Philips J. Res. **43**, *268-290, 1988* *R 1186*

SYSTOLIC ALGORITHMS FOR DIGITAL SIGNAL PROCESSING

by J.-P. CHARLIER, M. VANBEGIN and P. VAN DOOREN

Philips Research Laboratory Brussels, Av. Van Becelaere 2, B-1170 Brussels, Belgium

Abstract

In this paper we give an introduction to new developments in the interdisciplinary area of parallel algorithms for digital signal processing. Our purpose is not to be exhaustive but to introduce the reader to the current developments in the area, and to guide him through the existing literature.

Keywords: Digital signal processing, linear algebra, systolic algorithms.

1. Introduction

Parallel Algorithms (PA's) constitute presently a tentacular trend which extends its arms over a variety of mathematical fields. Parallelism seems to provide an adequate alternative means to speed up computation or data treatment, in consideration of the following facts:

(i) the rate of growth of computing power available from a single processor has been slowing down more and more for one or two decades;
(ii) at the same time, the hardware cost of executing an elementary operation has decreased dramatically;
(iii) also, problems have grown in difficulty and complexity.

Therefore, design of new algorithms, parallelization of existing ones, and study of their properties are in rapid development.

Digital Signal Processing (DSP) is an application area where early specific types of parallel implementation (namely systolic arrays) were proposed, partly because of the inherent resemblance between representations of digital filters and systolic arrays, but also because there is a strong demand for highly performant parallel algorithms in this area. The main reason for this is that several applications now imply real time implementations with very

high throughput rate of data and computational load. Typical examples of this are:

- speech processing where the sample rate can go up to 20.000/sec. and the number of operations may reach 1.500.000/sec. such as in speech recognition,
- adaptive filtering for devices incorporated in commercial products (such as a TV set or a compact disc) which have to operate in real time,
- medical scanners which process up to 350.000 samples/sec. and perform up to 50.000.000 operations/sec,
- radar processing which seems to have the most ambitious processing requirements with 10.000.000 samples/sec. and 5.000.000.000 operations/sec.

Especially for the last two examples, the desired throughput rate and computational load can hardly be handled by conventional machines anymore. Even supercomputers – with a moderate level of parallelism but with very high speed processors – can hardly tackle these tasks. Moreover they would not exploit the special features of these problems which are:

- repetitive nature of the computations both on the level of the processing (e.g. every 10 msec. a portion of a scanner signal is transformed to the frequency domain via an FFT) and on the level of the algorithmic details (an FFT is intrinsically repetitive in its fine details),
- use of elementary operations such as fixed point arithmetic, small word-length (6 to 16 bits for speech), cordic transformations, etc. is typical in this area,
- dedicated hardware allows to translate the special features of a specific application in (VLSI) hardware implementation.

In the next section we first give a brief outline of the specific parallel architectures we are considering here. In sec. 3 we then discuss the interaction of parallel algorithms with DSP, while in sec. 4 we emphasize the relevance of numerical linear algebra for future developments. Finally, we present some concluding remarks in the last section.

2. Distributed architectures and parallel processing

VLSI technology offers very fast and inexpensive computational elements which can be combined in highly parallel special-purpose structures for a wide range of potential applications. The concept of systolic architecture has been developed at Carnegie-Mellon University as a general methodology for mapping high-level computations into hardware structures. In this section we first describe features of systolic arrays and we justify them as our choice of

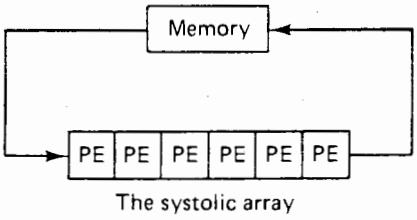


Fig. 1. Basic principle of a systolic system (taken from ref. 1).

architectural model in the sequel of this paper. Then we briefly describe how the performance of PA's can be measured.

In a systolic system, 'data flow from the computer memory in a rythmic fashion, passing through many processing elements before it returns to the memory, much as blood circulates to and from the heart' (see fig. 1)[2,3]). More precisely a systolic system is a computing network constituted by an array of processing elements or nodes (ideally implemented in a single chip), and possessing the following features[4]):

- Synchrony. The data are rhytmically computed (timed by a global clock) and passed through the array.
- Regularity, i.e. modularity and local interconnection. The array consists

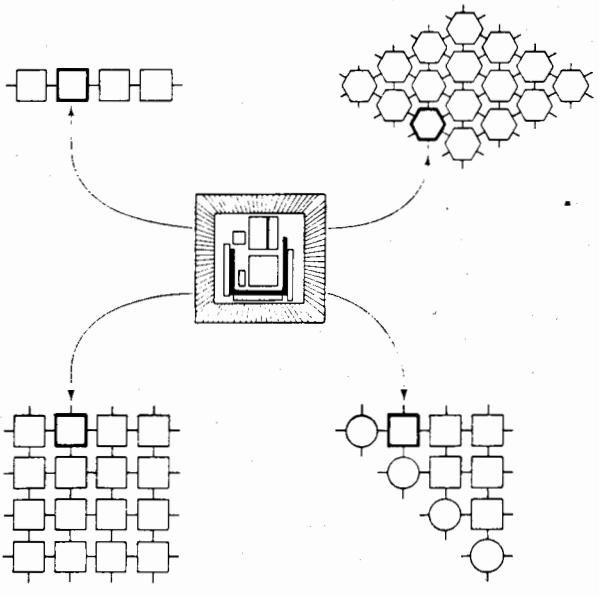


Fig. 2. Systolic array configurations using a same building block chip (taken from ref. 1).

of modular processing units with regular and local interconnections. Moreover the computing network may be extended indefinitely.
- Temporal locality. There will be at least one unit-time delay allotted so that signal transactions from one node to the next can be completed.
- Effective pipelinability. Ideally, successive computations are to be pipelined, such that all processing elements are working at any time, which leads to a linear 'speed-up' (see below).

Regularity involves simplicity of design and control, thus allowing to cope with the large hardware resources offered by VLSI. Systolic arrays are suitable for implementing a diversity of parallel algorithms and can assume many different configurations for different algorithms. Moreover it is often possible to use the same (programmable) processor or chip as building block for these arrays (see fig. 2).

The major limitation of systolic arrays, and of VLSI structures in general, is their restricted use to compute-bound algorithms. In a compute-bound algorithm, the number of computing operations is larger than the total number of input and output elements; otherwise the problem is I/O-bound. The I/O-bound problems are not appropriate for VLSI because VLSI packaging must be constrained with limited I/O pins. This limits the size of systolic arrays in practice. However, an alternative has been suggested to overcome that difficulty, by partitioning the initial problem into smaller parts and implementing them in a pipelined network [5]). Our choice of systolic arrays as an architectural reference is founded on the following facts:
- they are in close relation with actual and probably future technology, and receive a great deal of attention from universities and industry,
- by the inherently high and systematic degree of concurrency of their organization, and due to their formal definition, they constitute an abstract context in which existing algorithms can be efficiently parallelized and hopefully new algorithms could be designed,
- important applications of systolic arrays have been proposed in signal and image processing, matrix arithmetic, pattern recognition, ... ,
- procedures exist to express parallel algorithms in a form that can be easily implemented into systolic arrays (see for example refs 4, 6–8).

