Laboratoire de l'Informatique du Parallélisme

Ecole Normale Supérieure de Lyon
Unité de recherche associée au CNRS n°1398

Optimal fine and medium grain parallelism detection in polyhedral reduced dependence graphs

Alain Darte and Frédéric Vivien

April 1996

Research Report N$^{\text{O}}$ 96-06

Ecole Normale Supérieure de Lyon
46 Allée d'Italie, 69364 Lyon Cedex 07, France
Téléphone : (+33) 72.72.80.00 Télécopieur : (+33) 72.72.80.80
Adresse électronique : lip@lip.ens−lyon.fr

Optimal fine and medium grain parallelism detection in polyhedral reduced dependence graphs

Alain Darte and Frédéric Vivien

April 1996

Abstract

This paper presents an optimal algorithm for detecting fine or medium grain parallelism in nested loops whose dependences are described by an approximation of distance vectors by polyhedra. In particular, it is optimal for direction vectors: this result generalizes, to the case of several statements, Wolf and Lam's algorithm which is optimal for a single statement. Our algorithm relies on a dependence uniformization process and on parallelization techniques related to system of uniform recurrence equations.

Keywords: automatic parallelization, multi-dimensional schedule, loop nest, system of uniform recurrence equations, dependence analysis, polyhedral reduced dependence graph.

Résumé

Ce rapport présente un algorithme de parallélisation de boucles qui est optimal pour toute représentation des dépendances fournissant une approximation polyédrique des vecteurs de distance. En particulier, il est optimal pour les vecteurs de direction ce qui généralise au cas de plusieurs instructions l'algorithme de Wolf et Lam qui est optimal pour les nids ne contenant qu'une seule instruction. Notre algorithme est basé sur un procédé d'uniformisation des dépendances et sur une technique de parallélisation issue des systèmes d'équations récurrentes uniformes.

Mots-clés: parallélisation automatique, ordonnancement multi-dimensionnel, nid de boucles, systèmes d'équations récurrentes uniformes, analyse de dépendances, graphe de dépendance réduit polyédrique.

Optimal fine and medium grain parallelism detection in polyhedral reduced dependence graphs

Alain Darte and Frédéric Vivien*
Laboratoire LIP, URA CNRS 1398
Ecole Normale Supérieure de Lyon, F - 69364 LYON Cedex 07
e-mail: [Alain.Darte,Frederic.Vivien]@lip.ens-lyon.fr

April 1996

Contents

*Supported by the CNRS-INRIA project *ReMaP*.

1 Introduction: a motivating example

Consider the following simple code:

Example 1

```
DO i = 1, n
  DO j = 1, n
    a(i, j) = a(j, i) + a(i, j - 1)          (Statement S)
  ENDDO
ENDDO
```

The exact dependence relations are the following:

$$\begin{cases} \text{if } 1 \le i \le n,\ 1 \le j < n & S(i,j) \xrightarrow{\text{flow}} S(i,j+1) \\ \text{if } 1 \le i < j \le n & S(i,j) \xrightarrow{\text{flow}} S(j,i) \\ \text{if } 1 \le i < j \le n & S(j,i) \xrightarrow{\text{anti}} S(i,j) \end{cases}$$

Let us try to apply well known parallelization algorithms to this code, Allen and Kennedy's algorithm (that uses levels of dependences), Wolf and Lam's algorithm (that uses direction vectors), Darte and Robert's algorithm, and Feautrier's algorithm (that both use exact dependences). Figure 1 shows the corresponding (reduced) dependence graphs when dependence edges are labeled respectively with levels and direction vectors.

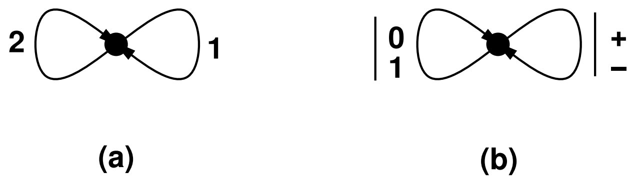

Figure 1: Reduced Dependence Graph for Example 1: (a) by levels, (b) by direction vectors.

Allen and Kennedy [AK87] The basic technique of the algorithm is the decomposition of the reduced dependence graph into strongly connected components. Here, the levels of the three dependences are respectively 2, 1, and 1. There is a dependence cycle at depth 1 and at depth 2. Therefore, no parallelism is detected.

Wolf and Lam [WL91] The algorithm is based on a cone separation technique adapted to the case of direction vectors. Here, the respective dependence vectors are $(0,1)$, $(+,-)$, and $(+,-)$. In the second dimension, the 1 and the $-$ prevents to detect two levels of fully permutable loops. Therefore, the code remains unchanged. No parallelism is detected.

Darte and Robert [DR94, DR95] Darte and Robert look for an affine schedule for each statement that satisfies all dependences. Exact dependence analysis is needed, and a quite large linear system (obtained by the duality theorem of linear programming) has to be solved. This technique leads to the valid schedule $T(i,j) = 2i + j - 3$. One level of parallelism is detected.

Feautrier [Fea92a, Fea92b] Feautrier's technique is similar to Darte and Robert's technique for the one-dimensional case (except that the linear program is obtained by the affine form of Farkas' lemma). Here, the same schedule is found, $T(i,j) = 2i + j - 3$. However, Feautrier's algorithm is more general, since it is able to derive multi-dimensional affine schedule when no one-dimensional schedule exists. So far, Feautrier's algorithm is indeed the most powerful algorithm for parallelism detection in nested loops.

In this particular example, the representation of dependences by level and by direction vectors is not accurate enough to reveal parallelism, this is the reason why Allen and Kennedy's algorithm, and Wolf and Lam's algorithm are not able to detect any parallelism. Exact dependence analysis, associated to linear programming methods that require to solve large[1] parametric linear programs to be solved, permits to reveal one degree of parallelism. The corresponding parallelized code is:

DO $j = 3, 3n$
  DOPAR $i = \max\left(1, \lceil \frac{j-n}{2} \rceil\right), \min\left(n, \lfloor \frac{j-1}{2} \rfloor\right)$
    $\mathrm{a}(i,\, j-2i) = \mathrm{a}(j-2i,\, i) + \mathrm{a}(i,\, j-2i-1)$
  ENDDO
ENDDO

However, there is a large gap between the complexity of the two first algorithms and the complexity of the two last algorithms, both in terms of dependence abstractions and in terms of running complexity. The goal of this paper is to fill this gap and to propose an intermediate algorithm, thus of medium complexity but still optimal for all classical approximations of dependences, namely approximations of distance vectors by polyhedra.

In Example 1, exact dependence analysis is indeed not necessary to reveal maximal parallelism. One has just to notice that there is one uniform dependence $d = (0,1)$ and a set of distance vectors $\{(j-i, i-j) = (j-i)(1,-1) \mid 1 \leq j-i \leq n-1\}$ that can be approximated by the set $P = \{(1,-1) + \lambda(1,-1) \mid 0 \leq \lambda\}$. P is a polyhedron with one vertex $v = (1,-1)$ and one ray $r = (1,-1)$.

Suppose that, as in the above algorithms, we are looking for a linear schedule $T(i,j) = x_1 i + x_2 j$. For T to be a valid schedule, we impose that $X(0,1) \geq 1$ and $Xp \geq 1$ for all $p \in P$, where $X = (x_1, x_2)$. Instead of using Farkas' lemma to solve the second constraint, we can remark that it is equivalent to $Xv \geq 1$ and $Xr \geq 0$. Therefore, one has just to solve the three inequalities:

$$Xd \geq 1 \quad Xv \geq 1 \quad Xr \geq 0 \quad \text{i.e.} \quad X\begin{pmatrix} 0 \\ 1 \end{pmatrix} \geq 1 \quad X\begin{pmatrix} 1 \\ -1 \end{pmatrix} \geq 1 \quad X\begin{pmatrix} 1 \\ -1 \end{pmatrix} \geq 0$$

which leads, as above, to $X = (2,1)$. In other words, we have "uniformized" the constraints and transformed the underlying affine scheduling problem into a simple scheduling problem where all dependences are uniform (d, v, and r). However, compared to the classical framework of uniform loop nests, there are two fundamental differences:

- the uniform dependence vectors are not necessarily lexico-positive (a ray equal to $(0,-1)$ for example is possible). This makes the problem a lot more difficult, but it can be solved by techniques related to the problem of computability of a system of uniform recurrence equations.
- the constraint imposed for a ray r is weaker: the constraint is $Xr \geq 0$ instead of $Xr \geq 1$. This freedom must be taken into account when deriving the parallelization algorithm.

[1] the number of inequalities and variables is related to the number of constraints that define the validity domain of each dependence relation.

To better understand this “uniformization” principle, think in terms of dependence path: actually, we consider an edge e labeled by the distance vector $p = v + \lambda r$ as a path ϕ that uses once the “uniform” dependence vector v and λ times the “uniform” dependence vector r. However, we consider that the use of the dependence r counts for 0 instead of 1 (constraint $Xr \geq 0$ instead of $Xr \geq 1$) when defining the length of the path ϕ so that e and its equivalent path ϕ have same length. This simulation is summarized in Figure 2: we introduce a new node S' that permits to simulate ϕ and a null-weight edge to come back to the initial node S.

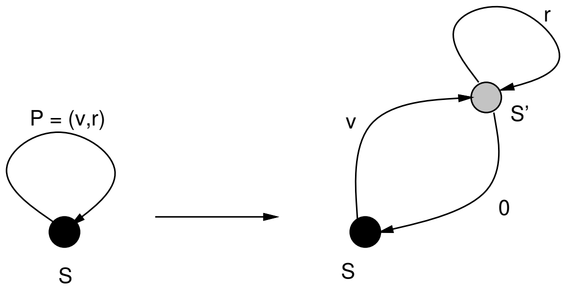

Figure 2: Simulation of an edge labeled by a polyhedron with one vertex and one ray.

This “uniformization” principle is the underlying idea of the loop parallelization algorithm proposed in this paper. This algorithm has the following properties:

- It does not requires exact dependence analysis, but it is optimal for dependence graph whose edges are labeled by a polyhedral approximation of distance vectors. In particular, it is optimal for level of dependences and direction vectors. Actually, it behaves exactly as Allen and Kennedy's algorithm when dependences are expressed by dependence levels and it generalizes Wolf and Lam's algorithm [WL91] to the case of multiple statements when dependences are expressed by direction vectors (Wolf and Lam's algorithm is optimal if there is **only one** statement).

- It points out precisely which dependences prevent the parallelization or are responsible for a loss of parallelism. This permits to better understand the link between the maximal degree of parallelism that can be detected and the accuracy of dependence abstractions. See [DV95b] for a complete study about this question.

- By construction, it can be naturally adapted to the search for maximal sets of fully permutable loops which is, in theory, an equivalent problem, and, in practice, a way to exploit medium-grain parallelism.

