

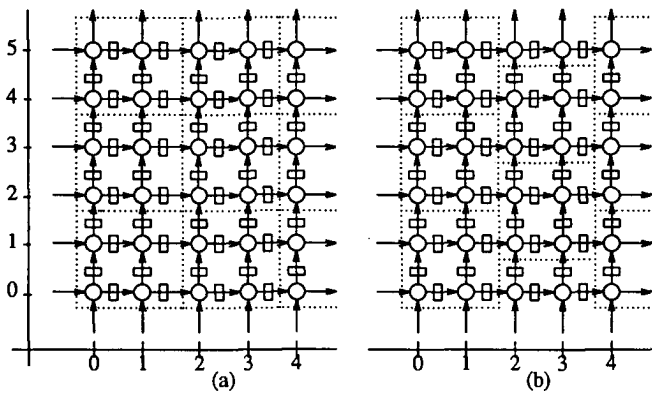


Figure 3.1. Two different tessellations, with square tiles.

$\mathbf{r} = 0$

$$B = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

$$\mathrm{F} = \left\{ \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \begin{bmatrix} 1 \\ 1 \end{bmatrix} \right\},$$

$$U = \begin{bmatrix} 2 & 0 \\ -1 & 2 \end{bmatrix},$$

$$\mathrm{H} = \left\{ \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \begin{bmatrix} 0 \\ 2 \end{bmatrix}, \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \begin{bmatrix} 1 \\ 2 \end{bmatrix}, \begin{bmatrix} 1 \\ 3 \end{bmatrix}, \begin{bmatrix} 2 \\ 1 \end{bmatrix}, \begin{bmatrix} 2 \\ 2 \end{bmatrix}, \begin{bmatrix} 2 \\ 3 \end{bmatrix} \right\}.$$

Notice that for given B, F and U the number of elements in H, i.e., the number of tiles, depends on the vector $\mathbf{r}$. For instance, putting $\mathbf{r} = [2\ 2]^t$ in Figure 3.1(b), gives :

$$\mathrm{H} = \left\{ \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \begin{bmatrix} 0 \\ -1 \end{bmatrix}, \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \begin{bmatrix} 1 \\ 2 \end{bmatrix}, \begin{bmatrix} 1 \\ -1 \end{bmatrix}, \begin{bmatrix} -1 \\ 0 \end{bmatrix}, \begin{bmatrix} -1 \\ 1 \end{bmatrix} \begin{bmatrix} -1 \\ -1 \end{bmatrix}, \begin{bmatrix} -1 \\ -2 \end{bmatrix} \right\}.$$

That is, 11 tiles compared to 9 tiles for $\mathbf{r} = 0$.

3.2 STRATEGY I : LOCAL PARALLEL, GLOBAL PIPELINED PARTITIONING

The different steps that are involved in the LPGP partitioning strategy are the following.

1. Tessellate the FSA such that there are no reverse interconnections across an edge of a tile and such that the number of tiles is minimal. Reverse interconnections across the same edge of a tile are not allowed, because computations of the next tile are scheduled on the RSA only when a processor element has finished all its computations for the currently scheduled tile. Notice that reverse interconnections give rise to a loop in the dependency relations between two tiles.

2. If necessary, extend the FSA with dummy processor elements, such that it can be tessellated in tiles of, say, p processor elements each, for a particular choice of the offset vector $\mathbf{r}$ and the displacement matrix U. A dummy processor element just performs an identity I/O map.

3. Find an admissable order of the tiles. The order is admissible when the dependency constraints between the tiles are not violated and when the order is such that no two or more tiles are assigned to the same position in the order.

4. Identify the processor elements in a tile which communicate with processor elements in other tiles. The interconnection pattern between these processor elements defines the buffered communication from the outputs to the inputs of the RSA. See Figure 1.10. The RSA is defined to have $card(\mathrm{F})$ processor elements and the same interconnections as the FSA, between processor elements.

3.2.1 TESSELLATION OF THE FULL SIZE ARRAY

Depending on the offset vector **r** and the displacement matrix U the tessellation of an FSA may yield tiles with less than *card*(F) processor elements. See for instance Figure 3.1(a). In this figure the tiles which cover the right most column of processor elements have each only 2 processor elements, instead of 4. Since the RSA will consist of 4 processor elements, an extra column of dummy processor elements must be added to the FSA, so that each tile has exactly *card*(F) processor elements. By adding this extra column of processor elements it suffices to have all communication between the RSA and peripherals (like buffers and a host computer) to take place at the boundaries of the RSA.

Since the computations of all tiles are going to be scheduled on the RSA, the functionality of the processor elements of the RSA must include the identity I/O map of a dummy processor element. Moreover, control must be added to select between different functionalities of a processor element. There are different ways to accomplish this, all with their own drawbacks and merits. We shall present two obvious methods and discuss their advantages and disadvantages.

Method 1. The first method consists of having control lines running through all processor elements in the FSA. By means of a differentiation in the initialization of the control lines, a distinction can be made between dummy and real processor elements. For instance, in Figure 3.1(a) we can run vertical control lines through all columns of processor elements and initialize the control line for the column of added processor elements to 1, in distinction to the rest of the control lines, which are initialized to 0. In this way the control is propagating systolically, in the same way as the data. Similarly we may run vertical and horizontal control lines through the extended FSA and initialize these properly. The advantage of this method is that the extended FSA remains regular and systolic. However, at the cost of increased mesh complexity. An other disadvantage of this method is that we may need more than 1 bit of information per control line to identify dummy processor elements when their distribution is more complicated.

Method 2. The second method consists of specifying a control structure on top of the array. This control structure contains the information which tells a processor element in the RSA when it is either a dummy or real processor element. In this method the control structure resides outside the RSA, so that the mesh complexity of the RSA itself is unaffected. The disadvantage is however, that this complexity is shifted to the complexity of the communication network between the control structure and the RSA. In the worst case the number of interconnections from the control structure to the RSA is equal to the number of processor elements in the RSA. However, each control line carries just 1 bit of information.

In these two methods of control specification we need to modify the instruction set of a processor element in the RSA as follows :

case c
0 : I/O map for real processor element
1 : I/O map for dummy processor element

where the control variable c is issued from the controller to the processor, in case of method 2. In case of method 1 c is a vector valued variable. The entries of the vector being the values of the different control variables that were introduced in the FSA.

3.2.2 COMPUTING ORDERINGS OF THE TILES

Since the computations of the tiles in the full size array are going to be scheduled in pipeline on the RSA, we have to determine an order for the tiles. This order does not change the phase difference between the schedule time steps of the processor elements in a tile. It merely tells which tile is next in the execution of its computations on the RSA. An order is found as follows.

The dependencies between the processor elements of the FSA give rise to a dependency graph $G = \{V, E\}$, of the tessellated FSA. The set V of vertices of this graph

consists of the tiles. A vertex or tile is referred to by the coordinates of the lower left corner of the tile. The set E of edges consists of the dependencies between tiles. These are implied by the dependencies between processor elements in different tiles. The dependencies are computed as follows. Let $\mathbf{d}$ be an interconnection in the FSA and let $\mathbf{p}_1$ and $\mathbf{p}_2$ be two distinct tiles. If $B^{-1}(\mathbf{p}_1 + B\mathbf{n} + \mathbf{d} - \mathbf{p}_2) \in$ F with $\mathbf{n} \in$ F, then $(\mathbf{p}_2 - \mathbf{p}_1)$ is a dependency in the dependency graph of the tessellated FSA. If there exists at least one pair $(\mathbf{p}_i, \mathbf{p}_j)$, $\mathbf{p}_i \neq \mathbf{p}_j$ and $\mathbf{p}_i \neq \mathbf{p}_1$, $\mathbf{p}_j \neq \mathbf{p}_1$, such that $(\mathbf{p}_i - \mathbf{p}_1) = -(\mathbf{p}_j - \mathbf{p}_1)$, then there exists a loop in the dependency graph and the tessellation is not valid for an LPGP partitioning of the FSA. For example, the tessellation in Figure 2.1(a) has vertex set V = $\{U\mathbf{i} \mid \mathbf{i} \in \mathrm{H}\}$, where the matrix U and the domain H are from the 5–*tuple* T of the tessellation in Figure 2.1(a). Thus, V = {(0,0), (0,2), (0,4), (2,0), (2,2), (2,4), (4,0), (4,2), (4,4)}. The interconnections are $[1\ 0]^t$ and $[0\ 1]^t$. Let $\mathbf{p}_1 = 0$. In this case we can restrict the computations $B^{-1}(\mathbf{p}_1 + B\mathbf{n} + \mathbf{d} - \mathbf{p}_2)$ to the "neighborhood" N = {(2,0), (2,2), (0,2), (–2,2), (–2,0), (–2,–2), (0,–2), (2,–2)} of $\mathbf{p}_1$, with $\mathbf{p}_2 \in$ N. With B and F as given in the 5–*tuple* T of the tessellation in Figure 2.1(a), we find :

a. $\mathbf{n} = 0$, $\mathbf{d} = [1\ 0]^t$ gives $([1\ 0]^t - \mathbf{p}_2) \notin$ F for all $\mathbf{p}_2 \in$ N;

b. $\mathbf{n} = 0$, $\mathbf{d} = [0\ 1]^t$ gives $([0\ 1]^t - \mathbf{p}_2) \notin$ F for all $\mathbf{p}_2 \in$ N;

c. $\mathbf{n} = [1\ 0]^t$, $\mathbf{d} = [1\ 0]^t$ gives $([2\ 0]^t - \mathbf{p}_2) \in$ F for $\mathbf{p}_2 = [2\ 0]^t$;

d. $\mathbf{n} = [1\ 0]^t$, $\mathbf{d} = [0\ 1]^t$ gives $([1\ 1]^t - \mathbf{p}_2) \notin$ F for all $\mathbf{p}_2 \in$ N;

e. $\mathbf{n} = [0\ 1]^t$, $\mathbf{d} = [1\ 0]^t$ gives $([1\ 1]^t - \mathbf{p}_2) \notin$ F for all $\mathbf{p}_2 \in$ N;

f. $\mathbf{n} = [0\ 1]^t$, $\mathbf{d} = [0\ 1]^t$ gives $([0\ 2]^t - \mathbf{p}_2) \in$ F for $\mathbf{p}_2 = [0\ 2]^t$;

g. $\mathbf{n} = [1\ 1]^t$, $\mathbf{d} = [1\ 0]^t$ gives $([2\ 1]^t - \mathbf{p}_2) \in$ F for $\mathbf{p}_2 = [2\ 0]^t$;

h. $\mathbf{n} = [1\ 1]^t$, $\mathbf{d} = [0\ 1]^t$ gives $([1\ 2]^t - \mathbf{p}_2) \in$ F for $\mathbf{p}_2 = [0\ 2]^t$;

Hence, the set of dependencies is E = $\{[2\ 0]^t, [0\ 2]^t\}$. the resulting dependency graph is depicted in Figure 2.2(a). Similarly, the dependency graph of the tessellation in

Figure 3.1(b) is shown in Figure 3.2(b).

At this point we should stress the fact that these dependency graphs do not contain any scheduling information whatsoever, that relates to the scheduling of the processor elements in the tiles. The only information that they contain is the plain fact that the computations of a tile depend on those of other tiles. For this reason the delays in a tile are not made explicit in these graphs.

We restrict ourselves to *linear* orderings of the tiles. I.e., orderings of which the elements are computed by a function of the form $f(\mathbf{v}) = \mathbf{p}^t\mathbf{v}$, where $\mathbf{v} \in V$ and $\mathbf{p}$ is a constant vector (the *order* vector), which is found from the following constraints :

$$\mathbf{p}^t[\mathbf{e}_1 \ \cdots \ \mathbf{e}_m] \geq [1 \ \cdots \ 1] \tag{3.4.a}$$

$$\mathbf{p}^t\mathbf{v} \neq \mathbf{p}^t\mathbf{w}, \ \text{for } \mathbf{v} \neq \mathbf{w}, \tag{3.4.b}$$

where $\mathbf{e}_i \in E$ and $\mathbf{v}, \mathbf{w} \in V$. Constraint (3.4.a) is the familiar scheduling constraint as given in (1.12), while constraint (3.4.b) expresses the fact that no two or more tiles may be at the same position in the order. Unlike the scalar products $\mathbf{s}^t\mathbf{i}$ in Chapter 1, the values $f(\mathbf{v})$ do not denote time steps. They merely state precedences, in the sense that if $f(\mathbf{v}) > f(\mathbf{w})$, then the computations of tile $\mathbf{w}$ on the RSA are preceded by the computations of tile $\mathbf{v}$. Examples of order vectors for Figure 3.2.(a) are $\mathbf{p}^t = [3\ 1]$, $\mathbf{p}^t = [2\ 3]$ and $\mathbf{p}^t = [1\ 3]$. These, respectively, lead to the following orderings of the tiles :

1. {(0,0), (0,2), (0,4), (2,0), (2,2), (2,4), (4,0), (4,2), (4,4)};
2. {(0,0), (2,0), (0,2), (4,0), (2,2), (0,4), (4,2), (2,4), (4,4)};
3. {(0,0), (2,0), (4,0), (0,2), (2,2), (4,2), (0,4), (2,4), (4,4)}.

For each order we can keep track of the number of sets of intermediate results that are to be stored at a certain schedule time step, for each output of the RSA. If we would extend the graph in Figure 3.2(a) to a graph of 4×4 tiles, the order vector $\mathbf{p}^t = [3\ 4]$, which enumerates the tiles along the diagonals of the graph, would cause a larger number

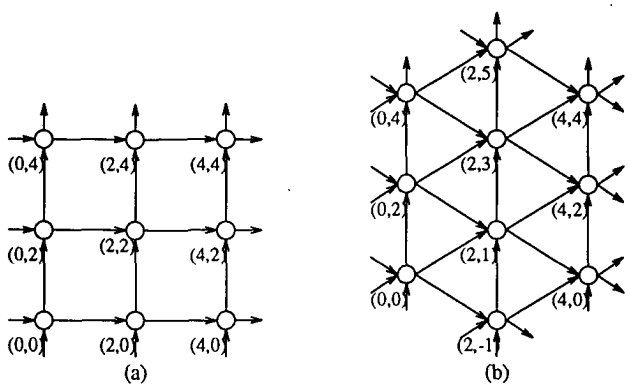


Figure 3.2. Dependency graphs for the tessellations in Figure 3.1.

of sets of intermediate results to be stored temporarily, compared to order vectors which enumerate the tiles along columns or rows. If our strategy were to minimize the total amount of storage for intermediate results at each of the outputs of the array, then we should choose among orderings which yield this minimum.

3.2.3 PIPELINING COMPUTATIONS OF THE TILES ON THE RSA

The model that we shall employ for the pipelining of the computations of the tiles on the RSA is shown in Figure 3.3. It consists of the RSA, selectors (s_i, i=1,...), buffers ($buffer_i$, i=1,...), a controller and an interconnection network. The controller operates a selector s_i via the control signal sc_i :

1. to select either data produced by the RSA or initialization data from the environment (the host computer for example);

2. to re-route the selected input data to one of the two outputs of the selector.

Via the signal tc_i the controller instructs a processor element p_i of the RSA to perform either the identity or the default I/O map. Via the signals denoted by w_i and r_i the

controller specifies, respectively, where data is written into and read from the buffer $buffer_i$. The interconnection network routes the outputs of the RSA (via the selectors and the buffers) to the inputs of the RSA. The routing is determined by the tessellation of the RSA (see Figure 3.2).

The organization of a buffer $buffer_i$, is shown in Figure 3.3(b). The buffer consists of a number of memory banks, **TILE_1** to **TILE_N** and two decoders. A memory bank stores either the data produced by an output of the RSA or the input data of the FSA. The decoders select a specific memory bank, depending on the value of the signals w_{i1} and r_{i1}. The address in a memory bank is specified by the signals w_{i2} and r_{i2}. A memory bank must be envisualized as divided in two separate sections to make simultaneous reading and writing in a memory bank possible. Data from the memory section into which data is written from a selector output is copied into the memory section from which data is read by the RSA. In this model the number of memory banks per buffer has been taken to be equal to the number of tiles in the tessellation of the FSA. Of course, in practice it will suffice to have a smaller number of memory banks, because the contents of a memory bank can be overwritten as soon as the data in this bank has been processed by the RSA. However, such schemes require a more sophisticated memory management algorithm for the controller, than will be the case for the model used here. Moreover, the organization of a buffer, as shown in Figure 3.3(b), may be too general for specific tessellations and order vectors. For instance, consider the tessellation in Figure 3.1(a) and its dependency graph in Figure 3.2(a), with order vector $\mathbf{p}^t = [3\ 1]$. In this case each buffer for the vertical outputs of the RSA can be a simple FIFO (first-in-first-out). And each buffer for the horizontal outputs can be a concatenation of 3 FIFO's. When the first tile in the next column of tiles in Figure 3.2(a) is about to be scheduled, the left most FIFO contains the data needed for the computations of the first tile in a column of the graph in Figure 3.2(a) (counted from top to bottom), the middle FIFO contains the data needed for the computations of the second tile in the column and the right most FIFO contains the data for the computations of the third tile in the column. The algorithm of the controller will be also simpler, because no memory bank selection (w_{i1} and r_{i1}) and address selection signals (w_{i2} and r_{i2}) are needed in this particular case. We would like to remark here

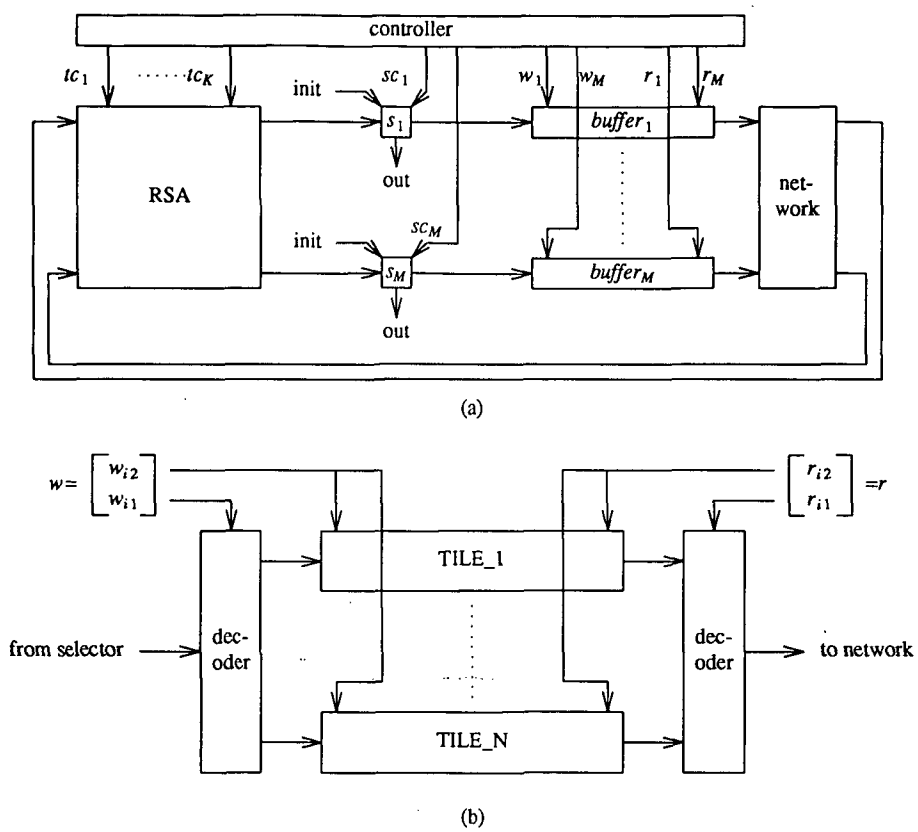


Figure 3.3. (a) Model for an LPGP partitioned FSA. (b) Organization of buffer $buffer_i$.

that the buffer organization in Figure 3.3(b) does not necessarily represent a hardware implementation. The hardware implementation may be in the form of a disk, for instance, in which case the organization in Figure 3.3(b) represents an algorithm for data storage and access on the disk.

THE STRUCTURE OF THE CONTROLLER

For the model in Figure 3.3 we derive an algorithm that implements the controller. Let the following be given. S_in_$buffer_i$ is the sequence of tiles receiving input data of

the FSA which must be supplied to an input of the RSA via $buffer_i$. For instance, if the horizontal input of processor element (0,1) in a tile of Figure 3.1(a) is connected to $buffer_1$, then S_in_$buffer_1$ = {(0,0), (0,2), (0,4)}. S_out_s_i is the sequence of tiles producing output data of the FSA which leave via the output marked by **out** of selector s_i. For instance, if the horizontal output of processor element (1,1) in a tile of Figure 3.1(a) is connected to selector s_1, then S_out_s_1 = {(4,0), (4,2), (4,4)}. S_tiles is the sequence of tiles, ordered according to a specific order vector **p**. For instance S_tiles = {(0,0), (0,2), (0,4), (2,0), (2,2), (2,4), (4,0), (4,2), (4,4)}, for $\mathbf{p}^t = [3\ 1]$ in Figure 3.2(a). S_I/O_p is the sequence of tiles to which a processor element p had to be added in order to achieve the required number of processor elements per tile (the co-ordinates of p are specified relative to the local frame of reference of a tile). For instance, S_I/O_(1,0) = {(4,0), (4,2), (4,4)} in Figure 3.1(a), for processor element (1,0) (relative to the local frame of reference of a tile). S_added is the sequence of processor elements that were added to tiles to achieve the required number of processor elements per tile. For instance S_added = {(1,0), (1,1)} in Figure 3.1(a), where the coordinates of the processor elements are with respect to the local frame of reference of a tile. "dependency_i" is the edge in the dependency graph of the tessellated FSA which is associated with the input of the RSA that connects to the buffer $buffer_i$. Given these sets and dependencies, the algorithms run by the controller for the signals sc_i, tc_i, w_i and r_i are as follows (First(S) returns the first element of sequence S and Remainder(S) returns the remainder of the sequence S when its first element is deleted).

```
/* Initialization of buffers with the input data of the FSA */
for (each buffer buffer_i and selector s_i)
  set signal sc_i of s_i to select the input "init"
  and the output connected to buffer buffer_i;
  /* select memory bank and reset address field w_i2 */
  while (S_in_buffer_i ≠ ∅)
    w_i1 ← (First(S_in_buffer_i) - dependency_i); /* select memory bank corresponding
    to the tile from which input data will be read by the RSA for the
    computations of the tile First(S_in_buffer_i) */
    w_i2 ← 0;
    S_in_buffer_i ← Remainder(S_in_buffer_i);
    /* point to address in selected memory bank, where data for the computations of a tile must be stored */
    while (S_w_i1_buffer_i ≠ ∅) /* S_w_i1_buffer_i is the sequence of schedule
      time steps of the processor element in tile w_i1, which
      will receive its input data from buffer buffer_i */
      w_i2 ← w_i2 + 1;
      S_w_i1_buffer_i ← Remainder(S_w_i1_buffer_i);
    endwhile
  endwhile
endfor
```

```
/* Issue control signals w_i = [w_i2 w_i1]^t to the buffers buffer_i */
for (each buffer buffer_i and selector s_i)
  t_i,0 = 1st schedule time step of the processor element in
  tile First(First(S_tiles)), for which input data is collected in buffer_i;
  do
    no-operation;
  until (schedule_time_step == t_i,0) /* do nothing until time step t_i,0 */
  /* start at t_i,0 */
  while (S_tiles ≠ ∅)
    w_i1 ← First(S_tiles); /* select memory banks to which the results
    of the computations of tile First(S_tiles) will be written */
    w_i2 ← 0;
    S_tiles ← Remainder(S_tiles);
    if (w_i1 ∈ S_out_s_i)
      set signal sc_i to select the input of s_i which is connected
      to the RSA and to select the output of s_i which is labeled "out";
    endif
    else
      set signal sc_i to select the input of s_i which is connected to the RSA
      and to select the output of s_i which is connected to buffer_i;
    endelse
    while (S_w_i1_buffer_i ≠ ∅) /* S_w_i1_buffer_i is the sequence of schedule
      time steps of the processor element in tile (w_i1 + dependency_i)
      for which the input data is being collected in buffer buffer_i */
      w_i2 ← w_i2 + 1;
      S_w_i1_buffer_i ← Remainder(S_w_i1_buffer_i);
    endwhile
  endwhile
endfor
```

```
/* Issue control signals r_i = [r_i2 r_i1]^t to the buffers buffer_i */
for (each buffer buffer_i)
  t_i,00 = 1st schedule time step of the processor element in the tile
  First(S_tiles) which requests data from a buffer buffer_i;
  do
    no-operation;
  until (schedule_time_step == t_i,00) /* do nothing until schedule time step t_i,00 */
  /* start at t_i,00 */
  while (S_tiles ≠ ∅)
    r_i1 ← (First(S_tiles) - dependency_i); /* select memory bank which contains
    the input data for the processor element which gets its input data
    from buffer_i and which starts with its computations for tile First(S_tiles) */
    r_i2 ← 0;
    S_tiles ← Remainder(S_tiles);
    while (S_r_i1_buffer_i ≠ ∅) /* S_r_i1_buffer_i is the sequence of
      schedule time steps of the processor element in tile (r_i1 + dependency_i)
      which requests input data from buffer buffer_i */
      r_i2 ← r_i2 + 1;
      S_r_i1_buffer_i ← Remainder(S_w_i1_buffer_i);
    endwhile
  endwhile
endfor
```

```
/* Issue control signals tc_i to the processor elements of the RSA */
for (each processor element p_i of the RSA)
  t_i = 1st schedule time step of the processor element p_i in First(S_tiles);
  do
    no-operation;
  until (schedule_time_step == t_i) /* do nothing until schedule time step t_i */
  /* start at t_i */
  while (S_tiles ≠ ∅)
    tile ← First(S_tiles);
    S_tiles ← Remainder(S_tiles);
    while (S_tile_p_i ≠ ∅) /* S_tile_p_i is the sequence
      of schedule time steps of processor element p_i in tile "tile";
      if (p_i ∈ S_added & tile ∈ S_I/O_p_i)
        tc_i ← identity_map; /* identity I/O map */
      endif
      else
        tc_i ← default_map; /* default I/O map */
      endelse
    endwhile
  endwhile
endfor
```

3.3 STRATEGY II : LOCAL SEQUENTIAL, GLOBAL PARALLEL PARTITIONING

The LSGP partitioning strategy divides the full size array into congruent clusters of p processor elements each. The scheduling of processor elements in each cluster is serialized and each cluster is replaced by a single cluster processor. In the thesis we restrict ourselves to the LSGP partitioning of 2-D FSAs.

The different steps that are involved in the LSGP partitioning strategy are the following.

1. Find a schedule vector **s**, such that a processor element is scheduled once every *pth* schedule time step in a cluster of p processor elements and one and only one processor element in the cluster is scheduled at a schedule time step.

2. Generate the tessellation of the FSA from the shape of the clusters and, if necessary, add dummy processor elements to tiles (clusters) which contain less than p processor elements.

3. By tracing the sequence of activity in a cluster, determine the control sequences for input selectors and the selection of the appropriate functionality of the cluster processor. An input selector selects either the output of its own cluster processor or that of a different cluster processor. Because, as we shall see, the source of input data for an input of a cluster processor may depend on the schedule time step.

The outline of this section is as follows. We start developing the concepts of a *cluster number*, *cluster* and *cluster direction*. Based on these concepts we formulate a lemma which states the conditions for the existence of a 1-D cluster. The 1-D cluster is spanned by a cluster direction. Based on linear combinations of two linearly independent cluster directions we formulate three propositions for the existence of 2-D clusters. Each proposition states the conditions for the existence of a certain cluster with its own characteristic tessellation pattern. Next we show how the 5-tuple T in (3.3) is derived to characterize the tessellation in case of each of the propositions. And finally, we illustrate how to derive control sequences for the selectors of a cluster processor.