A slight variation of systolic arrays is obtained by removing the constraint of synchrony. The processing elements (PE's) then perform each their computations as soon as all the required data are available (the PE's are thus not synchronized by a global clock). As illustrated later this propagation resembles the propagation of waves through a medium, explaining the name of wavefront array often associated with these architectures [4]). In the rest of this paper, we put emphasis on algorithms for systolic-like arrays, by which

we mean algorithms that have already been implemented or are likely to be implementable on systolic arrays, wavefront arrays, or closely related architectures.

In studying PA's, we need some measure of their performance. Two related quantities are usually considered: speed-up measures the improvement in solution time using parallelism, while efficiency measures how well the processing power is being used. Issues of complexity, consistency and stability are also relevant.

The performance of a parallel algorithm strongly depends on the architectural features of the computer on which it is implemented. For instance, memory interference, interprocessor communication, and synchronization, lead to overhead and may affect seriously program execution times. In order to estimate precisely the importance of such factors, detailed computer models are necessary (see ref. 5 for an introduction). In this text, those aspects will not really be taken into acount in a quantitative way. Therefore performances will have to be interpreted as valid for an ideal implementation of corresponding algorithms.

Let us examine measures of performance on array processors or multiprocessor systems. Speed-up (S) is defined by the ratio of the execution time T_s on a serial computer or uniprocessor to the execution time T_p on the parallel computer:

$$S = \frac{T_s}{T_p}. \tag{1}$$

In practice, T_p incorporates any overhead of architectural nature. In the definition (1), T_s and T_p often refer to the same algorithm. For a computer that can support P simultaneous processes, the ideal value of S is then P. In fact, to achieve a fair comparison, T_s would be the execution time of the fastest algorithm for the same problem on one processor (as done e.g. in refs 9 and 10). S is then often much lower than P. Related to S, efficiency E is defined by

$$E = \frac{S}{P} \tag{2}$$

for a system of P processors. E indicates the utilization rate of the available resources (the processors) and is given by the ratio of the actual speed-up S to the ideal speed-up P. Note that the maximum value 1 of E is always reached for $P = 1$, so the number of processors is not necessarily chosen in order to maximize E. Rather, the goal is to construct algorithms exhibiting

linear speed-up in P and hence utilizing the processors efficiently. Speed-up of $kP/\log_2 P$ is also acceptable: the speed increase is almost two when the number of processors is doubled. More rapidly decreasing speed-up functions characterize algorithms that are poorly suited for parallelism.

As a rule, the evaluation of algorithms on serial computers is based upon computational complexity analysis, that is, arithmetic operation counts. In this context Lambiotte and Voigt [11]) have introduced the notion of consistency: a parallel algorithm is said to be (asymptotically) consistent if its arithmetic complexity is of the same order of magnitude as that of the best serial algorithm. Clearly linear speed-up can only be obtained for consistent algorithms. Yet, on array processors or multiprocessor systems, consistency is not always that crucial. If one step is counted for each set of operations performed simultaneously, the most important consideration is the number of steps (T_p in (1)), not the total number of operations performed by all the processors. Indeed, non-consistent algorithms are sometimes advantageous when the extra amount of computation does not lead to additional steps while communication needs between processors are reduced. E.g. with Csanky's method [12]), the inversion of an $n \times n$ matrix can be obtained in $O(\log^2 n)$ time-units (the time required for 1 flop) but using $O(n^4)$ PE's. This algorithm is thus not consistent with the standard $O(n^3)$ algorithms for sequential machines, but the computing time of $O(\log^2 n)$ is nevertheless very appealing.

The stability of parallel algorithms has not received much attention until now, but a growth of specific developments in this area will presumably take place in the near future. A recent study dealing with the evaluation of arithmetic expressions on a pipeline computer is given in ref. 13. Clearly, the behaviour of new algorithms is to be analysed. Also the use of a non-consistent algorithm may lead to a loss of accuracy, due to the greater number of operations that are required per result. Moreover, in the situation where the operations of a serial algorithm are purely rearranged in a new parallel order, stability properties may be affected. The fact that parallel processing of an algorithm may yield more accurate results than its serial version, is e.g. illustrated by the fan-in algorithm for sums and inner products [14]), and by recursive least-squares updating and downdating algorithms [15, 16]), but this property is not true in general.

3. Parallel algorithms in digital signal processing

In this section we first point out the similarities between 'flow graphs' often used in DSP and 'computational networks' used for systolic arrays. Then we

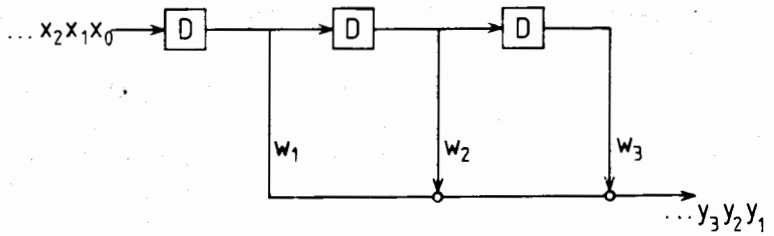


Fig. 3. A simple example: FIR convolution for $n = 3$; D represents a delay.

list various constraints that are relevant in the choice between different possible computational networks for a same DSP algorithm.

3.1. *A simple example: linear convolution*

In order to illustrate how PA's can efficiently be applied to DSP problems we take one of the simplest possible examples. Consider the linear convolution of a signal $\{x_i;\ i = 0,\ldots,\infty\}$ with a finite impulse response (FIR) filter $w(z) = w_1 \cdot z^{-1} + \ldots + w_n \cdot z^{-n}$, (where z^{-1} stands for the delay operator) yielding the output signal $\{y_i;\ i = ,\ldots,\infty\}$ defined by:

$$y_i = \sum_{j=1}^{n} w_j \cdot x_{i-j}. \tag{3}$$

A typical DSP representation of (3) is the 'computational flow graph' shown in fig. 3, where we have chosen $n = 3$ for illustrative simplicity. Several different systolic implementations of this formula are possible [17]) and we mention here only a few. In each of them one clearly recognizes the classical 'flow graph' representation of fig. 3.

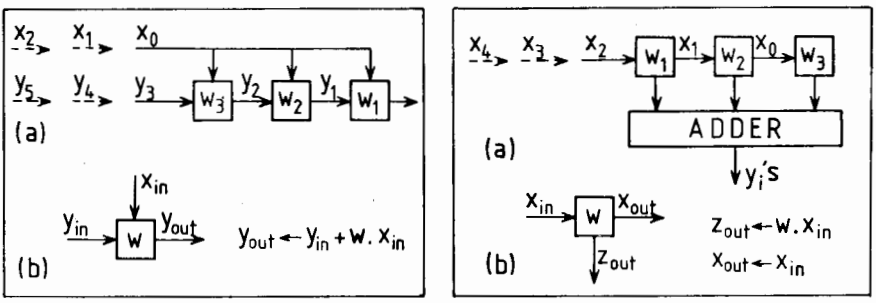


Fig. 4. Linear processor arrays for convolution (modified from ref. 3).

In a first design (fig. 4 left), the x_i's are 'broadcasted' simultaneously to all processors, each containing one value w_i, and the y_i's move systolically through the array of processors. In the second design (fig. 4 right), the x_i's move in systolically and the y_i's are obtained by an addition of all current values of the processors. Both these designs need a global connection (a bus for broadcasting or a global adder for reconstrucing y_i) which is a drawback for these two approaches.

