Manuel d'utilisation de Diastol.Version préliminaire

Patrice Quinton, Pierrick Gachet

▶ **To cite this version:**

Patrice Quinton, Pierrick Gachet. Manuel d'utilisation de Diastol.Version préliminaire. [Rapport de recherche] RT-0041, INRIA. 1984, pp.18. inria-00070117

HAL Id: inria-00070117
https://inria.hal.science/inria-00070117

Submitted on 19 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

IRIA

CENTRE DE RENNES

IRISA

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
tél.(3)954 90 20

Rapports Techniques

N° 41

MANUEL D'UTILISATION DE DIASTOL

Version Préliminaire

Patrice QUINTON
Pierrick GACHET

Octobre 1984

IRISA

INSTITUT DE RECHERCHE EN INFORMATIQUE
ET SYSTÈMES ALÉATOIRES

Campus Universitaire de Beaulieu
Avenue du Général Leclerc
35042 - RENNES CÉDEX
FRANCE
Tél. : (99) 36.20.00
Télex : UNIRISA 95 0473 F

MANUEL D'UTILISATION DE DIASTOL

Version Préliminaire

Publication Interne n° 233
29 pages
Août 1984

Patrice QUINTON
Pierrick GACHET

RESUME : On décrit le système DIASTOL. Ce système permet la conception automatique d'architectures systoliques à partir d'équations. La méthode utilisée consiste à décrire un algorithme comme un système d'équations récurrentes sur un domaine convexe. DIASTOL trouve automatiquement la datation des calculs et détermine un réseau de processeurs les supportant. Des exemples sont donnés et les perspectives d'un tel système sont décrites.

SUMMARY : The system DIASTOL is described. This system allows systolic arrays to be designed automatically from equations. The method used [1] consists in describing an algorithm as a system of recurrent equations over a convex domain. DIASTOL finds automatically the schedule of the equations, and allocates the computations on a network of processors. Examples are provided, and perspectives of such a system are explained.

Remerciements :

Les auteurs remercient le Microelectronics Center of North Carolina et le Departement of Computer Science de North Carolina State University, organismes ayant accueilli Patrice QUINTON au cours d'un séjour sabbatique, d'Octobre 1983 à Juillet 1984, et où une partie de ce travail a été effectuée.

DIASTOL USER'S MANUAL
Preliminary Version

Patrice QUINTON (*)
Pierrick GACHET (+)

July 1984

Summary:

The system DIASTOL is described. This system allows systolic arrays to be designed automatically from equations. The method used [1] consists in describing an algorithm as a system of recurrent equations over a convex domain. DIASTOL finds automatically the schedule of the equations, and allocates the computations on a network of processors. Examples are provided, and perspectives of such a system are explained.

Key-words:
Parallel Computation, Special-Purpose Architectures, Systolic Arrays, Design Methodologies, Convex Analysis.

Acknowledgements:
The authors would like to thank the Microelectronics Center of North Carolina, and the Department of Computer Science of North Carolina State University, where part of this work has been done during the stay of Patrice QUINTON, on leave from IRISA, from October 1983 to July 1984.

(*) Author is on leave from IRISA,Campus de Beaulieu, 35042 RENNES-CedexFRANCE.
(+) IRISA,Campus de Beaulieu, 35042 RENNES-CedexFRANCE.

PAPIER RECUPERE ET RECYCLE

DIASTOL USER'S MANUAL

Patrice QUINTON
Pierrick GACHET

1. INTRODUCTION

Systolic arrays are considered a very promising structure for implementing special-purpose hardware. The regular structure of systolic arrays, the local communications, simple control and multiple use of data make this kind of device particularly attractive for VLSI implementation. We describe a program named **DIASTOL** which allows one to design systolic arrays from so-called Uniform Recurrent Equations. The theory underlying this program is described in [1]. In this paper, we focus on the practical aspects of the method, and show on various examples how DIASTOL can be used.

2. LEARNING BY EXAMPLE: THE CONVOLUTION PRODUCT

Given a sequence $x(0), x(1), \ldots, x(i), \ldots$ and coefficients $w(0), w(1), \ldots, w(K)$, the convolution algorithm consists in computing the sequence $y(0), y(1), \ldots, y(i), \ldots$ where $y(i)$ is given by the following equation

$$y(i) = \sum_{k=0}^{K} w(k)\, x(i-k) \qquad (1)$$

We are interested here in finding systolic arrays that compute (1). The process of finding systolic arrays go through several steps that will be exemplified on the convolution product example. The first step, which unfortunately must be done by hand right now, consists in transforming the equations of the problem to be solved in such a way that they become a uniform recurrent system of equations. For some problems (like the convolution, or the matrix product), this is not very difficult. For other, in particular problems involving recursive computations, this may be very tricky. The second step consists in finding a timing-function for the uniform recurrent system. This is done by DIASTOL, but some explanations about this step are needed in order to understand what is going on. Finally, the last step consists in allocating the computations on a finite, regular array of processors. Starting from the example of the convolution product, we will in the following examine these three steps in turn.

2.1. Transforming the equations

A first transformation of (1) consists in expanding the Σ operator in such a way that only remain elementary calculations. These elementary calculations actually define the structure of the elementary cells of the systolic array. Equation (1) may be rewritten as:

$$\forall i, 0 \leqslant i\ ;\ \forall k, 0 \leqslant k \leqslant K:\ y(i,k) = y(i,k-1) + w(k)\, x(i-k) \qquad (2)$$

$$Vi: 0 \leqslant i, \quad y(i,-1) = 0$$

where $y(i,k)$ are partial accumulated values for $y(i)$. The basic idea of DIASTOL is to consider these computations to be associated with points (i,k) of the plane, and more generally, to integer coordinate points of the Euclidean space. For any integer coordinate point (i,k) lying in the domain $D = \{ 0 \leqslant i ; 0 \leqslant k \leqslant K \}$ we have to perform the elementary computation $y_{out} = y_{in} + w_{in}x_{in}$, the result of which will be $y(i,k)$ provided y_{in}, w_{in}, and x_{in} are given correct values $y(i,k-1)$, $x(i-k)$ and $w(k)$.

Before these equations can be entered in DIASTOL, it is necessary that every variable in (2) appears with all the indexes. We can see that w appears only with index k, and x only with index $i-k$. There are generally several ways to do that. Intuitively, one can find a transformation in the following way.

Consider first $w(k)$. We can see that for all the points (i,k) such that k is constant, equation (2) involves the same value $w(k)$. One way to make i appear is to suppose that the coefficient $w(k)$ which is used at point (i,k) is those that was previously used at point $(i-1,k)$. Provided that point $(0,k)$ uses the actual coefficient $w(k)$, these scheme will work. As far as x is concerned, we can see that since this variable appears as $x(i-k)$, all points (i,k) such that $i-k$ is constant have the same variable x. Thus, one way to restore all the indexes in x is to suppose that $x(i-k)$ at point (i,k) comes from point $(i-1,j-1)$. Let us denote as $W(i,k)$ the new variable w, and $X(i,k)$ the new variable x. Equation (2) may be replaced by:

$$\begin{aligned} &Vi: 0 \leqslant i \; ; \; Vk: 0 \leqslant k \leqslant K \\ &y(i,k) = y(i,k-1) + W(i-1,k)\, X(i-1,k-1) \\ &W(i,k) = W(i-1,k) \\ &X(i,k) = X(i-1,k-1) \end{aligned} \tag{3}$$

with the following initial conditions:

$$Vi, 0 \leqslant i; Vk, 0 \leqslant k \leqslant K:$$

$$y(i,-1) = 0; \quad W(-1,k) = w(k); \quad X(i-1,-1) = x(i); \quad X(-1,k-1) = 0$$

Such a system of recurrent equations is said to be uniform, since computation at point (i,k) depends only on values computed at points that are obtained by a translation which does not depend on i or k [1]. This system may be represented by a graph such as that of Fig. 1. The nodes of this graph represent the computations to be achieved and the edges represent values that are to be transmitted from one node to another.