3.3.1 CLUSTER NUMBERS, CLUSTER DIRECTIONS AND CLUSTERS

Let the following be given. $G=\{I^3, D\}$ is the dependency graph of an FSA. From now on we only consider index sets I^3 which contain $(0,0,0)^1$. The vectors **s** and **t** are a

schedule and a projection vector, respectively. The matrix T is a transformation matrix, such that $T\mathbf{t} = 0$ and which maps index set I^3 to index set I^2.

Definition 3.1 : *A cluster* P *is defined to be any subset of* I^2, *such that :*

1. P *has finite cardinality;*
2. P *is homogeneous (see Definition III in CONVENTIONS, SYMBOLS AND DEFINITIONS) in the vector space* $\{\bar{\mathbf{i}} \mid \bar{\mathbf{i}} = T\mathbf{i}, \mathbf{i} \in \mathbb{Z}^3\}$;
3. *no two or more processor elements in* P *are simultaneously scheduled;*
4. *a processor element is scheduled once every card* (P) *time steps.*

Definition 3.2 : *The cluster number of a cluster* P *is equal to card* (P).

Definition 3.3 : *Let* $\bar{\mathbf{d}}_x \in \mathbb{Z}^2$ *be such that there are no processor elements between two processor elements with relative position* $\bar{\mathbf{d}}_x$ *in the processor space* I^2. *If, for every* $\mathbf{p} \in I^2$ *and for some integer* $k > 1$, *only processor elements at ...,* $-k\bar{\mathbf{d}}_x + \mathbf{p}$, $\mathbf{p}$, $k\bar{\mathbf{d}}_x + \mathbf{p}$, $2k\bar{\mathbf{d}}_x + \mathbf{p}$, ... *are scheduled simultaneously, then* $\bar{\mathbf{d}}_x$ *is said to be a cluster direction.*

Notice that it follows from the definition of a homogeneous set that a cluster number is always greater than 1. Also notice that along a cluster direction we find 1-D clusters with cluster number equal to k.

The following lemma is important for the derivation of 2-D clusters.

Lemma 3.1 :

---

1. If I^3 does not contain (0,0,0) it can always be translated such that it does.

Let the following be given. $\mathbf{s} \in \mathbb{Z}^3$ *is a schedule vector,* $\mathbf{t} \in \mathbb{Z}^3$ *is a projection vector and* T *is a* 2×3 *transformation matrix such that* $T\mathbf{t} = 0$. *Let* $\mathbf{s}$ *and* $\mathbf{t}$ *be such that* $|\mathbf{s}^t\mathbf{t}| > 1$. *Let* $\mathbf{d}_x \in \mathbb{Z}^3$ *be such that* $\alpha_x = \mathbf{s}^t\mathbf{d}_x$ *is not a multiple of* $|\mathbf{s}^t\mathbf{t}|$ *and there are no processor elements between two processor elements with relative position* $T\mathbf{d}_x = \bar{\mathbf{d}}_x$. *Then,* $\bar{\mathbf{d}}_x$ *is a cluster direction with cluster number :*

$$c_x = \frac{|\mathbf{s}^t\mathbf{t}|}{gcd(\mathbf{s}^t\mathbf{t},\ \alpha_x)} \tag{3.5}$$

and

$$P = \{\bar{\mathbf{i}} \mid \bar{\mathbf{i}} = k\bar{\mathbf{d}}_x,\ 0 \le k < c_x\} \tag{3.6}$$

is a cluster.

Proof :

Let $\bar{\mathbf{i}} + k_1\bar{\mathbf{d}}_x$ be a line in the processor space I^2, for k_1 a free variable ($k_1 \in \mathbb{Z}$). Let $k_0 \in \mathbb{Z}$ be a free variable. The set of all points in index set I^3 that are mapped by transformation T to this line is given by :

$$M = \{\mathbf{j} \mid \mathbf{j} = \mathbf{i} + k_1\mathbf{d}_x + k_0\mathbf{t},\ T\mathbf{i} = \bar{\mathbf{i}},\ \mathbf{i},\mathbf{j} \in I^3\}.$$

Two points $\mathbf{i}$ and $\mathbf{j}$ in M are simultaneously scheduled if $\mathbf{s}^t\mathbf{i} = \mathbf{s}^t\mathbf{j} = \mathbf{s}^t(\mathbf{i} + k_1\mathbf{d}_x + k_0\mathbf{t})$. Or,

$$[k_0\ \ k_1]\begin{bmatrix}\mathbf{s}^t\mathbf{t}\\ \alpha_x\end{bmatrix} = 0,\ \alpha_x = \mathbf{s}^t\mathbf{d}_x.$$

For the coprime pair (k_0, k_1) and its multiples ..., $(-2k_0, -2k_1)$, $(-k_0, -k_1)$, $(0, 0)$, $(2k_0, 2k_1)$, ... that satisfies this equation, it follows that the processor elements at ..., $-k_1\bar{\mathbf{d}}_x + \bar{\mathbf{i}}$, $0.\bar{\mathbf{d}}_x + \bar{\mathbf{i}}$, $k_1\bar{\mathbf{d}}_x + \bar{\mathbf{i}}$, $2k_1\bar{\mathbf{d}}_x + \bar{\mathbf{i}}$, ... are simultaneously scheduled. Hence, since there are no processor elements between two processor elements which have a difference of position equal to $\bar{\mathbf{d}}_x$, there are exactly $|k_1|$ processor elements that can be clustered between $pk_1\bar{\mathbf{d}}_x$ and $(p+1)k_1\bar{\mathbf{d}}_x$ ($p \in \mathbb{Z}$). Putting $|k_1| = \frac{|\mathbf{s}^t\mathbf{t}|}{gcd(\mathbf{s}^t\mathbf{t},\ \alpha_x)}$ and

$|k_0| = \dfrac{|\alpha_x|}{gcd(\mathbf{s}^t\mathbf{t}, \alpha_x)}$, we have $gcd(k_0, k_1) = 1$. Hence, there are exactly $|k_1| = c_x$ processor elements between $pk_1\bar{\mathbf{d}}_x$ and $(p+1)k_1\bar{\mathbf{d}}_x$ that can be clustered. And since $\alpha_x \neq k\,|\mathbf{s}^t\mathbf{t}|$, it follows that $c_x > 1$. Consequently, $\bar{\mathbf{d}}_x$ is a cluster direction with cluster number c_x and $P = \{\bar{\mathbf{i}} \mid \bar{\mathbf{i}} = k\bar{\mathbf{d}}_x,\ 0 \leq k < c_x\}$ is a cluster.

□

Summarizing, given a certain cluster number c_x and cluster direction $\bar{\mathbf{d}}_x$ a (set of) schedule vector(s) is found from the following sets of constraints :

$$c_x = \frac{|\mathbf{s}^t\mathbf{t}|}{gcd(\mathbf{s}^t\mathbf{t}, \alpha_x)} \tag{3.8.a}$$

$$\mathbf{s}^t[\mathbf{d}_1 \ \cdots \ \mathbf{d}_m] \geq [1 \ \cdots \ 1]. \tag{3.8.b}$$

($\mathbf{d}_1, \ldots, \mathbf{d}_m$ are the dependencies of the dependency graph G). A lemma comparable to Lemma 3.1 was also proved in [Neli1988]. However, there it was proven that, if $|\mathbf{s}^t\mathbf{t}| > 1$ and the transformation T is chosen of the special form $T = \begin{bmatrix} t_3 & 0 & -t_1 \\ 0 & t_3 & -t_2 \end{bmatrix}$ where $\mathbf{t}^t = [t_1\ t_2\ t_3]^t$, then there is at least one cluster direction $[t_3\ 0\ 0]^t$, $[0\ t_3\ 0]^t$, or $[-t_1\ -t_2\ 0]^t$. Lemma 3.1 is different in the sense that it characterizes all possible cluster directions (which may be more than three) for given s and t, such that $|\mathbf{s}^t\mathbf{t}| > 1$, without any assumptions made on the explicit form of the transformation T.

Given the result of Lemma 3.1 for 1-D clusters, a logical extension to 2-D clusters can be based upon linear combinations of two linearly independent cluster directions $\bar{\mathbf{d}}_x$ and $\bar{\mathbf{d}}_y$. With this in mind we specialize Definition 3.1 in case of a 2-D cluster to the following definition.

Definition 3.4 : *Given two linearly independent cluster directions* $\bar{\mathbf{d}}_x$ *and* $\bar{\mathbf{d}}_y$ *which span the processor space* I^2. *I.e., the linear combinations* $k_x\bar{\mathbf{d}}_x + k_y\bar{\mathbf{d}}_y$, *with* $k_x, k_y = \ldots, -1, 0, 1, \ldots$ *are in* I^2. *Any subset* $P \subset I^2$ *of processor elements is defined to be a*

2-D cluster if :

1. *card* (P) = $|\mathbf{s}^t\mathbf{t}|$;
2. *card* (P) > 2 *(we must have card* (P) > 2 *in order to define a plane);*
3. P *is homogeneous in the vector space spanned by* $\bar{\mathbf{d}}_x$ *and* $\bar{\mathbf{d}}_y$;
4. *no two or more processor elements in* P *are simultaneously scheduled;*
5. *a processor element in* P *is scheduled once every* $|\mathbf{s}^t\mathbf{t}|$ *time steps.*

Suppose $\bar{\mathbf{d}}_x$ and $\bar{\mathbf{d}}_y$ span a 2-D cluster. From Lemma 3.1 it follows that cluster numbers c_x and c_y for cluster directions $\bar{\mathbf{d}}_x$ and $\bar{\mathbf{d}}_y$, respectively, are divisors of $|\mathbf{s}^t\mathbf{t}|$. That is :

$$\begin{cases} |\mathbf{s}^t\mathbf{t}| = m_x c_x, \\ |\mathbf{s}^t\mathbf{t}| = m_y c_y, \end{cases}$$

for some positive integers m_x and m_y. Notice that α_x and α_y are the phase differences in schedule time steps between two processor elements with relative position $\bar{\mathbf{d}}_x$ and $\bar{\mathbf{d}}_y$, respectively. If $\alpha_x = \alpha_y$ it means that two processor elements with relative position $(\bar{\mathbf{d}}_x - \bar{\mathbf{d}}_y)$ are simultaneously scheduled. Provided that $(\bar{\mathbf{d}}_x - \bar{\mathbf{d}}_y)$ extents outside the cluster, this is not in conflict with Definition 3.4. Only then $m_x = m_y$ is allowed. Otherwise the remaining possibilities for m_x and m_y are :

1. $m_x > 1$ and $m_y = 1$ (or vice versa);
2. $m_x > 1, m_y > 1$ and $m_x \neq m_y$.

Definition 3.1 does not allow us to find the abundance of 2-D clusters that may exist for $|\mathbf{s}^t\mathbf{t}| > 2$, without an extensive search procedure. However, provided that there exist cluster directions $\bar{\mathbf{d}}_x$ and $\bar{\mathbf{d}}_y$ with cluster numbers c_x and c_y, respectively, we can prove the existence of 2-D clusters which have the shape of a parallelogram, when either of the

following two sets of constraints is satisfied :

1. $gcd(c_x, c_y) = 1$, $c_x c_y = |\mathbf{s}^t\mathbf{t}|$ and $c_x, c_y > 1$, or
2. $c_y = |\mathbf{s}^t\mathbf{t}| = nc_x$ and $n, c_x > 1$;

and we can also prove the existence of 2-D clusters which have a trapezoidal or triangular shape when the following set of constraints is satisfied :

$c_y = |\mathbf{s}^t\mathbf{t}| > 2$ and $c_x > 1$.

The sets 1) and 2) of constraints for clusters with the shape of a parallelogram imply $m_x > 1$, $m_y > 1$ and $m_x > 1$, $m_y = 1$, respectively. The set of constraints for the trapezoidally or triangularly shaped clusters implies $m_x \geq 1$ and $m_y = 1$. Before presenting the existence proofs for these types of clusters we need the following lemma.

Lemma 3.2 :

If $c = |\mathbf{s}^t\mathbf{t}|/gcd(\mathbf{s}^t\mathbf{t}, \alpha)$ *and* α *is not a multiple of* $|\mathbf{s}^t\mathbf{t}|$, *then* $\alpha c = p|\mathbf{s}^t\mathbf{t}|$, *for some non zero integer* p.

Proof :

If $c = |\mathbf{s}^t\mathbf{t}|/gcd(\mathbf{s}^t\mathbf{t}, \alpha)$, then $\alpha = pq$ and $|\mathbf{s}^t\mathbf{t}| = cq$, for some non zero integers p and q with $gcd(p,c) = 1$. Hence, $\alpha c = pqc = p|\mathbf{s}^t\mathbf{t}|$.

□

In the following we denote by $\bar{\mathbf{d}}_x$ and $\bar{\mathbf{d}}_y$ two linearly independent cluster directions with cluster numbers c_x and c_y. $\bar{\mathbf{d}}_x$ and $\bar{\mathbf{d}}_y$ are assumed to span the processor space I^2 of a dependency graph $G=\{I^3, D\}$. I.e., the linear combinations $k_x\bar{\mathbf{d}}_x + k_y\bar{\mathbf{d}}_y$, with $k_x, k_y = ..., -1, 0, 1, ...$ belong to I^2. Processor elements which have a difference in position of $\bar{\mathbf{d}}_x$ or $\bar{\mathbf{d}}_y$ have a phase difference in schedule time steps of α_x or α_y, respectively. If T is the 2×3 transformation matrix ($T\mathbf{t} = 0$, for chosen projection vector $\mathbf{t}$) which maps I^3 to I^2, then $\mathbf{d}_x$ and $\mathbf{d}_y$ are such that $T\mathbf{d}_x = \bar{\mathbf{d}}_x$ and $T\mathbf{d}_y = \bar{\mathbf{d}}_y$. Moreover, from now on we shall

assume that $\mathbf{d}_x$ and $\mathbf{d}_y$ are such that $\alpha_x = \mathbf{s}^t\mathbf{d}_x > 0$ and $\alpha_y = \mathbf{s}^t\mathbf{d}_y > 0$.

Proposition 3.1 :

Let a schedule vector $\mathbf{s}$ *and a projection vector* $\mathbf{t}$ *be given for a dependency graph* $G=\{I^3, D\}$. *Let* $\mathbf{s}$ *and* $\mathbf{t}$ *be such that there exist two linearly independent cluster directions* $\bar{\mathbf{d}}_x$ *and* $\bar{\mathbf{d}}_y$ *with cluster numbers* c_x *and* c_y, *respectively, such that* $gcd(c_x,c_y) = 1$, $c_xc_y = |\mathbf{s}^t\mathbf{t}|$ *and* $c_x,c_y > 1$. *Then there exists a cluster :*

$$P = \{(x,y) \mid [x\ y]^t = k_x\bar{\mathbf{d}}_x + k_y\bar{\mathbf{d}}_y,\ 0 \le k_x < c_x,\ 0 \le k_y < c_y\}, \tag{3.9}$$

with cluster number $|\mathbf{s}^t\mathbf{t}|$.

Proof :

The proof of this proposition is given in two steps. First we shall prove that no two or more processor elements at $k_x\bar{\mathbf{d}}_x$ and $k_y\bar{\mathbf{d}}_y$ are scheduled simultaneously, for $0 \le k_x < c_x$ and $0 \le k_y < c_y$ $((k_x, k_y) \neq (0,0))$. Next, using this result we prove that no two or more processor elements at $k_{x1}\bar{\mathbf{d}}_x + k_{y1}\bar{\mathbf{d}}_y$ and $k_{x2}\bar{\mathbf{d}}_x + k_{y2}\bar{\mathbf{d}}_y$ are scheduled simultaneously, for $0 \le k_{xi} < c_x$ and $0 \le k_{yi} < c_y$ $((k_{x1}, k_{y1}) \neq (k_{x2}, k_{y2}))$, $i=1,2$. But first, from $c_xc_y = |\mathbf{s}^t\mathbf{t}|$ we deduce :

1. $c_x = c_xc_y/gcd(c_xc_y, \alpha_x)$, so that $gcd(c_xc_y, \alpha_x) = c_y$ and thus, $\alpha_x = c_yq_x$, where $gcd(c_x, q_x) = 1$;
2. $c_y = c_xc_y/gcd(c_xc_y, \alpha_y)$, so that $gcd(c_xc_y, \alpha_y) = c_x$ and thus, $\alpha_y = c_xq_y$, where $gcd(c_y, q_y) = 1$.

Step 1. We show that :

$$[k_x\alpha_x] \bmod |\mathbf{s}^t\mathbf{t}| = [k_y\alpha_y] \bmod |\mathbf{s}^t\mathbf{t}|$$

if $k_x = v_xc_x$ and $k_y = v_yc_y$, for some $v_x, v_y \in \mathbb{Z}$, and when $gcd(c_x, c_y) = 1$, $c_xc_y = |\mathbf{s}^t\mathbf{t}|$ and $c_x,c_y > 1$, then

$$[k_x\alpha_x] \mid \mathbf{s}'\mathbf{t} \mid = [k_y\alpha_y] \mid \mathbf{s}'\mathbf{t} \mid$$

only if $k_x = v_x c_x$ and $k_y = v_y c_y$, for $v_x, v_y \in \mathbb{Z}$. The *if* part is easy to prove and follows immediately from Lemma 3.2, after substitution of $k_x = v_x c_x$ and $k_y = v_y c_y$, since in this case $k_x\alpha_x$ and $k_y\alpha_y$ belong to the same equivalence class with respect to congruence modulo $\mid \mathbf{s}'\mathbf{t} \mid$. To prove the *only if* part, we write :

$$k_x\alpha_x = k_y\alpha_y + n \mid \mathbf{s}'\mathbf{t} \mid .$$

Substitution of $\alpha_x = c_y q_x$, $\alpha_y = c_x q_y$ and $\mid \mathbf{s}'\mathbf{t} \mid = c_x c_y$, gives :

$$k_x q_x c_y = k_y q_y c_x + n c_x c_y .$$

Thus :

a. $k_x q_x = k_y q_y \dfrac{c_x}{c_y} + n c_x$, or

b. $k_x q_x \dfrac{c_y}{c_x} = k_y q_y + n c_y$.

Now, since $k_x q_x$ in case a, or $k_y q_y$ in case b, is an integer it must be that :

1. $k_y q_y = p_y c_y$, or $k_y = p_y \dfrac{c_y}{q_y}$ for some integer p_y, since $gcd(c_x, c_y) = 1$ and $c_y > 1$; but, since k_y is an integer it must follow that $p_y = v_y q_y$ and thus, $k_y = v_y c_y$;

2. $k_x q_x = p_x c_x$, or $k_x = p_x \dfrac{c_x}{q_x}$ for some integer p_x, since $gcd(c_x, c_y) = 1$ and $c_x > 1$; but, since k_x is integer it must follow that $p_x = v_x q_x$ and thus, $k_x = v_x c_x$;

Hence, it is proved that processor elements at $k_1 \bar{\mathbf{d}}_x$ and $k_2 \bar{\mathbf{d}}_y$, for any $0 \le k_x < c_x$ and $0 \le k_y < c_y$ with $(k_x, k_y) \neq (0,0)$, are never simultaneously scheduled when $gcd(c_x, c_y) = 1$, $c_x c_y = \mid \mathbf{s}'\mathbf{t} \mid$ and $c_x, c_y > 1$.

Step 2. We show that :

$$[(k_{x1}\alpha_x + k_{y1}\alpha_y)] \mid \mathbf{s}^t\mathbf{t} \mid \neq [(k_{x2}\alpha_x + k_{y2}\alpha_y)] \mid \mathbf{s}^t\mathbf{t} \mid$$

for $0 \leq k_{xi} < c_x$, $0 \leq k_{yi} < c_y$, with ($k_{x1} \neq k_{x2}$ and $k_{y1} \neq k_{y2}$). The proof of this will be by contradiction. Suppose

$$[(k_{x1}\alpha_x + k_{y1}\alpha_y)] \mid \mathbf{s}^t\mathbf{t} \mid = [(k_{x2}\alpha_x + k_{y2}\alpha_y)] \mid \mathbf{s}^t\mathbf{t} \mid$$

for some $0 \leq k_{xi} < c_x$, $0 \leq k_{yi} < c_y$ (i=1,2), with $k_{x1} \neq k_{x2}$ and $k_{y1} \neq k_{y2}$. Then,

$$k_{x1}\alpha_x + k_{y1}\alpha_y = k_{x2}\alpha_x + k_{y2}\alpha_y + n \mid \mathbf{s}^t\mathbf{t} \mid ,$$

for some $n \in \mathbb{Z}$. Hence,

$$(k_{x1} - k_{x2})\alpha_x = (k_{y2} - k_{y1})\alpha_y + n \mid \mathbf{s}^t\mathbf{t} \mid .$$

Or, equivalently,

$$k_x\alpha_x = k_y\alpha_y + n \mid \mathbf{s}^t\mathbf{t} \mid ,$$

where $0 \leq k_x = k_{x1} - k_{x2} < c_x$, $0 \leq k_y = k_{y2} - k_{y1} < c_y$ and $(k_x, k_y) \neq (0,0)$. Hence,

$$[k_x\alpha_x] \mid \mathbf{s}^t\mathbf{t} \mid = [k_y\alpha_y] \mid \mathbf{s}^t\mathbf{t} \mid .$$

But, in **Step 1** we found that

$$[k_x\alpha_x] \mid \mathbf{s}^t\mathbf{t} \mid \neq [k_y\alpha_y] \mid \mathbf{s}^t\mathbf{t} \mid ,$$

for $0 \leq k_x < c_x$, $0 \leq k_y < c_y$ with $(k_x, k_y) \neq (0,0)$. Thus, we have a contradiction. And since the linear combinations $k_x\bar{\mathbf{d}}_x + k_y\bar{\mathbf{d}}_y$ reach all processor elements in the region $0 \leq k_x < c_x$, $0 \leq k_y < c_y$, there are exactly $c_xc_y = \mid \mathbf{s}^t\mathbf{t} \mid$ processor elements in this cluster.

□

Example 3.1 :

Take $c_x = 3$ *and* $c_y = 4$. *Hence,* $c_xc_y = 12$. *From* $c_x = c_xc_y/gcd(c_xc_y, \alpha_x)$, *we could possibly have* $\alpha_x = 4$ *and from* $c_y = c_xc_y/gcd(c_xc_y, \alpha_y)$, *we could possibly have* $\alpha_y = 3$. *With* $0 \leq k_x < 3$, $0 \leq k_y < 4$ *in (3.9), Table 3.1 shows the phase differences between the schedule time steps of the processor element at* $(k_x,k_y) = (0,0)$ *and the schedule time*

steps of the rest of the processor elements in the cluster. These phase differences are given by : $[k_x\alpha_x + k_y\alpha_y] \mid \mathbf{s}^t\mathbf{t} \mid$.

Proposition 3.2 :

Let a schedule vector $\mathbf{s}$ *and a projection vector* $\mathbf{t}$ *be given for a dependency graph* $G=\{I^3, D\}$. *Let* $\mathbf{s}$ *and* $\mathbf{t}$ *be such that there exist two linearly independent cluster directions* $\bar{\mathbf{d}}_x$ *and* $\bar{\mathbf{d}}_y$ *with cluster numbers* c_x *and* c_y, *respectively, such that* $c_y = |\mathbf{s}^t\mathbf{t}| = nc_x$ *and* n, $c_x > 1$. *Then there exists a cluster :*

$$P = \{(x,y) \mid [x\ y]^t = k_x\bar{\mathbf{d}}_x + k_y\bar{\mathbf{d}}_y,\ 0 \le k_x < c_x,\ 0 \le k_y < n\}, \tag{3.10}$$

with cluster number $|\mathbf{s}^t\mathbf{t}|$.

Proof :

If we can prove that there are exactly $\frac{c_y}{n} = c_x$ processor elements between (0,0) and $c_y\bar{\mathbf{d}}_y$, such that there exists a bijection between these c_x processor elements and the c_x processor elements between (0,0) and $c_x\bar{\mathbf{d}}_x$, in the sense that a processor element between (0,0) and $c_x\bar{\mathbf{d}}_x$ is related to one of the c_x processor elements between (0,0) and $c_y\bar{\mathbf{d}}_y$ if they are scheduled simultaneously, then the proposition is proved. Indeed, since the phase difference between schedule time steps of two processor elements along a cluster direction is independent of the absolute position of the processor elements, it will then follow that no two or more processor elements between $k\bar{\mathbf{d}}_x$ and $k\bar{\mathbf{d}}_x + n\bar{\mathbf{d}}_y$ are simultaneously scheduled ($0 \le k < c_x$). And, consequently, for every k_1 and k_2, $0 \le k_1, k_2 < c_x$, there will be no processor element between $k_1\bar{\mathbf{d}}_x$ and $k_1\bar{\mathbf{d}}_x + n\bar{\mathbf{d}}_y$ which is simultaneously scheduled with a processor element between $k_2\bar{\mathbf{d}}_x$ and $k_2\bar{\mathbf{d}}_x + n\bar{\mathbf{d}}_y$.

Thus, we must show that :

$$[k_y n\alpha_y] \mid \mathbf{s}^t\mathbf{t} \mid = [k_x\alpha_x] \mid \mathbf{s}^t\mathbf{t} \mid,\ \text{for } 0 \le k_x, k_y < c_x.$$

But first we need the following. From $c_y = |\mathbf{s}^t\mathbf{t}|$ and $c_y = |\mathbf{s}^t\mathbf{t}| / gcd(\mathbf{s}^t\mathbf{t}, \alpha_x)$ it follows

k_y \ k_x	0	1	2
0	0	4	8
1	3	7	11
2	6	10	2
3	9	1	5

TABLE 3.1. Phase differences between schedule time steps of processor elements in a cluster with $c_x = 3$ and $c_y = 4$.

that

$gcd(c_y, \alpha_y) = 1$. And from $|\mathbf{s}'\mathbf{t}| = c_y = nc_x$ and $c_x = |\mathbf{s}'\mathbf{t}|/gcd(\mathbf{s}'\mathbf{t}, \alpha_x)$ it follows that $gcd(nc_x, \alpha_x) = n$ and, consequently, $\alpha_x = nq$ with $q \geq 1$ and $gcd(c_x, q) = 1$.

Along $\bar{\mathbf{d}}_y$ the phase difference α_y defines c_y distinct equivalence classes $[k'_y\alpha_y]_{c_y}$, i.e., $[0]_{c_y}$, $[1]_{c_y}$, ..., $[c_y - 1]_{c_y}$, for the c_y integer values of k'_y in the interval $[0, c_y - 1]$, since $gcd(c_y, \alpha_y) = 1$ and $gcd(c_y, k) = 1$ for any k in this interval. For a fixed value of k'_y the equivalence class $[k'_y\alpha_y]_{c_y}$ is :

$$\{k'_y\alpha_y + pc_y\},\ p = ...,-1,0,1,... .$$

Dividing all the elements of this class by n gives :

$$\{\frac{k'_y\alpha_y}{n} + pc_x\},\ p = ...,-1,0,1,...,$$

since $c_y = nc_x$. This class is an equivalence class of $\frac{k'_y\alpha_y}{n}$ with respect to congruence modulo c_x, only if $\frac{k'_y\alpha_y}{n}$ is an integer. Since $gcd(c_y, \alpha_y) = 1$ and since n is a divisor of c_y, it follows that k'_y must be a multiple of n if $\{\frac{k'_y\alpha_y}{n} + pc_x\}$, $p = ...,-1,0,1,...$, is to be

an equivalence class of $\frac{k'_y \alpha_y}{n}$ with respect to congruence modulo c_x. That is, $k'_y = nk_y$, $0 \le k_y < c_x$. In that case we have :

$$0 \le r \equiv k_y \alpha_y \ (mod\ c_x) < c_x,\ 0 \le k_y < c_x.$$

From the fact that $\alpha_x = nq$ ($q \ge 1$) we also have that :

$$0 \le r \equiv k_x q \ (mod\ c_x) < c_x,\ 0 \le k_x < c_x.$$

(This follows by dividing the elements of the equivalence class $[k_x \alpha_x]_{c_y} = \{k_x \alpha_x + pc_y\}$, $p = ...,-1,0,1,...$, by n). But this means that :

$$[k_x q]_{c_x} = [k_y \alpha_y]_{c_x},\ 0 \le k_x, k_y < c_x$$

and hence,

$$[k_x nq]_{c_y} = [k_x \alpha_x]_{c_y} = [k_y n \alpha_y]_{c_y},\ 0 \le k_x, k_y < c_x.$$

Thus, it follows that, for each k_x, $0 \le k_x < c_x$, there are n processor elements between $k_x \bar{\mathbf{d}}_x$ and $k_x \bar{\mathbf{d}}_x + n\bar{\mathbf{d}}_y$ which are never scheduled simultaneously. And for every k_1 and k_2, $0 \le k_1, k_2 < c_x$ there is no processor element between $k_1 \bar{\mathbf{d}}_x$ and $k_1 \bar{\mathbf{d}}_x + n\bar{\mathbf{d}}_y$ simultaneously scheduled with a processor element between $k_2 \bar{\mathbf{d}}_x$ and $k_2 \bar{\mathbf{d}}_x + n\bar{\mathbf{d}}_y$. Therefore, (3.10) is a cluster with cluster number $|\mathbf{s}^t \mathbf{t}|$.

