ENSMP-CAI-87-E94

Document No. E94

ECOLE NATIONALE SUPERIEURE
DES MINES DE PARIS

CENTRE D'AUTOMATIQUE ET INFORMATIQUE

Computing Dependence Direction Vectors and Dependence Cones with Linear Systems

F. Irigoin and R. Triolet

September 1987

35, rue Saint Honoré
77305 FONTAINEBLEAU CEDEX
FRANCE

Tel. : (1) 64.22.48.21 Tfx. : (1) 64.22.39.03 Tlx. : MINEFON 694736 F

Abstract

Program parallelization is usually based on dependence graph analysis. The dependence graph is built on program statements and conveys ordering constraints on statements and statement iterations. These constraints must be summarized since statement iterations are virtually unbounded. This is usually done by using *Dependence Direction Vectors* (DDV). We introduce here a new concept called *Dependence Cone* (DC), that provides a more accurate dependence summary than DDV, and show that parallelization techniques based on DDV can be adapted to DC. DC's computation is based on linear systems and can exploit intra- and inter-procedural information on variable values and CALL effects. Furthermore DC's accuracy increases the number of possible reorderings with transformations like the hyperplane method and supernode partitioning and global parallelization.

Introduction

Automatic program parallelization is not an easy task. Supercomputer architectures provide many features like vector registers or cache memory that cannot be exploited only by detecting parallel DO-loops in a program. Statements and statement iterations must be transformed and/or reordered without changing the program semantics. Reordering transformations must be compatible with Bernstein's conditions[4] and keep statement iterations which write and read the same memory location in the same temporal order.

Statements and statement iterations that refer the same memory location are said to be **dependent** on each other and dependence analysis[12, 1] has become a major tool in program parallelization. The dependence relation can be defined at the statement[i] level, or at the statement iteration level. The iteration level would usually require very large[ii] dependence graphs to represent the dependence relation. The dependence graph would not fit in memory and would not be easy to use. Thus vectorizers and parallelizers usually build a dependence graph at the statement level, but label each arc with summary information on dependences between statement iterations. One such summary information is the Dependence Direction Vector[21] which is useful for execution reordering transformations like DO-loop parallelization[iii] or DO-loop interchange.

The amount of information available with dependence direction vectors is not approriate for complex reorderings of elementary computations. On the contrary the other kind of summary mechanism presented in this paper, the Dependence Cone (DC), suits very well the need of global parallelization as performed by the hyperplane method or supernode partitioning methods.

In the first section, we define our notations and explain which assumptions are made on the program to be parallelized: control structures, form of subscript expressions, etc...

In the second section, we explain how a linear system can be built to denote a potential dependence between two statements due to two array references. Then, we show how this system, called the dependence system, is used to either prove the dependence does not hold or to compute additional information on this

[i] A few transformations like node-splitting require dependence relations to be defined at the variable reference level.

[ii] They are even potentially unbounded since DO-loop bounds cannot always be evaluated at compile time.

[iii] Loop parallelization is a reordering since this transformation destroies many couples of the execution order relation. This is a sufficient condition but not a necessary one for DO-loop vectorization. For example, the DO-loop $A_i = A_{i+1}$ is a vector loop but not a parallel one. See [15] for a thorough discussion of the difference.

dependence.

In the third section, we show how dependence information can be summarized with dependence direction vectors or with dependence cones. Both representations can be computed from the linear system built in the second part but dependence cones are more accurate than dependence direction vectors.

In each section many examples are given.

1. Basics of Parallelization. Assumptions and Notations.

Program parallelization and restructuring must preserve the initial program semantics. The order of read and write accesses to each memory location is kept while accesses to distinct memory locations can be exchanged. Thus the value history of each memory location remains the same and final states are equal at the bit level.

Dependence based parallelization enforces the order between any read and any write and between writes. This is a sufficient condition for histories to be preserved.

1.1. Assumptions

We assume there is no aliasing by equivalence or by parameter passing or between common variables and parameters. Thus memory locations referred to with different names are distinct.

Scalar variables are easy to deal with since dependences can be tested with a string comparison. This paper considers only array references enclosed in DO-loop bodies.

We assume the program is structured with sequence and DO-loop operators because backward and forward GOTO and IF statements can be dealt with with the IF conversion algorithm described in [2]. Furthermore, DO-loops are *normalized*: their lower bounds and their increments are equal to 1. Thus statements and statement iterations can be designated by statement numbers and enclosing DO-loop index values. The execution order is the lexicographic order (see figure 1).

Finally we suppose that subscript expressions and DO-loop bounds are linear expressions based on integer scalar variables. However, non-linear expressions can be dealt with by assuming they can take any value in the declaration range and by deleting all equations or inequations built with them. For example, the

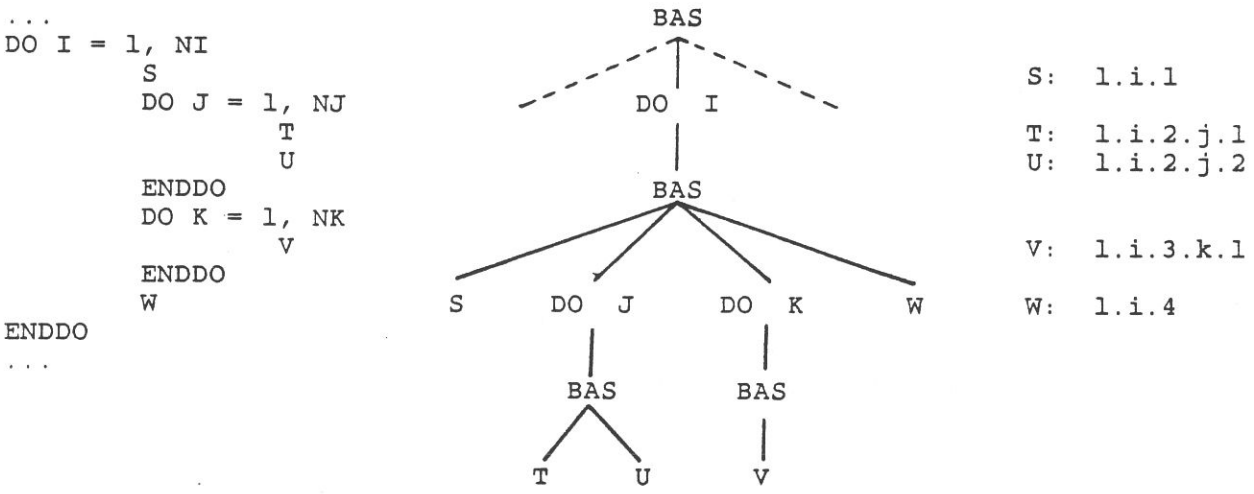


Figure 1. Designations for Statement Iterations

non-scalar-linear references A(I,B(K)) and A(I+1,K**2) would be reduced to A(I,*) and A(I+1,*), which still provides useful information.

These assumptions may seem to be too restrictive at first sight but are usual in automatic parallelization because they hold for most parallel DO-loops.

1.2. Notations and Definitions

Let S1 and S2 be two potentially dependent statements. Let R1 and R2 be two references to the same array. Let k be the number of enclosing DO-loops common to S1 and S2. If non common DO-loops are ignored, and if S1 and S2 are both enclosed in at least one DO-loop, then it is not necessary to designate statements as shown in figure 1. Coordinates which are not DO-loop index values can be dropped except the last one which must be kept to indicate the order of two statements within the DO-loop body. For the sake of simplicity, this last coordinate is not shown in the following.

So iterations of S1 and S2 are integer elements of a k-dimensional vector space called the **iteration space**, and are labeled by **iteration vectors** whose coordinates are equal to DO-loop index values relative to iterations. Loop bounds define a subset of the iteration space called the **iteration domain**. Under our linearity assumptions, this subset is a convex polyhedron.