The paper is organized as follows. In Section 2, we recall generalities on dependence analysis and dependence graphs. We formally define what we call polyhedral reduced dependence graphs, and we demonstrate the expressive power of this dependence abstraction.

In Section 3, we give an overview of the different steps of the parallelization algorithm, for perfect nested loops. We illustrate its capabilities on an ad-hoc example. The proofs of correctness and optimality of the algorithm are detailed in the two next sections: Section 4 is devoted to the problem of computability of the dependence graph while Section 5 addresses the scheduling problem.

Finally, in Section 6, we discuss some implementation strategies that permit to reduce the complexity of the algorithm and we briefly show how it can be naturally extended to non perfect loop nests, while keeping the optimality result. We conclude in Section 7.

2 Dependence abstractions

For the sake of clarity, we restrict ourselves to the case of perfectly nested **DO** loops with affine loop bounds. Non perfectly nested loops are considered in Section 6.5. This restriction permits to identify, as usual, the iterations of n nested loops (n is called the **depth** of the loop nest) with vectors in $\mathbb{Z}^n$ (called the **iteration vectors**) contained in a finite convex polyhedron bounded by the loop bounds (called the **iteration domain**). The i-th component of an iteration vector is the value of the i-th loop counter in the nest, counting from the outermost to the innermost loop. In the sequential code, the iterations are therefore executed in the lexicographic order of their iteration vectors.

In the next sections, we denote by $\mathcal{D}$ the polyhedral iteration domain, by I and J n-dimensional iteration vectors in $\mathcal{D}$, and by S_i the i-th statement in the loop nest. We write $I >_l J$ if I is lexicographically greater than J and $I \geq_l J$ if $I >_l J$ or $I = J$.

Section 2.1 recalls the different concepts of dependence graphs: expanded dependence graphs (EDG), reduced dependence graphs (RDG), apparent dependence graphs (ADG) and the notion of distance sets. In Section 2.2, we formally define what we call polyhedral reduced dependence graphs (PRDG), i.e. reduced dependence graphs whose edges are labeled by polyhedra. Finally, in Section 2.3, we show how the model of PRDG generalizes classical dependence abstractions of distance sets.

2.1 Dependence graphs and distance sets

Dependence relations between operations are defined by Bernstein's conditions [Ber66]. Briefly speaking, two operations are considered dependent if both operations access the same memory location and if at least one of the accesses is a write. The dependence is directed according to the sequential order, from the first executed operation to the last. Depending on the order of write(s) and/or read, the dependence corresponds to the so called **flow dependence**, **anti dependence** or **output dependence** [2]. We write: $S_i(I) \Longrightarrow S_j(J)$ if statement S_j at iteration J depends on statement S_i at iteration I. The partial order defined by $\Longrightarrow$ describes the **expanded dependence graph (EDG)**. Note that $(J - I)$ is always lexicographically non negative when $S_i(I) \Longrightarrow S_j(J)$.

In general, the EDG can not be computed at compile-time, either because some information is missing (such as the values of size parameters or even worse, precise memory accesses), or because generating the whole graph is too expensive. Instead, dependences are captured through a smaller (in general) cyclic directed graph, with s vertices, called the **reduced dependence graph (RDG)** (or statement level dependence graph).

The RDG is a compression of the EDG. In the RDG, two statements S_i and S_j are said dependent (we write $S_i \rightarrow S_j$) if there exists at least one pair (I, J) such that $S_i(I) \Longrightarrow S_j(J)$. Furthermore, the dependence $S_i \xrightarrow{e} S_j$ is labeled by the set $\{(I, J) \in \mathcal{D}^2 \mid S_i(I) \Longrightarrow S_j(J)\}$, or by an approximation D_e that contains this set. The precision and representation of this approximation makes the power of the dependence analysis.

In other words, the RDG describes, in a condensed manner, an iteration level dependence graph, called (maximal) **apparent dependence graph (ADG)**, that is a superset of the EDG. The ADG and the EDG have the same vertices, but the ADG has more edges, defined by:

$$(S_i, I) \Longrightarrow (S_j, J) \text{ (in the ADG) } \Leftrightarrow \exists\, e = (S_i, S_j) \text{ (in the RDG) such that } (I, J) \in D_e.$$

[2]In some cases, output and anti dependences can be removed by data renaming and/or expansion. See for example [Fea91].

For a certain class of nested loops, it is possible to express exactly this set of pairs (I, J) (see [Fea91]): I is given as an affine function $f_{i,j}$ of J where J varies in a polyhedron $\mathcal{P}_{i,j}$:

$$\{(I, J) \in \mathcal{D}^2 \mid S_i(I) \Longrightarrow S_j(J)\} = \{(f_{i,j}(J), J) \mid J \in \mathcal{P}_{i,j} \subset \mathcal{D}\} \tag{1}$$

In most dependence analysis algorithms however, rather than the set of pairs (I, J), one computes the set $E_{i,j}$ of all possible values $(J - I)$. $E_{i,j}$ is called the set of **distance vectors**, or **distance set**:

$$E_{i,j} = \{(J - I) \mid S_i(I) \Longrightarrow S_j(J)\}$$

When exact dependence analysis is feasible, Equation 1 shows that the set of distance vectors is the projection of the integer points of a polyhedron. This set can be approximated by its convex hull or by a more or less accurate description of a larger polyhedron (or a finite union of polyhedra). When the set of distance vectors is represented by a finite union, the corresponding dependence edge in the RDG is decomposed into multi-edges.

Note that the representation by distance vectors is not equivalent to the representation by pairs (as in Equation 1), since the information concerning the **location** in the EDG of such a distance vector is lost. This may even be the cause of a loss of parallelism[3]. However, this representation remains important, especially when exact dependence analysis is either too expensive or not feasible.

Classical representations of distance sets (by increasing precision) are:

- **level of dependence**, introduced in [AK82, AK87] for Allen and Kennedy's parallelizing algorithm.
- **direction vector**, introduced by Wolfe in [Wol82, Wol89] and used in Wolf and Lam's parallelizing algorithm ([WL91]).
- **dependence polyhedron**, introduced in [IT87] and used in Irigoin and Triolet's supernode partitioning algorithm ([IT88]). We refer to the PIPS ([IJT91]) software for more details on dependence polyhedra.

In the rest of the paper, we explore this last representation. We now first define formally reduced dependence graph whose edges are labeled by dependence polyhedra and we show the expressive power of this model.

2.2 Polyhedral reduced dependence graphs (PRDG)

We first recall the mathematical definition of a polyhedron and how it can be decomposed into vertices, rays and lines.

Definition 1 (Polyhedron, polytope)

A set P of vectors in $\mathbb{Q}^n$ is called a (convex) polyhedron if there exists an integral matrix A and an integral vector b such that:

$$P = \{x \mid x \in \mathbb{Q}^n,\ Ax \leq b\}$$

A polytope is a bounded polyhedron.

[3] See [DV95b], Example 6, for an illustration of this fact

A polyhedron can always be decomposed as the sum of a (convex) polytope and of a polyhedral cone (for more details see [Sch86]). A polytope is defined by its vertices, and any point of the polytope is a non-negative barycentric combination of the polytope vertices. A polyhedral cone is finitely generated and can be defined by its rays and lines. Any point of a polyhedral cone is the sum of a non-negative combination of its rays and of any combination of its lines.

Therefore, a convex dependence polyhedron P can be equivalently defined by a set of *vertices* (denoted by $\{v_1, \ldots, v_\omega\}$), a set of *rays* (denoted by $\{r_1, \ldots, r_\rho\}$), and a set of *lines* (denoted by $\{l_1, \ldots, l_\lambda\}$). Then, P is the set of all vectors p such that:

$$p = \sum_{i=1}^{\omega} \mu_i v_i + \sum_{i=1}^{\rho} \nu_i r_i + \sum_{i=1}^{\lambda} \xi_i l_i \text{ with } (\mu, \nu, \xi) \in (\mathbb{Q}^+)^\omega \times (\mathbb{Q}^+)^\rho \times \mathbb{Q}^\lambda \text{ and } \sum_{i=1}^{\omega} \mu_i = 1 \qquad (2)$$

We now define what we call a polyhedral reduced dependence graph (or PRDG), i.e. a reduced dependence graph labeled by dependence polyhedra. Actually, we will be interested only in integral vectors that belong to the dependence polyhedra, since dependence distance are indeed integral vectors.

Definition 2 (Polyhedral reduced dependence graphs)
A PRDG is a RDG, i.e. a graph with one node per statement of the loop nest, and with an edge from node S_i to node S_j if there exists at least one pair $(I, J) \in \mathcal{D} \times \mathcal{D}$ such that $S_i(I) \Longrightarrow S_j(J)$. Furthermore, each edge e in the PRDG is labeled by a dependence polyhedron $P(e)$ that approximates the set of distance vectors: the associated ADG contains an edge from instance I of node S_i to instance J of node S_j if and only if $(J - I) \in P(e)$.

In the rest of the paper, we explore this representation of dependences, where distance sets are approximated by polyhedra, specified by their vertices, rays and lines. To avoid a possible confusion between the vertices of a dependence graph and the vertices of a dependence polyhedron, we call the first one **nodes** instead of **vertices**.

2.3 Simulation of classical dependence representations

We now come back to more classical dependence abstractions: level of dependence and direction vector. We recall their definition and show that RDGs labeled by direction vectors or levels of dependence are actually particular cases of polyhedral reduced dependence graphs.

2.3.1 Direction vectors

When the set of distance vectors is a singleton, the dependence is said uniform and the only distance vector is called a **uniform dependence vector**.

Otherwise, the set of distance vectors can still be represented by a n-dimensional vector (called the **direction vector**), whose components belong to $\mathbb{Z} \cup \{*\} \cup (\mathbb{Z} \times \{+, -\})$. Its i-th component is an approximation of the i-th components of all possible distance vectors: it is equal to $z+$ (resp. $z-$) if all i-th components are greater (resp. smaller) than or equal to z. It is equal to $*$ if the i-th component may take any value and to z if the dependence is uniform in this dimension with unique value z. In general, $+$ (resp. $-$) is used as shorthand for $1+$ (resp. $(-1)-$).

We denote by e_i the i-th canonical vector, i.e. the n-dimensional vector whose components are all null except the i-th component equal to 1. Then, a direction vector is nothing but an approximation by a polyhedron, with a single vertex and whose rays and lines, if any, are canonical vectors.

Indeed, consider an edge e labeled by a direction vector d and denote by I^+, I^- and I^* the sets of components of d which are respectively equal to $z+$ (for some integer z), $z-$, and $*$. Finally, denote by d_z the n-dimensional vector whose i-th component is equal to z if the i-th component of d is equal to z, $z+$ or $z-$, and to 0 otherwise.