□

Example 3.2 :

Take $c_x = 3$ *and* $c_y = |\mathbf{s}^t \mathbf{t}| = 9$. *From* $c_y / c_x = n$ *we get* $n = 3$. *From* $c_x = 9/gcd(9, \alpha_x)$, *we could possibly have* $\alpha_x = 3$ *and from* $c_y = 9/gcd(9, \alpha_y)$, *we could possibly have* $\alpha_y = 4$. *With* $0 \le k_x < 3$ *&* $0 \le k_y < 3$ *in (3.10), Table 3.2 shows the phase differences between the schedule time steps of the processor element at* $(k_x, k_y) = (0,0)$ *and the schedule time steps of the rest of the processor elements in the cluster.*

k_y \ k_x	0	1	2
0	0	3	6
1	4	7	1
2	8	2	5
3	3		
4	7		
5	2		
6	6		
7	1		
8	5		

TABLE 3.2. Phase differences between schedule time steps in a cluster with $c_y = |\mathbf{s}^t\mathbf{t}| = 9$ and $c_x = 3$.

Proposition 3.3 :

Let a schedule vector $\mathbf{s}$ *and projection vector* $\mathbf{t}$ *be given for a dependency graph* $G=\{I^3, D\}$. *Let* $\mathbf{s}$ *and* $\mathbf{t}$ *be such that there exist two linearly independent cluster directions* $\bar{\mathbf{d}}_x$ *and* $\bar{\mathbf{d}}_y$, *with cluster numbers* $c_x > 1$ *and* $c_y = |\mathbf{s}^t\mathbf{t}| > 2$, *respectively, and* $\alpha_x \neq \alpha_y$. *Then there exists a cluster :*

$$P = \{(x,y) \mid [x\ y]^t = k_y\bar{\mathbf{d}}_y,\ k_y = 0,1,...,(m-1),\ [m\alpha_y]_{c_y} = [\alpha_x]_{c_y}\}$$

$$\cup\ \{(x,y) \mid (x,y)^t = k_y\bar{\mathbf{d}}_y + k_x\bar{\mathbf{d}}_x,\ k_y = 0,1,...,(c_y-m-1),\ k_x = 1\} \qquad (3.11)$$

with cluster number $|\mathbf{s}^t\mathbf{t}|$.

Proof :

Since there are exactly c_y equivalence classes $[m\alpha_y]_{c_y}$ for $0 \le m < c_y$, surely for some $0 \le m < c_y$ we have $[\alpha_x]_{c_y} = [m\alpha_y]_{c_y}$. Because the phase difference between the schedule time steps of two processor elements with a difference in position of $\bar{\mathbf{d}}_y$ is independent of the absolute positions of the processor elements, and $\alpha_x \neq \alpha_y$, it follows that the processor elements between (0,0) and $m\bar{\mathbf{d}}_y$ (the processor element at $m\bar{\mathbf{d}}_y$ excluded) are never simultaneously scheduled with those between $\bar{\mathbf{d}}_x$ and $\bar{\mathbf{d}}_x + (c_y - m)\bar{\mathbf{d}}_y$ (the processor element at $\bar{\mathbf{d}}_x + (c_y - m)\bar{\mathbf{d}}_y$ excluded). And since $c_y = |\mathbf{s}^t\mathbf{t}|$, there are exactly $|\mathbf{s}^t\mathbf{t}|$ processor elements in the cluster. Hence, the proof.

□

Example 2.3 :

Take $c_x = 3$ *and* $c_y = |\mathbf{s}^t\mathbf{t}| = 9$. *Hence, we could possibly have* $\alpha_x = 3$ *and* $\alpha_y = 4$. *With* $0 \le k_y < 3$ *for* $k_x = 0$ *&* $0 \le k_y < 6$ *for* $k_x = 1$ *in (2.11), Table 2.3 shows the phase differences between the schedule time steps of the processor element at* $(k_x,k_y) = (0,0)$ *and the schedule time steps of the rest of the processor elements in the cluster.*

Since the nullspace of the transformation matrix T is spanned by $\mathbf{t} \neq 0$, there may be more than one pair $(\mathbf{d}_x, \mathbf{d}_y)$ that maps to the same pair $(\bar{\mathbf{d}}_x, \bar{\mathbf{d}}_y)$. It remains to show that it does not matter which pair $(\mathbf{d}_x, \mathbf{d}_y)$ we choose with regard to α_x and α_y in the computation of the clusters (2.9)-(2.11). Suppose $\mathbf{d}_x$ as well as $\mathbf{d}_x'$ map to $\bar{\mathbf{d}}_x$. This means that :

$$\mathbf{d}_x = \mathbf{d}_x' + k\mathbf{t},$$

for some integer k. Thus, with $\mathbf{s}^t\mathbf{d}_x = \alpha_x$ and $\mathbf{s}^t\mathbf{d}_x' = \alpha_x'$, we get the relation :

$$\alpha_x = \alpha_x' + k\mathbf{s}^t\mathbf{t}.$$

The question is now, whether $gcd(\mathbf{s}^t\mathbf{t}, \alpha_x) = gcd(\mathbf{s}^t\mathbf{t}, \alpha_x')$? Assume that $|\mathbf{s}^t\mathbf{t}| = pq$ and $\alpha_x' = pr$, with $gcd(q,r) = 1$. Hence, $gcd(\mathbf{s}^t\mathbf{t}, \alpha_x') = p$ and $\alpha_x = p(r + kq)$. Thus, we have to show that $gcd(q, r + kq) = 1$. Assuming that $q = nm$ and $r + kq = nl$, with $gcd(m,l) = 1$, leads to :

k_y \ k_x	0	1
0	0	3
1	4	7
2	8	2
3	3	6
4	7	1
5	2	5
6	6	
7	1	
8	5	

TABLE 3.3. Phase differences in schedule time steps in a cluster with $c_y = |\mathbf{s}^t\mathbf{t}| = 9$ and $c_x = 3$.

$$l = \frac{r}{n} + km.$$

But, since $gcd(q,r) = 1$, it follows that $n = 1$. Thus, showing that $gcd(\mathbf{s}^t\mathbf{t}, \alpha_x) = gcd(\mathbf{s}^t\mathbf{t}, \alpha_x')$. Moreover, it does not make any difference whether we use α_x or α_x' to calculate the relative schedule time steps in the clusters, since $[\alpha_x]_{|\mathbf{s}^t\mathbf{t}|} = [\alpha_x']_{|\mathbf{s}^t\mathbf{t}|}$, as follows from the fact that $\alpha_x = \alpha_x' + k\mathbf{s}^t\mathbf{t}$.

3.3.2 TESSELLATION OF THE FSA

Each of the clusters defined by Proposition 3.1-3.3 can be characterized by a non singular, integer matrix B and a support region F as in Section 3.1. Moreover, displacement matrices can be derived for each of these clusters.

CLUSTERS OF TYPE (3.9)

For clusters of this type we have that cluster :

$$P = \{\mathbf{x} \mid B\mathbf{n} = \mathbf{x},\ \mathbf{n} \in F\}$$

is spanned by the vectors $\bar{\mathbf{d}}_x$ and $\bar{\mathbf{d}}_y$. Hence, from (3.9) we have :

$$B = [\bar{\mathbf{d}}_x \mid \bar{\mathbf{d}}_y]$$
$$\mathbf{n} = [k_x \ \ k_y]^t$$

$$F = \left\{ \begin{bmatrix} k_x \\ k_y \end{bmatrix} \right\},\ k_x = 0,1,\ldots,(c_x - 1),\ k_y = 0,\ldots,(c_y - 1).$$

Now, since $[c_x \alpha_x]_{|S^t t|} = 0$, it follows that there is a cluster at relative position $c_x\bar{\mathbf{d}}_x$. Similarly, there is a cluster at relative position $c_y\bar{\mathbf{d}}_y$. Hence, the displacement matrix U is of the form :

$$U = [c_x\bar{\mathbf{d}}_x \mid c_y\bar{\mathbf{d}}_y].$$

The situation is sketched in Figure 3.4, where we have $\mathbf{u}_1 = c_x\bar{\mathbf{d}}_x$ and $\mathbf{u}_2 = c_y\bar{\mathbf{d}}_y$.

Example 3.4 :

As an example, consider the matrix-matrix multiplication example in Figure 1.4, with F and X 8×4 *and* 4×4 *matrices, respectively. Projecting the dependency graph of this algorithm along* $\mathbf{t}^t = [0\ 0\ 1]$ *results in a systolic array as shown in Figure 3.5 (only the topology is shown for simplicity). Choose cluster directions* $\bar{\mathbf{d}}_x = [1\ 0]^t$ *and* $\bar{\mathbf{d}}_y = [0\ 1]^t$,

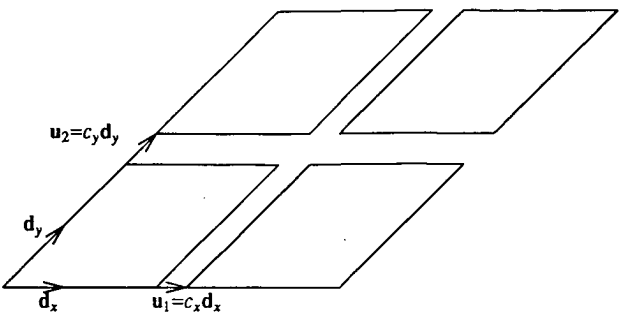


Figure 3.4. Resulting tessellation for tiles of the type in (3.9).

with cluster numbers $c_x = 2$ *and* $c_y = 3$, *respectively. From* $c_x c_y = |\mathbf{s}^t\mathbf{t}|$ *we must have* $|\mathbf{s}^t\mathbf{t}| = 6$. *With* $\mathbf{s} = [s_1\ s_2\ s_3]^t$, *we find that* $s_3 = 6$. *From* $c_x = 2$ *a possible choice for* α_x *could be* $\alpha_x = 3$, *and from* $\alpha_x = \mathbf{s}^t[1\ 0\ 0\ ]^t$ *it follows that* $s_1 = 3$. *From* $c_y = 3$ *we can choose* $\alpha_y = 2$, *and from* $\alpha_y = \mathbf{s}^t[0\ 1\ 0]^t$ *it follows that* $s_2 = 2$. *Hence, we get* $\mathbf{s} = [3\ 2\ 6]^t$. *And for the columns of the displacement matrix U of the tessellation we get* $\mathbf{u}_1 = [2\ 0]^t$ *and* $\mathbf{u}_2 = [0\ 3]^t$, *so that :*

$$U = \begin{bmatrix} 2 & 0 \\ 0 & 3 \end{bmatrix}.$$

The tessellation is shown in Figure 3.5. The numbers adjacent to the processor elements (circles) denote the order in which they are scheduled in a tile.

CLUSTERS OF TYPE (3.10)

For clusters of this type we have :

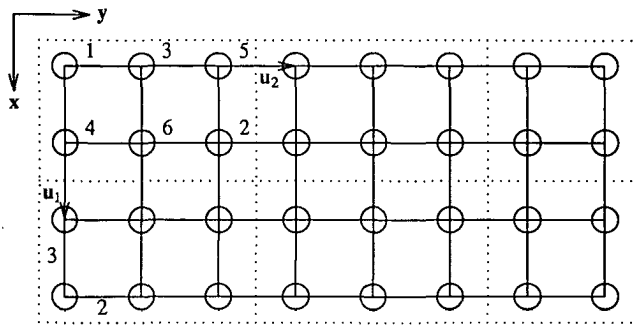


Figure 3.5. Tessellation of the matrix-matrix multiplication array for $\mathbf{s}^t = [3\ 2\ 6]$.

$B = [\bar{\mathbf{d}}_x \mid \bar{\mathbf{d}}_y]$

$\mathbf{n} = [k_x\ k_y]^t$

$$F = \left\{ \begin{bmatrix} k_x \\ k_y \end{bmatrix} \right\},\ k_x=0,\ldots,(c_x-1),\ k_y=0,\ldots,(n-1).$$

Now, since $[c_x\alpha_x]_{|\mathbf{s}^t\mathbf{t}|} = 0$, it follows that there is a cluster at relative position $c_x\bar{\mathbf{d}}_x$. Similarly, there is a cluster at relative position $k\bar{\mathbf{d}}_x + n\bar{\mathbf{d}}_y$, where $0 < k < c_x$ is such that $[k\alpha_x + n\alpha_y]_{c_y} = 0$. Hence, the displacement matrix U is of the form :

$$U = [c_x\bar{\mathbf{d}}_x \mid k\bar{\mathbf{d}}_x + n\bar{\mathbf{d}}_y].$$

The situation is sketched in Figure 3.6, where we have $\mathbf{u}_1 = c_x\bar{\mathbf{d}}_x$ and $\mathbf{u}_2 = k\bar{\mathbf{d}}_x + n\bar{\mathbf{d}}_y$.

Example 3.5 :

Consider again the matrix-matrix multiplication example, with $\mathbf{t}^t = [0\ 0\ 1]$. *Choose cluster directions* $\bar{\mathbf{d}}_x = [1\ 0]^t$ *and* $\bar{\mathbf{d}}_y = [0\ 1]^t$ *with cluster numbers* $c_x = 2$ $c_y = |\mathbf{s}^t\mathbf{t}| = 6$, *respectively. Possible choices of* α_x *and* α_y *are* 3 *and* 5, *respectively. Thus,* $\mathbf{s}^t = [3\ 5\ 6]$. *From* $[k\alpha_x + n\alpha_y]_{c_y} = 0$, *with* $\alpha_x = 3$, $\alpha_y = 5$ *and* $n = c_y/c_x = 3$, *it follows that* $k = 1$. *Hence, the displacement matrix of the tessellation is :*

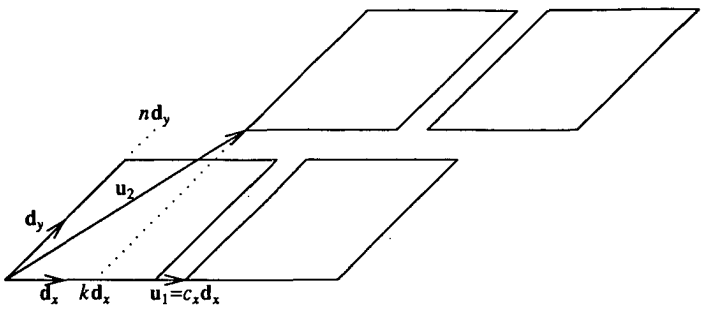


Figure 3.6. Resulting tessellation for tiles of the type in (3.10).

$$U = \begin{bmatrix} 2 & 1 \\ 0 & 3 \end{bmatrix}.$$

The tessellation is shown in Figure 3.7.

CLUSTERS OF TYPE (3.11)

For clusters of this type we have :

$$B = [\bar{\mathbf{d}}_x \mid \bar{\mathbf{d}}_y]$$
$$\mathbf{n} = [k_x \;\; k_y]^t$$

$$\mathrm{F} = \left\{ \begin{bmatrix} 0 \\ k \end{bmatrix} \right\} \cup \left\{ \begin{bmatrix} 1 \\ l \end{bmatrix} \right\}, \; k = 0,1,\ldots,m-1, \;\; l = 0,1,\ldots,(c_y - m - 1)$$

where $[m\alpha_y]_{c_y} = [\alpha_x]_{c_y}$.

From (3.11) we see that adjacent clusters are at relative positions $(\bar{\mathbf{d}}_x + (c_y - m)\bar{\mathbf{d}}_y)$ and $(\bar{\mathbf{d}}_x - m\bar{\mathbf{d}}_y)$. Hence, the displacement matrix U is of the form :

$$U = [\bar{\mathbf{d}}_x + (c_y - m)\bar{\mathbf{d}}_y \mid \bar{\mathbf{d}}_x - m\bar{\mathbf{d}}_y].$$

The situation is sketched in Figure 3.8, where we have $\mathbf{u}_1 = (\bar{\mathbf{d}}_x + (c_y - m)\bar{\mathbf{d}}_y)$ and $\mathbf{u}_2 = (\bar{\mathbf{d}}_x - m\bar{\mathbf{d}}_y)$.

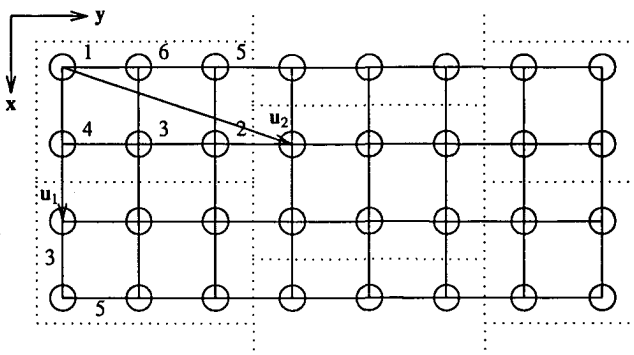


Figure 3.7. Tessellation for $\mathbf{s}^t = [3\ 5\ 6]$.

Example 3.6 :

Consider again the matrix-matrix multiplication example, with $\mathbf{t}^t = [0\ 0\ 1]$. *Choose cluster directions* $\bar{\mathbf{d}}_x = [1\ 0]^t$ *and* $\bar{\mathbf{d}}_y = [0\ 1]^t$ *with cluster numbers* $c_x = 3$ *and* $c_y = |\mathbf{s}^t\mathbf{t}| = 9$, *respectively. Possible choices for* α_x *and* α_y *are* 3 *and* 2, *respectively. Thus,* $\mathbf{s}^t = [3\ 2\ 9]$. *From the equation* $[\alpha_x]_{c_y} = [m\alpha_y]_{c_y}$ *(see the proof of Proposition 3.3) it follows that* $m = 6$. *This gives :*

$$U = \begin{bmatrix} 1 & 1 \\ 3 & -6 \end{bmatrix}$$

for the displacement matrix of the tessellation. The tessellation is shown in Figure 3.9.

3.3.3 THE DERIVATION OF CONTROL SEQUENCES FOR INPUT SELECTORS

Similar to the LPGP case, we are able to construct a dependency graph for the LSGP partitioned FSA. For examples we refer to Figure 3.2. In such a graph the nodes represent the cluster processor. A cluster processor consists of :

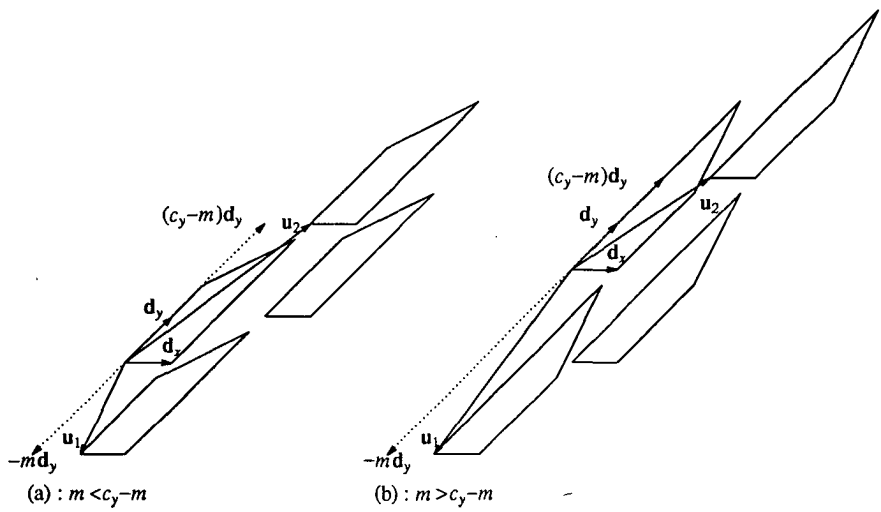


Figure 3.8. Resulting tessellations for tiles of the type in Proposition 3.3.

1. the processing element;
2. selectors, which are used to control input selection;
3. *first-in-first-out* (FIFO) memory for intermediate results;
4. a controller which selects different functionalities of a processor element and which controls the selector.

This model of the cluster processor is depicted in Figure 3.10.
There is a selector for each input of the processing element. Similarly, there is a feed back loop with a FIFO buffer for each output of the processing element. The FIFO buffer at an output of the processing element has length equal to the number of delay units that is associated with the interconnection in the FSA to which this output corresponds.

The issue that remains is the derivation of the control for the input selectors. This is not a hard problem to tackle. The control is easily derived by tracing for each input of the processing element in the cluster processor from which cluster processor input data is

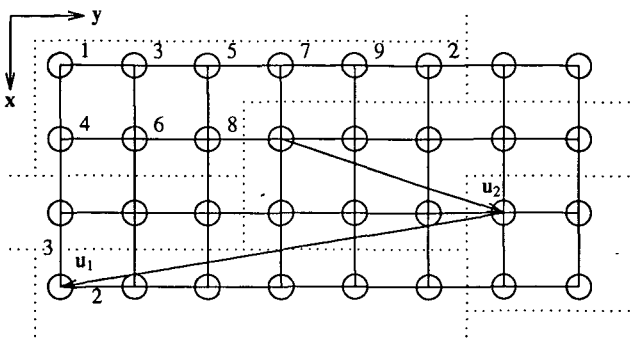


Figure 3.9. Tessellation for $\mathbf{s}^t = [3\ 2\ 9]$.

requested. Below we give a hint towards the implementation of such a trace procedure.

Let us define an indexing function I for a cluster :

$$I : [k_x\alpha_x + k_y\alpha_y]_{|\mathbf{s}^t\mathbf{t}|} \rightarrow (k_x\bar{\mathbf{d}}_x + k_y\bar{\mathbf{d}}_y),$$

where k_x, k_y are defined within the extend of the cluster. I.e., we use the relative schedule time steps as indices for the processor elements in the cluster.

With $\overline{\mathrm{D}}$ the set of interconnection primitives in the FSA, D(I) the domain of index function I and E the set of dependencies in the graph of the tessellated FSA we can define the following map :

$$M : \mathrm{D}(I) \times \overline{\mathrm{D}} \rightarrow \overline{\mathrm{D}} \cup \mathrm{E}$$

This map relates the input edges (which are members of the set $\overline{\mathrm{D}}$) of the processor element in the cluster processor at each relative schedule time step (an element of D(I)) to either an *inter*-cluster dependence (an element of E) or an *intra*-cluster dependence (an element of $\overline{\mathrm{D}}$).

At each schedule time step we can now specify a formal set of rules to extract the control information of the selectors of a cluster processor :

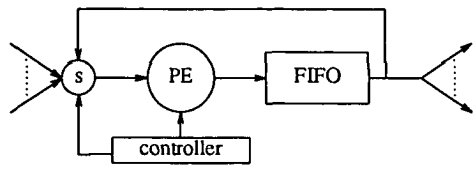


Figure 3.10. Model of a cluster processor.

for all $(x_i, y_i) \in \mathrm{P}$, *with* $I(i) = (x_i, y_i)$ *& all* $(t, u) \in \overline{\mathrm{D}}$ *:*

1. $(x_i, y_i) - (t, u) \in \mathrm{P} \Rightarrow M(i, (t, u)) = (t, u)$

2. $(x_i, y_i) - (t, u) \notin \mathrm{P} \Rightarrow M(i, (t, u)) = (v, w)$, $(v, w) \in \mathrm{E}$ *and* $(v, w) = -U\mathbf{i}$, *where* $\mathbf{i}$ *is such that* $(x_i - t, y_i - u) \in \{\mathbf{x} \mid \mathbf{x} = \mathbf{y} + U\mathbf{i} + \mathbf{r}, \mathbf{y} \in \mathrm{P}\}$.

In plain words : for a processor element (x_i, y_i) scheduled at time step i, relative to the processor elements in the same cluster, we test for all its inputs (t, u), whether the input is coming from a processor element in the same cluster P, or whether it comes from a different cluster. If it comes from a different cluster, we have to determine from which one. Since we know that $(x_i - t, y_i - u)$ belongs to some cluster $\{\mathbf{x} \mid \mathbf{x} = \mathbf{y} + U\mathbf{i} + \mathbf{r}, \mathbf{y} \in \mathrm{P}\}$, we conclude that the input is originating from a cluster with relative position $U\mathbf{i}$.

The controller can control the I/O map of the processing element of the cluster processor by keeping track of the next virtual processor element in the cluster that has to be simulated by the cluster processor. Notice that this is not necessary if systolically propagating control signals were already available in the FSA to select the proper I/O map for each of the processor elements in the FSA.

4. DESIGN OF A SYSTOLIC ARRAY FOR SOLVING SYSTEMS OF LINEAR EQUATIONS

4.1 INTRODUCTION

In this chapter a fixed size systolic array is designed which executes the feed forward *QR* factorization method as well as the feed forward *Schur-Cholesky* factorization method of Section 2.2.2, Equation (2.23) and Section 2.2.3, Equation (2.46), respectively. However, instead of performing orthogonal and hyperbolic rotations, the processor elements of this array will execute orthogonal and hyperbolic Householder transformations. Thus we show that Householder transformations can be used in the feed forward direct methods of Chapter 2. First, two full size arrays will be designed, each implementing one of the two feed forward methods. These designs are done with the aid of **SYSTARS**, a CAD system for the synthesis of systolic arrays [Omtz1987]. **SYSTARS** has a graphics interface which displays a specified dependency graph and allows one to obtain from it a full size systolic array interactively, by specifying a projection vector **t** and a schedule vector **s**, as was explained in Chapter 1.

Next, reduced size arrays will be derived from the full size arrays, using the LPGP partitioning strategy of Chapter 3. It will turn out that the RSAs for both feed forward methods have identical interconnection topologies, so that a single RSA will be obtained for both methods. This RSA will be designed such that the following constraints are satisfied :

1. the size of the local memory of a processor element does not depend on the size of the problem;

2. the size of problems to be executed on the RSA can be arbitrary larger than the size of the array itself;

3. all I/O with peripherals are handled at the boundaries of the RSA;

4. the RSA is flexible with respect to communication speed requirements of different peripherals (disks, host computer), attached to the array.

Constraints 1) and 2) will be satisfied by using the LPGP partitioning strategy of Chapter 3. Constraint 3) will be satisfied by properly tessellating the FSAs and by adding dummy processor elements where needed. Constraint 4) will be satisfied by implementing the Householder transformations on innerproduct-step processor elements. An innerproduct-step processor executes an innerproduct sequentially. The entries of a data vector on which a Householder transformation is applied are multiplexed at a single input of the processor element. Likewise, the entries of the resulting vector are multiplexed at a single output. As we shall see, by defining the processor element as a sequentially operating processing unit, we are able to control the I/O bandwidth of the RSA by the size of the input vectors. Thus, we obtain the effect of an LSGP partitioning (see Section 3.3).

