Abstract Advanced high resolution methods for real-time signal processing require fast multiprocessor architectures, realized as Very Large Scale Integrated systems, for their implementation. A class of parallel architectures, 'systolic arrays', fulfills the demands of signal processing and at the same time conforms to the limitations of VLSI technology.

The design of efficient systems of systolic arrays is an iterative process in which the information gathered from mapping algorithms onto hardware is influential in the development of the algorithms. Presently, the absence of mapping tools makes it extremely difficult to find good systolic implementations for many important problems. Often, one cannot do better than implementing structurally simple but numerically inferior algorithms, which is undesirable for most signal processing tasks.

A methodology is proposed that automates the mapping of recurrence equations to processor arrays. Two aspects distinguish our methodology from extant work : (1) complex ***coupled systems*** of recurrence equations can be systematically treated and (2) the resulting systolic systems are ***optimal***. The methodology takes as input recurrence equations describing the algorithm, along with certain desirable hardware features. A sophisticated optimization procedure is then applied to generate the description of optimal arrays that implement the algorithm. If there is no satisfactory implementation, the algorithm will have to be revised, in order to improve pipelinability, without sacrificing numerical stability. Several classes of applications will significantly benefit from this approach : fast analysis of parallel algorithms, efficient partitioning of algorithms, and determination of optimal architectures for the implementation of multiple algorithms.

Efficient Systolic Arrays for the Solution of Toeplitz Systems : An Illustration of a Methodology for the Construction of Systolic Architectures in VLSI

Jean-Marc Delosme[1], Ilse C.F. Ipsen[2]

Research Report YALEU/DCS/RR-370
June 1985

[1] Department of Electrical Engineering, Yale University
[2] Department of Computer Science, Yale University

The work presented in this paper was supported by the Office of Naval Research under contracts N000014-82-K-0184 and N00014-84-K-0092.

1. Introduction

Systems for such applications as radar and sonar detection and identification, robot vision, medical imaging and geophysical exploration have to extract features from or make decisions based on large amounts of data. Such systems can typically be decomposed into two parts, a front-end and a classification subsystem. The front-end collects the data and performs filtering and parameter estimation to extract useful information from the measurements; the classification sub-system infers from the information forwarded by the front-end the occurrence and nature of certain events.

In order to minimize the decision time or to reduce storage by orders of magnitude, data often must be processed in real time. In order to keep up with the acquisition rate of the uncompressed data, the front-end has to be able to perform several hundred million operations per second, and therefore has to consist of *many* processors. The cost of such systems is minimized when the desired throughput is attained on the smallest number of processors. This is accomplished by implementing 'fast' pipelinable algorithms which allow a minimal operation count as well as chaining of operations in a high efficiency pipeline.

The processors themselves should be fast enough and hence must be special purpose arithmetic computers. For reasons of reliability, cost, size, weight and power consumption, they are presently realized as VLSI chips. Because of their impact on the whole system, some limitations of the VLSI technology must be taken into account at the outset of the design process. The most important are the high design cost for developing novel processor architectures and the limited Input/Output capabilities of chips due to small pin counts (on the order of one hundred) [59]. These constraints combined with the desire for efficient pipelining have led to the concept of 'systolic arrays' [36, 42].

Systolic arrays consist of a few types of processors that perform elementary operations (such as additions or multiplications) selected by a simple control unit. The pattern of interconnections among processors is regular and each processor is connected to only a few other processors. Since the types of operations performed are data independent, the processors can be made to operate synchronously without affecting the global performance; this simplifies the control of the processors and reduces the area needed to implement them in silicon.

The process of front-end design can be decomposed into three steps :

- Development of signal processing techniques. Modern techniques optimize a given criterion (for instance to minimize some error norm) and exploit prior information (often through the choice of a particular statistical model for the incoming signals).
- Development of fast, stable and pipelinable algorithms that perform the optimization. It will often be necessary to increase the number of operations (by introducing redundancy for example) in order to increase stability (permitting simpler processors) and pipelinability (resulting in higher efficiency).
- Mapping of the algorithms onto minimal cost systolic architectures that deliver the desired throughput and comply with the current technological constraints. The best architecture may be a combination of several systolic arrays with different processors or interconnection patterns.

Clearly the three design steps are not completely independent and the best designs will be obtained through an iterative procedure; if there is no satisfactory implementation of an algorithm it will have to be revised. To make such iterations practically feasible the time consuming mapping process must be automated and minimal cost mappings must be determined at every iteration.

Recently there have been many attempts to formalize the usual trial-and-error procedure for the mapping of computations onto an array. (These are surveyed in the Appendix.) Generally these mappings are unduly restricted and, at best, the designs are enumerated; the corresponding techniques do not guarantee the determination of a minimal cost design. However, we believe

that it *is* possible to find *optimal* mappings within a reasonable amount of time, even for complex algorithms that are mapped onto a collection of systolic arrays or for algorithms that require several passes on the same array.

The class of algorithms implemented on front-ends is large. If the optimization criterion consists of minimizing summed squared errors, the algorithms call either for matrix inversions and factorizations, or for matrix eigenvalue or singular value decompositions, depending on the model for the signals [69]. The processing techniques may also require the evaluation of correlations or convolutions. All these algorithms can be expressed as systems of recurrence equations. Due to the wide variety of these systems the goal of the proposed mapping method is to find the optimal mappings for *any* system of recurrence equations.

2. Motivation for a Design Methodology

Up to now most of the time and effort in systolic array design has gone into finding an array that performs the desired computation (the terminology 'systolic array' should be understood in the wider sense as including collections of systolic arrays). If an algorithm had what seemed to be 'too complex' a structure to be mapped on an array, an 'easier' method that appeared to have a better chance of being implementable had to be chosen. Consequently, the numerical performance of the algorithm had to be sacrificed in order to obtain a processor array, frequently one which would execute a numerically inferior, possibly even unstable, method. For most signal processing tasks this kind of inaccurate and unreliable information is unacceptable. Increasing the precision (the number of bits used to represent a numerical value) is usually no remedy : it slows down the computation, increases the area of each processing element, may also render the computation Input/Output bound, and, after all, still does not lead to satisfactory numerical results.

Application of our design methodology will allow the designer of systolic systems to concentrate on the algorithm development, particularly on modifications for improved numerical behavior or pipelinability, rather than on the mapping process. Besides reducing the design time, this will give the designer confidence in the hardware, since the resulting systolic array will be *correct* by construction. Consequently, no *a posteriori* verification process, as described in [10, 12, 38, 39, 51, 52, 53, 70], will be necessary.

Automatic design of systolic arrays has been addressed from different points of view : algebraic manipulations with a time delay operator [13, 26, 27, 28, 34, 37, 71, 72], iterative improvement of a given design by retiming, [33, 40, 41, 42, 44, 45, 46], inductive construction from first order recurrence equations [11], and determination of linear transformations applied to a geometric representation of the recurrence equations [2, 6, 7, 8, 9, 17, 19, 18, 30, 31, 32, 49, 50, 54, 55, 56, 57, 58, 66, 67, 68]. This latter approach determines systolic designs as follows. Taking advantage of the regularity in the recursions, the partial order of the operations in the algorithm is summed up through 'dependence vectors'. There are many ways of evaluating recursions consistent with the partial order. The ones which are systematic and regular can be represented by projecting on a single 'time direction', i.e., there is a global time arrow. Constraints on the possible time directions are easily deduced from the dependence vectors. A similar notion of 'processor direction' allows the projection of operations onto processors in a regular fashion. The only constraint on the processor directions is linear independence with the time direction (which means that no two computations can be performed on the same processor at the same time). By expanding these ideas, our methodology is the only one [18] capable of systematically handling *coupled systems* of recurrence equations as well as being able to generate provably *optimal* systolic arrays.

Optimal designs can be found for various user-specified constraints on the architecture; these may include restriction of Input/Output to boundary processors, specified ordering and timing of input data, avoidance of broadcasting or a limit on the number of processors. Optimality of the mapping is critical, since an inaccurate assessment of the parallelism inherent in an algorithm can

lead the designer to make unnecessary modifications to the algorithm. An array is defined to be optimal if, among all designs satisfying some user-specified hardware constraints, it best meets a *hierarchy* of selected criteria. A hierarchy of objectives to be satisfied in sequence is introduced to precisely delineate the implementation since there are generally many designs that attain the optimum for a single objective. In a real-time implementation, for example, given throughput is a constraint and a minimal number of processors would be a primary objective, while minimal latency and storage could be secondary and tertiary objectives. The use of such a cascade of objectives permits a precise formulation of the designer's *desiderata*, avoids long and overwhelming enumerations of possible solutions, and reduces the computation time needed to find the final design. Still, in most cases these multiple objectives and constraints will not uniquely specify an optimal design, and application of the methodology will result in a succinct description of a *class* of systolic arrays. The information about the members of the class can then be employed for further refinement of the details and 'fine tuning' of the systolic array specifications, such as number of registers per processors, or complexity of control.

3. Applications of the Design Methodology

An important and direct application of the methodology is the determination of architectures on which to implement multiple algorithms. Three more classes of applications for which the design methodology can be most beneficial are illustrated in this section : improvement of algorithm design and analysis, application to complex problems, and problem partitioning.