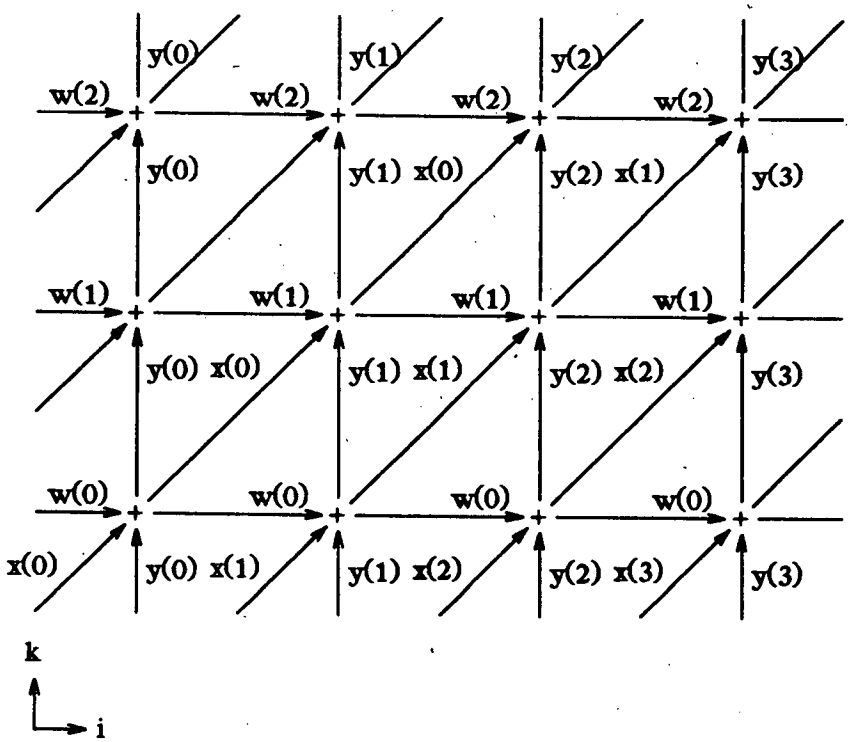

Fig.1: Dependence graph for the convolution product (K=2).

In (3), there are two different ingredients. The first one is the **domain of computation,** that is the set of indexes for which the equations are defined, and the second one is the **equations.** DIASTOL provides separate processors for the equations and the domains.

2.2. Domain of computation

A domain is described by giving its dimension (2 or 3), then by defining the names of the indexes (here, i and k) and finally by entering linear inequalities (called **constraints**) defining the domain. Each inequality has the form:

$$a_1 i + a_2 k \geqslant c$$

The coefficients of the inequalities should be rationnal numbers, as well as the constant. The exact syntax of the constraints is given later on. In the case of the convolution product the constraints would be (assuming K =2):

$$\begin{aligned} i &\geqslant 0 \\ k &\leqslant 2 \\ k &\geqslant 0 \end{aligned} \qquad (4)$$

Before going on, let us give a few definitions on convex domains. A set defined by a finite set of inequalities such as (4) is called a **Convex Polyhedral Domain** (CPD). A CPD has **vertices** and **rays.** A vertex is an extremal point of the domain. For example, points (0,0) and (0,2) are the vertices of the domain defined by equations (4) (see Fig. 1). A ray is a vector defining a direction in which the domain is open. For example, vector (1,0) is a ray of the domain of computation for the convolution product. Rays are defined up to a positive scalar multiple, since if r is a ray, so is of course μr, for every $\mu>0$. Extremal rays are rays that cannot be expressed as positive combination of other rays. Although a CPD may have an infinite number of rays, it has finitely many extremal rays. Extremal rays and vertices are sufficient to characterize a CPD. There exist algorithms that allow rays and vertices to be found from the inequalities.

The domains of computation that can be handled by DIASTOL should have at most one ray. The reason is simple: if there are more than one ray, the domain cannot be projected on a finite machine.

2.3. Equations

A system of equations is described, again by giving its dimension (2 or 3), then defining variable names (here, y, x and w), and then by entering the equations. For example, equations (3) are entered in the following way:

$$
\begin{aligned}
y &= (add\ y.<0\ -1>\ (mult\ w.<-1\ 0>\ x.<-1\ -1>)) \\
x &= x.<-1\ -1> \\
w &= w.<-1\ 0>
\end{aligned}
\tag{5}
$$

Implicitely, the equations are meant to take place at any point (i,k). Thus, y means actually $y(i,k)$. To denote a variable at another point, one uses the notation $V.<v>$ where V is a variable name, and the brackets contain a constant translation vector v. This notation represents $V((i,k)+v)$, where + represents the sum of vectors. For example, $y.<0\ -1>$ denotes the variable $y(i,k-1)$. Finally, the syntax of the equations is "functional". The right-hand side of an equation is either a variable, or an expression of the form (op arg1 arg2 ... argn), where op is an operator (add, mult, sub, etc...), and arg1, arg2, ... argn are arguments, which in turn can be complex expressions.

Also given with the equations is the assumed duration of calculation of each equation. By default, it is assumed to be one unit of time.

2.4. Timing-functions

From a domain such as (4) and equations such as (5), one can build a systolic array as we shall see. The next step consists in finding a schedule for the computations, called a **timing-function.** A timing-function is a integer mapping $t(i,k)$ which gives the time at which the computation associated with point (i,k) can occur. In DIASTOL, we consider timing-functions that are of the form:

$$t(i,k) = \left\lfloor \lambda_1 i + \lambda_2 k - \alpha \right\rfloor$$

where $\lfloor x \rfloor$ is the floor function, i.e. the greatest integer smaller than or equal to x.