Let suppose iteration $\vec{j}_2$ of S2 reads a memory location written by iteration $\vec{j}_1$ of S1. Then S2 *depends* on S1 and $\vec{d} = \vec{j}_2 - \vec{j}_1$ is a dependence distance vector. If $\vec{d}$ is lexico-positive, $\vec{j}_2$ must occur after $\vec{j}_1$ and the dependence is called **data (flow) dependence** or **true dependence** since its expresses the way values are re-used in the computation. If $\vec{d}$ is lexico-negative, the read must occur before the write, because of a memory location reuse; the dependence is from S2 to S1, and is called **anti-dependence**; its dependence distance vector is $-\vec{d}$. If R1 and R2 are both write references, the dependence is called **output-dependence** and $\vec{d}$ is kept lexico-positive by choosing S1 or S2 as origin of the dependence. In the following all kinds of dependences are unioned in one dependence relation.

Control dependences are ignored because the program is assumed to be structured (see § 1.1).

2. Dependence System

This section is on setting, in terms of linear equations and inequations, the conditions for a dependence between two statements S1 and S2 to exist because of references R1 of S1 and R2 of S2. This system, called the dependence system, is used to compute the set of points $\vec{j}_2$ of the iteration space which are dependent on a given point $\vec{j}_1$. This set may be approximated by a convex polyhedron (see figure 2-a):

$$\Pi_{\vec{j}_1} = \left\{ \vec{j}_2 - \vec{j}_1 \;/\; \vec{j}_2 \text{ depends on } \vec{j}_1 \right\}$$

The shape and size of $\Pi_{\vec{j}_1}$ vary with $\vec{j}_1$ and we approximate all possible polyhedra by their convex hull (see figure 2-b):

$$\Pi = \bigcup_{\vec{j}_1} \Pi_{\vec{j}_1}$$

where the union symbol denotes the convex hull. As can be seen in figure 2-c, this convex hull is an upper approximation of $\Pi_{\vec{j}_1}$ for any $\vec{j}_1$.

Such dependence systems are built for each pair (R1, R2) of references to the same array. They contain information on the references (subscript expressions), and information valid for any iteration of S1 and S2, like variables constant between S1 and S2 or enclosing DO-loop bounds.

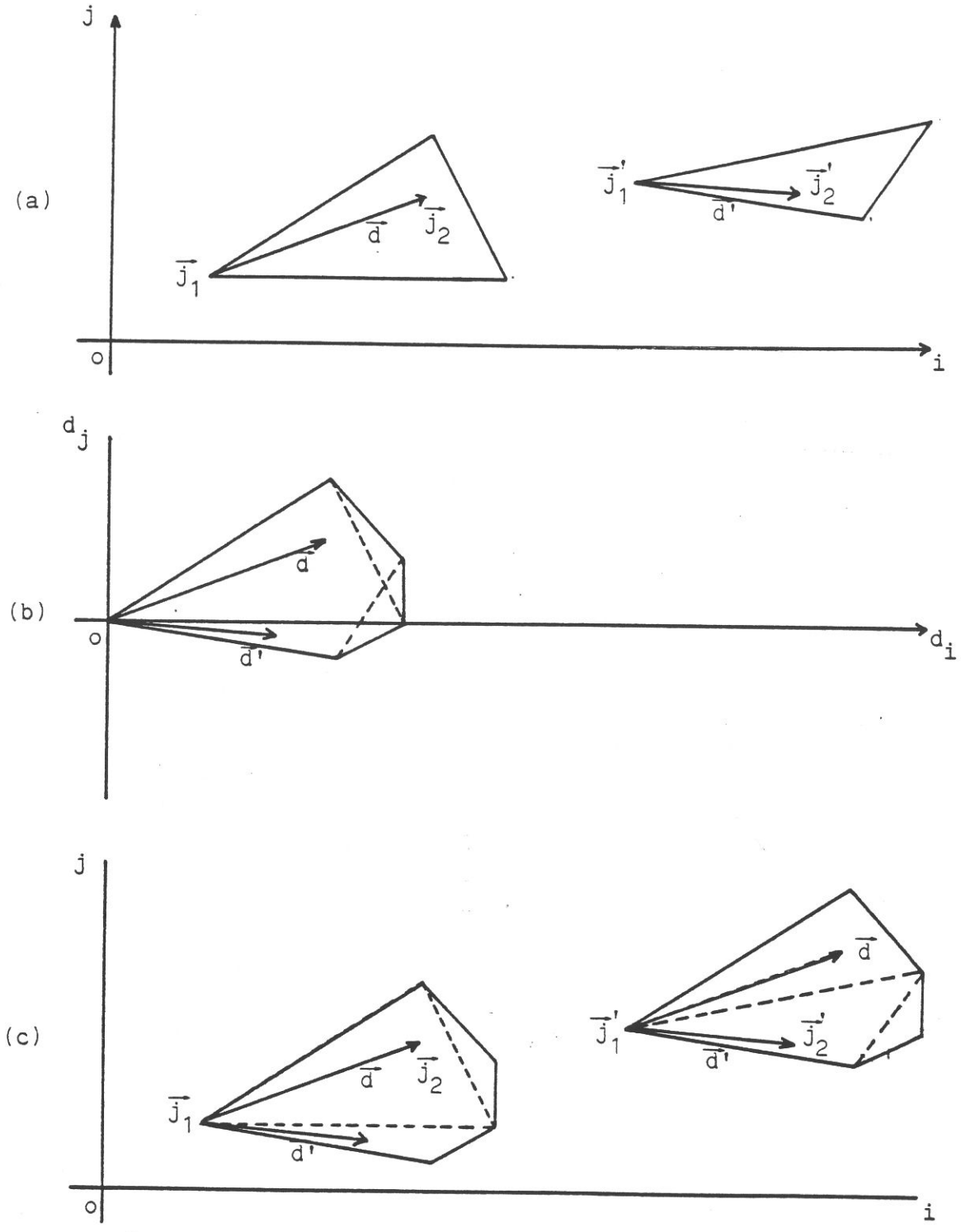


Figure 2. Dependence Polyhedra

2.1. Variables of the Dependence System

Let $\vec{j}_1$ and $\vec{j}_2$ be the iteration vectors of S1 and S2:

$$\vec{j}_1 = \begin{pmatrix} i_1^1 \\ i_2^1 \\ \vdots \\ i_k^1 \end{pmatrix} \qquad \vec{j}_2 = \begin{pmatrix} i_1^2 \\ i_2^2 \\ \vdots \\ i_k^2 \end{pmatrix}$$

where i_l^j is the value of index I_l for statement Sj.

Let $\vec{d}$ be the dependence distance vector. By definition $\vec{d} = \vec{j}_2 - \vec{j}_1$.

Index variables are not the only variables to appear in subscript expressions or DO-loop bounds. Other scalar variable values are kept in vectors $\vec{v}_1$ and $\vec{v}_2$ for statements S1 and S2. These vectors are called *variable vectors.*

2.2. Equations and Inequations of the Dependence System

By definition a dependence exists if both statements S1 and S2 refer the same memory location at some iterations $\vec{j}_1$ and $\vec{j}_2$. Let R_1 be the matrix of the affine function mapping the iteration vector $\vec{j}_1$ and the variable vector $\vec{v}_1$ onto array indices and $\vec{r}_1$ the constant term. Let R_2 and $\vec{r}_2$ be the same for R2 of S2. The following equation must hold for a dependence to exist:

$$R_1 \begin{pmatrix} \vec{j}_1 \\ \vec{v}_1 \end{pmatrix} + \vec{r}_1 = R_2 \begin{pmatrix} \vec{j}_2 \\ \vec{v}_2 \end{pmatrix} + \vec{r}_2$$

The definition of the dependence distance vector can be used as such:

$$\vec{d} = \vec{j}_2 - \vec{j}_1$$

Information at S1 and S2 si usually a set of inequations like DO-loop bounds or results of a semantic analysis[7] although equations between variables sometimes exist. Variables may also be disguised constants. This information is added to the system:

$$C_1 \begin{pmatrix} \vec{j}_1 \\ \vec{v}_1 \end{pmatrix} \leq \vec{c}_1 \qquad C_2 \begin{pmatrix} \vec{j}_2 \\ \vec{v}_2 \end{pmatrix} \leq \vec{c}_2$$

Moreover equations between variable values at S1 and S2 sometimes exist:

$$\vec{v}_1 = F\,\vec{v}_2 + \vec{f}$$

Very often, $F = I$ and $\vec{f} = \vec{0}$ when S1 and S2 are close to each other in the program because not all scalar variables are modified in the DO-loop body.