This chapter mainly illustrates the feasibility of the design of a systolic array, dedicated to the feed forward *QR* and *Schur-Cholesky* factorization method, using the design principles of Section 1.2 and Section 3.2. The architecture of the Householder innerproduct-step processor is a conservative signal processor like architecture [Bara1988], containing standard components such as a floating point multiply-add system [Hwan1979], registers, FIFO memories, random access memory (RAM) and a micro-controller with a micro-instruction read only memory (ROM) and a sequencer as its main components. Hence, it can be expected that the design turn-around time of such a processor element will be short.

Note : the design of a systolic array which uses circular or hyperbolic rotations is not studied here. The availability in the near future of standard, high throughput pipelined

CORDIC (Coordinate Rotating Digital Computer) processor elements [Walt1971], [Lang1988] for these operations will make the design of such systolic arrays feasible and attractive.

The outline of this chapter is as follows. In Section 4.2 the feed forward *QR* and *Schur-Cholesky* factorization methods are restated in terms of Householder transformations. Regular recurrent algorithms are derived for these methods, from which dependency graphs are deduced. In Section 4.3 full size arrays are synthesized from these dependency graphs, using the principles of Section 1.2. In Section 4.4 these FSAs are partitioned, using the LPGP partitioning strategy of Section 3.2. And finally, in Section 4.5 the architecture of an innerproduct-step Householder processor element is presented.

4.2 FEED FORWARD SOLVERS

4.2.1 THE QR AND SC SOLVERS

For convenience of the reader the feed forward *QR* and *Schur-Cholesky* factorization methods of Chapter 2 are summarized here in the form of two theorems.

Theorem I (QR solver): *Let* $A \in \mathbb{R}^{N\times N}$ *be a non singular matrix and* $\mathbf{b} \in \mathbf{R}^{N}$. *Then, there exists an* $(N+1)\times(N+1)$ *orthogonal matrix* Θ *:*

$$\Theta = \prod_{j=1}^{\overleftarrow{N}} \prod_{i=j}^{\overleftarrow{N}} \Theta_{ij}, \tag{4.1}$$

where the Θ_{ij} *are* $(N+1)\times(N+1)$ *orthogonal matrices of the form :*

$$\Theta_{ij} = \begin{bmatrix} I_{j-1} & & & \\ & cos(\theta_{ij}) & sin(\theta_{ij}) & \\ & I_{i-j} & & \\ & -sin(\theta_{ij}) & cos(\theta_{ij}) & \\ & & & I_{N-i} \end{bmatrix}, \tag{4.2}$$

such that the solution $\mathbf{x}$ *of the system of linear equations* $A\mathbf{x} = \mathbf{b}$ *obeys :*

$$\Theta \begin{bmatrix} A^t & I & 0 \\ -\mathbf{b}^t & 0 & 1 \end{bmatrix} = \begin{bmatrix} R & * & * \\ 0 & k_\Theta \mathbf{x}^t & k_\Theta \end{bmatrix}, \tag{4.3}$$

where R *is an* $N \times N$ *upper triangular matrix, satisfying* $R^tR = AA^t + \mathbf{b}\mathbf{b}^t$ *and* k_Θ *is equal to :*

$$k_\Theta = (1 + \mathbf{x}^t\mathbf{x})^{-\frac{1}{2}}. \tag{4.4}$$

□

A matrix Θ which satisfies Equation (4.3) will be referred to as the *QR solver*, because Equation (4.3) expresses a *QR* factorization.

Theorem II (SC solver) : *Let* $A \in \mathbb{R}^{N \times N}$ *be a symmetric positive definite matrix with diagonal elements* $a_{ii} = 1$, $i = 1,...,N$, *and* $\mathbf{b} \in \mathbb{R}^N$, *such that the matrix :*

$$\bar{A} = \begin{bmatrix} 1 & -\mathbf{b}^t \\ -\mathbf{b} & A \end{bmatrix} \tag{4.5}$$

is a positive definite matrix. Let U *and* Y *be the lower, respectively, strictly lower triangular part of the matrix* $\bar{A}$ *and let* J *be the signature matrix* $J = I_{N+1} \oplus -I_{N+1}$. *Then, there exists a* $2(N+1) \times 2(N+1)$ *matrix* Φ *:*

$$\Phi = \prod_{j=1}^{\overleftarrow{N}} \prod_{i=j}^{\overleftarrow{N}} \Phi_{ij}, \tag{4.6}$$

where the Φ_{ij} *are* $2(N+1) \times 2(N+1)$ *J-orthogonal matrices of the form :*

$$\Phi_{ij} = \begin{bmatrix} I_i & & & \\ & cosh(\phi_{ij}) & & sinh(\phi_{ij}) \\ & & I_{N-j-1} & \\ & sinh(\phi_{ij}) & & cosh(\phi_{ij}) \\ & & & & I_{N+j-i-1} \end{bmatrix}, \tag{4.7}$$

such that the solution $\mathbf{x}$ *of the system of linear equations* $A\mathbf{x} = \mathbf{b}$ *obeys :*

$$\Phi \begin{bmatrix} U^t & I_{N+1} \\ Y^t & I_{N+1} \end{bmatrix} = \begin{bmatrix} R & * & * \\ 0 & k_\Phi \mathbf{x}^t & k_\Phi \\ 0 & * & * \end{bmatrix} \begin{matrix} N+1 \\ 1 \\ N \end{matrix}, \tag{4.8}$$

where R *is an* $N \times N$ *upper triangular matrix, satisfying* $R^t R = \bar{A}$ *and* k_Φ *is equal to :*

$$k_\Phi = (1 - \mathbf{x}^t A \mathbf{x})^{-1/2}, \quad \mathbf{x}^t A \mathbf{x} < 1. \tag{4.9}$$

□

A matrix Φ which satisfies Equation (4.8) will be referred to as the *SC solver*, because Equation (4.8) expresses a *Schur-Cholesky* factorization.

4.2.2 ORTHOGONAL AND HYPERBOLIC HOUSEHOLDER TRANSFORMATIONS

Instead of defining the matrices Θ and Φ in (4.3) and (4.8) in terms of the orthogonal and J-orthogonal rotations in (4.2) and (4.7), respectively, they can also be defined in terms of orthogonal and hyperbolic Householder transformations, respectively.

ORTHOGONAL HOUSEHOLDER TRANSFORMATIONS [Hous1975] :

The orthogonal Householder transformation is defined to be the following $N \times N$ matrix :

$$H = I_N - \frac{2}{\mathbf{u}^t\mathbf{u}}\mathbf{u}\mathbf{u}^t, \tag{4.10}$$

where $\mathbf{u}$ is a non-zero $N\times 1$ vector. Given a non-zero vector $\mathbf{x}$, we can determine the vector $\mathbf{u}$, such that :

$$H\mathbf{x} = \pm \| \mathbf{x} \|_2 \mathbf{e}_1, \tag{4.11.a}$$

where $\| \mathbf{x} \|_2 = (\mathbf{x}^t\mathbf{x})^{1/2}$ and $\mathbf{e}_1$ is the vector $[1\ 0\ \cdots\ 0]^t$. It is readily verified that

$$\mathbf{u} = \mathbf{x} \pm \| \mathbf{x} \|_2 \mathbf{e}_1. \tag{4.11.b}$$

We choose $\mathbf{u} = \mathbf{x} + sign(x_1)\| \mathbf{x} \|_2 \mathbf{e}_1$, to guarantee that $\| \mathbf{u} \|_2 \geq \| \mathbf{x} \|_2$.

HYPERBOLIC HOUSEHOLDER TRANSFORMATIONS [Rade1985] :

The hyperbolic Householder transformation is defined to be the following $N\times N$ matrix :

$$\underline{H} = J - \frac{2}{\underline{\mathbf{u}}^t J \underline{\mathbf{u}}}\underline{\mathbf{u}}\underline{\mathbf{u}}^t, \tag{4.12}$$

where $\underline{\mathbf{u}}$ is a non-zero $N\times 1$ vector and J is an $N\times N$ signature matrix :

$$J = diag\,(1, \pm 1, \cdots, \pm 1). \tag{4.13}$$

(Notice that (4.12) expresses an orthogonal Householder transformation in case all diagonal entries of J are positive). The hyperbolic Householder transformation is J-orthogonal, i.e.,

$$\underline{H}^t J \underline{H} = \underline{H} J \underline{H}^t = J. \tag{4.14}$$

Given a non-zero vector $\mathbf{x}$, such that $\mathbf{x}^t J\mathbf{x} > 0$, we can determine the vector $\underline{\mathbf{u}}$, such that

$$\underline{H}\mathbf{x} = \pm \| \mathbf{x} \|_J \mathbf{e}_1, \tag{4.15.a}$$

where $\| \mathbf{x} \|_J = (\mathbf{x}^t J\mathbf{x})^{1/2}$. In this case it is readily verified that :

$$\underline{\mathbf{u}} = J\mathbf{x} \pm \| \mathbf{x} \|_J \mathbf{e}_1 . \qquad (4.15.b)$$

We choose $\underline{\mathbf{u}} = J\mathbf{x} + sign(x_1) \| \mathbf{x} \|_J \mathbf{e}_1$, to guarantee that $\| \underline{\mathbf{u}} \|_J \geq \| \mathbf{x} \|_J$.

The computation of the reflection vector **u** or $\underline{\mathbf{u}}$, from a given vector **x** and the computations of $\| \mathbf{x} \|_2 \mathbf{e}_1$ or $\| \mathbf{x} \|_J \mathbf{e}_1$ will be referred to as *vectorizing*, while the application of a given H or $\underline{H}$ matrix to a given vector **y**, will be referred to as *reflection*.

4.2.3 ORTHOGONAL REFLECTIONS AS ELEMENTARY OPERATIONS IN THE QR SOLVER

The matrix Θ in (4.3) can be replaced by a product of (embedded) orthogonal Householder transformations of fixed size. For instance, let A be an 8×8 matrix and **b** an 8×1 vector and write :

$$P = \begin{bmatrix} A^t & I_8 & 0 \\ -\mathbf{b}^t & 0^t & 1 \\ 0^t & 0^t & 0 \\ 0^t & 0^t & 0 \\ 0^t & 0^t & 0 \end{bmatrix} = [p_{xy}], \; x=1,...,12, \; y=1,...,17.$$

And let H_{ij}, $i=1,...,8$, $j=\lceil i/3 \rceil,...,3$, be orthogonal Householder transformations of the form $H_{ij} = I_4 - \frac{2}{\mathbf{u}_{ij}^t \mathbf{u}_{ij}^t} \mathbf{u}_{ij} \mathbf{u}_{ij}^t$. Then, making the proper choices for the H_{ij}, the matrix :

$$
\begin{aligned}
\Theta = {} & (I_7 \oplus H_{83} \oplus 1) \\
& (I_6 \oplus H_{73} \oplus I_2) \\
& (I_5 \oplus H_{63} \oplus I_3)(I_8 \oplus H_{62}) \\
& (I_4 \oplus H_{53} \oplus I_4)(I_7 \oplus H_{52} \oplus 1) \\
& (I_3 \oplus H_{43} \oplus I_5)(I_6 \oplus H_{42} \oplus I_2) \\
& (I_2 \oplus H_{33} \oplus I_6)(I_5 \oplus H_{32} \oplus I_3)(I_8 \oplus H_{31}) \\
& (1 \oplus H_{23} \oplus I_7)(I_4 \oplus H_{22} \oplus I_4)(I_7 \oplus H_{21} \oplus 1) \\
& (H_{13} \oplus I_8)(I_3 \oplus H_{12} \oplus I_5)(I_6 \oplus H_{11} \oplus I_2),
\end{aligned} \tag{4.16}
$$

will upper triangularize the matrix P in the product ΘP. Notice that the *ith* row (counting from bottom to top) of Θ eliminates the *ith* column of the strictly lower triangular part of P. Application of the *ith* row will be referred to as the *ith* sweep and the part of a column on which a matrix H_{ij} acts will be called a partial vector.

In order to formulate the computation of the product ΘP in terms of a regular recurrent algorithm (see Chapter 1), we set up the following scheme of operating on the columns of the matrix P. Let us consider the first sweep. Denote the 4×1 partial vector on which H_{11} $(i=j=1)$ acts by $\mathbf{p}_{11k} = [\mathbf{v}_{11k}^t \; e_{11k}]^t$, where the first three entries of this vector are contained in the 3×1 vector $\mathbf{v}_{11k}$ and the last entry is contained in e_{11k}. The first index of the variables $\mathbf{v}$ and e denotes the current sweep, the second index denotes the current partial vector and the third index denotes the current column of P. Since the matrix H_{12} acts on a partial vector which has last entry equal to the first entry of $H_{11}\mathbf{p}_{11k}$, we decompose the partial vector $H_{11}\mathbf{p}_{11k}$ into its first entry, denoted by e_{12k} $(2=j+1)$ and the remaining three entries, denoted by the 3×1 vector $\mathbf{v}_{21k}$ $(2=i+1)$. These remaining entries are going to be used in the second sweep. Denoting the first three entries of the partial vector on which H_{12} will act, by $\mathbf{v}_{12k}$, this partial vector is then denoted by $[\mathbf{v}_{12k}^t \; e_{12k}]^t$. Similarly for the rest of the transformations in (4.16). In order to express the application of a transformation H_{ij} on all of the columns of P, we denote the reflection vector $\mathbf{u}_{ij}$ by $\mathbf{u}_{ijk}$ - where the indices have the same interpretation as above - and copy it to $\mathbf{u}_{i,j,k+1}$, when H_{ij} must be applied to the partial vector in column $k+1$ after it has been applied to the partial vector in column k. However, since a reflection vector

$\mathbf{u}_{ij}$ has to be computed first from the *jth* partial vector in column $i=k$ before it can be applied to subsequent columns, we introduce a control variable c_{ijk} which is initialized to 0 for $i=k$ and to 1 everywhere else. Whenever $c_{ijk}=0$, a reflection vector $\mathbf{u}_{ijk}$ is computed. Else the computed reflection vector is used to reflect the partial vector. The value of c_{ijk} is copied to $c_{i+1,j,k+1}$, because the partial vectors in column $k+1$ in sweep $i+1$, will be next eliminated and it is from this column that new reflection vectors are computed. The regular recurrent algorithm for this scheme of implementing the computation of the product ΘP is given in Figure 4.1. A reflection vector $\mathbf{u}_{i,j,k+1}$ is computed from partial vector $[\mathbf{v}_{ijk}^t \; e_{ijk}]^t$ by the function $f()$ (see (4.11.b)). The function $g()$ computes $\pm\|[\mathbf{v}_{ijk}^t \; e_{ijk}]^t\|_2\mathbf{e}_1$. That is, $e_{i,j+1,k}=\pm\|[\mathbf{v}_{ijk}^t \; e_{ijk}]^t\|_2$ and $\mathbf{v}_{i+1,j,k}=[0\,0\,0]^t$. The function $k()$ computes a reflection of $[\mathbf{v}_{ijk}^t \; e_{ijk}]^t$ with the previously computed reflection vector $\mathbf{u}_{ijk}$.

The set of displacement vectors for the variables $\mathbf{v}_{ijk}$, e_{ijk}, c_{ijk} and $\mathbf{u}_{ijk}$ is: $D_{QR}=\{[1\,0\,0]^t, [0\,1\,0]^t, [1\,0\,1]^t, [0\,0\,1]^t\}$. The index set I^3_{QR} of the algorithm is: $I^3_{QR}=\{(i,j,k) \mid 1\le i\le 8, \lceil i/3\rceil \le j\le 3, i\le k\le 17\}$. The set of index points (i,j,k) and the displacement vectors, define the vertices and the edges, respectively, of the dependency graph of the algorithm in Figure 4.1. This graph is depicted in Figure 4.2, for k only until 9 to avoid a too complex graph. At a node (i_1,j_1,k_1) of the graph the computations at $i=i_1$, $j=j_1$, $k=k_1$, in the regular recurrent algorithm, are performed. The results for the variables e, $\mathbf{v}$, $\mathbf{u}$ and c at node (i_1, j_1, k_1) are communicated to nodes at relative positions $[0\,1\,0]^t$, $[1\,0\,0]^t$, $[0\,0\,1]^t$ and $[1\,0\,1]^t$, respectively. The location of initialization data in the graph and the location of output data follows immediately from the algorithm in Figure 4.1.

4.2.4 HYPERBOLIC REFLECTIONS AS ELEMENTARY OPERATIONS IN THE SC SOLVER

The matrix Φ in (4.8) can be replaced by a product of (embedded) hyperbolic transformations which act on vectors of n entries. For instance, let $\bar{A}$ in (4.5) be a 10×10

/* initializations */

$c_{ijk}, e_{ijk} \in \mathbf{R}$; $\mathbf{u}_{ijk} \in \mathbf{R}^4$; $\mathbf{v}_{ijk} \in \mathbb{R}^3$;

$e_{i1k} = 0$;

$\mathbf{v}_{1jk} = [p_{m(j)-2,k}\ p_{m(j)-1,k}\ p_{m(j),k}]^t$ with $m(j) = 9 - 3(j-1)$, j=1,2,3;

$c_{1j1} = 0$; $c_{1jk} = 1$ for $k \neq 1$;

/* transformations */

for i=1 **to** 8

 for $j = \lceil i/3 \rceil$ **to** 3

  for k=i **to** 17

   case c_{ijk}

$$0: \quad \mathbf{u}_{i,j,k+1} = f\left(\begin{bmatrix}\mathbf{v}_{ijk}\\ e_{ijk}\end{bmatrix}\right);$$

$$\begin{bmatrix}e_{i,j+1,k}\\ \mathbf{v}_{i+1,j,k}\end{bmatrix} = g\left(\begin{bmatrix}\mathbf{v}_{ijk}\\ e_{ijk}\end{bmatrix}\right);$$

$$1: \quad \mathbf{u}_{i,j,k+1} = \mathbf{u}_{ijk};$$

$$\begin{bmatrix}e_{i,j+1,k}\\ \mathbf{v}_{i+1,j,k}\end{bmatrix} = h\left(\begin{bmatrix}\mathbf{v}_{ijk}\\ e_{ijk}\end{bmatrix}, \mathbf{u}_{ijk}\right);$$

   endcase

   $c_{i+1,j,k+1} = c_{ijk}$;

  endfor

 endfor

endfor

Figure 4.1. A regular recurrent algorithm for the QR solver.

matrix with lower and strictly lower triangular parts U and Y, respectively, and write :

$$M = \begin{bmatrix}P\\ Q\end{bmatrix},$$

with :

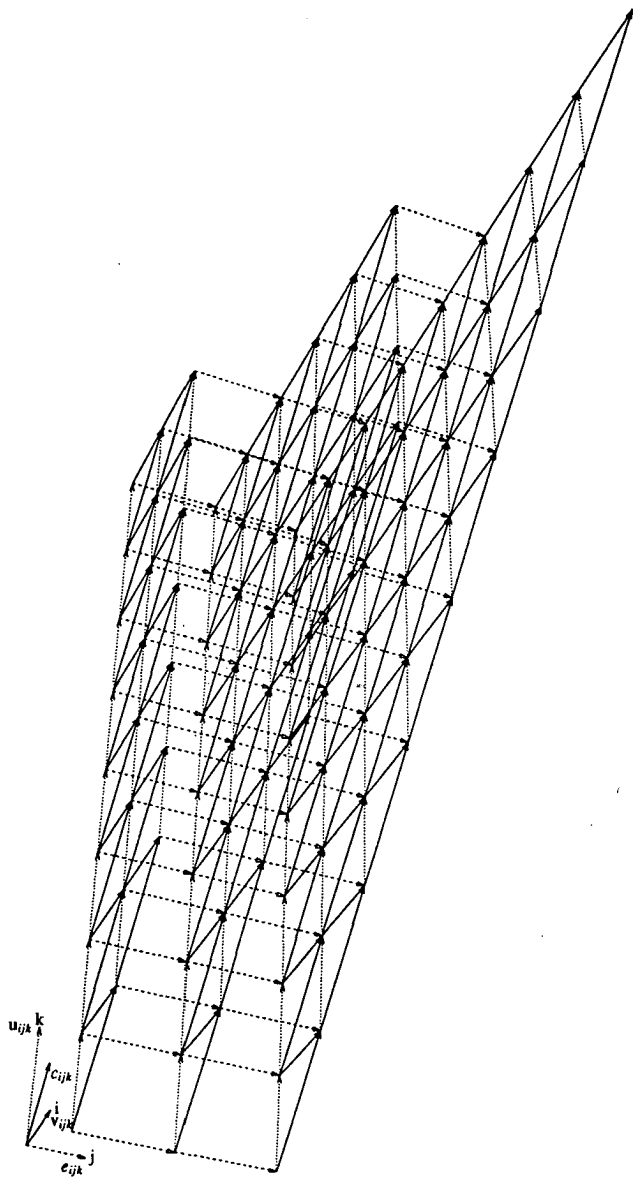


Figure 4.2. Dependency graph of the algorithm in Figure 4.1.

$$P = [U^t \ I_{N+1}] = [p_{xy}], \ x=1,...,10, y=1,...,20,$$
$$Q = [Y^t \ I_{N+1}] = [q_{xy}], \ x=1,...,10, y=1,...,20.$$

Let $\underline{H}_{ij}$, $i = 1,...,9$, $j = 1,...,3-\lfloor(i-1)/3\rfloor$, be hyperbolic Householder transformations of the form $\underline{H}_{ij} = J_{13-i} - \frac{2}{\underline{\mathbf{u}}_{ij}^t J_{13-i} \underline{\mathbf{u}}_{ij}} \underline{\mathbf{u}}_{ij} \underline{\mathbf{u}}_{ij}^t$, with $\underline{\mathbf{u}}_{ij}^t = [\underline{u}_1^{(j)} \underbrace{0 \cdots 0}_{9-i} \underline{u}_2^{(j)} \ \underline{u}_3^{(j)} \ \underline{u}_4^{(j)}]$ and $J_{13-i} = I_9 \oplus -I_{4-i}$. In this way $\underline{H}_{ij}$ acts only on a vector of 4 entries in a column of $13-i$ entries. These vectors will be called partial vectors. Making the proper choices for the $\underline{H}_{ij}$, the matrix :

$$\begin{aligned}
\Phi = &(I_9 \oplus \underline{H}_{91} \oplus I_7) \\
&(I_8 \oplus \underline{H}_{81} \oplus I_7) \\
&(I_7 \oplus \underline{H}_{71} \oplus I_7) \\
&(I_9 \oplus \underline{H}_{62} \oplus I_4)(I_6 \oplus \underline{H}_{61} \oplus I_7) \\
&(I_8 \oplus \underline{H}_{52} \oplus I_4)(I_5 \oplus \underline{H}_{51} \oplus I_7) \\
&(I_7 \oplus \underline{H}_{42} \oplus I_4)(I_4 \oplus \underline{H}_{41} \oplus I_7) \\
&(I_9 \oplus \underline{H}_{33} \oplus 1)(I_6 \oplus \underline{H}_{32} \oplus I_4)(I_3 \oplus \underline{H}_{31} \oplus I_7) \\
&(I_8 \oplus \underline{H}_{23} \oplus 1)(I_5 \oplus \underline{H}_{22} \oplus I_4)(I_2 \oplus \underline{H}_{21} \oplus I_7) \\
&(I_7 \oplus \underline{H}_{13} \oplus 1)(I_4 \oplus \underline{H}_{12} \oplus I_4)(1 \oplus \underline{H}_{11} \oplus I_7),
\end{aligned} \tag{4.17}$$

will eliminate the non-zero entries of Y^t in the product ΦM. Notice that the transformations in a row of Φ can be applied in parallel to the matrix M. Application of the *ith* row of Φ will be referred to as the *ith* sweep.

In order to formulate the computation of the product ΦM in terms of a regular recurrent algorithm, we set up the following scheme of operating on the columns of the matrix M. Let us consider the first sweep. Denote the partial vector on which $\underline{H}_{12}$ (i=1, j=2) acts, by $\mathbf{p}_{12k} = [e_{12k} \ \mathbf{v}_{12k}^t]^t$, where e_{12k} denotes the first entry and $\mathbf{v}_{12k}$ is a 3×1 vector which denotes the remaining entries of the partial vector. The first index denotes the current sweep, the second index denotes the partial vector and the third index denotes column k, $k=i+3(j-1)+1,...,20$. Applying $\underline{H}_{12}$ to partial vector $\mathbf{p}_{12k}$ results in a partial vector $\underline{H}_{12}\mathbf{p}_{12k}$. The first entry of this vector will be the first element of the first

partial vector in column k to which $\underline{H}_{41}$ ($4=i+3$, $1=j-1$) will be applied (that is, in the fourth sweep). Denote this entry by e_{41k}. The remainder of the partial vector $\underline{H}_{12}\mathbf{p}_{12k}$ will be used in the second sweep and is denoted by the 3×1 vector $\mathbf{v}_{22k}$. Denoting the last three elements of the partial vector to which $\underline{H}_{41}$ is applied, by the 3×1 vector $\mathbf{v}_{41k}$, this partial vector is denoted by $[e_{41k}\ \mathbf{v}_{41k}^t]^t$. Similarly for the rest of the transformations in (4.17). In order to express the application of a transformation $\underline{H}_{ij}$ to all of the columns of M, we denote the reflection vector $\underline{\mathbf{u}}_{ij}$ by $\underline{\mathbf{u}}_{ijk}$ - where the indices have the same interpretations as above - and copy it to $\underline{\mathbf{u}}_{i,j,k+1}$, when $\underline{H}_{ij}$ must be applied to the partial vector in column $k+1$ after it has been applied to the partial vector in column k. However, since a reflection vector $\underline{\mathbf{u}}_{ij}$ has to be computed first from the partial vector in column $k=i+3(j-1)+1$, $j=1,...,3-\lfloor(i-1)/3\rfloor$, before it can be applied to subsequent columns, we introduce a control variable c_{ijk} which is initialized to 0 for $k=i+3(j-1)+1$, $j=1,...,3-\lfloor(i-1)/3\rfloor$, and to 1 every where else. Whenever $c_{ijk}=0$ a reflection vector $\underline{\mathbf{u}}_{ijk}$ is computed. Else the computed reflection vector is used to reflect the partial vector. The value of c_{ijk} is copied to $c_{i+1,j,k+1}$, because the partial vectors in columns $k=i+3(j-1)+2$, $j=1,...,3-\lfloor(i-1)/3\rfloor$, in sweep $i+1$, will be next eliminated. The regular recurrent algorithm for this scheme of implementing the computation of the product ΘM is given in Figure 4.3. The function $f()$ computes the reflection vectors $\underline{\mathbf{u}}_{ijk}$ according to (4.15.b). The function $g()$ computes $\pm\|[e_{ijk}\ \mathbf{v}_{ijk}^t]^t\|_J\mathbf{e}_1$ (see (4.15.a). These computations are done in case the control variable c_{ijk}=0. In case c_{ijk}=1, a reflection is computed by the function $h()$ with the computed reflection vector $\underline{\mathbf{u}}_{ijk}$.

The set of displacement vectors for the variables $\mathbf{u}_{ijk}$, $\mathbf{v}_{ijk}$, c_{ijk} and e_{ijk} is given by: $D_{SC}=\{[0\ 0\ 1]^t,\ [1\ 0\ 0]^t,\ [1\ 0\ 1]^t,\ [3\ -1\ 0]^t\}$. The index set I^3_{SC} of the algorithm is: $I^3_{SC}=\{(i,j,k)\mid 1\le i\le 9,\ 1\le j\le 3-\lfloor(i-1)/3\rfloor,\ i+3(j-1)\le k\le 19\}$. The corresponding dependency graph is shown in Figure 4.4, for k only until 9 to avoid a too complex graph. The location of initialization data in the graph and the location of output data follows immediately from the algorithm in Figure 4.3.