Then, by definition of the symbols $+$, $-$ and $*$, the direction vector d represents exactly all n-dimensional vectors p for which there exist integers (ν, ν', ξ) in $\mathbb{N}^{|I^+|} \times \mathbb{N}^{|I^-|} \times \mathbb{Z}^{|I^*|}$ such that:

$$p = d_z + \sum_{i \in I^+} \nu_i e_i - \sum_{i \in I^-} \nu'_i e_i + \sum_{i \in I^*} \xi_i e_i \tag{3}$$

In other words, the direction vector d represents all integer points that belong to the polyhedron defined by the single vertex d_z, the rays e_i for $i \in I^+$, the rays $-e_i$ for $i \in I^-$ and the lines e_i for $i \in I^*$.

For example, the direction vector $(2+, *, -, 3)$ defines the polyhedron with one vertex $(2, 0, -1, 3)$, two rays $(1, 0, 0, 0)$ and $(0, 0, -1, 0)$, and one line $(0, 1, 0, 0)$.

2.3.2 Level of dependences

The representation by level is the less accurate (though powerful [DV96]) dependence abstraction. In a loop nest with n nested loops, the set of distance vectors is approximated by an integer l, in $[1 \ldots n] \cup \{\infty\}$, defined as the largest integer such that the $l-1$ first components of the distance vectors are zero.

A dependence at level $l \leq n$ means that the dependence occurs at depth l of the loop nest, i.e. at a given iteration of the $l-1$ outermost loops. In this case, one says that the dependence is a **loop carried dependence** at level l. If $l = \infty$, the dependence occurs inside the loop body, between two different statements, and is called a **loop independent dependence**.

Consider an edge e of level l. By definition of the level, the first non zero component of the distance vectors is the l-th component and it can possibly take any positive integer value. Furthermore, we have no other knowledge on the remaining components. Therefore, an edge of level $l < \infty$ is equivalent to the direction vector: $(\overbrace{0, \ldots, 0}^{l-1}, 1+, \overbrace{*, \ldots, *}^{n-l})$ and an edge of level ∞ corresponds to the null dependence vector. As any direction vector admits an equivalent polyhedron, so does a representation by level.

For example, level 2, in a 3-dimensional loop nest, means direction vector $(0, 1+, *)$ which corresponds to the polyhedron with one vertex $(0, 1, 0)$, one ray $(0, 1, 0)$ and one line $(0, 0, 1)$.

3 Overview of the parallelization algorithm

To help the reader follow, we first state the different steps of our parallelization algorithm, without entering neither into implementation details, nor into technical proofs. The detailed proofs of the correctness and optimality of our algorithm are delayed until Sections 4 and 5. An optimized version of the algorithm is presented in Section 6.

The parallelization algorithm consists of two main steps, a "uniformization" step presented in Section 3.2 and a "scheduling" step summarized in Section 3.3. We illustrate both steps with Example 2 below.

3.1 Illustrating example

We will work out the following example, assuming that in the reduced dependence graph, edges are labeled by direction vectors.

Example 2

```
DO i = 1, n
  DO j = 1, n
    DO k=1, j
      a(i, j, k) = c(i, j, k - 1) + 1
      b(i, j, k) = a(i - 1, j + i, k) + b(i, j - 1, k)
      c(i, j, k + 1) = c(i, j, k) + b(i, j - 1, k + i) + a(i, j - k, k + 1)
    ENDDO
  ENDDO
ENDDO
```

The graph depicted in Figure 3 has been found by the dependence analyzer Tiny [Wol90].

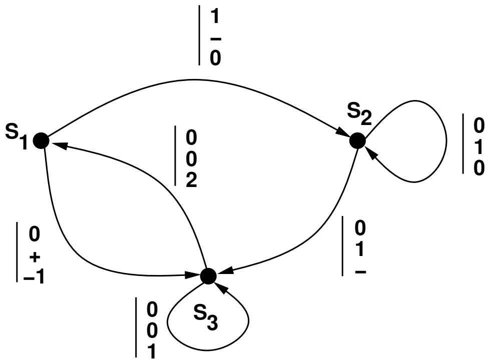

Figure 3: Reduced dependence graph with direction vectors, for Example 2.

The reader can check that neither Allen and Kennedy's algorithm, nor Wolf and Lam's algorithm, is able to find the full parallelism for this code: the third statement seems to be purely sequential.

However, the parallelism detection algorithm that we propose in the next sections is able to build the following multi-dimensional schedule: $(2i+1, 2k)$ for the first statement, $(2i, j)$ for the second statement and $(2i+1, 2k+3)$ for the third statement. This schedule corresponds to the code with explicit parallelism given below (but in which no effort, such as loop peeling, has been made so as to remove "if" tests). For each statement, one level of parallelism can be detected.

This code has been generated by the the procedure "codegen" of the Omega Calculator delivered with Petit [KMP+95], thanks once again to Bill Pugh's team. We point out that it is a "virtual" code in the sense that it only reveals hidden parallelism. We do not claim that it must be implemented as such.

```
DOSEQ i = 1, n
  DOSEQ j = 1, n
    DOPAR k = 1, j
      b(i, j, k) = a(i - 1, j + i, k) + b(i, j - 1, k)
    ENDDO
  ENDDO
```

```
    DOSEQ k = 1, n + 1
      IF (k ≤ n) THEN
        DOPAR j = k, n
          a(i, j, k) = c(i, j, k - 1) + 1
        ENDDO
      ENDIF
      IF (k ≥ 2) THEN
        DOPAR j = k - 1, n
          c(i, j, k) = c(i, j, k - 1) + b(i, j - 1, k + i - 1) + a(i, j - k + 1, k)
        ENDDO
      ENDIF
    ENDDO
  ENDDO
```

3.2 Uniformization step

We first show how PRDGs (polyhedral reduced dependence graphs) can be captured into an equivalent (but simpler to manipulate) structure, the structure of uniform dependence graphs, i.e. graphs whose edges are labeled by constant dependence vectors. This uniformization scheme is achieved by the **translation algorithm**, given below.

To avoid possible confusions, the initial PRDG that describes the dependences in the code to be parallelized is called the **original graph** and denoted by $G_o = (V, E)$. The uniform RDG, equivalent to G_o, built by the translation algorithm, is called the **uniform graph** or the **translated** of G_o and is denoted by $G_u = (W, F)$.

The translation algorithm builds G_u by scanning all edges of G_o. It starts from $G_u = (W, F) = (V, \emptyset)$, and for each edge e of E, it adds to G_u new nodes and new edges depending on $P(e)$. We call **virtual nodes** the nodes that are created as opposed to **actual** nodes which correspond to nodes of G_o.

Let e be an edge of E. We denote by x_e and y_e, respectively the **tail** and **head** of e, i.e. the nodes that e respectively leaves and enters: $x_e \xrightarrow{e} y_e$. This definition is generalized to paths: the head (resp. tail) of a path is the head (resp. tail) of its last (resp. first) edge.

We follow the notations introduced in Section 2.2: we denote respectively by ω, ρ, and λ the number of vertices v_i, rays r_i, and lines l_i of the polyhedron $P(e)$.

Translation Algorithm

- Let $W = V$ and $F = \emptyset$
- For $e : x_e \to y_e \in E$ do
  - i. If $\rho = 0$ and $\lambda = 0$ (the polyhedron is reduced to a polytope)
    - Add to F ω edges of weights $v_1, v_2, \ldots, v_\omega$ directed from x_e to y_e.
  - ii. If $\rho \neq 0$ or $\lambda \neq 0$
    - Add to W a new virtual node n_e,
    - Add to F ω edges of weights $v_1, v_2, \ldots, v_\omega$ directed from x_e to n_e,
    - Add to F ρ self-loops around n_e of weights $r_1, r_2, \ldots, r_\rho$,
    - Add to F λ self-loops around n_e of weights $l_1, l_2, \ldots, l_\lambda$,
    - Add to F λ self-loops around n_e of weights $-l_1, -l_2, \ldots, -l_\lambda$,

– Add to F a null weight edge directed from n_e to y_e.

Figure 4 shows an example of translation in the case where the dependence polyhedron has no lines.

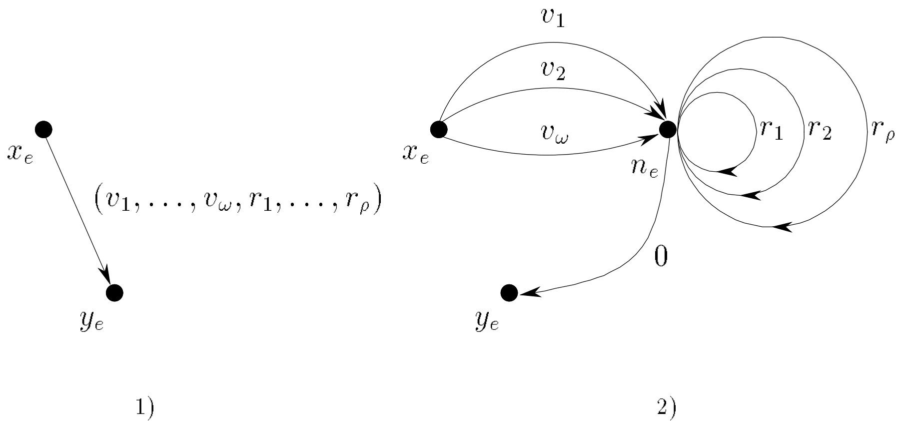

Figure 4: Example of graph translation: 1) Original graph with a dependence polyhedron, 2) The equivalent uniform dependence graph.

Back to Example 2 The uniform dependence graph associated to the PRDG of Example 2 (see Figure 3) is depicted in Figure 5. It has three new nodes (i.e. virtual nodes) that correspond to the symbol "+" and the two symbols "−" in the initial direction vectors.

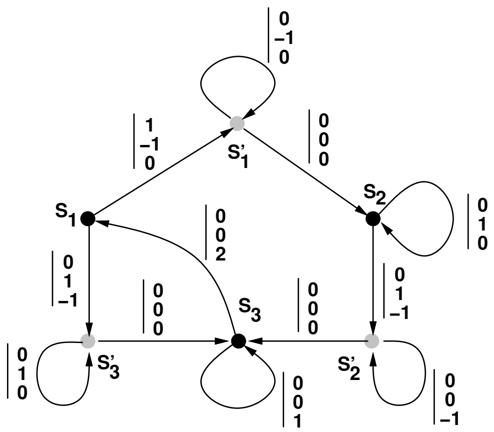

Figure 5: Translated uniform reduced dependence graph

3.3 Scheduling step

The scheduling step takes as input the translated dependence graph G_u and builds a multi-dimensional schedule for each actual node, i.e. for each node of G_u that corresponds to a node of G_o. G_u is assumed to be strongly connected (otherwise, the algorithm has to be called for each strongly connected component of G_u).