The dependence system characterizes all possible dependence distance vectors between two references of two statement iterations. The projection Π of this system on the subspace of dependence distance vectors $\vec{d}$ can be approximated by the smallest convex hull of these distance vectors, which is again a linear system. If the dependence system is not feasible, Π is empty and there are no dependences between the two statements for the references that were considered.

Kuhn proposed in [13] to test dependences by solving a linear system but he used mostly equalities and built his system on $\vec{j}_1$ and $\vec{j}_2$, ignoring $\vec{v}_1$ and $\vec{v}_2$. Loop bounds had to be linear functions of the outer DO-loop indices.

2.3. Examples of Dependence Free Statements

Let's consider program 1 which initializes the main diagonal, the upper-part and the lower part of a square matrix to three different values.

```
      DO I = 1, N
S1:     T(I,I) = 0.
        DO K = 1, I - 1
S2:       T(I,K) = 1.
S3:       T(K,I) = -1.
        ENDDO
      ENDDO
```

Program 1: Matrix Initialization

In the following upper case letters denote program variables, while lower case letters are used for their values at some iteration $\vec{j}$. To analyze the dependence between S1 and S2, the variables are:

$$\vec{j}_1 = \begin{pmatrix} i_1 \end{pmatrix} \quad \vec{j}_2 = \begin{pmatrix} i_2 \end{pmatrix} \quad \vec{v}_1 = \begin{pmatrix} k_1 \\ n_1 \end{pmatrix} \quad \vec{v}_2 = \begin{pmatrix} k_2 \\ n_2 \end{pmatrix} \quad \vec{d} = \begin{pmatrix} d_i \end{pmatrix}$$

The dependence system can now be written:

$$\begin{cases} i_1 = i_2,\ i_1 = k_2 \\ i_2 = i_1 + d_i \\ 1 \leq i_1 \leq n_1 \\ 1 \leq i_2 \leq n_2,\ 1 \leq k_2 \leq i_2 - 1 \\ n_1 = n_2 \end{cases}$$

This system contains a non-consistent subsystem:

$$\begin{cases} i_1 = i_2,\ i_1 = k_2 \\ 1 \leq k_2 \leq i_2 - 1 \end{cases}$$

It is non feasible and statement S1 and S2 never depend on each other.

In the same way variables for statements S2 and S3 are:

$$\vec{j}_2 = \begin{pmatrix} i_2 \\ k_2 \end{pmatrix} \quad \vec{j}_3 = \begin{pmatrix} i_3 \\ k_3 \end{pmatrix} \quad \vec{v}_2 = \begin{pmatrix} n_2 \end{pmatrix} \quad \vec{v}_3 = \begin{pmatrix} n_3 \end{pmatrix} \quad \vec{d} = \begin{pmatrix} d_i \\ d_k \end{pmatrix}$$

The dependence system becomes:

$$\begin{cases} i_2 = k_3,\ k_2 = i_3 \\ i_3 = i_2 + d_i,\ k_3 = k_2 + d_k \\ 1 \leq i_2 \leq n_2,\ 1 \leq k_2 \leq i_2 - 1 \\ 1 \leq i_3 \leq n_3,\ 1 \leq k_3 \leq i_3 - 1 \\ n_2 = n_3 \end{cases}$$

This new system is not feasible as can be seen by substituting i_2 and i_3 by k_3 and k_2 in the inequations. A third system should be tested for the couple (S1, S3). This would show that the initialization procedure is fully parallel. Let remark that usual dependence computation methods cannot handle triangular DO-loops. In such cases, dependences are assumed.

2.4. Example of Dependent Statements

We need another example to show what happens when a dependence exists. Consider program 2 which is part of a Gaussian elimination algorithm for banded matrices[8] where F is a side-effect free function intended to represent the whole DO-loop body. Three references, R1: Y(I+J), R2: Y(I) and R3: Y(I+J), appear in S and three systems should be built to compute the dependences between iterations of S. These systems, as well as dependence nature, will label arcs from S to S in the statement dependence graph.

```
      DO I = 1, N
        DO J = 1, M
S:        Y(I+J) = F(Y(I), Y(I+J))
        ENDDO
      ENDDO
```

Program 2: Gaussian Elimination of banded Matrices

Here is the dependence system for references R1 and R2:

$$\begin{cases} i_1 + j_1 = i_2 \\ i_2 = i_1 + d_i,\ j_2 = j_1 + d_j \\ 1 \leq i_1 \leq n_1,\ 1 \leq j_1 \leq m_1 \\ 1 \leq i_2 \leq n_2,\ 1 \leq j_2 \leq m_2 \\ m_1 = m_2,\ n_1 = n_2 \end{cases}$$

Its projection[iv] on the dependence subspace is given by the following system $\Pi_{1,2}$:

$$\Pi_{1,2} = \begin{cases} d_i \geq 1 \\ d_i + d_j \geq 1 \end{cases}$$

The dependence system for references R1 and R3, which is equal to the previous dependence system except for the subscript expressions, is also projected and equal to:

$$\Pi_{1,3} = \left\{ d_i + d_j = 0 \right.$$

Thus dependences exist[v] for statement S of the Gaussian Elimination program. Such systems define **dependence polyhedra**.

2.5. Interprocedural Information

Triolet has shown in[19] that effects of procedure calls on a d-dimension array can be summarized with convex polyhedra of $\mathbf{Z}^d$. One polyhedron provides possible values for subscript indices, and so defines a convex part of the array. Such array parts are called **regions**.

For instance, if T is a NxN-array, the region:

$$T(\phi_1, \phi_2) \text{ where } \left\{ \phi_1 = N, 1 \leq \phi_2 \leq \phi_1 \right\}$$

defines the lower triangular part of T. Two pseudo-variables ϕ_1 and ϕ_2 are introduced to represent possibles values of T's subscript indices. The linear system defining a region is built over pseudo-variables, DO-loop indices ($\vec{j}$) and other scalar integer variables ($\vec{v}$).

[iv] In fact, the convex hull of the projection. This does not matter since it is a pessimistic assumption and since reordering transformations are based on convex hulls of many elementary dependences.

[v] It is not yet possible to say rigorously if they exist or not because they should be tested for lexico-positivity and their nature should be computed. But it is more convenient to do this later.

The dependence test presented above can be applied to DO-loops containing procedure calls, providing minor modifications are introduced in the way dependence systems are built. First, pseudo-variables must be added to the system variables. Second, the set of equalities:

$$R_1 \begin{pmatrix} \vec{j}_1 \\ \vec{v}_1 \end{pmatrix} + \vec{r}_1 = R_2 \begin{pmatrix} \vec{j}_2 \\ \vec{v}_2 \end{pmatrix} + \vec{r}_2$$

must be replaced by the systems defining the regions being tested. Other steps remain identical. More details are available in [19].

Thus, dependence testing with linear systems provides an excellent support for interprocedural parallelization.

2.6. Semantics Information Exploitation

It is well known that program restructuring is better done when more information is available on the program variables. Inter-[6] or intra-procedural constant propagation, linear equalities and inequalities on scalar variables can provide key information for parallelization like a dependence sign or a loop upper bound.

The most accurate information that can be computed automatically is made of linear systems of equalities and inequalities[7] . It can be used in a straightforward manner with a dependence test based on linear systems.

Moreover semantic information precludes the need for basic transformations like inductive variable replacement or loop normalization. Information is conveyed by invariants attached to statements to the dependence test algorithm and no actual transformation of the source program is needed[9] .

2.7. Concluding Remarks

When the dependence distance vectors are constant for all iterations of a pair of statements (see program 4), the system Π is reduced to a set of equations.

The projection algorithm is potentially expensive when using a straight Fourrier-Motzkin algorithm as in [20] because the number of inequations doubles at each variable elimination in the worst case. Moreover equations must be converted into inequations and some information is lost because the system should be

solved on **Z**. Fortunately this loss of information leads to an overestimation of dependences and to a correct parallel code.

