Systolic algorithms: concepts, synthesis, and evolution

Siang W. Song*

University of São Paulo
Institute of Mathematics and Statistics
Department of Computer Science
C.P. 20570 - São Paulo, SP – 01452-990 – Brazil
e-mail: song@ime.usp.br

Abstract

In this mini-course we present the main concepts of systolic algorithms. Systolic algorithms are designed with desirable characteristics to be implemented in VLSI silicon chips. The first examples of systolic designs were conceived in an "ad hoc" manner, requiring enormous amount of creativity and inspiration of its inventors. More recently, considerable amount of research effort has been spent in the development of formal methods to synthesize and generate systolic algorithms. In the first part of this mini-course we present the main ideas involved in these systolic algorithm synthesis methods. We will attempt to illustrate the method of dependency transformation by several examples and present a geometric interpretation to facilitate its understanding. We then proceed to give a more formal presentation of the synthesis method. In the second part we discuss an application of the results, originally derived for the area of systolic computing, in another area. We show that these results can be very important in a parallelizing compiler, to exploit parallelism in nested loops. We present a new method for cycle shrinking that can outperforms previous results.

1 Introduction

During the previous decades, enormous technological advances have been achieved in the area of VLSI ("Very Large Scale of Integration") circuits. Such technological advances gave rise to a totally new way of computing, constituted of highly parallel computing systems for specific applications. An intereteresting approach is the so-called *systolic array computing*, proposed originally by Kung and Leiserson [12], at the end of the seventies. A systolic array is a parallel computing device for a specific application, that is constituted of large number of simple processing elements called cells, interconnected in a regular way, with local communication. In a similar way as blood circulates in the human body, *data* circulate inside the cells of a systolic array, and interact with other data. Results computed in the cells are again pumped into other cells for further computing. The regularity and simplicity of the layout of the computing elements constitutes a desirable characteristic for their direct implementation in silicon, in the form of VLSI chips. Some excellent references about systolic algorithms include [3, 8, 11, 13, 14, 26, 29]. Some of the earlier systolic designs are described in [36].

*Supported by FAPESP (Fundação de Amparo à Pesquisa do Estado de São Paulo) - Proc. No. 93/0603-1 and CNPq (Conselho Nacional de Desenvolvimento Científico e Tecnológico) - Proc. No. 306063/88-3 and PROTEM-CC/SP.

Systolic arrays implemented in silicon chips are typically laid out in a linear array or bi-dimensional grid of cells. The first examples of systolic designs were conceived in an "ad hoc" manner, requiring enormous amount of creativity and inspiration of its inventors. More recently, considerable amount of interest has been aroused in the Computer Science community regarding the development of formal methods to synthesize and generate systolic algorithms. Such efforts gave rise to many works in this area, including the works by Fortes and Moldovan [6, 7, 18, 19], Quinton and Robert [25, 26], Delosme and Ipsen [4], Nelis and Deprettere [20], Mirandker and Winkler [17] and Huang and Lengauer [9]. In a certain way, all the proposed methods by the above researchers vary very little, differing slightly in the degree of formalism and the way to approach the problem. In the ultimate instance, they all use the concept of transformation of dependencies.

In the first part of this mini-course we present the main ideas involved in these systolic algorithm synthesis methods. We will attempt to illustrate the method of dependency transformation by several examples and present a geometric interpretation to facilitate its understanding. We then proceed to give a more formal presentation of the synthesis method. In the second part we discuss an application of the results originally derived for the area of systolic computing, in another area. We show that these results can be very important in a parallelizing compiler, to exploit parallelism in nested loops. The text used here is based on several works [26, 30, 37].

2 Examples of systolic algorithms

The pioneer work that introduced the systolic algorithm concept was due to Kung and Leiserson [12]. They proposed a novel way of computing the product of two matrices through the use of very simple computing cells and very fine grain of parallelism. We refer to their algorithm as version 1. Several other versions will be presented later on.

Consider two $n \times n$ matrices $A = (a_{ij})$ and $B = (b_{ij})$. To simplify the example, let $n = 3$. The product $C = (c_{ij})$

$$C = \begin{pmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{pmatrix}$$

can be expressed as the following recurrence equations:

$$\begin{aligned} c_{ij}^{1} &= 0 \\ c_{ij}^{k+1} &= c_{ij}^{k} + a_{ik} b_{kj} \\ c_{ij} &= c_{ij}^{n+1} \end{aligned}$$

2.1 Version 1: Kung-Leiserson

We illustrate here the systolic solution version 1 (Kung-Leiserson [12]). We use a basic cell constituted of three inputs a_{in}, b_{in}, c_{in} and three outputs $a_{out}, b_{out}, c_{out}$, as indicated in Figure 1.

The Figure 2 illustrates the layout of the cells to multiply 3×3 matrices.

The Kung-Leiserson systolic algorithm performs the multiplication of two $n \times n$ matrices in linear time, or $5n$ steps, using a total of $3n^2$ processing elements or cells.

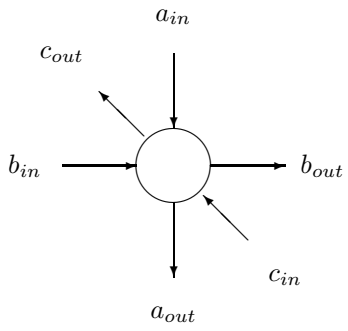

Figure 1: Basic cell version 1

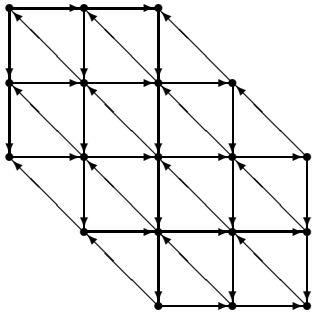

Figure 2: Systolic solution of Kung-Leiserson

2.2 Version 2: Weiser-Davis

Weiser and Davis [38] improved the previous solution by inverting the flow of the c values, as indicated by Figure 3.

Figure 4 illustrates the systolic solution of Weiser-Davis.

Version 2 realizes the multiplication of $n \times n$ matrices in $3n$ steps, using the same number of $3n^2$ cells.

2.3 Version 3: Okuda-Song

In order to reduce the number of cells utilized, without affecting the processing time, Okuda and Song [21] proposed a systolic solution illustrated in Figure 5, in which the c values stay in the cells.

Version 3 performs matrix multiplication also in $3n$ steps and uses only n^2 cells.

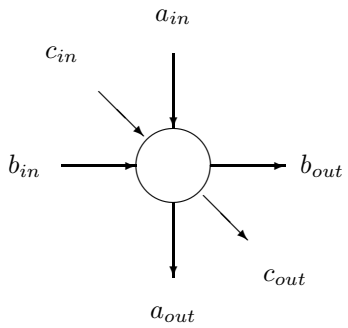

Figure 3: Basic cell version 2

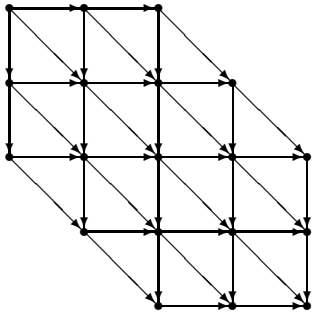

Figure 4: Systolic solution of Weiser-Davis

3 Dependence between computations

The method of synthesis of systolic algorithms is based on the transformation of the so-called *dependence vectors*. The method transforms a sequential algorithm, expressed in the form of up to three nested loops, or in the form of *uniform recurrence equations* [10], into a systolic algorithm. For this initial treatment, we use the form of nested loops. Later on we introduce also the form of uniform recurrence equations.

3.1 Expressing an algorithm as nested loops

Consider again the example of matrix multiplication. We can write the following sequential algorithm, as three nested loops.

```
for k := 1 to n do
    for i := 1 to n do
        for j := 1 to n do
            c_ij := c_ij + a_ik b_kj
```

Notice that the computation at a point (k, i, j) uses the computations done at other points.

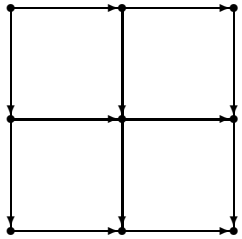

Figure 5: Systolic solution of Okuda-Song

Let us rewrite this algorithm in another equivalent form, with the following restrictions.

1. Every variable is indexed by all the indices k, i, j.
2. Every variable is computed explicitly and appears once on the left side of an assignment command.
3. No variable is used on both sides of the assignment operator.

Thus, we have the following equivalent sequential algorithm.

for $k := 1$ to n do
  for $i := 1$ to n do
    for $j := 1$ to n do
      begin
      $a_{ijk} := a_{i\ j-1\ k}$
      $b_{ijk} := b_{i-1\ j\ k}$
      $c_{ijk} := c_{i\ j\ k-1} + a_{ijk}b_{ijk}$
      end

The following initial values are assumed, for $1 \leq i, j, k \leq n$:

$$\begin{aligned} a_{i\ -1\ k} &= a_{ik} \\ b_{-1\ j\ k} &= a_{kj} \end{aligned}$$

3.2 Dependence vectors

Consider any assignment command, for example,

$$x_{ij} := x_{i-1\ j} + y_{i-1\ j}\ z_{i\ j-1}$$

The computation at the point of indices (i, j) utilizes values computed at points $(i-1, j)$ and $(i, j-1)$. This requires the previous computations done by commands of the type

$$
\begin{aligned}
x_{i-1\ j} &:= \cdots \\
y_{i-1\ j} &:= \cdots \\
z_{i\ j-1} &:= \cdots
\end{aligned}
$$

We say that the point (i, j) *depends* on points $(i-1, j)$ and $(i, j-1)$. We call the following vectors by the name of *dependence vectors*

$$
\begin{aligned}
\begin{pmatrix} 1 \\ 0 \end{pmatrix} &= \begin{pmatrix} i \\ j \end{pmatrix} - \begin{pmatrix} i-1 \\ j \end{pmatrix} \\
\begin{pmatrix} 0 \\ 1 \end{pmatrix} &= \begin{pmatrix} i \\ j \end{pmatrix} - \begin{pmatrix} i \\ j-1 \end{pmatrix}
\end{aligned}
$$

Let x be an indexed variable. If variable x is generated or computed at point i and used at point i', then we have a dependence vector $\mathbf{i'} - \mathbf{i}$, associated to variable x.

In the example of matrix multiplication, we have the following dependence vectors, expressed in the base (k, i, j), associated respectively to the variables c, a and b.

$$
\begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}
$$

The dependence vectors of an algorithm form the so-called *dependence matrix* D.

$$
D = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix}.
$$

4 Method to synthesize systolic algorithms

The computations involved in a sequential algorithm, expressed as nested loops, can be represented in a space of n dimensions, where n is the number of indices used in the loops. Figure 6 represents the computations of our example, with the arrows indicating the dependences between computations.

4.1 Computation graph

We can define a *computation graph*, where the nodes are points (k, i, j) of the space of the indices of the algorithm and the edges indicate the dependences. Figure 6 represents therefore a computation graph.