In [1], it is explained formally how one can find λ_1, λ_2, and α. Intuitively, there are two kinds of constraints put on these values. The first kind of constraints has to deal with the equations. If we want to be able to evaluate computation associated with point (i,k), it is necessary that all the computations involving input arguments of the equation be already done. Consider equation (3). Computation at point (i,k) depends on results of computations at points $(i-1,k)$, $(i,k-1)$, and $(i-1,k-1)$. Assuming that the computation associated with each point lasts one unit af time, we must have:

$$
\begin{aligned}
t(i,k) - t(i-1,k) &\geqslant 1 \\
t(i,k) - t(i,k-1) &\geqslant 1 \\
t(i,k) - t(i-1,k-1) &\geqslant 1
\end{aligned}
\tag{6}
$$

Forgetting about the floor function for the sake of simplicity (the general case is a little more difficult to explain), (6) reduces to:

$$\lambda_1 \geqslant 1$$

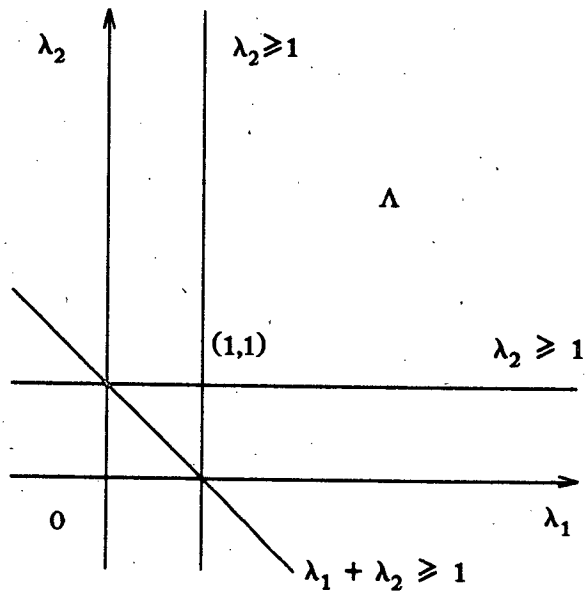

Fig. 2: The domain Λ for the convolution product.

$$\lambda_2 \geq 1$$
$$\lambda_1 + \lambda_2 \geq 1 \qquad (7)$$

The second type of constraints on the timing-function is related to the domain. If we want to be able to implement the problem on a real machine, it is necessary that the computation starts once, i.e, that $t(i,k)$ have a minimum value over the domain of computation, and also that there is a bounded number of points to be computed simultaneously. A condition upon which these properties are satisfied is that the timing-function is strictly increasing along the ray of the computation domain, whenever there is one. As a matter of fact, if the timing-function is constant along some ray, there are infinitely many points to be computed simultaneously, lying on the half-lines defined by the ray and contained in the domain. On the other hand, if the timing-function is decreasing along the ray, then $t(i,k)$ cannot have a minimum value on the computation domain. Let r denote the unique ray of the domain. The above condition can be expressed by saying that the dot product of (λ_1,λ_2) and r should be strictly positive. For our example, this gives the constraint:

$$\lambda_1 > 0 \qquad (8)$$

which turns out to be redundant with (7).

The last (arbitrary) constraint that we add is that $t(i,k)$ must be non-negative over the domain of computation. This constraint is met if t is non-negative at the vertices of the domain. For our example, we get, since the domain has two vertices (0,0) and (0,2):

$$t(0,0) = -\alpha \geq 0$$
$$t(0,2) = 2\lambda_2 - \alpha \geq 0 \qquad (9)$$

It turns out that equations (7) and (8) define a CPD where the coefficients (λ_1,λ_2) considered as a point should lie. This domain, called Λ (or the Lambda-domain) is shown on Fig. 2. Before explaining (still intuitively) where (λ_1,λ_2) should be taken, let us note that coefficient α is determined by (9) as soon as $\lambda 1$ and $\lambda 2$ are chosen.

Although any point in Λ gives a correct timing-function, there are obviously solutions that are better than others. For example, if λ_1 and λ_2 are correct coefficients, so are also $\mu\lambda_1$ and $\mu\lambda_2$ for any scalar $\mu > 1$. But obviously, the timing-function which result is slower. In fact, optimizing the choice consists in trying to pick up a point on the boundary of Λ, as in a conventional linear programming problem. Consider first of all the boundaries corresponding to the constraints (7) derived from the dependences in the equations. As an example, the smaller the quantity $\lambda_2 - 1$, the faster the timing function will be along the direction defined by the y dependency (see Fig. 1), and therefore, the smaller the latency between the beginning and the end of the computation of y. Optimizing the timing function along the other dependency direction is of no use here, since only the results y of the computations are interesting. On the other hand, consider the constraint given by (8). Trying to minimize the quantity λ_1 corresponds to maximizing the througput of a (possible) machine that will support the computations. The reason is that λ_1 is the dot product of the

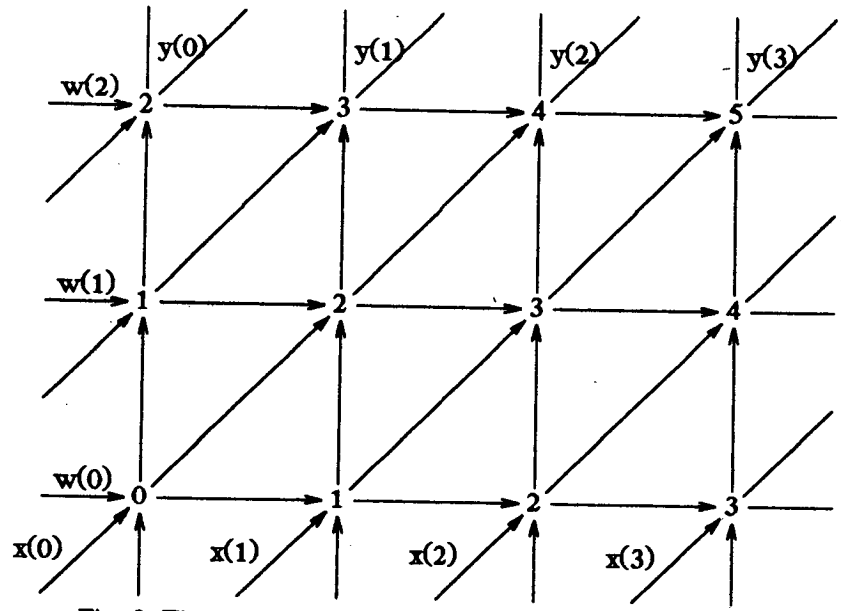

Fig. 3: Timing function for the convolution product (K=2).

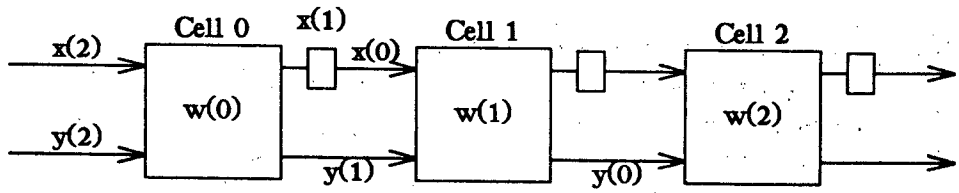

Fig. 4: Systolic array for the convolution product.

vector (λ_1,λ_2) and the ray of the domain, which is proportionnal to the inverse of the speed of the timing function allong the ray. It turns out that the two functions that we want to optimize reach a minimum on Λ, and this minimum is attained at a vertex of the domain. But since Λ has only one vertex, which is the point (1,1) (see Fig. 2), it is the best solution we can find. As a result, we obtain the timing-function:

$$t(i,k) = i + k \tag{10}$$

α being determined to be 0 by (9). Fig. 3 shows the resulting schedule of the equations.