Global connections could be avoided by letting the w_i's move systolically instead of the y_i's. Since then the y_i stay in the array, a systolic output path is needed to extract these values at the end of the convolution process, which is another drawback.

In fig. 5 we consider two designs that overcome these drawbacks. Here the w_i are each stored in one cell and both the x_i and y_i signals move systolically: in the first design in a direction opposite to each other and in the second design in the same direction but at different speeds. Since the y_i already move out systolically, no secondary path is needed to extract them.

This example is also representative of the fact that parallel algorithms in DSP are not 'really' (one should perhaps say here 'often') new algorithms but are rather clever implementations of existing DSP algorithms. In the next subsection we try to explain what exactly is meant by 'clever'.

3.2. *Criteria for systolization*

With criteria for systolization we mean those issues that have to be taken into consideration when trying to derive a 'systolic' implementation which is acceptable, good or even optimal for a hardware realization. We just list some main issues here and comment upon them.

- Minimal delay. Here one wants to minimize the amount of time elapsing between the input of (all) the data into the computational network and

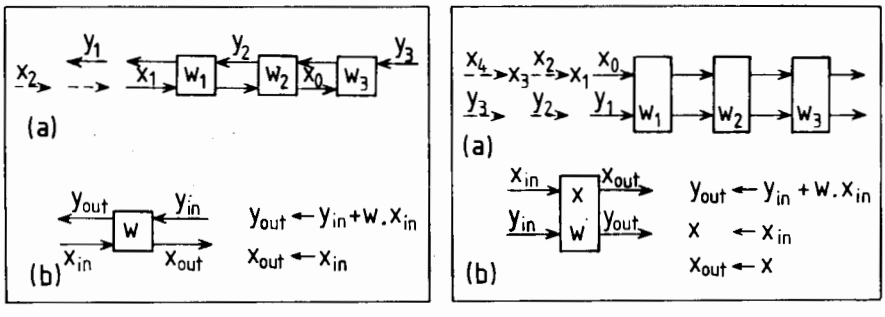


Fig. 5. Two 'optimal' designs for convolution (modified from ref. 3).

the output of (all) the results. The optimal speed is a function of the inherent degree of parallellism of the given algorithm – or the depth of the associated computational graph – since it is the longest computational path in that graph that determines the minimal delay between the input and the output of the scheme. The number of PE's to achieve this optimal speed (if it is known!) can be very large. In the above example the longest path is easily shown to require three binary operations which could be performed in one PE (each y_i is the result of three independent multiplications which can then be summed in two consecutive binary additions). Each PE then in fact computes one y_i and hence as many PE's are required as output data!

- Speed-up in steady state. Here one pursues roughly the same goal, but respecting the constraints of the input data flow. An example illustrates better this point. While the above 'minimal delay' solution requires each x_i to be entered at the same time and at n PE's concurrently, this usually does not coincide with the availability of the $\{x_i\}$. Indeed, these are often the samples of some real-time process and thus arrive typically in a piecemeal fashion. If in (1), the times correspond to the execution of an 'infinity' of identical computations either on one processor (T_s) or pipelined on a parallel computer (T_p), then S is the 'speed-up in steady state'. Since in this case one neglects the amount of time spent in loading the data and unloading the results, S is also given by the ratio of the throughput rates in sequential and parallel implementations. For the above problem a speed-up of 3 is then obtained in steady state by using the schedules of figs 4 and 5.
- Efficiency in steady state. This is the ratio of the speed-up in steady state and the number of processors. We note that optimal efficiency (in steady state) often implies to completely reorder or even rewrite the computation (see e.g. the FFT reformulation of a DFT), which makes it difficult to prove a certain scheme is optimal. Techniques for deriving such schemes are borrowed from complexity theory or sometimes use graphical considerations. Results in this area, though, are far from complete.
- Lay-out. Here one wants to find a systolization that makes the inplementation in hardware as simple as possible. Wire crossing (present in FFT, perfect shuffle, ...) or global connections (see second scheme of fig. 4) should typically be avoided. One usually prefers regular 1-dimensional or 2-dimensional arrangements of the PE's with only few connections between them (nearest neighbours only) in order to yield compact hardware implementations. Typical examples are the linear and square arrays of fig. 2.

– Robustness. Robustness means that the implemented algorithm should indeed yield the desired answers or at least answers that are 'reasonably close' to them. Therefore the design of the scheme should be such that overflows and underflows should not occur and that numerical errors should not propagate unboundedly (i.e. the scheme should be numerically stable in some sense). 'Simple' correction schemes for recovering from (or at least detecting) possible loss of accuracy are needed when numerical stability can not be ensured. This is still a trouble spot of some of the popular systolic algorithms.

We illustrate these different issues by two short examples. The first one is the discrete Fourier transform, defined by

$$X_i = \sum_{k=0}^{n} w^{ik} \cdot x_k = \sum_{k=0}^{n} \exp(-\frac{2ik\pi}{n+1} \cdot j) \cdot x_k, \qquad (4)$$

where $j = \sqrt{-1}$. The DFT of a signal $\{x_k; k = 0,\ldots,n\}$ is easily performed with a linear array of processors, as shown in the figure below:

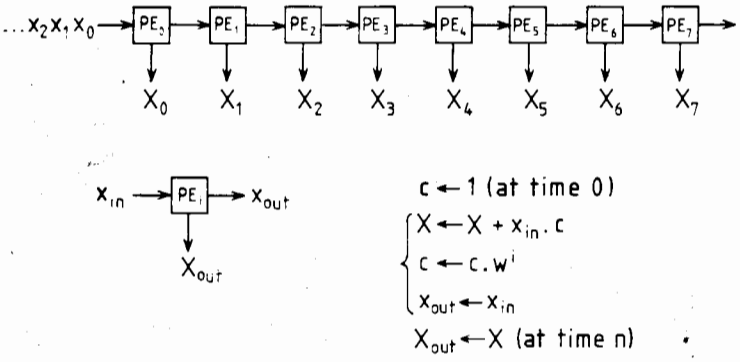


Fig. 6. A linear array for DFT with $n = 7$ (modified from ref. 6).

where in each processor a time-varying weight factor $c = w^{ik}$ is updated, with k (the time step) starting from 0 as x_0 enters it. The Fourier transform $\{X_i; i = 0,\ldots,n\}$ stays in the node and is eventually pumped out from each processor when the last x_n has passed it.

Just as for sequential machines the FFT formulation allows for a faster implementation of formula (4) on a parallel machine. This is shown in fig. 7, where c_i and d_i are appropriate powers of w stored in each processor and depending on the location of the processor in the graph. The drawback here, though, is that connections are not as regular anymore but require a global (Perfect Shuffle) communication network.

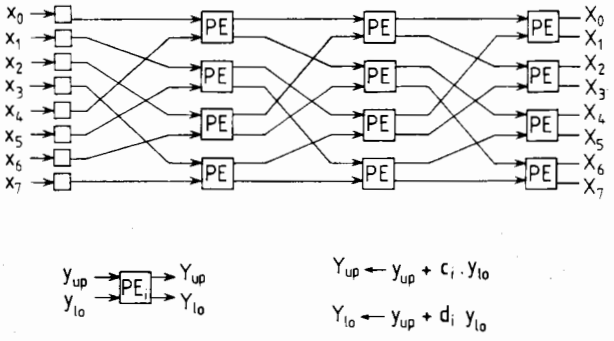


Fig. 7. A perfect shuffle array for FFT with $n = 7$ (taken from ref. 6).