4.2 Transformation of the computation graph

Let us obtain a linear transformation T that transforms the computation graph into another graph, called the *transformed graph*.

$$
\text{Computation graph} \xrightarrow{T} \text{Transformed graph}
$$

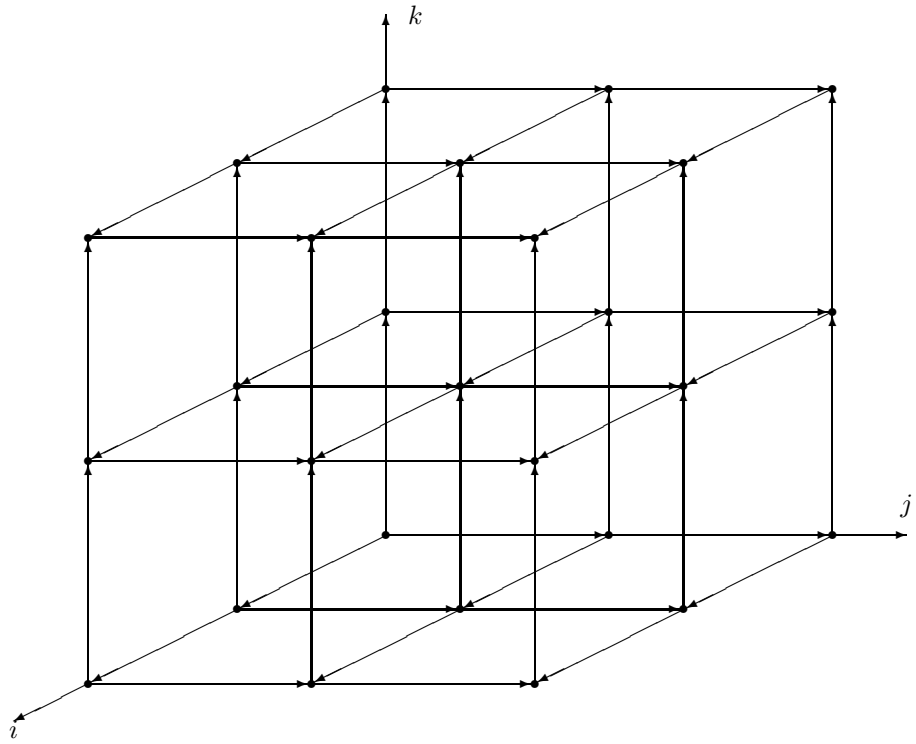

Figure 6: The computations of the example

In the transformed graph, each node is denominated (t, x, y). The interpretation of t is a time instant The interpretation of x and y is a position of a processing cell on a plane.

Transformation T should be such that if a node $v = (k, i, j)$ of the computation graph is mapped to node $v' = (t, x, y)$ of the transformed graph, then v is computed at the time instant t, by a processing cell located at the position of coordinates (x, y).

Let

$$T = \begin{pmatrix} t_{11} & t_{12} & t_{13} \\ t_{21} & t_{22} & t_{23} \\ t_{31} & t_{32} & t_{33} \end{pmatrix}, \quad t_{ij} \text{ integers.}$$

We have then

$$\begin{pmatrix} t_{11} & t_{12} & t_{13} \end{pmatrix} \begin{pmatrix} k \\ i \\ j \end{pmatrix} = \text{time to compute } (k, i, j)$$

$$\begin{pmatrix} t_{21} & t_{22} & t_{23} \\ t_{31} & t_{32} & t_{33} \end{pmatrix} \begin{pmatrix} k \\ i \\ j \end{pmatrix} = \text{cell position } (x, y) \text{ to compute } (k, i, j)$$

4.3 Obtention of T

Let us consider a dependence vector d and a point $\mathbf{i}' = (k', i', j')$ dependent on another point $\mathbf{i} = (k, i, j)$ in the computation graph, such that $d = \mathbf{i}' - \mathbf{i}$. The computation at point $\mathbf{i}'$ should

be executed *after* the computation at point **i**. We should be therefore

$$\begin{pmatrix} t_{11} & t_{12} & t_{13} \end{pmatrix} \begin{pmatrix} k' \\ i' \\ j' \end{pmatrix} - \begin{pmatrix} t_{11} & t_{12} & t_{13} \end{pmatrix} \begin{pmatrix} k \\ i \\ j \end{pmatrix} > 0$$

In other words, we have

$$\begin{pmatrix} t_{11} & t_{12} & t_{13} \end{pmatrix} d > 0$$

for all dependence vector d of the algorithm.

Considering points $\mathbf{i'}$ and $\mathbf{i}$,

$$\begin{pmatrix} t_{21} & t_{22} & t_{23} \\ t_{31} & t_{32} & t_{33} \end{pmatrix} \begin{pmatrix} k' \\ i' \\ j' \end{pmatrix} - \begin{pmatrix} t_{21} & t_{22} & t_{23} \\ t_{31} & t_{32} & t_{33} \end{pmatrix} \begin{pmatrix} k \\ i \\ j \end{pmatrix}$$

indicates the direction of the communication channel between neighboring cells. We require that such directions be easily implemented. Let us choose for example

$$\begin{pmatrix} t_{21} & t_{22} & t_{23} \\ t_{31} & t_{32} & t_{33} \end{pmatrix} d = \begin{cases} \pm(1,0)^T \\ \pm(0,1)^T \\ \pm(1,1)^T \end{cases}$$

for all dependence vector d.

5 Examples

Let us go back to the matrix multiplication example. Consider the following transformation

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

T transforms the dependence vectors D to the new vectors D'

$$TD = D',$$

that is,

$$\begin{pmatrix} 1 & 1 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix}.$$

With such a transformation, we reproduce the solution of Okuda-Song.

Use now the transformation

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

that transforms the dependence vectors to

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

Now we have reproduced the systolic solution of Kung-Leiserson.

Finally, let us use the transformation

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

that transforms the dependence vectors to

$$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix} \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \end{pmatrix}.$$

In this case, we have reproduced the systolic solution of Weiser-Davis.

6 Systolic algorithm synthesis method

We now give a more formal presentation of the method to synthesize systolic algorithms. This section is based almost entirely on the works of Quinton and Robert [25, 26]. We consider the specification of an algorithm as a system of uniform recurrence equations, that defines the set of computations associated with the points of a convex polyhedral domain. The computations involved in an algorithm, as well as the dependences between computations, are transformed by a time function and an space allocation function. The time function maps each computation of the algorithm to a positive integer that represents the time at which the computation is executed. The allocation function, on the other hand, obtains a position of the processing element to perform the computation involved. By using adequate allocation functions, we can obtain a layout of the processing elements easily implemented on a VLSI chip.

7 Uniform recurrence equations

Let D be the set of points of integer coordinates belonging to a convex polyhedral domain of R^n

$$D = \{z \in Z^n | Bz \leq b\}$$

where B is a $m \times n$ matrix and b a $m \times 1$ vector over Z.

Denote as *vertices* of D those points of D that cannot be expressed as *convex combinations* of other points of D. A point z is said to be a convex combination of points $z_0, z_1, \ldots, z_{q-1}$ if it can be expressed in the form

$$z = \sum_{i=0}^{q-1} \mu_i z_i, \quad \text{with real numbers } \mu_i \geq 0, \sum_{i=0}^{q-1} \mu_i = 1.$$

We consider limited convex polyhedral domains.

Definition 7.1 *Let D be a convex polyhedral domain. A system of uniform recurrence equations URE is a system of m equations of the type*

$$V_i(z) = f_i(V_{i_1}(z - \theta_{i_1}), \ldots, V_{i_k}(z - \theta_{i_k})), 0 \leq i < m.$$

where $z \in D, \theta_{i_1}, \ldots, \theta_{i_k}$ are vectors of Z^n.

We say that such a system computes m functions

$$V_0, V_1, \ldots, V_{m-1}.$$

To simplify the presentation, without loss of generality, we also consider the simplified case in which, among the computed functions, we are only interested in V_0.

Definition 7.2 *A system of (simplified) uniform recurrence equations URE is the following system of m equations.*

$$\begin{aligned} V_0(z) &= f(V_0(z-\theta_0), V_1(z-\theta_1), \ldots, V_{m-1}(z-\theta_{m-1})), \\ V_1(z) &= V_1(z-\theta_1), \\ &\vdots \\ V_{m-1}(z) &= V_{m-1}(z-\theta_{m-1}), \end{aligned}$$

where $z \in D$ and $\theta_i, 0 \leq i < m$, are vectors of Z^n.

7.1 Dependence graph

Definition 7.3 *Let $\Theta = \{\theta_0, \theta_1, \ldots, \theta_{m-1}\}$ the set of vectors θ_i of a URE system. The θ_i are called dependence vectors of the system.*

Consider points $z, y \in D$, we say that z is dependent on y, by θ_i, if there exists $\theta_i \in \Theta$, such that

$$z = y + \theta_i.$$

The dependences can be represented by a directed graph called dependence graph of the URE system, denoted by (D, Θ). Its nodes are the points of D. If point z is dependent on y by θ_i, there exists an edge from node y to node z. (See Figure 7.)

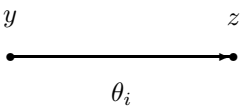

Figure 7: Point z dependent on y

A URE system can be used to specify an algorithm [10]. If $z \in D$ and $z - \theta_i \notin D$, we call the values $V_i(z-\theta_i), 0 \leq i < m$ as *inputs* of the algorithm. The instances of $V_i(z)$ that appear only on the right side of equations correspond to the *outputs* of the algorithm. In the case of an algorithm specified by a simplified URE system, we are only interested in the output V_0, the remaining outputs being merely reproductions of the inputs of the algorithm.

7.2 Examples

Example 1 *The algorithm to multiply two $N \times N$ matrices, $A = (a_{ij})$ and $B = (b_{ij})$, giving the product $C = (c_{ij}), c_{ij} = \sum_{k=0}^{N-1} a_{ik} b_{kj}$, can be expressed as a URE system.*

$0 \le i < N,\ 0 \le j < N,\ 0 \le k < N,$

$$C(i,j,k) = C(i,j,k-1) + A(i,j-1,k)B(i-1,j,k)$$
$$A(i,j,k) = A(i,j-1,k)$$
$$B(i,j,k) = B(i-1,j,k)$$

The input values $A(i,-1,k)$ and $B(-1,j,k)$ are defined as a_{ik} and b_{kj}, respectively. The initial values of $C(i,j,-1)$ are 0. The output values $C(i,j,N-1)$, that appear only on the right side of the equations, will be the results of c_{ij}.

The dependence vectors are

$$\theta_c = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}, \theta_a = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}, \theta_b = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}.$$

D is the convex polyhedral domain defined by

$$\{z \in Z^n | Bz \le b\}$$

with

$$B = \begin{pmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \quad b = \begin{pmatrix} 0 \\ 0 \\ 0 \\ N-1 \\ N-1 \\ N-1 \end{pmatrix}$$

Figure 8 illustrates the dependence graph for $N = 3$. The dependences are shown by directed edges,

Example 2 *The algorithm for the resolution of Laplace equation in a 3-dimensional space, by the iterative method based on differences [41], can be described by the following URE system. The algorithm performs M iterations.*

$0 \le t < M,\ 0 \le i < N, 0 \le j < N, 0 \le k < N,$

$$\begin{aligned} U(t,i,j,k) = [&U(t-1,i-1,j,k) + U(t-1,i+1,j,k) + \\ &U(t-1,i,j-1,k) + U(t-1,i,j+1,k) + \\ &U(t-1,i,j,k-1) + U(t-1,i,j,k+1)]/6 \end{aligned}$$

We have the following dependence vectors.

$$\theta_0 = \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \end{pmatrix},\ \theta_1 = \begin{pmatrix} 1 \\ -1 \\ 0 \\ 0 \end{pmatrix},\ \theta_2 = \begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \end{pmatrix},$$

$$\theta_3 = \begin{pmatrix} 1 \\ 0 \\ -1 \\ 0 \end{pmatrix},\ \theta_4 = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 1 \end{pmatrix},\ \theta_5 = \begin{pmatrix} 1 \\ 0 \\ 0 \\ -1 \end{pmatrix}.$$

The values of $U(-1,i,j,k)$ are the inputs to the algorithm. The result of the algorithm is the output $U(M-1,i,j,k)$.