For accuracy, it seems better to use an exact algorithm like Gomory's cutting plane[17] as it is done in the parallelizer project PAF[18] but this provides only feasibility and projection on **one** variable, or to solve the subsystem of linear diophantine equations and to substitute unknowns by unconstrained variables in the inequations before projection[3] .

The next section shows how Π systems are transformed into dependence direction vectors and dependence cones.

3. Dependence Direction Vector and Dependence Cone Computations with Linear Systems

In the first part, the dependence polyhedron is reduced to a set of dependence direction vectors as defined by Wolfe in [21]. These are computed by intersections and feasibility tests.

In the second part, the dependence polyhedron representation is changed. Its system of equalities and inequalities is transformed into a generating system with vertices, rays and lines. This generating system is basic for the **dependence cone** computation. For a DO-loop body, the cone is equivalent to the transitive closure of all dependence relations for all pairs of statements (S1, S2) and all pairs of references (R1, R2). So, all dependences for a DO-loop body can be defined by one dependence cone.

In part three, we show that dependence direction vectors and dependence cones are both cones of the dependence space and that it is easy to transform one into the other and reciprocally. But the function transforming a dependence cone into a dependence direction vector has no inverse function because it looses information. Dependence cones are more accurate than dependence direction vectors.

3.1. Dependence Direction Vector Computation

The dependence direction vector is defined in [21] as a vector of elements taking their value in the set $\{<, =, >\}$. These comparison symbols are used to express approximate possible directions for the dependence distance vectors, by comparing their coefficients to 0. For instance, the meaning of direction vector $(<, =, >)$ is:

$$d_1 > 0 \qquad d_2 = 0 \qquad d_3 < 0$$

Note that Wolfe's choice for his comparison symbols is opposite to ours.

Thus a DDV can be seen as a cone whose faces are hyperplanes orthogonal to basis vectors. The basic comparison symbol set is enlarged with other symbols like $\geq$, $\leq$, $\neq$ (different) and * (no information) to define an internal combination operation on dependence direction vector which is equivalent to their convex hull. The different symbol $\neq$ let Wolfe define a non convex set which is not a cone but which does not seem to provide more useful information than * for program transformations.

A dependence arc is labeled with a dependence direction vector if there exists at leats one dependence distance vector in the cone defined by the direction vector. If this is true, the intersection of the dependence polyhedron and the DDV cone cannot be empty. This condition is easy to test for any polyhedron Π and any dependence direction vector since both are linear systems. The two systems are put together and the consistency of the new system is tested.

For example, $\Pi_{1,2}$ and the direction vector $(<, >)$ are combined in:

$$\begin{cases} d_i \geq 1 \\ d_i + d_j \geq 1 \\ d_i > 0 \\ d_j < 0 \end{cases}$$

which is feasible ($d_i = 2$ and $d_j = -1$ is a solution).

The results of this test for all possible dependence direction vectors[vi] and for $\Pi_{1,2}$ and $\Pi_{1,3}$ are summarized in table 1 for lexico-positive direction vectors. The hierarchical test sequence described in [5] can be applied to gain time. So the dependence arc due to the first two references to Y (see program 2) is labeled with the combined DDV $(<,*)$ and the arc due to the first and third ones by $(<,>)$.

[vi] The $(=,=)$ vector is irrelevant because an iteration cannot depend on itself. Lexicographically negative vectors are either irrelevant when both references are definitions because an iteration cannot depend on an iteration executed afterwards or change a data dependence into an anti-dependence (and vice-versa) when one reference is a definition and the other a use. The set of lexico-positive vectors depends on the relative position of the two statements in the DO-loop body, unless statement numbers are introduced in iteration vectors and distance vectors.

	$\Pi_{1,2}$	$\Pi_{1,3}$
$(<,<)$	yes	no
$(<,=)$	yes	no
$(<,>)$	yes	yes
$(=,<)$	no	no
	$(<,*)$	$(<,>)$

Table 1: Dependence Direction Vector Existence

Each large dependence system is projected only once for a pair of references and only small subsystems are tested consistent for each potential direction vector. No tests at all are performed when Π is not consistent.

3.2. Dependence Cone Computation

Linear systems define convex polyhedra which can also be defined by generating systems. A generating system is a triplet of sets: a set of vertices, $\vec{v}_1$, $\vec{v}_2$,... , a set of rays, $\vec{r}_1$, $\vec{r}_2$,... and a set of lines $\vec{l}_1$, $\vec{l}_2$, ... A point $\vec{x}$ belongs to the corresponding convex polyhedron if:

$$\vec{x} = \sum \lambda_i \vec{v}_i + \sum \mu_j \vec{r}_j + \sum \nu_k \vec{l}_k$$

under the constraints:

$$\begin{cases} \sum & \lambda_i = 1 \\ \forall\, i & \lambda_i \geq 0 \\ \forall\, j & \mu_j \geq 0 \end{cases}$$

Generating systems and linear systems of equalities and inequalities define the same set, the set of convex polyhedra. Algorithms to convert generating systems into systems of inequalities and vice-versa exist[17] .

For example, the system $\Pi_{1,2}$ is transformed into a generating system with one vertex and two rays while $\Pi_{1,3}$ is a simple line (see figure 3):

$$\Pi_{1,2} = \left(\left\{ \begin{pmatrix} 1 \\ 0 \end{pmatrix} \right\}, \left\{ \begin{pmatrix} 1 \\ -1 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \end{pmatrix} \right\}, \emptyset \right)$$

$$\Pi_{1,3} = \left(\left\{ \begin{pmatrix} 0 \\ 0 \end{pmatrix} \right\}, \emptyset, \left\{ \begin{pmatrix} 1 \\ -1 \end{pmatrix} \right\} \right)$$

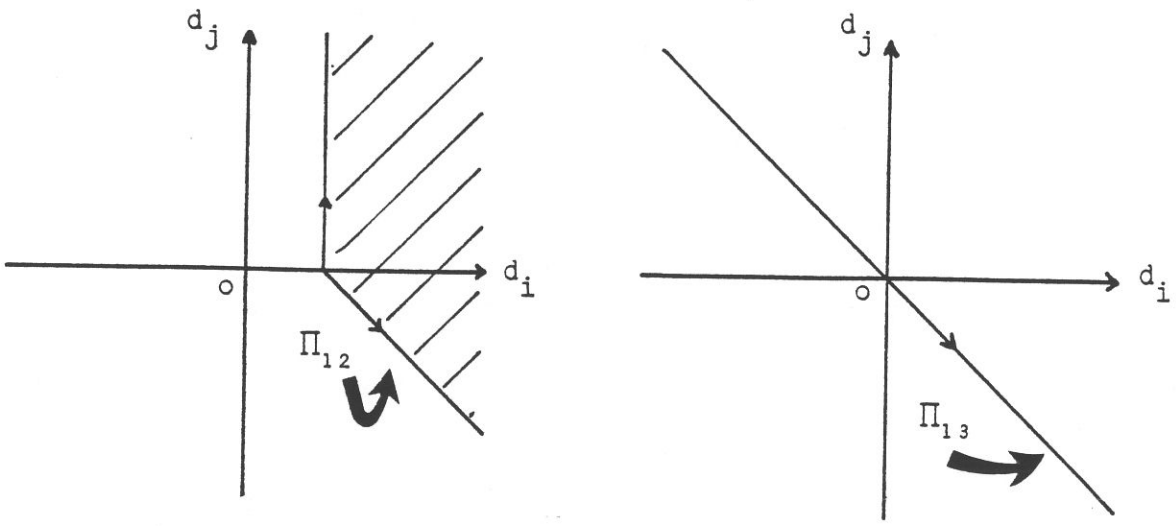


Figure 3. Generating Systems and Convex Polyhedra

Dependence Polyhedron and Lexicographic Order

Dependence nature and existence cannot be decided without studying the lexico-positivity of points in the dependence polyhedron Π.

If R1 is a definition and R2 a use, lexico-positive points of Π correspond to data dependences from S1 to S2, while lexico-negative points correspond to anti dependences from S2 to S1. If R1 and R2 are both definitions the dependence is always an output one, and lexico-positive points correspond to an arc from S1 to S2 and others to an arc from S2 to S1.

