Computing 34, 191 – 219 (1985)

Computing

© by Springer-Verlag 1985

A Systolic Array Algorithm for the Algebraic Path Problem (Shortest Paths; Matrix Inversion)*

Günter Rote, Graz

Received April 17, 1984

Abstract — Zusammenfassung

A Systolic Array Algorithm for the Algebraic Path Problem (Shortest Paths; Matrix Inversion). It is shown how the Gauß-Jordan Elimination algorithm for the Algebraic Path Problem can be implemented on a hexagonal systolic array of a quadratic number of simple processors in linear time. Special instances of this general algorithm include parallelizations of the Warshall-Floyd Algorithm, which computes the shortest distances in a graph or the transitive closure of a relation, and of the Gauß-Jordan Elimination algorithm for computing the inverse of a real matrix.

AMS subject classifications: 68A05, (05C35, 05C38, 16A78, 65F05, 68E10).

CR categories and subject descriptors: C.1.2 [processor architectures]: multiple data stream architectures (multiprocessors) – systolic arrays; G.1.0 [numerical analysis]: general – parallel algorithms; G.1.3 [numerical analysis]: numerical linear algebra – matrix inversion; G.2.2 [discrete mathematics]: graph theory – path problems; B.6.1 [logic design]: design styles – cellular arrays; B.7.1 [integrated circuits]: types and design styles – algorithms implemented in hardware; VLSI (very large scale integration).

General terms: algorithms, design, performance.

Additional key words and phrases: Algebraic path problem, shortest paths, transitive closure, closed semirings, Gauß-Jordan elimination.

Ein systolic-array-Algorithmus für das algebraische Wegproblem (kürzeste Wege; Matrizeninversion). Es wird dargestellt, wie man den Gauß-Jordanschen Eliminationsalgorithmus für das algebraische Wegproblem auf einem hexagonalen systolischen Feld (systolic array) mit einer quadratischen Anzahl einfacher Prozessoren in linearer Zeit ausführen kann. Zu den Anwendungsbeispielen dieses allgemeinen Algorithmus gehört der Warshall-Floyd-Algorithmus zur Berechnung der kürzesten Wege in einem Graphen oder zur Bestimmung der transitiven Hülle einer Relation sowie der Gauß-Jordansche Eliminationsalgorithmus zur Inversion reeller Matrizen.

1. Introduction

The Algebraic Path Problem unifies three streams of evolution each of which independently developed its own algorithms: the determination of the transitive closure of a relation and the determination of shortest paths in networks; Kleene's construction of the regular expression representing the language accepted by a finite automaton [1956] as the starting point of formal language and automata theory with the subsequent development of regular algebra; and numerical linear algebra.

---

* This work was initiated as a project in a lecture on Languages and VLSI design which Professor Karel Čulik II gave during his stay as a visiting professor at the Institutes for Information Processing, Technical University of Graz, Austria, in the summer semester 1983, and was later supported by the Computer Center Graz (Rechenzentrum Graz).

The similarity between direct and iterative methods for solving systems of linear equations and for inverting matrices of real or complex numbers, which were known for a long time, and the corresponding algorithms for graphs, which were relatively new, was first noted by Carré [1971]. He introduced semirings to show that the solution to these graph problems can be formulated in a unified manner, either as sum of "measures" of paths or as a solution of a system of linear equations. He also called the general graph algorithms after their prototypes in numerical mathematics; hence the name "Gauß-Jordan elimination" for the algorithm which the algorithm in this paper is a version of. In Aho, Hopcroft and Ullmann [1975] and in Backhouse and Carré [1975], the link from the Algebraic Path Problem to regular algebra and language theory was established. However, these authors considered only idempotent semirings. They could therefore not provide a common algebraic framework for the graph algorithms *and* the numerical algorithms. This was only achieved by Lehmann [1977].

Numerous applications of the Algebraic Path Problem and the algorithms for its solutions are known in different areas; many of them were of course conceived independently and without reference to each other. For an overview of applications see e.g. Gondran and Minoux [1979, section 3.3], Carré [1979, chapter 3, section 3.2.2 and chapter 4], Zimmermann [1981, chapter 8], Brucker [1974, chapter 4]. Several applications in global flow analysis of computer programs, which is useful for code optimization, are discussed by Tarjan [1981].

Mahr [1982] provides a comprehensive overview of basic results concerning semirings, as they are dealt with in the algebraic part of this paper.

Designing algorithms for many cooperating parallel processors has been made desirable by the relative stagnation in the increase of processor speed (of single processors); and the development of VLSI technology with the enormous reduction of production costs has made it feasible to build special-purpose multi-processor chips implementing these algorithms. In such algorithms special attention has to be paid to the organization of parallelism, and difficulties arise which surpass those that arise in connection with ordinary sequential algorithms. As one general scheme for the organization of parallelism, systolic arrays (for VLSI) were introduced by Kung and Leiserson [1978]. Their main characteristic is that the layout of processors and connections is simple and regular. Every processor regularly "pumps" (hence the name "systolic") data in and out, each time performing some short computation, so that a regular flow of data is kept up in the network of processors. A discussion of previous systolic arrays which are similar to the one which is the subject of this paper is deferred until the latter has been described (see section 8).

To summarize the *main result* of the paper: Any of the problems mentioned at the beginning can be solved on an array of $(n+1)^2$ processors in $7n-2$ steps. (One step typically requires one addition and one multiplication in the underlying algebra.) The operations can be pipelined in such a way that a new instance of the problem can be solved every n time steps.

Outline:

The material in this paper has two quite different aspects: the algebraic issues of the formulation of the Algebraic Path Problem and its solution, which are dealt with in the following three sections; and the computer-oriented questions of the systolic array algorithm, which shall occupy the remaining sections. The treatment of the latter subject is intended to be comprehensive, whereas I discuss the algebraic framework only as far as it is needed in the later sections, in order to make the paper self-contained in this area.

Section 2 contains elementary definitions from graph theory and algebra which are necessary to introduce the Algebraic Path Problem. Along with these definitions I give three examples for the algebraic structure (the semiring) underlying the Algebraic Path Problem, which yield its most important instances: the inverse of a real matrix; the shortest distances in a weighted graph; and the transitive and reflexive closure of a binary relation. The semiring of the square matrices of fixed size over a semiring is introduced.

In section 3 a sequential algorithm called sGJE for computing the solution of the general Algebraic Path Problem is given. This algorithm, which is a version of Gauß-Jordan elimination, can be adapted to solve any Algebraic Path Problem.

The application of the algorithm to the three examples of semirings given in section 2 is discussed in section 4.

Section 5 presents the systolic arrary GJE0 used to solve the Algebraic Path Problem: the pattern of the processors and their connections, the operations that the processors perform, and the data flow.

Section 6 analyzes the performance of the basic algorithm presented so far. The possibility of pipelining and extensions and variations of this basic design allowing to *improve* processor utilization are discussed. Questions concerning the applicability are also addressed, namely the modular extensibility of the design and the *decomposability* of the Algebraic Path Problem.

Section 7 contains some details about the implementation of the organization of the data flow in the various schemes presented, which have been bypassed in the preceding sections: the control flow (as opposed to the data flow).

In section 8 the relation to the existing literature on systolic arrays is discussed.

Proofs, additional details, and more variations are contained in [Rote 1984].

2. Definitions and Examples

A *graph* $G=(V,E)$ consists of a finite *vertex set* V and an *arc set* $E\subseteq V\times V$. Thus, I am considering finite directed graphs with no multiple arcs but possibly with loops. In the following I shall always assume that $V=\{1,2,\dots,n\}$.

A ***path*** p in a graph is an alternating sequence of vertices and arcs of the form

$$p=(v_0,a_1,v_1,a_2,v_2,\dots,v_{l-1},a_l,v_l),$$

where $l \geqq 0$, the v_i are vertices of the graph, and the a_i are arcs satisfying $a_i = (v_{i-1}, v_i)$. p is called a path *from* vertex v_0 *to* vertex v_l.

A *weighted graph* (V, E, w) consists of a graph (V, E) together with a *weight function* $w: E \to H$, where H is an arbitrary set. $w(a)$ is called the *weight* of the arc a. If $(H, \otimes)$ is a monoid, we can extend the function w to the set of all paths:

$$w((v_0, a_1, v_1, a_2, v_2, \ldots, v_{l-1}, a_l, v_l)) := w(a_1) \otimes w(a_2) \otimes \ldots \otimes w(a_l).$$

The weight of an empty path (a path containing no arcs) is of course ①, the neutral element of the monoid. The weight of a path can thus be computed from the sequence of its arcs alone.

A *semiring* $(H, \oplus, \otimes)$ *with zero* ⓪ *and unity* ① is an algebraic structure with two binary operations, fulfilling the following four axioms:

(A_1) $(H, \oplus)$ is a commutative semigroup with neutral element ⓪.

(A_2) $(H, \otimes)$ is a semigroup with neutral element ①.

(A_3) $\otimes$ is distributive over $\oplus$:

$$a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c), \text{ and}$$
$$(a \oplus b) \otimes c = (a \otimes c) \oplus (b \otimes c).$$

(A_4) (The zero rule:) Zero is absorptive with respect to $\otimes$:

$$⓪ \otimes a = a \otimes ⓪ = ⓪.$$

Example 1:

$H_1 = (\mathbb{R}, +, .)$ the set of real numbers with ordinary addition and multiplication, is a semiring with zero 0 and unity 1.

Example 2:

$H_2 = (\bar{\mathbb{R}}, \min, +)$, where $\bar{\mathbb{R}} = \mathbb{R} \cup \{\infty, -\infty\}$, the real numbers extended by plus and minus infinity, is a semiring with zero ∞ and unity 0, if the convention $\infty + a = \infty$, $(-\infty) + a = -\infty$, for all real a, and $(-\infty) + \infty = \infty$, is used.

Example 3:

$H_B = (\{0, 1\}, \vee, \wedge)$, where $\vee = \max$ and $\wedge = \min$, is semiring with zero 0 and unity 1, the so-called Boolean semiring. (Every bounded distributive lattice is a semiring.)

Remark 2.1, matrix semirings:

The $n \times n$-matrices $H^{n \times n}$ $(n \geqq 1)$ over a semiring $(H, \oplus, \otimes)$ with zero ⓪ and unity ①, with matrix addition $\oplus$ and matrix multiplication $\otimes$ both defined just as in conventional linear algebra, form themselves a semiring with the matrix containing only ⓪ entries as zero and I, the matrix containing ① in the main diagonal and ⓪ otherwise, as unity.