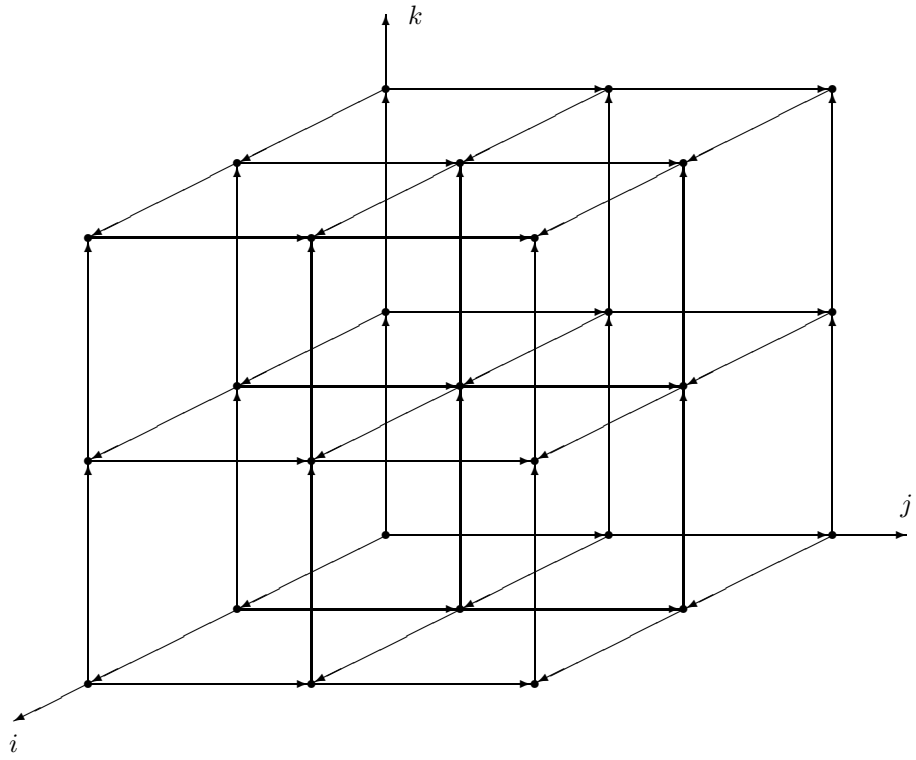

Figure 8: Dependence graph ($N = 3$)

8 Time function τ

Given a URE system, we want to obtain a time function τ that schedules the computations associated to the points of domain D.

For a given $z \in D$, we assume that the computations $V_i(z)$ are performed in parallel, and take a unit time.

In order to perform the computations of $V_i(z)$, its arguments naturally should have been computed before.

The time function τ associates to each point $z \in D$ the time instant it is computed. If such a function exists, then we say that the URE system is computable. Informally we note that in order for V_i to be computable at point z, it should not depend on itself. For example, the following two URE systems are not computable. One characterization for computability is given by Karp, Miller and Winograd in [10].

$0 \leq i < N$,
$$U(i) = f(V(i-1), U(i))$$

$0 \leq i < N$,
$$U(i) = g(V(i-2), W(i-1))$$
$$W(i) = h(U(i+1))$$

Definition 8.1 *Let* $x = \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{pmatrix} \in D.$

Let $\delta \in Z$, *and* $\lambda = \begin{pmatrix} \lambda_0 \\ \lambda_1 \\ \vdots \\ \lambda_{n-1} \end{pmatrix}$, $\lambda_i \in Z$.

The time function $\tau : Z^n \to Z$ *is the following linear function*

$$\begin{aligned} \tau(x) &= \lambda_0 x_0 + \lambda_1 x_1 + \ldots + \lambda_{n-1} x_{n-1} + \delta \\ &= \lambda^T x + \delta \end{aligned}$$

that satisfies the two conditions:

1. τ *is non negative over* D.
2. *If* z *depends on* y, *then* $\tau(z) > \tau(y)$.

The first condition is for a question of convenience, since τ is interpreted as time. The second condition makes scheduling of dependent computations possible.

Theorem 1 (Quinton and Robert [26])
For a given URE system, with dependence graph (D, Θ) *and set* V_D *of vertices of* D, *the parameters* $\lambda_0, \lambda_1, \ldots, \lambda_{n-1}$ *and* δ *define a time function* τ *if and only if*

1. $\forall v_i \in V_D, \quad \lambda^T v_i + \delta \geq 0$ *and*
2. $\forall \theta_i \in \Theta, \quad \lambda^T \theta_i > 0$, *or,* $\lambda^T \theta_i \geq 1$.

Proof:

The proof is immediate.

Condition 1 is equivalent to having τ non negative over D, or,

$$\forall v_i \in V_D,\ \lambda^T v_i + \delta \geq 0 \quad \text{is equivalent to} \quad \forall z \in D,\ \tau(z) \geq 0.$$

In fact, if τ is always non negative over D, in particular $\tau(v_i) \geq 0$, where v_i are vertices of D. Therefore,

$$\tau(v_i) = \lambda^T v_i + \delta \geq 0.$$

On the other hand, if $\lambda^T v_i + \delta \geq 0,\ \forall v_i \in V_D$, let us show that $\tau(z) \geq 0, \forall z \in D$.

Any point $z \in D$ can be expressed as a convex combination of vertices of D. So,

$$z = \sum_{i=0}^{\nu-1} \mu_i v_i, \quad \mu_i \geq 0, \sum_{i=0}^{\nu-1} \mu_i = 1.$$

We have then

$$\tau(z) = \lambda^T z + \delta = \sum_{i=0}^{\nu-1} \mu_i \lambda^T v_i + \delta = \sum_{i=0}^{\nu-1} \mu_i (\lambda^T v_i + \delta) \geq 0.$$

Condition 2 is equivalent to requiring $\tau(z) > \tau(y)$ if z depends on y.

In fact, if z depends on y, then

$$\exists \theta_i \in \Theta, \quad z - y = \theta_i.$$

We can write

$$\tau(z) - \tau(y) = \lambda^T z + \delta - \lambda^T y - \delta = \lambda^T \theta_i.$$

Thus,

$$\tau(z) > \tau(y) \quad \text{equivalent to} \quad \lambda^T \theta_i > 0.$$

□

8.1 Examples

In Example 1 on page 10,

$$\Theta = \{\theta_c = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}, \theta_a = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}, \theta_b = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}\}.$$

A time function $\tau(x) = \lambda_0 x_0 + \lambda_1 x_1 + \lambda_2 x_2 + \delta$ that satisfies the conditions of Theorem 1 should satisfy the following restrictions.

For condition 1, it suffices to consider vertex $(0, 0, 0)$:

$$\delta \geq 0.$$

For condition 2, we should have

$$\begin{aligned} \lambda_0 &\geq 1, \\ \lambda_1 &\geq 1, \\ \lambda_2 &\geq 1. \end{aligned}$$

Thus we can have the following time function

$$\tau(x) = x_0 + x_1 + x_2.$$

In Example 2 of page 11, application of Theorem 1 gives the following restrictions.

$$\begin{aligned} \delta &\geq 0, \\ \lambda_0 + \lambda_1 &\geq 1, \\ \lambda_0 - \lambda_1 &\geq 1, \\ \lambda_0 + \lambda_2 &\geq 1, \\ \lambda_0 - \lambda_2 &\geq 1, \\ \lambda_0 + \lambda_3 &\geq 1, \\ \lambda_0 - \lambda_3 &\geq 1. \end{aligned}$$

A solution is

$$\lambda_0 = 1,\ \lambda_1 = \lambda_2 = \lambda_3 = 0.$$

We have then

$$\tau(x) = x_0.$$

8.2 Interpretation of the parameter $\lambda^T\theta_i$

Theorem 1 requires $\lambda^T\theta_i \geq 1$, for each dependence vector θ_i.

Let z be dependent on y, by θ_i, or $z = y + \theta_i$. In the proof of Theorem 1 we saw:

$$\lambda^T\theta_i = \lambda^T(z - y) = \tau(z) - \tau(y).$$

Thus $\lambda^T\theta_i$ expresses the *delay* between the computation of z and the computation of y. Theorem 1 requires this delay to be greater or equal than 1.

8.3 Optimal time functions

Theorem 1 gives conditions for the obtention of a time function. We now consider the question of optimality of time functions. Several criteria for optimality can be adopted.

Consider a URE system with dependence graph (D, Θ). One criterium is to minimize the delay between dependent computations according to a dependence θ_k of Θ. Thus we want to:

$$\text{minimize } \lambda^T\theta_k,$$

subject to

$$\lambda^T\theta_i \geq 1, \quad \forall\theta_i \in \Theta.$$

Example 3 *Let the URE system*

$$0 \leq i < 13,\ \ 0 \leq j < 7,$$
$$W(i,j) = f\left(W(i-1,j-1), W(i-1,j-2), W(i-2,j-1)\right)$$

We have the following dependence vectors, represented in Figure 9.

$$\Theta = \{\theta_0 = \begin{pmatrix} 1 \\ 1 \end{pmatrix}, \theta_1 = \begin{pmatrix} 1 \\ 2 \end{pmatrix}, \theta_2 = \begin{pmatrix} 2 \\ 1 \end{pmatrix}\}$$

The conditions of Theorem 1 give the following restrictions.

$$\begin{aligned} \lambda_0 + \lambda_1 &\geq 1, \\ \lambda_0 + 2\lambda_1 &\geq 1, \\ 2\lambda_0 + \lambda_1 &\geq 1, \\ \delta &\geq 1. \end{aligned}$$

Minimizing the delay along the dependence

$$\theta_1 = \begin{pmatrix} 1 \\ 2 \end{pmatrix},$$

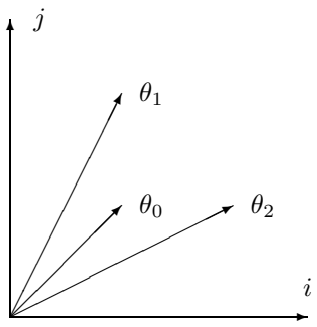

Figure 9: Dependence vectors

or, minimizing $\lambda_0 + 2\lambda_1$, we obtain

$$\begin{aligned} \lambda_0 &= 1, \\ \lambda_1 &= 0, \\ \tau_1(x) &= x_0. \end{aligned}$$

Using the time function τ_1, the URE system of this example takes 12 time units to be computed.

Alternatively we can minimize the delay along the dependence

$$\theta_2 = \begin{pmatrix} 2 \\ 1 \end{pmatrix},$$

and we obtain the result

$$\begin{aligned} \lambda_0 &= 0, \\ \lambda_1 &= 1, \\ \tau_2(x) &= x_1. \end{aligned}$$

Using the time function τ_2, the URE system of this same example takes only 6 time units to be computed.

Instead of using local criterium, we can also adopt a global one, by minimizing the maximum difference between the computation times of the points of D:

$$\text{minimize } \max_{x,y \in D} |\tau(x) - \tau(y)|,$$

subject to the restrictions of Theorem 1.

Returning to Example 3, the largest difference between the computation times occur at the points $x = (12, 6)$ and $y = (0, 0)$.

By minimizing $|\tau(x) - \tau(y)| = 12\lambda_0 + 6\lambda_1$, we obtain

$$\begin{aligned} \lambda_0 &= 0, \\ \lambda_1 &= 1, \end{aligned}$$

$$\tau_3(x) = \tau_2(x) = x_1.$$

9 Allocation function α

The time function τ maps to each point of D the time instant in which the computation associated to that point is performed. We now examine another function, called *allocation function* $\alpha : Z^n \rightarrow Z^{n-1}$, that maps to each point of D a position $\alpha(z)$ where the computation associated to that point is performed. Point $\alpha(z)$ can be interpreted as the position of a processing element of a certain computing system.

We assume that each processing element is capable of realizing the computation associated to a point of D in a unit time.

The allocation function α must not map to the same processing element different points of D if the associated computations are executed at the same time.

Definition 9.1 *The allocation function* $\alpha : Z^n \rightarrow Z^{n-1}$ *is a function of the form*

$$\alpha(x) = (\alpha_0(x), \alpha_1(x), \ldots, \alpha_{n-2}(x))$$

where each α_i *is a linear function of* Z^n *in* Z*, such that*

$$\forall x, y \in D, \quad \alpha(x) = \alpha(y) \Rightarrow \tau(x) \neq \tau(y).$$

9.1 Time hyperplanes

A *time hyperplane* is a hyperplane of the points $z \in D$ that possess the same value of $\tau(z)$, i.e., the points whose computations are executed at the same time. We illustrate this concept through a simple example.