3.1. Improvement of Algorithm Design and Analysis

Since the methodology will automate the mapping to hardware, more time and effort can be spent on the design and analysis of numerical algorithms. An example is the solution of linear systems of equations with symmetric positive definite Toeplitz coefficient matrix. An algorithm due to Levinson [47] exploits the Toeplitz structure and is routinely used on single-processor machines. Although it consists of only two simple coupled recursions, one to update partial solutions to the system and the other to compute certain inner products, the algorithm does not readily lend itself to a systolic array implementation.

The methodology first establishes that the algorithm cannot admit to an efficient systolic array, because the same vector is used from both ends in the recursion for the partial solution update, thus leading to highly restrictive dependences. This difficulty is resolved by working with two copies of the vector instead of just one [29]. This repetition duplicates certain computations and increases the total operation count by fifty percent, but eliminates some of the more constraining dependences. However, the methodology shows that the modified algorithm still does not admit to an efficient systolic implementation because of the nature of the interdependence between partial solution update and inner product recursions.

Further analysis indicates a way of drastically improving the pipelinability of the Levinson algorithm. This modification of the algorithm takes advantage of the Toeplitz structure when evaluating the inner products (and increases the Levinson operation count by another fifty percent). At this point, one notices that the twice modified Levinson algorithm resembles Schur's algorithm, except for some extraneous computations. This is the reason why Schur's method is used instead of Levinson's, see [1, 4, 5, 14, 35, 62].

Schur's algorithm is a fast way of performing the factorization of the Toeplitz coefficient matrix. To solve a system of equations two more steps are necessary, a forward and a backward substitution. In [15] recursions are derived for these two steps that closely resemble the Schur algorithm and minimize the number of intermediate quantities to be stored. Numerical experiments show that the stability and round-off errors are competitive with those of the best, currently available methods. In [16] the methodology is applied to find efficient systolic arrays for that algorithm under various

user-specified hardware constraints. This example is presented in the next section to illustrate our methodology.

3.2. Application to Complex Problems

Recent years have witnessed the development of a family of multichannel exact least-squares adaptive algorithms [20, 21, 63] and their application to important problems such as the estimation of time-differences of arrival for target tracking, noise cancelling or adaptive equalization. These recursive algorithms exhibit excellent performance and are ideally suited for real-time processing. However, their implementation is not an easy task. The difficulty stems from the fact that they are expressed in terms of matrix operations (the order of the matrices being the number of channels) and that pipelinability requires the proper choice of algorithms for decomposing the matrix operations into scalar operations. This implementation problem displays a tight coupling between algorithm design and mapping of steps. Our methodology will allow fast evaluation of the maximal efficiency inherent in the algorithms and will guide us in exploring alternative decompositions of matrix operations.

3.3. Problem Partitioning

Most algorithms A are formulated in terms of a small number of parameters on which their computation time and hardware requirements depend. For instance, the Gaussian elimination process on a square matrix of size n takes time n^3 on a single processor, or can be implemented in time n on a systolic processor grid of size proportional to n^2 [3].

Of interest here are those problems $A(n)$ whose resources depend on the maximal number n of iterations in a system of recurrence equations (for matrix problems this would be equivalent to dealing with a matrix of order no larger than n). Partitioning the problem $A(n)$ then means to break its equations down and possibly reorder them, so that the resulting problem $\tilde{A}$ can be partitioned into p independent and similar subproblems $\tilde{A}(i)$ which can then be solved p times faster. Partitioning is used, for example, when the number of processors p is strictly smaller than the dimension n of the problem or not all of the input parameters can be input at the same time; it is also of fundamental interest when the hardware is used for problems of differing sizes.

For example, in [25] Gaussian elimination is discussed when each of the $p < n$ processors performs eliminations on n/p (not necessarily adjacent) rows or columns of a $n \times n$ matrix. Heller [23] discusses the decomposition of recursive filters for linear systolic arrays while Moldovan et al. [58] sketch a way of mapping the unshifted QR algorithm for symmetric matrices onto a fixed size rectangular array. In [24] partitioned algorithms and corresponding systolic systems are derived for LU-decompostion, matrix multiplication, triangular system solution and matrix inversion. Priester et al. [64, 65] rearrange the data for banded matrix vector multiplication to fit a fixed size systolic array of [36]. A general survey of partitioning methods and their influence on systolic processor design is discussed in [22] for some basic matrix methods.

On account of the dependence vector concept, the methodology can first determine the *possible* partitionings $\tilde{A}(i)$ compatible with p processors. Each partitioning $\tilde{A}(i)$ is then treated like a set of coupled steps by the optimization algorithm, with the constraint that the resulting array may consist of only p processors.

4. Illustration of the Design Methodology

In order to illustrate our approach an example is given which shows how to systematically derive optimal systolic arrays for the solution of Toeplitz linear systems. The algorithm consists of several steps and hence is complex enough to provide the main ideas for dealing with general systems of coupled recurrence equations.

Our design methodology consists of several tasks; each of the following sections is devoted to one such task. Within each section, the necessary notions and terminology are introduced and the

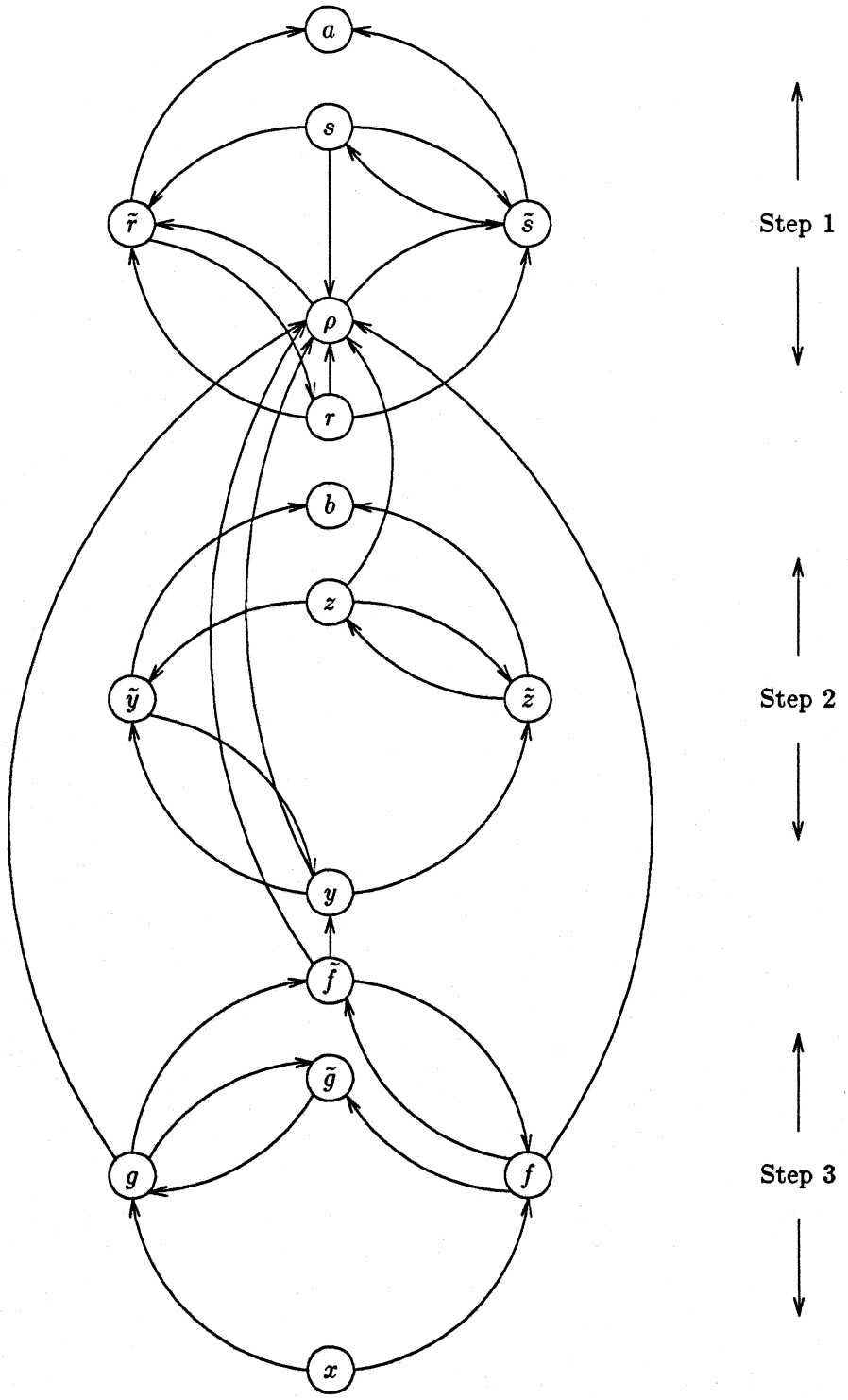


Figure 1: Dependence Graph for Steps 1, 2 and 3.

nature of the task is illustrated by means of the example. We will start with a short description of the algorithm.