2.5. Allocation function

Once we have the schedule of the computations, it remains to map these computations on a finite machine, in a "systolic" way. A very convenient way to do so is to project the domain of computations along a direction defined by some conveniently chosen vector u. Each point of the resulting projected domain will represent a processor of the systolic architecture. Before giving more details about this step, consider again the convolution product example. A convenient way to project the domain (see Fig. 1), is to project it along the i-axis. In this way, all the points lying on lines parallel to the i-axis will be computed by the same processor. Since these points are computed at different times according to the timing-function we have chosen, a processor will never have more than a computation to do at a given time. Fig. 4 shows the resulting architecture, which is a well-known design for convolution [2]. To see how this architecture can be derived from the system, the timing-function, and the mapping, one can look at Fig. 3. Since the domain of computation has three lines parallel to the i-axis, there are only three processors. Let us call cell K the processor which takes care of the computations associated with points lying on line $k = K$. The data movement between the cells result from the data dependences shown on Fig. 3. Coefficients w stay on each processor. Values y and x go from processor 0 to processor 1 to processor 2. Note also that the y's move twice as fast as the x's. This appears clearly by examining Fig. 1, since the x's move diagonally and therefore, reach a processor (i.e., a line $k=K$) every other time.

Let us come back to the way the mapping is defined. We denote as $a(i,k)$ the function that defines on which processor we want to execute the computation associated with point (i,k). This function, that we want to determine, is called the **allocation function.** Let $u = (u_1,u_2)$. Since we want a to be a projection along vector u, two points (i_1,k_1) and (i_2,k_2) will be on the same processor if there exist a constant l such that:

$$i_2 = i_1 + l\ u_1$$

$$k_2 = k_1 + l\ u_2$$

The choice of u is constrained by two conditions. The first one is that there are never on the same processor two points that must be computed at the same time. Since the timing-function is linear, (or more exactly affine), equation $t(i,k)$ = constant represent parallel lines of the plane (still forgetting the floor function). Therefore, this condition is obviously met if u is not parallel to these lines. Recalling that (λ_1,λ_2) is the normal vector to these lines, it is

sufficient that the dot product of (λ_1,λ_2) and u is not null. The second condition to be satisfied by u is that the domain resulting from the projection be bounded. There is of course no problem when the domain of computation is itself bounded, i.e. has no ray. When the domain is not bounded, it has at most one ray. The only way to project the computation domain so that the result is a bounded domain is to take u parallel to the ray of the domain, as it is the case for the convolution product (*). The way DIASTOL defines the allocation function is as follows. If the domain of computation has one ray, u is taken along this vector. Otherwise, DIASTOL asks you for a projection direction u. It checks that this vector is not parallel to the timing-lines, and if not, computes the allocation function.

3. USING DIASTOL

The following section is devoted to the organization and use of the DIASTOL system. DIASTOL is a fully interactive program consisting of two levels. DIASTOL keeps tracks of three types of objects: domains, equations, and solutions. Domains and equations have already been described. Each domain and equation has a name. Lambda-domains, that is domains built for the timing-function (see 2.4), are particular domains whose name starts whith the character 'Z'. Projected domains, as obtained by the allocation function, have a name starting with 'ZZ'. A solution is a (possibly partially) elaborated systolic array. A solution consists of a reference to a domain of computation (called D), a reference to a system of equations, a reference to a lambda-domain (called L), a timing-function, an allocation function, and a reference to a projected domain (called A).

DIASTOL is organized in two levels. At the first level, commands allows housekeeping of domains, equations, and solutions to be done. At the second level, there are the processors for the domains, the equations, and the solutions. The following describes the various command available under DIASTOL. The explanations are almost identical as the on-line documentation of the system.

3.1. First Level of Diastol

The following summarizes the commands of the first level of diastol, and describes in more detail each command.

3.1.1. List of DIASTOL first level commands

ed {<nomdomaine>}.....enter domain editor
ee {<nomdomaine>}.....enter (system of) equations editor
dd <name>.....delete a domain
dad .....delete all domains
ds <solution>.....delete a solution
das .....delete all solutions
de <name>.....delete a (system of) equations
dae .....delete all equations
pd <nomdomaine>.....print a domain
ps <nomsolution>.....print a solution

(*) In [1], we present a generalization to the case when u is not parallel to the ray, but this has not yet been

pe <nomsysteme> .....print a (system of) equations
ld .....list domain names
ls .....list solution names
le .....list system of equations names
sys .....build a systolic array
save .....save solutions
load .....load solutions
? (or help).....this command
help <command name> .....explain command
q.....quit
list .....turn on listing
nolist .....turn off listing

3.1.2. Detail of the commands.

ed: edit domain.

The ed command enters the domain editor. When you enter the domain editor, you are under a new environment in which you can define convex domains for new algorithms or modify a previously defined domain. Syntax of this command is: ed {domain name}. Section 3.3 describes the domain editor.

ee: enter (system of) equations editor.

The ee command enters the equation editor. When you enter the equation editor, you are under a new environment in which you can define equations for new algorithms or modify previously defined equations. Syntax of this command is: ee <equations name> Section 3.2 describes the equation editor.

dd: delete a domain.

The dd command deletes a domain which has been previously defined. Syntax is dd <name> where name is a domain name. To know the names of the domains you have defined, use the ld command.

dad: delete all domains.

The dad command deletes all domains which has been previously defined. Since the solutions, domains and equations are saved only at the end of a session (unless the command save was used), it is possible to recover the deleted domains by loading them again (using the command load).

ds: delete a solution.

.....

implemented in DIASTOL.

The command ds deletes a solution which has been previously defined. Syntax is ds <name> where name is a solution name. To know the names of the solutions you have defined, use the ls command. A new solution is created each time you use the sys command.

das: delete all solutions.

The das command deletes all solutions which has been previously defined. Since the solutions, domains and equations are saved only at the end of a session (unless the command save has been used), it is possible to recover the deleted solutions by loading them again (using the command load).

de: delete a (system of) equations.

The de command deletes equations which have been previously defined. Syntax is de <equations name>. To know the names of the equations you have defined, use the ld command.

dae: delete all (systems of) equations.

The dae command deletes all previously defined equations. Since the solutions, domains and equations are saved only at the end of a session (unless the command save has been used), it is possible to recover the deleted equations by loading them again (using the command load).

pd: print a domain.

The command pd lists the content of a domain. Syntax of this command is: pd <domain> where <domain> is the name of a previously defined domain. As a result, pd gives the constraints that define <domain>, the vertices and rays of the domain.

ps: print a solution.

ps prints the content of a solution. A solution consists of a domain, a set of dependence vectors, a domain LAMBDA, a timing function, an allocation function, and a projected domain. Syntax of this command is ps <solution>.

pe: print a (system of) equations.

The command pe lists the content of a system of equations. Syntax of this command is: pe <system> where <system> is the name of a previously defined system of equations.

ld: list domain.

The ld command gives you the list of the domains that have been previously defined.

ls: list solutions.

ls prints the names of the solutions that have previously been elaborated.

le: list equations.

le prints the names of the systems of equations that have previously been elaborated.

sys: build a Systolic Array.

This command allows you to define a systolic array, under the control of DIASTOL, by entering the so-called solution processor. Before you enter this environment, you must have defined a system of equations (using the command ee), and a domain of computation (using the command ed). Section 3.4 describes the solution processor.

save: save solutions.

The save command saves the domains and solutions as they are currently. Note that a save command is issued automatically when you quit (under normal conditions) Diastol.

load: load solutions.

The load command loads the domains and solutions from the saving file 'diastolsauve'. Unless you have saved your solutions during the session using the **save** command, load will restore the solutions as they were at the beginning of the session.

help: command help.

The help command gives you information about the various command you can enter from the first level of Diastol. Syntax is: help <command name>

list: turn on listing option.

list records in the file diastollisting the answers that are given by DIASTOL to the print commands issued at the first level of DIASTOL. The nolist command turns off the listing option.

q: quit.