Example 4 *Consider the following URE system.*

$0 \leq i < 4, \;\; 0 \leq j < 4,$
$$U(i,j) = f\left(U(i,j-1), U(i-1,j)\right)$$

We have

$$\Theta = \{\theta_0 = \begin{pmatrix} 0 \\ 1 \end{pmatrix}, \theta_1 = \begin{pmatrix} 1 \\ 0 \end{pmatrix}\}$$

Let the time function be $\tau(x) = x_0 + x_1 = i + j$, that is,

$$\lambda = \begin{pmatrix} 1 \\ 1 \end{pmatrix}.$$

The time hyperplanes are orthogonal to λ. In Figure 10, the time hyperplanes are represented by thick lines.

In the following theorem, we obtain an allocation function α by projecting the domain D along a direction given by u, that is not orthogonal to λ.

Theorem 2 (Quinton and Robert [26])
Let u *be a non-null vector of* Z^n *such that* $\lambda^T u \neq 0$.
Let $u_j \neq 0$ *be a component of* u*. The following application defines an allocation function* $\alpha(x) = (\alpha_0(x), \alpha_1(x), \ldots, \alpha_{n-2}(x))$*, where*

$$\begin{aligned} \alpha_k(x) &= u_j x_k - u_k x_j \qquad \text{if } 0 \leq k < j \\ \alpha_k(x) &= u_j x_{k+1} - u_{k+1} x_j \qquad \text{if } j \leq k < n-1. \end{aligned}$$

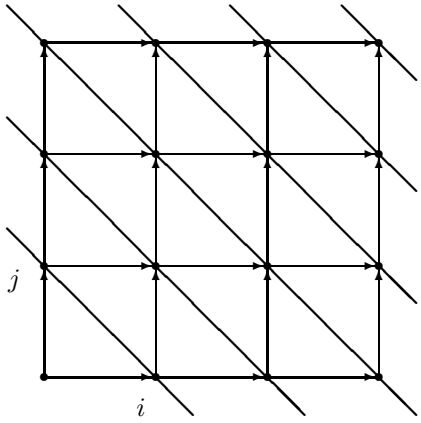

Figure 10: Time hyperplanes

Proof:

The allocation vector $\alpha(x)$ can be written as $\alpha(x) = Sx$ where S is the following $(n-1) \times n$ matrix.

$$S = \begin{pmatrix} u_j & & -u_0 & & \\ & \ddots & \vdots & & 0 \\ & u_j & -u_{j-1} & & \\ & & -u_{j+1} & u_j & \\ 0 & & \vdots & & \ddots \\ & & -u_{n-1} & & u_j \end{pmatrix}$$

We can verify that $\alpha(u) = 0$.

Consider $x, y \in D,\ x \neq y$ and $\alpha(x) = \alpha(y)$. Then there exists a non-zero scalar μ such that $x - y = \mu u$. In fact, let $z = x - y$. We have $\alpha(z) = 0$, which implies

$$\begin{aligned} \alpha_k(z) &= u_j z_k - u_k z_j = 0, \qquad 0 \leq k < j \\ \alpha_k(z) &= u_j z_{k+1} - u_{k+1} z_j = 0, \quad j \leq k < n-1. \end{aligned}$$

Thus, $z_k = (z_j/u_j)u_k = \mu u_k$ and $z = \mu u$ or $x - y = \mu u$.

As $\lambda^T u \neq 0$, we have

$$\tau(x) - \tau(y) = \lambda^T (x - y) = \mu \lambda^T u \neq 0.$$

□

From now on we use also the following notation to express the allocation function α:

$$\alpha(x) = Sx = \begin{pmatrix} s_{00} & s_{01} & \cdots & s_{0\ n-1} \\ \vdots & \vdots & \ddots & \vdots \\ s_{n-2\ 0} & s_{n-2\ 1} & \cdots & s_{n-2\ n-1} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{pmatrix}.$$

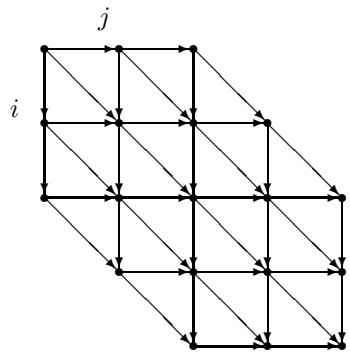

Figure 11: Revisiting the systolic solution of Weiser-Davis

9.2 Examples

We return to Example 1 of page 10. We have obtained

$$\lambda = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$$

$$\tau(x) = \lambda^T x = x_0 + x_1 + x_2.$$

Let us apply Theorem 2, using

$$u = \begin{pmatrix} -1 \\ -1 \\ 1 \end{pmatrix}.$$

We have $\lambda^T u \neq 0$. Let us choose $u_j = u_2 = 1$. We have

$$\alpha_0(x) = u_2 x_0 - u_0 x_2 = x_0 + x_2$$

$$\alpha_1(x) = u_2 x_1 - u_1 x_2 = x_1 + x_2.$$

Therefore,

$$\alpha(x) = Sx = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} x_0 + x_2 \\ x_1 + x_2 \end{pmatrix}.$$

With this allocation function α, the points of D are mapped to the processing elements with the layout of Figure 11.

Figure 11 shows a computing system that carries out the computations of the given URE system. The communication between processing elements is indicated by the arrows of the figure. The communication directions are obtained by applying function α to the dependence vectors of the system. Thus,

$$\alpha(\theta_c) = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

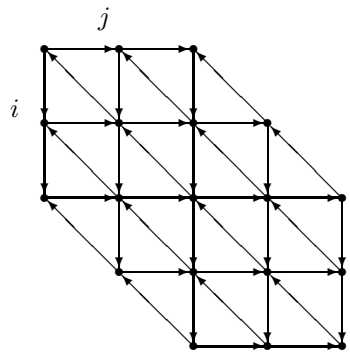

Figure 12: Revisiting the systolic solution of Kung-Leiserson

$$\alpha(\theta_a) = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

$$\alpha(\theta_b) = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

It is interesting to observe that with these time and allocation functions we have again reproduced the systolic solution for matrix multiplication proposed by Weiser and Davis [38]. Furthermore if we choose

$$u = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$$

to project the domain D, we obtain the allocation function

$$\alpha(x) = Sx = \begin{pmatrix} 1 & 0 & -1 \\ 0 & 1 & -1 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} x_0 - x_2 \\ x_1 - x_2 \end{pmatrix}.$$

Figure 12 shows the layout of the processing elements obtained by this new allocation function. Now we have derived again the systolic solution of Kung and Leiserson [12].

9.3 Choice of allocation function

We now examine the question of *best* allocation functions. There are many criteria to choose allocation functions. One criterium usually considered is the number of processing elements or cells needed. The computing systems obtained by Weiser-Davis and Kung-Leiserson, for example, need $3N^2 - 3N + 1$ processing cells, to multiply $N \times N$ matrices. The criterium of minimizing the number of cells is justified in systolic systems for VLSI implementation to obtain a smaller silicon chip area.

We may wish to have an allocation function such that the resulting computing system can be easily implemented on a given computing system with specific communication channels. Thus, we may want an allocation function such that each dependence vector is mapped onto an existing communication channel of the given computing system.

9.4 Examples

Consider again Example 1 of page 10. We wish to get an allocation function α such that

$$\begin{aligned}\alpha(\theta_c) &= 0 \text{ or } I_{s_1}\\ \alpha(\theta_a) &= 0 \text{ or } I_{s_2}\\ \alpha(\theta_b) &= 0 \text{ or } I_{s_3}, \quad \text{with } s_1, s_2, s_3 = 0 \text{ or } 1.\end{aligned}$$

Let $u = \begin{pmatrix} u_o \\ u_1 \\ u_2 \end{pmatrix}$, $\lambda^T u = u_0 + u_1 + u_2$.

Let $u_j = u_2 \neq 0$.

We have

$$\begin{aligned}\alpha_0(x) &= u_2 x_0 - u_0 x_2\\ \alpha_1(x) &= u_2 x_1 - u_1 x_2\\ \alpha(x) &= \begin{pmatrix} u_2 & 0 & -u_0 \\ 0 & u_2 & -u_1 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix}.\end{aligned}$$

By applying α to the dependence vectors, we have

$$\alpha(\theta_c) = \begin{pmatrix} u_2 & 0 & -u_0 \\ 0 & u_2 & -u_1 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} -u_0 \\ -u_1 \end{pmatrix}$$

$$\alpha(\theta_a) = \begin{pmatrix} u_2 & 0 & -u_0 \\ 0 & u_2 & -u_1 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ u_2 \end{pmatrix}$$

$$\alpha(\theta_b) = \begin{pmatrix} u_2 & 0 & -u_0 \\ 0 & u_2 & -u_1 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} u_2 \\ 0 \end{pmatrix}.$$

A possible solution is $u_2 = 1, u_0 = u_1 = 0$, that is

$$u = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$$

giving

$$\alpha(x) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} x_0 \\ x_1 \end{pmatrix}.$$

The computing system obtained is shown in Figure 13 and corresponds to the Okuda-Song solution of [21].

We can obtain another interesting solution. Consider

$$u = \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix} \text{ e } u_j = u_0 = 1.$$

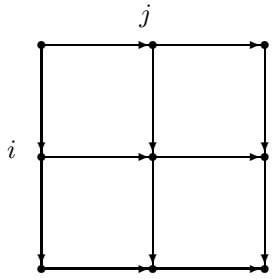

Figure 13: Reproducing the Okuda-Song solution

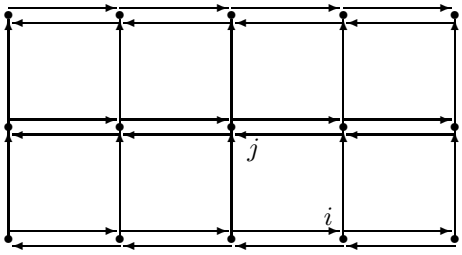

Figure 14: Communication channels on a rectangular grid

We have

$$\begin{aligned}
\alpha_0(x) &= u_0x_1 - u_1x_0 = x_1 - x_0 \\
\alpha_1(x) &= u_0x_2 - u_2x_0 = x_2 \\
\alpha(x) &= \begin{pmatrix} -1 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} -x_0 + x_1 \\ x_2 \end{pmatrix}.
\end{aligned}$$

By applying α to the dependence vectors of the example, we have

$$\begin{aligned}
\alpha(\theta_c) &= \begin{pmatrix} -1 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \\
\alpha(\theta_a) &= \begin{pmatrix} -1 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \\
\alpha(\theta_b) &= \begin{pmatrix} -1 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} -1 \\ 0 \end{pmatrix}.
\end{aligned}$$

With this new function α, the points are transformed to the layout of Figure 14.

10 Systolic computation – evolution along another direction

In the previous sections, we have examined the main concepts of systolic algorithms. The first systolic algorithms that appeared in the early eighties were conceived in an "ad hoc" manner. As

a natural evolution in the area of systolic computation, formal synthesis methods were proposed so that many previous algorithms constitute mere special cases of this method, as we have seen.

Along another direction of this evolution, several researchers noticed that the dependence transformation method, originally conceived to derive systolic algorithms, can also be useful in other areas. For instance, Ribas used the Fortes-Moldovan synthesis method as a starting point of a new method to generate code for the Warp machine [28]. Fortes and his collaborators used it to generalize the so-called *cycle shrinking* technique. In other words, we are no longer using the method to derive systolic algorithms, but rather to aid a parallelizing compiler. Many other important topics, such as task partitioning in parallel computers, should also benefit from results of systolic computing.

We now proceed to show some related results, along this direction, due to Robert and Song [30].