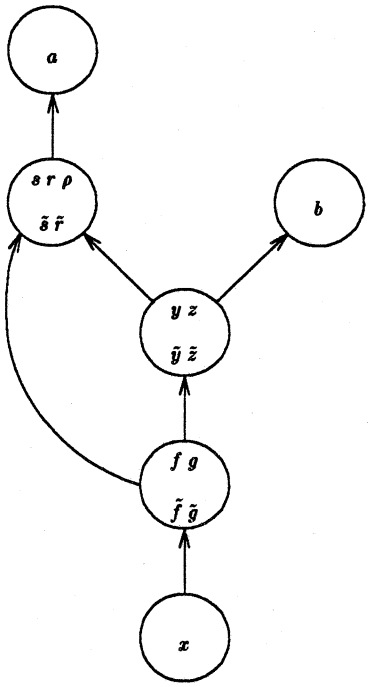


Figure 2: Acyclic Graph Obtained from the Directed Cut-Sets of the Dependence Graph.

4.1. Description of the Toeplitz Hyperbolic Cholesky Solver

The Toeplitz Hyperbolic Cholesky Solver (THCS) [15] has been designed as an efficient, minimal storage, algorithm for the parallel solution of linear systems of equations,

$$Ax = b,$$

where the $n \times n$ coefficient matrix $A = (a_{ij})$ is a symmetric positive definite Toeplitz matrix, $a_{|i-j|} \equiv a_{ij}$, that is, its elements are constant along each diagonal.

Several highly concurrent systolic architectures for the solution of Toeplitz systems have recently been presented [4, 5, 14, 35, 62]. They start with the pipelined factorization of A via variants of an old algorithm due to I. Schur (cf. [1, 60]) but differ in the way these results are used to determine the solution vector. In [15] it was shown how to use the rotation parameters computed in the Schur algorithm so that storage is minimized, and subsequent steps can be easily pipelined and closely resemble the first step.

The first step, given in the recurrences below, computes the Cholesky factor U of A (such that $A = U^T U$) via an elimination procedure based on hyperbolic rotations. The quantities r_{ij} correspond to the elements of the Cholesky factor U, ρ_j are the Schur parameters (hyperbolic tangents of the rotations) and s_{ij} are auxiliary quantities.

Step 1 :

$$
\begin{aligned}
&1 \le i \le n, \quad \tilde{r}_{i,0} \equiv a_{i-1} \\
&\tilde{s}_{1,0} \equiv 0 \\
&2 \le i \le n, \quad \tilde{s}_{i,0} \equiv a_{i-1} \\
&1 \le j \le n, \quad \rho_{j-1} = \tilde{s}_{j,j-1} / \tilde{r}_{j,j-1} \\
&\qquad j \le i \le n, \quad r_{i,j-1} = \tilde{r}_{i,j-1} - \rho_{j-1} \tilde{s}_{i,j-1} \\
&\qquad\qquad\quad s_{i,j-1} = -\rho_{j-1} \tilde{r}_{i,j-1} + \tilde{s}_{i,j-1} \\
&\qquad j+1 \le i \le n, \quad \tilde{r}_{ij} = r_{i-1,j-1} \\
&\qquad\qquad\quad \tilde{s}_{ij} = s_{i,j-1}
\end{aligned}
$$

In essence, the first step performs a premultiplication of A by U^{-T}. In the second step the same operations are applied to the right-hand side vector b. The elements of $y = U^{-T} b$ are the quantities $y_{j,j-1}$.

Step 2 :

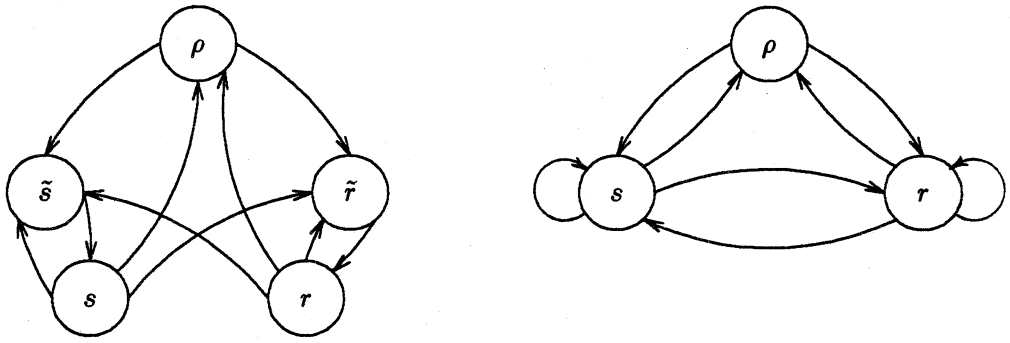
$$
\begin{aligned}
&1 \le i \le n, \quad \tilde{y}_{i,0} \equiv b_i \\
&\qquad\quad \tilde{z}_{i,0} \equiv b_i \\
&1 \le j \le n, \\
&\qquad j \le i \le n, \quad y_{i,j-1} = \tilde{y}_{i,j-1} - \rho_{j-1} \tilde{z}_{i,j-1} \\
&\qquad\qquad\quad z_{i,j-1} = -\rho_{j-1} \tilde{y}_{i,j-1} + \tilde{z}_{i,j-1} \\
&\qquad j+1 \le i \le n, \quad \tilde{y}_{ij} = y_{i,j-1} \\
&\qquad\qquad\quad \tilde{z}_{ij} = z_{i-1,j-1}
\end{aligned}
$$

During the last step, y is essentially premultiplied by U^{-1}. Since U^{-1} is the transpose of U^{-T}, the operations are the same hyperbolic rotations as in Steps 1 and 2 but performed in reverse order. The solution vector $x = U^{-1} y$ is obtained as the sum of f and g. Note that of all the quantities computed in Step 1 only the Schur parameters ρ_{j-1} and the diagonal elements $r_{j,j-1}$ of U, hence just $O(n)$ quantities, are used in later steps.

Step 3 :

$$
\begin{aligned}
&1 \le j \le n, \quad \tilde{f}_{n-j+1,j-1} \equiv y_{n-j+1,n-j} / r_{n-j+1,n-j} \\
&\qquad\quad \tilde{g}_{n,j-1} \equiv 0 \\
&\qquad\quad n-j+2 \le i \le n, \quad \tilde{f}_{i,j-1} = f_{i,j-1} \\
&\qquad\qquad\qquad\quad \tilde{g}_{i-1,j} = g_{i,j} \\
&\qquad\quad n-j+1 \le i \le n, \quad f_{i,j} = \tilde{f}_{i,j-1} - \rho_{n-j} \tilde{g}_{i,j-1} \\
&\qquad\qquad\qquad\quad g_{i,j} = -\rho_{n-j} \tilde{f}_{i,j-1} + \tilde{g}_{i,j-1} \\
&1 \le i \le n, \quad x_i = f_{i,n} + g_{i,n}
\end{aligned}
$$

Associated with the algorithm is a *dependence graph* whose nodes are the variables in the algorithm and whose edges represent the dependences between variables, see Figure 1. The decomposition of an algorithm into steps is determined from the *directed cut-sets* of this graph. Here, the variables are decomposed into six groups, where the graph of the dependences between these groups is acyclic, see Figure 2. These groups define six steps (the functional decomposition into three steps used to present the example is a simplification). The dependences between variables within a group are called *internal* dependences while the ones between groups are called *external* dependences.



(a) (b)

Figure 3: Dependence Graph for Step 1 Before (a) and After (b) Removal of Redundant Variables.

Renaming occurs when a variable depends on exactly one other variable within the same group (though it could depend on other variables outside the group). In Step 1, for instance, the variables $\tilde{r}$ and $\tilde{s}$ serve no other purpose but renaming and introduce unnecessary dependences, see Figure 3. Upon their removal the equations of the first step are rewritten as

Step 1 :

$$
\begin{aligned}
&1 \le i \le n, && r_{i-1,-1} \equiv a_i \\
&s_{1,-1} \equiv 0 && \\
&2 \le i \le n, && s_{i,-1} \equiv a_{i-1} \\
&1 \le j \le n, && \rho_{j-1} = s_{j,j-2}/r_{j-1,j-2} \\
&\qquad j \le i \le n, && r_{i,j-1} = r_{i-1,j-2} - \rho_{j-1} s_{i,j-2} \\
&\qquad && s_{i,j-1} = -\rho_{j-1} r_{i-1,j-2} + s_{i,j-2}
\end{aligned}
$$

Similarly, the removal of redundant variables in the other two steps yields the equations below :

Step 2 :

$$
\begin{aligned}
&1 \le i \le n, && y_{i,-1} \equiv b_i \\
&\qquad && z_{i-1,-1} \equiv b_i \\
&1 \le j \le n, j \le i \le n, && y_{i,j-1} = y_{i,j-2} - \rho_{j-1} z_{i-1,j-2} \\
&\qquad && z_{i,j-1} = -\rho_{j-1} y_{i,j-2} + z_{i-1,j-2}
\end{aligned}
$$

Step 3 :

$$
\begin{aligned}
&1 \le j \le n, && f_{n-j+1,j-1} = y_{n-j+1,n-j}/r_{n-j+1,n-j} \\
&\qquad && g_{n+1,j-1} = 0 \\
&\qquad n-j+1 \le i \le n, && f_{i,j} = f_{i,j-1} - \rho_{n-j} g_{i+1,j-1} \\
&\qquad && g_{i,j} = -\rho_{n-j} f_{i,j-1} + g_{i+1,j-1} \\
&1 \le i \le n, && x_i = f_{i,n} + g_{i,n}
\end{aligned}
$$

In general, the decomposition of an algorithm into steps and the detection of redundant variables is a two-pass procedure on the graph of the dependences in the algorithm.