The q command makes you quit the Diastol system. All the domains and solutions you have defined during the session are saved automatically.

3.2. The System Editor

The system editor makes it possible to define, or modify a system of equations. It may be called from the first level of diastol.

3.2.1. Summary of the commands

a.....enter add mode
l <system name>.....load working system

d <equation number>.....delete equation
dim <dimension>.....set system dimension (2 or 3)
d2/d3.....equivalent to dim 2 or dim 3
p.....print working system
cv <index> <string>.....add or replace variable
pv <index> <string>.....print variables
s <system name>.....save working system
? or help.....this command
help <command>.....explain command
q.....quit

3.2.2. Detail of the commands

a: enter add mode.

This command puts you in add mode. Everything you type is understood to be an equation defining a system of equations. To quit this mode, type 'q' or '.'. The equations you enter are added to the previously entered equations. The syntax of an equation may be explained using the following example:

y = (add y.<-1 0> x.<-1 -1>)

The left-hand side is the name of a so-called 'variable', which must have been declared using the command 'cv'. Implicitely, y stands for y(z), which means that this equation will define the way you compute y for all the points of the domain you will later on associate with the system of equations.

The right-hand side defines the value to be assigned to y. The syntax of the right member of an equation follows more or less a "functional" notation (OP arg1 ... argp), where OP is one of the predefined operators, and arg1 ... argp are either variables or, recursively, functional expressions. Here, the example says that we have to add the value of y taken at point (z - (-1,0)) and the value of x taken at point (z - (-1,-1)). x.<-1 -1> stands for the value obtained at point z + (-1,-1). Currently, only the operators add, mult and minus are accepted. This has however no importance, since no semantic is associated with these operators in the current state of DIASTOL.

l: load a system of equations.

This loads a system of equations as a current working system. The syntax is: l <system name>.

d: delete an equation.

This command allows you to delete an equation. The syntax is: d <equation number>. In order to know the numbers of the equations, use the command p (print).

dim: set dimension.

Sets the dimension of the system (currently, only 2 and 3 are possible). Syntax is: dim <number>. Equivalent commands are d2 and d3.

d2: dimension 2.

Sets the dimension of the current system to 2. Equivalent to dim 2.

d3: dimension 3.

Sets the dimension of the current system to 3. Equivalent to dim 3.

p: print current system.

This command prints the content of the working system.

cv: define or change a variable name.

This command defines a new variable, or changes the name of a variable whenever it exists. The syntax is **cv** n <string> where n is the number of a variable, and string is its name (8 characters at most). Notice that it is perfectly possible to modify the names of the variables after equations have been entered. Since the variables in the equations are recorded by their number, the names will be changed accordingly. To list the current variables, use pv.

pv: print variables.

This command print the currently defined variables.

s: save (system of) equations.

Saves the working equations. Syntax is s {<equation name>}. If no name is given, DIASTOL assumes that the equation will be saved in the loaded system, if the equation editor has been entered via a command ee <name>, or if the l (load) command has been issued. This means that you can quit the system editor without destroying your most recent work... However, note that the system is not definitively saved, since DIASTOL may crash... If you want to be sure that it is saved, use the command save at the first level of DIASTOL.

3.3. The domain editor

3.3.1. Summary of the commands

a.....enter add mode
l <domain name>.....load working domain
d <constraint number>.....delete constraint
dim <dimension>.....set domain dimension (2 or 3)
d2/d3.....equivalent to dim 2 or dim 3
p.....print working domain
ci <index> <string>.....modify index
pi.....print indexes
s <domain name>.....save working domain
? or help.....this command
help <command>.....explain command
q.....quit

3.3.2. Detail of the commands

Only the commands which have not been described in the previous section are detailed.

a: enter add mode.

This command puts you in add mode. Everything you type is understood to be a constraint defining the domain. To quit this mode, type 'q' or '.'. The constraints you enter are added to the previously entered constraints. The syntax of a constraint is fairly natural. For example: i + 2/3 k <= - 5/4 is a constraint. The inegality operators recognized are <, >, <= and >=. To add an egality, add twice the constraint with opposite operators The indexes of the constraints must match the indexes defined previously. Default index names are 'x1', 'x2' and 'x3'. You may change them by using the command 'cv'. The coefficients must be signed rational numbers. Signed integers are accepted.

ci: change index name.

ci allows you to change the name of an index. Default values are 'x1', 'x2', and 'x3'. The syntax is ci n <string> where n is the number of the index and string the new name. To list the current index names, use pi. Note that you can change the index name without modifying the inequalities defining the domain.

d: delete constraint.

This command allows you to delete a constraint. Syntax is d <constraint number> In order to know the numbers of the constraints, use the command p (print).

l: load domain.

This command makes it possible to load a domain as a current working domain. The syntax is: l <domain name>.

p: print current domain.

This command prints the content of the working domain.

pi: print index names.

this command print the current index names.

s: save domain.

Saves the working domain. Works as the s command of the equation editor.

3.4. The solution processor

3.4.1. Summary of the commands

dD.....define computation domain

pD.....print computation domain
de.....define system of equations
pe.....print system of equations
dt.....define timing-function
pt.....print timing-function
da.....define allocation function
pa.....print allocation function
pL....print Lambda domain
pA....print projected domain
l <solution>.....load a solution
q.....quit
?.....this command
help x...explains command x

3.4.2. Detail of the commands

dD: define domain of computation.

This command allows you to choose a domain of computation you have previously entered using the domain editor (see the command ed, on the first level of DIASTOL). Syntax is dD <domain name>. When you define the domain, and if the solution you are building has not yet received a name, DIASTOL will ask you for the name you want to give to it.

da: define allocation function.

This command allows you to define the (or one possible) allocation function for your problem. If the domain D, or the system, or the timing function have not yet been defined, Diastol will tell you. Otherwise, two cases are possible (at least, in the current version):
- if the domain D has a ray (and it must have only one ray), then the projection direction chosen by DIASTOL is this ray. Note that a future incarnation of the system will give you the flexibility to choose any projection direction you want.
- if the domain D has no ray, then DIASTOL will ask you which direction you want. Answer with a 2- or 3- dimensional vector (depending on the dimension of your problem). For example, -1 2 1 .

de: define (system of) equations.

This command allows you to choose a system of equations you have previously entered using the system editor (see the command ee, at the first level of DIASTOL). Syntax is de <equation name>. The computation domain which corresponds to the system of equations should have been defined before, using the command dD.

dt: define a timing function.

This command allows you to define the (or one possible) timing-function for your problem. If the domain D, or the system have not been defined, Diastol will tell you. Otherwise, DIASTOL will try to compute a timing-function. This will be done by building

another convex domain called the LAMBDA domain, where the vector defined by the coefficients of t must lie. Several cases may arise:
- the domain LAMBDA is void or has no vertex: this means that your system cannot be solved by any explicit calculation. There is probably a mistake in your system... (the most common being that the dimension of the domain D is incorrectly specified, which results in fancy vertices and rays either for the domain D and for the domain LAMBDA).
- LAMBDA has only one vertex: the timing-function is automatically computed and is garanteed to be througput and pipeline optimal;
- LAMBDA has several vertices: DIASTOL asks you which one you want. The choice depends on which constraint you want to optimize. Refer to section 2.4 for the choice.

l: load a solution.