The processing time is in favour of the FFT, with a ratio of $\log_2(n + 1)$ against $(n + 1)$ (assuming ideally that this is a power of 2). The data have indeed to pass $\log_2(n + 1)$ levels (3 here) of the above array before the signal $\{X_i; i = 0,\ldots,n\}$ is available at the output.

The speed-up of the DFT scheme in fig. 6 is $O(n + 1)$ using $(n + 1)$ PE's which is optimal and consistent with the formulation (4) of the Fourier transform. If the speed-up is measured against the best available sequential algorithm, then it is not optimal anymore for the systolic scheme in fig. 6 since the FFT formulation only requires $(n + 1) \cdot \log_2(n + 1)$ flops for the same computation. The scheme given in fig. 7, on the other hand, is consistent with the sequential FFT algorithm and yields (probably) minimal delay – namely $\log_2(n + 1) = 3$ transmissions and twice as many flops (in each PE) – corresponding to the longest computing path. Notice that for the execution of just one FFT this scheme is not efficient ($E = O(1/\log_2(n + 1))$), while its efficiency in steady state is optimal. The lay-out of fig. 7, though, is less appealing (interleaved connections) and the components $\{x_i\}$ of one data vector must be all available simultaneously, which is not the case in general.

If only a few FFT's have to be performed, a disadvantage of the scheme

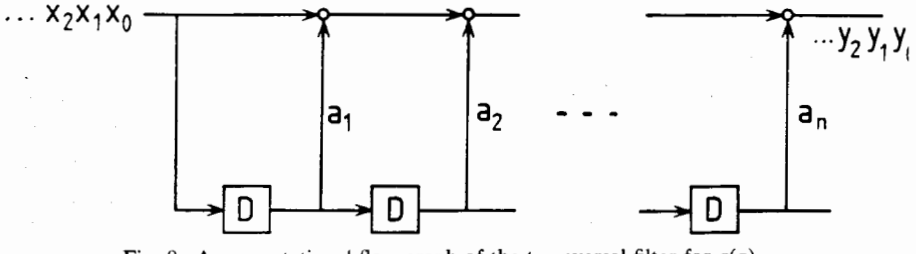


Fig. 8. A computational flow graph of the transversal filter for $a(z)$.

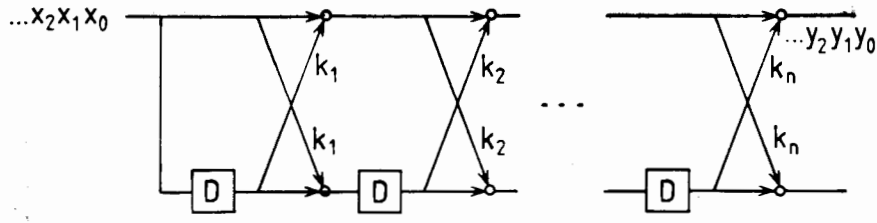


Fig. 9. A computational flow graph of the ladder filter for $a(z)$.

of fig. 7 is its high number of processors, namely $\log_2(n + 1)$ layers of $(n + 1)/2$ processors. This can easily be replaced by one layer of $(n + 1)/2$ processors through which the data circulate $\log_2(n + 1)$ times. This of course reduces the number of processors by a factor $\log_2(n + 1)$ but also reduces its throughput rate by the same factor (each new data vector has to wait $\log_2(n + 1)$ steps after the previous data vector before entering the array). The efficiency is now optimal both for one FFT and in steady state. On the other hand, the processors are slightly more complicated since they must keep track of what layer they are processing at each time step (the coefficients c_i and d_i vary with time, while before they were fixed for each PE).

The second example is the linear systolic array for implementing FIR digital filters. Consider a convolution with $a(z) = 1 + a_1 \cdot z^{-1} + \ldots + a_n \cdot z^{-n}$ applied to an input sequence $\{x_i;\ i = 0,\ldots,\infty\}$. A possible computational scheme for the output sequence $\{y_i; i = 0,\ldots,\infty\}$ is schematically represented by the signal flow graph of fig. 8, which is often called a transversal filter representation of $a(z)$.

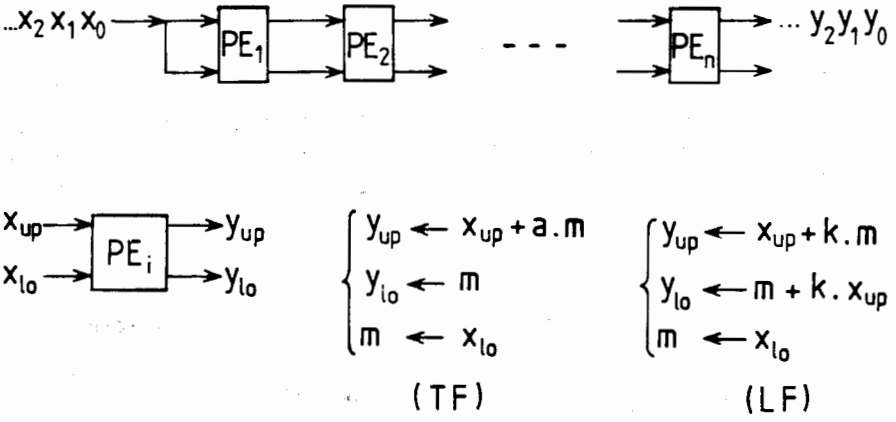


Fig. 10. Systolic implementation for transversal filter (TF) and ladder filter (LF).

If $a(z)$ is a stable polynomial in z^{-1}, then it can also be represented by its reflection coefficients $\{k_i; i = 1,\ldots,n\}$, linked with $a(z)$ via the following recursion:

$$\begin{aligned} &a_0(z) := 1, \\ &\text{for } i = 1 \text{ to } n, \quad a_i(z) := a_{i-1}(z) + k_i \cdot z^{-i} \cdot a_{i-1}(z^{-1}), \\ &a(z) := a_n(z) \end{aligned} \tag{5}$$

It turns out that these coefficients k_i are all bounded by 1 if and only if $a(z)$ is stable [18]). Moreover a computational graph for the output of the filter can be based on these coefficients as well. This is shown in fig. 9 (see also refs 1 and 18).

Both flow graphs represent the same filter $a(z)$ but using a different parametrization. Yet there are clear similarities between both representations and even more so when one looks at their parallel implementation on a linear array of n processors. This is shown in fig. 10, where the only difference between both schemes lies in the little programs stored in each processor.

Notice that the throughput rate of both filters is the same although seemingly twice as much work (multiplications) has to be performed in the ladder filter. The fact is that the assignments in each of the processors could be performed in parallel as well (within each processor) and the execution time in each processor is then really determined by its slowest operation (one multiplication and one addition). In practice, preference is given to the ladder filter for reasons of sensitivity (e.g. in speech synthesizers [18]). Remark that the TF is in fact a convolution (see the similarity with fig. 5 right) and thus that the convolution can be implemented as a LF if the corresponding polynomial is stable.