It is a recursive algorithm. Each step of the recursion builds a subgraph G' of the current graph G being processed: G' is the subgraph of G generated by all edges of G that belong to at least one multi-cycle (i.e. union of cycles) of null weight. Once G' is built [4], a set of linear constraints is derived and a valid schedule that satisfies all dependence edges not in G' can be computed. Then, the algorithm keeps working on the remaining edges, i.e. the edges of G' (more precisely G' and some additional edges, see below).

The scheduling step can be summarized by the recursive algorithm given below. The initial call is DARTE-VIVIEN(G_u, 1). The algorithm builds, for each actual node S of G_u, a sequence of vectors $X_S^1, \ldots, X_S^k, \ldots, X_S^{d_S}$ and a sequence of constants $\rho_S^1, \ldots, \rho_S^k, \ldots, \rho_S^{d_S}$ that define a valid multi-dimensional schedule.

DARTE-VIVIEN(G, k)

i. Build G' the subgraph of G generated by all edges that belong to at least one null weight multi-cycle of G.

ii. Add in G', all edges from x_e to y_e and all self-loops on y_e if $e = (x_e, y_e)$ is an edge already in G', from an actual node x_e to a virtual node y_e.

iii. Select a vector X and, for each node S in G, a constant ρ_S such that:

$$\begin{cases} e = (x_e, y_e) \in G' \text{ or } x_e \text{ is a virtual node} & \Rightarrow \quad Xw(e) + \rho_{y_e} - \rho_{x_e} \geq 0 \\ e = (x_e, y_e) \notin G' \text{ and } x_e \text{ is an actual node} & \Rightarrow \quad Xw(e) + \rho_{y_e} - \rho_{x_e} \geq 1 \end{cases}$$

For all actual nodes S of G, let $\rho_S^k = \rho_S$ and $X_S^k = X$.

iv. If G' is empty or has only virtual nodes, return.

v. If G' is strongly connected and has at least one actual node, G is not computable (and the initial PRDG G_o is not consistent), return.

vi. Otherwise, decompose G' into its strongly connected components G_i and for each G_i that has at least one actual node, call DARTE-VIVIEN(G_i, $k+1$).

Remarks

- We will see that Step (ii) is necessary only for general PRDGs: for example, it could be removed for RDGs labeled by direction vectors (see Section 5). In this case, the two Steps (i) and Step (iii) can be solved by a single linear program resolution.

- In Step (iii), we do not specify, on purpose, how the vector X and the constants ρ are selected, so as to allow various selection criteria. We refer to Section 6 for more details. For example, a maximal set of linearly independent vectors X can be selected if the goal is to derive fully permutable loops (see [DV95b]).

Back to Example 2 Consider the uniform dependence graph of Figure 5. There are two elementary cycles of weights $(1,0,1)$ and $(0,1,1)$, and five self-loops of weights $(0,0,1)$, $(0,0,-1)$, $(0,1,0)$ (twice) and $(0,-1,0)$. Therefore, all edges (except the edges that only belong to the cycle of weight $(1,0,1)$) belong to a multi-cycle of null weight. The subgraph G' is depicted in Figure 6.

[4] by a solving a linear program, see Section 5.1

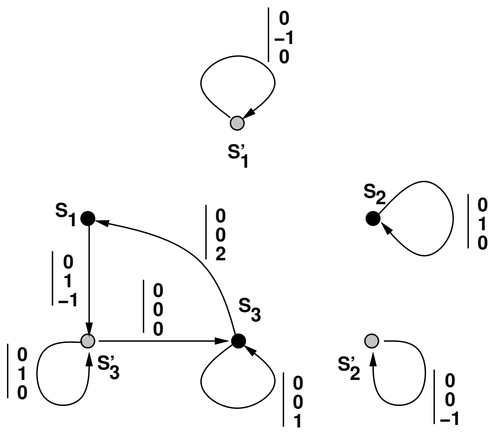

Figure 6: Subgraph of null weight multi-cycles for Example 2.

The constraints coming from edges in G' make that $X = (x, y, z)$ must be orthogonal to the weight of all cycles of G'. Therefore, $y = z = 0$. Finally, considering the other constraints, we find the solution $X = (2, 0, 0)$, $\rho_{S_1} = \rho_{S_3} = 1$ and $\rho_{S_2} = 0$.

In G', there remains four strongly connected components, and two of them are not considered since they only have virtual nodes. The two other components have no null weight multi-cycle. The strongly connected component with the single node S_2 can be scheduled with the vector $X = (0, 1, 0)$, whereas studying the other strongly connected component leads, among other solutions, to $X = (0, 0, 2)$, $\rho_{S_1} = 0$, and $\rho_{S_3} = 3$.

Finally, summarizing the results, we find, as claimed in Section 3.1, the 2-dimensional schedules: $(2i, j)$ for S_2, $(2i + 1, 2k)$ for S_1 and $(2i + 1, 2k + 3)$ for S_3.

We are now ready to explain in full details the different steps of this parallelization algorithm. It is based on scheduling techniques that we previously developed for systems of uniform recurrence equations. Therefore, we will frequently refer to [DV95a].

4 Computability of a PRDG

A dependence graph that defines exactly the dependences of a loop nest is always computable since, for example, the sequential ordering is a valid schedule. However, if dependences are given by an approximation, the dependence graph may describe a structure of computations that is not computable. For example, the PRDG of Figure 7(a) is computable (even if the direction vector $(0, *)$ could be refined in the case of nested loops) whereas the PRDG of Figure 7(b) is not computable, i.e. can not be scheduled. This comes from the fact that there is a dependence path of null weight in the second PRDG, but not in the first one.

Therefore, before even thinking of parallelization, we need to check if a PRDG is computable or not. This section shows how it can be done.

In Section 4.1, we explore the links between G_o and its translated G_u. This study permits to manipulate G_u, that has a more classical structure, instead of G_o. Indeed, G_u is very similar to a reduced dependence graph associated to a system of uniform recurrence equations, i.e. a graph whose edges are labeled by integral vectors. We show how well-known results for systems of uniform recurrence equations can be adapted to G_u: Section 4.2 is devoted to the computability problem, while Section 4.3 focuses on the length of longest paths described by G_u.

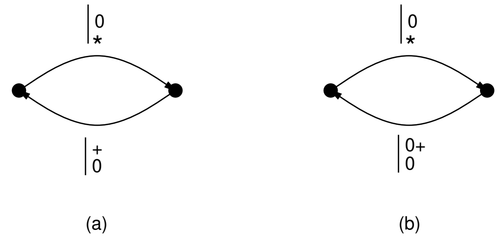

Figure 7: Two different PRDGs: (a) computable, (b) not computable.

From now on, we assume that the PRDG G_o is strongly connected (and so is G_u). Otherwise, the results presented hereafter are true for each of the strongly connected components of G_o.

4.1 Correspondence between a PRDG G_o and its translated G_u

Recall that edges in G_o are labeled not by a particular weight (as in G_u), but by a polyhedron, i.e. a set of vectors. To avoid a possible confusion between edges labeled by a polyhedron and edges labeled by an integral vector, we call polyhedral edge e an edge of G_o labeled by a polyhedron $P(e)$, and dependence edge e an edge of G_o labeled by an integral vector $w(e)$ such that $w(e) \in P(e)$.

In other words, a polyhedral edge e corresponds to as many dependence edges as there are integral vectors in $P(e)$: a dependence edge is an instantiation of a polyhedral edge. We define a dependence path as a sequence of adjacent dependence edges. The weight of a dependence path is the sum of the weights of its edges.

We now show the links between paths in G_u and dependence paths in G_o.

4.1.1 From G_u to G_o

Note first the particular structure of G_u. There is no edge between distinct virtual nodes. Thus, any path between two virtual nodes has to pass through an actual node and any path between two actual nodes whose intermediate nodes are all virtual nodes, visits only one virtual node, possibly many times. We call such a path, a **basic path**. The link between basic paths in G_u and dependence edges in G_o is straightforward:

Lemma 1 *A basic path Π_u of G_u corresponds to a unique polyhedral edge G_o and the total weight of Π_u is a vector that belongs to the polyhedron $P(e)$ associated to e.*

Proof: If Π_u has no virtual nodes, then Π_u is a path with a single edge f from an actual node to an actual node: $\Pi_u = x_e \xrightarrow{f} y_e$. f has been defined in G_u by case (i) of the translation algorithm, while considering a particular polyhedral edge e of G_o. Thus, $P(e)$ is a polytope and the weight of f is the value of one of the vertices of this polytope.

If Π_u has some virtual nodes, they all correspond to a unique virtual node that may be visited several times. This unique virtual node n_e has been created while considering a unique polyhedral edge e of G_o.

$$\Pi_u = x_e \xrightarrow{v_i} n_e \xrightarrow{r_i, l_i, -l_i} n_e \ldots n_e \xrightarrow{r_j, l_j, -l_j} n_e \xrightarrow{0} y_e$$

The first edge of Π_u goes from x_e to n_e and corresponds to a vertex of the polyhedron $P(e)$, the last edge of Π_u goes from n_e to y_e and has a null weight, and all other edges of Π_u are self-loops around n_e of weight, either equal to a ray of $P(e)$ or to a line of $P(e)$ (or its opposite). Thus, according to Equation 2, the weight of the path Π_u is included in $P(e)$. ■

From lemma 1, we immediately deduce the following corollary:

Corollary 1 *A path Π_u of G_u, from an actual node to an actual node, defines an equivalent dependence path Π_o in G_o: each basic sub-path of Π_u corresponds exactly to a polyhedral edge e of G_o whose dependence polyhedron contains the weight of the basic sub-path.*

4.1.2 From G_o to G_u

Lemma 1 (and a fortiori Corollary 1) is in general not a strict equivalence: it is not always possible to build an equivalent basic path in G_u for any dependence edge of G_o, since an integral vector in $P(e)$ may be a rational (but not integral) linear combination of vertices, rays or lines of $P(e)$.

However, when the dependence path is a cycle, there is still a correspondence as stated in Lemma 2. In the following, if $\mathcal{C}$ is a cycle, $m\mathcal{C}$ denotes the cycle formed by m times the cycle $\mathcal{C}$.

Lemma 2 *Let $\mathcal{C}$ be a dependence cycle of G_o, i.e. a cycle of dependence edges. Then, for some integer m, the cycle $m\mathcal{C}$ is equivalent to a cycle $\mathcal{C}_u$ in G_u of same structure and same weight.*

Proof: Let e be a dependence edge of $\mathcal{C}$. By definition, $w(e)$ belongs to $P(e)$ and thus, according to Equation 2, satisfies:

$$w(e) = \sum_{i=1}^{\omega(e)} \mu(e,i)v(e,i) + \sum_{i=1}^{\rho(e)} \nu(e,i)r(e,i) + \sum_{i=1}^{\lambda(e)} \xi(e,i)l(e,i)$$

with $\mu(e,i) \in \mathbb{Q}^+$, $\nu(e,i) \in \mathbb{Q}^+$, $\xi(e,i) \in \mathbb{Q}$ and $\sum_{i=1}^{\omega(e)} \mu(e,i) = 1$ and where the $v(e,i)$, $r(e,i)$, and $l(e,i)$ denote the $\omega(e)$ vertices, $\rho(e)$ rays and $\lambda(e)$ lines of $P(e)$.