This command loads a solution as a current working solution. The syntax is: ls <solution name>.

pA: print projected domain a(D).

This command prints the domain a(D) (also called the projected domain) of your solution, i.e the final convex domain defining your systolic architecture. To each point of this domain, is attached one processor of this architecture.

pD: print domain of computation.

This command prints the Domain of computation D of your solution. It gives you the inequalities defining the domain, the dimension (2 or 3), the vertices and rays of D. Note that only domains having zero or one ray can be processed to become systolic solutions. This is not checked by the domain editor, but will be checked later on when you'll try to built an allocation function.

pL: print domain LAMBDA.

This command prints the Domain LAMBDA of your solution. It gives you the inequalities defining the domain, the dimension (2 or 3), the vertices and rays of LAMBDA. The domain LAMBDA is defined automatically by the system, when it computes a timing-function. It is the convex domain in which the coefficients of the timing-function must lie. The name of the domain LAMBDA is built automatically by adding a Z character to the name of the corresponding domain D.

pa: print the allocation function.

This command prints the allocation-function of your solution

pe: print system of equations.

```
SOLUTION : conv

 COMPUTATION DOMAIN:conv
   NAME: conv DIMENSION : 2
   CONSTRAINT(S) :
     C1 : - i <= 0
     C2 :   k <= 4
     C3 : - k <= 0
   VERTICES :
     S1  (  0 ,  4 ,  0) SATURATES :  C1 C2
     S2  (  0 ,  0 ,  0) SATURATES :  C1 C3
   RAY(S) :
     R1  (  1 ,  0 ,  0) SATURATES :  C2 C3

 SYSTEME:conv
   NAME: conv DIMENSION : 2
   VARIABLE(S) :
     VAR1 y
     VAR2 x
     VAR3 w
   DEPENDENCE VECTOR(S):
     DV1 : ( 0 -1 )  1 refs
     DV2 : ( -1 -1 )  2 refs
     DV3 : ( -1 0 )  2 refs
    3 EQUATION(S) :
     E1 : y =  ( add y.< 0 -1> ( mult x.< -1 -1> w.< -1 0>) )
     E2 : x =  x.< -1 -1>
     E3 : w =  w.< -1 0>

 TIMING FUNCTION: t(i,k)=     1 i   + 1 k

 ALLOCATION FUNCTION
     a1(z) =  0
     a2(z) =  + k

 PROJECTED DOMAIN :ZZconv
   NAME: ZZconv DIMENSION : 2
   CONSTRAINT(S) :
     C1 :   k <= 4
     C2 : - k <= 0
     C3 : - u <= 0
     C4 :   u <= 0
   VERTICES :
     S1  (  0 ,  4 ,  0) SATURATES :  C1 C3 C4
     S2  (  0 ,  0 ,  0) SATURATES :  C2 C3 C4
   NO RAY
```

Fig. 5: Printout of the solution corresponding to Fig. 4.

```
SOLUTION : conv1

  COMPUTATION DOMAIN:conv
    NAME: conv DIMENSION : 2
    CONSTRAINT(S) :
      C1 : - i <=  0
      C2 :   k <=  4
      C3 : - k <=  0
    VERTICES :
      S1  (  0 ,  4 ,  0) SATURATES :  C1 C2
      S2  (  0 ,  0 ,  0) SATURATES :  C1 C3
    RAY(S) :
      R1  (  1 ,  0 ,  0) SATURATES :  C2 C3

  SYSTEME:conv
    NAME: conv DIMENSION : 2
    VARIABLE(S) :
      VAR1 y
      VAR2 x
      VAR3 w
    DEPENDENCE VECTOR(S):
      DV1 : ( 0 -1 )  1 refs
      DV2 : ( -1 -1 )  2 refs
      DV3 : ( -2 0 )  2 refs
     3 EQUATION(S) :
      E1 : y =  ( add y.< 0 -1> ( mult x.< -1 -1> w.< -2 0>) )
      E2 : x =  x.< -1 -1>
      E3 : w =  w.< -2 0>

  TIMING FUNCTION: t(i,k)=     1/2 i   + 1 k

  ALLOCATION FUNCTION
      a1(z) =  + i  MOD   2
      a2(z) =  + k

  PROJECTED DOMAIN :ZZconv1
    NAME: ZZconv1 DIMENSION : 2
    CONSTRAINT(S) :
      C1 :   k <=  4
      C2 : - k <=  0
      C3 : - u <=  0
      C4 :   u <=  1
    VERTICES :
      S1  (  0 ,  4 ,  0) SATURATES :  C1 C3
      S2  (  1 ,  4 ,  0) SATURATES :  C1 C4
      S3  (  0 ,  0 ,  0) SATURATES :  C2 C3
      S4  (  1 ,  0 ,  0) SATURATES :  C2 C4
    NO RAY
```

Fig. 6: Printout of the systolic convolution, when w(k) "jumps". This solution corresponds to the block–convolver.

This command prints the system of equations of your solution.

pt: print timing function.

This command prints the timing-function of your solution

4. LEARNING MORE, STILL BY EXAMPLE

In this section, we describe a few examples that have actually been produced by DIASTOL. These examples allow other features of DIASTOL to be explained.

4.1. Interpreting the results

Fig. 5 is a printout of the convolution example as processed by DIASTOL, and printed using the command 'ls' at the first level. From top to bottom, we have:
- the computation domain, called conv. Here, K has been taken to be 4. The vertices and rays of this domain appear with their third coordinate, although not significant.
- the system of equations. The dependency vectors are also given, together with the number of their references in the equations.
- the timing-function
- the allocation function
- finally, the projected domain.

Missing are the connectivity and the timing of the resulting systolic array. By hand, it can be obtained in the following way (next version will include this feature).

The systolic array has one processing cell for each integer coordinate point of the projected domain. Here, processors are thus denoted as $P_{0,0}$ through $P_{0,4}$. The connectivity of the array results of projecting the dependency vectors. Consider processor P_π (where π is (0,0) to (0,4)). Let $V.<d>$ be a dependency that occurs in an equation. Then, there exists a link from $P_{\pi + a(d)}$ to P_π, through which variables $V((i,k) + d)$ are transmitted during the computation. The delay of this link, that is, the time the variables have to wait between the two processors is $\Delta = | t(d) |$. Finally, processor P_π has to evaluate all the computations associated to points (i,k) such that $a(i,k) = \pi$.

In our example, processor $P_{0,k}$ receives y from processor $P_{0,k-1}$ with delay 1, since $y.<0-1>$ appears in the equations, and $a(0,-1) = (0,-1)$. Processor $P_{0,k}$ receives also x from $P_{0,k-1}$, since $a(-1,-1) = (0,-1)$. However, the delay is 2, since $|t(-1,-1)| = 2$. Finally, w stays in the processors, since $a(-1,0) = (0,0)$. We thus find the systolic array given by Fig. 4 (for the case where $K = 2$).