The definition of the weight of a path in terms of the arc weights involves one algebraic operation. This link between the graph theoretical concepts and the algebraic definitions presented in this section is now going to be exploited and extended to define the *Algebraic Path Problem* (Zimmermann [1981]):

Given a weighted graph $G=(V,E,w)$, $w: E \to H$, with weights from a semiring $(H, \oplus, \otimes)$ with zero ⓪ and unity ①, find for all pairs of vertices (i,j) d_{ij}, where

$$d_{ij} := \bigoplus_{\substack{p \text{ is a path} \\ \text{from } i \text{ to } j}} w(p). \tag{1}$$

($\bigoplus_{i \in I} a_i$ is a notation for the sum of the members of an indexed family.)

A solution to this problem need not exist, because the set of weights of the paths from i to j, over which the sum in (1) is taken, may be infinite. However, there may be semirings where such countably infinite sums may be defined in a consistent way, at least for some cases. "Consistency" means that the following axioms must hold in addition to $(A_1 - A_4)$:

(A_5) (distributive law):
Let I and J be two countable (finite or infinite) sets; then

$$\bigoplus_{(i,j) \in I \times J} a_i \otimes b_j = \bigoplus_{i \in I} a_i \otimes \bigoplus_{j \in J} b_j$$

must hold, whenever both sums on the right side are defined.

(A_6) (associative law):
Let I, K, and J_k be countable (finite or infinite) sets such that $\{J_k \mid k \in K\}$ is a partition of I; then

$$\bigoplus_{i \in I} a_i = \bigoplus_{k \in K} \bigoplus_{i \in J_k} a_i$$

must hold, whenever the left side of the equation is defined.

Such semirings are called *partially complete semirings*, or, if the sum is defined for all countable families, (countably) *complete semirings*.

These axioms are sufficient for the applications in this paper and at the same time restricted enough to accommodate the important case of the real numbers (example 1 below). For a more extensive discussion of summability in semirings see Mahr [1984] and Mahr [1982, chapter 6].

In our computational procedure below we have to compute infinite sums of the following kind, for which a special notation is introduced:

$$c^* := \bigoplus_{i \geq 0} c^i = ① \oplus c \oplus (c \otimes c) \oplus (c \otimes c \otimes c) \oplus \ldots \tag{2}$$

Example 1:
To turn H_1 into a partially complete semiring the value of a countably infinite sum of elements from H_1 may be defined to be the sum of any corresponding infinite series, if the latter is absolutely convergent. In particular:

$$c^* = (1-c)^{-1}, \text{ if } |c| < 1; \text{ otherwise undefined.}$$

Example 2:
H_2 is a complete semiring, where $\bigoplus a_i$ exists always and is simply the infimum of the set $\{a_i\}$ (in the extended set of real numbers).

The star operation in H_2 is:

$$\text{if } c \geqq 0 \text{ then } c^* = 0 \text{ else } c^* = -\infty .$$

Example 3:
In H_B there are no proper infinite sums since addition in H_B is idempotent and H_B contains only two elements. Thus, H_B is a complete semiring.

$$0^* = 1^* = 1 .$$

The Algebraic Path Problem (1) can also be formulated in a different way:

With the weighted graph (V, E, w) one can associate an $n \times n$ weight matrix

$$A = (a_{ij}), \text{ where } a_{ij} = \begin{cases} w((i,j)), & \text{if } (i,j) \in E, \\ ⓪, & \text{if } (i,j) \notin E. \end{cases}$$

Let us now look at successive powers of the square matrix A:

$$A^2 = (a_{ij(2)}), \text{ where } a_{ij(2)} = \bigoplus_{1 \leqq k \leqq n} a_{ik} \otimes a_{kj}$$

$$A^3 = (a_{ij(3)}), \text{ where } a_{ij(3)} = \bigoplus_{1 \leqq k_1, k_2 \leqq n} a_{ik_1} \otimes a_{k_1 k_2} \otimes a_{k_2 j} \text{ etc.}$$

Thus, if A is the weight matrix of some weighted graph (V, E, w), then we have for all $m \geqq 0$:

$$A^m = (a_{ij(m)}), \text{ where } a_{ij(m)} = \bigoplus_{\substack{p \in M_{ij} \\ p \text{ contains exactly } m \text{ arcs}}} w(p).$$

Therefore, if the matrix $D = (d_{ij})$ is the matrix of the elements defined in (1), we get, by the associative law (A_6), the *matrix formulation* of the Algebraic Path Problem:

$$\boxed{\begin{aligned} D = A^* &= \bigoplus_{m \geqq 0} A^m \\ &= I \oplus A \oplus A \otimes A \oplus A \otimes A \otimes A \oplus \dots \end{aligned} \qquad (3)}$$

Note however that the infinite sum in (3) may exist even if not all sums in (1) are defined.

Example 1:
This second formulation is particularly interesting for the semiring H_1, because in this semiring we have

$$D = I + A + A^2 + A^3 + \ldots = (I - A)^{-1}, \text{ if } D \text{ exists}.$$

Therefore the Algebraic Path Problem amounts to finding the inverse of a real matrix in some cases.

Example 2:
In the above example semiring H_2, it is more convenient to look at the first formulation of the Algebraic Path Problem, which becomes then the problem of finding the lengths d_{ij} of shortest paths between each pair of vertices, if we interpret the weight a_{ij} of an arc as its length. If there is no arc from i to j then a_{ij} is set to infinity, the zero element of the semiring.

Since arcs of negative length are allowed, there may be circuits of negative length in the graph and some minima d_{ij} may then not exist, i.e. become minus infinity.

Example 3:
If we interpret the elements of an H_B-matrix A as representing a binary relation $R \subseteq V \times V$ on a set V of n elements, a_{ij} being 1 if and only if the relation holds between the elements i and j, then the relation represented by the matrix D is R^*, the transitive and reflexive closure of the relation R.

Remark 2.2, decomposability of the Algebraic Path Problem:
If we have a partition $\{I_k \mid k \in K\}$ of the vertex set V of the graph G, a matrix A_{kl} of order $|I_k| \times |I_l|$ which corresponds to the arcs of G which are elements of $I_k \times I_l$ can be associated with each pair (I_k, I_l). We may attach these matrices A_{kl}, $1 \leqq k,l \leqq n$, which are elements of the semiring $M(H)$ of all matrices over H, as weights to the arcs of the graph with vertex set J (the quotient graph of G with respect to the partition $\{I_k\}$) and pose the Algebraic Path Problem in this new graph. (Actually, the set $M(H)$ of all matrices over a semiring does not form a proper semiring, since there is not a single zero and a single unity matrix, and addition and multiplication are only defined for matrices of compatible sizes; it would require a few tricks to make $M(H)$ conform to the axioms of a semiring. In this particular Algebraic Path Problem, however, this causes no troubles since all operations are defined.) If the solutions d_{ij} to the Algebraic Path Problem in the original graph G exist, then the elements of the matrices D_{kl} which are the solutions for the quotient graph are just the corresponding d_{ij}.

The matrix formulation of the Algebraic Path Problem is just a special case of this remark for the partition $\{V\}$, the trivial partition of V into one set.

3. The Gauß-Jordan Elimination Algorithm for the Algebraic Path Problem

I am going to present an algorithm for solving the general Algebraic Path Problem, i.e. a procedure which computes the d_{ij} from the a_{ij} using only the semiring operations $\oplus$ and $\otimes$ and the $*$-operation. This algorithm can be used to solve any particular instance of the Algebraic Path Problem, *provided that the problem is solvable*, because then the infinite sums occurring in the algorithm (they occur implicitly in the $*$-operations) exist.

Let i and j be two vertices, and let $0 \leqq k \leqq n$; then

$M_{ij}^{(k)} :=$ the set of all paths from i to j, which contain only vertices x with $1 \leqq x \leqq k$ as intermediate vertices, and

$$c_{ij}^{(k)} := \bigoplus_{p \in M_{ij}^{(k)}} w(p).$$

The intermediate vertices of a path are all vertices except the initial vertex and the final vertex. Thus, the direct path consisting only of the arc from i to j is the one and only path contained in $M_{ij}^{(0)}$, and is of course also contained in every other $M_{ij}^{(k)}$, since these sets are isotone in k. The definition is to be understood such that the empty path starting and ending at vertex i is *not* contained in $M_{ii}^{(i-1)}$ (unlike the loop at node i), but is contained in $M_{ii}^{(i)}$.

$M_{ij}^{(n)}$ is the set of all paths from i to j, i.e. M_{ij}, and therefore $c_{ij}^{(n)}$ is d_{ij}, which we want to compute. $c_{ij}^{(0)}$ is just the weight of the arc from i to j, if it exists, and otherwise zero, i.e. the values $c_{ij}^{(0)}$ are the quantities that we start with.

Recursion formulas for $c_{ij}^{(k)}$:

$$\text{A}: \quad c_{ij}^{(k)} = c_{ij}^{(k-1)} \oplus c_{ik}^{(k)} \otimes c_{kj}^{(k-1)}, \text{ for } 1 \leqq k \leqq n \text{ and } k \neq i,\ k \neq j.$$

A path in $M_{ij}^{(k)}$ either does not pass vertex k at all, in which case it is a member of the set $M_{ij}^{(k-1)}$, or it is uniquely decomposable into the initial segment reaching to the last occurrence of k on the path, which is contained in $M_{ik}^{(k)}$, and the rest of the path, which is in $M_{kj}^{(k-1)}$. By the distributive law (A_5), this relation between the sets $M_{ij}^{(k)}$ etc. carries over to the sum of the path weights over these sets and yields this equation.

$$\text{B}: \quad \left.\begin{aligned} c_{ij}^{(i)} &= c_{ii}^{(i)} \otimes c_{ij}^{(i-1)}, \text{ and} \\ c_{ij}^{(j)} &= c_{ij}^{(j-1)} \otimes c_{jj}^{(j)}, \end{aligned}\right\} \text{ for } i \neq j.$$

The first formula is the same as above, except that a path in $M_{ij}^{(i)}$ must always pass through i (remember that $M_{ii}^{(i)}$ contains the empty path); the second one follows by a symmetric argument.

$$\text{C}: \quad c_{ii}^{(i)} = (c_{ii}^{(i-1)})^*, \text{ for } 1 \leqq i \leqq n.$$

Every path in $M_{ii}^{(i)}$ is uniquely decomposable into a unique number of partial paths from $M_{ii}^{(i-1)}$.