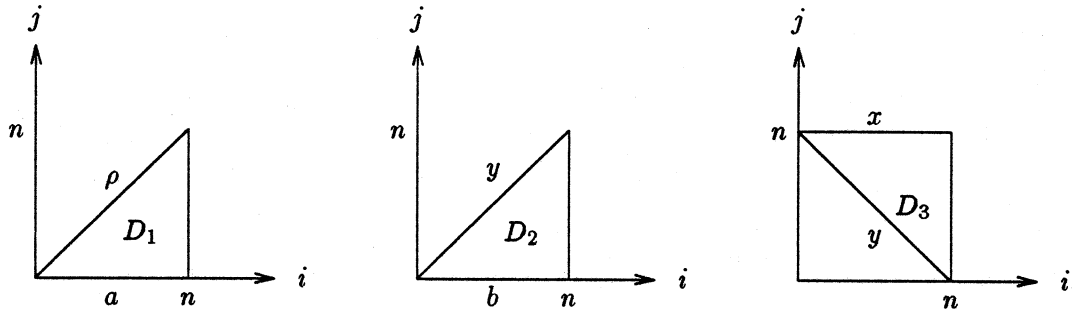


Figure 4: Domains of Computation for Steps 1, 2 and 3.

4.2. Internal Dependence Vectors and Local Transformations

The first task of the methodology, a concise formalization of the input, is accomplished by expressing the dependences *within* each step mathematically. All variables in the above recurrence equations have at most two indices. If u_{ij} is a variable on the lefthand side of a recurrence equation then its computation can be visualized as taking place at coordinate (i,j) in two-space. The set of points corresponding to all the computed variables in Step k will be called its ***domain of computation*** D_k, $k = 1,2,3$.

In this case, the domains of computation D_k take the form of triangles and are depicted in Figure 4. Within each domain, computations occurring at the same point (i,j) are performed in the same processor during the same clock cycle while computations occurring at different points (i,j) will be executed in different processors or different cycles. In Figure 4 sides of the triangles D_k are distinguished by labels corresponding to quantities computed or input at the sides. The recurrence equations in Step k can be represented as a directed, regular acyclic computation graph. The nodes of the graph are the points of D_k, its edges are called *internal dependence vectors* [55, 56]. The latter relate the indices of variables *within* the group of variables associated with Step k as follows : for each variable u_{ij} on the lefthand side of an equation the dependence vectors are the vectors from point (i,j) toward the indices of the quantities needed for that computation, i.e., of the variables on the right-hand side of that equation. In contrast to the more general notion of 'dependence' which is used to determine steps and detect redundant variables, dependence vectors are derived from the *indices* of the variables. Internal data dependences define the valid orders of evaluating the recurrence equations within a step.

A vector τ is a *time vector* for Step k if and only if any two points x and y in D_k for which the computation at y precedes the computation at x satisfy $y^T\tau < x^T\tau$ (here and in the sequel, all vectors are column vectors). It can be shown that a time vector τ satisfies

$$\delta^T\tau < 0$$

for all internal dependence vectors δ ('causality constraint'). Note the strict inequality in the above characterization. If $\delta^T\tau = 0$ for some δ either broadcast or 'rippling', an $O(n)$ number of operations sequentially applied to the same datum in one clock cycle, would occur. With the strict inequality the clock cycle is always on the order of the time needed to perform an elementary operation (division, multiply-and-accumulate) and is essentially independent of the number of processors.

It follows from the causality constraint that the time vectors for Step k form a convex cone C_k (the origin of this notion goes back to the theory of multi-dimensional systems [48]). The 'wider' the cone C_k, the more choices for the evaluation of the recurrences in Step k exist. Thus, if one objective is to minimize computation time, the freedom in assigning operations to points in D_k should be exploited in order to obtain a time cone as wide as possible. Next, assignments of

operations within the domains of computation for the three steps of THCS are derived in such a way that the associated time cones are maximal.

It is important to observe that there are many ways of writing the recursions in each step. For example, in Step 1 one could translate the indices of s while leaving unchanged the indices of r, e.g., $s_{i,j-1}$ would become s_{ij} and the inner recursion in Step 1 would take the form

$$\begin{aligned} r_{i,j-1} &= r_{i-1,j-2} - \rho_{j-1} s_{i,j-1} \\ s_{ij} &= -\rho_{j-1} r_{i-1,j-2} + s_{i,j-1}. \end{aligned}$$

The dependence vectors change accordingly, and so does the time cone for Step 1. Since the coordinates of the points in the domains of computations are integers, the vectors and transformations considered now and in the sequel are all integer-valued. In general, if the recursions contain variables coupled through dependence vectors of length $o(n)$, one should only consider 'small' relative translations of $o(n)$ for the indices of these variables. If the relative transformation were an affine linear transformation that either is a translation of size $O(n)$ or is not a translation the time cone would actually become smaller. The relative translations of size $o(n)$ on the indices of the variables within a step are ***local transformations***.

The local transformations yielding the largest time cone impose minimal constraints on the data flow, and minimum time designs are obtained by assuming that such a transformation has been selected. Since the boundaries of the domains of computation may differ slightly, by $o(n)$, for these various transformations, general area/time results are expressed up to additive terms of size $o(n)$ (the stated results hold for arbitrary, sufficiently large n). If there are specific constraints on the data flow (e.g., limitations on interprocessor connections such as a fixed interconnection pattern for all the steps) the local transformations will be selected to maximize the time cones under these constraints.

For the recursions of Step 1, an example of a local transformation inducing minimal constraints is one which leaves the indices of s unchanged (with respect to r) and assigns the computation of ρ_{j-1} to point $(j, j-1)$, i.e., ρ_{j-1} becomes $\rho_{j,j-1}$. Alternatively, the computation of ρ_{j-1} could be assigned to $(j, j-2)$, resulting in equivalent, least restrictive, constraints on the data flow.

The dependence vectors for Step 1 are then

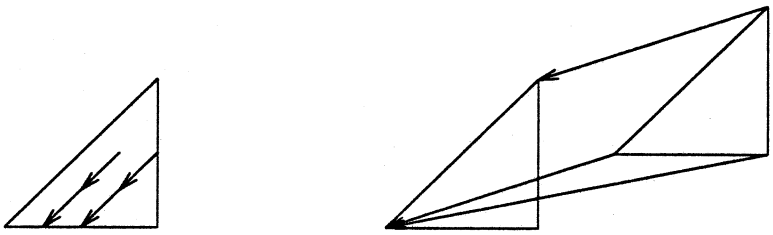
$$\begin{pmatrix} -1 \\ -1 \end{pmatrix}, \quad \begin{pmatrix} 0 \\ -1 \end{pmatrix}, \quad \begin{pmatrix} j-i \\ 0 \end{pmatrix}, \qquad 1 \le j \le i \le n,$$

which can be summarized in the matrix

$$\Delta_1 = \begin{pmatrix} -1 & 0 & 0 & -1 & -2 & \dots & 1-n \\ -1 & -1 & 0 & 0 & 0 & \dots & 0 \end{pmatrix}.$$

The $(0 \quad 0)^T$ vector constitutes a constraint within a processor : the computation of $r_{j,j-1}$ requires ρ_{j-1} (both quantities can actually be computed in parallel with a non-restoring division algorithm). The $(-2 \quad 0)^T \dots (1-n \quad 0)^T$ vectors all have the same orientation and direction as the $(-1 \quad 0)^T$ vector and do not add more constraints to the optimization problem. Since the vector $(-1 \quad -1)^T$ is inside the convex cone spanned by the vectors $(-1 \quad 0)^T$ and $(0 \quad -1)^T$, the dependence vector matrix for Step 1 may be further reduced to

$$\Delta_1 = \begin{pmatrix} -1 & 0 \\ 0 & -1 \end{pmatrix}$$



(a) (b)

Figure 5: Examples of Dependence Vectors, (a) Internal, (b) External.

for time optimization purposes. A similar treatment of the dependence vectors for Steps 2 and 3 results in dependence vector matrices

$$\Delta_2 = \begin{pmatrix} -1 & 0 \\ -1 & -1 \end{pmatrix}, \quad \Delta_3 = \begin{pmatrix} 1 & 0 \\ -1 & -1 \end{pmatrix}.$$

Local transformations influence the size of the time cone. A maximal time cone, i.e., one that is as 'wide' as possible, induces minimal constraints on the data flow and hence results in designs of minimal computation time. Finding the local transformations that result in a maximal time cone constitutes an optimization problem. We believe that for general systems of recurrence equations, this problem can be viewed as a particular graph problem similar to the one arising in optimal placement for river routing [43] and hence can be solved efficiently.

4.3. External Dependence Vectors and Global Transformations

In order to jointly position all three domains D_k in space-time for the optimal realization of THCS, separate *global* transformations are symbolically applied to each D_k. The purpose of these transformations is to 'stretch' or 'skew' the domains of computation and pack them as tightly as possible, so that the computation time and number of processors are kept to a minimum. In doing so, one must take into account both 'internal' and 'external' dependences, the latter referring to the dependences *between* steps, see Figure 5. In order to compute all steps along a certain time direction without violating causality, that time direction should be inside the time cones of all steps as well as inside the time cone defined by the external dependences. This establishes precise constraints on the possible stretchings and skewings for the steps as well as on their respective placements, and therefore on the parameters of the global transformations. Thus, the values of the global transformations will be determined by an optimization procedure subject to these causality constraints.