11 Cycle shrinking

Several loop transformations techniques have been designed to extract parallelism from nested loop structures [1, 5, 24, 39, 40]. As pointed out by Polychronopoulos [23], by Wolfe [39] or by Shang, O'Keefe and Fortes [34, 35], this task is often performed by optimizing and parallelizing compilers that have as their goal the transformation and mapping of a serial program into a parallel form that can be executed on a particular architecture. Nested loop structures offer the most fruitful sources of parallelism in serial programs, and it is therefore of paramount importance that the analysis necessary for such programs be both precise and efficient.

In [23], three loop transformation techniques - *simple cycle shrinking, selective cycle shrinking* and *true dependence cycle shrinking* - were introduced to transform sequential loops into parallel loops. Shang, O'Keefe and Fortes [34, 35] have recently improved these results along two directions:

- the three different transformations are put into a unified framework
- methods to derive optimal cycle shrinking are presented

Optimal cycle shrinkings are determined through an optimization procedure described in [33].

Another technique for improving the parallelism of loop structures has been introduced by Liu, Ho and Sheu [16] as the *Index Shift Method*. The idea behind this method is to defer or advance the execution steps of some statements such that the total execution time of loops can be reduced.

The main result is a new methodology that permits to combine cycle shrinking techniques with the index shift method.

12 Background material

Throughout the paper, we consider perfect nests of loops of the following form:

GLN (General Loop Nest) class

for $i_1 = l_1$ to u_1 do
for $i_2 = l_2(i_1)$ to $u_2(i_1)$ do

for $i_3 = l_3(i_1, i_2)$ to $u_3(i_1, i_2)$ do
  $\vdots$
for $i_n = l_n(i_1, i_2, \ldots, i_{n-1})$ to $u_n(i_1, i_2, \ldots, i_n - 1)$ do
  begin
    { statement S_1 }
    { statement S_2 }
      $\vdots$
    { statement S_k }
  end

Here, l_1 and u_1 are assumed to be constants, while $l_j(i_1, \ldots, i_{j-1})$ (resp: $u_j(i_1, \ldots, i_{j-1})$) is the minimum (resp: maximum) of a finite number of affine functions of $i_1, \ldots, i_{j-1}$. Also, we require that all variables instanciated in the statements S_1 to S_k be affine functions of the loop indices $i_1, i_2, \ldots, i_n$. The index set for the loop nest is defined as $Dom = \{I = (i_1, \ldots, i_n) \in Z^n, l_j \leq i_j \leq u_j \text{ for } 1 \leq j \leq n\}$. This framework seems to be pretty general and widely used [1, 5, 40].

Data dependences between instances of the statements $S_1, \ldots, S_k$ are defined according to Banerjee [1] or Polychronopoulos [23]. We write $S_u \delta S_v$ if there exist index values $(i_1, \ldots, i_n)$ and $(j_1, \ldots, j_n)$ in Dom such that

1. $(i_1, \ldots, i_n) \leq (j_1, \ldots, j_n)$ where $\leq$ is the lexicographic order over Z_n.
2. statement $S_u(i_1, \ldots, i_n)$ must be executed before statement $S_v(j_1, \ldots, j_n)$ to preserve the semantics of the nest and we let $\delta = (j_1 - i_1, j_2 - i_2, \ldots, j_n - i_n)$ be the dependence vector between the two statements.

As pointed out in [23], dependences can be further divided into three categories (flow-, anti- or output-dependences). Also, given two statements S_u and S_v, there can be several index pairs (I, J) in Dom^2 for which $S_v(J)$ depends upon $S_u(I)$, hence several dependence vectors between S_u and S_v.

An important subclass of GLN is when all dependences between statements correspond to a fixed dependence vector, independent of the index pairs instanciating the dependence. Such nests of loops are termed *uniform*. Uniform nests of loops have a tremendous importance and have received considerable attention, and this for at least the following reasons:

- they arise naturally in many scientific applications,
- powerful mathematical tools can be used to deal with them, and
- all results for uniform nests can be applied to general nests if the conservative approach of retaining only the smallest dependence vector among all dependence vectors linking two statements is taken.

Consider the following example.

Example 5 *This is a slightly modified example from Polychronopoulos [23]:*

for $i = 0$ *to* N_1 *do*
for $j = 0$ *to* N_2 *do*
begin
{ *Statement* S_1 } $a(i,j) = b(i-3, j-5)$
{ *Statement* S_2 } $b(i,j) = a(i-2, j-4)$
end

In Example 5, the index set is a rectangle

$$Dom = \{(i,j) \in Z^2, 0 \leq i \leq N_1, 0 \leq j \leq N_2\}.$$

Variable $a(i,j)$ is produced in statement $S_1(i,j)$ and consumed in statement $S_2(i-2, j-4)$ so there is a dependence between S_1 and S_2 of vector $(2,4)$. Similarly, there is a dependence between S_2 and S_1 of vector $(3,5)$. Both dependences are uniform, so we let

$$D = \begin{pmatrix} d_1 & d_2 \end{pmatrix} = \begin{pmatrix} 2 & 3 \\ 4 & 5 \end{pmatrix}$$

represent all the dependences. Note that all dependence vectors are lexicographically positive (their first non-zero component is greater than 0), due to the condition (i). The dependence graph is represented in Figure 15.

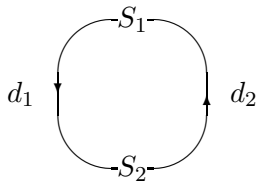

Figure 15: Dependence graph of Example 5

In this example, there is a dependence cycle, since there is a dependence between S_1 and S_2 and another one between S_2 and S_1. More generally, we say that there is a dependence cycle of length L if there are L different statements $S_{l_1}, S_{l_2}, \ldots, S_{l_L}$ such that $S_{l_1} \delta S_{l_2} \delta \ldots \delta S_{l_L} \delta S_{l_1}$. In our example the cycle is of length 2. Of course there can exist several cycles of different lengths involving the k statements S_1 to S_k.

In general, loops whose statements are involved in a dependence cycle are considered to be serial. However, techniques such as cycle shrinking can be used to extract parallelism that may still be present in the loop. We discuss such techniques in the next section.

Beforehand, we point out that most of the examples that we use hereafter will have the following simplified form (as the previous example):

RUN (Regular Uniform Nest) class

1. the index set is a parallelepiped

for $i_1 = 0$ to N_1 do
for $i_2 = 0$ to N_2 do
$\vdots$
for $i_n = 0$ to N_n do

```
begin
        { Statement S_1 }
                    ⋮
        { Statement S_k }
end
```

2. the dependences are uniform and all dependence vectors are recorded in the dependence matrix D of size $n \times m$, where m is the number of such vectors.

The restriction that the index domain is a parallelepiped is only for technical convenience. We will even use hypercubic domains where all N_i's are equal. Also, the fact that the loop indices range from 0 to N by increment 1 is no loss of generality. Such nests are called normalized by [23]. The main restriction is indeed that we deal with *uniform* nests.

13 Generalized Selective Cycle Shrinking

Generalized cycle shrinking has been introduced by Shang, O'Keefe and Fortes [35] as a generalization of a well-known technique in compiler optimization called selective shrinking [23]. Consider a GLN nest. The idea is to determine a vector $\Pi = (\pi_1, \ldots, \pi_n)$ such that all index points I, J in Dom such that $\Pi I = \Pi J$ can be executed simultaneously (assuming a target multiprocessor machine). Such a vector Π will be called a *scheduling vector*.

Of course not any vector can be used as a scheduling vector. The semantics of the loop has to be preserved. In [35], the following theorem is proven:

GSS theorem: Consider a GLN nest with uniform dependences, and let $D = (d_1, \ldots, d_m)$ be the $n \times m$ dependence matrix. Then any vector $\Pi = (\pi_1, \ldots, \pi_n)$ such that

$$\begin{array}{ll} (i) & \Pi D > 0 \\ (ii) & \gcd(\pi_1, \ldots, \pi_n) = 1 \end{array}$$

can be used as a scheduling vector. Any index I in Dom will be executed at time-step $\lfloor \frac{\Pi I}{\mathrm{disp}(\Pi)} \rfloor$, where $\mathrm{disp}(\Pi) = \min \{ \Pi d_j, 1 \leq j \leq m \}$ is the displacement.

The idea can be cast into the following terms: we group the index points into packets, and to execute the nest we have an outer sequential loop that represents time. All index points that are assigned the same time are executed concurrently. In other words, we have something like:

```
for time = min_time(Dom) to max_time(Dom) do
    execute all index points I in Dom such that
            ⌊ΠI / disp(Π)⌋ = time
```

Of course, min_time(Dom) and max_time(Dom) represent the smallest and largest values of the scheduling time for an index I in Dom.

Intuitively, condition (i) ensures that the semantics of the loop will be preserved. In other words the scheduling vector Π is safe if it forms an acute angle with the dependence vectors.

Also, we see that the scheduling time for an index point is homogeneous, hence the restriction that the components of Π be relatively prime.

Now, the problem is to determine the best scheduling vector, that is a vector Π that satisfies (i) and (ii) and for which max_time(Dom) - min_time(Dom) is minimum. This last condition is clearly equivalent to minimizing

$$\frac{\max\{\Pi I - \Pi J, I, J \in Dom\}}{\text{disp}(\Pi)}.$$

Shang, O'Keefe and Fortes [35] propose a procedure that is able to determine the best scheduling vector in some cases, including the RUN case. The reason why they call their approach Generalized Selective Shrinking is the following: assume that there is a cycle between the statements S_1 to S_k of the nest. A technique from compiler optimization known as selective shrinking [23] is to consider the rows of the dependence matrix D one after the other, and to select the first one such that all entries are positive. The smallest of these entries can safely be chosen as the displacement. We see that it amounts to try successively the basis vectors as possible scheduling vectors.

Another technique from compiler optimization is known as true dependence shrinking [22, 23]. It consists in computing the true distance for each dependence vector d_i, defined to be the number of loop iterations separating the instanciation of the corresponding dependent statements.

For a RUN, we get

$$\text{true\_distance}(d_i) = d_{n,i} + d_{n-1,i} * (N_n + 1) + d_{n-2,i} * (N_n + 1)(N_{n-1} + 1) + \ldots + d_{1,i} \prod_{j=2}^{n} (N_j + 1)$$

where

$$d_i = \begin{pmatrix} d_{1i} \\ d_{2i} \\ \vdots \\ d_{ni} \end{pmatrix}$$

and the displacement is the minimum of the true distances over all dependence vectors. This technique can be interpreted as proposing $\Pi = (\prod_{j=2}^{n}(N_j+1), \prod_{j=3}^{n}(N_j+1), \ldots, 1)$ for scheduling vector.

We conclude this section by stating the following problem.

GSS optimization problem: Find a vector Π with relatively prime components and such that $\Pi D > 0$ which minimizes

$$\text{GSS}(\Pi) = \frac{\max\{\Pi I - \Pi J, I, J \in Dom\}}{\text{disp}(\Pi)}.$$

Consider again Example 5. Let $\Pi = (a, b)$ be a scheduling vector. We have $\Pi d_1 = 2a + 4b, \Pi d_2 = 3a + 5b$, hence $\text{disp}(\Pi) = \min(2a + 4b, 3a + 5b)$ and max_time(Dom) - min_time(Dom) $= \mid a \mid N_1 + \mid b \mid N_2$. The problem is then to find two integers (a, b) such that the quantity

$$\frac{\mid a \mid N_1 + \mid b \mid N_2}{\min(2a + 4b, 3a + 5b)}$$

is minimized, subject to the constraints

$$\gcd(a,b) = 1, \quad 2a + 4b \geq 1, \quad 3a + 5b \geq 1.$$

Shang et al. use the method in [31] to find the best scheduling vector. They identify two possible solutions, $\Pi_1 = (1,0)$ and $\Pi_2 = (0,1)$. Π_1 leads to

$$\mathrm{GSS}(\Pi_1) = \frac{N_1}{2}$$

and Π_2 leads to

$$\mathrm{GSS}(\Pi_2) = \frac{N_2}{4}.$$

Depending upon the domain shape, either vector can be optimal.

14 The Index Shift Method

14.1 Description of the Index Shift Method

The Index Shift Method (ISM) has been introduced by Liu, Ho and Sheu [16]. Consider Example 6.

Example 6 *This example is from Peir and Cytron [22]:*

for $i = 0$ *to* N *do*
for $j = 0$ *to* N *do*
begin
{ *Statement* S_1 } $a(i,j) = b(i,j-6) + d(i-1,j+3)$
{ *Statement* S_2 } $b(i+1,j-1) = c(i+2,j+5)$
{ *Statement* S_3 } $c(i+3,j-1) = a(i,j-2)$
{ *Statement* S_4 } $d(i,j-1) = a(i,j-1)$
end

The dependences are the following:

$$\begin{aligned}
S_1 \longrightarrow S_3 : d_1 &= \begin{pmatrix} 0 \\ 2 \end{pmatrix} \\
S_3 \longrightarrow S_2 : d_2 &= \begin{pmatrix} 1 \\ -6 \end{pmatrix} \\
S_2 \longrightarrow S_1 : d_3 &= \begin{pmatrix} 1 \\ 5 \end{pmatrix} \\
S_1 \longrightarrow S_4 : d_4 &= \begin{pmatrix} 0 \\ 1 \end{pmatrix} \\
S_4 \longrightarrow S_1 : d_5 &= \begin{pmatrix} 1 \\ -4 \end{pmatrix}
\end{aligned}$$

We obtain the following dependence matrix:

$$D = \begin{pmatrix} 0 & 1 & 1 & 0 & 1 \\ 2 & -6 & 5 & 1 & -4 \end{pmatrix}.$$

The index domain is a square:

$$Dom = \{(i,j) \in Z^2, 0 \leq i,j \leq N\}.$$

There are two dependence cycles, namely $C_1 = (S_1, S_3, S_2)$ and $C_2 = (S_1, S_4)$.

Let us first compute the best scheduling vector with the GSS approach. We search for $\Pi = (a,b)$ such that

$$\begin{aligned} &\gcd(a,b) = 1 \\ &\Pi D > 0 \Longleftrightarrow \begin{cases} b \geq 1 \\ a \geq 6b+1 \end{cases} \end{aligned}$$

Note that a and b are necessarily positive. We get $\text{disp}(\Pi) = \min(2b, a-6b, a+5b, b, a-4b) = \min(a-6b, b)$ and we want to minimize the quantity

$$\text{GSS}(\Pi) = \frac{(a+b)N}{\text{disp}(\Pi)}.$$

The following straightforward case analysis shows that $\Pi = (7,1)$, for which $\text{GSS}(\Pi) = 8N$, is the best solution:

1. if $b = 1$, then minimize $\text{GSS}(\Pi) = (a+1)N$, hence take $a = 6b+1 = 7$
2. if $b \geq 2$, then consider two subcases:
   (a) if $6b+1 \leq a \leq 7b-1$, then $\text{disp}(\Pi) = a-6b$, and $\text{GSS}(\Pi) = \frac{a+b}{a-6b}N$, hence take $a = 7b-1$ and get $\text{GSS}(\Pi) = \frac{8b-1}{b-1}N \geq 8N$
   (b) if $7b+1 \leq a$, then $\text{disp}(\Pi) = b$, and $\text{GSS}(\Pi) = \frac{a+b}{b}N$, hence take $a = 7b+1$ and get $\text{GSS}(\Pi) = \frac{8b+1}{b}N \geq 8N$. (Note that $a = 7b$ is excluded because $\gcd(7b,b) = b \geq 2$.)

Next, draw a directed graph (Figure 16) whose nodes are the statements S_1 to S_4 and whose edges are weighted by the values $e_i = \Pi d_i$.

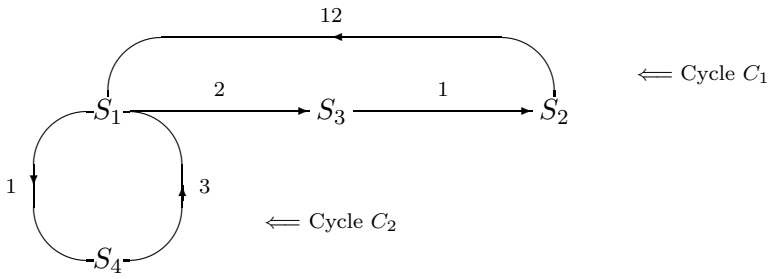

Figure 16: Dependence cycles of Example 6

The idea of the Index Shift Method is to apply some retiming to this graph. Indeed, consider cycle C_2 with the edge $e_{14} = (S_1, S_4)$ weighted $w_{14} = 1$ and the reverse edge $e_{41} = (S_4, S_1)$

weighted $w_{41} = 3$. Since disp(Π) is the minimum of the weights of the edges in the graph, we would like to remove 1 from w_{41} and to add 1 to w_{14}. Similarly, $w_{21} = 12$ and we can shift part of this weight to w_{13} and w_{32}. Such shifts can be conducted so that the minimum weight of an edge in the graph becomes 2.

Systolicians will have recognized the retiming procedure of Leiserson and Saxe [15]. The idea is to modify weights by adding some given quantity (possibly negative) to all incoming edges to a given node, and to subtract the same quantity from all outgoing edges from that node. Obviously, such transformations do not modify the total weight of each cycle. Hence, if the goal is to achieve a good balance among all weights involved in a cycle of length k and total weight T, the best you can do is to assign the weight $\lfloor T/k \rfloor + 1$ to (T mod k) edges and the weight $\lfloor T/k \rfloor$ to the remaining ones.

Liu et al. [16] provide a method for determining the transformations needed to achieve this balancing of the weights among all cycles. The method is the same as in [26, page 252], and it is simpler than the one originally proposed in [15]. Let us denote by S_i^k the transformation that consists of adding k to the weights of all incoming edges to node S_i, and subtracting k from the weights of all outgoing edges from S_i. In our example, the (commutative) sequence $S_2^1 S_4^1$ leads to the graph of Figure 17.

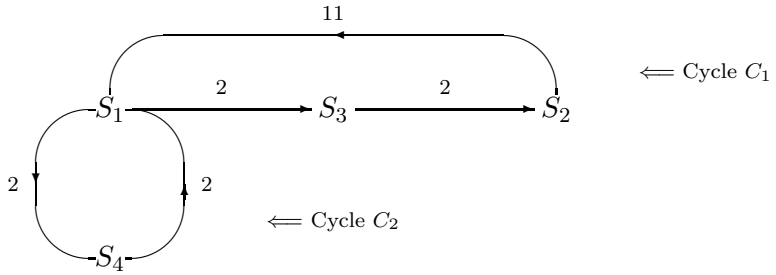

Figure 17: New weights obtained by $S_2^1 S_4^1$

The minimum weight is now 2, which means that the new displacement is 2.

What does it mean to apply a transformation S_i^k to the graph for the nest of loops, and how the range of the loop indices is affected? First of all, applying the transformation S_i^k amounts to replacing the execution of a given instance $S_i(J_1)$ of statement S_i by the execution of another instance $S_i(J_2)$ such that the scheduling difference between the two instances is equal to k. In other words, J_2 is such that

$$\Pi(J_1 - J_2) = k.$$

Liu et al. [16] have a very nice way of determining J_2: let u be a vector such that $\Pi u = 1$ (remember that Π has relatively prime components). Then $J_2 = J_1 - ku$ satisfies $\Pi(J_1 - J_2) = k\Pi u = k$. Moreover, the vector u enables us to compute the new bounds for the loop indices. In the example let $u = (0, 1)$. Then the transformation S_2^1 amounts to shifting index j by one unit in statement S_2. Hence the second loop index j will loop from 1 to $N + 1$ in statement S_2. But as it moves from 0 to N in statement S_1, the total range is extended from 0 to $N + 1$. Some iteration instances will be meaningless for some statements: in our example this is the case when $j = N + 1$ for statements S_1 and S_3 or when $j = 0$ for statements S_2 and S_4. Applying this process, we derive the new loop nest:

for $i = 0$ to N do
for $j = 0$ to $N + 1$ do

begin
$\{S_1\}$: $a(i,j) = b(i,j-6) + d(i-1,j+3)$
$\{S_2\}$: $b(i+1,j-2) = c(i+2,j+4)$
$\{S_3\}$: $c(i+3,j-1) = a(i,j-2)$
$\{S_4\}$: $d(i,j-2) = a(i,j-2)$
end

Suppose we were given this nest from the beginning: we would have computed the dependence matrix as

$$D = \begin{pmatrix} 0 & 1 & 1 & 0 & 1 \\ 2 & -5 & 4 & 2 & -5 \end{pmatrix}$$

We check that $\Pi = (7,1)$ is a scheduling vector with displacement $\text{disp}(\Pi) = 2$, hence we can achieve

$$\text{GSS}(\Pi) = \frac{7N + (N+1)}{2} \leq 4N + 1,$$

nearly halving the execution time derived with the GSS approach.

We can now state more formally the ISM technique, as described by Liu et al [16]:

Index Shift Method: Given a scheduling vector Π with relatively prime components such that $\Pi D > 0$, let $Cycl$ denote the set of all cycles in the dependence graph where nodes are the statements S_1 to S_k and each edge (S_i, S_j) with dependence vector d_{ij} is weighted by the value Πd_{ij}. For a cycle C in $Cycl$, let $T(C)$ denote its total weight and $k(C)$ denote its length.

The ISM technique enables us to replace the value of the reduction factor

$$\lambda_1 = \text{disp}(\Pi) = \min\{\Pi d_i, 1 \leq i \leq m\}$$

by the value

$$\lambda_2 = \min\{\lfloor T(C)/k(C) \rfloor, C \in Cycl\}.$$

Since the transformation is at the price of an increase in the range of the loop index that depends only on Π and not on the domain size, the improvement factor is very close to the ratio $\frac{\lambda_2}{\lambda_1}$.

14.2 An improvement of the Index Shift Method

Consider again Example 5. Remember that depending upon the shape of the domain, the GSS approach will determine either $\Pi_1 = (1,0)$ or $\Pi_2 = (0,1)$. Assume we want to use the ISM. For Π_1 we obtain the dependence graph of Figure 18(a), while similarly for Π_2 we get the graph of Figure 18(b).

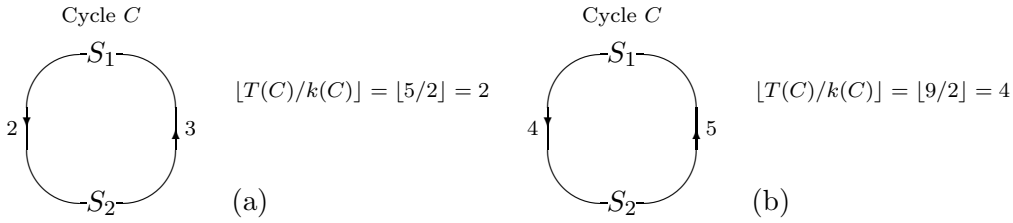

Figure 18: Dependence graph obtained by (a) Π_1 and (b) Π_2

We see that there is no possible improvement. However, using Π_1 say, we would like to replace $\text{disp}(\Pi_1) = 2$ by the real-valued average of the weights over the cycle, that is 2.5. The goal of this section is to show how we can deduce from Π_1 another scheduling vector Π_1^* that will achieve a reduction factor of 2.5.

In the general case, suppose we are given a cycle C of k statements S_1 to S_k with dependence matrix $D = (d_1, ..., d_k)$. Here vector d_j represents the dependence between statements S_j and $S_{j+1 \bmod k}$. Suppose we are also given a scheduling vector Π with relatively prime components and such that $\Pi D > 0$. Finally, suppose that the total weight of the cycle $T(C) = \Pi d_1 + ... + \Pi d_k$ is not divisible by k.

Of course, scaling Π by multiplying all its components by k would render $T(C)$ divisible by k, but would also violate the constraint of relatively prime components. So the idea is indeed to perform some scaling, but to use some perturbation afterwards to retrieve relatively prime components while satisfying the constraint $\Pi D > 0$.

Let r be a vector with relatively prime components and such that $\Pi r = 0$, and let s be a vector such that $rs = \pm 1$ with

$$sd_1 + ... + sd_k \geq 0.$$

Finding s from r is easy, and finding r from Π is done as follows. Compute the Hermite form of Π to get

$$\Pi = Q \begin{pmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix},$$

where Q is unimodular and let r be the second row of Q^{-1}: r has relatively prime components because Q^{-1} is unimodular, and $r\Pi = 0$ by construction, since

$$\begin{pmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix} = Q^{-1}\Pi.$$

Now, let $\Pi^* = \lambda k \Pi + s$. The weight $T^*(C)$ of cycle C with respect to Π^* is

$$T^*(C) = \lambda k T(C) + s(d_1 + .. + d_k) \geq \lambda k T(C)$$

hence $\lfloor T^*(C)/k \rfloor \geq \lambda T(C)$.

Now $r\Pi^* = \pm 1$, hence Π^* has relatively prime components. For λ large enough, $\Pi^* D > 0$, and after application of the ISM method we do achieve a reduction factor arbitrarily close to the real value of $T(C)/k$.

Rather than going on formally, let us apply the method to Example 5. Let us take $\Pi_1 = (1, 0)$. Then we have trivially

$$Q = \text{identity matrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

and $r = s = (0, 1)$.

With $\Pi_1^* = 2\lambda\Pi_1 + s = (2\lambda, 1)$, we obtain the dependence graph of Figure 19 (a).

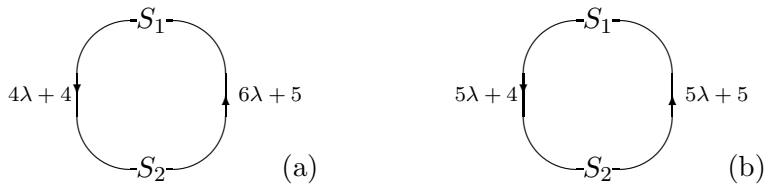

Figure 19: Dependence graph obtained (a) by Π_1^* and (b) after S_2^λ

The transformation S_2^λ produces the dependence graph of Figure 19 (b). Using $u = (0, 1)$, the second index j will loop from 0 to $N_2 + \lambda$. The resulting time will therefore be

$$\text{Time}(\Pi_1^*) = \frac{2\lambda N_1 + (N_2 + \lambda)}{5\lambda + 4} \leq \frac{N_1}{2.5} + 1$$

for large λ.

Similarly, if we take $\Pi_2 = (0, 1)$, we can obtain a Π_2^* that can achieve

$$\text{Time}(\Pi_2^*) \leq \frac{N_1}{4.5} + 1.$$

To conclude this section, we note that vector r has been introduced only to ease the presentation. In fact, computing Q^{-1} is needless: we just let s be the second column of Q (or its opposite).

15 Combining both methods

There is no reason a priori for the application of the ISM method or its refinement to the best GSS scheduling vector to give the best execution time. To see this, we return to Example 6.

15.1 GSS followed by ISM is not enough

We know that the best GSS scheduling vector is $\Pi = (7, 1)$ and for such a Π we had the dependence graph of Figure 16. Since for the cycle C_2 the total weight 4 is divisible by the length 2, we cannot improve on the original ISM method.

But why not try ISM with other scheduling vectors? Let $\Pi = (a, b)$ be an arbitrary scheduling vector. Hence $b \geq 1$ and $a \geq 6b + 1$, as seen earlier.

For cycle C_1 we get $T(C_1) = \Pi(d_1 + d_2 + d_3) = 2a + b$ and $k(C_1) = 3$. For cycle C_2 we get $T(C_2) = \Pi(d_4 + d_5) = a - 3b$ and $k(C_2) = 2$. For $b \geq 1$ and $a \geq 6b + 1$, we always have $(2a + b)/3 \geq (a - 3b)/2$, hence we want to minimize

$$\frac{(a + b)N}{\lfloor (a - 3b)/2 \rfloor}.$$

We can have this quantity as close to $2N$ as we want by letting $a = 2c + 1$ and $b = 1$ with c large (necessarily, $c \geq 3$). With $\Pi = (2c + 1, 1)$ we get the dependence graph of Figure 20.

By applying the transformation S_1^{2-c}, we get the dependence graph of Figure 21. Usage of $u = (0, 1)$ gives

$$\text{Time } \leq \frac{(2c + 1)N + (N + 2 - c)}{c - 1} \leq 2\frac{c + 1}{c - 1}N.$$

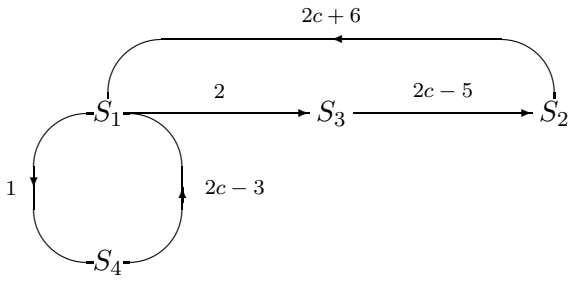

Figure 20: Dependence cycles obtained by $\Pi = (2c + 1, 1)$

We conclude that time is close to $2N$, obtaining thus a factor 2 over GSS+ISM.

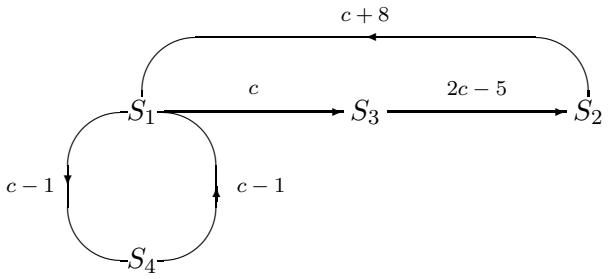

Figure 21: Dependence cycles obtained after S_1^{2-c}

15.2 A new optimization method

Consider a RUN with an index set *Dom* and a dependence matrix

$$D = \begin{pmatrix} d_1 & \dots & d_m \end{pmatrix}$$

where

$$d_i = \begin{pmatrix} d_{1i} \\ d_{2i} \\ \vdots \\ d_{ni} \end{pmatrix}$$

Let *Cycl* denote the set of all cycles between the statements S_1 to S_k. For each cycle C in *Cycl*, let $k(C)$ denote its length.

The optimization problem can be stated as follows:

New Optimization Problem: Find a vector $\Pi = (\pi_1, \dots, \pi_n)$ such that

$$\begin{array}{ll} (i) & \Pi D > 0 \\ (ii) & \gcd(\pi_1, \dots, \pi_n) = 1 \end{array}$$

and which minimizes

$$\mathrm{NEW}(\Pi) = \frac{\max\{\Pi I - \Pi J, I, J \in Dom\}}{\mathrm{cycle}(\Pi)}$$

where

$$\text{cycle}(\Pi) = \min\{\lfloor \frac{T_\Pi(C)}{k(C)} \rfloor, C \in Cycl\}$$

with

$$T_\Pi(C) = \sum_{d \in C} \Pi d$$

Note that dependence vectors that are not involved in any cycle need not be taken into account for the computation of cycle(Π) (they still appear in condition (i)). This is clear from the ISM method, shifts can be freely applied to the corresponding edges in the dependence graph.

Note also that the problem is not scalable: due to the floor function, NEW(Π) ≠ NEW(αΠ) for a nonzero constant α. However, our optimization problem is very close to the GSS problem. In [33], Shang and Fortes show how to solve GSS for a RUN or for a larger class of problems where the convex hull of the index set is a non-degenerate polyhedron. The idea is to partition the solution space into convex subcones, and to solve a linear fractional problem for each of these subcones. Two alternatives are proposed: either use a general linear programming approach or benefit from a less expensive method derived by the authors.

We can use the results of [33] as follows: we first remove condition (ii) that expresses that a scheduling vector must have relatively prime components. The problem can be expressed without floor functions and solved using Shang and Fortes' results. Finally, we perform some perturbation on the solution vector to retrieve condition (ii), as explained in section 14.2.

15.3 Quantifying the improvement factor

In this section, we show that our new method can outperform GSS followed by ISM by an arbitrary factor. Let λ be a positive integer, and consider the following example:

Example 7 *Consider the nested loop:*

for $i = 0$ *to* N *do*
for $j = 0$ *to* N *do*
begin
{ *Statement* S_1 } $a(i,j) = f_1(b(i-1, j-\lambda), c(i-1, j+\lambda), \ldots)$
{ *Statement* S_2 } $b(i,j) = f_2(a(i, j-1), \ldots)$
{ *Statement* S_3 } $c(i,j) = f_3(a(i, j-1), \ldots)$
end

The dependences are the following:

$$S_1 \longrightarrow S_2 : d_1 = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

$$S_2 \longrightarrow S_1 : d_2 = \begin{pmatrix} 1 \\ \lambda \end{pmatrix}$$

$$S_1 \longrightarrow S_3 : d_3 = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

$$S_3 \longrightarrow S_1 : d_4 = \begin{pmatrix} 1 \\ -\lambda \end{pmatrix}$$

We obtain the following dependence matrix:

$$D = \begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & \lambda & 1 & -\lambda \end{pmatrix}.$$

The index domain is a square:

$$Dom = \{(i,j) \in Z^2, 0 \leq i,j \leq N\}.$$

There are two dependence cycles, namely $C_1 = (S_1, S_2)$ and $C_2 = (S_1, S_3)$.

Let us first compute the best scheduling vector with the GSS approach. We search for $\Pi = (a,b)$ such that

$$\text{(i) } \gcd(a,b) = 1$$
$$\text{(ii) } \Pi D > 0 \Longleftrightarrow \begin{cases} b \geq 1 \\ a \geq \lambda b + 1 \end{cases}$$

Note that a and b are necessarily positive. We get $\text{disp}(\Pi) = \min(b, a-\lambda b, a+\lambda b) = \min(a-\lambda b, b)$ and we want to minimize the quantity

$$\text{GSS}(\Pi) = \frac{(a+b)N}{\text{disp}(\Pi)}.$$

The following straightforward case analysis shows that $\Pi = (\lambda+1, 1)$, for which $\text{GSS}(\Pi) = (\lambda+2)N$, is the best solution:

1. if $b = 1$, then minimize $\text{GSS}(\Pi) = (a+1)N$, hence take $a = \lambda + 1$
2. if $b \geq 2$, then consider two subcases:
   (a) if $\lambda b + 1 \leq a \leq (\lambda+1)b - 1$, then $\text{disp}(\Pi) = a - \lambda b$, and $\text{GSS}(\Pi) = \frac{a+b}{a-\lambda b}N$, hence take $a = (\lambda+1)b - 1$ and get $\text{GSS}(\Pi) = \frac{(\lambda+2)b-1}{b-1}N \geq (\lambda+2)N$
   (b) if $(\lambda+1)b+1 \leq a$, then $\text{disp}(\Pi) = b$, and $\text{GSS}(\Pi) = \frac{a+b}{b}N$, hence take $a = (\lambda+1)b+1$ and get $\text{GSS}(\Pi) = \frac{(\lambda+2)b+1}{b}N \geq (\lambda+2)N$. (Note that $a = (\lambda+1)b$ is excluded because $\gcd((\lambda+1)b, b) = b \geq 2$.)

Next, draw a directed graph (Figure 22) whose nodes are the statements S_1 to S_3 and whose edges are weighted by the values $e_i = \Pi d_i$.