4. Linear algebra and digital signal processing

Some of the techniques that are typical to the interdisciplinary area of signal processing and parallel algorithms are also slowly getting more attention in numerical linear algebra. A classical example is the problem of updating and downdating Choleski and QR decompositions. It typically occurs e.g. in the recursive solution of least squares problems for filtering with sliding windows [19]). We do not aim here to give a complete derivation of these problems since they have already been treated extensively by several authors [16,20,21]).

In updating a QR decomposition one starts from an $n \times n$ system of equations

$$R \cdot x = b \tag{6}$$

with R upper triangular, resulting from a previous QR decomposition. Adding a new equation

$$a^T \cdot x = c \tag{7}$$

with a^T of dimension $1 \times n$, one then wants to determine the least squares solution of the updated system [22])

$$\begin{bmatrix} a^T \\ R \end{bmatrix} \cdot x = \begin{bmatrix} c \\ b \end{bmatrix}, \tag{8}$$

i.e. to find the QR factorization of the 'nearly triangular' matrix

$$\begin{bmatrix} a^T \\ R \end{bmatrix} = \begin{bmatrix} a_1 & a_2 & \dots & a_n \\ r_{11} & r_{12} & \dots & r_{1n} \\ & r_{22} & & \\ & & \ddots & \vdots \\ & & & r_{nn} \end{bmatrix}. \tag{9}$$

It is shown in [23]) that this only involves a sequence of n Givens rotations $G_{i,i+1}$ each operating on adjacent rows i and $i + 1$:

$$G_{i,i+1} = \begin{bmatrix} 1 & \dots & 0 & 0 & \dots & 0 \\ \vdots & & \vdots & \vdots & & \vdots \\ 0 & \dots & c_i & s_i & \dots & 0 \\ 0 & \dots & -s_i & c_i & \dots & 0 \\ \vdots & & \vdots & \vdots & & \vdots \\ 0 & \dots & 0 & 0 & \dots & 1 \end{bmatrix}, \quad c_i^2 + s_i^2 = 1 \tag{10}$$

$$G_{n,n+1} \cdot \dots \cdot G_{1,2} \cdot \begin{bmatrix} a^T \\ R \end{bmatrix} = \begin{bmatrix} R^+ \\ 0 \end{bmatrix}, \tag{11}$$

where R^+ is the updated triangular factor of the QR decomposition.

In refs 20 and 21, it is shown that these operations can nicely be implemented in parallel on a triangular array of processors, each initially containing just one element r_{ij} of the matrix R as drawn in fig. 11. Each round cell is a PE that generates an elementary orthogonal (Givens) rotation and each square cell is a PE that propagates these as row transformations. The trans-

formation $G_{1,2}$ between the first row of R and a^T is entirely performed in the first row of the triangular array of processors displayed in the figure. The diagonal processor constructs $G_{1,2}$ as soon as the element a_1 enters it, and it then propagates the rotation via the parameters $\{c_1, s_1\}$ to the processors on the right. The new row a^T is fed into the array in a skewed fashion in order that the parameters $\{c_1, s_1\}$ reach the PE's simultaneously with a_j. All data required for performing the elementary rotation

$$\begin{bmatrix} r_{1i}^{+} \\ a_i^{(1)} \end{bmatrix} = \begin{bmatrix} c_1 & s \\ -s_1 & c_1 \end{bmatrix} \cdot \begin{bmatrix} a_i \\ r_{1i} \end{bmatrix} \tag{12}$$

are therefore present in each processor at the right time. As a result of this a transformed (and shortened) row a^T is passed on to the second row of pro-

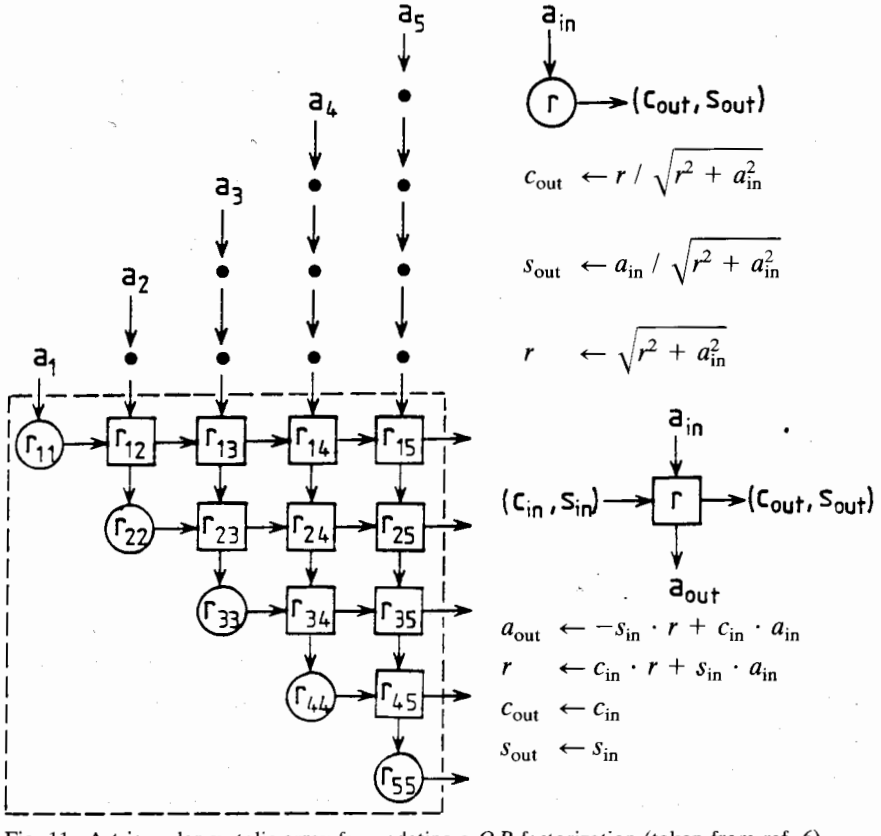


Fig. 11. A triangular systolic array for updating a QR-factorization (taken from ref. 6).

cessors, where similarly $G_{2,3}$ is constructed and applied. This process is repeated in the whole array so that the new factor R^+ is gradually constructed as the row a^T 'passes' through it (see refs 20 and 21 for details).

An important advantage of the above architecture is that it can be applied as well for downdating as for updating a triangular factor. With downdating one wants to reverse the above process and derive the 'old' triangular factor R from the new triangular factor R^+ and the row a^T. This corresponds to removing or deleting an equation from a least squares problem. In refs 16 and 23 it is shown how this involves the further reduction of the 'nearly triangular' matrix

$$\begin{bmatrix} a^T \\ R^+ \end{bmatrix} = \begin{bmatrix} a_1 & a_2 & \dots & a_n \\ r_{11}^+ & r_{12}^+ & \dots & r_{1n}^+ \\ & r_{22}^+ & & \\ & & \ddots & \vdots \\ & & & r_{nn}^+ \end{bmatrix} \tag{13}$$

to a triangular form using a sequence of n skew-Givens rotations $S_{i,i+1}$ operating on adjacent rows i and $i+1$:

$$S_{i,i+1} = \begin{bmatrix} 1 & \dots & 0 & 0 & \dots & 0 \\ \vdots & & \vdots & \vdots & & \vdots \\ 0 & \dots & \gamma_i & -\sigma_i & \dots & 0 \\ 0 & \dots & -\sigma_i & \gamma_i & \dots & 0 \\ \vdots & & \vdots & \vdots & & \vdots \\ 0 & \dots & 0 & 0 & \dots & 1 \end{bmatrix}, \quad \gamma_i = 1/c_i,\ \sigma_i = s_i/c_i,\ c_i^2 + s_i^2 = 1 \tag{14}$$

$$S_{n,n+1} \cdot \dots \cdot S_{1,2} \cdot \begin{bmatrix} a^T \\ R^+ \end{bmatrix} = \begin{bmatrix} R \\ 0 \end{bmatrix}. \tag{15}$$

Here R is the desired triangular matrix of the downdated QR decomposition. There is an apparent analogy between the updating equations (9 to 11) and the above downdating equations (13 to 15). Therefore these can clearly be implemented on the architecture of fig. 11 by a minor modification of the programs in the two types of PE's. Moreover, a careful implementation of the skew rotations $S_{i,i+1}$ yields comparable stability properties with the rotations $G_{i,i+1}$ used for the updating problem [16]).