4.2. The block convolver

Fig. 6 gives a printout of another solution for the convolution example. The domain is the same as in the previous solution. Only the equation defining the w-dependence is different. Here, w is supposed to 'jump' points when moving along lines parallel to the i-axis. Fig. 7a illustrates the equations. The timing-function which results is given by:

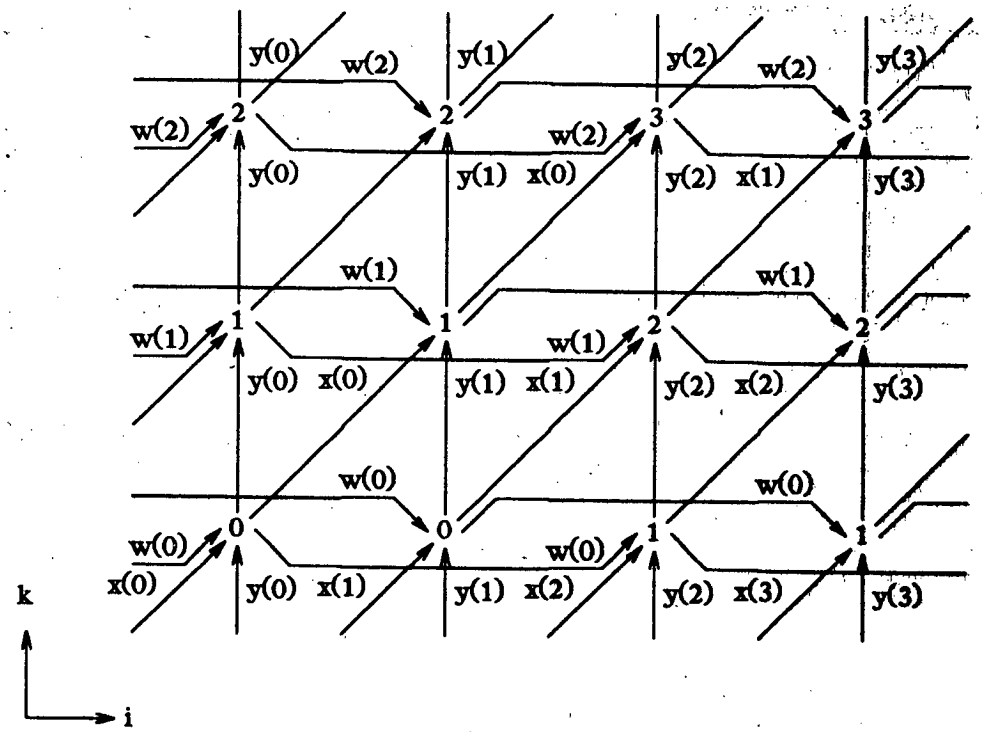

Fig. 7a: Another dependance graph for the convolution product (K=2).
The timing function is $t(i,k) = \left\lfloor \frac{i}{2} + k \right\rfloor$.

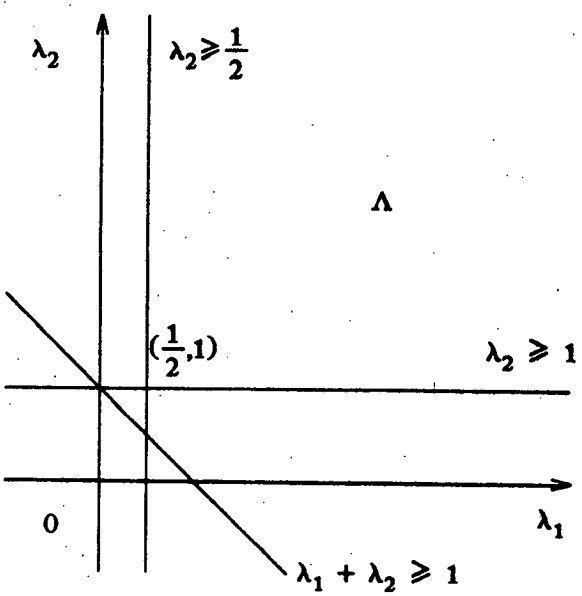

Fig. 7b: The domain Λ
for the URE of Fig. 5a.

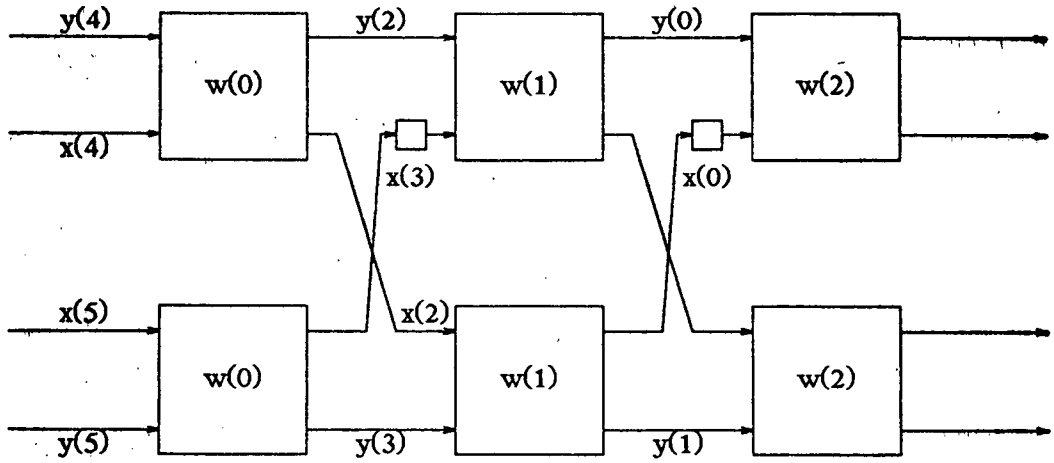

Fig 8: Block-convolver.

```
SOLUTION : mult

  COMPUTATION DOMAIN:mult
    NAME: mult DIMENSION : 3
    CONSTRAINT(S) :
      C1 : - i <=  -1
      C2 :   i <=  2
      C3 : - j <=  -1
      C4 :   j <=  2
      C5 : - k <=  -1
      C6 :   k <=  2
    VERTICES :
      S1  (  1 ,  1 ,  1) SATURATES :  C1 C3 C5
      S2  (  1 ,  1 ,  2) SATURATES :  C1 C3 C6
      S3  (  1 ,  2 ,  1) SATURATES :  C1 C4 C5
      S4  (  1 ,  2 ,  2) SATURATES :  C1 C4 C6
      S5  (  2 ,  1 ,  1) SATURATES :  C2 C3 C5
      S6  (  2 ,  1 ,  2) SATURATES :  C2 C3 C6
      S7  (  2 ,  2 ,  1) SATURATES :  C2 C4 C5
      S8  (  2 ,  2 ,  2) SATURATES :  C2 C4 C6
    NO RAY

  SYSTEME:
    NAME:  DIMENSION : 3
    VARIABLE(S) :
      VAR1 a
      VAR2 b
      VAR3 c
    DEPENDENCE VECTOR(S):
      DV1 : ( 0 0 -1 )  1 refs
      DV2 : ( 0 -1 0 )  2 refs
      DV3 : ( -1 0 0 )  2 refs
     3 EQUATION(S) :
      E1 : c =  ( add c.< 0 0 -1> ( mult a.< 0 -1 0> b.< -1 0 0>) )
      E2 : a =  a.< 0 -1 0>
      E3 : b =  b.< -1 0 0>

  TIMING FUNCTION: t(i,j,k)=     1 i   + 1 j   + 1 k    -3

  ALLOCATION FUNCTION
      a1(z) =  + i
      a2(z) =  + j
      a3(z) =  0

  PROJECTED DOMAIN :ZZmult
    NAME: ZZmult DIMENSION : 3
    CONSTRAINT(S) :
      C1 : - i <=  -1
      C2 :   i <=  2
      C3 : - j <=  -1
      C4 :   j <=  2
      C5 : - u <=  0
      C6 :   u <=  0
    VERTICES :
      S1  (  1 ,  1 ,  0) SATURATES :  C1 C3 C5 C6
      S2  (  1 ,  2 ,  0) SATURATES :  C1 C4 C5 C6
      S3  (  2 ,  1 ,  0) SATURATES :  C2 C3 C5 C6
      S4  (  2 ,  2 ,  0) SATURATES :  C2 C4 C5 C6
    NO RAY
```

Fig. 9: Multiplication of matrices.