The global transformations for Step k are defined through

$$T_k \begin{pmatrix} x \\ y \end{pmatrix} \equiv \begin{pmatrix} a_k & b_k \\ c_k & d_k \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} u_k \\ v_k \end{pmatrix},$$

with $a_k d_k - b_k c_k \neq 0$, as shown below. Nonsingularity is required to avoid mapping operations corresponding to different points in the domain of computation onto the same processor at the same time. The matrices of extreme internal dependence vectors Δ_k are transformed to

$$\Delta'_k \equiv T_k \Delta_k,$$

where

$$\Delta'_1 = \begin{pmatrix} -a_1 & -b_1 \\ -c_1 & -d_1 \end{pmatrix}, \quad \Delta'_2 = \begin{pmatrix} -a_2 - b_2 & -b_2 \\ -c_2 - d_2 & -d_2 \end{pmatrix}, \quad \Delta'_3 = \begin{pmatrix} a_3 - b_3 & -b_3 \\ c_3 - d_3 & -d_3 \end{pmatrix}.$$

The domains of computation are mapped into

$$T_1 \begin{pmatrix} 0 & n & n \\ 0 & 0 & n \end{pmatrix} = \begin{pmatrix} 0 & a_1 n & a_1 n + b_1 n \\ 0 & c_1 n & c_1 n + d_1 n \end{pmatrix}$$

$$T_2 \begin{pmatrix} 0 & n & n \\ 0 & 0 & n \end{pmatrix} = \begin{pmatrix} u_2 & a_2 n + u_2 & a_2 n + b_2 n + u_2 \\ v_2 & c_2 n + v_2 & c_2 n + d_2 n + v_2 \end{pmatrix}$$

$$T_3 \begin{pmatrix} n & 0 & n \\ 0 & n & n \end{pmatrix} = \begin{pmatrix} a_3 n + u_3 & b_3 n + u_3 & a_3 n + b_3 n + u_3 \\ c_3 n + v_3 & d_3 n + v_3 & c_3 n + d_3 n + v_3 \end{pmatrix},$$

where without loss of generality u_1 and v_1 were set to zero. The extreme external dependence vectors between Steps 1 and 2 for the variable ρ_{j-1} are given by Δ'_{1-2}, between Steps 1 and 3 for variables ρ_{j-1} and $r_{j,j-1}$ by Δ'_{1-2}, and between Steps 2 and 3 for the variables $y_{j,j-1}$ by Δ'_{2-3}, where

$$\Delta'_{1-2} = \begin{pmatrix} -u_2 & -a_2 n - u_2 & a_1 n + b_1 n - a_2 n - b_2 n - u_2 \\ -v_2 & -c_2 n - v_2 & c_1 n + d_1 n - c_2 n - d_2 n - v_2 \end{pmatrix}$$

$$\Delta'_{1-3} = \begin{pmatrix} -b_3 n - u_3 & -a_3 n - b_3 n - u_3 & a_1 n + b_1 n - a_3 n - u_3 \\ -d_3 n - v_3 & -c_3 n - d_3 n - v_3 & c_1 n + d_1 n - c_3 n - v_3 \end{pmatrix}$$

$$\Delta'_{2-3} = \begin{pmatrix} a_2 n + b_2 n + u_2 - a_3 n - u_3 & u_2 - b_3 n - u_3 \\ c_2 n + d_2 n + v_2 - c_3 n - v_3 & v_2 - d_3 n - v_3 \end{pmatrix}.$$

Without loss of generality the time vector in the transformed domain may be selected as $\tau = [1 \quad 0]^T$. The causality constraints $\delta'^T \tau < 0$ for all transformed dependence vectors δ' are thus summed up by requiring the first components in the dependence vector matrix

$$\Delta = (\Delta'_1 \quad \Delta'_2 \quad \Delta'_3 \quad \Delta'_{1-2} \quad \Delta'_{1-3} \quad \Delta'_{2-3})$$

to be negative :

(1) $-a_1 < 0, \quad -b_1 < 0$

(2) $-a_2 - b_2 < 0, \quad -b_2 < 0$

(3) $a_3 - b_3 < 0, \quad -b_3 < 0$

(1-2) $-u_2 < 0, \quad -a_2 n - u_2 < 0, \quad a_1 n + b_1 n - a_2 n - b_2 n - u_2 < 0$

(1-3) $-b_3 n - u_3 < 0, \quad -a_3 n - b_3 n - u_3 < 0, \quad a_1 n + b_1 n - a_3 n - u_3 < 0$

(2-3) $a_2 n + b_2 n + u_2 - a_3 n - u_3 < 0, \quad u_2 - b_3 n - u_3 < 0$

4.4. Optimization

The computation time for THCS is given by the difference of the times between the first and the last computation,

$$t_T \equiv \max_{(i,j) \in \cup D'_k} \{i\} - \min_{(i,j) \in \cup D'_k} \{i\},$$

where $\cup D'_k$ is the union of the transformed domains of computation for Steps 1, 2, and 3. Systematic manipulations of the above inequalities show that the total computation time amounts to

$$t_T = \max\{b_3 n + u_3, \; a_3 n + b_3 n + u_3\}.$$

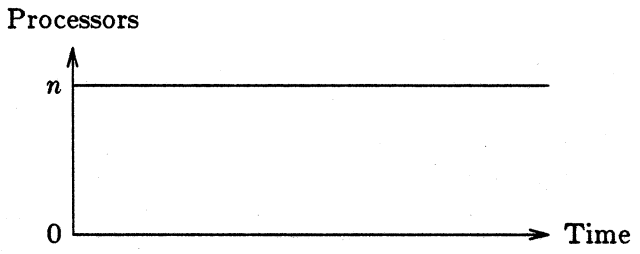


Figure 6: Horizontal Band for Arrays with Minimal Processor Count.

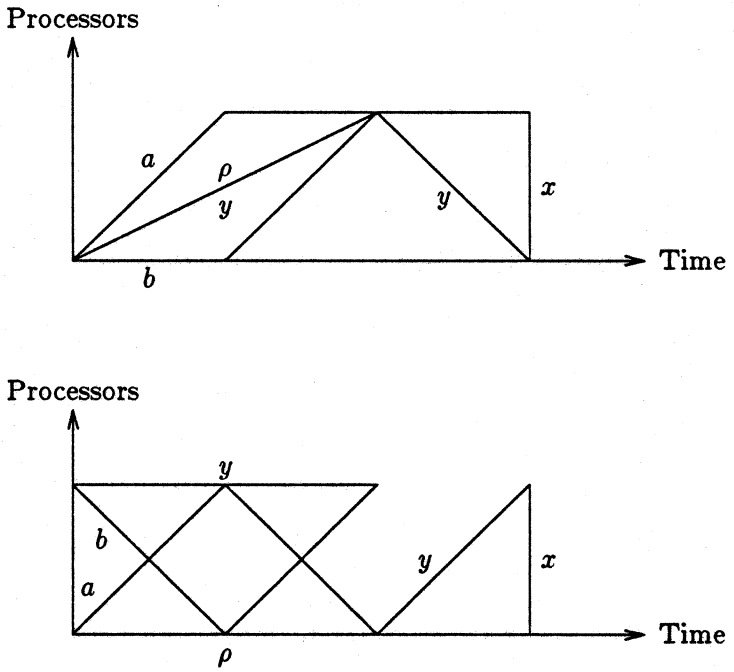


Figure 7: Two Examples of Systolic Arrays with n Processors and Computation Time $3n$.

Further manipulation,

$$\begin{aligned} 0 < b_1 n < a_1 n + b_1 n < a_2 n + b_2 n + u_2 \\ < a_3 n + u_3 < a_3 n + b_3 n + u_3 \\ \leq t_T, \end{aligned}$$

and the fact that the transformations are integer-valued imply

$$t_T \geq 3n.$$

Thus, the total computation time is bounded below by $3n + o(n)$.

There are in general many transformations resulting in designs that achieve this lower bound and hence minimize the computation time. They are characterized by solving for the parameters a_k, b_k and u_k, $k = 1, 2, 3$, the optimization problem

$$\min t_T \tag{4.1}$$

subject to the causality constraints. Because the objective function t_T is the maximum of two linear functions and the constraints are linear, the optimization can be cast as a standard linear integer programming problem. To fully determine the transformations T_k, the remaining parameters c_k, d_k and v_k are found subject to the conditions that no two computations from the same step are performed in the same processor at the same time ('nonsingularity constraint'), and that no two computations from different steps are performed in the same processor at the same time ('no-overlap constraint').