As a consequence, several arcs may be generated by a single dependence system but all of them are labeled with a lexico-positive polyhedron, i.e. a polyhedron whose points are all lexico-positive. In case the initial polyhedron is not strictly lexico-positive or strictly lexico-negative[vii], we have to split it into two smaller polyhedra: a lexico-positive one and a lexico-negative one.

We do not have a direct test to check if a polyhedron is lexico-positive, except if all generating elements are lexico-positive. This happens in most cases with usual programs.

In the general case, the lexico-positive part of a polyhedron is computed by intersecting it with each elementary linear condition of the lexico-positive order, and unioning all intersections. The lexico-negative

[vii] In this case, the *opposite* polyhedron is computed by negating all generating elements.

part is computed the same way and then is negated.

Here are the elementary conditions for a 3-D lexicographic order:

lexico-positive conditions	lexico-negative conditions
$d_i > 0$	$d_i < 0$
$d_i = 0$ and $d_j > 0$	$d_i = 0$ and $d_j < 0$
$d_i = 0$ and $d_j = 0$ and $d_k > 0$	$d_i = 0$ and $d_j = 0$ and $d_k < 0$

These complex computations can sometimes be simplified. For example, the lexico-positive part of a line is obtained by replacing in the generating system its direction vector by a ray and the smallest lexico-positive point.

Let examine our example. There are no problems with $\Pi_{1,2}$ whose generating elements are lexico-positive: $\Pi_{1,2}$ is a data dependence. But this is not true for $\Pi_{1,3}$ which contains a line, i.e. two opposite rays. $\Pi_{1,3}$ contains two dependences as can be seen with:

$$j_1 = \begin{pmatrix} 2 \\ 1 \end{pmatrix} \qquad j_2 = \begin{pmatrix} 1 \\ 2 \end{pmatrix}$$

Iteration j_1 must write Y(3) after j_2's read. And j_2's write must occur before j_1's read. Read/Write and Write/Read conflicts can be tested simultaneously because only one statement is considered. Because the same reference appears twice, the Write/Write conflict is also tested at no cost.

$\Pi_{1,3}$ must be intersected with the two polyhedra whose union define the lexico-positive part of the plane:

$$\Pi_{1,3} \bigcap \left\{ \vec{d} \;/\; d_i = 0 \quad d_j > 0 \right\} = \emptyset$$

$$\Pi_{1,3} \bigcap \left\{ \vec{d} \;/\; d_i \geq 1 \right\} = \left(\left\{ \begin{pmatrix} 1 \\ -1 \end{pmatrix} \right\}, \left\{ \begin{pmatrix} 1 \\ -1 \end{pmatrix} \right\}, \emptyset \right)$$

Since the first intersection is empty, the last generating system defines the lexico-positive part of $\Pi_{1,3}$ denoted $\Pi_{1,3}^{+}$ (see figure 4).

Now let's consider program 3. The dependence system for references R1 and R2 of statement S leads to the following dependence polyhedron:

$$d_i + d_j + d_k = 0$$

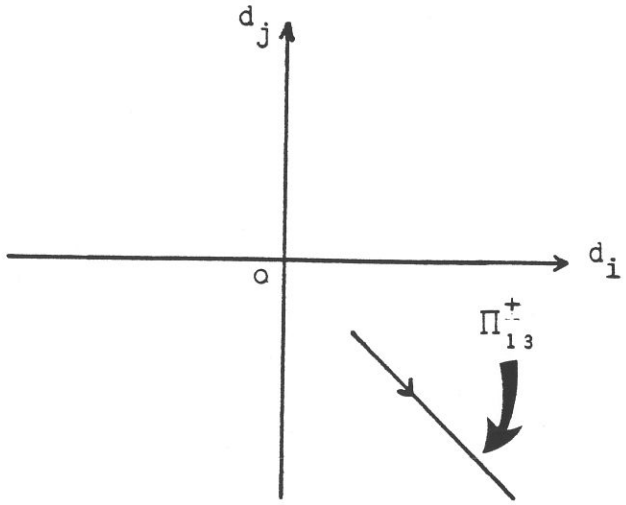


Figure 4. Lexico-positive Part of $\Pi_{1,3}$

```
      DO I = 1, N
        DO J = 1, M
          DO K = 1, P
S           A(I+J+K) = f(A(I+J+K))
```

Program 3: a Dependence Plane

which can also be expressed by the generating system:

$$\left(\left\{\begin{pmatrix}0\\0\\0\end{pmatrix}\right\}, \emptyset, \left\{\begin{pmatrix}1\\0\\-1\end{pmatrix}\begin{pmatrix}0\\1\\-1\end{pmatrix}\right\}\right)$$

Its intersections with lexico-positive constraints are:

	d_i	d_j	d_k	
Π_1	$>$	*	*	$d_i > 0 \quad d_i + d_j + d_k = 0$
Π_2	$=$	$>$	*	$d_j > 0 \quad d_j = -dk \quad d_i = 0$
Π_3	$=$	$=$	$>$	*non-consistent*

Π_3 is empty, while Π_1 and Π_2 can be expressed as generating systems:

$$\Pi_1 = \left(\left\{\begin{pmatrix}1\\-1\\0\end{pmatrix}\right\}, \left\{\begin{pmatrix}1\\-1\\0\end{pmatrix}\right\}, \left\{\begin{pmatrix}0\\1\\-1\end{pmatrix}\right\}\right)$$

$$\Pi_2 = \left(\left\{ \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \right\}, \emptyset, \left\{ \begin{pmatrix} 0 \\ 1 \\ -1 \end{pmatrix} \right\} \right)$$

The convex hull of Π_1 and Π_2, which is the convex hull of the lexico-positive subset of the dependence polyhedron, can be generated by:

$$\Pi_{P3} = \left(\left\{ \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \right\}, \left\{ \begin{pmatrix} 1 \\ -1 \\ 0 \end{pmatrix} \right\}, \left\{ \begin{pmatrix} 0 \\ 1 \\ -1 \end{pmatrix} \right\} \right)$$

which contains one lexico-positive ray and one line. Some accuracy is lost when the parts are combined and replaced by their convex hull.

Dependence Cone

The goal of the dependence cone is to study how iterations of perfectly nested DO-loops can be re-ordered. The DO-loop body is considered as a single statement, and no transformations are applied on it.

An iteration $\vec{j}_1$ must be executed after an iteration $\vec{j}_n$ if there exists a list of statement iterations:[viii]

$$S_1(\vec{j}_1),\ S_2(\vec{j}_2),\ \cdots,\ S_k(\vec{j}_k),\ \cdots,\ S_n(\vec{j}_n)$$

such that $S_k(\vec{j}_k)$ is dependent on $S_{k-1}(\vec{j}_{k-1})$ because of a couple of array references c_k, for k in [2,n].

Let $\vec{d_k} = \vec{j}_k - \vec{j}_{k-1}$, we have:

$$\vec{d} = \vec{j}_n - \vec{j}_1 = \sum_{k=2}^{n} \vec{d}_k$$

where $\vec{d}_k$ belongs to the elementary dependence polyhedron Π_{c_k}, and can be written as:

$$\vec{d}_k = \sum_{\alpha} \lambda_\alpha^k \vec{v}_\alpha^{c_k} + \sum_{\beta} \mu_\beta^k \vec{r}_\beta^{c_k} + \sum_{\gamma} \nu_\gamma^k \vec{l}_\gamma^{c_k}$$

the dependence distance vector $\vec{d}$ can be rewritten as:

$$\vec{d} = \sum_{k=2}^{n} \sum_{\alpha} \lambda_\alpha^k \vec{v}_\alpha^{c_k} + \sum_{k=2}^{n} \sum_{\beta} \mu_\beta^k \vec{r}_\beta^{c_k} + \sum_{k=2}^{n} \sum_{\gamma} \nu_\gamma^k \vec{l}_\gamma^{c_k}$$

We can reorder the three parts of the previous expression by grouping all terms produced by the same couple c of array references. We obtain:

$$\vec{d} = \sum_{c} \sum_{\alpha} \left(\sum_{\substack{k \\ c_k = c}} \lambda_\alpha^k \right) \vec{v}_\alpha^{c} + \sum_{c} \sum_{\beta} \left(\sum_{\substack{k \\ c_k = c}} \mu_\beta^k \right) \vec{r}_\beta^{c} + \sum_{c} \sum_{\gamma} \left(\sum_{\substack{k \\ c_k = c}} \nu_\gamma^k \right) \vec{l}_\gamma^{c}$$

[viii] $S(\vec{j})$ denotes iteration $\vec{j}$ of statement S.