A drawback of the parallel updating/downdating schemes, though, is their low efficiency. Indeed, as a^T passes through the triangular array – compare this to a wave travelling in a medium – only the processors on the wavefront

are active simultaneously. It is therefore rather inefficient to use $2n$ time steps on $n(n + 1)/2$ processors for something which requires only $2n^2$ operations on 1 processor ($E = O(1/n)$). But the nice feature of this architecture becomes apparent when performing several updates and/or downdates consecutively. These can indeed be pipelined in the triangular array one after the other as shown in fig. 12. Each row to be updated/downdated is fed into the array in a skewed fashion as indicated by the consecutive data wavefronts on the right, leading to a $O(1)$ steady-state efficiency. Applying an update or a downdate on a single architecture can e.g. be realized by adjoining a flag to each data row a^T indicating which type of rotation is to be performed on it. The (steady state) speed-up in now roughly $n(n+1)/2$ (i.e. the number of processors) since all processors are always active at each time step.

The parallel architecture of figs 11 and 12 thus combines nicely several desired properties: consistency, high performance, numerical stability, flexibility, appealing lay-out, etc. The algorithm is certainly one of the successful examples of the advantages of systolic implementations.

From these updating and downdating procedures one can also rederive other algorithms that are relevant to the area of signal processing. An ex-

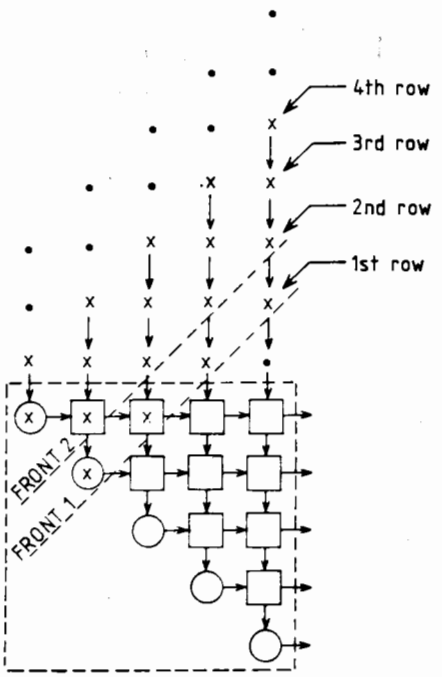


Fig. 12. Wavefronts for consecutive updatings and downdatings (taken from ref. 6).

ample is the Schur algorithm[24]) which is an $O(n^2)$ algorithm for constructing the (upper triangular) Choleski factor U of a symmetric positive definite $n \times n$ Toeplitz matrix T_n:

$$T_n = \begin{bmatrix} t_0 & t_1 & \cdots & t_{n-1} \\ t_1 & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & t_1 \\ t_{n-1} & \cdots & t_1 & t_0 \end{bmatrix} = U^T \cdot U. \tag{16}$$

Without loss of generality we assume $t_0 = 1$ in order to simplify the formulas in the sequel. It is easily seen that T_n can also be written as:

$$T_n = V_0^T \cdot V_0 - W_0^T \cdot W_0 \tag{17}$$

with

$$V_0 = \begin{bmatrix} 1 & t_1 & \cdots & t_{n-1} \\ & \ddots & \ddots & \vdots \\ & & \ddots & t_1 \\ & & & 1 \end{bmatrix}; W_0 = \begin{bmatrix} 0 & t_1 & \cdots & t_{n-1} \\ & \ddots & \ddots & \vdots \\ & & \ddots & t_1 \\ & & & 0 \end{bmatrix}. \tag{18}$$

Therefore the upper triangular factor U may be obtained by downdating the rows of W_0 from the upper triangular matrix V_0 with a skew unitary transformation Q_0:

$$\begin{bmatrix} U \\ 0 \end{bmatrix} = Q_0 \begin{bmatrix} V_0 \\ W_0 \end{bmatrix}; Q_0^T \cdot \begin{bmatrix} I_n & 0 \\ 0 & -I_n \end{bmatrix} \cdot Q_0 = \begin{bmatrix} I_n & 0 \\ 0 & -I_n \end{bmatrix}, \tag{19}$$

How this finally leads to the Schur algorithm is e.g. explained in ref. 25 and we follow their derivation here. First, since the diagonal elements of W_0 are zero, the first row of U is merely the first row of V_0. Moreover, the rest of the matrix U, say U_1, is determined from the following downdating problem:

$$\begin{bmatrix} U_1 \\ 0 \end{bmatrix} = Q_1^T \begin{bmatrix} V_1 \\ W_1 \end{bmatrix}; Q_1 \cdot \begin{bmatrix} I_{n-1} & 0 \\ 0 & -I_{n-1} \end{bmatrix} \cdot Q_1 = \begin{bmatrix} I_{n-1} & 0 \\ 0 & -I_{n-1} \end{bmatrix}, \tag{20}$$

involving the $(n-1) \times (n-1)$ triangular matrices:

$$V_1 = \begin{bmatrix} 1 & t_1 & \dots & t_{n-2} \\ & \ddots & \ddots & \vdots \\ & & \ddots & t_1 \\ & & & 1 \end{bmatrix}; W_1 = \begin{bmatrix} t_1 & t_2 & \dots & t_{n-1} \\ & \ddots & \ddots & \vdots \\ & & \ddots & t_2 \\ & & & t_1 \end{bmatrix}. \tag{21}$$