It is natural now to ask which of the arrays, among all those with minimal computation time, possess the minimal number of processors. Clearly, the minimal number of processors is n, assuming that no 'problem partitioning' is undertaken. It then follows that the transformed domains of computation of all the steps must belong to a horizontal band of width n, as drawn in Figure 6, leading to a set of constraints on the possible values of the parameters c_k, d_k and v_k :

(1) $0 \le c_1 n \le n, 0 \le c_1 n + d_1 n \le n$

(2) $0 \le v_2 \le n, 0 \le c_2 n + v_2 \le n, \quad 0 \le c_2 n + d_2 n + v_2 \le n$

(3) $0 \le c_3 n + v_3 \le n, 0 \le d_3 n + v_3 \le n, \quad 0 \le c_3 n + d_3 n + v_3 \le n.$

Solving the optimization problem (4.1) subject to these constraints combined with the causality constraints for a_k, b_k and u_k, the condition that T_k be nonsingular and the no-overlap constraint results in thirty-two systolic arrays with minimal computation time $3n$ and minimal processor count n :

either

$$\begin{bmatrix} a_1 & b_1 \\ c_1 & d_1 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}, \quad \begin{bmatrix} a_2 & b_2 & u_2 \\ c_2 & d_2 & v_2 \end{bmatrix} = \left\{ \begin{bmatrix} 1 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} -1 & 2 & n \\ -1 & 1 & n \end{bmatrix} \right.$$

or

$$\begin{bmatrix} a_1 & b_1 \\ c_1 & d_1 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}, \quad \begin{bmatrix} a_2 & b_2 & u_2 \\ c_2 & d_2 & v_2 \end{bmatrix} = \left\{ \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} -1 & 2 & n \\ 0 & 1 & 0 \end{bmatrix} \right.$$

or

$$\begin{bmatrix} a_1 & b_1 \\ c_1 & d_1 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}, \quad \begin{bmatrix} a_2 & b_2 & u_2 \\ c_2 & d_2 & v_2 \end{bmatrix} = \left\{ \begin{bmatrix} 1 & 1 & 0 \\ -1 & 1 & n \end{bmatrix} \begin{bmatrix} -1 & 2 & n \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 & 1 & 0 \\ 1 & -1 & 0 \end{bmatrix} \begin{bmatrix} -1 & 2 & n \\ -1 & 0 & n \end{bmatrix} \right.$$

together with

$$\begin{bmatrix} a_3 & b_3 & u_3 \\ c_3 & d_3 & v_3 \end{bmatrix} = \left\{ \begin{bmatrix} 0 & 1 & 2n \\ -1 & -1 & 2n \end{bmatrix} \begin{bmatrix} 0 & 1 & 2n \\ -1 & 0 & n \end{bmatrix} \begin{bmatrix} 0 & 1 & 2n \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 0 & 1 & 2n \\ 1 & 1 & -n \end{bmatrix} \right. .$$

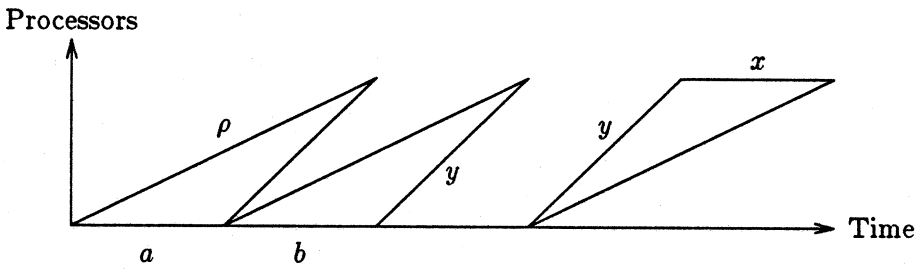


Figure 8: An Example of a Systolic Array with Computation Time $5n$, n Processors and Input/Output on the two Boundary Processors.

Figure 7 depicts the space-time representation for two such arrays. Note, that enforcing the non-overlap of computations is more involved than just ensuring that the polygons which bound the domains of computation are not superimposed.

The one characteristic shared by all of the above designs is that Input/Output, i.e., communication with the outside world, takes place in *each* processor. In case this is undesirable, the next step then consists of determining the class of arrays with the minimal number of processors among those of shortest computation time subject to the constraint that Input/Output may take place on at most two processors. The requirement for a minimal processor count and the Input/Output constraint reduce the set of possible choices for the parameters c_k, d_k and v_k to

$$[c_1 \quad d_1] = [0 \quad 1], \quad [c_2 \quad d_2 \quad v_2] = \begin{cases} [0 & -1 & n] \\ [0 & 1 & 0] \end{cases}, \quad [c_3 \quad d_3 \quad v_3] = \begin{cases} [0 & -1 & n] \\ [0 & 1 & 0] \end{cases}.$$

Solving the subsequent optimization problem demonstrates that the minimal computation time on an array with n processors and Input/Output restricted to the two boundary processors comes to $5n + o(n)$ and is attained for the following thirty-two designs :
either

$$\begin{bmatrix} a_1 & b_1 \\ c_1 & d_1 \end{bmatrix} = \begin{cases} \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \\ \begin{bmatrix} 1 & 2 \\ 0 & 1 \end{bmatrix} \end{cases}, \quad \begin{bmatrix} a_2 & b_2 & u_2 \\ c_2 & d_2 & v_2 \end{bmatrix} = \begin{cases} \begin{bmatrix} 1 & 1 & n \\ 0 & 1 & 0 \end{bmatrix} \\ \begin{bmatrix} -1 & 2 & 2n \\ 0 & 1 & 0 \end{bmatrix} \end{cases}$$

or

$$\begin{bmatrix} a_1 & b_1 \\ c_1 & d_1 \end{bmatrix} = \begin{bmatrix} 2 & 1 \\ 0 & 1 \end{bmatrix}, \quad \begin{bmatrix} a_2 & b_2 & u_2 \\ c_2 & d_2 & v_2 \end{bmatrix} = \begin{cases} \begin{bmatrix} 2 & 1 & 0 \\ 0 & 1 & 0 \end{bmatrix} \\ \begin{bmatrix} 2 & 1 & 0 \\ 0 & -1 & n \end{bmatrix} \\ \begin{bmatrix} -2 & 3 & 2n \\ 0 & 1 & 0 \end{bmatrix} \\ \begin{bmatrix} -2 & 3 & 2n \\ 0 & -1 & n \end{bmatrix} \end{cases}$$

together with

$$\begin{bmatrix} a_3 & b_3 & u_3 \\ c_3 & d_3 & v_3 \end{bmatrix} = \begin{cases} \begin{bmatrix} -1 & 1 & 4n \\ 0 & -1 & n \end{bmatrix} \\ \begin{bmatrix} 1 & 2 & 2n \\ 0 & -1 & n \end{bmatrix} \\ \begin{bmatrix} -1 & 1 & 4n \\ 0 & 1 & 0 \end{bmatrix} \\ \begin{bmatrix} 1 & 2 & 2n \\ 0 & 1 & 0 \end{bmatrix} \end{cases} .$$

A space-time representation for one possible design is given in Figure 8.

5. Summary

Our methodology extracts data dependences from the recurrence equations to summarize the information about all possible ways in which these equations can be evaluated. The optimal design is selected from these possibilities by symbolically applying linear (affine) transformations to the computation graph of the equations and then determining the transformation parameters so that the resulting array is optimal with respect to a hierarchy of objectives. Hardware regularity is obtained by exploiting the regularity of the data dependences, that is, by taking advantage of their invariance under translation in a geometric representation of the computations. The employed transformations preserve the regularity and, in some cases, might even be used to enhance it.

When minimization of computation time is the main objective, decomposition of the optimization problem into separate local and global optimization problems has parallels in the design of MOS networks [61]. In both cases, the internal dependences within each step (equivalent to areas of intraconnections within a cell) can be optimized with almost no consideration for the external dependences between steps (equivalent to interconnections outside the cells). For other hierarchies of objectives, the decomposition into separate optimization problems for local and global transformations may not be possible, which could translate into a *quadratic* programming problem. We have come across techniques that solve these quadratic problems in an acceptable amount of time.

Although the methodology was illustrated by way of an example involving only two-index variables, it should be clear that there is no conceptual difficulty in applying it to recursions on more indices. The most general case involves several time and processor directions, where multiple time directions are best understood as directions for scanning the operations and are mapped accordingly onto the physical time.

Acknowledgments

We are pleased to thank Mark Angevine, Stan Eisenstat and Alfred Ganz for helpful discussions and comments.

References

[1] Ahmed, H.M., Delosme, J.-M. and Morf, M., *Highly Concurrent Computing Structures for Matrix Arithmetic and Signal Processing,* IEEE Computer, 15 (1982), pp. 65–82.

[2] Andre, F., Frison, P. and Quinton, P., *Algorithmes Systoliques : De la Theorie a la Pratique,* Research Report 196, IRISA, Rennes-Cedex, France, 1983.

[3] Bojanczyk, A., Brent, R.P. and Kung, H.T., *Numerically Stable Solution of Dense Systems of Linear Equations Using Mesh-Connected Processors,* SIAM J. Sci. Stat. Comp., 5 (1984), pp. 95–104.

[4] Brent, R.P., Kung, H.T. and Luk, F.T., *Some Linear-Time Algorithms for Systolic Arrays,* Technical Report TR-CS-82-15, Dept. of Computer Science, The Australian National University, 1982.

[5] Brent, R.P. and Luk, F.T., *A Systolic Array for the Linear-Time Solution of Toeplitz Systems of Equations,* Technical Report CS-83-02, Dept. of Computer Science, The Australian National University, 1983.

[6] Cappello, P.R., *VLSI Architectures for Digital Signal Processing,* Ph.D. Thesis, Princeton University, 1982.

[7] Cappello, P.R. and Steiglitz, K., Unifying VLSI Array Design with Linear Transformations of Space-Time, *Advances in Computing Research,* JAI Press, 1984, pp. 23–65.