That the above formulas form indeed a recursion by which the $c_{ij}^{(n)}$ can be calculated starting from the $c_{ij}^{(0)}$ can be seen from the following

Sequential algorithm (*Gauß-Jordan Elimination, sGJE*):

```
PHASE 1:
for i from 1 to n
for j from 1 to n
begin
    for k from 1 to $\min(i,j)-1$
                        $c_{ij}^{(k)} := c_{ij}^{(k-1)} \oplus c_{ik}^{(k)} \otimes c_{kj}^{(k-1)}$;        (A)
    if $i=j$ then       $c_{ii}^{(i)} := (c_{ii}^{(i-1)})^*$;                                              (C)
    if $i>j$ then       $c_{ij}^{(j)} := c_{ij}^{(j-1)} \otimes c_{jj}^{(j)}$;                             (B)
end;

PHASE 2:
for i from 1 to n
for j from 1 to n
begin
    if $i<j$ then       $c_{ij}^{(i)} := c_{ii}^{(i)} \otimes c_{ij}^{(i-1)}$;                             (B)
    for k from $\min(i,j)+1$ to $\max(i,j)-1$
                        $c_{ij}^{(k)} := c_{ij}^{(k-1)} \oplus c_{ik}^{(k)} \otimes c_{kj}^{(k-1)}$;        (A)
    if $i<j$ then       $c_{ij}^{(j)} := c_{ij}^{(j-1)} \otimes c_{jj}^{(j)}$;                             (B)
end;

PHASE 3:
for i from 1 to n
for j from 1 to n
begin
    if $i>j$ then       $c_{ij}^{(i)} := c_{ii}^{(i)} \otimes c_{ij}^{(i-1)}$;                             (B)
    for k from $\max(i,j)+1$ to n
                        $c_{ij}^{(k)} := c_{ij}^{(k-1)} \oplus c_{ik}^{(k)} \otimes c_{kj}^{(k-1)}$;        (A)
end;
```

The order in which each of the three loops of the variables i and j is executed is relevant only as to ensure that all (i',j') with $i' \leqq i$ and $j' \leqq j$ are processed before (i,j).

The first loop over i and j (*phase* 1) starts from $c_{ij}^{(0)}$ and computes $c_{ij}^{(j)}$ if $i \geqq j$ and $c_{ij}^{(i-1)}$ if $i<j$;

the second loop (*phase* 2) starts from these values and computes $c_{ij}^{(j)}$ if $i \leqq j$ and $c_{ij}^{(i-1)}$ if $i>j$;

the final loop (*phase* 3) then computes the $c_{ij}^{(n)}$.

In this algorithm the parenthesized superscript of each of the terms c_{ij} may in fact be understood as differentiating between successive values of one variable c_{ij}.

To see this and to verify that each variable on the right hand side of the assignments in each of the three great loops carries the correct superscript is now a routine task, if you take into account in each case whether the variable has already gone through that loop or not.

Note that although all *-operations in the Gauß-Jordan algorithm may be defined even in the strict sense of (C) that the corresponding infinite series (2) converges, the sum (3) and consequently the sum (1) need not exist.

The number of assignment statements that are executed in this algorithm is n^3: For each combination of i, j, k, there is exactly one assignment to $c_{ij}^{(k)}$. There are n *-operations, $n^3 - n$ multiplications and $n \cdot (n-1)^2$ additions.

4. The Gauß-Jordan Elimination Algorithm for Special Semirings

Example 1: Inverting a real matrix:

The operations for the semiring H_1 are of the following kinds:

$$c := c + ab \qquad \text{(A)}$$

$$c := ab \qquad \text{(B)}$$

$$c := \begin{cases} c^* = 1/(1-c), & \textit{if } -1 < c < +1 \\ \text{undefined}, & \text{otherwise} \end{cases} \qquad \text{(C)}$$

The algorithm computes $(I-A)^{-1}$ if the solution to the Algebraic Path Problem exists. This is for example the case if all elements of A are less than $1/n$ in absolute value, i.e. if A is sufficiently close to the zero matrix. But the elements of the inverse $(I-A)^{-1}$ are just rational functions in the elements of A, and the algorithm likewise computes rational functions in its input. Since the two rational functions coincide in an open subset of the input space $\mathbb{R}^{n \times n}$, namely in a neighborhood of the zero matrix, they must coincide everywhere in their common domain. Thus we may extend the operations of type C as follows:

$$c := \begin{cases} 1/(1-c), & \text{if } c \neq 1 \\ \text{undefined}, & \text{if } c = 1, \end{cases} \qquad \text{(C')}$$

and the algorithm will now compute $(I-A)^{-1}$ almost always.

Usually one wants to compute A^{-1} directly for a given matrix A and not $(I-A)^{-1}$. This can be achieved with some simple cosmetic transformations:

$$c := c + ab \qquad \text{(A)}$$

$$c := ac \qquad \text{(B}_{\text{I}}\text{)} \quad \text{in lines 12 and 20 of the algorithm,}$$

$$c := -cb \qquad \text{(B}_{\text{O}}\text{)} \quad \text{in lines 7 and 15 of the algorithm,}$$

$$c := \begin{cases} 1/c, & \text{if } c \neq 0 \\ \text{undefined}, & \text{if } c = 0, \end{cases} \qquad \text{(C'')}$$

A "1" has been omitted in operation (C''), which corresponds to the missing identity matrix, and additionally, the sign has been changed in operations (B_O) and (C''). The correctness of the transformation can be proved by comparing the computations of the two algorithms phase by phase.

In this form the algorithm corresponds to the conventional Gauß-Jordan elimination algorithm without pivoting of numerical mathematics.

The three phases of the algorithm as arranged in sGJE have a very natural interpretation in terms of linear algebra:

In phase 1 an LU-decomposition is performed, except that the diagonal elements computed after phase 1 are already $c_{ii}^{(i)} = (c_{ii}^{(i-1)})^{-1}$ instead of $c_{ii}^{(i-1)}$:

$$\begin{pmatrix} c_{11}^{(0)} & c_{12}^{(0)} & c_{13}^{(0)} & \cdots & c_{1n}^{(0)} \\ c_{21}^{(0)} & c_{22}^{(0)} & c_{23}^{(0)} & \cdots & c_{2n}^{(0)} \\ c_{31}^{(0)} & c_{32}^{(0)} & c_{33}^{(0)} & \cdots & c_{3n}^{(0)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ c_{n1}^{(0)} & c_{n2}^{(0)} & c_{n3}^{(0)} & \cdots & c_{nn}^{(0)} \end{pmatrix} = \begin{pmatrix} 1 & & & & 0 \\ -c_{21}^{(1)} & 1 & & & \\ -c_{31}^{(1)} & -c_{32}^{(2)} & 1 & & \\ \vdots & \vdots & \vdots & \ddots & \\ -c_{n1}^{(1)} & -c_{n2}^{(2)} & -c_{n3}^{(3)} & \cdots & 1 \end{pmatrix} \cdot \begin{pmatrix} c_{11}^{(0)} & c_{12}^{(0)} & c_{13}^{(0)} & \cdots & c_{1n}^{(0)} \\ & c_{22}^{(1)} & c_{23}^{(1)} & \cdots & c_{2n}^{(1)} \\ & & c_{33}^{(2)} & \cdots & c_{3n}^{(2)} \\ 0 & & & \ddots & \vdots \\ & & & & c_{nn}^{(n-1)} \end{pmatrix}$$

$$A \qquad = \qquad L \qquad \cdot \qquad U.$$

The two triangular matrices are then inverted in phase 2:

$$L^{-1} = \begin{pmatrix} 1 & & & & 0 \\ c_{21}^{(1)} & 1 & & & \\ c_{31}^{(2)} & c_{32}^{(2)} & 1 & & \\ \vdots & \vdots & \vdots & \ddots & \\ c_{n1}^{(n-1)} & c_{n2}^{(n-1)} & c_{n3}^{(n-1)} & \cdots & 1 \end{pmatrix} \text{ and } U^{-1} = \begin{pmatrix} c_{11}^{(1)} & c_{12}^{(2)} & c_{13}^{(3)} & \cdots & c_{1n}^{(n)} \\ & c_{22}^{(2)} & c_{23}^{(3)} & \cdots & c_{2n}^{(n)} \\ & & c_{33}^{(3)} & \cdots & c_{3n}^{(n)} \\ 0 & & & \ddots & \vdots \\ & & & & c_{nn}^{(n)} \end{pmatrix}.$$

Finally, in phase 3, the two inverted matrices are multiplied, and we have:

$$U^{-1} \cdot L^{-1} = \begin{pmatrix} c_{11}^{(n)} & c_{12}^{(n)} & c_{13}^{(n)} & \cdots & c_{1n}^{(n)} \\ c_{21}^{(n)} & c_{22}^{(n)} & c_{23}^{(n)} & \cdots & c_{2n}^{(n)} \\ c_{31}^{(n)} & c_{32}^{(n)} & c_{33}^{(n)} & \cdots & c_{3n}^{(n)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ c_{n1}^{(n)} & c_{n2}^{(n)} & c_{n3}^{(n)} & \cdots & c_{nn}^{(n)} \end{pmatrix} = A^{-1}.$$

Example 2: Shortest distances:

The operations for the semiring H_2 are of the following kinds:

$c := \min(c, a+b)$ (A) (the so-called "triple operation")

$c := c + b$ (B)

$$c := \begin{cases} 0 & \text{if } c \geqq 0 \\ -\infty & \text{if } c < 0 \end{cases} \quad \text{(C)}$$

This is the well-known algorithm of Floyd [1962].

Example 3: Transitive closure:

The operations for the semiring H_B are of the following kinds:

$$c := c \vee (a \wedge b) \qquad \text{(A)}$$
$$c := c \wedge b \qquad \text{(B)}$$
$$c := 1 \qquad \text{(C)}$$

Here the diagonal elements of the solution are known in advance, they will be 1, and so will therefore be the operand b of all type (B) operations. Thus, the type (B) operations are identity operations and can be omitted. If it is desired to compute the transitive closure but not the transitive reflexive closure of a relation, this can be achieved by a slight modification of the algorithm (omitting type B and C operations). The resulting algorithm is the well-known Warshall-Roy transitive closure algorithm (Roy [1959], Warshall [1962]).

5. Description of the Systolic Array GJE0

The systolic array which performs the computations of the sGJE algorithm of section 3 is an $(n+1)\times(n+1)$ hexagonal array of processors which are arranged in a diamond shape and connected in a regular way as shown in the upper part of Fig. 1. The connections are one-way connections, as indicated by the arrows on them; the long connections below and above the array are the (global) input and output connections to the outside world. There are several types of processors, depending on the location in the array (marked with different letters in Fig. 1).

In principle, the array operates synchronously in time units which I shall call steps. In one step each processor performs a few simple operations on the data from its input connections, the type of the operations depending on the processor type, and sends the results of these operations along its output connections. In every step the inputs of a processor are the outputs of its connected processors from the previous step (or the data arriving from outside along the global input lines).

In the present scheme the processors need no additional memory to remember any data between successive steps. Conceptually, it is therefore most convenient to think of there being one register associated with each connection, capable of holding a single value (element of the semiring).