Let Δ be the least common multiple of the denominators of all $\mu(e,i)$, $\nu(e,i)$, and $\xi(e,i)$ that appear in the decomposition of all $w(e)$. By construction, for each edge e in $\mathcal{C}$, we now have:

$$\Delta w(e) = \sum_{i=1}^{\Delta} v(e,k_i) + \sum_{i=1}^{\rho(e)} \nu'(e,i)r(e,i) + \sum_{i=1}^{\lambda(e)} \xi'(e,i)l(e,i)$$

with $1 \leq k_i \leq \omega(e)$, $\nu'(e,i) \in \mathbb{N}$, and $\xi'(e,i) \in \mathbb{Z}$.

We are now able to build the desired cycle $\mathcal{C}_u$, equivalent to the cycle $\Delta\mathcal{C}$. Let $e_1, \ldots, e_n$ be the dependence edges of $\mathcal{C}$. For each i, $1 \leq i \leq \Delta$, we define the cycle $\mathcal{C}_u^i$ as:

$$\mathcal{C}_u^i = x_{e_1} \xrightarrow{v(e_1,k_i)} n_{e_1} \xrightarrow{0} y_{e_1} = x_{e_2} \xrightarrow{v(e_2,k_i)} n_{e_2} \ldots \xrightarrow{0} y_{e_2} \ldots y_{e_{n-2}} = x_{e_{n-1}} \xrightarrow{v(e_{n-1},k_i)} y_{e_n} = x_{e_1}$$

We concatenate all C_u^i and finally, we modify C_u^1 by connecting, for each edge e of $\mathcal{C}$, $\nu'(e,i)$ times the self-loop around n_e of weight $r(e,i)$ and $\xi'(e,i)$ times (resp. $-\xi'(e,i)$ times) the self-loop around n_e of weight $l(e,i)$ (resp. $-l(e,i)$) if $\xi'(e,i)$ is non negative (resp. negative). This defines a cycle equivalent to $\Delta\mathcal{C}$ both in terms of structure and weight. ■

4.1.3 Computability condition

In the case of bounded iteration domains, a reduced dependence graph is computable if and only if the apparent dependence graph that it describes is acyclic, i.e. if there is no dependence path from an instance of a statement to the same instance of the same statement. In other words, a PRDG G_o is computable if and only if it has no dependence cycle of null weight. According to Corollary 1 and Lemma 2, the computability of G_o is translated into G_u as follows:

Theorem 1 *G_o is computable if and only if G_u contains no null weight cycle with at least one actual node.*

4.2 Computability of the translated G_u

In the previous section, we moved the computability problem from G_o to G_u. Now, checking the existence of null weight cycles in G_u is simpler than in G_o since G_u is nothing but the reduced dependence graph of a system of uniform recurrence equations, except that some nodes are called actual and some other virtual. We can thus apply all techniques developed for system of uniform recurrence equations (see [KMW67] and [DV95a]), with a slight modification so as to take into account the fact that in G_u, some nodes are virtual nodes and exist only to simulate dependences in G_o.

Many algorithms that check the computability of systems of uniform recurrence equations have been proposed ([KMW67, Rao85, Roy88, KS88, DV95a]), the first one being Karp, Miller, and Winograd's decomposition [KMW67]. However, we suggest the reader to refer to the algorithm formulation and proofs proposed in [DV95a] because they have been developed with the scheduling problem (more than with the computability problem) in mind, which is more interesting from a parallelization point of view.

The basic algorithm looks for null weight cycles in a uniform reduced dependence graph G. It is a recursive algorithm, based on the search for null weight multi-cycles (i.e. union of cycles, not necessarily connected). However, in our case (cf Theorem 1), we are not interested in all null weight cycles, but only in null weight cycles which contain an actual node. We have thus to refine the decomposition algorithm. This leads to the following algorithm which enable us to determine the computability of G_o via G_u. The only modification we make compared to Karp, Miller, and Winograd's decomposition is to not consider subgraphs of G_u that have only virtual vertices. Therefore, the correctness of this new algorithm follows is a direct consequence of the correctness of Karp, Miller, and Winograd's decomposition.

4.2.1 Decomposition algorithm: DA

Boolean DA(G)

i. Build G' the subgraph of G generated by all the edges that belong to at least one null weight multi-cycle of G.

ii. Compute the strongly connected components of G' and let G'_1, G'_2, ..., G'_s be the s components that have at least one actual node.

- If G' is empty or has only virtual nodes, return TRUE.
- If G' is strongly connected and has at least one actual node, return FALSE.
- Otherwise, return $\bigwedge_{i=1}^{s} \mathrm{DA}(G'_i)$ (logical AND).

Then, G has no cycle of null weight containing an actual node if and only if DA(G) = TRUE. Therefore, a PRDG G_o is computable if and only if DA(G_u) = TRUE.

Definition 3 *For a graph G, we define the depth of G (denoted by $d(G)$ or simply d) as the maximal number of recursive calls generated by the initial call DA(G) (counting the first one), except if G is acyclic, in which case we let $d(G) = 0$.*

For systems of uniform recurrence equations, the depth of the decomposition algorithm is a measure of the degree of parallelism described by G. We will see that this result still holds for PRDGs and the depth d defined for algorithm DA.

4.2.2 Dependence polyhedra considered to be atomic: algorithm DA*

For technical reasons that will become clear in the scheduling section (Section 5), we need to slightly modify the decomposition algorithm DA.

Remember how G_u has been built (refer to the translation algorithm in Section 3.2 and to Figure 4). We want to keep together, during the decomposition algorithm, all edges that come from the translation of the same dependence polyhedron $P(e)$, as soon as one of the vertices of $P(e)$ is kept in G'. For that, we add in G', all edges from x_e to y_e and all self-loops on y_e if $e = (x_e, y_e)$ is an edge already in G', from an actual node x_e to a virtual node y_e. This leads to the following algorithm:

Boolean DA*(G)

i. Build G' the subgraph of G generated by all the edges that belong to at least one null weight multi-cycle of G.

ii. Add in G', all edges from x_e to y_e and all self-loops on y_e if $e = (x_e, y_e)$ is an edge already in G', from an actual node x_e to a virtual node y_e.

iii. Compute the strongly connected components of G' and let G'_1, G'_2, ..., G'_s be the s components that have at least one actual node.

- If G' is empty or has only virtual nodes, return TRUE.
- If G' is strongly connected and has at least one actual node, return FALSE.
- Otherwise, return $\bigwedge_{i=1}^{s}$ DA*(G'_i) (logical AND).

Note that the edges we add to G' do not change the structure of strongly connected components of G', since we add edges only between nodes that already belong to the same strongly connected component of G'. Furthermore, in each strongly connected component, the structure of null weight multi-cycles is also unchanged since we add only edges that do not belong to a null weight multi-cycle. Therefore, both algorithms DA* and DA have exactly the same behavior (in terms of nodes).

4.3 Longest dependence paths in G_o

Once we know whether a polyhedral reduced dependence graph is computable, it is interesting to give an estimate on the length of the longest path in its apparent dependence graph. This length gives a lower bound on the sequentiality described by the PRDG, and thus, gives an upper bound on the parallelism it contains.

In [DKR91], it is shown that for a single uniform recurrence equation (i.e. a uniform reduced dependence graph with a single node), this length is equivalent to the latency of the optimal linear schedule, for full dimensional polyhedral iteration domains whose size tends to infinity. Remember that the latency of a linear schedule is nothing but the number of sequential iterations. To say it briefly, on domains of size parameterized by N, the latency of the optimal linear schedule is equivalent to λN for some constant λ and so does the length of the longest dependence path. Both are in N and the multiplicative constants are the same. This result has been extended in [d'A96], for strongly connected uniform dependence graphs, if a "shifted linear" schedule exists.

In our case, which is similar to the case of general uniform dependence graphs, one-dimensional affine schedules may not exist and the length of the longest path is not necessarily $O(N)$ anymore, it can be equivalent to kN^p for some constants k and p. We will not try to be precise on the multiplicative constant k, we will just focus on p the power of N.

Actually, for a system of uniform recurrence equations, it has been shown that this power is equal to the depth of Karp, Miller, and Winograd's decomposition as recalled by the following theorem (Theorem 2). Indeed, if d is the depth of G for Karp, Miller, and Winograd's decomposition, there exists a path in the apparent dependence graph of length $\Omega(N^d)$. A complete proof of this result can be found in [DV94]. The iteration domain is supposed to contain (resp. to be contained in) a full dimensional cube of size $\Omega(N)$ (resp. $O(N)$). Actually, we have the following stronger result:

Theorem 2 *For each node S of G, let d_S be the depth where node S is removed, in Karp, Miller, and Winograd's decomposition. Then, for each strongly connected component G^i of G, there exists a path in the apparent dependence graph of G whose projection in G is a cycle that visits $\Omega(N^{d_S})$ times each node S of G^i.*

A similar theorem can be stated for PRDGs.

Theorem 3 *For each node S of G_o, let d_S be the depth where the node S is removed, in the decomposition algorithm DA (or equivalently DA*) applied to G_u. Then, for each strongly connected component G_o^i of G_o, there exists a path in the apparent dependence graph of G_o whose projection in G_o is a cycle that visits $\Omega(N^{d_S})$ times each node of G_o^i.*

Proof: Note first that for each actual node of G_u, the depth for Karp, Miller, and Winograd's decomposition (in which actual nodes and virtual nodes are not distinguished) and the depth for Algorithm DA are equal. We can thus apply Theorem 2 which gives, for each strongly connected component G_u^i of G_u, a path in the apparent dependence graph of G_u, whose projection in G_u visits $\Omega(N^{d_S})$ times each node S of G_u^i. Furthermore, the projection in G_u is a cycle with at least one actual node. We then use Corollary 1 to obtain an equivalent path in G_o. Lemma 1 ensures that the number of occurrences of an actual node in both paths are equal. ■

5 Scheduling a PRDG

In this section, we detail the scheduling algorithm for PRDGs introduced in Section 3.3. In Section 5.1, we recall how G' can be computed and the link with "shifted-linear" schedules. In Section 5.2, we show that the scheduling algorithm is correct, i.e. that it produces a schedule that respect all dependences. Finally, in Section 5.3, we show the optimality of the scheduling algorithm.

5.1 Construction of G' and related properties

In this section, we recall how to build G', the subgraph of null weight multi-cycles. We also recall why it is closely related to the construction of separating hyperplanes, thus to the construction of multi-dimensional schedules.