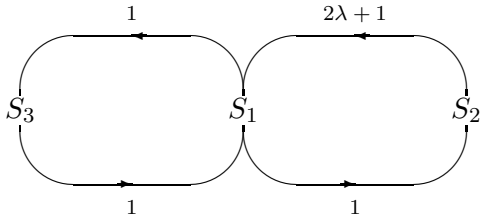

Figure 22: Dependence cycles of Example 7 with GSS

Because of cycle C_2, there is no possible improvement using ISM. However, if we solve the new optimization problem, we have to search for $\Pi = (a, b)$ such that (i) and (ii) hold (note that a and b are necessarily positive). We get $\text{cycle}(\Pi) = \min(\lfloor \frac{b+(a+\lambda b)}{2} \rfloor, \lfloor \frac{b+(a-\lambda b)}{2} \rfloor) = \lfloor \frac{a+(1-\lambda)b}{2} \rfloor$, and we want to minimize the quantity

$$\text{NEW}(\Pi) = \frac{(a+b)N}{\text{cycle}(\Pi)}.$$

We let $\lambda = 2\gamma - 1$ and we take $\Pi = (a, b) = (4\gamma, 1)$ as a scheduling vector (we check that $a \geq \lambda b + 1$, i.e., $4\gamma \geq (2\lambda - 1).1 + 1$). We obtain the new dependence graph shown in Figure 23.

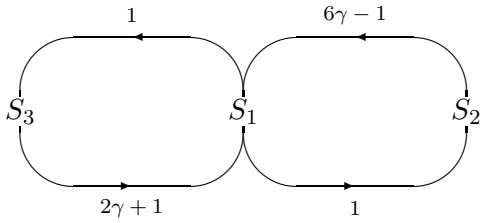

Figure 23: Dependence cycles with $\Pi = (4\gamma, 1)$

We have $\text{cycle}(\Pi) = \gamma + 1$. Next we perform the transformation S_2^γ S_3^γ (see Figure 24).

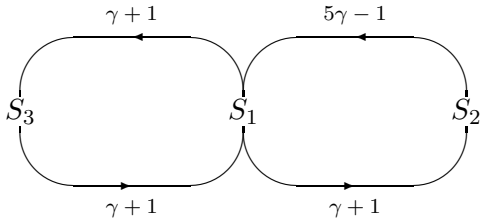

Figure 24: Dependence cycles obtained after S_2^γ S_3^γ

We use vector $u = (0, 1)$ such that $\Pi u = 1$ to compute the new index bounds:

$$\begin{array}{l} 0 \leq i \leq N \\ \left\{ \begin{array}{ll} 0 \leq j \leq N & \text{(statement } S_1\text{)} \\ \gamma \leq j \leq \gamma + N & \text{(statements } S_2 \text{ and } S_3\text{)} \end{array} \right. \end{array}$$

The new domain is then

$$Dom = \{(i, j) \in Z^2, 0 \leq i \leq N, 0 \leq j \leq N + \gamma\}.$$

We get

$$\text{NEW}(\Pi) = \frac{4\gamma N + (N + \gamma)}{\gamma + 1} \leq 4N + 1.$$

Hence we gain a factor close to $\frac{\lambda+2}{4} = O(\lambda)$ over GSS+ISM, and this factor can be arbitrary high, as was claimed.

16 Conclusion

Systolic algorithms have been conceived in an "ad hoc" manner, requiring a great amount of effort. We have seen that some of the systolic algorithms proposed in the literature for matrix multiplication are merely special cases of a synthesis method. Synthesis methods were proposed by Fortes and Moldovan, Robert and Quinton and some other researchers. The geometric interpretation used in the previous sections has the purpose of allowing a better understanding of the proposed methods. We have been deliberately informal in the first part. We then presented a more formal treatment, based mainly on the work by Quinton and Robert [26]. Finally we have discussed the evolution of systolic computing along another direction. Several results from the area of systolic computation have been used in other areas, namely that of a parallelizing compiler. We have shown such an application, in the context of cycle shrinking of nested loops.

Several loop transformations techniques have been designed to extract parallelism from nested loop structures. We have reviewed two important approaches, known as *Generalized Cycle Shrinking* presented by Shang, O'Keefe and Fortes and *the Index Shift Method* introduced by Liu, Ho and Sheu. We have used an illustrative example (Example 6) to show the gains that can be obtained. The quadratic sequential time is reduced to $8N$ through GSS and to $4N$ through GSS followed by ISM. One result of the paper is an improvement of the index shift method, for a given scheduling vector Π. The main result of the paper is a new methodology that permits to combine cycle shrinking techniques with the index shift method. Through this combination, the illustrative example renders a further factor of 2, giving a time of $2N$. The combination of the two techniques gave rise to a new optimization method that produces the best scheduling vector.

References

[1] Banerjee, U. "An introduction to a formal theory of dependence analysis". *The Journal of Supercomputing 2*, 1988, pp. 133 - 149.

[2] Cappello, P. R. and Kenneth, S. "Unifying VLSI array design with linear transformations of space-time". *Advances in Computing Research*, vol. 2, p. 23-65, 1984.

[3] Cosnard, M., Quinton, P., Robert, Y. and Tchuente M. (editors) *Parallel Algorithms and Architectures*. North Holland, 1986.

[4] Delosme, J. M. and Ipsen, I. C. F. "Systolic array synthesis: computability and time cones", in: *Parallel Algorithms and Architectures*, M. Cosnard et al. (editors), Elsevier Science Publishing, North Holland, p. 295-312, 1986.

[5] Dowling, M. L. "Optimal code parallelization using unimodular transformations". *Parallel Computing*, 16, 1990, pp. 157 - 171.

[6] Fortes, J. A. B. *Algorithm Transformations for Parallel Processing and VLSI Architecture Design*. Ph.D. thesis, Department of Electrical Engineering-Systems, University of Southern California, December, 1983.

[7] Fortes, J. A. B. and Moldovan, D. I. "Parallelism detection and transformation techniques useful for VLSI algorithms". *Journal of Parallel and Distributed Computing* 2, p. 277-301, 1985.

[8] Foster, M. J. and Kung, H. T. "The design of special-purpose VLSI chips". *Computer*, 13, p. 26-40, 1980.

[9] Huang, C. H. and Lengauer, C. "The derivation of systolic implementations of programs". *Acta Informatica*, 24, p. 595-632, 1987.

[10] Karp, R. M., Miller, R. E. and Winograd, S. "The organization of computations for uniform recurrence equations". *Journal of the ACM*, 14, p. 563-590, 1967.

[11] Kung, H. T. "Let's design algorithms for VLSI systems". *Proceedings of Caltech Conference on VLSI*, p. 65-90, January, 1979.

[12] Kung, H. T. and Leiserson, C. E. "Systolic arrays for VLSI", in:*Introduction to VLSI Systems*, C. A. Mead and L. A. Conway, Chapter 8.3, Addison-Wesley, 1980.

[13] Kung, H. T. "The structure of parallel algorithms". *Advances in Computing*, 19, p. 65-111, 1980.

[14] Kung, H. T. "Why systolic architectures". *Computer*, 15, p. 37-46, 1982.

[15] Leiserson, C. E. and Saxe, J. B. "Optimizing synchronous systems". *Journal of VLSI Computer Systems*, Vol. 1, April, 1983, pp. 41 - 67.

[16] Liu, L. S., Ho, C. W. and Sheu, J. P. "On the parallelism of nested for-loops using index shift method". Proceedings of International Conference on Parallel Processing, August 1990, pp. II-119 - II-123.

[17] Miranker, W. L. and Winkler A. "Spacetime representation of computational structures". *Computing* 32, p. 93-114, 1984.

[18] Moldovan, D. I. "On the design of algorithms for VLSI systolic arrays". Proceedings of the IEEE, vol. 71, no. 1, January, p. 113-120, 1983.

[19] Moldovan, D. I. "ADVIS: a software package for the design of systolic arrays". *IEEE Transactions on Computer-Aided Design*, CAD-6, p. 33-40, January, 1987.

[20] Nelis, H. W. and Deprettere, E. F. "Automatic Design and partitioning of systolic/wavefront arrays for VLSI". *Circuit System Signal Processing* , Vol. 7, number 2, p.235-252, 1988.

[21] Okuda, Kunio and Song, Siang W. "Um Algoritmo de Multiplicação de Matrizes para implementação em VLSI". Anais do I Congresso da Sociedade Brasileira de Microeletrônica, Campinas, pp. 383-393, 1986.

[22] Peir, J. K. and Cytron, R. "Minimum distance: a method for partitioning recurrences for multiprocessors". *IEEE Transactions on Computers*, Vol. 38, No. 8, August 1989, pp. 1203 - 1211.

[23] Polychronopoulos, C. D. "Compiler optimization for enhancing parallelism and their impact on architecture design". *IEEE Transactions on Computers*, Vol. 37, No. 8, August 1988, pp. 991 - 1004.

[24] Polychronopoulos, C. D. *Parallel Programming and Compilers*. Kluwer Academic Publishers, Boston, 1988.

[25] Quinton, P. "The systematic design of systolic arrays", in:*Automata Networks in Computer Science*, F. Fogelman, Y. Robert and M. Tchuente (editors) , Manchester University Press, p. 229-260, 1987.

[26] Quinton, P. and Robert, Y. *Algorithmes et architectures systoliques*. Masson, Paris, 1989.

[27] Rajopadhye, S. V. and Fujimoto, R. M. "Systolic array synthesis by static analysis of program dependencies", in: *Parallel Architectures and Languages Europe*, J. W. Baker et al. (editors), Springer-Verlag, p. 295-315, 1987.

[28] Ribas, H. B. "Automatic Generation of systolic programs from nested loops", Ph.D. thesis, Department of Electrical and Computer Engineering, Carnegie Mellon University, June, 1990.

[29] Robert, Y. "Systolic algorithms and architectures", in:*Automata Networks in Computer Science*, F. Fogelman, Y. Robert and M. Tchuente (editors) , Manchester University Press, p. 187-228, 1987.

[30] Robert, Yves and Song, Siang W. "Revisiting Cycle Shrinking". *Parallel Computing*, Vol. 18, Number 5, May 1992, pp. 481-496.

[31] Shang, W. and Fortes, J. A. B. "Time optimal linear schedules for algorithms with uniform dependences". *Proceedings of International Conference on Systolic Arrays*, May 1988, pp. 393 - 402.

[32] Shang, W. and Fortes, J. A. B. "Time optimal and conflict-free mappings for uniform dependence algorithms into lower dimensional processor arrays". Proceedings of International Conference on Parallel Processing, August 1990, pp. I-101 - I-110.

[33] Shang, W. and Fortes, J. A. B. "Time optimal linear schedules for algorithms with uniform dependencies". *IEEE Transactions on Computers*, Vol. 40, No. 6, June 1991, pp. 723-742.

[34] Shang, W., O'Keefe, M. T. and Fortes, J. A. B. "On loop transformations for generalized cycle shrinking". Proceedings of International Conference on Parallel Processing, August 1991, pp. II-132 - II-141.

[35] Shang, W., O'Keefe, M. T. and Fortes, J. A. B. "Generalized cycle shrinking", in: *Parallel Algorithms and VLSI Architectures II*, P. Quinton and Y. Robert (editors), North Holland, 1991.

[36] Song, S. W. *Algoritmos Paralelos e Arquitetura VLSI*. São Paulo, 1984.

[37] Song, S. W. "Método de síntese de algoritmos sistólicos: uma interpretação geométrica". Anais da Jornada EPUSP/IEEE sobre Sistemas de Computação de Alto Desempenho, São Paulo, março, 1991, pp. 165-176.

[38] Weiser, U. and Davis, A. "A wavefront notation tool for VLSI array design", in: *VLSI Systems and Computations*, H. T. Kung et al. (editors), p. 226-234, Computer Science Press, 1981.

[39] Wolfe, M. *Optimizing Supercompilers for Supercomputers*. MIT Press, Cambridge MA, 1989.

[40] Wolfe, M. "Data dependence and program restructuring". *The Journal of Supercomputing 4*, 1990, pp. 321 - 344.

[41] Young, D. M. *Iterative Solution of Large Linear Systems*. Academic Press, 1971.