Types of processors and their operations (see Fig. 2):

Firstly, there are three types corresponding to the three types of assignments in the sGJE algorithm:

Type A is the so-called "inner product step processor", because it carries out one step in the computation of the inner product of two vectors:

$$c' := c \oplus (a \otimes b);$$

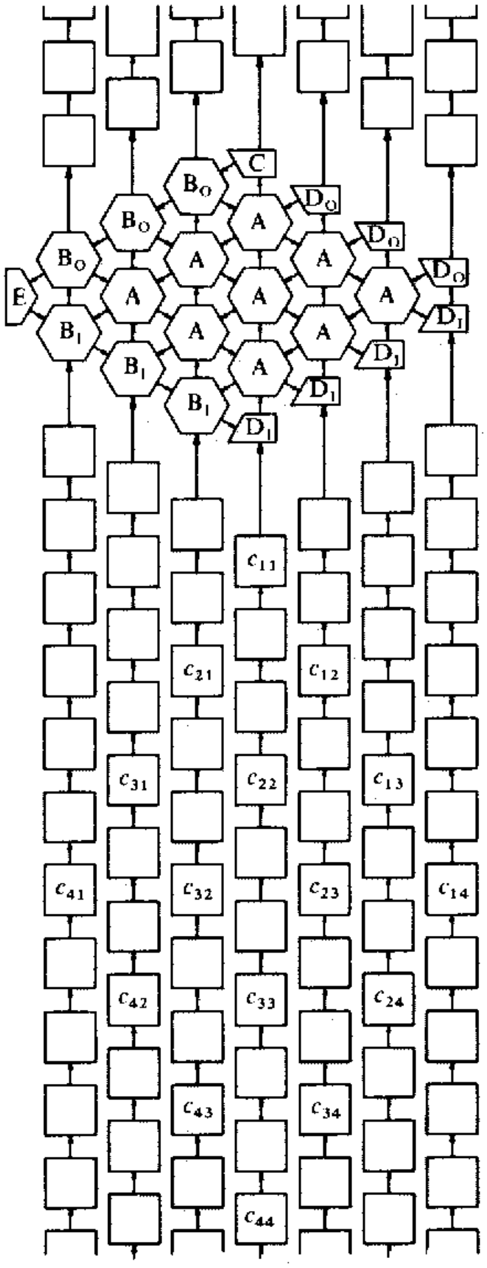

C
B$_O$
D$_O$
A
E
D$_I$
B$_I$
c_{11}
c_{21}
c_{12}
c_{31}
c_{22}
c_{13}
c_{41}
c_{32}
c_{23}
c_{14}
c_{42}
c_{33}
c_{24}
c_{43}
c_{34}
c_{44}

Fig. 1. GJE0

The upper part shows the systolic array GJE0 for a 4 × 4-matrix. In the squares the geometric shape in which the matrix elements have to be input is shown. (The squares are not part of the array proper but serve only to illustrate the places and times at which the input data must be presented to the array.)

Type B, which exists in two variations, B_I and B_O, performs simply a multiplication:

$$b' := a \otimes b, \text{ or}$$

$$c' := c \otimes b;$$

and *Type C* performs the star operation:

$$c' := c^*;$$

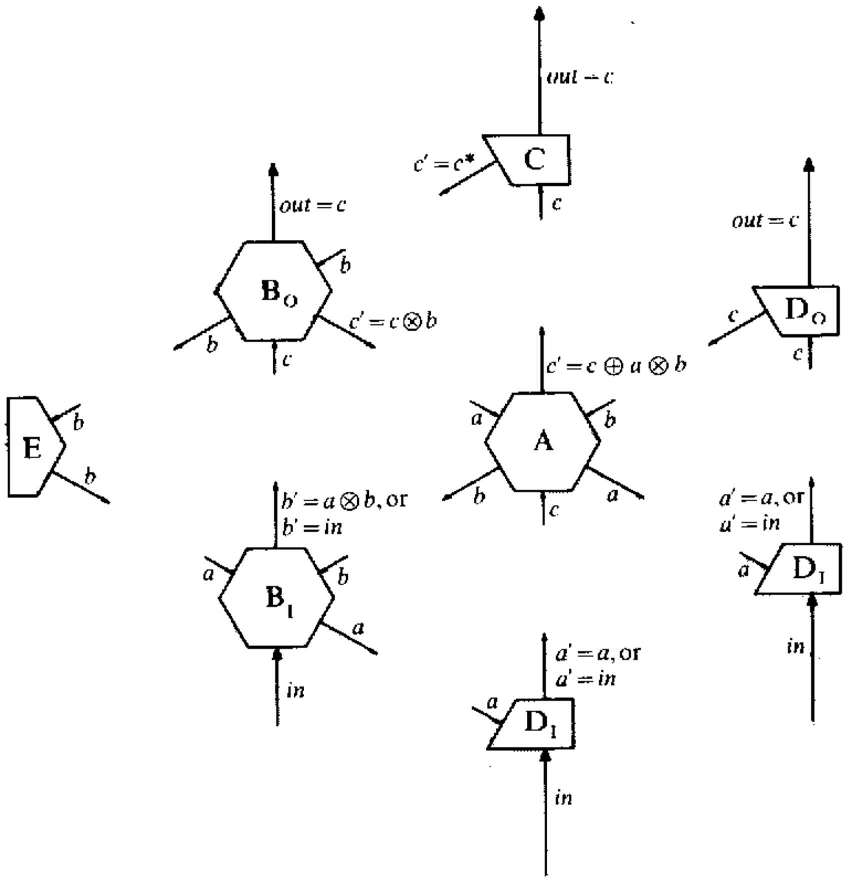


Fig. 2. Types of processors
The diagram shows only the data operations (the control operations are dealt with in section 7)

plus three types of processors, D_I, D_O, and E, which do nothing but copy their inputs to their outputs, thus acting merely as a one-step delay, as do also the former processors with all but one of their inputs.

In the beginning the processors are assumed to be cleared to zero such that the operations of the processors are identity operations.

Then the matrix is input to the processors on the lower border according to Fig. 1.

When the processors on the lower (input) border (B_I and D_I) receive input from below they pass it upwards unaltered; this supersedes their normal computation. However, there is no serious problem, because the input that is fed into the processor array is organized in such a way (see immediately below and section 7) that there is always either input from below or data arriving on the other connections, but not both at the same time. Similarly, as an alternative action to sending the output (c' or c) diagonally downward, the output processors at the top (B_O, C, and D_O) pass the data from below to the global output ports of the array unaltered. Only one of these two actions is asked for at any time, depending on the desired data flow.

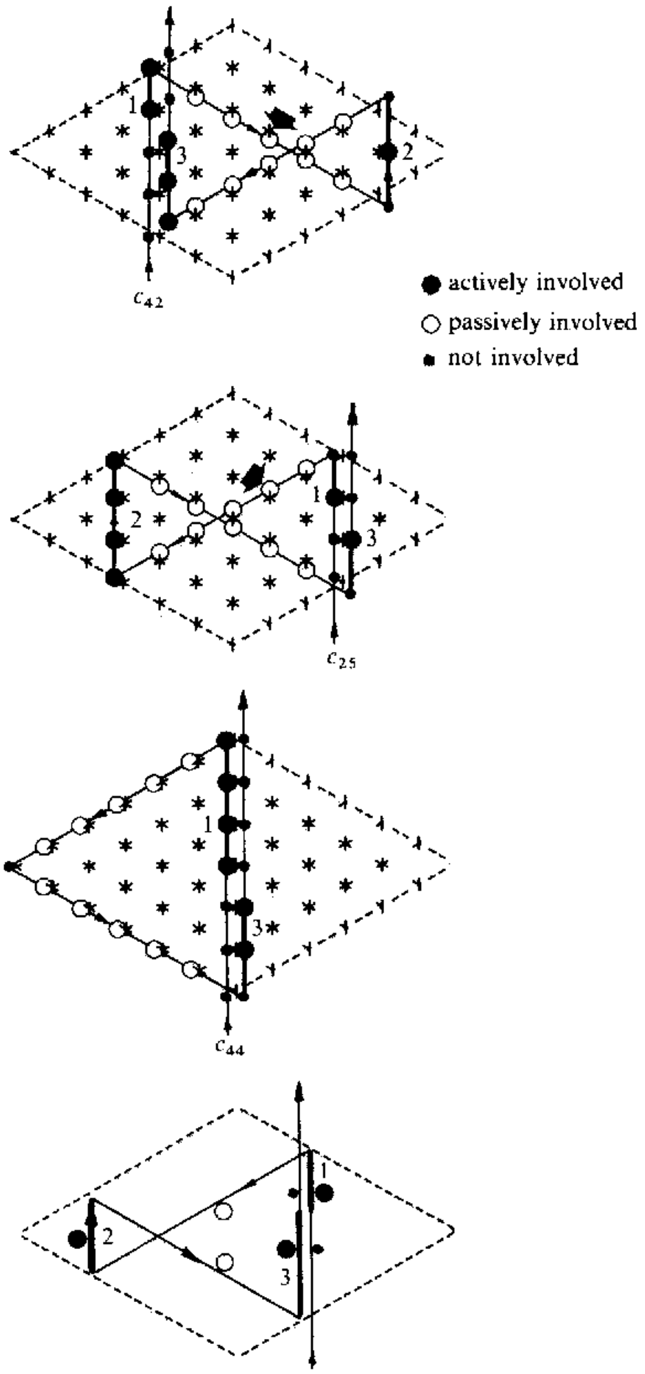


Fig. 3. The paths that the elements of a matrix take through the array

Three elements c_{ij} of a 6×6-matrix have been chosen representative of the three cases $i \geqq j$, $i \leqq j$, and $i = j$. The last figure is somewhat idealized to show the essential of the situation clearly.

The figure shows the places where the elements are actively involved in computations (changing their values), passively involved, and not involved at all for lack of other data. The two thick arrows indicate the place where the elements c_{42} and c_{25} meet. (Their values are used to update c_{45} there.) Each element c_{ij} has exactly $n = 6$ active nodes (●) along its path, where its value is updated from $c_{ij}^{(k-1)}$ to $c_{ij}^{(k)}$, for $k = 1, 2, \ldots, n$. Note that the active nodes lie only on the three vertical sections of the path of each element. The labels near the active sections denote the three phases of the sGJE algorithm; in the case $i = j$ phase 2 is degenerate. Observe that phases 1 and 3 are complementary on the vertical line of processors carrying the matrix element initially.

The data flow:

The most characteristic feature of a systolic array algorithm, how the data move to meet in one processor at the right times, is now going to be described.