Downdating the first row of W_1 from V_1 then consists of first applying the 2 × 2 skew rotation (see earlier):

$$\begin{bmatrix} 1/c_1 & -s_1/c_1 \\ -s_1/c_1 & 1/c_1 \end{bmatrix} \cdot \begin{bmatrix} 1 & t_1 & \dots & t_{n-2} \\ t_1 & t_2 & \dots & t_{n-1} \end{bmatrix} = \begin{bmatrix} v_1 & v_2 & \dots & v_{n-1} \\ 0 & w_2 & \dots & w_{n-1} \end{bmatrix}, \tag{22}$$

where $s_1 = t_1$, $c_1 = \sqrt{1 - s_1^2}$ are chosen in order to annihilate the (2,1) entry in this matrix. Due to the Toeplitz structure of V_1 and W_1, the same 2 × 2 skew rotation applied to all the pairs of corresponding rows of V_1 and W_1, yields:

$$\begin{bmatrix} 1/c_1 I_{n-1} & -s_1/c_1 I_{n-1} \\ -s_1/c_1 I_{n-1} & 1/c_1 I_{n-1} \end{bmatrix} \cdot \begin{bmatrix} V_1 \\ W_1 \end{bmatrix} = \begin{bmatrix} v_1 & v_2 & \dots & v_{n-1} \\ & \ddots & \ddots & \vdots \\ & & \ddots & v_2 \\ & & & v_1 \\ 0 & w_2 & \dots & w_{n-1} \\ & \ddots & \ddots & \vdots \\ & & \ddots & w_2 \\ & & & 0 \end{bmatrix}, \tag{23}$$

so that all the diagonal elements of W_1 are annihilated. As before we obtain that $[v_1 \dots v_{n-1}]$ is the first row of U_1 (and thus the second row of U). One is then left with the downdating of the $(n-2) \times (n-2)$ matrices:

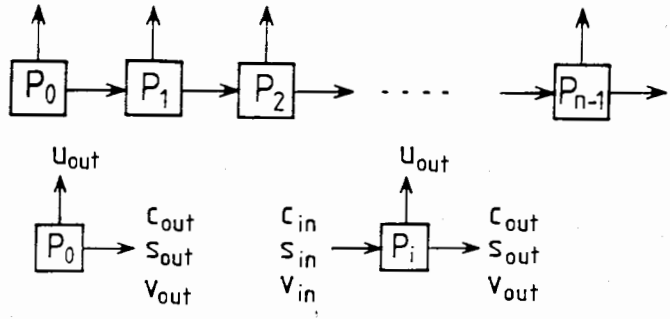
$$V_2 = \begin{bmatrix} v_1 & v_2 & \dots & v_{n-2} \\ & \ddots & \ddots & \vdots \\ & & \ddots & v_2 \\ & & & v_1 \end{bmatrix}; W_2 = \begin{bmatrix} w_2 & w_3 & \dots & w_{n-1} \\ & \ddots & \ddots & \vdots \\ & & \ddots & w_3 \\ & & & w_2 \end{bmatrix}. \tag{24}$$

The algorithm continues recursively (see ref. 25 for details) in the same manner as in the first two steps: at each step i, one additional row i of U is constructed and Toepliz matrices V_i and W_i of decreasing size are generated until they finally vanish. Clearly all the computations needed at each step i require only the processing of two rows (of decreasing length $n - i$).

A linear array of n processors can be used for a parallel implementation of the algorithm (see fig. 13). Indeed let each processor P_{j-1} contain initially the two elements of the j-th column of the matrix:

$$\begin{bmatrix} v_0^{(0)} & v_1^{(0)} & \dots & v_{n-1}^{(0)} \\ w_0^{(0)} & w_1^{(0)} & \dots & w_{n-1}^{(0)} \end{bmatrix} = \begin{bmatrix} 1 & t_1 & \dots & t_{n-1} \\ 0 & t_1 & \dots & t_{n-1} \end{bmatrix} \tag{25}$$

(where $v_i^{(0)}$ and $w_i^{(0)}$ are the elements of V_0 and W_0, respectively). Then the processing is as sketched below [16]):



$$\begin{cases} s_{out} \leftarrow w/v \\ c_{out} \leftarrow \sqrt{1 - s_{out}^2} \\ v_{out} \leftarrow v \cdot c \\ u_{out} \leftarrow v_{out} \\ w \leftarrow 0 \\ v \leftarrow 0 \end{cases} \qquad \begin{cases} s_{out} \leftarrow s_{in} \\ c_{out} \leftarrow c_{in} \\ v_{out} \leftarrow (v - s_{in} \cdot w)/c_{in} \\ u_{out} \leftarrow v_{out} \\ w \leftarrow c_{in} \cdot w - s_{in} \cdot v_{out} \\ v \leftarrow v_{in} \end{cases}$$

Fig. 13. A linear array for the Schur algorithm.

Notice that after step 0 the first processor becomes inactive (v and w are then 0) and that the role of the 'leading' processor P_0 is taken over by P_1, etc. (see refs 24 and 25 for more details). Hence, after n steps all the processors have become inactive and all the rows of U have been generated. The speed-up is easily seen to be $O(n)$ since $O(n^2)$ operations are performed in $O(n)$ time steps.

A second example of the use of updating and downdating is the QR-decomposition of an arbitrary rectangular Toeplitz matrix T_{mn} (with $m > n$):

$$T_{mn} \doteq \begin{bmatrix} t_0 & t_1 & \dots & t_{n-1} \\ t_{-1} & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & t_1 \\ t_{-m+n} & & \ddots & t_0 \\ & \ddots & & t_{-1} \\ \vdots & & & \vdots \\ t_{-m+1} & \dots & & t_{-m+n} \end{bmatrix} = Q \cdot R \tag{26}$$

In refs 26 and 27 it is shown that the factors Q and R can be generated in $O(mn) + O(n^2)$ operations using one updating and two downdating operations of easily constructed matrices. Moreover the method can efficiently be implemented on a linear array of processors in a very similar fashion to the Schur algorithm. Both algorithms are in fact closely related, as recently shown in ref. 28, and constitute key algorithms in several problems of signal processing. Their new derivation from updating and downdating techniques also suggests that the stability results known for updating and downdating [16,22]) could be applied to these signal processing decompositions as well. This could eventually yield new and/or improved bounds for the error propagation in these algorithms.

We are convinced that it is exactly this kind of interplay between parallel algorithms in linear algebra and signal processing that can lead to new and fruitful results in the future.

5. Concluding remarks

Digital signal processing is one of the application areas where the first systolic implementations were proposed, partly because the inherent resemblance between representations of digital circuits and systolic arrays (see e.g. ref. 4). Topics covered here are, among others, convolution [17]), FFT [29]), lattice or ladder filters [24]), etc. For more specialized fields such as speech processing systolic designs are also found at other levels, especially when a large amount of data has to be processed in a regular fashion. Examples are codebook processing in vector quantization, dynamic programming and time warping in word recognition (refs. 30 to 33, ...). Also in image processing there are typical techniques of a regular nature [34,35]): all pixels of an image are often treated in a similar 'local' fashion, which only involves a few of the neighbouring pixels. These techniques are therefore well suited for parallel implementation on systolic arrays. Examples are 2D convolution [17]), statistical analysis of images [36]), or real time processing [37,38])). From all this it is clear that Parallel Algorithms and DSP have a serious impact on each other.

Moreover some new problems have emerged from the interference of PA's and DSP. One is e.g. working on proofs of correctness of parallel algorithms [39]), complexity theory of parallel algorithms, languages for systolic arrays, simulators of systolic arrays, automatic parallelization of algorithms, automatic systolization of parallel algorithms, hardware implementation of systolic arrays, and so on. In some of these areas significant advances have been made. In ref. 4 the systematic derivation of systolic arrays from DSP-like signal flow graphs is proposed and some existence theorems in this context are derived. More systematic treatments are given in refs 7 and 40.

Geometric methods for deriving various systolic schemes from each other, have been proposed. Theories in this context are slowly emerging and are related to complexity theory and computational algebra[41]). Automatic translators are being considered as well.