/* initializations */

$c_{ijk}, e_{ijk} \in \mathbf{R}$; $\mathbf{u}_{ijk} \in \mathbf{R}^4$; $\mathbf{v}_{ijk} \in \mathbb{R}^3$;

$e_{ijk} = p_{i+m(j),k}$; $i=1,2,3$; $j=1,2,3$;

$k=i+m(j),...,19$;

$v_{1jk} = [q_{m(j),k} \; q_{m(j)+1,k} \; q_{m(j)+2,k}]^t$;

$j=1,2,3$; $k=2+3(j-1),...,20$;

with $m(j) = m(j-1) + 3, m(0) = -2$;

$c_{1jk} = 0$ for $k=2+3(j-1)$, $j=1,2,3$; $c_{1jk} = 1$ everywhere else;

/* transformations */

for $i=1$ **to** 9

for $j=1$ **to** $3-\lfloor(i-1)/3\rfloor$

for $k=i+3(j-1)+1$ **to** 20

case c_{ijk}

$$0: \quad \underline{\mathbf{u}}_{i,j,k+1} = f\left(\begin{bmatrix} e_{ijk} \\ \mathbf{v}_{ijk} \end{bmatrix}\right);$$

$$\begin{bmatrix} e_{i+3,j-1,k} \\ \mathbf{v}_{i+1,j,k} \end{bmatrix} = g\left(\begin{bmatrix} e_{ijk} \\ \mathbf{v}_{ijk} \end{bmatrix}\right);$$

$$1: \quad \underline{\mathbf{u}}_{i,j,k+1} = \underline{\mathbf{u}}_{ijk};$$

$$\begin{bmatrix} e_{i+3,j-1,k} \\ \mathbf{v}_{i+1,j,k} \end{bmatrix} = h\left(\begin{bmatrix} e_{ijk} \\ \mathbf{v}_{ijk} \end{bmatrix}, \underline{\mathbf{u}}_{ijk}\right);$$

endcase

$c_{i+1,j,k+1} = c_{ijk}$;

endfor

endfor

endfor

Figure 4.3. A regular recurrent algorithm for the SC solver.

4.3 FULL SIZE SYSTOLIC ARRAYS FOR THE QR AND SC SOLVER

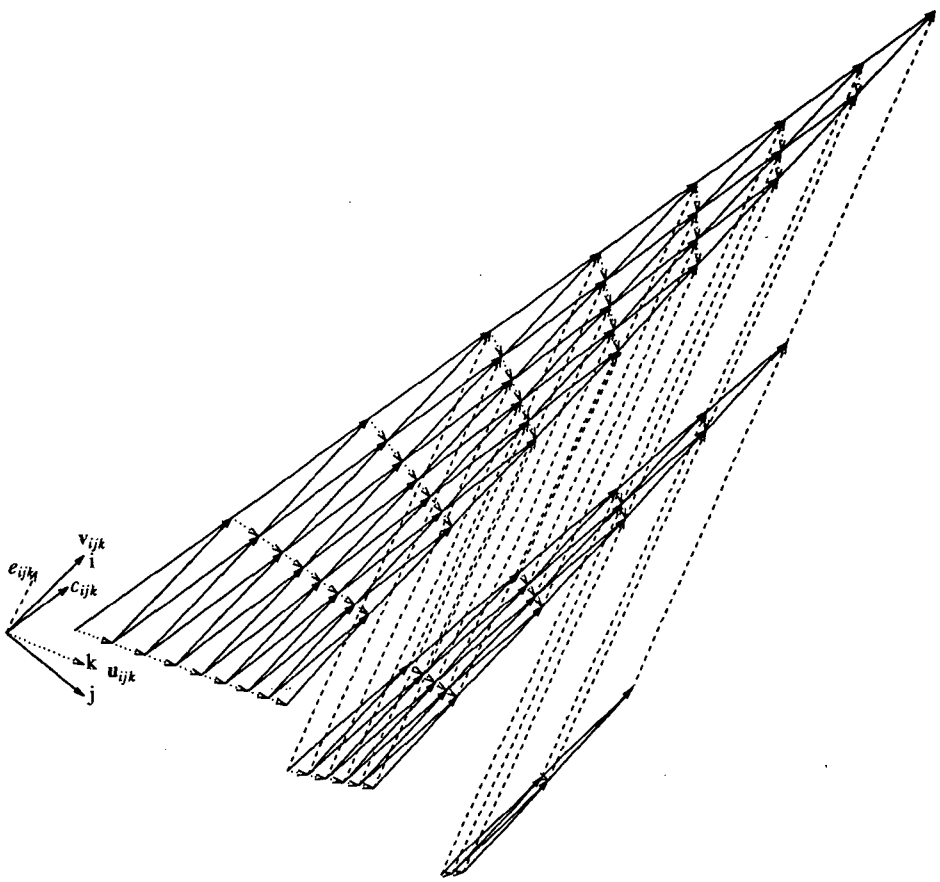


Figure 4.4. Dependency graph of the algorithm in Figure 4.3.

4.3.1 DESIGN OF AN FSA FOR THE HOUSEHOLDER QR SOLVER

From the dependency graph $G=\{I^3_{QR}, D_{QR}\}$, with :

$$I^3_{QR} = \{\mathbf{i} = [i\ j\ k\ ]^t \mid 1 \le i \le 8,\ \lceil i/3 \rceil \le j \le 3,\ i \le k \le 17\} \qquad (4.18.a)$$

and

$$D_{QR} = \{[1\ 0\ 0]^t, [0\ 1\ 0]^t, [1\ 0\ 1]^t, [0\ 0\ 1]^t\}, \tag{4.18.b}$$

a full size systolic array can be synthesized by the procedure given in Section 1.2. Let the schedule vector $\mathbf{s}$ be $\mathbf{s} = [1\ 1\ 1]^t$ and the projection vector $\mathbf{t} = [0\ 0\ 1]^t$. Choose a transformation matrix T :

$$T = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$$

(such that $T\mathbf{t} = 0$). As a matter of convenience we translate the dependency graph G to the origin (0,0,0) by the vector $[-1\ -1\ -1]^t$. The set of processor elements in the FSA is then given by :

$$I^2_{QR} = \{\bar{\mathbf{i}} \mid \bar{\mathbf{i}} = T(\mathbf{i} - [1\ 1\ 1]^t), \mathbf{i} \in I^3_{QR}\}. \tag{4.19}$$

The set of interconnections in the FSA is given by :

$$\bar{D}_{QR} = \{T\mathbf{d} \mid \mathbf{d} \in D_{QR}\} = \{[1\ 0]^t, [0\ 1]^t, [0\ 0]^t\}. \tag{4.20}$$

The set of schedule time steps for a processor element $\bar{\mathbf{i}} \in I^2_{QR}$ is given by :

$$S_{\bar{\mathbf{i}}} = \{\mathbf{s}^t(\mathbf{i} - [1\ 1\ 1]^t) \mid \mathbf{i} \in I^3_{QR} \text{ and } T(\mathbf{i} - [1\ 1\ 1]^t) = \bar{\mathbf{i}}\}. \tag{4.21}$$

The number of delay units associated with the different interconnections in the FSA is given by :

$$\mathbf{s}^t \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} = 1,\ \mathbf{s}^t \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} = 1,\ \mathbf{s}^t \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} = 2,\ \mathbf{s}^t \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} = 1.$$

The FSA is shown in Figure 4.5.

Many different systolic arrays can be deduced from the dependency graph, but we shall focus on this particular one, because it satisfies the constraint that all I/O communication with peripherals is handled at the boundaries.

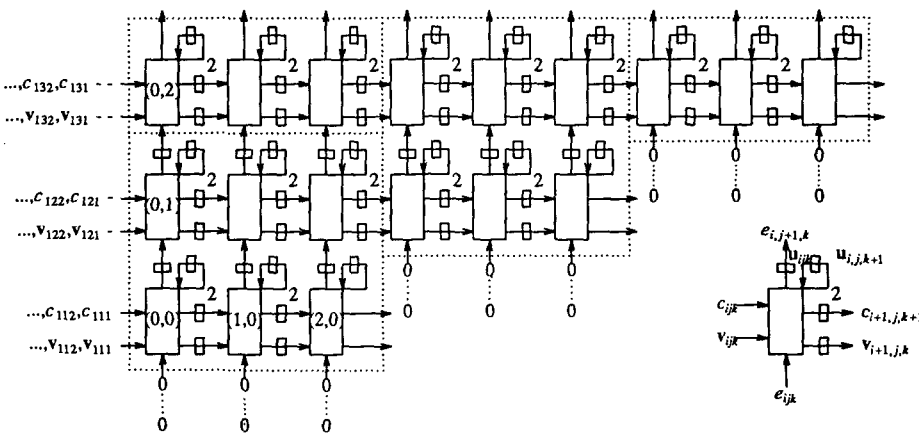


Figure 4.5. A full size systolic array for the QR solver.

4.3.2 DESIGN OF AN FSA FOR THE HYPERBOLIC HOUSEHOLDER SC SOLVER

From the dependency graph $G=\{I^3_{SC}, D_{QR}\}$, with :

$$I^3_{SC} = \{\mathbf{i} = [i\ j\ k]^t \mid 1 \le i \le 9,\ 1 \le j \le 3 - \lfloor (i-1)/3 \rfloor,\ i + 3(j-1) \le k \le 20\} \qquad (4.22.a)$$

and

$$D_{SC} = \{[1\ 0\ 0]^t, [3\ -1\ 0]^t, [1\ 0\ 1]^t, [0\ 0\ 1]^t\}, \qquad (4.22.a)$$

a full size systolic array for the SC solver can be synthesized by the procedure given in Section 1.2. Let the schedule vector s be $s = [1\ -3\ 1]^t$ and the projection vector $\mathbf{t} = [0\ 0\ 1]^t$. Choose a transformation matrix T :

$$T = \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \end{bmatrix}$$

(such that $T\mathbf{t} = 0$). As a matter of convenience we translate the dependency graph G to the origin (0,0,0) by the vector $[-1\ -1\ -2]^t$. The set of processor elements in the FSA is

then given by :

$$I^2_{SC} = \{\bar{\mathbf{i}} \mid \bar{\mathbf{i}} = T(\mathbf{i} - [1\ 1\ 2]^t),\ \mathbf{i} \in I^3_{SC}\}. \tag{4.23}$$

The set of interconnections in the FSA is given by :

$$\bar{D}_{SC} = \{T\mathbf{d} \mid \mathbf{d} \in D_{SC}\} = \{[1\ 0]^t, [3\ 1]^t, [0\ 0]^t\}. \tag{4.24}$$

The set of schedule time steps for a processor element $\bar{\mathbf{i}} \in I^2_{SC}$ is given by :

$$S_{\bar{\mathbf{i}}} = \{\mathbf{s}^t(\mathbf{i} - [1\ 1\ 2]^t) \mid \mathbf{i} \in I^3_{SC} \text{ and } T(\mathbf{i} - [1\ 1\ 2]^t) = \bar{\mathbf{i}}\}. \tag{4.25}$$

The number of delay units associated with the different interconnections in the FSA is given by :

$$\mathbf{s}^t \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} = 1,\ \mathbf{s}^t \begin{bmatrix} 3 \\ -1 \\ 0 \end{bmatrix} = 6,\ \mathbf{s}^t \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} = 2,\ \mathbf{s}^t \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} = 1.$$

Notice that we can apply a transformation :

$$K = \begin{bmatrix} 1 & -3 \\ 0 & 1 \end{bmatrix}$$

to the elements of I^2_{SC} and the interconnections in $\bar{D}_{SC}$, so that the FSA has an orthogonal 2-D mesh of interconnections. I.e., $K[1\ 0\ ]^t = [1\ 0]^t$ and $K[3\ 1]^t = [0\ 1]^t$. The result is shown in Figure 4.6. Again there exist many other possible arrays, but we chose this particular one, because all I/O with peripherals is handled at the boundaries of the array.

4.4 FIXED SIZE SYSTOLIC ARRAYS FOR THE QR AND SC SOLVER

Problems of which the size exceeds that of the systolic array on which they have to be executed, must be partitioned. Here we shall demand that the partitioning preserves the property that all I/O with peripherals is handled at the boundaries of the systolic array. To achieve problem size independency of local memory, the full size arrays of the

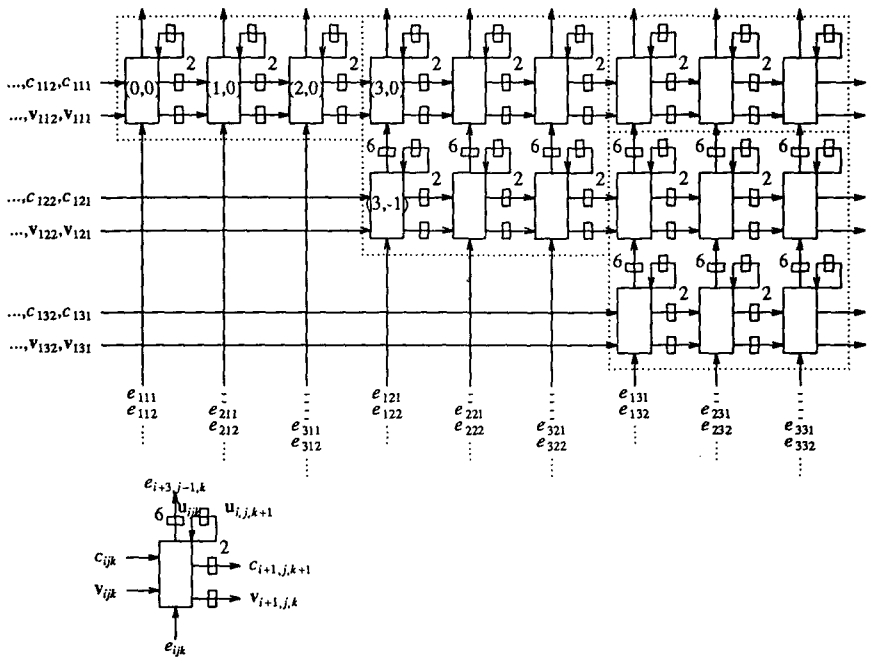


Figure 4.6. A full size systolic array for the SC solver.

previous section are partitioned by employing the LPGP partitioning strategy of Chapter 3. Summarizing, this strategy tessellates the array in tiles of, say, p processor elements each and schedules the computations of the tiles in pipeline on the reduced size array (RSA). An important property of this partitioning strategy is that all memory, required to store intermediate results, is kept outside the array. Thus, preventing local storage to depend on the size of the problem.

The different steps that are involved in the LPGP partitioning strategy of Chapter 3 are the following.

1. Tessellate the full size array into congruent tiles of, say, p processor elements each (if necessary, dummy processor elements are added such that each tile has exactly p processor elements).

2. Find an admissable ordering of the tiles. This ordering dictates in which order the computations of the different tiles are pipelined on the RSA.

3. Identify the set of processor elements which receive data from a particular tile. By scanning the interconnection pattern of these processors with the processor elements in this tile, the routing paths from the outputs to the inputs of the FSA are established.

4. Compute schedules for the read and write signals of the buffers of the RSA.

4.4.1 TESSELLATION OF THE FSAs FOR THE QR AND SC SOLVER

THE QR SOLVER

The first step in the LPGP partitioning procedure is the tessellation of the full size array. A possible tessellation of the FSA in Figure 4.5 is envisualized by the dotted lines in this figure. The RSA consists of 6 processor elements. According to Section 3.1 this tessellation is characterized as follows :

$$\mathbf{r} = 0$$

$$B = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \quad \mathrm{F} = \{[0\,0]^t, [1\,0]^t, [2\,0]^t, [0\,1]^t, [1\,1]^t, [2\,1]^t\}$$

$$U = \begin{bmatrix} 3 & 0 \\ 1 & 2 \end{bmatrix}, \quad \mathrm{H} = \{[0\,0]^t, [1\,0]^t, [2\,0]^t, [0\,1]^t\}.$$

Although many different tessellations exist, this particular one has the advantage that all inputs of the RSA will be at its boundaries. Notice that not all tiles consist of a sub-array of 6 processor elements. Dummy processor elements must be added to these tiles. Either method 1 or method 2 of Section 3.2 can be used to control the functionality of the processor elements in the RSA. We select method 2 here, because it does not enhance the

complexity of the interconnections in the RSA.

From Figure 4.5 we construct a dependency graph, which expresses the dependencies between tiles, as they are implied by the interconnections in the full size array. This graph, G={V,E}, is shown in Figure 4.7, where a vertex represents a tile.

The set of vertices is given by V = {(0,0), (0,2), (3,1), (6,2)} , where the vertex coordinates correspond to the coordinates of the processor element at the lower left corner of the corresponding tile. The set of edges (dependencies) is given by E = {[3 1]^t, [0 2]^t, [3 −1]^t}.

As was explained in Section 3.2, an order vector **p** is computed which obeys the following set of constraints :

$$\mathbf{p}^t\mathbf{e} \geq 1, \text{ for all } \mathbf{e} \in E \tag{4.26.a}$$

$$\mathbf{p}^t\mathbf{v} \neq \mathbf{p}^t\mathbf{w}, \text{ for all } \mathbf{v}, \mathbf{w} \in V \text{ and } \mathbf{v} \neq \mathbf{w}. \tag{4.26.b}$$

There are many solutions for the above set of constraints. However, we choose $\mathbf{p} = [6\ 1]^t$, since it results in a minimum of storage for intermediate results at each output of the RSA. With this choice of order vector the tiles are ordered as follows : (0,0), (0,2), (3,1), (6,2).

THE SC SOLVER

Similarly, we derive for the FSA of the SC solver a tessellation, dependency graph and ordering of the tiles. The tessellation is envisualized by the dotted lines in Figure 4.6. Its characterization is as follows :

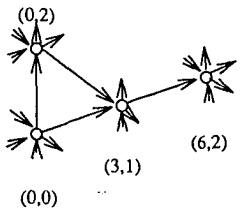


Figure 4.7. Dependency graph of the tessellated FSA of the QR solver.

$\mathbf{r} = 0$

$$B = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \quad F = \{[0\ 0]^t, [1\ 0]^t, [2\ 0]^t, [0\ 1]^t, [1\ 1]^t, [2\ 1]^t\}$$

$$U = \begin{bmatrix} 3 & 3 \\ -1 & 1 \end{bmatrix}, \quad H = \{[0\ 0]^t, [1\ 0]^t, [2\ 0]^t, [1\ 1]^t\}.$$

This tessellation has the advantage that all inputs of the RSA will be at its boundaries. Dummy processor elements must be added where necessary and method 2 of Section 3.2 is used to control the functionality of processor elements in the RSA.

The dependency graph, G={V,E}, of the tessellated FSA of the SC solver is depicted in Figure 4.8. For this graph we find : V = {(0,0), (3,−1), (6,−2), (6,0)} and E = {$[3\ -1]^t$, $[0\ 3]^t$, $[3\ 2]^t$}.

Out of all existing solutions for the order vector **p** of graph G, we choose $\mathbf{p}^t = [6\ \ 1]$, since it results in a minimum of storage for intermediate results at each output of the RSA. With this choice of order vector the tiles are ordered as follows : (0,0), (3,−1), (6,−2), (6,0).

Since the interconnection pattern is the same for both RSAs, only the algorithms for the controller in the model of the LPGP partitioned FSAs (see Figure 3.3) and the number of delays along an interconnection shall differ. Taking appropriate measures for this difference in delays, a single, programmable RSA can be designed to implement

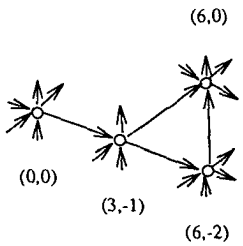


Figure 4.8. Dependency graph of the tessellated FSA of the SC solver.

both solvers.

4.4.2 PIPELINING THE COMPUTATIONS OF THE TILES ON THE RSA

The model for the LPGP partitioned FSAs of the QR and SC solvers is depicted in Figure 4.9. Interpretation of this model is given in Section 3.2. The delays numbered with x can be programmed to implement either 1 or 6 delay units, as required in the FSA for the QR and SC solver, respectively. The controller for the model is implemented by the algorithms given in Section 3.2 for the controller. The different parameters of these algorithms are in the case of the QR solver :

S_in_$buffer_1$ = {(0,0), (0,2)}

S_in_$buffer_2$ = {(0,0), (0,2), (6,2)}

S_in_$buffer_3$ = S_in_$buffer_4$ = S_in_$buffer_5$ = {(0,0), (3,1), (6,2)}

S_out_s_1 = {(0,2), (6,2)}

S_out_s_2 = {(0,0), (3,1), (6,2)}

S_out_s_3 = S_out_s_4 = S_out_s_5 = {(0,2), (3,1), (6,2)}

S_tiles = {(0,0), (0,2), (3,1), (6,2)}

S_I/O_(0,1) = S_I/O_(1,1) = S_I/O_(2,1) = {(0,2), (6,2)}

S_added = {(0,1), (1,1), (2,1)}

dependency_1 = $[3\ 1]^t$

dependency_2 = $[3\ -1]^t$

dependency_3 = dependency_4 = dependency_5 = $[0\ 2]^t$

and in the case of the SC solver :

S_in_$buffer_1$ = {(0,0), (3,−1), (6,−2)}

S_in_$buffer_2$ = {(0,0), (6,0)}

S_in_$buffer_3$ = S_in_$buffer_4$ = S_in_$buffer_5$ = {(0,0), (3,−1), (6,−2)}

S_out_s_1 = {(0,0), (6,−2), (6,0)}

S_out_s_2 = {(6,−2), (6,0)}

S_out_s_3 = S_out_s_4 = S_out_s_5 = {(0,0), (3,−1), (6,0)}

S_tiles = {(0,0), (3,−1), (6,−2), (6,0)}

S_I/O_(0,1) = S_I/O_(1,1) = S_I/O_(2,1) = {(0,0), (6,0)}

S_added = {(0,1), (1,1), (2,1)}

dependency_1 = $[3\ 1]^t$

dependency_2 = $[3\ -1]^t$

dependency_3 = dependency_4 = dependency_5 = $[0\ 2]^t$.

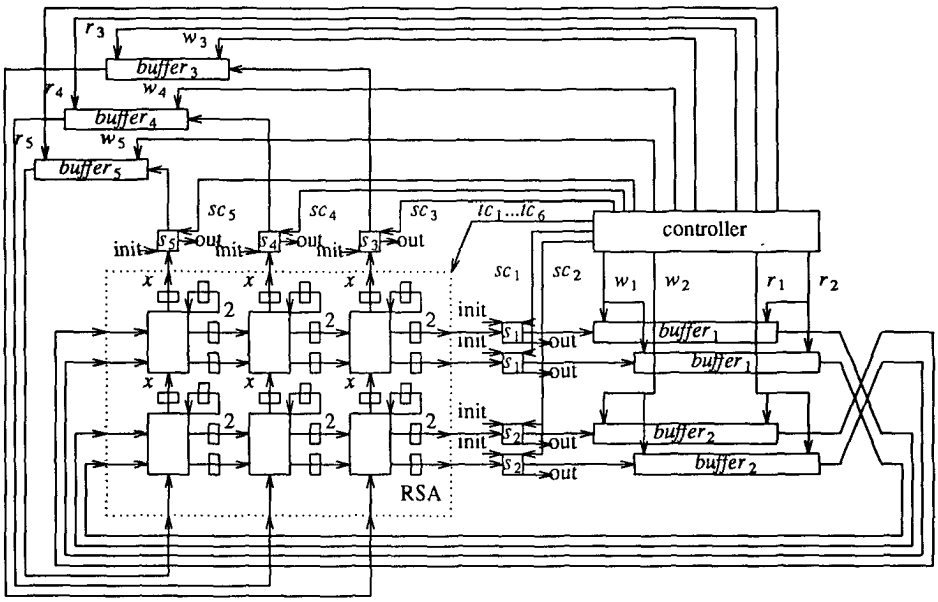


Figure 4.9. LPGP partitioned array model for a 2×3 RSA for the QR and SC solver.

4.5 DESIGN OF A HOUSEHOLDER PROCESSOR ELEMENT

In this section we address the implementation of a processor element in the RSA of Figure 4.9. We shall focus on an *innerproduct-step* approach for the implementation of the orthogonal and hyperbolic Householder transformations. An innerproduct-step implementation serializes computations involved in the application of a Householder transformation to a vector, so that we are able to satisfy the constraint of being flexible with respect to communication bandwidth requirements of peripherals, attached to the I/O processors of the array (see constraint 4 in Section 4.1).

4.5.1 IDENTIFYING, MINIMIZING AND SCHEDULING OF COMPUTATIONS

A Householder transformation is relatively complex. The computations that are involved in orthogonal and hyperbolic Householder transformations are summarized in Table 4.1.

The number of computations for the calculation of β and β_J can be minimized by writing $\beta = 1/|\,|\mathbf{x}|\,|_2\,|u_1|$ and $\beta_J = 1/|\,|\mathbf{x}|\,|_J\,|\underline{u}_1|$. The number of multiply-add operations that are required for vectoring and reflection for the orthogonal case, are given in Table 4.2. The table is similar for the hyperbolic case. All vectors are assumed to be of size n and all results in the table are assumed to be obtained by one or more multiply-add operations.

Since the RSA is timed systolically (synchronous), the computation of a reflection vector should take as much time as the computation of a reflection. Hence, from Table 4.2 we deduce the following condition :

$$n = n_i + n_s + 1, \tag{4.27}$$

where n_i and n_s are the number of multiply-add steps required for the computation of an inverse and a square root, respectively. Since n is the number of entries in the data vectors, (4.27) determines the size of the partial vectors $[\mathbf{v}_{ijk}^t\ e_{ijk}]^t$ and $[e_{ijk}\ \mathbf{v}_{ijk}^t]^t$ in Figure 4.1 and 4.3, respectively, once the numbers n_i and n_s are fixed. Moreover, the choice of n_i and n_s will fix the minimum amount of local memory for a processor element. This memory is needed to store the reflection vector and the data vector in case of vectorizing and reflection, respectively. It is also needed to store Taylor series coefficients for the calculation of $f(x) = x^{1/2}$ and $g(x) = x^{-1}$, as we shall see in the sequel.

Vectoring		Reflection	
Orthogonal	Hyperbolic	Orthogonal	Hyperbolic
$\|\|\mathbf{x}\|\|_2$	$\|\|\mathbf{x}\|\|_J$	$\mathbf{u}^t y$	$\underline{\mathbf{u}}^t y$
$\mathbf{u} = \mathbf{x} + sign(x_1)\|\|\mathbf{x}\|\|_2 \mathbf{e}_1$	$\underline{\mathbf{u}} = \mathbf{x} + sign(x_1)\|\|\mathbf{x}\|\|_J \mathbf{e}_1$	$y_i - \beta(\mathbf{u}^t y)u_i$	$\pm y_i - \beta_J(\underline{\mathbf{u}}^t y)\underline{u}_i$
$\beta = \frac{2}{\|\|\mathbf{u}\|\|_2^2}$	$\beta_J = \frac{2}{\|\|\underline{\mathbf{u}}\|\|_J^2}$		

TABLE 4.1. Summary of orthogonal and hyperbolic Householder computations.