Let define new coefficients:

$$x_\alpha^c = \sum_{\substack{k \\ c_k=c}} \lambda_\alpha^k \qquad y_\beta^c = \sum_{\substack{k \\ c_k=c}} \mu_\beta^k \qquad z_\gamma^c = \sum_{\substack{k \\ c_k=c}} \nu_\gamma^k$$

There are no constraints on coefficients z_γ^c. Coefficients y_β^c are positive since all μ_β^k are positive. Finally, $\left(\sum_c \sum_\alpha x_\alpha^c\right)$ is greater than 1 since each term $\left(\sum_\alpha \lambda_\alpha^k\right)$ is equal to 1 for every k.

Thus we can find two sets $\left\{\overline{x}_\alpha^c\right\}$ and $\left\{\overline{\overline{x}}_\alpha^c\right\}$ such that:

$$\begin{cases} x_\alpha^c = \overline{x}_\alpha^c + \overline{\overline{x}}_\alpha^c \\ \sum \overline{x}_\alpha^c = 1 \\ \overline{x}_\alpha^c \geq 0 \text{ and } \overline{\overline{x}}_\alpha^c \geq 0 \end{cases}$$

$\vec{d}$ can be rewritten as:

$$\vec{d} = \sum_c \sum_\alpha \overline{x}_\alpha^c \vec{v}_\alpha^c + \sum_c \sum_\alpha \overline{\overline{x}}_\alpha^c \vec{v}_\alpha^c + \sum_c \sum_\beta y_\beta^c \vec{r}_\beta^c + \sum_c \sum_\gamma z_\gamma^c \vec{l}_\gamma^c$$

We recognize the usual way of writing that $\vec{d}$ belongs to the convex polyhedron C whose generating system is:

$$\left(\left\{\vec{v}_\alpha^c\right\}, \left\{\vec{v}_\alpha^c\right\} \cup \left\{\vec{r}_\beta^c\right\}, \left\{\vec{l}_\gamma^c\right\}\right)$$

This convex polyhedron is called the *Dependence Cone.*

The generating system computed above for C is not canonical: some elements may be redundant and lines may be hidden in rays. However, algorithms to eliminate redundancy from generating systems are presented in [17].

The dependence cone summarize the dependence relations with which the parallel execution ordering must be compatible. It may contain too many elements but for any $\vec{j}_2$ depending on any $\vec{j}_1$, $\vec{j}_2 - \vec{j}_1 \in C$.

Examples

For instance, DO-loops of program 2 are characterized by the following cone:

$$\left.\begin{aligned} \Pi_{1,2}{}^+ &= \left(\left\{\begin{pmatrix}1\\0\end{pmatrix}\right\}, \left\{\begin{pmatrix}1\\-1\end{pmatrix}, \begin{pmatrix}0\\1\end{pmatrix}\right\}, \emptyset\right) \\ \Pi_{1,1}{}^+ = \Pi_{1,3}{}^+ &= \left(\left\{\begin{pmatrix}1\\-1\end{pmatrix}\right\}, \left\{\begin{pmatrix}1\\-1\end{pmatrix}\right\}, \emptyset\right) \end{aligned}\right\} \rightarrow C_2 = \left(\left\{\begin{pmatrix}1\\0\end{pmatrix}, \begin{pmatrix}1\\-1\end{pmatrix}\right\}, \left\{\begin{pmatrix}1\\-1\end{pmatrix}, \begin{pmatrix}0\\1\end{pmatrix}\right\}, \emptyset\right)$$

This generating system can be made non redundant:

$$C_2 = \left(\left\{ \begin{pmatrix} 1 \\ 0 \end{pmatrix} \right\}, \left\{ \begin{pmatrix} 1 \\ -1 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \end{pmatrix} \right\}, \emptyset \right)$$

This DC is drawn on figure 5-a for a particular point $\vec{j}_1$. It defines the set of points $\vec{j}_2$ which might be dependent on $\vec{j}_1$. The DDVs for the same program are drawn on figure 5-b&c. One can see that the DC is more accurate than the union of the 2 DDVs (the half-space $d_i > 0$).

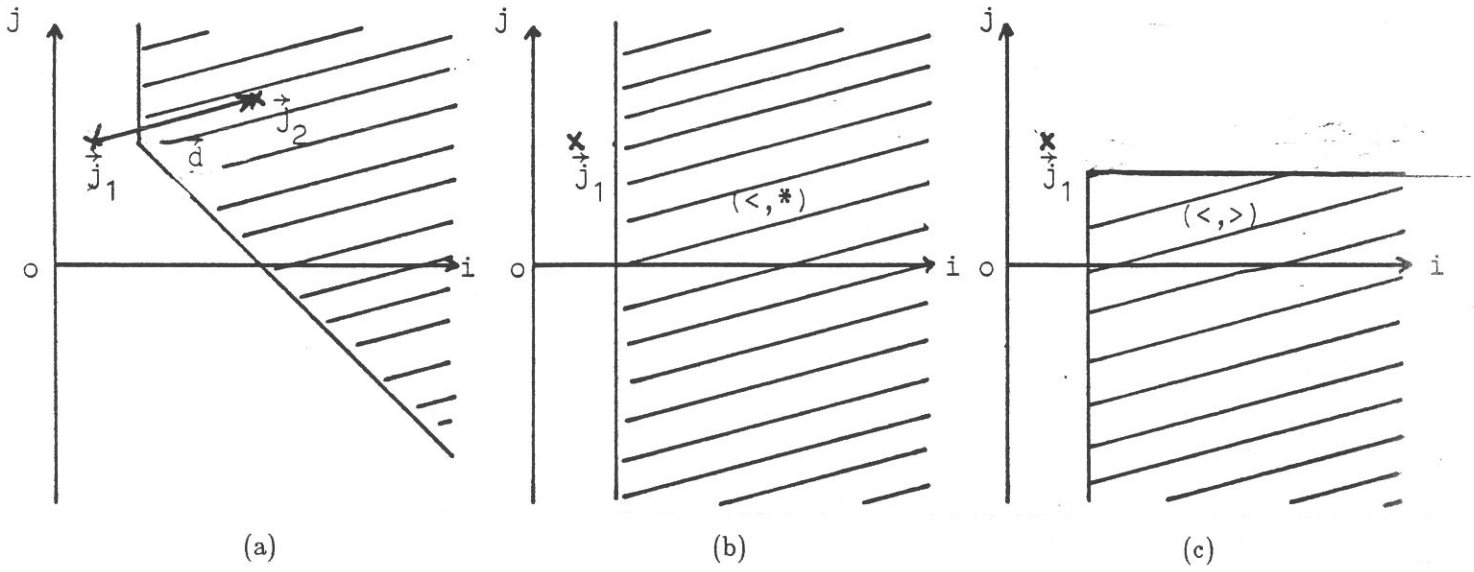


Figure 5. DC and DDVs for Program 2

Another example to show that points which are dependent on each other because of transitivity are taken into account. The following constant dependence polyhedron[ix], computed for program 4, is changed into the following cone:

$$\Pi_4 \left(\left\{ \begin{pmatrix} 1 \\ -1 \end{pmatrix}, \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \end{pmatrix} \right\}, \emptyset, \emptyset \right) \rightarrow C_4 = \left(\left\{ \begin{pmatrix} 1 \\ -1 \end{pmatrix}, \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \end{pmatrix} \right\}, \left\{ \begin{pmatrix} 1 \\ -1 \end{pmatrix}, \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \end{pmatrix} \right\}, \emptyset \right)$$

which can be simplified into:

$$C_4 = \left(\left\{ \begin{pmatrix} 1 \\ -1 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \end{pmatrix} \right\}, \left\{ \begin{pmatrix} 1 \\ -1 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \end{pmatrix} \right\}, \emptyset \right)$$

The dependence polyhedron and the corresponding cone are shown on figure 6.