[8] ————, Unifying VLSI Array Design with Geometric Transformations, *Proc. Int. Conf. Parallel Processing,* 1983, pp. 448–57.

[9] ————, Selecting Systolic Designs Using Linear Transformations of Space-Time, *Proc. SPIE Symp. 549 (Real Time Signal Processing VII),* 1984, pp. 75–85.

[10] Chen, M.C., *Space-Time Algorithms : Semantics and Methodology,* Ph.D. Thesis, California Institute of Technology, 1983.

[11] ————, *A Synthesis Method for Systolic Designs,* Research Report 334, Dept. of Computer Science, Yale University, 1985.

[12] Chen, M.C. and Mead, C.A., Concurrent Algorithms as Space-Time Recursion Equations, *Proc. USC Workshop VLSI Modern Signal Processing,* 1982, pp. 31–52.

[13] Cohen, D., Mathematical Approach to Iterative Computational Networks, *Proc. Fourth Symp. Computer Arithmetic,* 1978, pp. 226–38.

[14] Delosme, J.-M., *Algorithms for Finite Shift-Rank Processes,* Ph.D. Thesis, Dept of Electrical Engineering, Stanford University, 1982.

[15] Delosme, J.-M. and Ipsen, I.C.F., *Efficient Parallel Solution of Linear Systems of Equations with Hyperbolic Rotations,* Research Report 341, Dept. of Computer Science, Yale University, 1984.

[16] ————, Efficient Systolic Arrays for the Solution of Toeplitz Systems : An Illustration of a Methodology for the Construction of Systolic Architectures in VLSI, *Proc. Second Int. Symposium on VLSI Technology, Systems and Applications,* Taipei, Taiwan, 1985, pp. 268–73.

[17] Fortes, J.A.B., *Algorithm Transformations for Parallel Processing and VLSI Architecture Design,* Ph.D. Thesis, Dept of Electrical Engineering Systems, USC Los Angeles, 1983.

[18] Fortes, J.A.B., Fu, K.S. and Wah, B.W., Systematic Approaches to the Design of Algorithmically Specified Systolic Arrays, *Proc. ICASSP,* 1985, pp. 8.9.1–4.

[19] Fortes, J.A.B and Moldovan, D.I., *Parallelism Detection and Transformation Techniques Useful for VLSI Algorithms,* J. Parallel Distributed Comp., (1985).

[20] Friedlander, B., Lattice Filters for Adaptive Processing, *Proc. IEEE,* 1982, pp. 829–67.

[21] ————, Lattice Methods for Spectral Estimation, *Proc. IEEE,* 1982, pp. 990–1017.

[22] Heller, D.E., Partitioning Big Matrices for Small Systolic Arrays, *Proc. USC Workshop VLSI Modern Signal Processing,* 1982.

[23] ————, Decomposition of Recursive Filters for Linear Systolic Arrays, *Proc. SPIE (Real Time Signal Processing VI),* 1983.

[24] Hwang, K. and Cheng, Y.H., Partitioned Algorithms and VLSI Structures for Large-Scaled Matrix Computations, *Proc. Fifth Symp. Comp. Arithmetic,* IEEE, 1981, pp. 222–32.

[25] Ipsen, I.C.F., Saad, Y. and Schultz, M.H., *Complexity of Dense Linear System Solution on a Multiprocessor Ring,* Research Report 349, Dept. of Computer Science, Yale University, 1984.

[26] Johnsson, L. and Cohen, D., A Mathematical Approach to Modelling the Flow of Data and Control in Computational Networks, *CMU Conference on VLSI Systems and Computations,* Computer Science Press, 1981, pp. 213–25.

[27] Johnsson, L., Weiser, U., Cohen, D. and Davis, A., Towards a Formal Treatment of VLSI Arrays, *Proc. Second Caltech Conf. VLSI,* 1981.

[28] Jover, J.-M. and Kailath, T., Design Framework for Systolic-Type Arrays, *Proc. ICASSP,* 1984, pp. 8.5.1–4.

[29] Kailath, T., *A View of three Decades of Linear Filtering Theory,* IEEE Trans. Inf. Theory, IT-20 (1974), pp. 146–81.

[30] King, R.M. and Brown, T.C., Research on Synthesis of Concurrent Computing Systems, *Proc. Tenth Ann. Symp. Computer Architecture,* 1983, pp. 39–46.

[31] Kuhn, R.H., Efficient Mapping of Algorithms to Single-Stage Interconnections, *Proc. Seventh Ann Symp. Computer Architecture,* 1980, pp. 182–9.

[32] ————, Transforming Algorithms for Single-Stage and VLSI Architectures, *Proc. Workshop Interconnection Networks for Parallel and distributed Processing,* 1980, pp. 11–9.

[33] Kung, S.-Y., On Supercomputing with Systolic/Wavefront Array Processors, *Proc. IEEE,* 1984, pp. 867–84.

[34] Kung, S.-Y., Arun, K.S., Gal-Ezer, R.J. and Bhaskar Rao, D.V, *Wavefront Array Processor : Language, Architecture, and Applications,* IEEE Trans. Comp., C-31 (1982), pp. 1054–66.

[35] Kung, S.-Y. and Hu, Y.H., *A Highly Concurrent Algorithm and Pipelined Architecture for Solving Toeplitz Systems,* IEEE Trans. Acoustics, Speech, and Signal Processing, ASSP-21 (1983), pp. 66–76.

[36] Kung, H.T. and Leiserson, C.E., Systolic Arrays (for VLSI), *Sparse Matrix Proceedings,* SIAM, Philadelphia, PA, 1978, pp. 256–82.

[37] Kung, H.T. and Lin, W.T., An Algebra for VLSI Algorithm Design, *Proc. Conf. Elliptic Problem Solvers,* 1983.

[38] Kuo, C.J., *The Specification and Verification of Systolic Wave Algorithms,* LIDS-P-1368, MIT, 1984.

[39] Kuo, C.J., Levy, B.C. and Musicus, B.R., The Specification and Verification of Systolic Wave Algorithms, Cappello, P.R. et al. eds., *Proc. 1984 IEEE Workshop VLSI Signal Processing,* USC, Los Angeles, 1984, pp. 271–81.

[40] Lam, M.S., Mostow, J., A Transformational Model of VLSI Systolic Design, *Proc. IFIP Sixth Int. Symp. Comp. Hardware Descriptive Lang. Appl.,* CMU, Pittsburgh, PA, 1983.

[41] ————, *A Transformational Model of VLSI Systolic Design,* IEEE Computer, (1985), pp. 42–52.

[42] Leiserson, C.E., *Area-Efficient VLSI Computation,* The MIT Press, 1983.

[43] Leiserson, C.E. and Pinter, R.Y., Optimal Placement for River Routing, *CMU Conference on VLSI Systems and Computations,* Computer Science Press, 1981, pp. 126–42.

[44] Leiserson, C.E., Rose, F.M. and Saxe, J.B., Optimizing Synchronous Circuitry by Retiming, Bryant, R. eds., *Proc. Third Caltech Conf. VLSI,* Computer Science Press, Rockville, MD, 1983.

[45] Leiserson, C.E. and Saxe, J.B., Optimizing Synchronous Systems, *Proc. 22nd Ann. Symp. FOCS,* IEEE, 1981.

[46] ————, *Optimising Synchronous Systems,* J. VLSI and Computer Systems, 1 (1983), pp. 41–68.

[47] Levinson, N., *The Wiener RMS (Root-Mean-Square) Error Criterion in Filter Design and Prediction,* J. Math. Phys., 25 (1947), pp. 261–78.

[48] Levy, B.C., *2-D Polynomial and Rational Matrices, and their Applications for the Modeling of 2-D Dynamical Systems,* Ph.D. Thesis, Dept of Electrical Engineering, Stanford University, 1981.

[49] Lisper, B., *Description and Synthesis of Systolic Arrays,* Technical Report TRITA-NA-8318, Dept. of Numerical Analysis and Computing Science, The Royal Institute of Technology, Stockholm, Sweden, 1983.

[50] ————, *Hardware Synthesis from Output Specification with Ploynomials,* Technical Report, Dept. of Numerical Analysis and Computing Science, The Royal Institute of Technology, Stockholm, Sweden, 1984.

[51] Melhem, R.G., *Formal Verification of a Systolic System for Finite Element Stiffness Matrices,* Technical Report ICMA-83-56, Dept. of Mathematics and Statistics, University of Pittsburgh, 1983.

[52] ————, *Simulation of Systolic Networks with a Syntax Directed Solver for Systems of Sequence Equations,* Technical Report ICMA-83-58, Dept. of Mathematics and Statistics, University of Pittsburgh, 1983.

[53] Melhem, R.G. and Rheinboldt, W.C., *A Mathematical Model for the Verification of Systolic Networks,* SIAM J. Comput., 13 (1984), pp. 541–65.

[54] Miranker, W.L. and Winkler, A., *Space Time Representation of Computational Structures,* Computing, 32 (1984), pp. 93–114.

[55] Moldovan, D.I., *On the Design of Algorithms for VLSI Systolic Arrays,* IEEE Trans. Comp., C-31 (1982), pp. 1121–6.

[56] ————, On the design of Algorithms for VLSI Systolic Arrays, *Proc. IEEE,* 1983, pp. 113–20.