Each element of the matrix can be regarded as a variable moving or flowing through the processor array, preserving its identity, but possibly changing its value. The direction in which each processor sends each of the data elements it receives is indicated in Fig. 2 by similar names attached to corresponding inputs and outputs of a processor (a and a', b and b', c and c') or equal names if the value does not change. From this you can see that generally data elements are passed on straight in the direction in which they move, except when they hit an edge of the processor array: then they are (geometrically) reflected.

The matrix C is input from below as illustrated in the lower part of Fig. 1 and finds its way round the processor array as follows (to summarize what can be concluded from Fig. 2) (see Fig. 3):

After an element of the matrix has been input from below it continues flowing upward until it hits the top border for the first time. There it is reflected downward: right downward, if it hits the top left edge, and left downward, if it hits the right edge or if it hits the middle corner (this last case has been decided arbitrarily). It continues in this direction until it hits the opposite parallel edge, which sends it upward for the second time. Again it reaches the top edge and is reflected downward: now it is reflected down right if it has been reflected down left the first time, and vice versa. (In the left corner in processor E, these last two reflections appear merged into one.) It is sent upward for the third time as soon as it hits the bottom edge. It is then in the same vertical line where it arrived. Now, *after the data element has been reflected once at each edge* of the rhombic array of processors, it will be passed on straight upward to the output when it hits the top edge.

Fig. 4 illustrates the positions of all elements of the matrix at one stage during the computation.

Details about the organization of this data flow are presented in section 7.

It is easy to see (Fig. 3) that the length of the path that is covered by an element of the matrix is $3n$ steps longer than it would be if that element had traveled right through the processor array without being reflected, and therefore the matrix comes out at the top in the same shape and orientation in which it was input, but with a delay of $3n$ steps. The first element, c_{11}, comes out $4n+1$ steps after it has been input. The last element, c_{nn}, comes out $7n-2$ steps after c_{11} has been input.

The left half of the data (corresponding to the lower triangular half of the arc length matrix) is reflected towards the right side when it first hits the top border, whereas the right half inclusive of the elements c_{ii} of the main diagonal of the matrix, which come up in the middle, is reflected to the left. Thus, the matrix is cut into two halves, which move independently of each other. However, when the elements move upwards for the third time, the two halves are reunited.

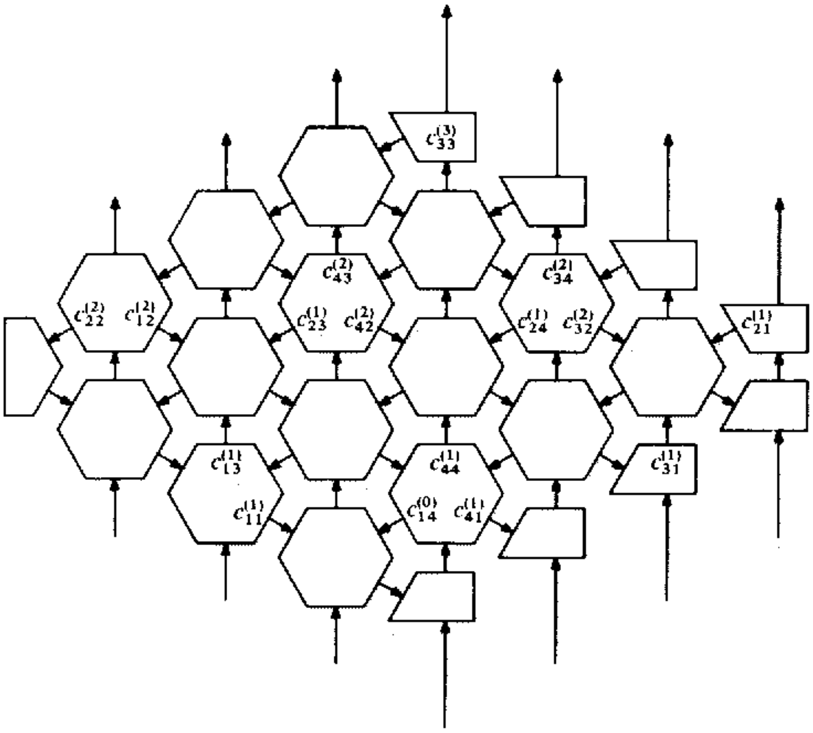


Fig. 4. The position of the elements of the matrix of Fig. 1 after 11 steps. The values are drawn in the processor which the respective connection comes from. Only one third of the processors are active

Since in each reflection the angle of incidence relative to the reflecting edge is equal to the angle of reflection, the shape of the matrix is not distorted when the matrix is reflected, and it is therefore not hard to view the data flow globally. (It may even be modestly modeled by wrapping a piece of paper in the shape of half of the matrix around a piece of cardboard in the shape of the array.)

Fig. 5 illustrates the global data flow of the whole matrix; it shows the space that the matrix occupies during several phases of the computation.

During its course through the systolic array each data element is at times "actively" involved in computation (it changes its value – during the time when it marches upward or hits the left edge), at times "passively" involved in computation (as input data to some other computation – when it marches along one of the two diagonal directions), and sometimes not involved in computations at all (when it meets no other matrix elements). These phases are indicated in Fig. 3.

The three upward (active) phases correspond to the three great loops in the sequential algorithm above. Note that during the second active phase the two halves of the matrix (the c_{ij} with $i>j$, and the c_{ij} with $i \leq j$) compute separately in each half of the array, not interacting with each other.

The following considerations, in connection with Fig. 7, are intended to explain intuitively why the data flow described above implements the sGJE algorithm of section 3 correctly.

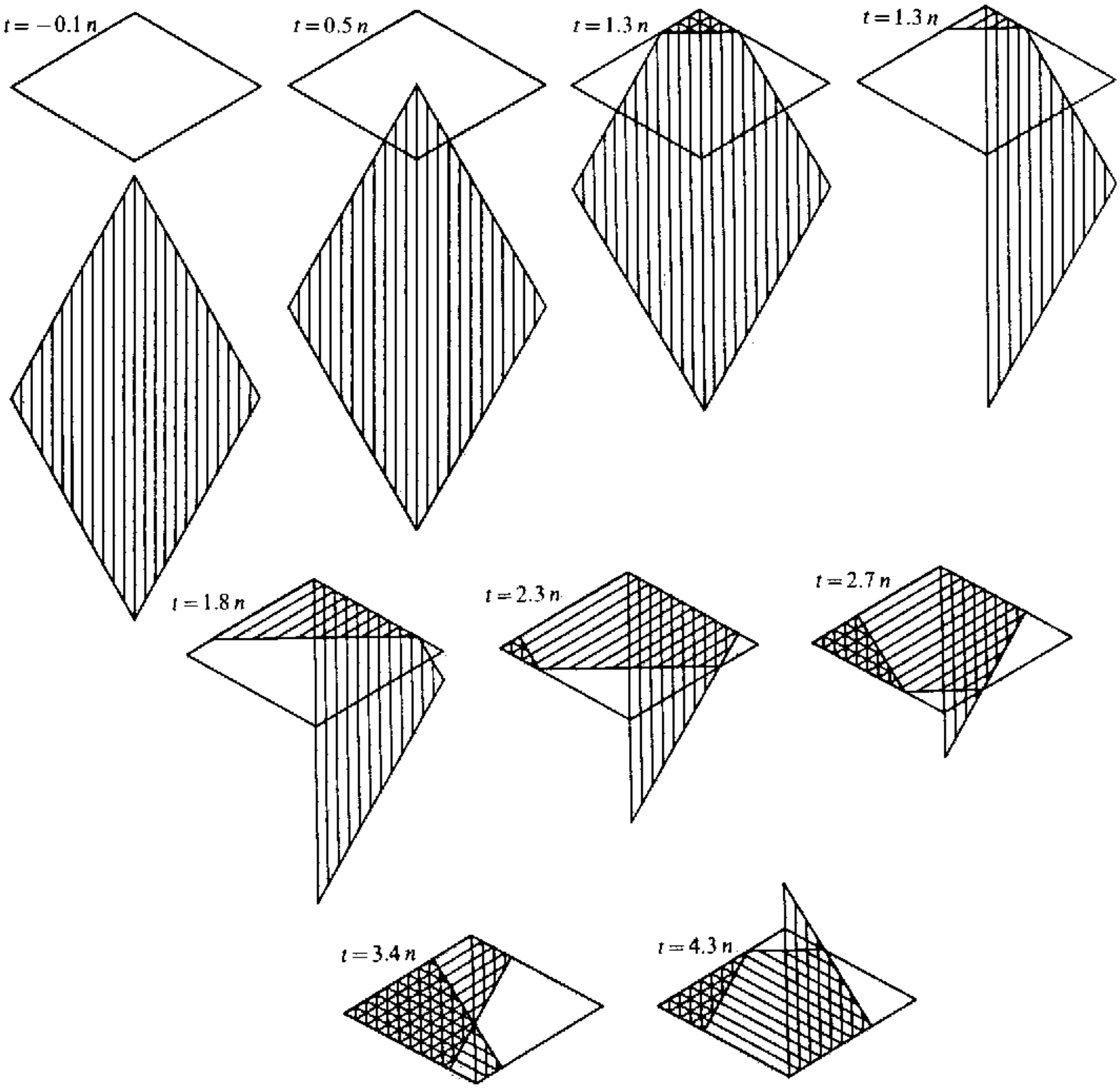


Fig. 5. Successive phases of the computation

This sequence of figures provides a global view of the data flow. In each picture, the small $(n \times n)$-rhombus represents the processor array, and the matrix is represented by the full area which it would occupy. The parallel lines inside the matrix indicate the current direction of movement of the elements. To show the situation more clearly, the diagrams are idealized; actually, there is only a discrete number of processors and matrix elements, and the time t is measured in discrete steps.

In the beginning (time $t=-0.1\,n$) the matrix lies outside the array, like in Fig. 1. It moves upwards ($t=0.5\,n$), and at time n the first elements are reflected ($t=1.3\,n$). Only one half of the matrix is shown from now on lest the pictures become too confusing; the other half of the matrix lies symmetrically to this one. In each of the remaining diagrams, the picture of the full matrix could be obtained by superimposing the original picture with its vertically reflected image (reflected along the vertical symmetry axis of the array). After time $2\,n$ ($t=2.3\,n$), the matrix is reflected three times. To clarify this diagram, Fig. 6 shows how it can be obtained from the original shape of the matrix by three successive foldings along the broken lines. Between time $3\,n$ and $4\,n$ ($t=3.4\,n$), the matrix is folded four times, and it is now completely inside the processor array. At time $4\,n$, the matrix starts to leave the array. The remaining phases of the computation can be obtained by turning the pictures upside down and taking them in the opposite order; for example, the diagram for time $t=4.3\,n$ is the reflection of the diagram for $t=2.7\,n$ along the horizontal symmetry axis of the array; in general, the diagrams for time t and for time $t^*=7\,n-t$ are reflected images of each other.