Step (i) of the decomposition algorithm (DA or DA*), devoted to the construction of G', can be implemented by one linear programming resolution. Indeed, it has been shown (see [DV95a]) that the edges of G' are exactly the edges e for which $v_e = 0$ in any optimal solution of the following linear program:

$$\min \left\{ \ \sum_e v_e \quad | \quad q \geq 0, \quad v \geq 0, \quad w \geq 0, \quad q + v = w + 1, \quad Cq = 0, \quad Wq = 0 \ \right\} \tag{4}$$

where C is the connection matrix of G (with as many rows as nodes in G, and as many columns as edges in G) and W the dependence matrix (i.e. whose columns are the weights of edges of G). This linear program (program 4) is linked to the computability problem, whereas its dual (program 5) is linked to the scheduling problem.

$$\max \left\{ \ \sum_e z_e \quad | \quad 0 \leq z \leq 1, \quad Xw(e) + \rho_{y_e} - \rho_{x_e} \geq z_e \ \right\} \tag{5}$$

The relation between both programs is the following. Consider the set Σ defined by:

$$\Sigma = \{(X, \rho) \mid Xw(e) + \rho_{y_e} - \rho_{x_e} \geq 0 \text{ for all } e \in G\}$$

The set Σ is fundamental for deriving fully permutable loops nests (see [DV95b]). Edges of G' are characterized by the following lemma:

Lemma 3

- $e \in G' \Leftrightarrow$ *for all* $(X, \rho) \in \Sigma$, $Xw(e) + \rho_{y_e} - \rho_{x_e} = 0$.
- $e \notin G' \Leftrightarrow$ *there exists* $(X, \rho) \in \Sigma$ *such that* $Xw(e) + \rho_{y_e} - \rho_{x_e} \geq 1$.

Lemma 3 shows that considering the dual provides separating hyperplanes, which are strictly separating hyperplanes for the edges not in G' and weakly separating hyperplanes for edges in G'. Furthermore, any optimal solution of the dual program provides a separating hyperplane that is a strictly separating hyperplane simultaneously for all edges of G'. In particular if G' is empty, Lemma 3 permits to build a "shifted linear" schedule.

5.2 Correctness of the scheduling algorithm

We are now ready to prove that the scheduling algorithm proposed in Section 3.3 is correct.

Note first that the skeleton of Algorithm DARTE-VIVIEN is nothing but Algorithm DA*, whose correctness has already been studied in Section 4.2.2. Therefore, it remains to prove two things:

- that Step (iii) is feasible, i.e. that it is possible to define a vector X and constants ρ_S satisfying to the desired inequalities.
- that, as claimed in Section 3.3, the sequence of vectors $X_S^1, \ldots, X_S^{d_S}$ and the sequence of constants $\rho_S^1, \ldots, \rho_S^{d_S}$ permits to define a valid multi-dimensional schedule.

The first point is clear: indeed, from Lemma 3, we know that there exists a vector X and some constants ρ_S (for example, any optimal solution of the dual program (program 5)) such that:

$$\begin{cases} X\,w(e) + \rho_{y_e} - \rho_{x_e} = 0 & \text{for all } e \in G' \\ X\,w(e) + \rho_{y_e} - \rho_{x_e} \geq 1 & \text{for all } e \notin G' \end{cases}$$

Therefore, the constraints of Step (iii) can be satisfied since they are weaker (some inequalities ≥ 1 are relaxed to inequalities ≥ 0).

For making the proof of the second point simpler, we complete each sequence of vectors X_S^i with null vectors and each sequence of constants ρ_S^i with null constants so as to define sequences of same length d (remember that d is the depth of the decomposition algorithm DA on G_u).

Lemma 4 *If $G_o = (V, E)$ is computable, the multi-dimensional scheduling function T:*

$$\begin{array}{ccl} V \times \mathcal{D} & \longrightarrow & \mathbb{Z}^d \\ (S,\ I) & \rightarrow & (\lfloor X_S^1 I + \rho_S^1 \rfloor,\ \ldots,\ \lfloor X_S^{d_S} I + \rho_S^{d_S} \rfloor,\ 0,\ \ldots,\ 0) \end{array}$$

defines a valid multi-dimensional schedule for G_o.

Proof: A multi-dimensional scheduling function defines, for each statement S (i.e each node of G_o) and for each iteration vector $I \in \mathcal{D}$, a multi-dimensional vector $T(S, I)$. Then, computations are scheduled by lexicographic order of the $T(S, I)$. Therefore, in order to prove that T is a valid schedule, we need to prove that all dependence distances remain lexico-positive, i.e. that for all edges $e = (x_e, y_e)$ of G_o, for any weight $w(e)$ in $P(e)$, and for all point I of the domain $\mathcal{D}$, we have $T(y_e, I) >_l T(x_e, I - w(e))$ where $<_l$ denotes the strict lexicographic order (see Section 2). Actually, we prove a bit stronger property: there exists an integer $k \in [1 \ldots d]$ such that:

$$\begin{cases} X_{y_e}^i I + \rho_{y_e}^i \geq X_{x_e}^i (I - w(e)) + \rho_{x_e}^i & \text{for all } i,\ 1 \leq i \leq k-1 \\ X_{y_e}^k I + \rho_{y_e}^k \geq X_{x_e}^k (I - w(e)) + \rho_{x_e}^k + 1 & \end{cases} \tag{6}$$

which implies the desired property $T(y_e, I) >_l T(x_e, I - w(e))$.

The possible values of $w(e)$ are given by equation 2:

$$w(e) = \sum_{j=1}^{j=\omega} \mu_j v_j + \sum_{j=1}^{j=\rho} \nu_j r_j + \sum_{j=1}^{j=\lambda} \xi_j l_j \text{ with } (\mu, \nu, \xi) \in \mathbb{Q}^{+^\omega} \times \mathbb{Q}^{+^\rho} \times \mathbb{Q}^\lambda \text{ and } \sum_{j=1}^{j=\omega} \mu_j = 1$$

Remember how we built G_u. We introduced a new node n_e, and in G_u, all edges from x_e to n_e correspond to vertices of $P(e)$, all self-loops on n_e correspond to rays or lines, and there is a null weight edge from n_e to y_e.

If G_o is computable, all recursive calls in the decomposition algorithm DA end because the current G' is empty or has only virtual nodes. Thus, each edge from x_e to n_e is removed from the graph (i.e. do not belong to the current G') at some level of the decomposition. We denote by k the level where all edges that correspond to a vertex of $P(e)$ (i.e. edges from x_e to n_e) have been finally removed. Because of Step (ii) of algorithm DA* (equivalent to Step (ii) of the scheduling algorithm), all edges that correspond to $P(e)$ are actually kept together in the graph, until level k. More precisely, they are kept together in the same strongly connected component until level k, as well as the null weight edge from n_e to y_e.

Therefore, the sequences of vectors $X^i_{y_e}$, $X^i_{x_e}$ and $X^i_{n_e}$ are equal until level k: we can thus skip the subscript and we simply write X^i. The system 6 is reduced to:

$$\left\{ \begin{array}{ll} X^i w(e) + \rho^i_{y_e} - \rho^i_{x_e} \geq 0 & \text{for all } i, \ 1 \leq i \leq k-1 \\ X^k w(e) + \rho^k_{y_e} - \rho^k_{x_e} \geq 1 & \end{array} \right. \tag{7}$$

Refer to Step (iii) of the scheduling algorithm. Until level k, we have:

$$\left\{ \begin{array}{l} X^i l_j = X^i l_j + \rho^i_{n_e} - \rho^i_{n_e} \geq 0 \text{ and similarly, } X^i(-l_j) \geq 0 \text{ i.e. } X^i l_j = 0 \\ X^i r_j = X^i r_j + \rho^i_{n_e} - \rho^i_{n_e} \geq 0 \\ X^i v_j + \rho^i_{n_e} - \rho^i_{x_e} \geq 0 \text{ and } X^k v_j + \rho^k_{n_e} - \rho^k_{x_e} \geq 1 \\ X^i \, 0 + \rho_{y_e} - \rho_{n_e} \geq 0 \end{array} \right.$$

Thus, with the definition of $w(e)$, we get the desired result: all constraints of System 7 are satisfied and are obtained by summing the above inequalities in a suitable way. ■

Remark The modification we made to keep together all edges corresponding to the polyhedron $P(e)$ becomes clear. It permits to ensure that all constraints of System 6 are satisfied, even if $w(e)$ is a rational combinations of vertices, rays and lines of $P(e)$. Actually, if for all integral vectors $w(e) \in P(e)$, the decomposition of Equation 2 is a decomposition with integral (instead of rationals) coefficients - in this case, one says that $P(e)$ is decomposed on a **Hilbert basis**, see [Sch86] - then, the algorithm can be refined: edges f in G_u corresponding to $P(e)$ and for which $Xw(f) + \rho_{y_f} - \rho_{x_f} \geq 1$ in Step (iii) do not have to be kept in G'. This permits to relax constraints for the next levels. This is the case for example for direction vectors. The decomposition proposed in Section 2.3.1 is indeed a decomposition on a Hilbert basis. ■

5.3 Optimality of the multi-dimensional schedule

Now that we have a multi-dimensional schedule T, we prove its optimality in terms of degree of parallelism. We want to show that for each statement S (i.e. for each node of G_o), the number of instances of S that have been sequentialized by T is of the same order as the number of instances of S that are inherently sequentialized by the dependences.

The latency of T is the number of sequential steps induced by the order defined by T. In order to be more precise, we define for a statement S, the S-latency of T as the number of instances of S sequentialized by T. If the iteration domain $\mathcal{D}$ is contained in a n-dimensional cube of size $O(N)$, then the S-latency of T is $O(N^{d_S})$ since instances of S are scheduled by a d_S-multi-dimensional schedule. Similarly the latency of T is $O(N^d)$.

We define the S-length of a dependence path $\mathcal{P}$ in the apparent dependence graph associated to G_o as the number of vertices in $\mathcal{P}$ that correspond to instances of S. The S-length of a dependence path $\mathcal{P}$ is the number of instances of S in $\mathcal{P}$ that are inherently sequentialized by the dependences. Theorem 3 shows that there is a dependence path whose S-length is $\Omega(N^{d_S})$ if the iteration domain $\mathcal{D}$ contains a n-dimensional cube of size $\Omega(N)$.

This proves the following result:

Theorem 4 *The scheduling algorithm is nearly optimal: if d is the depth of the decomposition algorithm, the latency of the schedule is $O(N^d)$ and the length of the longest dependence path is $\Omega(N^d)$. More precisely, after code generation, each statement S is surrounded by exactly d_S sequential loops and these loops are inherently sequential.*

We illustrate our technique with a third example, in which the maximal parallelism can be detected only if dependences are approximated by a more accurate PRDG than a RDG labeled by direction vectors. The resulting S_1-latency is $O(N)$, the S_2-latency is $O(N^2)$.