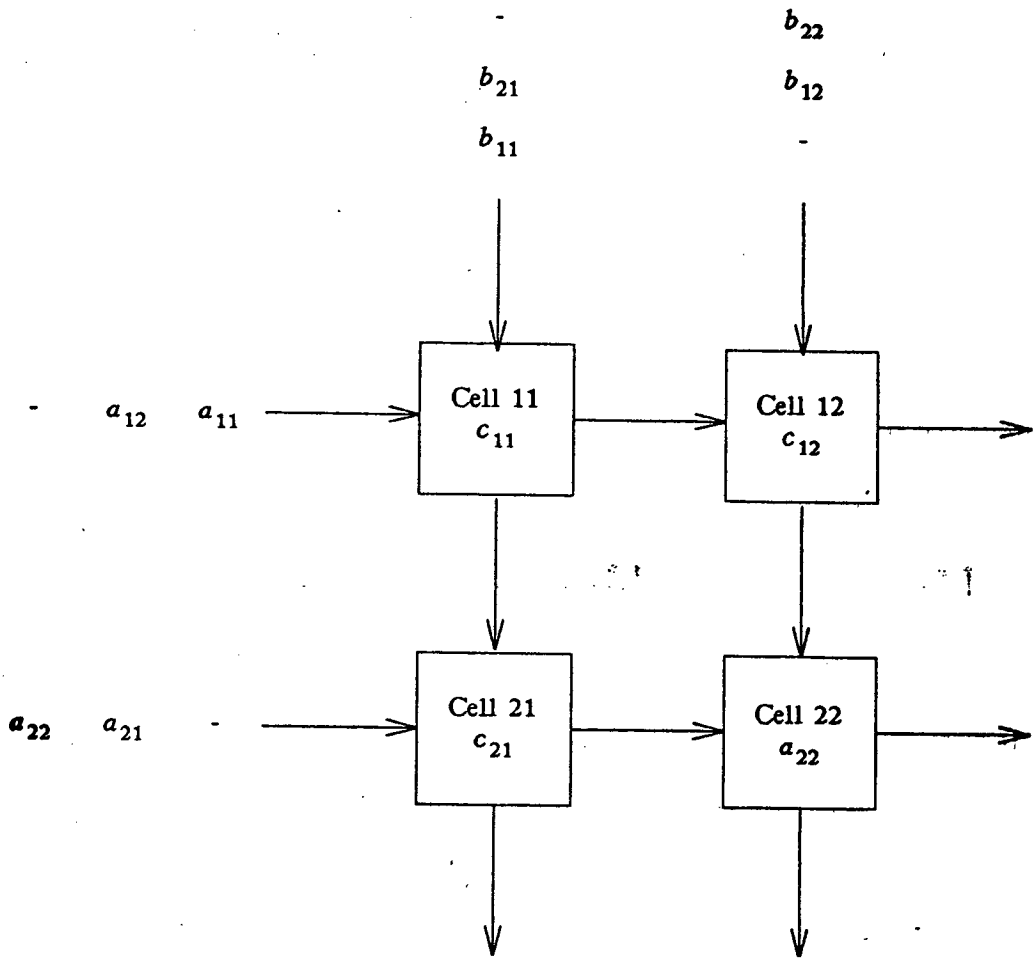

Fig. 10: Design for the matrix product; c_{ij} stays in cell (i,j). The network contains N^2 cells.

$$t(i,k) = \left\lfloor \frac{i}{2} + k \right\rfloor$$

and is obtained from the domain Lambda given by Fig. 7b. The allocation function is defined using the following trick. On Fig. 7a, it can be see that, although the dot product between the ray (1,0) and the vector $(\lambda_1,\lambda_2) = (\frac{1}{2},1)$ is not null, there are more than one computation that are done by a processor at a given time. However, this number of computations is bounded, so that we can allocate a bounded number of processors for each line. This is done by the allocation function:

$$a(i,k) = (i \ mod \ 2 \ , \ k)$$

The resulting systolic array is depicted by Fig. 8 (in the case $K = 2$). It is twice as fast as the previous one.

4.3. An example in dimension 3

Consider the multiplication of square matrices, $A = (a_{ij})$ and $B = (b_{ij})$, resulting in $C = (c_{ij})$ where $1 \leqslant i \ , \ j \leqslant N$. The equations giving $c_{i,j}$ are:

$$c_{ij} = \sum_{k=1}^{N} a_{ik} \, b_{kj}$$

Serializing this equation gives:

$$c_{ijk} = c_{ijk-1} + a_{ik} \, b_{kj}$$

$$c_{ij0} = 0$$

To obtain a uniform recurrent system of equations, one can replace a_{ik}, where index j is missing, by $a(i,j-1,k)$, and b_{kj} by $b(i-1,j,k)$. This gives the system shown by Fig. 9. The domain of computation is of course the cube defined by $1 \leqslant i \ , \ j \ ,k \leqslant N$. The timing-function is given by:

$$t(i,j,k) = i + j + k - 3$$

Now, since the domain D has no ray, any projection direction which is not parallel to the timing planes is OK. In Fig. 9, the projection chosen is (0,0,1), i.e. the k-axis. The resulting systolic array is depicted by Fig. 10. Of course, a number of other projections are interesting.

5. PERSPECTIVES

As it is, DIASTOL is unfortunately a toy. However, the way the system has been programmed makes it possible to extend its possibilities in such a way that it becames something (at leat hopefully) useful. Here is a list of the extensions that are planned.

5.1. Output of the systolic architecture

As it is, DIASTOL gives only an "abstract" specification of the solution. The (very) next version will include a command listing the processors, their actual interconnection, and their internal structure.

5.2. Graphic Editor

A graphic editor has already be programmed and partly debugged which allows convex domains to be visualized or plotted. It is possible to rotate, zoom, translate a domain interactively, as well as superimposing various domains. This facility seems to us very important, since it will allow somebody to really see the shape of a domain. The only reason why the graphic editor is not included in the current version is that it should be made device independent, which is not the case right now.

5.3. Extension to two-level pipeline systolic arrays

In [1], it is described how the method can be extended naturally to two-level pipeline systolic arrays. The modifications needed involve rewriting the part of DIASTOL concerning calculation of vertices and rays of domains, in such a way that Lambda-domains having a dimension greater than 3 can be handled.

5.4. A formal equation processor

The "formal" transformations that have been described in section 2.1 could be done automatically. A set of basic useful transformation could be defined and implemented quickly. Tools to help guessing which kind of transformation should be applied when they are not obvious could be provided by listing which nodes share the same variables, for example.

REFERENCES

[1] P. Quinton, "The Systematic Design of Systolic Arrays," MCNC Technical Report, TR84-11, May 1984.

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique