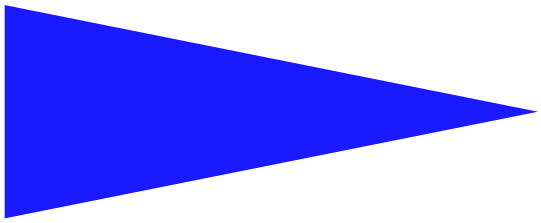
I R I S A
INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES

PUBLICATION
INTERNE
N° 621



# MEMORY-EFFICIENT DATA STRUCTURES FOR SYSTOLIC PROGRAMS

ZBIGNIEW CHAMSKI

ISSN 1166-8687

IRISA
CAMPUS UNIVERSITAIRE DE BEAULIEU - 35042 RENNES CEDEX - FRANCE

IRISA
INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES
Campus de Beaulieu – 35042 Rennes Cedex – France
Tél. : (33) 99 84 71 00 – Fax : (33) 99 38 38 32

# Memory-efficient data structures for systolic programs

Zbigniew Chamski

Programme 1 — Architectures parallèles, bases de données, réseaux et systèmes distribués
Projet API

Publication interne n ° 621 — Décembre 1991 — 22 pages

**Abstract:** Generating imperative programs from systolic specifications implies the use of a memory-efficient model for representing data used in the original, single-assignment problem. We present a new method of generating data structures, designed for use in systolic program compilers for distributed-memory parallel computers. Our method is based on the mathematical properties of systolic programs and consists of a set of algebraically defined elementary transformations. Using these transformations, the total size of data structures can be optimized, allowing very memory-efficient code to be produced. The method has been implemented as a part of an experimental compiler of the ALPHA language, currently under development.
**Key-words:** parallel programming, systolic arrays, distributed memory architectures

*(Résumé : tsvp)*

Centre National de la Recherche Scientifique
(URA 227) Université de Rennes 1 – Insa de Rennes

INRIA
Institut National de Recherche en Informatique
et en Automatique – unité de recherche de Rennes

# Generating memory-efficient imperative data structures from systolic programs

**Résumé :** Le génération de programmes impératifs à partir de spécifications systoliques exige un modèle de mémorisation efficace pour représenter les données utilisées dans la version à assignation unique du programme source. Nous décrivons une nouvelle méthode pour la génération de structures de données, destinée à la compilation de programmes systoliques pour des architectures parallèles à mémoire distribuée. Notre méthode est basée sur les propriétés mathématiques des programmes systoliques, et consiste en un ensemble de transformations algébriques élémentaires. Avec ces transformations, la taille totale des structures de données peut être optimisée, ce qui permet de générer un code très efficace en taille mémoire. La méthode a été implémentée dans un compilateur expérimental pour le language ALPHA, actuellement en cours de développement.
**Mots-clé :** programmation parallèle, architectures systoliques, architectures à mémoire distribuée

# 1 Introduction

Since its introduction in 1982 ([Kung82],) the concept of systolic array commands a still growing interest. The research carried out into this topic at Irisa ([Quin83], [Robe86], [QuVa88], [GMQS89], [Maur89], [MQRS90], [CLMQ90]) lead to the design of a declarative language for systolic array description, called ALPHA, and of the corresponding program transformation environment. The programming environment, called ALPHA DU CENTAUR ([GMQS89],) is built on the top of the Centaur system ([Borr88]) and inherits its expandability, allowing new program transformations to be easily added. In its current form, this environment is basically aimed at the design of dedicated VLSI chips.

A large availability of "conventional" parallel computers suggested to use ALPHA as the input language for a parallel program generator. Programmed this way, a distributed-memory parallel computer can be used to validate systolic architectures and to experiment with various parallel algorithms that can be obtained from a given specification. Moreover, by carefully designing the compiler it is also possible to use it as a performance evaluation tool, for both running time and memory utilization. The iPSC/2 hypercube from Intel installed at Irisa appeared as a suitable target for code generation, thus leading to the investigation of a real compiler, intended as a part of the ALPHA DU CENTAUR environment.

Choosing ALPHA as input language gives numerous advantages: given the underlying mathematical model — affine recurrence equations, or AREs — the dependencies are local and regular, the input programs are single-assignment, and the partitioning is described in a very natural way. As we consider only non-parametrized programs ([MQRS90],) that is, programs whose size is statically defined, the set of dependencies is entirely defined at compile time. Moreover, *all* dependence functions are affine, allowing simple algebraic methods to be used in the compiling process.

The generation of data structures for the target program is a challenging problem: the systolic programs used as input are single-assignment, thus requiring an excessive amount of memory and dramatically restricting the maximum size of treatable problems if they are directly rewritten into single-assignment imperative programs. Hence, the data structure generator must carry out two activities: define the arrays corresponding to every local variable of the input program, and reduce the size of these arrays while preserving the correctness of the target program. We chose the memory usage as the only quantitative criterion of the optimization process because the expected improvements can reach several orders of magnitude.

Optimizing data structures can be carried out in two complementary ways: one can only transform ("reduce") the unoptimized array associated with a given ALPHA variable, or search for a set of applicable source-to-source transformations of this variable, then reduce the arrays associated with every variable obtained by means of these transformations. These two approaches can be combined

and an analysis of the dependence functions allows applicable transformations to be computed using a constructive method.

This paper is organized as follows: section 2 contains vocabulary and hypotheses. Section 3 introduces the problems appearing in data structure generation on an ALPHA program example used throughout the paper. Section 4 gives a simple means of generating unoptimized data structures. Section 5 then presents the concept of "variable reduction", allowing a direct optimization of arrays computed using the method from section 4. Section 6 defines two source-to-source transformations of ALPHA programs useful in optimizing variable images. Finally, we give an overview of the current status of the data structure generator and a comparison of our results with previous research.

### Notations

Let $\mathbf{N}$ denote the set of nonnegative integers, $\mathbf{Z}$ denote the set of relative integers and $\mathbf{Z}/p\mathbf{Z}$ denote the set of equivalence classes modulo $p$ of elements of $\mathbf{Z}$. By convention, $\mathbf{Z}/0\mathbf{Z} = \mathbf{Z}$.

Let *null(v)* denote the position of the last null coordinate of a vector $v$ in the current basis.

## 2 Vocabulary and hypotheses

### 2.1 Vocabulary

As this report simultaneously addresses issues related to systolic arrays and to imperative parallel programming, it seems useful to define a unique and non-ambiguous vocabulary for the rest of the paper.

#### 2.1.1 ALPHA language

A *domain* (also called "spatial domain") is a finite convex polyhedron of $\mathbf{Z}^n$ defined by a finite set of affine inequalities (constraints). The notation is `{ <indices> | <constraints> }`.

A *variable* is a morphism of a domain into a set of values. This set is sometimes referred to as "domain of values", as opposed to "spatial domain". There are three basic sets of values: integers, reals and booleans. By extension, the type of a variable is the type of its values.

An *instance* $A(z)$ of a variable $A$ is its value at a given point $z$ of the corresponding spatial domain.

A *dependence function* is an affine morphism of a spatial domain into another. The expression $A.(z \rightarrow D(z))$ takes at point $z$ the value of $A(D(z))$. We say that there is a dependence $D$ on $A$.

An *equation* defines the relations between instances of (possibly different) variables. An equation consists of a list of variable names, and of an expression defining the values of instances of left-hand side (LHS) variables at different

points of their domain. The right-hand side (RHS) expression can depend on the point of the LHS domain, meaning a *restriction* of the corresponding subexpression to a subdomain. This is expressed using the `case` construct (see the example programs in the next section.)

A variable is called *input variable* if it appears only in RHS expressions. A *local variable* appears both on LHS and on RHS of equations (possibly distinct.) An *output variable* appears only on LHS of an equation.

A *timing function* is an affine function from a domain into a totally ordered set of integer points. The image of a point by a timing function is called *instant* of execution. An ALPHA program is said to be *computable* by a given timing function if for every definition-use pair the instant of definition precedes the instant of use.

### 2.1.2 Imperative parallel programming

A *loop* is an iterative instruction, totally defined by giving a *loop index*, a *lower bound*, an *upper bound*, and a sequence of instructions called *loop body*. The instructions of the loop body are executed for all successive integer values of the loop index from the lower bound up to the upper bound.

The loop body can contain other loops. Two loops belonging to the same loop body have disjoint index sets. Consider the following Pascal example:

```
for i := 0 to m do
  begin
    ...
    for j := 1 to i do
      A[i] := i+j ;
    ...
    for j := i+1 to n do
      B[j] := j-i ;
    ...
  end ;
```

The loop on index `i` is said to be the *outer* loop, whereas the loops on `j` are said to be *inner* loops. This structure, that we will call *nested loop sequence*, should be distinguished from the *nested loops* (used mainly in the parallelization of FORTRAN programs,) in which a loop body is either another loop or a sequence of statements containing no loops.

An *iteration* is a particular execution of the loop body. It is characterized by the current value of the associated loop index, and by the values of loop indices of outer loops. The vector of loop index values (in nesting order) characterizing a given iteration is called *iteration vector*.

An *elementary instruction* is an assignment or a communication instruction. An *elementary operation* is a particular execution of an elementary instruction.

It is totally defined by giving the corresponding elementary instruction and an iteration vector.

There is a *dependence* ([Kuck78]) between elementary operations `op1` and `op2` if the following conditions are all satisfied:

i) both operation access the same memory location,

ii) program semantics depends on the order of execution of these operations,

iii) for the results to be correct, `op1` must be executed before `op2`.

There is a *flow dependence* (also called *data dependence*) between `op1` and `op2` if `op1` produces a value used by `op2`. There is an *antidependence* between `op1` and `op2` if `op1` reads a memory location modified by `op2`. There is an *output dependence* between `op1` and `op2` if `op1` modifies a memory location further modified by `op2`.

## 2.2 Assumptions

ALPHA programs must be computable by a multidimensional timing function, common to all local variables and defined by the ascending lexicographical order of domain points restricted to temporal indices. All local variables are supposed to be defined on the same index space.

ALPHA programs can be partitioned. If the case arises, the partitioning is intended for a fixed-size, $p$-dimensional grid. It corresponds to the decomposition of the domains by $p$ families of equally spaced, canonical hyperplanes of dimension $n-1$ (i.e., hyperplanes defined by exactly one equation of the form $index_i = constant$). A non-partitioned ALPHA program expressed in a $n$-dimensional index space will be expressed in a $n+p$-dimensional index space after partitioning. By convention, the first $p$ indices in a partitioned program are supposed to be spatial coordinates.

The target system of the compiler consists of a host computer and a network of processor whose *logical* topology is a $p$-dimensional grid. Each processor works sequentially and is the only one to access its memory. The network itself operates in the SPMD[1] mode with a unique program for all processors, eventually containing conditionals on processor numbers.

The control structures used to express an ALPHA program are nested loop sequences, whose iteration spaces matches the union of domains of all local variables of the ALPHA program. The value of an instance $A(z)$ is computed by an assignment operation `A[...] := ...` whose iteration vector is $z$.

[1] Single Program stream, Multiple Data streams.

# 3 The data structure generation problem

The study of an example will help understanding problems that arise during data structure generation. The following ALPHA programs are an abstraction of a relaxation algorithm ([Tsen89].) The reader not familiar with the ALPHA language can refer to the beginning of the previous section whenever needed.

Let us define the meaning of the ALPHA programs shown below. For the sake of simplicity, there is no convergence test in the programs, as our main concern is in data dependences, not in the algorithm itself. For the same reason, we don't bother about boundary handling. In the program `initial_SOR`, index $k$ is the iterate index, while indices $i$ and $j$ are location indices. At iteration $k$, an element $A(k,i,j)$ is computed using the element $A(k-1,i,j)$ and its four neighbours $A(k-1,i-1,j)$, $A(k-1,i,j-1)$, $A(k-1,i,j+1)$ and $A(k-1,i+1,j)$, all computed at iteration $k-1$. Implicitly, the loops in the target program will be nested in the order $(k,i,j)$, with $k$ being the index of the outermost one. The initial values of $A$ are read on the host from variable $a$. After 100 steps of relaxation, the result is sent to the host and stored in variable *res*.

```
system initial_SOR
    (a : {i,j | 1<=i<=512; 1<=j<=512} of real)
  returns
    (res : {i,j | 1<=i<=512; 1<=j<=512} of real);
  var
    A : {k,i,j | 0<=k<=25; 1<=i<=512; 1<=j<=512} of real;
  let
    A = case
        {k,i,j|k=0} : a.(k,i,j -> i,j);
        {k,i,j|511>=i>=2;511>=j>=2;25>=k>=1} :
          (A.(k,i,j->k-1,i-1,j  )
          +A.(k,i,j->k-1,i,  j-1)
          +A.(k,i,j->k-1,i+1,j  )
          +A.(k,i,j->k-1,i,  j+1)) / 2.(k,i,j->)
          - A.(k,i,j->k-1,i,j) ;
        -- boundary operations should come here ...
        esac;

    res = A.(i,j->25,i,j) ;
tel;
```

The program `partitioned_SOR` corresponds to the previous one after partitioning for a linear array of four processors. Data are distributed by blocks of 128 adjacent columns of the original matrix. The new index $p$ is interpreted as the processor number, while the meaning of indices $k$, $i$ and $j$ remains unchanged. Communication operations introduced by the partitioning imply an extension of the *case* structure, defining the source of the datum to be received.

```
system partitioned_SOR
    (a : {i,j | 1<=i<=512; 1<=j<=512} of real)
  returns
    (res : {i,j | 1<=i<=512; 1<=j<=512} of real);
  var
    A : {p,k,i,j | 1<=p<=4; 0<=k<=100; 1<=i<=512; 128p-127<=j<=128p} of real;
  let
    A = case
        {p,k,i,j|k=0} : a.(p,k,i,j -> i,j);
        -- calculations without communication
        {p,k,i,j|2<=i<=511;128p-126<=j<=128p-1;1<=k} :
          (A.(p,k,i,j->p,k-1,i-1,j  )
          +A.(p,k,i,j->p,k-1,i,  j-1)
          +A.(p,k,i,j->p,k-1,i+1,j  )
          +A.(p,k,i,j->p,k-1,i,  j+1)) / 2.(p,k,i,j->)
          - A.(p,k,i,j->p,k-1,i,j) ;
-- receiving a datum from the "predecessor"
        {p,k,i,j|2<=p;2<=i<=511;128p-127=j;1<=k} :
          (A.(p,k,i,j->p,  k-1,i-1,j  )
          +A.(p,k,i,j->p-1,k-1,i,  j-1)
          +A.(p,k,i,j->p,  k-1,i+1,j  )
          +A.(p,k,i,j->p,  k-1,i,  j+1)) / 2.(p,k,i,j->)
          - A.(p,k,i,j->p,k-1,i,j) ;
-- receiving a datum from the "successor"
        {p,k,i,j|p<=3;2<=i<=511;j=128p-1;1<=k} :
          (A.(p,k,i,j->p,  k-1,i-1,j  )
          +A.(p,k,i,j->p,  k-1,i,  j-1)
          +A.(p,k,i,j->p,  k-1,i+1,j  )
          +A.(p,k,i,j->p+1,k-1,i,  j+1)) / 2.(p,k,i,j->)
          - A.(p,k,i,j->p,k-1,i,j) ;
-- we do not bother about domain boundaries ...
        esac;

    res = case
    {i,j|  1<=j<=128} : A.(i,j->1,25,i,j) ;
    {i,j|129<=j<=256} : A.(i,j->2,25,i,j) ;
    {i,j|257<=j<=384} : A.(i,j->3,25,i,j) ;
    {i,j|385<=j<=512} : A.(i,j->4,25,i,j) ;
  esac;
tel;
```

A straightforward method of generating data structures corresponding to the $A$ variable of program `initial_SOR` is to associate an array element with

every instance (i.e., element) of $A$. In this case, the corresponding C language declaration will be

```
float A[101][512][512];
```

Clearly, this solution is far from optimal, as once calculated, the values are held in memory until the execution ends, whereas they become useless after a bounded delay. Given the implicit sequencing, and restricting the analysis to assignment operations, we can say that the value calculated at the point $(p, i, j, k)$ is successively used by calculations carried out at points $(p, k+1, i-1, j)$, $(p, k+1, i, j-1)$, $(p, k+1, i, j)$, $(p, k+1, i, j+1)$ and $(p, k+1, i+1, j)$, in this order, and that *this value will not be used after this last reference*. In other words, after evaluating $(p, k+1, i+1, j)$ the memory location associated with the instance $A(p, k, i, j)$ can be released and possibly used to hold another instance of $A$.

From this observation we can deduce the set of all instances calculated before a given instant $t$ and which will be used at or after $t$. Let us call *lifetime* of an instance the delay $v$ between its calculation and its last use. Let $v_{max}$ be the maximum of all lifetimes of instances of $A$. Then at an instant $t$, all the instances calculated at or after the instant $t - v_{max}$ must be held in memory. As we shall see, the locality of dependences inherent to the ALPHA language implies the existence of an upper bound of the maximum lifetime for a given variable.

The set of preserved instances being bounded, we must define its shape. As it has to be identified with an imperative array, the choice depends on the characteristics of the target language. In most cases, array bounds must be independent and statically defined, enforcing the use of independent constraints, each relating to exactly one dimension. In such context, transformations of instance sets can be defined as compositions of independent transformations on single dimensions.

With the above assumptions, the sets of useful instances can be identified with imperative arrays and their elementary transformations will be meaningful in the imperative context. Two of them are of special interest, given their algebraic properties: a mere suppression of a dimension (see fig. 1,) and the circular (alias "modulo") handling of a limited number of elements along a dimension (fig. 2.)

The section 5 shows how to characterize and compute these transformations in a unified way using a simple mathematical model.

# 4 Unoptimized data structures

An unoptimized memory image of an ALPHA variable $V$ is a data structure `imV` of the target language such that there is an injective morphism of the instances of $V$ into the set of elements of `imV`. If $V$ is partitioned, the corresponding data

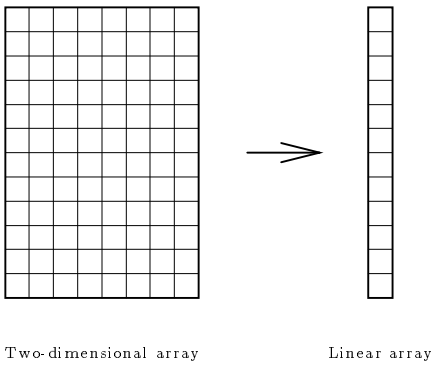



Figure 1: Suppression of a dimension

structure `imV` will be distributed on the network according to a scheme compatible with the partitioning relation of the ALPHA variable, i.e., if an instance $V(z)$ is associated with processor index $p$, then the corresponding element of `imV` will also be mapped onto processor $p$.

Given the constraints of the target languages, the index sets of arrays can be identified with rectangle parallelepipeds of appropriate dimension and size. Thus, arrays themselves can be identified with ALPHA variables of appropriate type and whose domains are rectangle parallelepipeds. This bijection gives a simple means of generating an array corresponding to a non-partitioned ALPHA variable.

The unicity of the target program implies a unique data structure declaration. This declaration specifies in a generic form the part of the array that will be mapped to a given processor. The hypothesis on the partitioning schemes of accepted ALPHA programs ensures that the partitioning of an ALPHA variable is a valid partitioning for the corresponding array.

With such a partitioning scheme, the array corresponding to a slice of the partitioned variable can be identified with the rectangle hull of this slice, eventually translated. As lower bounds of the partitioned index are different for any two distinct processors, a translation is needed unless the target language allows lower bounds of the arrays to be explicitly defined (e.g., a translation is needed in C, where lower bounds are implicitly null.)

**Example 1:** Consider the $A$ variable from the ALPHA program `partitioned_SOR` from section 3. The partitioning relation for $A$ is shown in figure 3.

In C, the array corresponding to the $A$ variable will be declared as

```
float A[101][512][128];
```

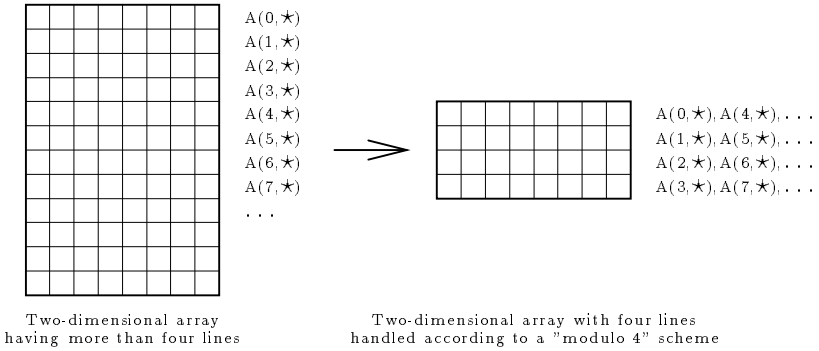



Figure 2: Circular handling of a dimension

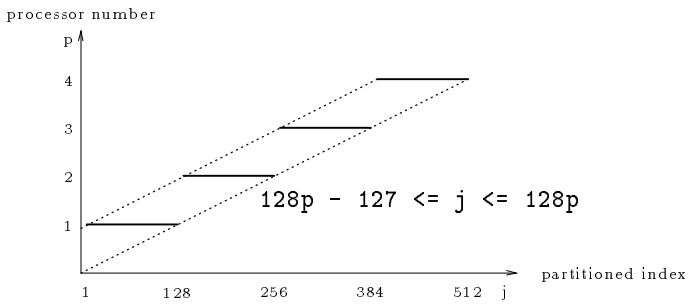



Figure 3: Constant-thickness slicing

Moreover, one will have to handle the translation on the origin during code generation. ■

## 5 Variable reduction

The optimization of memory images of ALPHA variables is aimed at the *minimization* of a criterion characterizing the cost of the execution of the target program. In this study, the only optimization criterion will be the total memory space needed to hold the imperative images of local variables appearing in a given ALPHA program. Other criteria can also be used, but they are beyond the scope of this paper (e.g., synchronization overhead in shared-memory computers, or the execution time overhead due to additional assignment operations.)

Two complementary approaches to optimization problem can be distinguished: a direct transformation of unoptimized data structures, without modifying the ALPHA program, and a source-to-source rewriting of the input program followed by the generation and the optimization of the corresponding data structures. This section is devoted to the former approach. The direct optimization can be done only by *reducing the number of required memory locations* (hence the name "variable reduction",) implying multiple modifications of some or all of them. Therefore, the target program will no longer be single-assignment, and will contain additional dependences between elementary operations.

These dependences can be of two kinds: either antidependences, or output dependences. Output dependences appear only between two assignment operations and will only cause problems when their execution order will be unknown. There is no memory locations common to two distinct processors, hence output dependences can only appear between operations executed by the same processor. Then a sufficient condition of output dependence preservation is that the order of assignment operations be the same in the ALPHA program and in the target program. Between distinct iterations, it is enforced by the control structure generation scheme and by the operation mode of the processor network (see section 2.) Inside an iteration there is no sequencing defined in the input program, as all the calculations at a given point are supposed to be carried out concurrently. The proper intra-iteration sequencing must therefore be enforced by the code generator.

The antidependences correspond to the case sketched briefly in the introductory example: all read accesses to a value must be finished before modifying this value. In the case of distributed-memory computers it is necessary and sufficient to ensure antidependence respect separately on each processor. Therefore, with the hypotheses from section 2 it is legal to restrict the study to non-partitioned programs, as it has to be done separately for each slice of the variable. Meanwhile, the resulting optimization must be common to all slices because the target program is unique for all processors.

## 5.1 Principle of variable reduction

We search for an algebraic method of determining the minimum size of the memory image of a local ALPHA variable. The transformation has to be affine in order to preserve the properties of the dependence functions and it should match the properties of target language arrays. Also, it should be defined by means of a composition of appropriate elementary affine functions.

Let us consider an ALPHA program complying with the hypotheses from section 2 and a local variable $V$ of this program. Let $n$ be the number of dimensions of the domain of $V$. Let $\mathcal{B} = (e_i)_{1 \leq i \leq n}$ be the canonical basis of the index space of local variables and let $\succ$ be the strict ascending lexicographical order. The choice of the basis is motivated by the use of canonical constraints in the definition of target arrays. By identifying the timing function with the order $\succ$, we have

$$\forall i,\ 1 \leq i \leq n-1,\ \forall k \in \mathbf{N} \quad e_i \succ k e_{i+1}$$

and

$$e_1 \succ e_2 \succ \ldots \succ e_{n-1} \succ e_n.$$

Let $\mathcal{D}$ denote the set of dependence functions on $V$ and $d$ the maximum delay between the production of and the last reference to an instance of $V$, as defined by

$$d = \max\{z' - z \mid z \in \mathrm{dom}(V), D_i \in \mathcal{D}, z' \in D_i^{-1} z\}.$$

A composition of elementary array transformations (see section 3) will be called a **reduction function** (RF.) A formal definition is

**Definition 1** *Let $(k_i)_{1 \leq i \leq n}$ be a collection of nonnegative integers. A* **reduction function** *(RF) is an affine morphism $M : \mathbf{Z}^n \rightarrow \mathbf{Z}/k_1\mathbf{Z} \times \mathbf{Z}/k_2\mathbf{Z} \times \ldots \times \mathbf{Z}/k_n\mathbf{Z}$ defined by*

$$M(z) = (z \bmod k_1, z \bmod k_2, \ldots, z \bmod k_n)$$

If $k_i$ equals 0, the corresponding coordinate is left unchanged, that is, no reduction is done along $e_i$. If $k_i$ equals 1, the coordinate is suppressed (it is replaced by 0 for any value,) and finally, if $k_i$ equals a constant $k > 1$, the coordinate is mapped to its equivalence class modulo $k_i$. The corresponding transformation is the replacement of a set of contiguous values by exactly $k_i$ contiguous values handled in a circular way. Therefore, a reduction function applied to the rectangle hull of the domain of $V$ determines the number of dimensions and the bounds of the corresponding optimized array. It also defines the access scheme to be used later during code generation.

Unless trivial (all $k_i = 0$), a reduction function is not injective, hence distinct points $z$ and $z'$ of the domain of $V$ will be mapped to the same element $M(z)$ of the optimized array. It easy to see that this is the case if the difference $z' - z$

is an integer combination of $k_i e_i$. These combinations can be viewed as delays between two modifications of a memory location, expressed in terms of iteration vectors.

Not all possible reduction functions allow to preserve loop-carried antidependences. For the final program to be correct, one must ensure that the longest lifetime $d$ of an instance of $V$ is shorter than the shortest delay between two successive modifications of the corresponding memory location. We will call **valid** a reduction function such that for every positive, nonnull vector $v$ of its nullspace we have $v \succeq d$. If a reduction function $M$ is valid, then in the imperative program in which the unoptimized array corresponding to $V$ was replaced by its image by $M$, all loop-carried antidependences will be preserved.

An *elementary reduction function* (ERF) is a reduction function $M$ such that there are unique integers $k > 0$ and $r$ such that $\forall z,\ M(z) = (z_1, \ldots, z_{r-1}, z_r \bmod k, z_{r+1}, \ldots, z_n)$. The integer k is called **image factor**. The canonical vector $e_r$ defines the **reduction direction**. By construction, any two ERFs whose reduction directions are distinct can be composed in any order (the intersection of their nullspaces is $\{0\}$.) This result can be easily extended to any number of ERFs with distinct reduction directions.

The commutativity of the composition of ERFs is also true for valid ERFs. Moreover, the composition of two valid ERFs whose reduction directions are distinct is clearly valid: let $M_{k,r}$ and $M_{k',r'}$ be two valid ERFs such that $r < r'$. Since $M_{k',r'}$ is valid and for all $k$, $e_r \succ k e_{r'}$, $M_{k,r}$ is also valid.

In order to construct a valid elementary RF, we have to derive a constructive method from the definition of a valid reduction function. As we already saw, it is sufficient to ensure that the smallest positive vector of the nullspace of the RF be greater than the longest lifetime $d$ of instances of $V$. Thus, the sufficient condition for $M_{k,r}$ to be valid is that we have $k e_r - d \succeq 0$. This implies also a necessary condition, useful in establishing the existence of a valid ERF: for a given coordinate $r$ the preceding $r-1$ coordinates of $d$ must all be null. As a consequence, there is no valid ERF other than identity for any coordinate of $d$ following the first nonnull one.

The image factor (IF) is intended for evaluating the quantitative aspect of variable reduction. The IF of an elementary reduction function is the number of array elements that are held in memory along the corresponding array dimension. By extending the concept of the image factor to identity RFs (i.e., for which no reduction is done), it is possible to define the image factor of any reduction function, be it elementary or not. Let the image factor of an identity elementary RF $M_{0,r}$ be the width of the rectangle hull of $V$ along the corresponding dimension. Then for a given reduction function $M$, the product of the image factors of all elementary RFs composing $M$ (including identities) is equal to the size of the memory image of $V$ optimized using $M$.

We can now define the optimality of an elementary reduction function in terms of of the size of the optimized array. An ERF $M_{k,r}$ is said to be **minimal** if there is no other ERF $M_{k',r}$ valid for $V$ and whose image factor is smaller, i.e.,

such that $k' < k$. Thus, to construct a minimal ERF we must find $k$ verifying $(k-1)e_r \prec d$. As the image factor of a composed reduction function $M$ is the product of the image factor of all ERFs composing $M$, the composition of minimal ERFs is also minimal. In particular, we have:

**Theorem 2** *(construction of a minimal reduction function) Let $V$ be a local variable of an* ALPHA *program, $d = (d_1, \ldots, d_n)$ the longest lifetime of instances of $V$, and $k$ and $r$ two integers satisfying:*

*i)* $d - ke_r \preceq 0$,

*ii)* $\text{null}(d) = r - 1$,

*iii)* $d_r \leq k$,

*iv)* $(k-1)e_r \prec d$.

*With these hypotheses, the reduction function $M$ defined by $M = M_{k,r} \circ M_{1,r-1} \circ M_{1,r-2} \circ \ldots \circ M_{1,1}$ is valid for $V$ and minimal.*

**Proof:** A non-trivial reduction can only be applied to coordinates 1 through $r$. As a minimal RF is a composition of minimal ERFs, we must choose the minimal ERF for any of the above coordinates. For every $i$ satisfying $1 \leq i \leq r-1$, we have $e_i \succ d$, hence the corresponding minimal ERFs are $M_{1,i}$. For the $r^{\text{th}}$ coordinate, the minimal ERF is $M_{k,r}$, hence the result. ■

Let us see the application of this method on some examples.

**Example 2:** Consider the ALPHA program `initial_SOR` from section 3. The $\mathcal{B}$ basis is
$([1,0,0],[0,1,0],[0,0,1])$. The set $\mathcal{D}$ of dependence functions is $\{(k,i,j \rightarrow k-1,i,j),(k,i,j \rightarrow k-1,i-1,j),(k,i,j \rightarrow k-1,i,j-1),(k,i,j \rightarrow k-1,i,j+1),(k,i,j \rightarrow k-1,i-1,j)\}$ and the maximum lifetime of an instance of $A$ is $[1,1,0]$. We obtain $r = 1$ and $k = 2$, hence the C declaration of the optimized array:

```
float  A[2][512][512];
```

The declaration of the variable $A$ from program `partitioned_SOR` will only differ in the number of elements along the last dimension (128 instead of 512). Thus, the memory space required for the image of $A$ is now 50 times smaller than before applying variable reduction.

**Target code outline:** for both ALPHA programs, the first coordinate of the array element corresponding to a given instance of $A$ is calculated by taking the value of index $k$ modulo 2. The innermost loop of the imperative program will be

```
A[k%2][i][j] = (A[(k-1)%2][i-1][j] + A[(k-1)%2][i][j-1] +
                A[(k-1)%2][i+1][j] + A[(k-1)%2][i][j+1]) / 2
               -A[(k-1)%2][i][j];
```

■

**Example 3:** Consider the following convolution program, mapped to a linear array of four processors (the processor $p = 0$ is a dummy one) :

```
system convolution (a : { j | 1<=j<=16 }  of integer;
                    x :  { i | i>=1 } of integer)
      returns (y :  { i | i>=16 }  of integer);
  var
    Y :  { p,i,j | 4>=p>=0 ; 4p>=j>=4p-3 ;
                 i>=1 }  of integer;
  let
    Y = case
         { p,i,j | j=0 } : 0 .(p,i,j->);
         { p,i,j | 16>=j>=1 } :
        Y .(p,i,j->p,i,j-1) + a .(p,i,j->p,j) * x .(p,i,j->i-j+1);
        esac;
    y = Y .(i->4,i,16);
  tel;
```

The $p$ index being implicitly the spatial one, the temporal basis is $([1,0],[0,1])$. The C declaration of the unoptimized array associated with $Y$ is then

```
int  Y[big_number][4];
```

where `big_number` is an arbitrary upper bound, depending on the amount of memory available for data. Given the dependence function $(i,j \rightarrow i,j-1)$, the longest lifetime for variable $Y$ is $[0,1]$ and we obtain

- $r = 2$,
- $k = 1$,

thus reducing the initial array to a scalar; the new declaration is

```
int  Y;
```

■

## 6 Source-to-source transformations

Variable reduction allows the transformation of the imperative image of an ALPHA variable, but it is limited by the dependences on this variable (supposed

to be fixed). A more in-depth analysis of reduction rules shows the importance of the lexicographical suffix of the maximum lifetime. If this suffix is strictly positive, it forces to hold in memory a large amount of instances that will not be used.

Intuitively, the solution to this problem is to reduce the the maximum lifetime in such a way that the suffix become negative or null. To do this, one has to modify the dependence functions of the ALPHA program, either by changing the order of the evaluation of instances, or by introducing new variables to "decompose" (cut) the dependence functions associated with the original variable.

Both solutions can be easily formalized and correspond to transformations used frequently when parallelizing sequential programs: the former is in fact the reversing of loop traversal order (ascending to descending or the opposite), whereas the second corresponds to the introduction of temporary variables.

The use of these transformations in the context of the ALPHA language is discussed below. The input information used to compute transformation parameters is basically the same as for variable reduction, however, as it will be shown later, reversing loop traversal order requires more information on a particular class of dependence functions.

## 6.1 Dependence decomposition

With the same notations as in the preceding section, let us consider a local variable $V$ whose associated lifetime is of the form $d = ke_r + d'$ with $e_r \succ d' \succ 0$. After variable reduction, the array associated with $V$ has $k+1$ elements in its first dimension (the first $r-1$ coordinates have been removed,) thus forcing all the instances calculated between instants $z-(k+1)e_r$ and $z$ to be held in memory, whereas only those calculated after $z-(ke_r+d')$ will be really used at or after $z$.

By holding the "oldest" values in a new variable during the delay $d'$, we can reduce the maximum lifetime of the instances of $V$: after a delay of $ke_r$ their values will be assigned to the instances of the auxiliary variable $V_{aux}$, thus immediately freeing the corresponding memory locations.

From the ALPHA point view, this transformation is an extension of the expression decomposition, but cannot be easily expressed using existing transformations. All the right-hand side occurrences of $V$ such that the instance effectively used *can* be "older" than $ke_r$ must be replaced by a conditional: if at a given point $z$, the instance used was calculated before $z-ke_r$, the occurrence of $V$ must replaced by the corresponding occurrence of $V_{aux}$; if not, it can be leaved as is.

The auxiliary variable $V_{aux}$ is defined as follows:

- the domain of $V_{aux}$ is given by $\mathrm{dom}(V_{aux}) = \mathrm{dom}(V) \cap \mathrm{transl}(\mathrm{dom}(V), ke_r)$, where "transl"$(D, v)$ is the translation of the domain $D$ by the vector $v$.
- the type of $V_{aux}$ is that of $V$,

- the equation defining $V_{aux}$ is $V_{aux} = V.(z \rightarrow z - ke_r)$.

Such a definition introduces the problem of intra-iteration sequencing: there is no straightforward ordering of assignments in the resulting loop body if both the initial and the auxiliary variable were reduced using the corresponding minimal reduction functions. This problem is a part of the more general loop body sequencing problem, which must be solved during code generation (possibly by introducing additional scalar variables to break intra-iteration dependence cycles).

**Example 4:** After the dependence decomposition has been applied to the variable $A$, the ALPHA program `initial_SOR` from section 3 has the following form:

```
system initial_SOR_decomposed (a : {i,j|i>=1;j>=1;512>=j;512>=i} of real)
       returns (res : {i,j|i>=1;j>=1;512>=j;512>=i} of real);
  var
    A_aux : {k,i,j|i>=1;k>=1;j>=2;510>=i;25>=k;511>=j} of real;
    A : {k,i,j|i>=1;k>=0;j>=1;512>=i;25>=k;512>=j} of real;
  let
    A_aux = A.(k,i,j->k-1,i,j);
    A = case
        {k,i,j|k=0} : a.(k,i,j->i,j);
        {k,i,j|i>=2;k>=1;j>=2;511>=i;25>=k;511>=j} :
        (A_aux.(k,i,j->k,i-1,j) + A_aux.(k,i,j->k,i,j-1) +
           A.(k,i,j->k-1,i+1,j) + A.(k,i,j->k-1,i,j+1)) / 2.(k,i,j->) -
          A.(k,i,j->k-1,i,j);
        esac;
    res = A.(i,j->25,i,j);
  tel;
```

Without dependence decomposition, the minimal RF for $A$ is characterized by $r = 1$ and $k = 2$ (see section 5). After applying dependence decomposition, there is no lifetime longer than $[1, 0, 0]$ for any instance of $A$. Hence, the minimal reduction function for $A$ is now characterized by $r = 1$ and $k = 1$, and the new C language declaration of $A$ is

```
float  A[512][512];
```

The maximum lifetime of instances of $A_{aux}$ is $[0, 1, 0]$, hence the minimal RF for $A_{aux}$ is given by $r = 2$ and $k = 1$, and the corresponding C declaration will be

```
float  A_aux[512];
```

If dependence decomposition was applied to the program `partitioned_SOR`, the new declarations of `A` and `A_aux` would be the same as for the program

`initial_SOR_decomposed`, excepted for the width of array slices along the last dimension.

In place of a three-dimensional array, we obtain now one two-dimensional and one one-dimensional array, and the memory space required for the data structures has been virtually reduced by a half (the size of `A_aux` represents less than 0.2% of the space recovered by suppressing a part of the old `A` array.) ■

## 6.2 Changing loop traversal order

This transformation is based on a change of basis leading to the modification of the $d'$ suffix of the maximum lifetime $d$. As we suppose $ke_r \prec d \prec (k+1)e_r$, initially the minimal reduction function for $V$ is $M_V = M_{k+1,r} \circ M_{1,r-1} \circ \ldots \circ M_{1,1}$. By applying the dependence decomposition it is possible to further limit the necessary memory space, as the array associated with the auxiliary variable $V_{aux}$ can be reduced using the reduction function $M_{V_{aux}} = M_{k_{r_0},r_0} \circ M_{1,r_0-1} \circ \ldots \circ M_{1,2} \circ M_{1,1}$ where $r_0$ is the position of the first nonnull coordinate of $d'$, and $k_{r_0}$ satisfies $k_{r_0}e_{r_0} \succeq d' \succ (k_{r_0}-1)e_{r_0}$.

The $r^{th}$ coordinate of the array associated with $V$ cannot be reduced to less than $k$ elements in any way, thus the further reduction of the memory use depends on the reduction of the image factor of $M_{V_{aux}}$. The $d'$ suffix being strictly positive, its first nonnull coordinate $d'_{r_0}$ is also positive. Then if we apply to the domain of $V$ a change of basis in which the sign of the coordinate $r_0$ is reversed, the suffix $d'$ will become strictly negative in the new basis $\mathcal{B}'$. The change of basis is given by

$$\mathcal{B} = (e_1, e_2, \ldots, e_{r_0}, \ldots, e_n) \longmapsto \mathcal{B}' = (e_1, e_2, \ldots, -e_{r_0}, \ldots, e_n).$$

This change of basis can, however, have undesirable effects if applied without precautions. If there are lifetimes $t = (t_1, t_2, \ldots, t_n)$ such that their $r_0$–th coordinates $t_{r_0}$ satisfy $t_{r_0} < -k_{r_0}$, then after the above change of basis the new dependences will generate *longer* lifetimes, thus requiring more memory space than before the transformation.

So, the construction of the appropriate change of basis is subject to additional conditions: if for each dependence function $D_i$ such that there are lifetimes $t = (t_1, t_2, \ldots, t_n)$ generated by $D_i$ and satisfying $t_r = k$, the inequality $-k_{r_0} < t_{r_0}$ holds, then the application of the change of basis $T$ given by

$$T \ : \ \mathcal{B} = (e_1, e_2, \ldots, e_{r_0}, \ldots, e_n) \longmapsto \mathcal{B}' = (e_1, e_2, \ldots, -e_{r_0}, \ldots, e_n),$$

to the domain of the variable $V$ reduces the minimum total memory space required by the optimized arrays associated with $V$ and with its auxiliary variable.

**Example 5:** The Gaussian elimination is a good example of application of loop traversal order reversing. Consider the following ALPHA program:

```
system GAUSS (a : {i,j|i>=1;j>=1;20>=j;20>=i} of real;
              b : {i|20>=i;i>=1} of real)
       returns (u : {i,j|20>=j;i>=1;j>=i} of real;
                x : {i|20>=i;i>=1} of real);
  var
    A : {k,i,j|i>=1;k>=0;j>=1;j>=k;20>=i;21>=j;20>=k} of real;
  let
    A = case
        {k,i,j|k=0;20>=j;j>=1} : a.(k,i,j->i,j);
        {k,i,j|k=0;21=j} : b.(k,i,j->i);
        {k,i,j|j>=k;k>=1;k=i;20>=k;21>=j} :
        A.(k,i,j->k-1,i,j) / A.(k,i,j->k-1,k,k);
        {k,i,j|k>=1;i>=k+1;j>=k;20>=i;21>=j} :
        A.(k,i,j->k-1,i,j) -
          A.(k,i,j->k-1,i,k) / A.(k,i,j->k-1,k,k) *
            A.(k,i,j->k-1,k,j);
        esac;
    x = {i|20>=i;i>=1} : A.(i->i,i,21);
    u = {i,j|20>=j;i>=1;j>=i} : A.(i,j->i,i,j);
  tel;
```

By noting $\infty$ the undefined bounds (such as the extreme values in the diffusions,) the four dependence functions on $A$: $D_1 = (k, i, j \rightarrow k-1, i, j)$, $D_2 = (k, i, j \rightarrow k-1, k, k)$, $D_3 = (k, i, j \rightarrow k-1, k, j)$ and $D_4 = (k, i, j \rightarrow k-1, i, k)$ generate the following lifetimes:

- $D_1$: $[1, 0, 0]$,
- $D_2$: $[1, 0, 0]$ through $[1, \infty, \infty]$,
- $D_3$: $[1, 0, 0]$ through $[1, \infty, 0]$,
- $D_4$: $[1, 0, 0]$ through $[1, 0, \infty]$.

$d$ equals $[1, \infty, \infty]$ and the minimal reduction function for $A$ is given by chosing $r = 1$ and $k = 2$. As the requirements for reversing loop traversal order are clearly satisfied, we can directly apply the change of basis $(e_1, e_2, e_3) \longmapsto (e_1, -e_2, e_3)$. The new program is

```
system GAUSS_new
             (a : {i,j|j>=1;i>=1;20>=i;20>=j} of real;
              b : {i|20>=i;i>=1} of real)
       returns (u : {i,j|20>=j;i>=1;j>=i} of real;
                x : {i|20>=i;i>=1} of real);
  var
    A : {k,i,j|0>=i+1;k>=0;j>=1;j>=k;i+20>=0;21>=j;20>=k} of real;
```

```
let
  A = case
      {k,i,j|k=0;20>=j;j>=1} : a.(k,i,j->-i,j);
      {k,i,j|k=0;21=j} : b.(k,i,j->-i);
      {k,i,j|j>=k;k>=1;0=k+i;20>=k;21>=j} :
      A.(k,i,j->k-1,i,j) / A.(k,i,j->k-1,-k,k);
      {k,i,j|k>=1;0>=k+i+1;j>=k;i+20>=0;21>=j} :
      A.(k,i,j->k-1,i,j) -
        A.(k,i,j->k-1,i,k) / A.(k,i,j->k-1,-k,k) * A.(k,i,j->k-1,-k,j);
      esac;
  x = {i|20>=i;i>=1} : A.(i->i,-i,21);
  u = {i,j|20>=j;i>=1;j>=i} : A.(i,j->i,-i,j);
tel;
```

The new dependence functions are now $D'_1 = (k,i,j \mapsto k-1,i,j)$, $D'_2 = (k,i,j \mapsto k-1,-k,k)$, $D'_3 = (k,i,j \mapsto k-1,-k,j)$ and $D'_4 = (k,i,j \mapsto k-1,i,k)$. Thus, the new maximum lifetime $d$ is $[1,0,\infty]$ instead of $[1,\infty,\infty]$.

Although one could decompose the dependences at this point, it can be easily seen that once again we can apply a change of basis, as the program `GAUSS_new` satisfies the requirements for reversing loop traversal order. After this second change of basis, defined by $(e_1,e_2,e_3) \longmapsto (e_1,e_2,-e_3)$ the maximum lifetime $d$ will be bounded by $[1,0,0]$, thus suppressing the need for dependence decomposition. ■

# 7 Current state and evolution

An experimental ALPHA compiler, called CFC (Control Flow Compiler) is currently being designed. It is subdivided into two major modules: the data structures generator, and the control structures (code) generator. In its current form, the data structures generator consists of a set of simple tools, integrated into the ALPHA du Centaur environment. These tools are intended to verify the applicability of the source-to-source transformations, to generate unoptimized arrays associated with local variables, and to apply variable reduction according to the results of the dependence test.

The analysis stage determines the maximum and minimum lifetimes of instances of a given variable, and performs an analysis of this lifetime information. In particular, it identifies the partitioning relations. Calculating minimum lifetimes in addition to the maximum lifetime appeared very useful when modifying loop traversal order: although minimum lifetimes do not always match the assumptions from previous section, in practice it appeared to be the case for a large number of problems, justifying the shortcut.

The lifetime calculation algorithm is of linear complexity and is coded in Lisp, whereas all other parts of the data structure generator were implemented

as sets of semantical rules and compiled into Prolog programs using Centaur's semantics-oriented subsystem called TYPOL.

The forthcoming extensions of the prototype generator consist of a built-in dependence decomposition transformation (by now, it has to be *applied* manually,) and of the automation of the complete generation chain. Also, an adaptation of the generator to shared-memory computers is being investigated.

The data structure generator is only a part of the ALPHA language compiler for regular distributed-memory computers. As such, it will be integrated in the near future into the complete compiling chain, still under development.

# 8 Discussion

Very few work seems to be currently done on the generation of data structures from systolic programs. There is a project aimed at the generation of imperative parallel programs from systolic ones, managed by Dr. Megson at the University of Newcastle upon Tyne (Great Britain,) but to date no information has been released on it. Nevertheless, this topic is related to some of the parallelization techniques introduced recently for the FORTRAN programs, such as the ones proposed by the PTRAN team at IBM ([Alle88]), and by Feautrier and Werth ([RaFe91]). In these approaches one first generates a single-assignment program from the input FORTRAN code, then one searches for a more efficient memory utilization scheme.

In PTRAN, a shared-memory model with dynamical loop scheduling is used, thus requiring the number of loop-carried dependences to be very low. In the approach of Feautrier and Werth, the target is a distributed-memory computer, whose exact characteristics are hidden using the concept of virtual processors. Remote memory reads are allowed, requiring global synchronization (e.g., a sequential loop) to ensure dependence preservation.

In such a context it is necessary to take very conservative assumptions when optimizing expanded data structures (on an example program, Feautrier reports a fourfold increase in memory requirements, when compared with the initial sequential program.) Also, an in-depth analysis of the program dependences must be performed before generating the single-assignment intermediate program. Finally, in both of the above-mentioned papers a large part of the research seems to be devoted to the definition of program representations used to make dependence analysis and program rewriting easier.

In our approach, no remote memory accesses are allowed, restricting interprocessor dependences are preserved by communication operations. Also, the mapping of iterations to the target topology is given in the input program, requiring neither runtime overhead nor complex mapping functions. Eventually, the complexity of the reduction transformation is very low, as its definition is given in terms of lexicographical inequalities. The efficiency of the variable reduction is close to the optimizations one could perform if the target program

was sequential: the only limitations come form the fact that the variable reduction is performed on a global basis, thus ignoring additional optimizations local to some processors.

# 9 Conclusion

The three optimization methods presented in this paper allow very memory-efficient data structures to be obtained when generating imperative programs from systolic specifications. They will be used in the complete ALPHA compiler for regular distributed-memory computers, permitting to expect (in long-term and for a class of problems) mathematical specifications to be a programming language for these architectures.

The methods presented in this paper fully benefit from the major characteristics of the ALPHA language: powerful theoretical basis, declarative semantics, and local, regular dependence functions. The characterization of the target architectures, although seeming restrictive, is very useful in defining optimizing transformations which remain realistic for the real-life applications thanks to their low complexity. The major drawback of the approach is that when reducing memory requirements, additional constraints on instruction sequencing are introduced, thus increasing the complexity of the code generation process.

# Acknowledgements

I'm very grateful to my thesis' advisor, Dr. Patrice Quinton, who supervised this research. I would also thank Professor Paul Feautrier for many helpful discussions we had on the optimization of data structures in parallelization of FORTRAN programs.

# References

[Alle88] F. Allen et al. A framework for determining useful parallelism. In *International Conference on Supercomputing*, pages 207–215, St. Malo, 1988.

[Borr88] P. Borras, D. Clément, Th. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. CENTAUR : the system. In *ACM SIGSOFT '88*, 1988.

[CLMQ90] Z. Chamski, H. Le Verge, C. Mauras, and P. Quinton. Interactive design of parallel algorithms using the ALPHA du Centaur environment. In *Int'l. Workshop on Compilers for Parallel Computers*, pages 399–410, ENSMP/UPMC, Paris, France, December 1990.

[GMQS89] P. Gachet, C. Mauras, P. Quinton, and Y. Saouter. Alpha Du Centaur: a prototype environment for the design of parallel regular algorithms. In *International Conference on Supercomputing*, pages 235–243, ACM, 1989.

[Kuck78] D. J. Kuck. *The Structure of Computers and Computations*, chapter 2 (section 2.4: 'Program Dependence and Transformation'), pages 135–155. Volume I, John Wiley & Sons, 1978.

[Kung82] H. T. Kung. Why systolic architecures. *Computer*, 37–45, January 1982.

[Maur89] Ch. Mauras. *Alpha : un langage équationnel pour la conception et la programmation d'architectures parallèles synchrones.* Ph.D. dissertation, Univ. Rennes I, Rennes, France, December 1989.

[MQRS90] C. Mauras, P. Quinton, S. Rajopadhye, and Y. Saouter. *Scheduling Affine Parameterized Recurrences by Means of Variable Dependent Timing Functions.* Technical Report 520, IRISA, Campus de Beaulieu 35042 Rennes Cédex, jan 1990.

[Quin83] P. Quinton. *The Systematic Design of Systolic Arrays.* Technical Report 193, Publication Interne IRISA, Avril 1983.

[QuVa88] P. Quinton and V. Van Dongen. The mapping of linear recurrence equations on regular arrays. October 1988. Submitted to *The Journal of VLSI Signal Processing.*

[RaFe91] M. Raji-Werth and P. Feautrier. On parallel program generation for massively parallel architectures. In M. Durand and F. El Dabaghi, editors, *High Performance Computing II*, North-Holland, October 1991.

[Robe86] Y. Robert. *Algorithmes et architectures systoliques.* Institut National Polytechnique de Grenoble, October 1986.

[Tsen89] P.-S. Tseng. *A Parallelizing Compiler for Distributed Memory Parallel Computers.* Ph.D. dissertation, Carnegie Mellon Univ., Pittsburgh, USA, June 1989.