4.5.2 COMPUTATION OF SQUARE ROOTS AND INVERSES

The square root and inverse calculations that are required to compute β and $\|\|\mathbf{x}\|\|_2$, can be easily formulated in terms of innerproduct-step algorithms. The number of steps in such algorithms will fix the quantities n_s and n_i, respectively. Square roots and inverses may be computed from a Taylor series expansion [Arde1963] by an approximating polynomial :

$$\hat{f}(x;z) = \sum_{i=0}^{n} b_i(z)(x-z)^i = (b_0(z) + (x-z)(b_1(z) + (x-z)(\ldots(b_n(z))\ldots))), \tag{4.28}$$

where the $b_i(z) = \frac{f^{(i)}(z)}{i!}$ are the coefficients of the polynomial in the point z and $f^{(i)}$ denotes the *ith* derivative of the function f. The approximation error E_{n+1} is known to be [Arde1963] :

$$E_{n+1} = \frac{f^{(n+1)}(\xi)}{(n+1)!}(x-z)^{(n+1)}, \quad \xi = z + \theta(x-z),\ 0 \le \theta \le 1.$$

The right hand side of (4.28) expresses the computation of $\hat{f}(x;z)$ in terms of an innerproduct-step algorithm. Although there are better (faster) algorithms for computing inverses and square roots, we choose a Taylor series expansion because then we will not need special hardware besides an innerproduct-step processor to perform all the different computations that are required for the application of a Householder transformation to a vector. With this in mind the Householder processor element can be designed cost-

Vectoring		Reflection	
Computation	multiply-add's	Computation	multiply-add's
$\|\|\mathbf{x}\|\|_2^2$	n	$\mathbf{u}^t y$	n
$\|\|\mathbf{x}\|\|_2$	n_s	$\beta(\mathbf{u}^t y)$	1
$u_1 = x_1 + sign(x_1)\|\|\mathbf{x}\|\|$	1	$\mathbf{y} - (\beta \mathbf{u}^t \mathbf{y})\mathbf{u}$	n
$u_1 \|\|\mathbf{x}\|\|_2$	1		
$\beta = \frac{1}{\|\|\mathbf{x}\|\|_2 \|u_1\|}$	n_i		

TABLE 4.2. Listing of computations and the numbers of multiply-add operations.

effectively and compactly.

Let a number x be represented as a normalized floating point number in the set :

$$R_f(l_m, l_e) = \{0\} \cup \{x = \sigma_m x_m \times 2^{\sigma_e x_e} \mid \sigma_m, \sigma_e \in \{+,-\}, x_m = \sum_{i=1}^{l_m} x_m[i]2^{-i}, x_m[i] \in \{0,1\},$$

$$x_{m[1]} = 1, x_e = \sum_{i=0}^{l_e} x_e[i]2^i, x_e \in \{0,1\}\}.$$

The elements of this set have the property that $x_m \in [0.5, 1)$. A unique representation for 0 is chosen to be the following : $\sigma_0 = +$, $0_m[i] = 0$, $i = 1,...,l_m$, $\sigma_e = -$ and $0_e = \sum_{i=0}^{l_e} 2^i$. In case of computing the square root of x, the exponent x_e is divided by 2. Hence, x_e has to be even. In case x_e is odd, it must be first incremented by 1 and the mantissa x_m must be divided by 2. In that case the mantissa is denormalized and therefore we choose the interval [0.25, 1) in which Taylor series expansions are computed for $f(x) = x^{1/2}$. For simplicity of implementation we also choose this interval in which the Taylor series expansions are computed for $g(x) = x^{-1}$. The interval [0.25, 1) is divided in sub intervals of length Δ and the points $z_k = (k\Delta + 0.25)$ $(k = 0,1,...)$ become the points around which the Taylor series expansions are computed. Of all points z_0, z_1,... in the interval [0.25, 1) the point z_k is selected as the one closest to x_m. Hence, z_k is computed as follows :

$$k = \left\lfloor \frac{x-0.25}{\Delta} + 0.5 \right\rfloor,$$

$$z_k = k\Delta + 0.25.$$

A simple algorithm for the computation of the square root of a number $x = \sigma_m x_m \times 2^{\sigma_e x_e}$ is the following.

sqrt (x_e, x_m)
  if (x_e is odd)
    $x_e \leftarrow x_e + 1$;
    $x_m \leftarrow x_m \times 2^{-1}$;
  endif
  $x_e \leftarrow x_e/2$;
  $k \leftarrow \left\lfloor \frac{x_m - 0.25}{\Delta} + 0.5 \right\rfloor$;
  $z_k \leftarrow k\Delta + 0.25$;
  $x_m \leftarrow \sum_{i=0}^{n} b_i(z_k)(x_m - z_k)^i$;
endsqrt

An algorithm for the computation of x^{-1} is similar to the one above, except that no checking is needed for the exponent. The only operation that is needed for the exponent is the inversion of its sign bit σ_e. Storing the coefficients $b_i(z_k)$ in tables, it is easy to compute $x^{1/2}$ or x^{-1} within desired accuracy. The computation of index k for z_k is at the same time the address computation of the entries of the tables. Apart from the coefficients $b_i(z_k)$ the points $-z_k$ may be computed in advance and also stored in the tables. Thus, the *kth* entry in the tables for $x^{1/2}$ and x^{-1} looks like :

$-z_k$	$b_n(z_k)$	....	$b_0(z_k)$

The parameters n_i and n_s are functions of Δ, the length of the approximation interval. In an actual implementation we have to seek for the optimum of the values n_i, n_s and the sizes of the tables for $x^{1/2}$ and x^{-1}. Preferably Δ is chosen to be a fixed point number, in order to simplify the computation of k.

4.5.3 ARCHITECTURE OF A HOUSEHOLDER PROCESSOR ELEMENT

Expressing the computations of β (β_J) and $||x||_2$-($||x||_J$) in Table 4.2 in terms of Taylor series expansions of the form (4.28), we are able to schedule all computations in this table on a micro-programmed innerproduct-step processor. The architecture of this processor is given in Figure 4.10. In this architecture it is assumed that the constants Δ^{-1} and $0.5 - 0.25\Delta^{-1}$ are available as precomputed numbers. The different components of the architecture are the following.

BUSSES

The processor element has a double-bus architecture and is optimized for computation of Householder transformations in a minimum number of instruction cycles. Access to the busses is obtained via tristate buffers, denoted by the triangles. Internally numbers are represented as a mantissa of r bits and an exponent of s bits wide. The parameters r and s are left to be made explicit in the final stage of the design, where the floorplan and the layout of the Householder chip are defined.

LOCAL MEMORY

The two tables **TABLE1** and **TABLE2** are read only memories (ROMs), which contain the coefficients of the Taylor series expansions of the functions $f(x) = x^{1/2}$ and

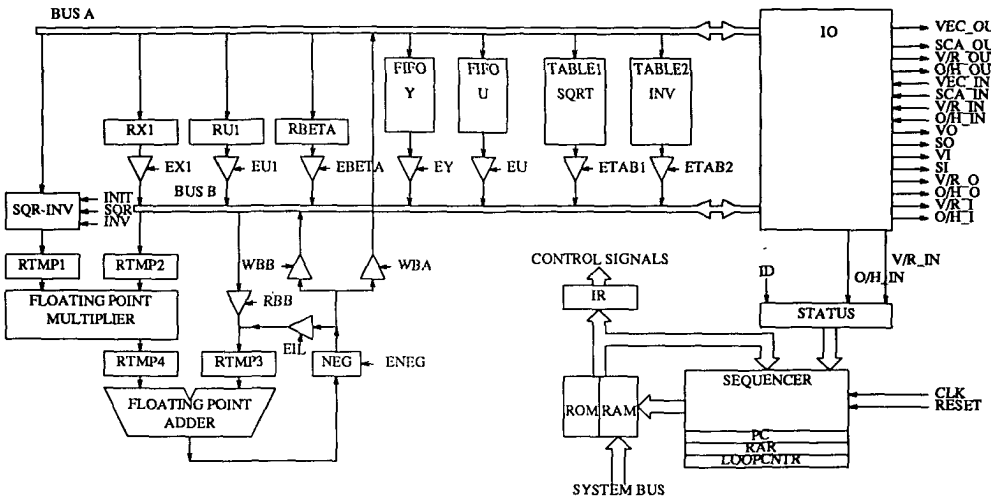


Figure 4.10. Architecture of the Householder processor.

$g(x) = x^{-1}$, respectively. In case of reflection the input data vector **y** is stored in the first-in-first-out buffer **FIFO-Y**. In case of vectorizing, the last $(n-1)$ entries of the input data vector are stored in the first-in-first-out buffer **FIFO-U**. The first entry u_1 of the reflection vector parameter is computed after several steps and is stored separately from the other entries in the register **RU1**. The first entry of the input vector is stored in the register **RX1**. In this way, when the processor has to perform a vectorizing operation on an input vector **x**, it can keep the first entry x_1 of this vector for future computations, such as for calculation of u_1 in Table 4.2. The scaling constant β, is computed last and stored in the register **RBETA**. The feed back loop from the output of the floating point adder to one of its inputs is meant to speed up the accumulation part of the innerproduct computations. Without this loop the output of the adder has to go via bus **A** or **B**, leading to a delay of one clock cycle per accumulation step, since registers **RTMP1** and **RTMP2** are also loaded via the same busses.

MICRO CONTROLLER

The microprograms for the Householder transformations are contained in the micro-instruction memory, which consist of a read only (**ROM**) and random access (**RAM**) part. In the **ROM** the sub-routines for the square root and inverse are stored. In the **RAM** the microprograms are stored which perform either orthogonal or hyperbolic transformations, using the sub-routines in the ROM. These programs are loaded into the **RAM** via the system bus of the host computer. The micro-instructions are loaded into the instruction register **IR**, containing the control signals for the data path of the processor element. The microprogram for the processor is found in [Hofw1988], together with the tables for the coefficients of the Taylor series expansions of the functions $f(x) = x^{1/2}$ and $g(x) = x^{-1}$.

The sequencer controls the execution of the microprograms by taking into account branch conditions and loops encountered in the computations listed in Table 4.2. Its main components are the program counter **PC**, the return address register **RAR** and the loop counter **LPCNTR**. The program counter computes the next microprogram address. The return address register stores a return address in order to recover from a sub-routine call. The loop counter implements *repeat-until* control for the innerproduct-step calculations.

EXPONENT HANDELING

The box denoted by **SQR_INV** has the following functions. In case of square root computations it tests whether the exponent must be incremented by 1 and it divides the exponent by 2. If the test was successful it also divides the mantissa by 2. In case of the computation of an inverse it reverses the sign of the exponent. The exponent is stored for normalization of a computed result to floating point format.

INPUT/OUTPUT INTERFACE

The box denoted by **IO**, controls the flow of input data to and output data from the busses. It also controls the buffers at the inputs and outputs of a processor element (see Figure 4.5 and Figure 4.6). Via the control signal **VO** the processor element instructs an output buffer, that contains a vector $\mathbf{v}_{ijk}$, to receive a new vector $\mathbf{v}_{i+1,j,k}$ from the

processor element. The entries of this vector are transmitted one by one to the buffer via the output **VEC_OUT**. Via the control signal **SO** the processor element instructs an output buffer, that contains a scalar e_{ijk}, to receive a new scalar $e_{i,j+1,k}$ or $e_{i+3,j-1,k}$, from the processor element. This scalar is transmitted to the buffer via the output **SCA_OUT**. Via the control signal **VI** the processor element instructs an input buffer, that contains a data vector v_{ijk}, to transmit the entries of this vector one by one to the processor element. The processor element receives the entries at the input **VEC_IN**. Via the signal **SI** the processor element instructs an input buffer, that contains a scalar e_{ijk}, to transmit this scalar to the processor element. The processor element receives the scalar at the input **SCA_IN**. Via the signal **V/R_O** the processor element instructs an output buffer, that contains a control bit c_{ijk}, to receive a new control bit $c_{i+1,j,k+1}$. This control bit is transmitted at the output **V/R_OUT**. Via the signal **V/R_I** the processor element instructs an input buffer, that contains a control bit c_{ijk}, to transmit its content to the processor element. The control bit is received at the input **VR_IN**. Via the signal **O/H_I** the processor element instructs an input buffer to transmit a control bit, which determines whether the processor element performs an orthogonal or hyperbolic Householder transformation. The processor element receives this control bit at its input **O/H_IN**. Via the signal **O/H_O** the processor element instructs an output register to receive a control bit which determines whether a processor element performs an orthogonal or hyperbolic Householder transformation. This control bit is transmitted at the output **O/H_OUT**.

PROCESSOR ELEMENT STATUS

The register **STATUS** is the status register of the processor element. It contains signal values upon which decisions are based for branching in the micro-instruction programs. The control **ID**, signals whether the processor element has to perform an identity or default I/O map (Householder transformation). **V/R_IN** signals whether the processor element must perform computations for vectoring or reflection. **O/H_IN** signals whether the processor element must perform orthogonal or hyperbolic Householder transformations.

FLOATING POINT MULTIPLY-ADD SYSTEM

The multiplier and adder are designed to, respectively, multiply and add floating point numbers in $R_f(l_m, l_e)$. The result of a multiply-add operation is normalized and rounded, such that it is again in $R_f(l_m, l_e)$. The floating point multiply-add system takes care of computing the sum of exponents when two numbers are multiplied. It also shifts the mantissa of the smallest of two numbers which have to be added and makes the exponent of the smallest number equal to that of the largest one. And it normalizes and rounds the result of a multiply-add operation, such that it is an element of $R_f(l_m, l_e)$. Details of the implementation of the different parts of the architecture, the microprogram for the Householder transformations, tables for the coefficients of the Taylor series expansion of the functions $f(x) = x^{\frac{1}{2}}$ and $g(x) = x^{-1}$, as well as accuracy analyses are to be found in [Hofw1988]. A microprogram, written in a pseudo language, which describes the operation of the Householder processor element is listed in Appendix D.

In case of a vectorizing operation the processor element reads data from its inputs, only during the n steps for the computation of the square of the norm of input vector **x**. During the next steps the processor element does not request data from its buffered inputs. Hence, via the numbers n_s and n_i (see (4.27)) we can control the latency of the vectorizing operation and thus, the I/O bandwidth during vectorizing.

In case of the reflection operation, the processor element reads input data from its buffered inputs, only during the computation of the inner-product $\mathbf{u}^t\mathbf{y}$. During the next $n + 1$ steps no data is requested from the input buffers. But, the number of these steps is related to n_s and n_i via (4.27). Hence, by means of this equation we can control the I/O bandwidth of the processor elements of the array at design time.

Fitting the scheduling of the computations in Table 4.2 in the scheduling of the RSA in Figure 4.9 is not hard to do, considering the following interpretation of the schedule time steps in the RSA : *all relevant input data must be available at the inputs of a processor element at time step t_i and all relevant output data must be available at the inputs of a connected processor element at time step t_{i+1}*. Hence, we are free to schedule the

sub expressions of the Householder transformations, as long as we meet the constraints imposed by the scheduling of the RSA. Since all outputs of a processor element in the RSA are buffered, a processor element can store its output in the buffers during the last n steps of a reflection (see Table 4.2). Notice that it is not necessary to output data during vectorizing. In this way we are certain to provide all input data from time step t_i to time step t_{i+1}.

5. CONCLUSIONS

The power of the feed forward direct methods of Section 2.2 is their high degree of parallelism and pipelining. This allows fast execution on systolic arrays, despite the fact that the amount of input data is nearly doubled due to the need for processing an identity matrix in (2.18) and (2.22) and a block identity matrix in (2.46). Both systolic arrays in Figure 2.1 and Figure 2.2 are roughly two times faster than the combination of the arrays in Figure 1.1(a) and (b). This combination requires $7N-5$ time steps to solve $A\mathbf{x}=\mathbf{b}$ ($A \in \mathbf{R}^{N\times N}$) as was explained in the introduction. A generalization of the systolic array in Figure 2.1 shows that $A\mathbf{x}=\mathbf{b}$ ($A \in \mathbf{R}^{N\times N}$) is solved in $4N$ time steps on this array, and a generalization of the systolic array in Figure 2.2 shows that $A\mathbf{x}=\mathbf{b}$ is solved in $3N+1$ time steps. The systolic arrays for the methods of Section 2.2 are simple and with complexity in the order of the complexity of a QR factorization. The feed forward method which uses only Givens rotations (see (2.23)) is a numerically stable and robust method for the complete class of non singular systems of linear equations. Moreover, the simple extensions given in Section 2.5 show that systolic arrays can be synthesized to compute expressions of the form $CA^{-1}B+D$. Thus making it possible to compute a wide spectrum of expressions by choosing appropriate values and dimensions for the matrices A, B, C and D.

The partitioning theory developed in Chapter 3 takes the synthesis of practical systolic arrays one step further into reality. The LPGP and the LSGP partitioning strategies are important in realizing systolic arrays of which the number of processor elements is independent of the size of the problem. By means of the LPGP partitioning of a full size array all memory for intermediate results is kept outside the systolic array. Therefore, the array can be designed once and for all, so that no modification in its architecture is

required when different sizes of a problem are executed on it. By means of the LSGP partitioning of a full size array the throughput of a processor element can be controlled, although at the expense of increased local memory. By controlling the throughput of the processor elements we control the I/O bandwidth of the systolic array and a close match can be found between the I/O bandwidths of the array and peripherals (such as disks or the host computer). Combining both strategies in the design of systolic arrays enables one to control I/O bandwidth as well as the number of processor elements. When added to a computer aided design environment for systolic arrays, such as SYSTARS [Omtz1987], these strategies can make a significant contribution to the synthesis of practical systolic arrays.

The design of a reduced size systolic array for two methods of the novel class has served to illustrate the practical use of the partitioning theory. Unfortunately, the choice of a Householder version of the methods has limited the versatility of the resulting array to solving systems of linear equations only. Generalizations in the sense of Section 2.5 are inefficiently realized with Householder transformations, since we have to separate the transformations for the computation of the factorization of A^t from the transformations on the vectors $\mathbf{b}_i^t$ (see Section 2.5). This is necessary in order to pass the result of the factorization of A^t unchanged to the transformations on all the $\mathbf{b}_i^t$. Such a scheme suffices with 2×2 transformations on the $\mathbf{b}_i^t$ and the result of the factorization of A^t. To implement these transformations with Householder processor elements which handle n×1 vectors (n significantly larger than 2) would be very inefficient. For the same reasons it is not efficient to implement Householder versions of (2.22) and (2.46) with a choice of, respectively, hyperbolic and orthogonal Householder transformations for the $\Theta_i(m)$.

APPENDIX A

Proposition 2.1 :

Let $L \in \mathbf{R}^{N\times N}$ *be a lower triangular matrix and* $\mathbf{b} \in \mathbf{R}^N$ *a vector, such that :*

$$LL^t - \mathbf{b}\mathbf{b}^t > 0. \tag{A.1}$$

Then, there exists a matrix $\Theta(-1)$ *as defined in (2.4.a) and (2.4.b) for* $m = -1$, *such that :*

$$\Theta(-1)\begin{bmatrix} L^t \\ -\mathbf{b}^t \end{bmatrix} = \begin{bmatrix} R(-1) \\ 0^t \end{bmatrix}, \tag{A.2}$$

where $R(-1)$ *is upper triangular.*

Proof :

The proof is by induction. Put $L_0 = L$ and $\mathbf{b}_0 = \mathbf{b} = [b_1^{(0)} \ \cdots \ b_N^{(0)}]^t$. Let $\mathbf{s}_0$ be the column vector

$$\mathbf{s}_0 = L_0^{-1}\mathbf{b}_0 = [s_1^{(0)} \ \cdots \ s_N^{(0)}]^t. \tag{A.3}$$

From (A.1) we derive that :

$$L_0 L_0^t - \mathbf{b}_0\mathbf{b}_0^t = L_0(I_N - \mathbf{s}_0\mathbf{s}_0^t)L_0^t > 0. \tag{A.4}$$

Hence,

$$I_N - \mathbf{s}_0\mathbf{s}_0^t > 0. \tag{A.5}$$

Since $s_1^{(0)} = \dfrac{b_1^{(0)}}{l_{11}^{(0)}}$, $(I_N - \mathbf{s}_0\mathbf{s}_0^t)$ is a matrix with its 11-entry equal to $(1 - (\dfrac{b_1^{(0)}}{l_{11}^{(0)}})^2) > 0$.

Hence, it follows that $\dfrac{|b_1^{(0)}|}{|l_{11}^{(0)}|} < 1$ and there exists a matrix $\Theta_1(-1)$ (see (2.4.b) for i=1),

with $\tanh(\alpha_1) = \dfrac{b_1^{(0)}}{l_{11}^{(0)}}$, such that

$$\begin{bmatrix} L_1^t \\ -\mathbf{b}_1^t \end{bmatrix} = \Theta_1(-1) \begin{bmatrix} L_0^t \\ -\mathbf{b}_0^t \end{bmatrix}, \tag{A.6}$$

where L_1 is a lower triangular matrix and $\mathbf{b}_1 = [0\ b_2^{(1)}\ \cdots\ b_N^{(1)}]^t$.

Now, suppose that there exists a $\Theta_{i-1}(-1)$, such that :

$$\begin{bmatrix} L_{i-1}^t \\ -\mathbf{b}_{i-1}^t \end{bmatrix} = \Theta_{i-1}(-1) \begin{bmatrix} L_{i-2}^t \\ -\mathbf{b}_{i-2}^t \end{bmatrix}, \tag{A.7}$$

with

$$\mathbf{b}_{i-1}^t = [\underset{i-1}{0 \ldots 0}\ b_i^{(i-1)}\ \cdots\ b_N^{(i-1)}] \tag{A.8.a}$$

and

$$\mathbf{b}_{i-2}^t = [\underset{i-2}{0 \ldots 0}\ b_{i-1}^{(i-2)}\ \cdots\ b_N^{(i-2)}]. \tag{A.8.b}$$

We show that there exists a matrix $\Theta_i(-1)$, such that :

$$\begin{bmatrix} L_i^t \\ -\mathbf{b}_i^t \end{bmatrix} = \Theta_i(-1) \begin{bmatrix} L_{i-1}^t \\ -\mathbf{b}_{i-1}^t \end{bmatrix},$$

with

$$\mathbf{b}_i^t = [\underset{i}{0 \ldots 0}\ b_{i+1}^{(i)}\ \cdots\ b_N^{(i)}].$$

Equation (A.7) can also be written as :

$$\begin{bmatrix} L_{i-1}^t \\ -\mathbf{b}_{i-1}^t \end{bmatrix} = \prod_{j=1}^{\overleftarrow{i-1}} \Theta_j(-1) \begin{bmatrix} L_0^t \\ -\mathbf{b}_0^t \end{bmatrix}. \tag{A.9}$$

Since the $\Theta_j(-1)$ are $\Sigma(-1)$-orthogonal, we get :

$$L_{i-1}L_{i-1}^t - \mathbf{b}_{i-1}\mathbf{b}_{i-1}^t = L_0L_0^t - \mathbf{b}_0\mathbf{b}_0^t > 0. \tag{A.10}$$

Thus,

$$I_N - \mathbf{s}_{i-1}\mathbf{s}_{i-1}^t > 0, \tag{A.11}$$

and

$$\mathbf{s}_{i-1} = L_{i-1}^{-1}\,\mathbf{b}_{i-1} = [\underset{i-1}{0\ldots0}\ s_i^{(i-1)}\ \cdots\ s_N^{(i-1)}]^t, \tag{A.12}$$

where

$$s_j^{(i-1)} = \frac{b_j^{(i-1)}}{l_{ii}^{(i-1)}}. \tag{A.13}$$

Hence, from (A.11) and (A.13) we see that :

$$1 - \left[\frac{b_j^{(i-1)}}{l_{ii}^{(i-1)}}\right]^2 > 0. \tag{A.14}$$

Hence, $\dfrac{|b_j^{(i-1)}|}{|l_{ii}^{(i-1)}|} < 1$, and there exists a matrix $\Theta_i(-1)$ (see (2.4.b)) with $tanh\,(\alpha_i) = \dfrac{b_j^{(i-1)}}{l_{ii}^{(i-1)}}$, such that :

$$\begin{bmatrix} L_i^t \\ -\mathbf{b}_i^t \end{bmatrix} = \Theta_i(-1) \begin{bmatrix} L_{i-1}^t \\ -\mathbf{b}_{i-1}^t \end{bmatrix}, \tag{A.15}$$

with L_i lower triangular and $\mathbf{b}_i^t = [\underset{i}{0\ldots0}\ b_{i+1}^{(i)}\ \cdots\ b_N^{(i)}]$.

□

APPENDIX B

Theorem 2.2 :

Let $A = [a_{ij}] \in \mathbf{R}^{N\times N}$ *be a symmetric positive definite matrix, normalized such that we can write*

$$A = R_L + I_N + R_L^t, \tag{B.1.a}$$

where $R_L = [a_{ij}]$, $i > j$, *is the strictly lower triangular part of A. Put :*

$$U = R_L + I_N \tag{B.1.b}$$

$$Y = R_L. \tag{B.1.c}$$

Then, there exists a matrix product $\Phi \in \mathbf{R}^{2N\times 2N}$ *:*

$$\Phi = \prod_{j=1}^{\overleftarrow{N-1}} \prod_{i=j}^{\overleftarrow{N-1}} \Phi_{ij}, \tag{B.2.a}$$

of embedded plane hyperbolic rotations :

$$\Phi_{ij} = \begin{bmatrix} I_i & & & & \\ & \cosh(\alpha_{ij}) & & \sinh(\alpha_{ij}) & \\ & & I_{N-j-1} & & \\ & \sinh(\alpha_{ij}) & & \cosh(\alpha_{ij}) & \\ & & & & I_{N+j-i-1} \end{bmatrix}, \tag{B.2.b}$$

such that the Cholesky factor L^t *of the matrix A is given from :*

$$\begin{bmatrix} L^t \\ O \end{bmatrix} = \Phi \begin{bmatrix} U^t \\ Y^t \end{bmatrix}. \tag{B.3}$$

Proof :

The proof is by induction. Put $U_0 = U$ and $Y_0 = Y$ and let S_0 be the strictly lower triangular matrix :

$$S_0 = U_0^{-1} Y_0. \tag{B.4}$$

Then,

$$A = [U_0 \quad Y_0] J [U_0 \quad Y_0]^t = U_0 U_0^t - Y_0 Y_0^t$$

$$= U_0 (I_N - S_0 S_0^t) U_0^t > 0.$$

Hence,

$$I_N - S_0 S_0^t > 0. \tag{B.5}$$

The first *off*-diagonal in the lower triangular part of the matrix S_0 has entries $\frac{y^{(0)}_{i+1,i}}{u^{(0)}_{i+1,i+1}}$, $i=1,...,N-1$, where $y^{(0)}_{kl}$ and $u^{(0)}_{kl}$ are the kl-entries of Y_0 and U_0, respectively. And it follows from (B.5) that :

$$\frac{|y^{(0)}_{i+1,i}|}{|u^{(0)}_{i+1,i+1}|} < 1, \quad i = 1,...,N-1. \tag{B.7}$$

Thus, there exist J-orthogonal matrices Φ_{i1} (see (2.26.b) for $j=1$) with $tanh(\alpha_{i1}) = -\frac{y^{(0)}_{i+1,i}}{u^{(0)}_{i+1,i+1}}$, $i=1,...,N-1$, such that :

$$\begin{bmatrix} U_1^t \\ Y_1^t \end{bmatrix} = \Phi_1 \begin{bmatrix} U_0^t \\ Y_0^t \end{bmatrix}, \tag{B.8}$$

$$\Phi_1 = \prod_{i=1}^{\overleftarrow{N-1}} \Phi_{i1}, \tag{B.9}$$

where Y_1 is strictly lower triangular with its first *off*-diagonal zero and U_1 is lower triangular and invertible. From the J-orthogonality of the matrix Φ_1 it follows that: $U_1U_1^t - Y_1Y_1^t = U_0U_0^t - Y_0Y_0^t$.

Now, suppose there exists a matrix Φ_{k-1} :

$$\Phi_{k-1} = \prod_{i=k-1}^{\overleftarrow{N-1}} \Phi_{i,k-1}, \tag{B.10}$$

(the matrices $\Phi_{i,k-1}$ as given in (2.26.b) for $j=k-1$) such that :

$$\begin{bmatrix} U_{k-1}^t \\ Y_{k-1}^t \end{bmatrix} = \Phi_{k-1} \begin{bmatrix} U_{k-2}^t \\ Y_{k-2}^t \end{bmatrix}, \tag{B.11}$$

with U_{k-1} and U_{k-2} lower triangular, invertible and Y_{k-1} and Y_{k-2} strictly lower triangular, with the first k and $k-1$ *off*-diagonals zero, respectively. We shall show next that there exists a matrix Φ_k :

$$\Phi_k = \prod_{i=k}^{\overleftarrow{N-1}} \Phi_{ik},$$

(the matrices Φ_{ik} as given in (2.26.b) for $j=k$) such that :

$$\begin{bmatrix} U_k^t \\ Y_k^t \end{bmatrix} = \Phi_k \begin{bmatrix} U_{k-1}^t \\ Y_{k-1}^t \end{bmatrix},$$

with U_k lower triangular, invertible and Y_k strictly lower triangular, with the first $k+1$ *off*-diagonals zero.

From (B.10) we can write (B.11) as :

$$\begin{bmatrix} U_{k-1}^t \\ Y_{k-1}^t \end{bmatrix} = \prod_{j=1}^{\overleftarrow{k-1}} \prod_{i=j}^{\overleftarrow{N-1}} \Phi_{ij} \begin{bmatrix} U_0^t \\ Y_0^t \end{bmatrix}. \tag{B.12}$$

Putting :

$$S_{k-1} = U_{k-1}^{-1} S_{k-1}, \tag{B.13}$$

and, since the Φ_{ij} are J-orthogonal, it follows from (B.12) that :

$$I_N - S_{k-1} S_{k-1}^t > 0. \tag{B.14}$$

The matrix S_{k-1} is strictly lower triangular, with the first $k-1$ *off*-diagonals zero and entries :

$$\frac{y_{i+1,i}^{(k-1)}}{u_{i+1,i+1}^{(k-1)}}, \quad i=k-1,...,N-1 \tag{B.15}$$

on the *kth off*-diagonal. The *ii*-entry of the product $S_{k-1} S_{k-1}^t$ is equal to the sum of squares of the elements of the *ith* row of the matrix S_{k-1}. Hence, it follows from (B.14) that :

$$\frac{|y_{i+1,i}^{(k-1)}|}{|u_{i+1,i+1}^{(k-1)}|} < 1, \quad i=k-1,...,N-1, \tag{B.16}$$

and there exist matrices Φ_{ik} (see (2.26.b) for $j=k$) with $tanh(\alpha_{ik}) = -\dfrac{y_{i+1,i}^{(k-1)}}{u_{i+1,i+1}^{(k-1)}}$, such that :

$$\Phi_k = \prod_{i=k}^{\overleftarrow{N-1}} \Phi_{ik}, \tag{B.17}$$

$$\begin{bmatrix} U_k^t \\ Y_k^t \end{bmatrix} = \Phi_k \begin{bmatrix} U_{k-1}^t \\ Y_{k-1}^t \end{bmatrix}, \tag{B.18}$$

with U_k lower triangular, invertible and Y_k strictly lower triangular, with its first k *off*-

diagonals zero.

Next, it remains to show that the matrix L in (B.3) is the Cholesky factor of the matrix A. Since the matrix Φ is J-orthogonal, it follows that :

$$LL^t = UU^t - YY^t. \tag{B.19}$$

And, from (B.1.b) and (B.1.c) it follows that $UU^t - YY^t = A$, so that L must be the Cholesky factor of the matrix A.

□

APPENDIX C

Theorem 2.3 :

Given an N×N positive definite matrix $A = I_N + R_L + R_L^t$ (R_L strictly lower triangular), let the matrices U and Y be defined as $U = I_N + R_L$ and $Y = R_L$. Then, in the recursion :

$$U_0 = U, \quad Y_0 = Y$$

$$\begin{bmatrix} U_j^t \\ Y_j^t \end{bmatrix} = \Phi_j \begin{bmatrix} U_{j-1}^t \\ Y_{j-1}^t \end{bmatrix}, \quad j = 1,...,N-1 \tag{C.1}$$

$$Y_{N-1} = 0,$$

$$\Phi_j = \prod_{i=j}^{\overleftarrow{N-1}} \Phi_{ij}, \tag{C.2}$$

(the matrices Φ_{ij} are as given in (2.26.b)) the entries in both U_j and Y_j are bounded by 1 in absolute value, for j=1,...,N-1.

Proof :

Consider recursion (C.1) applied on the $i \times i$ ($i > 1$) leading principal submatrix $\underline{A}_i$, of the $N \times N$ matrix A :

$$\begin{bmatrix} U^{(i)t} \\ Y^{(i)t} \end{bmatrix} = \Phi^{(i)} \begin{bmatrix} U^t \\ Y^t \end{bmatrix}, \tag{C.3}$$

with

$$\Phi^{(i)} = \prod_{j=1}^{\overleftarrow{i-1}} \begin{bmatrix} I_{i-1} & & & \\ & cosh(\alpha_{ij}) & & sinh(\alpha_{ij}) \\ & & I_{N-j-1} & \\ & sinh(\alpha_{ij}) & & cosh(\alpha_{ij}) \\ & & & & I_{N-i+j} \end{bmatrix} \Phi^{(i-1)}, \quad \Phi^{(1)} = I_{2N}. \tag{C.4}$$

Put :

$$\underline{A}_i = \underline{L}_i \underline{L}_i^t = \underline{R}_i \underline{R}_i^t, \tag{C.5}$$

where $\underline{L}_i$ and $\underline{R}_i$ are lower and upper triangular, respectively. According to (2.32) we find for $\Phi^{(i)}$:

$$\Phi^{(i)} = \begin{bmatrix} \mathbf{L}_i^{-1} & 0 \\ 0 & \mathbf{R}_i^{-1} \end{bmatrix} \begin{bmatrix} \mathbf{U}_i & -\mathbf{Y}_i \\ -\mathbf{Y}_i^t & \mathbf{U}_i^t \end{bmatrix}, \tag{C.6}$$

where

$$\mathbf{L}_i^{-1} = \begin{bmatrix} \underline{L}_i^{-1} & 0 \\ 0 & I_{N-i} \end{bmatrix} \tag{C.7}$$

$$\mathbf{R}_i^{-1} = \begin{bmatrix} \underline{R}_i^{-1} & 0 \\ 0 & I_{N-i} \end{bmatrix} \tag{C.8}$$

$$\mathbf{U}_i = \begin{bmatrix} \underline{U}_i & 0 \\ 0 & I_{N-i} \end{bmatrix} \tag{C.9}$$

$$\mathbf{Y}_i = \begin{bmatrix} \underline{Y}_i & 0 \\ 0 & 0 \end{bmatrix}. \tag{C.10}$$

The matrices $\underline{U}_i$ and $\underline{Y}_i$ are the $i \times i$ leading principal submatrices of U and Y, respectively. The structure of L_i^{-1}, R_i^{-1}, U_i and Y_i is infered from the fact that the matrix $\Phi^{(i)}$ must

leave the $N-i$ last rows of U^t and Y^t unaffected in (C.3), while the $i\times i$ leading principal sub matrix of $U^{(i)^t}$ must be equal to $\underline{L}_i^t$. Note that the matrix $\Phi^{(i)}$ is J-orthogonal so that :

$$[U \quad Y]\Phi^{(i)^t} J \Phi^{(i)} \begin{bmatrix} I_N \\ I_N \end{bmatrix} = I_N = U^{(i)}\mathbf{L}_i^{-1} - Y^{(i)}\mathbf{R}_i^{-1}. \tag{C.11}$$

Now, because the $i\times i$ leading principal submatrix of $Y^{(i)}$ is zero, it follows from (C.11) and (C.6) that the diagonal elements of $U^{(i)}\mathbf{L}_i^{-1}$ are 1 and also that the $i\times i$ leading principal sub matrix of $U^{(i)}$ is the inverse of $\underline{L}_i^{-1}$ as expected.

Substituting (C.6) in (C.3) gives :

$$U\mathbf{U}_i^t - Y\mathbf{Y}_i^t = U^{(i)}\mathbf{L}_i^t; \tag{C.12.a}$$

$$Y\mathbf{U}_i - U\mathbf{Y}_i = Y^{(i)}\mathbf{R}_i^t. \tag{C.12.b}$$

Substituting

$$U = Y + I_N \tag{C.13.a}$$

and

$$\mathbf{U}_i = \mathbf{Y}_i + I_N \tag{C.13.b}$$

in (C.12.a) gives

$$Y + I_N + \mathbf{Y}_i^t = (Y + Y^t + I_N) + (\mathbf{Y}_i - Y)^t =$$

$$A + (\mathbf{Y}_i - Y)^t = U^{(i)}\mathbf{L}_i^t.$$

Or,

$$A\mathbf{L}_i^{-t} = U^{(i)} - (\mathbf{Y}_i - Y)^t\mathbf{L}_i^{-t}. \tag{C.14}$$

From (C.7) and (C.10) it follows that the factor $(\mathbf{Y}_i - Y)^t\mathbf{L}_i^{-t}$ is strictly upper triangular. Hence,

$$\mathbf{e}_k^t A \mathbf{L}_i^{-t} \mathbf{e}_l = \mathbf{e}_k^t U^{(i)} \mathbf{e}_l = u_{kl}^{(i)}, \quad \text{for } k=1,...,N \text{ and } l=1,...,k, \tag{C.15}$$

with $\mathbf{e}_j$ the jth natural base vector in $\mathbb{R}^{N\times N}$. Equation (C.15) gives all entries in the lower triangular part of $U^{(i)}$.

Substituting

$$Y = U - I_N \tag{C.16.a}$$

and

$$\mathbf{Y}_i = \mathbf{U}_i - I_N \tag{C.16.b}$$

in (C.12.b) gives

$$U - \mathbf{U}_i = (U + Y^t) - (\mathbf{U}_i + Y^t) =$$

$$A - (\mathbf{U}_i + Y^t) = Y^{(i)} \mathbf{R}_i^t.$$

Or,

$$A \mathbf{R}_i^{-t} = Y^{(i)} + (\mathbf{U}_i + Y^t) \mathbf{R}_i^{-t}. \tag{C.17}$$

But, as follows from (C.8) and (C.9) we have

$$\mathbf{e}_k^t A \mathbf{R}_i^{-t} \mathbf{e}_l = \mathbf{e}_k Y^{(i)} \mathbf{e}_l = y_{kl}^{(i)}, \quad \text{for } k=i+1,...,N \text{ and } l=1,...,k-1. \tag{C.18}$$

I.e., these are all entries in the strictly lower triangular part of $Y^{(i)}$, outside the $i\times i$ leading principal sub matrix of $Y^{(i)}$.

Next, we define the inner product :

1) $<\mathbf{v},\mathbf{w}>_A = \mathbf{v}^t A \mathbf{w}$,

2) $<\mathbf{v},\mathbf{v}>_A = ||\mathbf{v}||_A^2 > 0$ (*A is positive definite*),

3) $||\mathbf{v}||_A = 0$, *iff* $\mathbf{v} = 0$ (*A is non-singular*),

for which the Gauchy-Schwarz inequality holds :

$$|\langle \mathbf{v},\mathbf{w}\rangle| \leq ||\mathbf{v}||_A \, ||\mathbf{w}||_A. \tag{C.19}$$

According to the above definition of the inner product, we find for (C.15) and (C.18) the following Gauchy-Schwarz inequalities :

$$|u_{kl}^{(i)}| = |\langle \mathbf{e}_k, \mathbf{L}_i^{-t}\mathbf{e}_l\rangle_A| \leq ||\mathbf{e}_k||_A \, ||\mathbf{L}_i^{-t}\mathbf{e}_l||_A, \; k=1,...,N; \; l=1,...,k, \tag{C.20}$$

$$|y_{kl}^{(i)}| = |\langle \mathbf{e}_k, \mathbf{R}_i^{-t}\mathbf{e}_l\rangle_A| \leq ||\mathbf{e}_k||_A \, ||\mathbf{R}_i^{-t}\mathbf{e}_l||_A, \; k=i+1,...,N; \; l=1,...,k-1. \tag{C.21}$$

From the fact that the matrix A has only 1's on the *main*-diagonal, it follows that :

$$||\mathbf{e}_k||_A = 1. \tag{C.22}$$

For $||\mathbf{L}_i^{-t}\mathbf{e}_l||_A$ we find from (C.14) :

$$\begin{aligned} ||\mathbf{L}_i^{-t}\mathbf{e}_l||_A^2 &= \mathbf{e}_l^t \mathbf{L}_i^{-1} A \mathbf{L}_i^{-t}\mathbf{e}_l \\ &= \mathbf{e}_l^t \mathbf{L}_i^{-1} U^{(i)}\mathbf{e}_l - \mathbf{e}_l^t \mathbf{L}_i^{-1}(\mathbf{Y}_i - Y)^t \mathbf{L}_i^{-t}\mathbf{e}_l \\ &= \mathbf{e}_l^t \mathbf{L}_i^{-1} U^{(i)}\mathbf{e}_l, \end{aligned} \tag{C.23}$$

since $\mathbf{L}_i^{-1}(\mathbf{Y}_i - Y)^t \mathbf{L}_i^{-t}$ is strictly upper triangular with zero $i\times i$ leading principal sub matrix. But, from (C.11) we know that the $i\times i$ leading principal sub matrix of $U^{(i)}$ is equal to $\underline{L}_i$, so that $\mathbf{L}_i^{-1}U^{(i)}$ has all 1's on the *main*-diagonal. Hence,

$$||\mathbf{L}_i^{-t}\mathbf{e}_l||_A = 1. \tag{C.24}$$

From (C.20) it then follows that :

$$|u_{kl}^{(i)}| \leq 1, \quad \text{for } k=1,...,N \text{ and } l=1,...,k. \tag{C.25}$$

For $||\mathbf{R}_i^{-t}\mathbf{e}_l||_A$ in (C.21) we find :

$$||\mathbf{R}_i^{-t}\mathbf{e}_l||_A^2 = \mathbf{e}_l^t \mathbf{R}_i^{-1} A \mathbf{R}_i^{-t}\mathbf{e}_l. \tag{C.26}$$

But, from the fact that the matrix A has all 1's on its *main*-diagonal and from (C.8) and (C.5) we deduce that the product $\mathbf{R}_i^{-1} A \mathbf{R}_i^{-t}$ also has all 1's on the *main*-diagonal. Hence,

$$\| \mathbf{R}_i^{-t} \mathbf{e}_l \|_A = 1. \tag{C.27}$$

From (C.21) it then follows that :

$$|y_{kl}^{(i)}| \leq 1, \quad \text{for } k=i+1,...,N \text{ and } l=1,...,k-1. \tag{C.28}$$

Although the proof, as given above, is based on the leading principal submatrices of U and Y, it is valid for the *diagonal*-recursive scheme in (C.1). In the leading principal submatrix approach we do not consider the elements below the ith row of U and Y (for $i \times i$ leading principals), unlike in the *diagonal*-recursive scheme. However, the elements of these rows can be considered by taking successively the $(i+1)\times(i+1)$, $(i+2)\times(i+2)$,... leading principals. In other words, it does not matter if we first process the $i \times i$ leading principal and continue with the $(i+1)\times(i+1)$,... leading principals, or if we adhere to a *diagonal*-recursive scheme as in (C.1).

□

APPENDIX D

In this appendix the micro-program for the Householder processor element of Chapter 4 is expressed in a pseudo programming language. All statements grouped within rectangular brackets are supposed to be exectuted in a single clock cycle.

```
do
 [
 O/H_I = READ_IN_O/H; /* enable input buffer at O/H_IN to send O/H control bit */
 V/R_I = READ_IN_V/R; /* enable input buffer at V/R_IN to send V/R control bit */
 VO = DISABLED; /* disable output buffer at VEC_OUT to receive */
 SO = DISABLED; /* disable output buffer at SCA_OUT to receive */
 VI = DISABLED; /* disable input buffer at VEC_IN to send */
 SI = DISABLED; /* disable input buffer at SCA_IN to send */
 O/H_O = DISABLED; /* disable output buffer at O/H_OUT to receive */
 V/R_O = DISABLED; /* disable output buffer at V/R_OUT to receive */
 ]
 if (O/H_IN == ORTH) /* orthogonal Householder transformation */
   micro_program_orth
 endif
 if (O/H_IN == HYPER) /* hyperbolic Householder transformation */
   micro_program_hyper
 endif
 [
 O/H_O = WRITE_OUT_O/H; /* enable output buffer at O/H_OUT to receive */
 V/R_O = WRITE_OUT_V/R; /* enable output buffer at V/R_OUT to receive */
 ]
```

```
 [
  O/H_OUT = O/H_IN; /* send O/H control bit from input buffer at O/H_IN
  to output buffer at O/H_OUT */
  V/R_OUT = V/R_IN; /* send V/R control bit from input buffer at V/R_IN
  to output buffer at V/R_OUT */
 ]
until (forever)
```

The micro-program **micro_program_orth** is listed below. It expresses the execution of a Householder transformation. Details of exponent handling in floating point arithmetic by the unit **SQR_INV** in Figure 4.10 have been omitted here. The details can be found in [Hofw88]. The micro-program **micro_program_hyper** for the execution of hyperbolic Householder transformations is identical, except for the necessary subtractions that are introduced by the signature matrix (see (4.13)).

```
micro_program_orth:
 if (V/R_IN == VECT & ID == DISABLED) /* a vectorizing operation */
   [
    LOOPCNTR = 1;
    VI = READ_IN_VECT; /* enable input buffer at VEC_IN to send entries of vector x */
   ]
   while (LOOPCNTR < n) /* compute Σ_{i=1}^{n-1} x_i^2 */
     [
      BUS A = BUS B = VEC_IN;
      RTMP1 = BUS A;
      RTMP2 = BUS B;
      if (LOOPCNTR > 1) /* x_2,...,x_n are the last n-1 entries of u */
        FIFO_U[LOOPCNTR -1] = BUS A;
      endif
      if (LOOPCNTR == 1) /* store x_1 to compute u_1 */
        [
         RTMP3 = 0;
         RX1 = BUS A;
        ]
      endif
```

```
 ]
  RTMP4 = RTMP1*RTMP2; /* x_i^2 */
 [
  LOOPCNTR = LOOPCNTR+1;
  RTMP3 = RTMP3+RTMP4; /* x_i^2 + sum_{j=1}^{i-1} x_j^2 */
 ]
endwhile
[
 SI = READ_IN_SCA; /* enable input buffer at SCA_IN to send */
 VI = DISABLED; /* disable input buffer at VEC_IN to send */
]
[
 BUS A = BUS B = SCA_IN; /* receive x_n from input buffer at SCA_IN */
 RTMP1 = BUS A;
 RTMP2 = BUS B;
]
 RTMP4 = RTMP1*RTMP2; /* x_n^2 */
[
 BUS B = RTMP3+RTMP4; /* ||x||_2^2 */
 RTMP2 = BUS B;
 RTMP1 = Δ^-1; /* starting computation of table-row entry k */
]
[
 RTMP3 = 0.5–0.25Δ^-1;
 RTMP4 = RTMP1*RTMP2;
]
[
 ROW_CNTR = ⌊(RTMP3+RTMP4)⌋ ; /* k */
 LOOPCNTR = 1;
 RTMP1 = 1;
]
[
 RTMP4 = RTMP1*RTMP2;
 BUS B = TABLE1[ROW_CNTR, LOOPCNTR]; /* −z_k */
 RTMP3 = BUS B;
]
[
 BUS B = RTMP4+RTMP3;
```

```
RTMP2 = BUS B; /* ||x||_2^2 - z_k */
RTMP1 = 0;
LOOPCNTR = LOOPCNTR+1; /* start Taylor series expansion of (||x||_2^2)^1/2 */
]
[
RTMP4 = RTMP1*RTMP2; /* 0 */
BUS B = TABLE1[ROW_CNTR, LOOPCNTR]; /* b_n_s */
RTMP3 = BUS B;
]
[
BUS A = RTMP4+RTMP3;
RTMP1 = BUS A; /* b_n_s */
LOOPCNTR = LOOPCNTR+1;
]
[
while (LOOPCNTR < n_s) /* compute the rest of the Taylor series expansion */
  [
   RTMP4 = RTMP1*RTMP2;
   BUS B = TABLE1[ROW_CNTR, LOOPCNTR]; /* b_i */
   RTMP3 = BUS B;
  ]
  [
   BUS A = RTMP4+RTMP3;
   RTMP1 = BUS A;
   LOOPCNTR = LOOPCNTR+1;
  ]
endwhile
[
RTMP4 = RTMP1*RTMP2;
BUS B = TABLE1[ROW_CNTR, LOOPCNTR]; /* b_1 */
RTMP3 = BUS B;
]
[
BUS B = ±(RTMP4+RTMP3);
RTMP2 = BUS B; /* ±||x||_2 */
RTMP1 = 1; /* start computation of u_1 */
]
[
RTMP4 = RTMP1*RTMP2;
```

```
BUS B = RX1;
RTMP3 = BUS B; /* x1 */
]
[
BUS A = RTMP4+RTMP3; /* u1 */
RTMP1 = RU1 = BUS A;
RTMP2 = |RTMP2|; /* ||x||2 */
]
[
RTMP4 = RTMP1*RTMP2;
RTMP3 = 0;
]
[
BUS B = RTMP4+RTMP3;
RTMP2 = BUS B; /* β^-1 */
RTMP1 = Δ^-1; /* starting computation of table-row entry k */
]
[
RTMP3 = 0.5-0.25Δ^-1;
RTMP4 = RTMP1*RTMP2;
]
[
ROW_CNTR = ⌊(RTMP4+RTMP3)⌋ ; /* k */
LOOPCNTR = 1;
RTMP1 = 1;
]
[
RTMP4 = RTMP1*RTMP2;
BUS B = TABLE2[ROW_CNTR, LOOPCNTR];
RTMP3 = BUS B; /* -z_k */
]
[
BUS B = RTMP4+RTMP3;
RTMP2 = BUS B; /* β^-1-z_k */
RTMP1 = 0;
LOOPCNTR = LOOPCNTR+1; /* start Taylor series expansion of (β^-1)^-1 */
]
[
RTMP4 = RTMP1*RTMP2;
```

```
  BUS B = TABLE2[ROW_CNTR, LOOPCNTR];
  RTMP3 = BUS B; /* b_{n_i} */
 ]
 [
  BUS A = RTMP4+RTMP3;
  RTMP1 = BUS A; /* b_{n_i} */
  LOOPCNTR = LOOPCNTR+1;
 ]
 while (LOOPCNTR < n_i) /* compute the rest of the Taylor series expansion */
  [
   RTMP4 = RTMP1*RTMP2;
   BUS B = TABLE2[ROW_CNTR, LOOPCNTR];
   RTMP3 = BUS B; /* b_i */
  ]
  [
   BUS A = RTMP4+RTMP3;
   RTMP1 = BUS A;
   LOOPCNTR = LOOPCNTR+1;
  ]
 endwhile
 [
  RTMP4 = RTMP1*RTMP2;
  BUS B = TABLE2[ROW_CNTR, LOOPCNTR];
  RTMP3 = BUS B; /* b_1 */
 ]
 [
  BUS A = RTMP4+RTMP3;
  RBETA = BUS A; /* β */
 ]
endif

if (V/R_IN == REFL & ID == DISABLED) /* a reflection operation */
 [
  LOOPCNTR = 1;
  VI = READ_IN_VECT; /* enable buffer at VEC_IN to send entries y_1 to y_{n-1} of vector y */
 ]
 while (LOOPCNTR < n) /* compute sum_{i=1}^{n-1} u_i y_i */
  [
```

```
  BUS A = VEC_IN;
  if (LOOPCNTR == 1) /* u1 */
    [
     RTMP3 = 0;
     BUS B = RU1;
    ]
  endif
  else
    BUS B = FIFO_U[LOOPCNTR-1]; /* u2 to un */
  endelse
  RTMP1 = BUS A;
  RTMP2 = BUS B;
  FIFO_Y[LOOPCNTR] = BUS A; /* store entries of vector y */
 ]
  RTMP4 = RTMP1*RTMP2;
 [
  RTMP3 = RTMP4+RTMP3;
  LOOPCNTR = LOOPCNTR+1;
 ]
endwhile
[
 SI = READ_IN_SCA; /* enable input buffer at SCA_IN to send yn */
 VI = DISABLED; /* disable input buffer at VEC_IN to send */
]
[
 BUS A = SCA_IN;
 BUS B = FIF0_U[LOOPCNTR-1]; /* un */
 RTMP1 = BUS A; /* yn */
 FIFO_Y[LOOPCNTR] = BUS A; /* store yn */
 RTMP2 = BUS B; /* un */
]
[
 RTMP4 = RTMP1*RTMP2;
 LOOPCNTR = LOOPCNTR+1;
]
[
 BUS A = RTMP4+RTMP3;
 RTMP1 = BUS A; /* u'y */
 BUS B = RBETA; /* β */
```

```
RTMP2 =BUS B;
]
[
RTMP4 = RTMP1*RTMP2;
RTMP3 = 0;
]
[
BUS A = -(RTMP4+RTMP3);
RTMP1 = BUS A; /* -β(u^t y) */
BUS B = RU1;
RTMP2 = BUS B;
]
while (LOOPCNTR ≤ n) /* compute y_i-β(u^t y) */
 [
 RTMP4 = RTMP1*RTMP2; /* -β(u^t y)u_i */
 BUS B = FIFO_Y[LOOPCNTR]; /* y_i */
 RTMP3 = BUS B;
 if (LOOPCNTR == 1) /* y_1-β(u^t y)u_1 goes to the scalar output */
   SO = WRITE_OUT_SCA; /* enable output buffer at SCA_OUT to receive scalar */
 endif
 else
  [
   SO = DISABLED; /* disable output buffer at SCA_OUT to receive */
   VO = WRITE_OUT_VEC; /* enable output buffer at VEC_OUT to receive */
  ]
 endelse
]
[
 BUS A = RTMP4+RTMP3; /* y_i-β(u^t y)u_i */
 if (LOOPCNTR == 1) /* to scalar output */
   SCA_OUT = BUS A; /* output buffer at SCA_OUT receives y_1-β(u^t y)u_1 */
 endif
 else /* to vector output */
   VEC_OUT = BUS A; /* output buffer at VEC_OUT receives entries y_2-β(u^t y)u_2
   to y_n-β(u^t y)u_n */
 endelse
 BUS B = FIFO_U[LOOPCNTR];
 RTMP2 = BUS B;
]
```

```
    LOOPCNTR = LOOPCNTR+1;
  endwhile
endif


if (ID == IDENTITY) /* the identity I/O map */
  [
   VI = READ_IN_VEC; /* enable input buffer at VEC_IN to send y_1 to y_{n-1} */
   LOOPCNTR = 1;
  ]
  while (LOOPCNTR < n) /* read in the n-1 elements of y */
    [
     BUS A = VEC_IN;
     FIFO_Y[LOOPCNTR] = BUS A; /* store y_i */
    ]
     LOOPCNTR = LOOPCNTR+1;
  endwhile
  [
   VI = DISABLED; /* disable input buffer at VEC_IN to send */
   SI = READ_IN_SCA; /* enable input buffer at SCA_IN to send y_n */
  ]
  [
   BUS A = SCA_IN;
   FIFO_Y[LOOPCNTR] = BUS A; /* store y_n */
  ]
  do /* wait until the (n+1)st clock cycle before sending output data */
    [
     SI = DISABLED; /* disable input buffer at SCA_IN to send */
     LOOPCNTR = LOOPCNTR+1;
    ]
  until (LOOPCNTR == n+1)
  [
   SO = WRITE_OUT_SCA; /* enable output buffer at SCA_OUT to receive */
   LOOPCNTR = 1;
  ]
  [
   BUS B = FIFO_Y[LOOPCNTR];
   SCA_OUT = BUS B; /* y_1 */
  ]
```

```
   [
    SO = DISABLED; /* disable output buffer at SCA_OUT to receive */
    VO = WRITE_OUT_VEC; /* enable output buffer at VEC_OUT to receive */
   ]
   while (LOOPCNTR ≤ n) /* output y2 to yn */
     [
      BUS B = FIFO_Y[LOOPCNTR];
      VEC_OUT = BUS B; /* yi */
     ]
      LOOPCNTR = LOOPCNTR+1;
   endwhile
 endif
endmicro_program_orth
```

CONVENTIONS, SYMBOLS AND DEFINITIONS

CONVENTIONS

In this thesis the following notational conventions are used. Scalars are denoted by (indexed) lower case letters in italic font or by indexed lower case Greek symbols. Occasionally an upper case Greek symbol is used to denote a scalar, but this will be clear from the context. Vectors are denoted by lower case letters in bold font or by lower case Greek symbols. Matrices and maps are denoted by upper case letters in italic font or by upper case Greek symbols. Sets are denoted by upper case letters in roman font.

SYMBOLS

$\mathbf{Z}$	: set of integers
$\mathbb{R}$	: set of reals
$\mathbf{Z}^{n\times m}$	: the vector space of $n\times m$ integral matrices
$\mathbb{R}^{n\times m}$	: the vector space of $n\times m$ real matrices
$\mathbf{Z}^{n}$	: the vector space of $n\times 1$ integral vectors
$\mathbb{R}^{n}$	: the vector space of $n\times 1$ real vectors
$\varnothing$	: the empty set
$v \in G$	: scalar v is a member of the set G
$v \notin G$	: scalar v is not a member of the set G
$H \subset G$	: H is a subset of G
$H \not\subset G$	: H is not a subset of G
$H \cup G$	: the union of sets H and G
card (H)	: the cardinality of the set H
$\{x \mid P\}$	: set of all elements with property P
$[a, b]$	: $\{x \mid a \leq x \leq b\}$
$(a, b]$	: $\{x \mid a < x \leq b\}$

$[a, b)$: $\{x \mid a \leq x < b\}$

(a, b) : $\{x \mid a < x < b\}$

I_N : $N \times N$ identity matrix

A^t : transpose of the matrix A

A^{-1} : inverse of the non singular matrix A

$A > 0$: a positive definite matrix A

$A = [a_{ij}]$: the matrix A with entries a_{ij}

$\langle \mathbf{a}, \mathbf{b} \rangle$: scalar product $\mathbf{a}^t\mathbf{b}$

$\langle \mathbf{a}, \mathbf{b} \rangle_A$: scalar product $\mathbf{a}^t A \mathbf{b}$ with respect to the matrix A

$\| \mathbf{a} \|_2$: $(\mathbf{a}^t\mathbf{a})^{1/2}$

$\| \mathbf{a} \|_A$: $(\mathbf{a}^t A \mathbf{a})^{1/2}$

$\| A \|_F$: the Frobenius norm $\left[\sum_{i=1}^{N}\sum_{j=1}^{M} a_{ij}^2\right]^{1/2}$ of the $N \times M$ matrix $A = [a_{ij}]$

$\lambda(A)$: the spectral radius of the matrix A

$\oplus$: orthogonal sum

$gcd(a_1, \ldots, a_n)$: a positive integer which is the greatest common divisor of the integers $a_1, \ldots, a_n$

$\lfloor a \rfloor$: the largest integer less than or equal to a

$\lceil a \rceil$: the smallest integer greater than or equal to a

$f^{(i)}$: the ith derivative of the function f

$\prod_{i=0}^{\leftarrow N} A_i$: the product $A_N \ldots A_1$

LU : the LU factorization of a matrix A

QR : the QR factorization of a matrix A

LQ : the LQ factorization of a matrix A

LL^t : the Cholesky factorization of a symmetric positive definite matrix A

DEFINITIONS

Let M be a subspace of $\mathbb{Z}^2$.

Definition I : *The neighborhood* N(**x**) *of* $\mathbf{x} \in M$ *is defined to be the set :*

$$N(\mathbf{x}) = \{\mathbf{y} \in M \mid \|\mathbf{x} - \mathbf{y}\|_2 \leq r \text{ and } r = \min_{all\ \mathbf{p} \in M,\ \mathbf{p} \neq \mathbf{x}} \|\mathbf{x} - \mathbf{p}\|_2\}.$$

Definition II : *Let* $L \subset M$. *The contour* C(L) *of* L *is defined to be the set :*

$$C(L) = \{\mathbf{x} \in L \mid N(\mathbf{x}) \not\subset L\}.$$

Definition III : *Let* $L \subset M$, *with contour* C(L) *and card*(L) > 1. *The set* L *is said to be homogeneous in* M, *if for every* $\mathbf{i} \in L$, *the total angle of revolution around* **i** *is not zero when the contour* C(L) *is traversed in counter clockwise direction from* $\mathbf{j} \in C(L)$ *back to* **j**.

Definition IV : *Let x and y be integers, and z a positive integer. We say that x is congruent to y modulo z, and write* $x \equiv y \pmod{z}$, *whenever* $x - y$ *is divisible by z.*

Definition V : *For any integer x and positive integer z we denote the equivalence class of x with respect to congruence modulo z by* $[x]_z$. *That is,* $[x]_z$ *consists of all integers y for which* $y - x$ *is a multiple of z.*

References

Ahme1982. Ahmed, H.M., "Signal Processing Algorithms and Architectures," *Ph.D. dissertation*, (Stanford University, 1982).

Arde1963. Arden, B.W. and K.N. Astill, "Chapter 7," pp. 174-183 in *Numerical Algorithms: Origins and Applications*, Addison-Wesley Publishing Company (1963).

Bara1988. Barazesh, B., J.C. Michalina, and A. Picco, "A VLSI Signal Processor with Complex Arithmetic Capability," *IEEE Transactions on Circuits and Systems* **35**(5) pp. 495-505 (May 1988).

Bu1988. Bu, J. and E.F. Deprettere, "Converting Sequential Iterative Algorithms to Recurrent Equations for Automatic Design of Systolic Arrays," *Proceedings ISCAS*, (1988).

Bult1981. Bultheel, A., "Error Analysis of Incoming and Outgoing Schemes for the Trigoniometric Problem," *Pade Approximation and its Applications*, pp. 100-109 Springer Verlag, (1981).

Chua1985. Chuang, H.Y.H. and G.He, "A Versatile Systolic Array for Matrix Computations," *Proceedings International Conference on Parallel Processing 1985*, pp. 315-322 (1985).

Delo1984. Delosme, J.M. and I.C.F. Ipsen, "Efficient Parallel Solution of Linear Systems with Hyperbolic Rotations," Research Report YALEU/DCS/RR-341, Yale University (Nov. 1984).

Delo1986. Delosme, J.M. and I.C.F. Ipsen, "Efficient Systolic Arrays for the Solution of Toeplitz Systems: An Illustration of a Methodology for the Construction of Systolic Architectures in VLSI," *Proceedings of the Int. Workshop on Systolic Arrays*, pp. F3.1-F3.22 (July 1986).

Depr1982. Deprettere, E.F., "Mixed Form Time-Variant Lattice Recursions," pp. 545-561 in *Outils et Modeles Mathematiques pour L'Automatique L'Analyse de*

Systemes et le Traitment du Signal, ed. I.D. Landau, , Paris (1982).

Dewi1981. Dewilde, P. and H. Dym, "Schur Recursions, Error Formulas and Convergence of Rational Estimators for Stationary Stochastic Sequences," *IEEE Tran. Inf. Th.* **IT-27**(4)(July 1981).

Dewi1987. Dewilde, P. and E.F. Deprettere, "Modelling VLSI Interconnects as an Inverse Scattering Problem," *IEEE Proceedings ISCAS*, pp. 147-153 (May 1987).

Dewi1978. Dewilde, P.M., A. Viera, and T. Kailath, "On a Generalized Szego-Levinson Realization Algorithm for Optimal Linear Predictors Based on a Network Synthesis Approach," *IEEE Tran. Circuits and Systems* **CAS-25** pp. 663-675 (Sept. 1978).

Dong1988. Dongen van V. and P. Quinton, "Uniformization of Linear Recurrence Equations: A Step towards the Automatic Synthesis of Systolic Arrays," *Proceedings of the International Conference on Systolic Arrays*, pp. 473-482 (1988).

Dym1981. Dym, H. and I. Gohberg, "Extensions of Band Matrices with Band Inverses," *Linear Algebra and its Applications*, pp. 1-24 Elsevier, (1981).

Fadd1959. Faddeeva, V.N., *Computational Methods of Linear Algebra*, Dover Publications, Inc., New York (1959).

Fort1985. Fortes, J.A.B., K.S. Fu, and B.W. Wah, "Systematic Approaches to the Design of Algorithmic Specified Systolic Arrays," *Proc. ICASSP*, (1985).

Fris1986. Frison, P., P. Gachet, and P. Quinton, "Designing Systolic Arrays with DIASTOL," pp. 93-105 in *VLSI Signal Processing II*, ed. S.Y. Kung, R.E. Owen and J.G. Nash, IEEE Press, New York (1986).

Gent1981. Gentleman, M.W. and H.T. Kung, "Matrix Triangularization by Systolic Arrays," *SPIE, Real-Time Signal Processing IV* **298** pp. 19-26 (1981).

Gent1975. Gentleman, W.M., "Error Analysis of QR Decompositions by Givens Transformations," *Linear Algebra and its Applications* **10** pp. 189-197 (1975).

Golu1983. Golub, G.H. and C.F. Loan van, *Matrix Computations*, The John Hopkins University Press, Baltimore, Maryland (1983).

Hell1985. Heller, D., "Partitioning Big Matrices for Small Systolic Arrays," pp. 185-199 in *VLSI and Modern Signal Processing*, ed. S.Y. Kung, H.J. Whitehouse, T. Kailath, Prentice Hall, Englewood Cliffs (1985).

Hofw1988. Hofwegen, K. and K. Jainandunsing, "Design of a Householder Processor Element for Systolic Array Processors," Technical Report TU Delft, The Netherlands, Delft (1988).

Hori1987. Horiike, S., S. Nishida, and T. Sakaguchi, "A Design Method of Systolic Arrays Under the Constraint of the Number of the Processors," *Proceedings ICASSP*, pp. 764-767 (1987).

Hous1975. Householder, A.S., *The Theory of Matrices in Numerical Analysis*, Dover Publications, Inc., New York (1975).

Hwan1979. Hwang, K., *Computer Arithmetic: Principles, Architecture and Design*, John Wiley & Sons, New York (1979).

Jain1986a. Jainandunsing, K. and E.F.A. Deprettere, "Design and VLSI Implementation of a Concurrent Solver for N Coupled Least-Squares Fitting Problems," *IEEE Journal on Selected Areas in Communications* **SAC-4**(1) pp. 39-48 (January 1986).

Jain1986b. Jainandunsing, K. and E.F. Deprettere, "A Novel VLSI System of Equations Solver for Real-Time Signal Processing," *Proceedings SPIE, Real Time Signal Processing IX* **698**(1986).

Kail1979. Kailath, T., S.Y. Kung, and M. Morf, "Displacement Ranks of Matrices and Linear Equations," *Journal of Math. Anal. and Appl.* **68**(2)(April 1979).

Karp1967. Karp, R.M., R.E. Miller, and S. Winograd, "The Organization of Computations for Uniform Recurrence Equations," *Journal of the ACM* **14** pp. 563-590 (1967).

Krek1988. Krekel, P. and E. Deprettere, "A Systolic Algorithm and Architecture for Solving Sets of Linear Equations with Multi-Band Coefficient Matrix," *Proceeedings of the International Conference on Systolic Arrays*, (May 25-27, 1988).

Kung1979a. Kung, H.T. and C.E. Leiserson, "Systolic Arrays (for VLSI)," *Sparse Matrix Proceedings 1978*, pp. 256-282 Society for Industrial and Applied Mathematics, (1979).

Kung1979b. Kung, H.T., "Let's Design Algorithms for VLSI Systems," *Proc. of the First Caltech Conference on VLSI*, pp. 65-90 (Jan. 1979).

Kung1982. Kung, H.T., "Why Systolic Architectures?," *Computer*, pp. 37-45 (1982).

Kung1988. Kung, S.Y., *VLSI Array Processors,* Prentice Hall, Englewood Cliffs, New Jersey (1988).

Lang1988. Lange de, A.A.J., A.J. Hoeven van der, E.F. Deprettere, and J.C. Bu, "An Optimal Floating Point Pipelined CMOS CORDIC Processor," *Proc. ISCAS 1988*, (June, 1988).

Leis1981. Leiserson, C.E., "Area-Efficient VLSI Computation," *Ph.D. dissertation*, (Carnegie-Mellon University, 1981).

Lev1984. Lev-Ari H. and T. Kailath, "Lattice Filter Parameterization and Modeling of Nonstationary Processes," *IEEE Trans. Inform. Th.* **30**(1)(Jan. 1984).

Lipt1986. Lipton, R.J. and D. Lopresti, "Comparing Long STrings on a Short Systolic Array," *Proceedings of the Int. Workshop on Systolic Arrays*, pp. K1.1-K1.10 (July 1986).

McCl1958. McCluskey, E.J., "Iterative Combinatorial Switching Networks - General Design Considerations," *IRE Trans. on Electronic Computers* **EC-7** pp. 285-291 (1958).

McWh1983. McWhirter, J.G., "Recursive Least-Squares Minimization Using a Systolic Array," *Proceedings SPIE* **431**(1983).

Mead1980. Mead, C. and L. Conway, *Introduction to VLSI Systems,* Addison-Wesley Publishing Company (1980).

Mold1983. Moldovan, D.I., "On the Design of Algorithms for VLSI Systolic Arrays," *Proceedings of the IEEE* **71**(1) pp. 113-120 (January, 1983).

Mold1986. Moldovan, D.I. and J.A.B. Fortes, "Partitioning and Mapping Algorithms into Fixed Size Systolic Arrays," *IEEE Tran. Computers* **C-35**(1) pp. 1-12 (January, 1986).

Nash1988. Nash, J.G. and S. Hansen, "Modified Faddeeva Algorithm for Concurrent Execution of Linear Algebraic Operations," *IEEE Transactions on Computers* **37**(2) pp. 129-137 (February, 1988).

Nava1986. Navarro, J.J., J.M. Llaberia, and M. Valero, "Solving Matrix Problems With no Size Restriction on a Systolic Array Processor," *Proceedings Int. Conf. Par. Proc.*, pp. 676-683 (1986).

Neli1988. Nelis, H. and E. Deprettere, "Automatic Design and Partitioning of Systolic/Wavefront Arrays for VLSI," *CSSP Special Issue on Array Processing*, (January, 1988).

Omtz1987. Omtzigt, E.T.L., "SYSTARS, a CAD Tool for the Synthesis and Analysis of VLSI Systolic/Wavefront Arrays," *Technical Report No. 87-68*, Delft University of Technology, Dept. of Electr. Eng., (1987).

Omtz1988. Omtzigt, E.T.L., "SYSTARS, a CAD Tool for the Synthesis and Analysis of VLSI Systolic/Wavefront Arrays," *Proceedings of the International Conference*

on Systolic Arrays, (1988).

Orfa1985. Orfanidis, S.J., *OPTIMUM SIGNAL PROCESSING* , Macmillan Publishing Company, New York (1985).

Rade1985. Rader, C.M. and A.O. Steinhardt, "Hyperbolic Householder Transformations," *IEEE Tran. Acoustics, Speech and Signal Processing*, (1985).

Rajo1987. Rajopadhye, S.V and R.M. Fujimoto, "Systolic Array Synthesis by Static Analysis of Program Dependencies," *Proceedings of the Conference on Parallel Architectures and Languages Europe*, pp. 295-310 Springer-Verlag, (1987).

Rao1985. Rao, S.K., "Regular Iterative Algorithms and their Implementations on Processor Arrays," *Ph.D. dissertation*, (Stanford University, October 1985).

Rao1988. Rao, S.K. and T. Kailath, "Regular Iterative Algorithms and Their Implementation on Processor Arrays," *Proceedings of the IEEE* **76**(3) pp. 259-269 (March 1988).

Royc1987. Roychowdhurry, V.P., L. Thiele, S.K. Rao, and T. Kailath, "On the Localization of Algorithms for VLSI Processor Arrays," Technical Report, Stanford University, California (1987).

Royc1988. Roychowdhury, V.P. and T. Kailath, "Regular Processor Arrays for Matrix Algorithms with Pivoting," *Proceedings of the International Conference on Systolic Arrays*, pp. 237-246 (May 25-27, 1988).

Schr1985. Schreiber, R. and P.J. Kuekes, "Systolic Linear Algebra Machines in Digital Signal Processing," pp. 389-405 in *VLSI and Modern Signal Processing*, ed. S.Y. Kung, H.J. Whitehouse, T. Kailath, Prentice Hall, Englewood Cliffs (1985).

Unge1958. Unger, S.H., "A computer Oriented Toward Spatial Problems," *Proceedings of the IRE* **46** pp. 1744-1750 (October, 1958).

Walt1971. Walter, J.S., "A Unified Algorithm for Elementary Functions," *Proceedings Spring Joint Computer Conference* **38** p. 397 AFIPS Press, (1971).

Wong1988. Wong, Y. and J.M. Delosme, "Broadcast Removal in Systolic Arrays," *Proceedings of the International Conference on Systolic Arrays*, pp. 403-412 (1988).

SAMENVATTING

STELSELS VAN LINEAIRE VERGELIJKINGEN

De bekende directe methodes (factorisatie gevolgd door terugsubstitutie en "feed forward" directe methodes) voor het oplossen van stelsels van lineaire vergelijkingen zijn niet optimaal voor implementatie op een systolisch array. Zo zijn de directe methodes, die de oplossing van $A\mathbf{x} = \mathbf{b}$ berekenen door middel van factorisatie van A gevolgd door terugsubstitutie, sequentieel van nature. Deze sequentiele geaardheid wordt tot uitdrukking gebracht in het feit dat de terugsubstitutie slechts kan worden uitgevoerd nadat de factorisatie van de matrix A beeindigd is. Hierdoor is de verkregen acceleratie beperkt, in geval dergelijke directe methodes geimplementeerd worden op systolische arrays.

De "feed forward" directe methode van Faddeev berekent de oplossing van $A\mathbf{x} = \mathbf{b}$ via een enkele LU factorisatie. Dergelijke "feed forward" methodes vereisen geen terugsubstitutie, zodat hun implementaties op systolische arrays een hoge mate van parallellisme vertonen. Echter de LU factorisatie zonder (partiele) pivotering is numeriek niet stabiel voor indefiniete matrices. Helaas is (partiele) pivotering moeilijk te implementeren op arrays met interconnecties tussen processor elementen die van constante lengte zijn, zoals het geval is bij systolische arrays. In deze thesis worden nieuwe "feed forward" methodes beschreven. Deze methodes herformuleren de terugsubstitutie in termen van een "updating" of "downdating" van een Cholesky factorisatie, of in termen van een LU factorisatie. De combinatie van deze factorisaties met een LU, LQ of LL^t factorisatie van de coefficienten matrix A leidt tot een klasse van geheel nieuwe "feed forward" directe methodes. In een dezer methodes wordt uitsluitend gebruik gemaakt van circulaire rotaties. Dit leidt ertoe dat deze methode numeriek stabiel en robuust is voor de gehele klasse van niet singuliere stelsels van lineaire vergelijkingen, in tegenstelling tot

Faddeev's "feed forward" directe methode. De hier beschreven nieuwe "feed forward" directe methodes hebben een simpele en in hoge mate parallelle systolische implementatie.

SYSTOLISCHE IMPLEMENTATIE

Een systolisch array is gedefinieerd als een array van processor elementen die synchroon opereren en die door middel van een regelmatig netwerk van interconnecties, waarvan de lengte onafhankelijk is van de grootte van het array, met elkaar verbonden zijn. Deze regelmatige topologie is een wenselijke eigenschap voor compacte VLSI implementaties. Bovendien is de vertraging over interconnecties onafhankelijk van de grootte van het array, doordat de lengte van interconnecties onafhankelijk is van de grootte van het array. Doordat systolische arrays een hoge mate van parallelle verwerking toelaten, wordt er veel aandacht geschonken aan systolische implementaties van rekenintensieve algoritmen. De meeste lineair algebraische, numerieke algoritmen, inclusief de hier beschreven "feed forward" directe methodes, kunnen uitgedrukt worden als reguliere, recurrente algoritmen. Deze kunnen automatisch afgebeeld worden op systolische arrays, waarvan het aantal processor elementen direct proportioneel is met de grootte van het probleem. Echter, omdat het aantal processor elementen van dergelijke arrays van volle grootte meevarieert met de grootte van het probleem, is het belangrijk te weten hoe grote problemen te partitioneren, zodat deze geexecuteerd kunnen worden op een array met een kleiner aantal processor elementen dan het array van volle grootte.

Twee partitioneringsstrategieen worden in de thesis ontwikkeld, ten behoeve van het partitioneren van systolische arrays van volle grootte naar systolische arrays van gereduceerde grootte. De LPGP (local-parallel-global-pipelined) partitioneringsstrategie verdeelt het array van volle grootte in congruente partities van, bijvoorbeeld, p processor elementen elk. De berekeningen die in de verschillende partities plaats vinden, worden in pipeline geexecuteerd op een systolisch array van gereduceerde grootte. Dit array bestaat uit p processor elementen en bezit een netwerk van interconnecties tussen de processor

elementen die dezelfde topologie heeft als het array van volle grootte. De LSGP (local-sequential-global-parallel) partitioneringsstrategie verdeelt het array van volle grootte ook in congruente partities van, bijvoorbeeld, p processor elementen elk. De processor elementen in een partitie worden nu vervangen door een enkel processor element, dat de taken van de p processor elementen in sequentie uitvoert. In dit geval heeft het resulterende array van gereduceerde grootte een topologie die verschillend is van het oorspronkelijke array. De LPGP partitioneringsstrategie heeft tot gevolg dat al het extra benodigd geheugen voor het opslaan van tussenresultaten buiten het array van gereduceerde grootte terecht komt. De LSGP partitioneringsstrategie heeft tot gevolg dat het locaal geheugen aangroeit en de I/O bandbreedte van processor elementen afneemt. Een combinatie van beide strategieen resulteert in ontwerpen van systolische arrays, waarvan het aantal processor elementen onafhankelijk is van het probleem en waarvan de I/O bandbreedte nagenoeg gelijk is aan die van perifere apparaten die op het array aangesloten zijn, zoals disks, de host computer, etc.

SYSTOLISCH ARRAY ONTWERP

In de thesis wordt de partitioneringstheorie toegepast in het ontwerp van een array van gereduceerde grootte. Dit array is in staat twee van de nieuwe "feed forward" directe methodes te executeren; 1) de methode bestaande uit een combinatie van een LQ factorisatie van de coefficienten matrix A en een "updating" van een Cholesky factorisatie en 2) de methode bestaande uit een LL^t factorisatie van de coefficienten matrix A en een "downdating" van een Cholesky factorisatie. Deze toepassing illustreert het practisch nut van de hier gepresenteerde partitioneringstheorie voor het ontwerpen van systolische arrays van gereduceerde grootte. De twee "feed forward" directe methodes zijn geformuleerd in termen van orthogonale en hyperbolische Householder transformaties, respectievelijk. Een processor element van het array van gereduceerde grootte is geimplementeerd als een inproduct processor, die de calculaties voor een Householder transformatie van een vector sequentialiseert.

ABOUT THE AUTHOR

Kishan Jainandunsing was born in Paramaribo, Surinam, on August 15, 1960. In 1978 he graduated cum laude from the Myranda Lyceum in Zorg en Hoop, Surinam. After an introductory 4 months of studying medicine at the Faculty of Medical Sciences in Rotterdam, The Netherlands, he started his studies for electrical engineering in January 1979 at the Delft University of Technology, in Delft, The Netherlands. In July 1984 he received the Ingenieurs' degree (the equivalent of an M.Sc.) cum laude from the Electrical Engineering Department, Delft University of Technology.

In September 1984 Mr. Jainandunsing joined the Laboratory for Network Theory of the EE Department at Delft University of Technology, where he worked towards the Ph.D. degree on parallel algorithms for solving systems of linear equations and their mapping on systolic arrays. Most of the results of this research have been published on international conferences and in the SIAM Journal on Scientific and Statistical Computing, 1989. In the Summer of 1985 he was a research assistant at the EE Department of the University of Southern California (USC), in the group of Professor S.Y. Kung, on a joint research program on partitioning strategies for systolic arrays.

During his Ph.D. research Mr. Jainandunsing held the position of Chairman of the IEEE Student Branch Delft, from 1984 until 1985. He was a co-organizer of the national Symposium on Automated VLSI Design and Interactive Graphics Workstations, May 1985, in Delft, The Netherlands. He also was a co-organizer of the International Workshop on SVD and Signal Processing, September 1987, in Les Houches, France.

STELLINGEN

1. Bij de evaluatie van de executiesnelheid van een algoritme dient altijd expliciet rekening gehouden te houden met het type architectuur waarop het geexecuteerd zal worden.

2. De ontwikkeling van nieuwe algoritmen en nieuwe computer architecturen ondervindt een wederzijdse beinvloeding.

3. Hoewel algoritmen voor het oplossen van stelsels van lineaire vergelijkingen met gebruikmaking van hyperbolische transformaties numeriek inferieur beschouwd worden ten opzichte van algoritmen die uitsluitend gebruik maken van orthogonale transformaties, is het bestaansrecht van de eerstgenoemden te vinden in het feit dat zij een groep van semi-directe methoden vormen die, hetzij de oplossing in een minimum aantal stappen berekenen, hetzij een approximatie van de werkelijke oplossing berekenen.

4. Volledige geautomatiseerde synthese van systolische arrays in het bijzonder en van de synthese van oplossingen in het algemeen kan mogelijk zijn, doch de uitkomst zal niet altijd geaccepteerd worden door de opdrachtgever indien het niet strookt met zijn verwachtingspatroon.

5. De stelling dat er altijd wel een probleem te vinden is dat groter is dan een gegeven computer aankan is door de voorzienigheid gegeven; want, volledige kennis over de invloed van elke parameter in een physisch processes zou tot volledige besluiteloosheid kunnen leiden.

6. Het gebruik van Householder transformaties (reflecties) voor het oplossen van stelsels van lineaire vergelijkingen valt in sommige opzichten te prefereren boven het gebruik van rotaties, omdat men in een systolische implementatie volstaan kan slechts een LPGP partitionering uit te voeren om zo tot een systolisch array te

komen, waarvan het aantal processor elementen onafhankelijk is van de omvang van het stelsel van lineaire vergelijken en waarvan de I/O bandbreedte te beheersen is zonder de noodzaak van een LSGP partitionering.

7. De beschaving zoals wij die ervaren is een illusie. Echte beschaving is onafhankelijk van de beschermingen en vervulling van behoeften die een gemeenschap biedt aan haar leden.

Errata

Page 49 : $(r-p+1)$ must be $(r-p)$

Page 51, Figure 2.3 : γ_{1r} must be $\gamma_{1,r-1}$, γ_{Nr} must be $\gamma_{N-1,r-1}$, γ_{N2} must be $\gamma_{N-1,2}$, γ_{N1} must be $\gamma_{N-1,1}$

Page 76, Figure 3.3(b) : $w=$ must be $w_i=$, $=r$ must be $=r_i$, from selector must be from selector s_i

Page 91 : $c_y = |\mathbf{s}^t\mathbf{t}|/gcd(\mathbf{s}^t\mathbf{t},\alpha_x)$ must be $|\mathbf{s}^t\mathbf{t}|/gcd(\mathbf{s}^t\mathbf{t},\alpha_y)$

Page 149 : ... the first k and $k-1$ *off*-diagonals ... must be ... the first $k-1$ and $k-2$ *off*-diagonals ...

Page 149 : ... with the first $k+1$ *off*-diagonals ... must be ... with the first k *off*-diagonals ...

Page 150 : $S_{k-1} = U_{k-1}^{-1}S_{k-1}$ must be $S_{k-1} = U_{k-1}^{-1}Y_{k-1}$