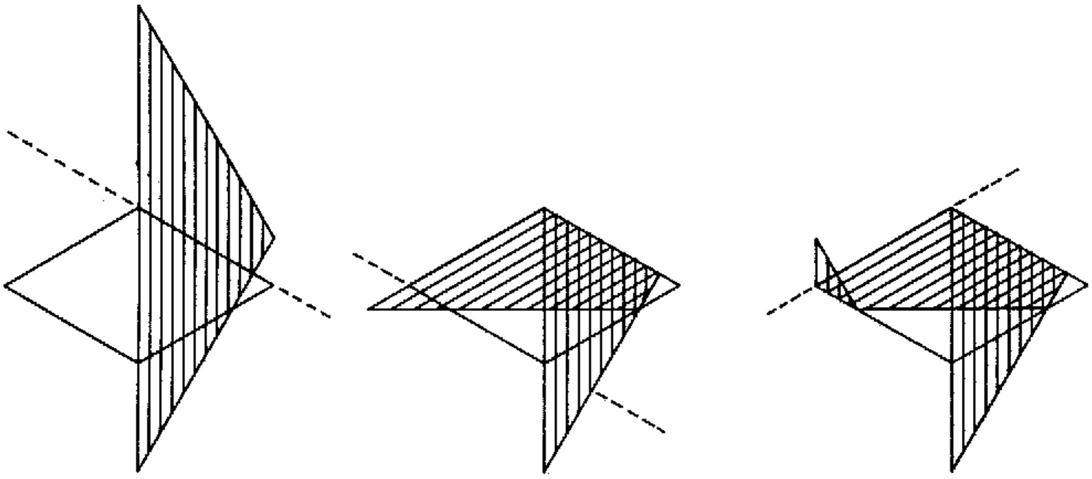


Fig. 6. How the diagram of Fig. 5 for $t = 2.3\,n$ can be obtained by a sequence of three successive foldings

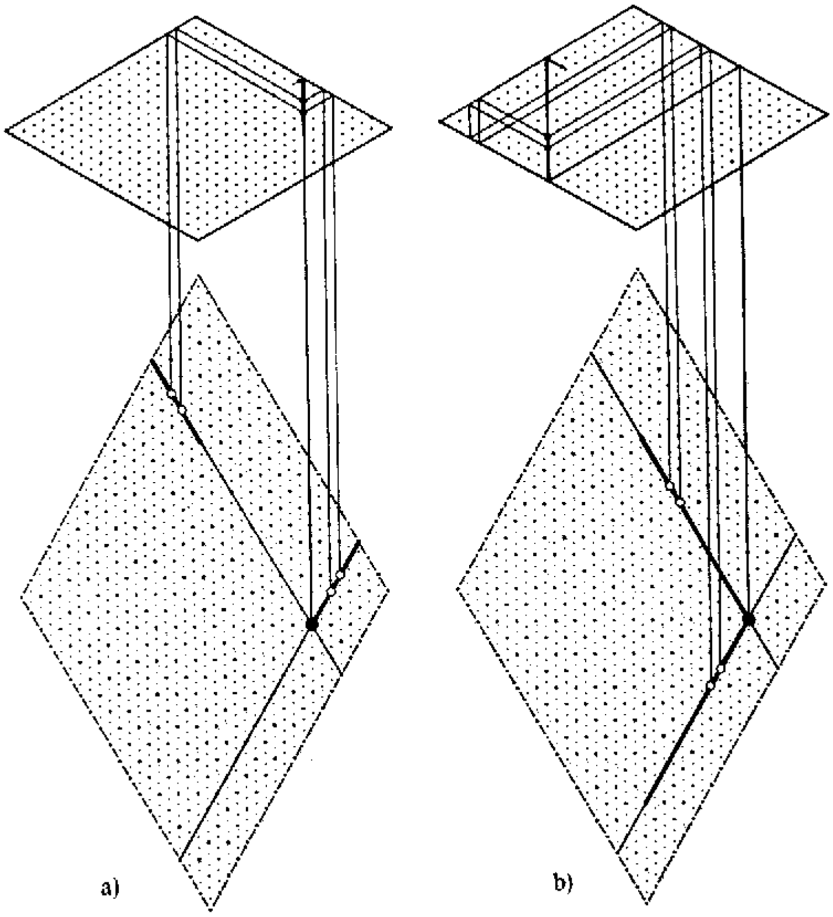
The initial position of the matrix elements, before the matrix enters the array (cf. Fig. 1) is related to their position at some later time (cf. Fig. 4) by the condition that they have traveled the same distance, because they all travel at the same speed. Thus it is possible to determine the original position in the matrix (and hence the identity) of the elements that meet in some processor at some instant, as is done in Fig. 7.

What is shown in Fig. 7 could of course be made stricter and proved analytically. It would be necessary to express explicitly the relation between the variables t, i, j, x, and y defined by the statement: "At time t, the matrix element c_{ij} is in processor P_{xy} (and leaves in a certain direction)." Then one could derive, which elements meet at what time in which type of processors, and which computations take place in those processors. In [Rote 1984] it is shown how the details of such a proof can be worked out.

To implement the algorithm given in example 1 of section 4 for computing the inverse A^{-1} of a real matrix directly, one simply has to assign the operations ($\mathrm{B_I}$), ($\mathrm{B_O}$), and (C″) to the respective processors $\mathrm{B_I}$, $\mathrm{B_O}$, and C.

6. Performance Analysis and Improvements

When the action of the array on a matrix is analyzed one can observe that not all processors are active in computation at any moment. Always at least two thirds of them are idle computing ⓪ ⊕ ⓪ ⊗ ⓪. This comes because the matrix does not fill up the processor array densely (see Figs. 1 and 4). This fact can be exploited, however, to allow some restricted form of pipelining: One step after inputing the first element of a matrix at the bottom corner of the array you may start inputing the first element of another matrix (at the same position in the array) and continue feeding in elements of this matrix straight after you have input the corresponding elements of the first matrix. The computations on the two matrices will then be completely independent of each other and each element of the second matrix will be output



● c_{ij}

○ elements that meet c_{ij}

: the processors where they meet

Fig. 7. The paths of some elements that meet in the same processor

The pictures show some element c_{ij} and four pairs of elements that meet c_{ij}. Correctness might be verified by counting the distance along the paths from the initial position of the elements to the processor where they meet, in units of processing steps.

By generalization, one can see that the elements that meet c_{ij} are pairs (c_{ik}, c_{kj}) of corresponding elements of the i-th row and the j-th column of the matrix. In the left picture, k ranges from 1 to $\min(i,j)$, as long as c_{ij} is on the first vertical (active) section of its path; in the right picture, c_{ij} is in its second active phase, and k ranges from $\min(i,j)$ to $\max(i,j)$. For the third phase, an analogous picture could be drawn.

It can also be seen that c_{ik} enters the processor always from the top left and c_{kj} from the top right.

exactly one step after the corresponding element of the first matrix. There is still room for a third matrix right after the second one, but after that one you have to wait until the first matrix leaves the processor array before you can input another one.

Thus, *the array can be regarded as consisting of three independent "tracks" for computation.* Each processor belongs in turn to each of the three tracks.

How long does one have to wait to put in another matrix on one track? The two matrices must not overlap, so element c_{11} of the second matrix can be input to the processor at the bottom corner of the array just when element c_{nn} of the previous matrix has left this processor for the second time, i.e. three steps after this event, to arrive in the same track. Therefore, since c_{11} and c_{nn} of the same matrix are $3(n-1)$ steps apart, and since it takes $3n$ steps for c_{nn} to come back to the processor where it has entered, element c_{11} of the next matrix can be input $6n$ *steps* after c_{11} of the first matrix has been input.

Taking into account the three tracks, every three $(n \times n)$ input matrices occupy n^2 processors (not counting the types D_I, D_O, and E, which do not perform any operations) for $6n$ steps to carry out $3n^3$ operations. Thus, processor utilization is one half. The other half of the time the processors compute the identity operation, i.e. they simply move the data about. The following considerations show how this can be improved such that the processors are fully utilized.

If we examine which processors are active and which are passive during the course of the matrix, we find that the processors on the same horizontal level are at the same time either all active or all passive (cf. Fig. 5). Fig. 8 shows at what time which levels are utilized. (An analytic proof that the active processors do not exceed the area indicated in Fig. 8 can be found in Rote [1984, section 7].)

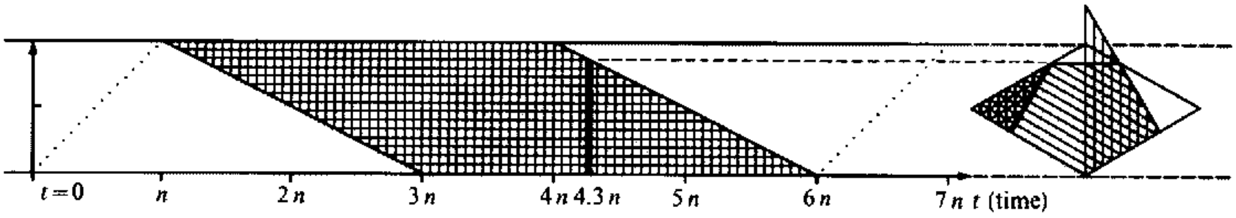


Fig. 8. Processor utilization in GJE0

This diagram expresses which processors are used at what time to compute new values for elements of the matrix (active) and which are used simply to shuffle the data about (passive). Since all processors (of one track) that lie on one horizontal line are identical in this respect, the diagram has been drawn in such a way that its vertical dimension corresponds to the vertical dimension of the processor array. The horizontal axis denotes time measured in steps since the input of the first element of the matrix. To show the situation more clearly the diagram is idealized, like in Fig. 5. At time $7n$, the last element has left the array. The cross-shaded area shows the active processors.
The picture to the right illustrates the position of the matrix elements after about $4.3n$ steps; it is reproduced from Fig. 5. The relation with the shaded area on the left is indicated by the broken line. The active processors are those where three matrix elements from all three directions meet, i.e. three layers of (both halves of) the matrix lie on top of each other.

The shape of the shaded region in Fig. 8 suggests that another specimen of it for a second matrix can be appended without conflict after $3n$ steps, not only after $6n$ steps. The elements of the second matrix will then occupy the same processors as the corresponding elements of the first matrix until these cross the top edge to be output and to leave the array where the elements of the second matrix are reflected. From then on the elements of the second matrix are single again on their ways. So all we have to do is double the vertical connection lines to allow two numbers to travel in parallel. There must now be two c-registers instead of one, four registers all in all for each type A processor. Fig. 9 shows the situation of Fig. 8 when another matrix has closely followed the first one. The resulting array design shall be called *GJE*1 in the sequel. The only question left to decide is which of the two numbers to compute with and which simply to pass on. This issue is discussed in section 7.