Due to the inherent constraints of systolic-like arrays, a number of completely new methods have also appeared in DSP (although they are not completely novel, they can certainly be called new to the DSP community). One of the most typical examples is the Number Theoretic Transform (NTT) which is rather popular in discrete mathematics. It seems to be a valuable candidate for implementing in parallel a number of classical DSP problems[42]). The derivation of new algorithms is also coming from areas such as differential equations or linear algebra[43]). The reason of this cross-fertilization is that the consideration of PA's in these fields helps to create common interests.

Although PA's in DSP can hardly be called a new 'discipline', one cannot deny that it has induces novel techniques and ideas.

REFERENCES

1) S.Y. Kung, H.J. Whitehouse and T. Kailath, VLSI and modern signal processing, Prentice Hall, 1985.
2) H.T. Kung and C.E. Leiserson, Introduction to VLSI systems (C. Mead and L. Conway eds.), Addison-Wesley, 1980, p. 271.
3) H.T. Kung, Computer **15**, 37 (1982).
4) S.Y. Kung, IEEE Proceedings, **72**, 867 (1984).
5) K. Hwang and F.A. Briggs, Computer architecture and parallel processing, McGraw-Hill, 1984.
6) S.Y. Kung, IEEE ASSP Magazine, **2**, 4 (1985).
7) D.I. Moldovan, IEEE Proceedings, **71**, 113 (1983).
8) V. van Dongen and P. Quinton, Int. Report M 235, Philips Res. Lab. Brussels.
9) H.S. Stone, Complexity of sequential and parallel numerical algorithms (J.F. Traub ed.), Academic Press, New York, 1973, p. 1.
10) D. Heller, SIAM Review, **20**, 740 (1978).
11) J.L. Lambiotte and R.G. Voigt, ACM Trans. math. Software, **1**, 308 (1975).
12) L. Csanky, SIAM J. Comput. **5**, 618 (1976).
13) W. Rönsch, Parallel Computing, **1**, 75, (1984).
14) T.L. Jordan, Parallel computations (G. Rodrigue ed.), Academic Press, New York, 1982, p. 1.
15) M. Gentleman, JIMA, **12**, 329 (1973).
16) A. Bojanczyk, R. Brent, P. Van Dooren and F. de Hoog, SIAM Scisc, **8**, 210 (1987).
17) H.T. Kung, Int. Report Dept. Comp. Sc., Canergie-Mellon, 1982.
18) J.D. Markel and A.H. Gray Jr., Linear Prediction of Speech, Springer Verlag, New York 1976.
19) M.L. Honig and D.G. Messerschmitt, Adaptive Filters, Kluwer Academic, Hingham, 1984.
20) W. Gentleman and H. Kung, Proc. SPIE Symp. 1981, **298**, Real Time Signal Processing IV, 1981, p. 19.
21) J. Mc Whirter, Proc. SPIE Symp. 1981, **298**, Real Time Signal Processing IV, 1981, p. 105.

[22]) G.H. Golub and C.F. Van Loan, Matrix computations, North Oxford Academic, Oxford, 1983.
[23]) G.H. Golub, Statistical Computation (R.C. Milton and J.A. Nelder, eds.), Academic press, New York, 1969, p. 365.
[24]) T. Kailath, VLSI and modern signal processing (S.Y. Kung, H.J. Whitehouse and T. Kailath, eds.), Prentice Hall, 1985, p. 5.
[25]) J.-M. Delosme and I. Ipsen, Linear Algebra & Applications, **77**, 75, 1986.
[26]) A. Bojanczyk, R. Brent and F. de Hoog, Int. Report CMA-R06-85, Centre for Mathemathical Analysis, Australian National University, 1985.
[27]) A. Bojanczyk, R. Brent and F. de Hoog, Numerische Mathematik, **49**, 81 (1986).
[28]) J. Chun, T. Kailath and H. Lev-Ari, SIAM Scisc., **8** 899 (1987).
[29]) S.Y. Kung, K.S. Arun, R.J. Gal-Ezer and D.V. Bhaskar Rao, IEEE Trans. Comp., Vol. **CS-31**, 1054 (1982).
[30]) J.-P. Banatre, P. Frison and P. Quinton, Int. Report No. 169, IRISA, Rennes, 1982 (also in Proc. ICASSP 82).
[31]) N. Weste, D. Burr and B. Ackland, IEEE Trans. Vomp., **C-32**, 731 (1983).
[32]) P. Frison and P. Quinton, Int. Report, IRISA, Rennes, 1984.
[33]) Y. Robert and M. Tchuente, RAIRO Th. Inf. **19**, 107 (1985).
[34]) Special Issue, IEEE Computer, Jan. 1983.
[35]) T. Bonnet, P. Martin, Y. Mathieu and O. Duthuit, Int. Report C-86-390, Laboratoire d'Electronique et de Physique Appliquée, Paris, France, 1986.
[36]) A.L. Fisher, Int. Report CMU-CS-81-130, Carnegie Mellon University, 1981.
[37]) M. Duff, Computing structures for image processing, Academic Press, new York, 1983.
[38]) G. Gaillat, Traitement du Signal, **1**, 19 (1984).
[39]) M. Ossefort, ACM Trans. Progr. Lang and Syst. 1983 p. 620.
[40]) H.V. Jagadish, S.K. Rao and T. Kailath, IEEE Proceedings, **75**, 1304 (1987).
[41]) R.M. Karp, R.E. Miller and S. Winograd, Journal ACM, **14**, 563, (1967).
[42]) A. Dennis and C. Marshall, Int. Report TN2472, Philips Research Laboratory Redhill, England, 1987.
[43]) J.-P. Charlier, M. Vanbegin and P. Van Dooren, Int. Report R502, Philips Res. Lab. Brussels, Belgium, 1986.

Authors

Jean-Paul Charlier; B.S. degree (Electronics), Catholic University of Louvain-la-Neuve, Belgium, 1976; M.S. degree (Nuclear science and radioprotection), Catholic University of Louvain-la-Neuve, Belgium, 1978; M.S. degree (Applied natural sciences), Catholic University of Louvain-la-Neuve, Belgium, 1986; Philips Research Laboratory, Brussels, 1979– . His main current interests lie in parallel algorithms for linear algebra problems, with possible digital signal processing and linear system applications.

Michel Vanbegin, Higher Technical School (Electronics), Brussels, Belgium, 1965; Philips Research Laboratory, Brussels, 1969– . His work has been mainly concerned with compiler writing (ALGOL 68), automatic processing of logical lay-out, speech recognition and systems and control. His current interests lie in the area of digital signal processing and parallel algorithms.

Paul Van Dooren; Ir. degree, Catholic University of Leuven, Belgium, 1974; Doctoral degree, Catholic University of Leuven, Belgium, 1979; Assistant in the Department of Applied Mathematics and Computer Science of the Catholic University of Leuven 1974–1977; Research Associate at the University of Southern California 1978–1979; Postdoctoral Fellow at Stanford University, 1979–1980, and visiting Fellow at the Australian National University in 1985; Philips Research Laboratory, Brussels, 1980– . His main interests lie in the areas of numerical linear algebra, linear system theory, digital signal processing and parallel algorithms. He is an Associate Editor of Systems and Control Letters, the Journal of Computational and Applied Mathematics, Numerische Mathematik and SIAM Journal on Matrix Analysis and Applications.