Example 3

```
DO i = 1, n
  DO j = 1, n
    DO k = 1, n
      a(i, j, k) = a(i, j - 1, k + j) + b(j, i - 1, k)
      b(i, j, k) = b(i, j, k -1) + a(i, j, k)
    ENDDO
  ENDDO
ENDDO
```

The graph G_o depicted in Figure 8 has been found by the dependence analyzer Tiny [Wol90]. The

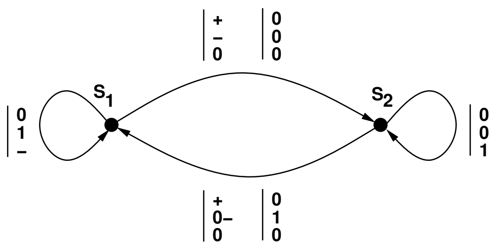

Figure 8: Reduced dependence graph with direction vectors, for Example 3.

uniformization step transforms G_o into G_u which is depicted in Figure 9.

There is a multi-cycle of null weight generated by all edges whose weight is orthogonal to $(1, 0, 0)$ (see Figure 9). In G', the strongly connected component that contains S_1 and S_2 still has a multi-cycle of null weight that visits an actual node (S_2). S_1 is removed at depth 2 but S_2 is removed at depth 3. S_2 is purely sequential, whereas one degree of parallelism is detected for S_1. The multi-dimensional schedules are $(i, 2j)$ for S_1 and $(i, 2j + 1, k)$ for S_2.

The resulting code is therefore the following:

```
DOSEQ i = 1, n
  DOSEQ j = 1, n
    DOPAR k = 1, n
      a(i, j, k) = a(i, j - 1, k + j) + b(j, i - 1, k)
    ENDDO
    DOSEQ k = 1, n
      b(i, j, k) = b(i, j, k -1) + a(i, j, k)
    ENDDO
  ENDDO
ENDDO
```

Note that this is exactly what Allen and Kennedy's algorithm would find.

However, if direction vectors are refined by more accurate dependence tests, one can find that the dependences can be approximated by the PRDG of Figure 10.

The reference to array b generates indeed two dependences, a flow dependence whose dependence polyhedron has one vertex $(0, 1, 0)$ and one ray $(1, -1, 0)$, and an anti dependence whose dependence polyhedron has one vertex $(1, -2, 0)$ and the same ray $(1, -1, 0)$. Note in Figure 10 how this

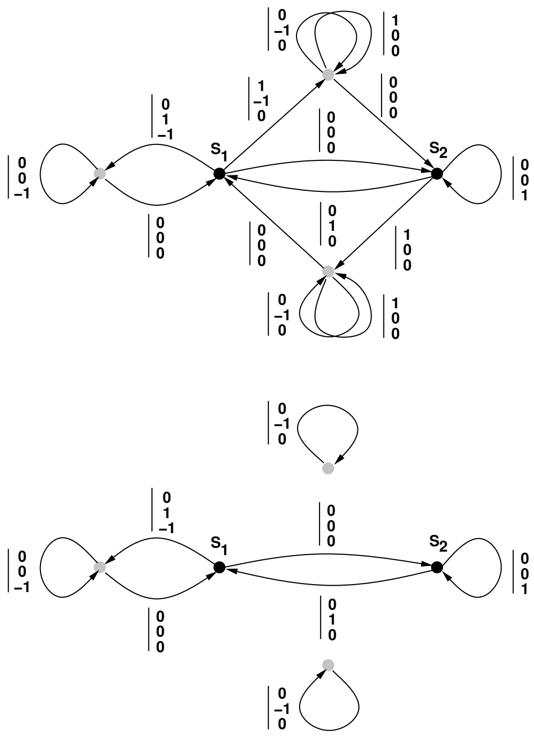

Figure 9: Translated uniform dependence graph for Example 3 and its corresponding G'.

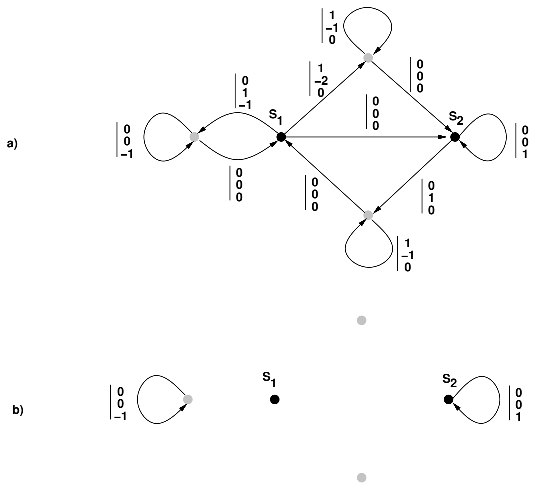

Figure 10: a) Polyhedral reduced dependence graph G_u, for Example 3
b) Graph of null-weight multicycles G'_u

modification changes the structure of G'. S_1 is now removed at depth 1 and S_2 at depth 2. For both statements, one more level of parallelism has been detected. The multi-dimensional schedules are $(4i + 2j)$ for S_1 and $(4i + 2j + 1, k)$ for S_2.

The resulting code is therefore the following:

```
DOSEQ j = 3, 3n
  DOPAR k = 1, n
    DOPAR i = max(1, ⌈(j-n)/2⌉), min(n, ⌊(j-1)/2⌋)
      a(i, j-2i, k) = a(i, j - 2i - 1, k + j -2i) + b(j - 2i, i - 1, k)
    ENDDO
  ENDDO
  DOSEQ k = 1, n
    DOPAR i = max(1, ⌈(j-n)/2⌉), min(n, ⌊(j-1)/2⌋)
      b(i, j - 2i, k) = b(i, j - 2i, k -1) + a(i, j - 2i, k)
    ENDDO
  ENDDO
ENDDO
```

6 Implementation strategies

The most time consuming steps of the scheduling algorithm presented in Section 3.3 are the resolutions of the different linear programs of Steps (i) and (iii). In order to speed up the algorithm, we propose to replace the linear programs previously described by some equivalent linear programs with fewer variables and fewer equations.

Consider the constraints of Step (iii). It turns out that they are equivalent to the constraints $Xw(C) \geq l(C)$ for all elementary cycles C where $l(C)$ denotes the number of edges $e = (x_e, y_e)$ of C that do not belong to G' and for which x_e is an actual node. Furthermore, once the constraints on cycles are satisfied, the different constants ρ can be computed by a technique similar to the Bellman-Ford algorithm, less expensive than a linear programming resolution.

This remark would permit to reduce the number of variables in the linear programs. Unfortunately, it would increase the number of constraints, since the number of elementary cycles in the graph may be exponential in the number of edges. Therefore, computing the weight of all cycles in the PRDG is not reasonable. To avoid this problem, we propose to use a **basis of cycles**, instead of considering all cycles. This technique permits to reduce both the number of constraints and the number of variables in the linear program of Step (iii). It is explained in the next sections.

We first recall the linear programs as they were expressed in Section 5. Then, we define the bases of cycles and we show how to build some equivalent linear programs easier to solve.

6.1 Original linear programs

At each level of the decomposition, we need to solve two different linear programs: the first one, linear program 8, to build G' from G; the second one, linear program 9, to build the vector X and the constants ρ_v.

$$\min \left\{ \textstyle\sum_e v_e \;\;|\;\; q \geq 0, \;\; v \geq 0, \;\; q + v \geq 1, \;\; Cq = 0, \;\; Wq = 0 \right\} \tag{8}$$

$$\left\{ \begin{array}{lll} e \in G' \text{ or } x_e \text{ is a virtual statement} & \Rightarrow & Xw(e) + \rho_{y_e} - \rho_{x_e} \geq 0 \\ e \notin G' \text{ and } x_e \text{ is an actual statement} & \Rightarrow & Xw(e) + \rho_{y_e} - \rho_{x_e} \geq 1 \end{array} \right. \tag{9}$$

Linear program 8 has $2|E|$ variables and $3|E|+V+n$ equalities or inequalities. Linear program 9 has $|V|+n$ variables and E inequalities.

6.2 Basis of cycles

We refer to [GM84] for a more precise definition of basis of cycles and cyclomatic number.

Definition 4 *A basis of cycles $\mathcal{BC}$ of G is a family of independent cycles $C_1, \ldots, C_m$ such that for each cycle C in G, there exists some rationals $\lambda_1, \ldots, \lambda_m$ such that:*

$$C = \sum_{i=1}^{m} \lambda_i C_i$$

Here, a cycle is a cycle in the graph considered as non-oriented, which means that edges can be used backwards. In this case, the corresponding vector that describes the cycle has a component equal to -1 for this edge. A cycle that uses all edges forwards is called a true cycle.

The classical way to build a basis of cycles is to build a maximal spanning tree, as the two objects are dual. This construction also proves that a basis of cycle contains exactly $m = p + |E| - |V|$ elements (the cyclomatic number), where p is the number of connected components in the graph. Remember that we are working separately on each strongly connected component of the PRDG, thus in our case, $p = 1$.

6.3 Computation of G'

In the linear program 8, the equality $Cq = 0$ ensures that the variable q represents a cycle (and the condition $q \geq 0$ that it is a true cycle). Thus this is equivalent to the fact that q can be decomposed on the basis $\mathcal{BC}$: $q = \sum_{i=1}^{m} \lambda_i C_i$, for some rationals $\lambda_1, \ldots, \lambda_m$. We thus obtain the following linear program 10:

$$\min \left\{ \textstyle\sum_e v_e \;\;\middle|\;\; Q\lambda \geq 0 \quad v \geq 0, \;\; Q\lambda + v \geq 1, \;\; \mathcal{W}\lambda = 0 \right\} \tag{10}$$

where Q denotes the $|E| \times m$-matrix whose columns are the cycles C_i of the basis, and $\mathcal{W}$ denotes the $n \times m$-matrix whose columns are the weights of the cycles C_i, i.e. $\mathcal{W} = WQ$. In other words, we replaced the constraints $q \geq 0$ and $Cq = 0$ by the constraints $Q\lambda \geq 0$. The edges of G' are still exactly the edges e for which $v_e = 0$ in any optimal solution of this linear program. It has $2|E| - |V| + 1$ variables (instead of $2|E|$) and $3|E| + n$ equalities or inequalities (instead of $3|E| + |V| + n$).

Computation of the vector X

Linear program 11 is dual to the linear program 10. It permits to compute the vector X, but not the constants ρ as in linear program 9. We will see below that they can be built with the Bellman-Ford algorithm, for any solution X of linear program 11.

$$\max \left\{ \textstyle\sum_{i=1}^{m} z_e \;\;\middle|\;\; x \geq 0, \;\; 0 \leq z \leq 1, \;\; X\mathcal{W} = (x+z)Q \right\} \tag{11}$$

Actually, in Step (iii) of the scheduling algorithm, we do not use the linear program dual to linear program 10, but we used a weaker version of this dual. We thus relax the same way the constraints in linear program 11. We prove at the end of this section that it still leads to valid schedules.