[ix] such polyhedra are called **polytopes.**

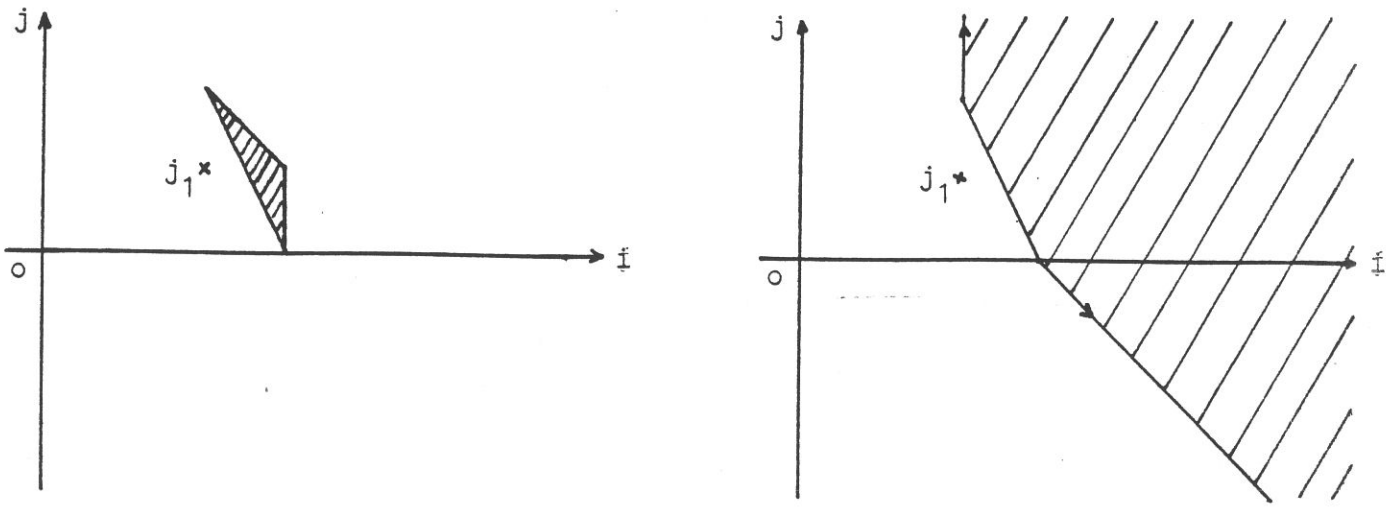


Figure 6. A Constant Dependence Polyhedron and its Dependence Cone

```
DO I = 1, N
  DO J = 1, M
    A(I,J) = f(A(I-1,J),A(I,J-1),A(I-1,J+1))
```

Program 4. Constant Dependence Distance Vectors

```
      DO I = 1, N
        DO J = 1, M
S         A(I,J) = f(A(I, J-1), A(I-1, N))
```

Program 5: Dependence Cone Simplification

A last example to show redundance elimination is not that simple. Consider program 5 which contains 3 references in statement S. There is no output dependence on A(I,J). There is a constant data dependence with A(I, J) and A(I, J–1):

$$\Pi_{1,2} = \left(\left\{ \begin{pmatrix} 0 \\ 1 \end{pmatrix} \right\}, \emptyset, \emptyset \right)$$

and a broadcast data dependence between A(I,J) and A(I–1, N):

$$\Pi_{1,3} = \left(\left\{ \begin{pmatrix} 1 \\ 0 \end{pmatrix} \right\}, \left\{ \begin{pmatrix} 0 \\ -1 \end{pmatrix} \right\}, \emptyset \right)$$

Although $\Pi_{1,3}$'ray is lexico-negative, $\Pi_{1,3}$ is lexico-positive since the first coordinate of all its points is necessary equal to 1.

As shown previously, the dependence cone is:

$$C_5 = \left(\left\{ \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \end{pmatrix} \right\}, \left\{ \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ -1 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \end{pmatrix} \right\}, \emptyset \right)$$

It contains the origin, and one line hidden in the ray set. After simplification, we obtain:

$$C_5 = \left(\left\{ \begin{pmatrix} 0 \\ 0 \end{pmatrix} \right\}, \left\{ \begin{pmatrix} 1 \\ 0 \end{pmatrix} \right\}, \left\{ \begin{pmatrix} 0 \\ 1 \end{pmatrix} \right\} \right)$$

3.3. Relation Between DDV and DC

Both are polyhedra. They can be represented either by a linear system or by a generating system.

The dependence direction vectors can be kept under the generating form and the intersection with Π is equivalent to a membership test of vertices and rays and lines in Π. Vice-versa, Π can be put into the generating form and its vertices and rays and lines checked for membership to the direction vector.

DDV is optimal for many transformations because it conveys the right amount of information (DO-loop parallelization, DO-loop interchange). This is not true for less straightforward transformations like the hyperplane method[14] or supernode partitioning[10] .

The lack of accuracy of DDV's is shown on figure 7 where three different dependence cones are summarized by the same DDV.

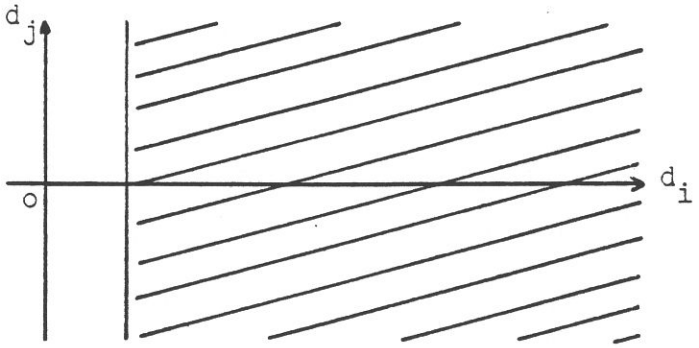
4. Conclusion

We have presented in this report a new way of computing dependence graphs for a nest of DO-loops. This new method is based on linear systems of equalities and inequalities.

We have shown how to express a potential dependence between two statements with a linear system. The two statements are dependent on each other if the system is consistent, and, when it is, it can be used to compute the corresponding dependence direction vectors which summarize the set of possible directions for dependence distance vectors.

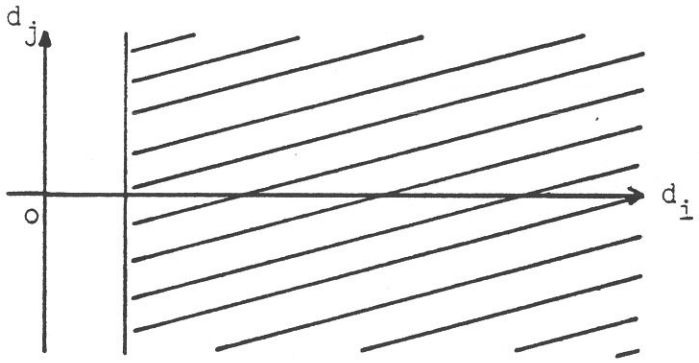
We have introduced a new concept, the dependence cone, to summarize these possible directions, and

DDV = (<, *)