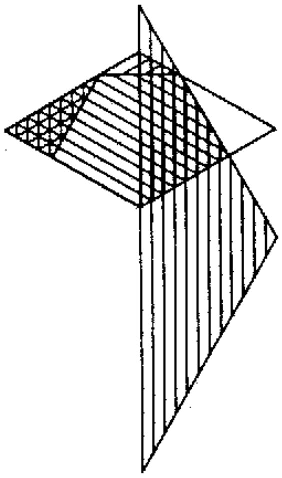


Fig. 9. Systolic array GJE1

The situation in the right part of Fig. 8 with an additional matrix following the first one. This figure is a superposition of the two diagrams from Fig. 5 with $t = 4.3\,n$ and $t = 1.3\,n$

Solving problems of different sizes:

A systolic array should be usable for problems of different sizes without changes in the single processors.

It is easy to use an $(n+1) \times (n+1)$ GJE1 array for a matrix of size smaller than n. Continuing the discussion of how to optimize processor utilization and adapting the shape of the shaded region in Fig. 8 (see Fig. 10), one verifies that it is possible to input the first element of a new matrix in the next free place on the same track (i.e. three steps) after the last element of the previous matrix regardless of the size of these matrices. Each one of the three tracks acts on an infinite block diagonal matrix of the form depicted in Fig. 11 a. Thus we get the result that an $m \times m$-matrix $(m \leqq n)$ occupies a track for $3\,m$ steps (the maximum horizontal width of the shaded area in Fig. 10), i.e. it occupies the whole array for m steps only, in the sense that the time that a great number of matrices take to compute is roughly the sum of the times each one occupies the array.

But with GJE1 it still takes $4\,n + 3\,m - 2$ steps for an $m \times m$-matrix between the input of the first element and the output of the last element of the matrix. For small matrices, this is a relatively long time. Next I shall show how this time can be reduced to $7\,m - 2$, which is the time which would be needed if we had an array

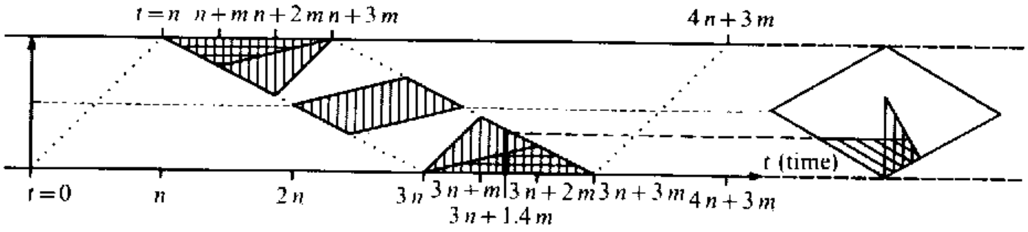


Fig. 10. Processor utilization with a smaller matrix

The diagram of Fig. 8 in the case of an $m \times m$-matrix smaller than the array. m is approximately $0.4\,n$. It is no longer true that a whole horizontal line of processors is completely active or completely passive. The horizontal lines where the processors are only partially active are represented by the vertically shaded area. The three disjoint shaded regions of the diagram correspond to the three active phases of the algorithm. When m becomes larger than $n/2$ the vertically shaded regions start to overlap.
On the right side the position of the matrix after about $3\,n+1.4\,m$ steps is shown in the fashion of Figs. 5 and 8.

especially for $m \times n$-matrices, by a variation of the array, which I shall refer to as *GJE*⟨ (the "⟨" is suggestive of the shape of the array). Imagine that the array is (physically) folded along the vertical symmetry line. There are then in general two processors on top of each other, or we could say, one processor that carries out the action of two; the action of the processors is similar as before, except that the connections of the processors at the folding line point to other directions as before: These processors now reflect the data passing through them. It is now a small step to think of this folded array as part of a large (potentially infinite) wedge (Fig. 12) in which the function of the reflecting processors along the vertical folding line is *programmed*, i.e. all processors are able to act either like central processors or like the processors on the folding line, and this choice is not hard-wired, depending on the location of the processor, but is programmable, for example by a signal that is input with the elements on the main diagonal of the matrix and travels on with them. In this way it is possible to have an array of appropriate size for any (not too large) matrix. How matrices of different sizes can be arranged consecutively on one track is shown in Fig. 11 b. The shape of the shaded region in Fig. 8 does not change, it only shrinks or grows proportionally to the size of the matrix and still lies equally above and below the horizontal symmetry line of the array. One further advantage of the array GJE⟨ is that it is open to the right and thus readily extensible.

Processing of a matrix larger than the array:

A different question is how a matrix larger than the array can be processed. Here the decomposability principle (remark 2.2) becomes useful. By this principle, which is the analogue of the fact that matrix multiplication can be carried out with matrices made up of matrix blocks just as with ordinary matrices, we can decompose a too large matrix into blocks, apply the Gauß-Jordan algorithm on the block level and, whenever the *-operation is needed on the higher level, use the algorithm on the level of a matrix consisting of single elements. Thus one way to process a large matrix is to apply sequential Gauß-Jordan elimination (or any other similar algorithm) on

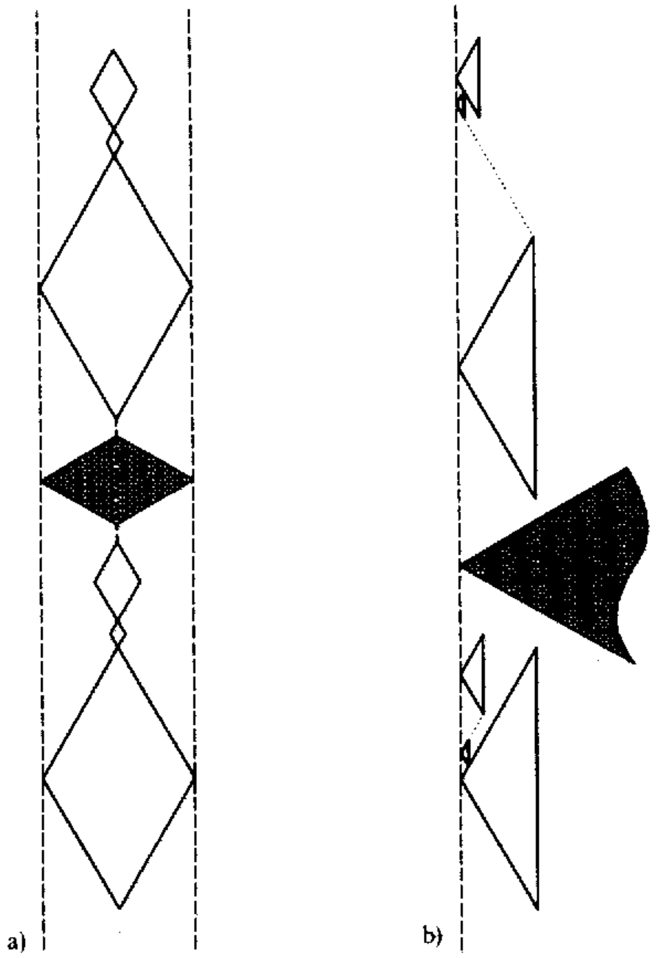


Fig. 11. Processing a sequence of matrices. a) GJE1, b) GJE⟨

The shapes and relative positions of three of an unbounded sequence of matrices of different sizes before (= below) and after (= above) passing through the systolic array (hatched) on one of the three tracks. The matrices are shown on a (conceptual) band of one-step delay processors that extend the array below and above like in Figs. 1, 5, 7, and 12

a) *Rhombic array GJE*1 with a fixed upper bound on the size of the matrix

The input stream can be regarded as an infinite block diagonal matrix. The position of a matrix after the array is $3n$ steps behind where it would be had it gone straight through the array without reflections, where n is the dimension of the array

b) *Open wedge GJE*⟨, extensible to the right

The matrices are folded along their main diagonal and aligned by their left corners. Observe that the relative positions of the matrices have changed when they have passed the array, and that smaller matrices have advanced relative to the larger ones. The position of a matrix of size $m \times m$ after the array is $3m$ steps behind where it would be had it gone straight through the array without reflections, which is just the length of the main diagonal

the higher level, resorting to systolic arrays for the single steps of this algorithm: multiplication of two blocks (see Kung and Leiserson [1978] or Kung [1980]) and the *-operation. The time is reduced by a factor proportional to n^2 compared with the sequential algorithm, where n is the size of the arrays.

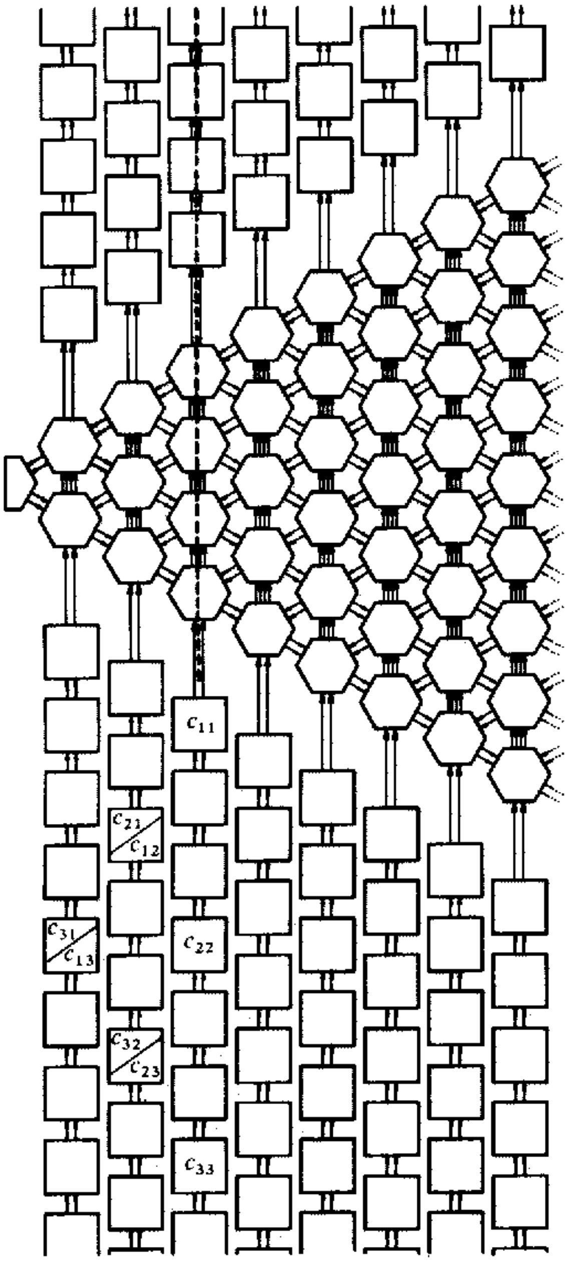


Fig. 12. The systolic array GJE<

A 3 × 3-matrix folded along its main diagonal is input to a wedge of processors which is open to the right. By a signal which is input together with c_{11} and propagates upwards the processors along the broken line are temporarily programmed to act as reflecting processors. The matrix has the impression of passing through an array for 3 × 3-matrices

7. Notes on the Implementation

The processors B_1 and D_1, which accept the input from the outside world, present a little difficulty, since their outputs b', or a', resp., are determined in two ways: in the beginning they should pass on the inputs from below, and the second time they should calculate their output from their top left and top right inputs, or copy the top left input, respectively (see Fig. 2).