[57] ————, ADVIS : A Software Package for the Design of Systolic Arrays, *Proc. 1984 IEEE ICCD : VLSI in Computers,* 1984, pp. 158–64.

[58] Moldovan, D.I., Wu, C.I. and Fortes, J.A.B, Mapping an Arbitrarily Large QR Algorithm into a Fixed Size Array, *Proc. 1984 Conf. Parallel Processing,* 1984, pp. 365–73.

[59] Moore, G.E., *VLSI : Some Fundamental Challenges,* IEEE Spectrum, (1979), pp. 30–7.

[60] Morf, M. and Delosme, J.-M., Matrix Decompositions and Inversions via Elementary Signature-Orthogonal Transformations, *Proc. Int. Symposium on Mini and Micro Computers in Control and Measurement,* 1981.

[61] Muroga, S., *VLSI System Design,* John Wiley& Sons, New York, NY, 1982.

[62] Nash, J.G., Hansen, S. and Nudd, G.R., VLSI Processor Array for Matrix Manipulation, *CMU Conference on VLSI Systems and Computations,* Computer Science Press, 1981, pp. 367–73.

[63] Porat, B., Friedlander, B. and Morf, M., *Square-Root Normalized Covariance Algorithms,* IEEE Trans. Automat. Control, AC-27 (1982), pp. 813–29.

[64] Priester, R.W., Whitehouse, H.J., Bromley, K. and Clary, J.B., Signal Processing with Systolic Arrays, *Proc. Intern. Conf. Parallel Proc.,* IEEE, 1981, pp. 207–15.

[65] ————, Problem Adaption to Systolic Arrays, *Proc. SPIE (Real Time Signal Processing IV),* 1981.

[66] Quinton, P., *The Systematic Design of Systolic Arrays,* Research Report 193, IRISA, Rennes-Cedex, France, 1983.

[67] ————, Automatic Synthesis of Systolic Arrays from Uniform Recurrent Equations, *Proc. 11th Annual Intern. Symp. Computer Architecture,* IEEE, 1984, pp. 208–14.

[68] Ramakrishnan, I.V., Fussell, D.S. and Silberschatz, A., On Mapping Homogeneous Graphs on a Linear Array-Processor Model, *Proc. Int. Conf. Parallel Processing,* 1983, pp. 440–7.

[69] Speiser, J.M. and Whitehouse, H.J., Architectures for Real-Time Matrix Operations, *Proc. 1980 Government Microcircuits Applications Conference,* 1980.

[70] Tiden, E., *Verification of Systolic Arrays – A Case Study,* Technical Report TRITA-NA-8403, Dept. of Numerical Analysis and Computer Science, Royal Institute of Technology, 100 44 Stockholm, Sweden, 1984.

[71] Weiser, U. and Davis, A., A Wavefront Notational Tool for VLSI Array Design, *CMU Conference on VLSI Systems and Computations,* Computer Science Press, 1981, pp. 226–34.

[72] ————, *Mathematical Representation for VLSI Arrays,* Technical Report UUCS-80111, Dept. of Computer Science, University of Utah, 1981.

Appendix : Literature Survey

It has been recognized that the characterization of systolic arrays by graphical representations of data flow is not sufficient to guarantee their correctness or explain their workings, see [11] for a recent survey. Attempts at appropriate formalizations can be essentially divided into two categories, verification of the correctness of systolic designs and systematic synthesis of systolic designs (which are correct by construction).

Methods aimed at mere verification include

- [6, 8] Algorithms are expressed as recursive space-time programs and inductive techniques are applied for their verification.
- [36, 37, 38] Data appearing at a processor interconnection at successive time instances are represented as a data sequence. The computations performed by a processor are modeled as causal sequence operators, the latter being difference equations which relate the input data sequences to the output data sequences. The operation of the whole array is verified by solving the systems of difference equations associated with all processors and showing that they comply with the intended I/O description of the whole array. However, it is not always possible to solve the systems of difference equations analytically. The model is applied to the verification of a pipelined systolic system that performs the assembly of Finite Element stiffness matrices.
- [48] An induction proof is given for the correctness of an array that performs the QR decomposition of a banded matrix [14].
- [25, 26] Based upon the wavefront concept (see below) space-time-data equations are used to describe the movements of data elements. The operation of the hexagonal matrix multiplication array [23] is then shown to be correct by basically 'simulating' the movement of all data elements through the whole array. The approach is tedious if the geometry is not very regular or the data flow patterns are complicated.

In all of the above approaches the notation becomes awkward and unwieldy when applied to complex computations. The following methodologies aim at the synthesis of new systolic designs which, by construction, are correct and do not need a posteriori verification.

- [9, 15, 16, 50] The storage elements in a systolic system are described via the so-called z-operator, $z[x(i)] \equiv x(i-1)$, which models a time delay operator. On account of the correspondence between mathematical expressions and systolic systems (consisting only of logical and memory elements) the mathematical formulae can be manipulated to yield different systolic arrays.
- [17, 22, 49] The uniform progression in time or space of a set of data elements, which do not belong to the same data stream, can be viewed as a wavefront; the z-operator is used to represent the delay or rotation in the direction of wavefronts.
- [2, 3, 4, 5, 18, 19, 20, 39, 46] Nested-loop algorithms are mapped onto systolic arrays, whereby the computation is modeled as a 'lattice' whose nodes represent operations and whose edges represent the computational data dependences. Geometric linear transformations are first applied to the lattice. The transformed lattice is then mapped onto the space-time domain, where the coordinates of each node indicate at which time and in which processor the computation is to be performed.
- [10, 11, 12, 40, 41, 42, 43] Extending the approach of [19, 20], nested-loop algorithms are mapped onto systolic arrays. Linear transformations are applied to the matrix representing the data dependences. The integer solution to this matrix equation yields a mapping from algorithm to architecture. The software implementation of this method enumerates systolic designs but contains no systematic way of reducing the search space of all systolic arrays

associated with a given algorithm. The method is applied to the unshifted QR method for calculating the eigenvalues of a symmetric matrix; one way of performing problem partitioning is sketched.

- [1, 44, 45] The algorithm is expressed as a set of uniform recurrence equations whose indices form a convex domain in the cartesian coordinate system. At first a timing function is determined which specifies the execution time of a single computation and is compatible with the data dependences of the algorithm. Then, an allocation function is found which maps the algorithm onto a processor array. This appears to be one of the mathematically cleanest and most powerful approaches.

- [33, 34] A highly mathematical theory is developed which combines and enhances the above concepts of data dependences and geometric linear transformations; it is illustrated for the basic linear algebra algorithms. The theory is extended to the process of hardware synthesis where the outputs of the algorithm can be specified as a set of polynomials in some universal, many-sorted algebra (which does not constitute a restriction, since it is shown that the output of any algorithm performed by a Turing machine can be described by a set of polynomials). *Non-linear* transformations are applied to increase hardware utilization. Together with [44], this is the most mathematical and powerful approach.

- [29, 30, 31, 32] A general methodology is derived that eliminates broadcasting from a synchronous circuit without changing its communication structure. Although it neither describes nor synthesises systolic arrays, this method can be used to justify some of transformations that replace global broadcasting with local signal propagation.

- [21] The algorithm is represented as a signal flow graph which is then converted to a systolic array. After determining the set of operations to be executed by a single processor, the time of computation of each operation is derived by using the techniques of the approach above (considering cut-sets, inserting and scaling of delays).

- [24] Based on the example of finite and infinite impulse response filters, this approach uses an algebraic representation to combine the notion of z-operator and 'retiming' techniques similar to the ones in [29, 30, 31, 32], leading to an algebra for deriving designs whose z-graphs have systolic properties.

- [27, 28] A program takes as input a nested-loop algorithm and outputs a corresponding systolic design after having performed the following bottom-up process : starting with the innermost loops of the algorithm, the program allocates processors to particular operations (with help from the user), schedules their computation and performs a small set of transformations (selected by the user) to improve the design. The program relies heavily on user intervention and insight, the sequence of applied transformations does not result in *the* optimal design, but constitutes a 'bag of tricks' whose usefulness was determined by experience. It has been used to derive basic systolic arrays for polynomial evaluation, matrix multiplication, dynamic programming and greatest common divisor computation.

- [7] First, the algorithm is cast into a system of *first-order* recurrence equations, by adding and substituting variables. Then it is mapped onto space-time recurrence equations, starting with the initial conditions and proceeding inductively along a chosen time direction. On account of the induction process, algorithms are allowed as input that do not necessarily lead to designs with a repetitive structure or to data flows of uniform speed. However, already for simple algorithms the notation becomes lengthy and awkward, while the restriction to first-order recurrences may exclude important designs.

- [35] Under considerable restrictions, the design of matrix multiplication arrays is attempted, where systolic arrays are characterized through velocities of data flows, spatial distributions of data and periods of computation.
- [47] Using a divide-and-conquer approach the efficiency of those designs is increased in which each processor operates only every k-th time step; the approach is illustrated on convolution and matrix multiplication.
- [13] The definition of functional operators for the expression of vector algorithms makes it possible to establish a correspondence between data flow models of MIMD computation and design of systolic arrays. Conditions are specified under which a systolic version of a computational graph can be executed asymptotically as fast as the fully concurrent version of the original data flow graph.