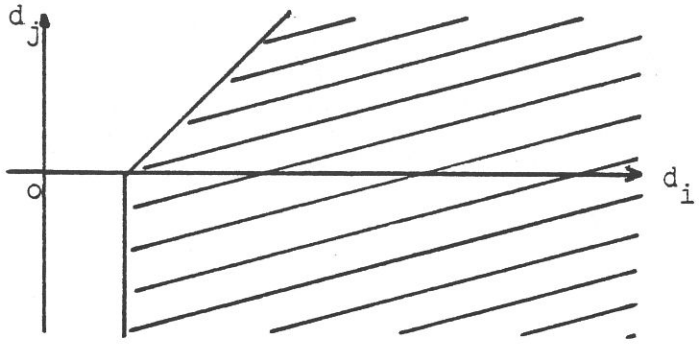
$$\begin{cases} d_i \geq 1 \end{cases}$$

$$\left(\left\{ \begin{pmatrix} 1 \\ 0 \end{pmatrix} \right\}, \left\{ \begin{pmatrix} 1 \\ 0 \end{pmatrix} \right\}, \left\{ \begin{pmatrix} 0 \\ 1 \end{pmatrix} \right\} \right)$$



$$\begin{cases} d_i \geq 1 \\ d_i - d_j \geq 0 \end{cases}$$

$$\left(\left\{ \begin{pmatrix} 1 \\ 0 \end{pmatrix} \right\}, \left\{ \begin{pmatrix} 1 \\ 1 \end{pmatrix} \begin{pmatrix} 0 \\ -1 \end{pmatrix} \right\}, \emptyset \right)$$



$$\begin{cases} d_i + d_j \geq 1 \\ d_i - d_j \geq 0 \end{cases}$$

$$\left(\left\{ \begin{pmatrix} 1 \\ 0 \end{pmatrix} \right\}, \left\{ \begin{pmatrix} 1 \\ 1 \end{pmatrix} \begin{pmatrix} 1 \\ -1 \end{pmatrix} \right\}, \emptyset \right)$$

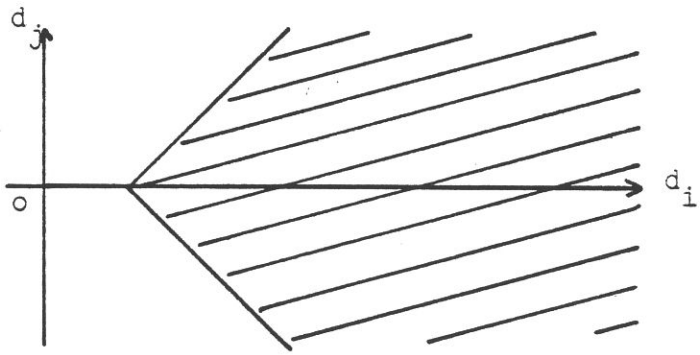


Figure 7. Accuracy of DC versus DDV

we have shown how dependence cones can be computed. The dependence cone provides a more accurate summary of dependence distance vectors than the dependence direction vector. The cone rays provides extreme dependence directions that may not be parallel to the basis vectors. The generating systems of the dependence cone and of the dependence polyhedron are finite description of infinite sets of dependence distance vectors encountered by Kuhn when a datum is broadcasted in the iteration space or when a memory location is used many times.

When constant dependence distance vectors are available, they should be kept as such to apply the minimum distance partitioning[16] because the generating system computation would divide their coordinates by their GCDs. Rays provide information on dependence direction and not on dependence distance. This is the price to pay to summarize data broadcasting.

The main advantage of dependence cones is to provide accurate information on possible directions for dependence vectors. Generating systems provide a representation that does not depend on the set of initial DO-loops, whereas dependence direction vector is based onthem. As a consequence, complex reorderings of computations are possible with dependence cones.

The hyperplane method[14] produces complex reorderings where computations are done simultaneously along hyperplanes of the iteration space. Supernode partioning[11] , a generalization of the hyperplane method, produced much more complex reorderings, where the initial set of computations is decomposed into parallel fronts of computation subsets (supernodes).

The hyperplane method, as well as supernode partitioning, can be applied with dependence direction vectors, but the lack of accuracy of dependence direction vectors prevents some times a possible application and other times limit unduly the range of valid transformation parameters.

Using linear systems in program parallelization and dependence testing is doomed to failure according to many people because of the exponential complexity of many algorithms. Although no implementation exists yet, it seems possible to cut the average cost by using simple techniques when references are simple and by using the full power of the integer linear theory only when necessary.

References

1. J. R. Allen and K. Kennedy, "PFC: a Program to Convert Fortran to a Parallel Form," in *Supercomputers, Design and Application*, ed. K. Hwang (1982). COMPSAC, Tutorial
2. J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren, "Conversion of Control Dependence to Data Dependence," pp. 177-189 in *Conference Record of the Tenth Annual Symposium on Principles of Programming Languages*, Austin (Jan. 1983).
3. C. Ancourt, "Utilisation de systèmes linéaires sur Z pour la parallélisation de programmes," ENSMP-CAI-87-E87, Ecole des Mines de Paris, Fontainebleau (France) (1987).
4. A. J. Bernstein, "Analysis of Programs for Parallel Processing," *IEEE Transactions on Electronic Computers* **15**(5), pp.757-763 (Oct. 1966).
5. M. Burke and R. Cytron, "Interprocedural Dependence Analysis and Parallelization," *SIGPLAN'86 Symposium on Compiler Construction*, pp.162-175 (June 1986).
6. D. Callahan, K. D. Cooper, K. Kennedy, and L.Torczon, "Interprocedural Constant Propagation," *Sigplan Notices* **21**(7), pp.152-161, Proceedings of the SIGPLAN '86 Symposium on Compiler Construction (1986).
7. P. Cousot and N. Halbwachs, "Automatic Discovery of Linear Restraints Among Variables of a Program," *Conference Record of the Tenth Annual Symposium on Principles of Programming Languages* (1978).
8. D. Gannon, "Restructuring Nested Loops on the Alliant Cedar Cluster: A Case Study of Gaussian Elimination of Banded Matrices," CSRD document no. 543, University of Ilinois at Urbana Champaign, analyse numerique / algorithme parallele (Feb. 1986).
9. A. Imadache, "Assertions et parallélisation," Rapport de DEA, ENSMP-CAI-86-E83, Ecole des Mines de Paris, Fontainebleau (France) (1986).
10. F. Irigoin and R. Triolet, "Automatic DO-Loop Partitioning for Improving Data Locality in Scientific Programs," ENSMP-CAI-87-E93, Ecole des Mines de Paris, Fontainebleau (France) (1987).
11. F. Irigoin, "Partitionnement des boucles imbriquées : une technique d'optimisation pour les programmes scientifiques," Thèse de Doctorat d'Université, Université PARIS-VI, PARIS (1987).
12. R. Kuhn, D. J. Kuck, B. Leasure, and M. Wolfe, "The Structure of an Advanced Retargetable Vectorizer," pp. 163-178 in *Supercomputers, Design and Application*, ed. K. Hwang (1984). (revised from COMPSAC'80)
13. R. H. Kuhn, "Optimization and Interconnection Complexity for: Parallel Processors, Single-stage Networks, and Decision Trees," Ph.D. Thesis, No. UIUCDCS-R-80-1722, University of Illinois at Urbana-Champaign (1980).
14. L. Lamport, "The Parallel Execution of DO Loops," *Communications of the ACM* **17**(2), pp.83-93 (Feb. 1974).
15. A. Lichnewsky and F. Thomasset, "Techniques de Base sur l'Exploitation Automatique du Parallélisme dans les Programmes," *Calcul Parallèle à usage scientifique* (Oct. 1985).
16. J.-K. Peir and R. Cytron, "Minimum Distance: A Method for Partitioning Recurrences for Multiprocessors," *1987 Int'l Conference on Parallel Processing*, pp.???-??? (Aug. 1987).
17. A. Schrijver, *Theory of Linear and Integer Programming*, Wiley, Chichester (1986).
18. N. Tawbi, A. Dumay, and P. Feautrier, "PAF: un Paralléliseur Automatique pour Fortran," MASI, rapport 185, Université Pierre et Marie Curie, Paris (Juin 1987).
19. R. Triolet, "Interprocedural Analysis for Program Restructuring with PARAFRASE," Technical Report CSRD-538, University of Illinois, Center for Supercomputing Research & Development, Urbana-Champaign (1985).

20. R. Triolet, F. Irigoin, and P. Feautrier, "Direct Parallelization of CALL Statements," *SIGPLAN'86 Symposium on Compiler Construction*, pp.176-185 (June 1986).

21. M. J. Wolfe, "Optimizing Supercompilers for Supercomputers," PhD. Thesis, Report No. UIUCDCS-R-82-1105, University of Illinois at Urbana-Champaign (1982).