B_O, C, and D_O, the output processors at the top, have a similar ambiguity: they should perform their calculation and reflect the data downwards the first time, and output the data unaltered the second time, since we don't want the data to continue bouncing from corner to corner, disturbing the flow and computations of data coming behind, but we need the processors to be in a cleared state afterwards.

A solution which needs only local control is to give a bit (actually, at least two bits are necessary) more intelligence to the processors, providing each data element with a flag that counts how often it has been reflected at the top line, and letting the processors act accordingly.

This device can also be used for the improved version GJE1 of section 6, where two data elements are allowed to travel upwards in parallel, but only one of them is to be modified by computations with the data from the other input lines. Of those two data elements, one has been reflected twice and the other one nonce. By looking at the flags of the other two inputs the question may be decided.

Similar techniques can be used in the array GJE$\langle$, where the signals indicating whether a processor has to act as a reflecting processor must be organized somehow. For more details see [Rote 1984, section 6].

8. Conclusion and Review

The processors of the presented arrays are of few types independent of the total size of the array and the size of the problem, and thus the arrays GJE0 and GJE1 meet one design objective for systolic arrays.

Moreover, the inner product step processor (type A), which is the "work horse" in the array GJE0, is also the only processing element in the systolic array for forming the inner product of two vectors and in the well-known systolic array which multiplies two matrices (Kung and Leiserson [1978]; also described in Kung [1980] or Rote [1984]). By restricting the array of this paper appropriately, one can obtain systolic arrays for the three phases of the algorithm, i.e. in the case of the ordinary real matrices (see example 1 in section 4) for *LU*-decomposition, inversion of an upper or lower triangular matrix, and the multiplication of an upper triangular with a lower triangular matrix. The last of these three is not so interesting, as it is also a restriction of the ordinary matrix multiplication array. The first one has been previously described by Kung and Leiserson [1978] and by Kung [1980]. The nice property of this family of systolic arrays is that they all consist of an area filled homogeneously with the same processors, and by bordering this area with different chains of processors one gets the different arrays.

In the literature there have been two systolic arrays, to my knowledge, solving instances of the Algebraic Path Problem. Both are $n \times n$ rectangular arrays. The first one by Guibas, Kung and Thompson [1979] computes the transitive closure of a Boolean matrix, the second one by Kramer and van Leeuwen [1982] inverts a real matrix. Both arrays are constructed as implementations of Gauß-Jordan elimination (i.e. the Warshall-Roy algorithm in the case of the transitive closure). The array of this paper was developed independently of these other arrays, but there are relations to both of them.

The sequence of computations in the array of Guibas, Kung and Thompson [1979] is akin to that of GJE0, in the sense that the rendez-vous among data elements, which may then be combined in computations, correspond roughly. However, this sequence of computations is realized by a different data flow: There are three copies of the matrix, one remains fixed and the two others simultaneously move across the array in perpendicular directions. Normally, the resident elements are updated using in the computations the stable values of the moving copies that are passing over them. However, when a moving element meets its resident copy then it assumes the updated value of the resident element; this has an effect on the data flow which is achieved by reflection in GJE0. Three identical passes over the array have to be run until the solution is stable. However, the array computes "too much", i.e., roughly speaking, it does carry out all steps of the sGJE algorithm in the right sequence but some steps of that algorithm occur repeatedly; thus some paths occur more than once instead of exactly once in the computed sum (1). This is the reason why the algorithm can in fact be extended to compute shortest distances, but not to solve Algebraic Path Problems in semirings in which addition is not idempotent.

The array by Kramer and Leeuwen [1982] is more like the one in this paper. It is a true parallelization of Gauß-Jordan elimination. One copy of the matrix moves through the array in three directions and is reflected when it hits the boundary. Each element goes through the same sequence of active and passive phases as in GJE0. However, since the connections in a rectangular array have only two directions the third direction is simulated by zig-zagging along the two existing directions. Besides yielding a different sequence of computations (the algorithm is a parallelization of Gauß-Jordan elimination "different" from GJE0) this causes some inconvenience: Each processor goes through four different cycles communicating with its four neighboring processors one at a time.

Hexagonal and rectangular (and also triangular) systolic arrays are computationally equivalent in that each type can simulate the other within a constant factor of time and space, but it seems that the hexagonal array is the most naturally suited structure for a parallelization of the Gauß-Jordan elimination algorithm, at least of the version sGJE of section 3.

The extensive report [Rote 1984] contains more details about the material in this paper, and it includes the following additional features:

The solution to the general Algebraic Path Problem in section 3 (the sGJE algorithm) is formulated differently, using the free semiring generated by the edges of the graph.

It is shown how the data flow in the systolic array GJE0 can be derived from the data flow in the systolic array for the multiplication of two matrices (Kung and Leiserson [1978], Kung [1980]) by successive steps of foldings, simplifications and combinations of systolic arrays.

There is a unidirectional hexagonal version of GJE0 in which the data flow only upwards, i.e. straight upwards or in one of the two diagonal directions. It contains approximately $5n^2$ processors. The main advantage of this array over GJE0 is that the initial and final traveling phases of the matrix in GJE0, where the head or the tail, respectively, of the matrix has to travel from the bottom end to the top end of the array for n steps without any computations going on, are eliminated, and that it therefore takes the new array only $5n-2$ steps to process an $n \times n$ matrix completely instead of $7n-2$ steps.

The drawbacks are that the data flow is not as regular as before, and processor actions are not as uniform, e.g., the type (C) operations are carried out at different places for each diagonal element.

It is also proved that any parallelization of the sGJE algorithm must take at least $5n-3$ steps if each of the n^2 data elements is used only once in each step.

Acknowledgements

I thank Mr. Karel Čulik and especially Mr. Jozef Gruska for helpful discussions through which they let me share some of their knowledge in the field of systolic arrays.

References

Aho, A. V., Hopcroft, J. E., Ullmann, J. D.: The Design and Analysis of Computer Algorithms. Reading (Mass.)-London-Amsterdam: Addison-Wesley Publishing Co. 1975.

Backhouse, R. C., Carré, B. A.: Regular algebra applied to path-finding problems. J. Inst. Math. Appl. *15*, 161–186 (1975).

Brucker, P.: Theorie of Matrix Algorithms. Mathematical Systems in Economics 13. Meisenheim am Glan: Verlag Anton Hain 1974.

Carré, B. A.: An algebra for network routing problems. J. Inst. Math. Appl. *7*, 273–294 (1971).

Carré, B. A.: Graphs and Networks. Oxford: The Clarendon Press, Oxford University Press, 1979.

Floyd, R. N.: Algorithm 97 – shortest path. Comm. ACM *5*, 345 (1962).

Gondran, M., Minoux, M.: Graphes et Algorithmes. Paris: Editions Eyrolles 1979.

Guibas, L. J., Kung, H. T., Thompson, C. D.: Direct VLSI Implementation of Combinatorial Algorithms. Proc. CalTech Conference on Very Large Scale Integration: Architecture, Design, Fabrication. California Institute of Technology, January 1979, Pasadena; Architecture session, pp. 509–525.

Kleene, S. C.: Representation of events in nerve nets and finite automata. In: Shannon, C., McCarthy, J. (eds.), Automata Studies, pp. 3–40. Princeton (New Jersey): Princeton University Press 1956.

Kramer, M. R., van Leeuwen, J.: Systolic computation and VLSI. In: de Bakker, J. W., van Leeuwen, J. (eds.), Foundations of Computer Science IV, part 1, pp. 75–103. Math. Centre Tracts 158. Math. Centre, Amsterdam, 1983.

Kung, H. T.: The structure of parallel algorithms. In: Advances in Computers *19*, pp. 65–112. New York-London-Toronto-Sydney-San Francisco: Academic Press 1980.

Kung, H. T., Leiserson, C. E.: Systolic arrays (for VLSI). In: Duff, I. S., Stewart, G. W. (eds.), Sparse Matrix Proceedings 1978 (Sympos. Sparse Matrix Comput., Knoxville, Tenn., 1978), pp. 256–282. – SIAM, Philadelphia 1979. – (A slightly different version has appeared as section 8.3 of the book: Mead, C. A., Conway, L. A., Introduction to VLSI Systems. Reading (Mass.)-London-Amsterdam: Addison-Wesley Publishing Co. 1979.

Lehmann, D. J.: Algebraic structures for transitive closure. Theoret. Comput. Sci. *4*, 59–76 (1977).

Mahr, B.: Semirings and Transitive Closure. Technische Universität Berlin, Fachbereich 20, report 82-5 (1982).

Mahr, B.: Iteration and summability in semirings. In: Burkard, R. E., Cuninghame-Green, R. A., Zimmermann, U. (eds.), Algebraic and Combinatorial Methods in Operations Research (Proceedings of the Workshop on Algebraic Structures in Operations Research, Bad Honnef, Federal Republic of Germany, April 13–17, 1982), Ann. Discrete Math. *19*, 229–256 (1984).

Rote, G.: A Systolic Array for the Algebraic Path Problem (which Includes the Inverse of a Matrix and Shortest Distances in a Graph). Rechenzentrum Graz, Bericht RZG-101 (1984).

Roy, B.: Transitivité et connexité. C. R. Acad. Sci. Paris Ser. A–B *249*, 216–218 (1959).

Tarjan, R. E.: A unified approach to path problems. J. Assoc. Comput. Mach. *28*, 577–593 (1981).

Warshall, S.: A theorem on Boolean matrices. J. Assoc. Comput. Mach. *9*, 11–12 (1962).

Zimmermann, U.: Linear and combinatorial optimization in ordered algebraic structures. (Especially chapter 8: Algebraic path problems.) Ann. Discrete Math. *10*, 1–380 (1981).

Dipl.-Ing. Günter Rote
Institut für Mathematik
Technische Universität Graz
Kopernikusgasse 24
A-8010 Graz
Austria

Printed in Austria

Druck: Paul Gerin, A-1021 Wien