For each edge $e = (x_e, y_e)$ of G, we define an integer δ_e equal to 0 if e is in G' or if x_e is a virtual statement, and equal to 1 otherwise, i.e. if e is not in G' and if x_e is an actual statement. Then, the weaker version of linear program 11 that we use is:

$$\left\{ \; x \geq 0, \quad X\mathcal{W} = (x+\delta)Q \; \right\} \tag{12}$$

This linear program has n variables (instead of $n + |V|$) and $|E| - |V| + 1$ inequalities (instead of $|E|$). However, it only computes the vector X. We now show how computing the constants ρ in polynomial times once the vector X is known.

Remark Actually, what we call here linear program 12 is only a set of constraints, with no objective function. This allows us to define many objective functions so as to optimize the choice of X. Currently, we minimize its norm $\sum_{i=1}^{n} |X_i|$. This is done by the following linear program:

$$\min \left\{ \; \sum_{j=1}^{n} Y_j \quad | \quad x \geq 0, \quad X\mathcal{W} = (x+\delta)Q, \quad X \leq Y, \quad -X \leq Y \; \right\} \tag{13}$$

Let us suppose in what follows that we have a solution (X, x) of the linear program 12.

6.4 Computation of the constants ρ

To compute the constants ρ, we use the Bellman-Ford algorithm [CLR90] which solves the single-source shortest-path problem. The Bellman-Ford algorithm determines whether or not there is a negative-weight cycle that is reachable from the source. If there is no such cycle, the algorithm produces the shortest paths and their weights. It runs in time $O(|V||E|)$.

We build a copy H of the graph G. For each edge e of G, we replace its distance vector weight $w(e)$ by the induced delay $w'(e)$ defined by $w'(e) = Xw(e) - \delta_e$.

As we want to use the Bellman-Ford algorithm on H, we need to prove that H contains no negative-weight (true) cycles.

Lemma 5 *The weight of any (true) cycle $\mathcal{C}$ in H is non-negative.*

Proof: Consider a true cycle $\mathcal{C}$ in H: $\mathcal{C} \geq 0$. Decompose C on the basis of cycle $\mathcal{BC}$: $\mathcal{C} = Q\lambda$ for some vector λ. The weight of $\mathcal{C}$ in H is:

$$w'(\mathcal{C}) = w'(Q\lambda) = (X\mathcal{W} - \delta Q)\lambda = xQ\lambda = x\mathcal{C}$$

Since both vectors x and $\mathcal{C}$ are non-negative by definition, $x\mathcal{C}$ is non-negative. ■

Then, we use the Bellman-Ford algorithm on H to produce the shortest paths and their weights. We denote by ρ_S the opposite of the weight of the shortest path to the node S.

Lemma 6 *The vector X and the constants ρ_S form a solution of linear program 9.*

Proof: Let $e = (x_e, y_e)$ be an edge of G. The delay induced on e by (X, ρ_v) is equal to $Xw(e) + \rho_{y_e} - \rho_{x_e}$. By definition of a shortest path, we have:

$$(-\rho_{y_e}) \leq (-\rho_{x_e}) + w'(e) \Leftrightarrow Xw(e) + \rho_{y_e} - \rho_{x_e} \geq \delta_e$$

which is the desired inequality. ■

This two steps strategy, computing X first by a linear programming approach, and then the constants ρ by a graph-based technique, is the way we implemented the scheduling algorithm. It is not clear from a practical point of view if it really speeds up the linear program resolutions. However, its main interest is that it permits to guarantee that the constants ρ are simple if the vector X is simple. This property is highly desirable for code generation.

6.5 Extension to non perfectly nested loops

As proved in the previous sections, our scheduling algorithm is perfectly adapted to a description of distance vectors. When the loops are non perfectly nested, the distance vector $J - I$ between two statements S_1 and S_2 is defined only for the first dimensions that correspond to common loops, i.e. loops that surround both S_1 and S_2.

Therefore, a natural way of extending the algorithm to non perfect loop nests is to ignore, in each strongly connected component that appears during the decomposition, all dimensions that are not common dimensions. In other words, at a given depth of the algorithm, we truncate all vectors to the same dimension and we apply on the truncated vectors the same technique as for perfectly nested loops. Finally, we complete each vector X derived with null components so that they fit the right dimension.

It turns out that this strategy remains nearly optimal, as long as no information is given on the non common dimensions. However, if at each level, the code is non perfect then this algorithm is not more powerful than Allen and Kennedy's algorithm, since there is only one common dimension at each step.

To avoid this problem, we suggest another approach that permits to exploit the information on non common dimensions, and to benefit from the power of our algorithm for perfectly nested loops. We first transform the code into perfectly nested loops, by loops fusions or more complex techniques, possibly introducing if tests. Then, the scheduling algorithm is applied on the transformed nest, reasoning on its dependence graph. Here is an example, borrowed from the examples proposed in Petit [KMP+95].

Example 4

```
DO i = 2, n
  s(i) = 0                         (Statement S1)
  DO j = 1, i-1
    s(i) = s(i) + a(j, i)*b(j)     (Statement S2)
  ENDDO
  b(i) = b(i) - s(i)               (Statement S3)
ENDDO
```

In this example, the dependence graph has two strongly connected components, one with S_1, the other one with S_2 and S_3. We can thus apply a loop distribution to separate S_1 from S_2 and S_3. Furthermore, we integrate S_3 into the second loop, so as to obtain only perfect loops nests. We get:

```
DOPAR i = 2, n
  s(i) = 0
ENDDO
DO i = 2, n
  DO j = 1, i
    IF (j ≤ i - 1) THEN
```

```
      s(i) = s(i) + a(j, i) b(j)
    ENDIF
    IF (j = i) THEN
      b(i) = b(i) - s(i)
    ENDIF
  ENDDO
ENDDO
```

Applying our scheduling algorithm, we find that the vector $X = (x, y)$ has to satisfy the constraints $y \geq 1$, $x + y \geq 0$, $x \geq 0$, and $y \geq 0$. We find $X = (0, 2)$ and $\rho_{S_2} = 1$ and $\rho_{S_3} = 0$ which corresponds to the following code (once again without any effort to remove if test):

```
DOPAR i = 2, n
  s(i) = 0
ENDDO
DO j = 1, n
  IF (j ≥ 2) THEN
      b(j) = b(j) - s(j)
  ENDIF
  DOPAR i = j+1, n
      s(i) = s(i) + a(j, i) b(j)
  ENDDO
ENDDO
```

7 Conclusion

We have presented an original scheduling algorithm to parallelize loops whose dependences are described through polyhedral reduced dependence graphs, i.e. reduced dependence graphs whose edges are labeled by an approximation of distance vectors by polyhedra. This representation of dependences is a generalization of direction vectors.

Our algorithm is nearly optimal, in the sense that it detects the maximal number of parallel loops that can be found, i.e. the minimal number of nested sequential loops (up to loop coalescing), as long as the only information available is the polyhedral reduced dependence graph. In particular, our algorithm is optimal for direction vectors, which generalizes Wolf and Lam's algorithm to the case of multiple statements.

We illustrated its power on several examples, examples with direction vectors as well as examples with more general polyhedral representations of distance vectors. All examples have been derived automatically with the algorithm we implemented and with the help of tools such as Tiny or Petit.

It remains some work for handling non perfectly nested loops. Our algorithm is indeed mainly well adapted for perfectly nested loops, or for common loops in non perfect codes. However, to better exploit information on non common loops, a promising approach is to develop a method to transform non perfect loop nests into perfect loop nests. This transformation remains to be fully automated.

Acknowledgments

The authors would like to thank François Irigoin for his wise advice, and for his always pertinent remarks.

References

[AK82] J.R. Allen and K. Kennedy. PFC: a program to convert programs to parallel form. Technical report, Dept. of Mathematical Sciences, Rice University, Houston, TX, March 1982.

[AK87] J.R. Allen and K. Kennedy. Automatic translations of Fortran programs to vector form. *ACM Toplas*, 9:491–542, 1987.

[Ber66] A. J. Bernstein. Analysis of programs for parallel processing. In *IEEE Trans. on El. Computers, EC-15*, 1966.

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.

[d'A96] Patrick Le Gouëslier d'Argence. *Contribution à l'étude des problèmes d'ordonnancement cyclique multidimensionnel*. PhD thesis, Université Paris VI, January 1996.

[DKR91] Alain Darte, Leonid Khachiyan, and Yves Robert. Linear scheduling is nearly optimal. *Parallel Processing Letters*, 1(2):73–81, 1991.

[DR94] Alain Darte and Yves Robert. Constructive methods for scheduling uniform loop nests. *IEEE Trans. Parallel Distributed Systems*, 5(8):814–822, 1994.

[DR95] Alain Darte and Yves Robert. Affine-by-statement scheduling of uniform and affine loop nests over parametric domains. *J. Parallel and Distributed Computing*, 29:43–59, 1995.

[DV94] Alain Darte and Frédéric Vivien. Automatic parallelization based on multi-dimensional scheduling. Technical Report 94-24, LIP, ENS-Lyon, France, September 1994.

[DV95a] Alain Darte and Frédéric Vivien. Revisiting the decomposition of Karp, Miller, and Winograd. *Parallel Processing Letters*, 5(4):551–562, December 1995.

[DV95b] Alain Darte and Frédéric Vivien. A classification of nested loops parallelization algorithms. In *INRIA-IEEE Symposium on Emerging Technologies and Factory Automation*, pages 217–224. IEEE Computer Society Press, 1995.

[DV96] Alain Darte and Frédéric Vivien. On the optimality of Allen and Kennedy's algorithm for parallelism extraction in nested loops. Technical Report 96-05, LIP, ENS-Lyon, France, February 1996.

[Fea91] Paul Feautrier. Dataflow analysis of array and scalar references. *Int. J. Parallel Programming*, 20(1):23–51, 1991.

[Fea92a] Paul Feautrier. Some efficient solutions to the affine scheduling problem, part I, one-dimensional time. *Int. J. Parallel Programming*, 21(5):313–348, October 1992.

[Fea92b] Paul Feautrier. Some efficient solutions to the affine scheduling problem, part II, multi-dimensional time. *Int. J. Parallel Programming*, 21(6):389–420, December 1992.

[GM84] M. Gondran and M. Minoux. *Graphs and Algorithms*. John Wiley, 1984.

[IJT91] F. Irigoin, P. Jouvelot, and R. Triolet. Semantical interprocedural parallelization: an overview of the PIPS project. In *Proceedings of the 1991 ACM International Conference on Supercomputing*, Cologne, Germany, June 1991.

[IT87] F. Irigoin and R. Triolet. Computing dependence direction vectors and dependence cones with linear systems. Technical Report ENSMP-CAI-87-E94, Ecole des Mines de Paris, Fontainebleau (France), 1987.

[IT88] F. Irigoin and R. Triolet. Supernode partitioning. In *Proc. 15th Annual ACM Symp. Principles of Programming Languages*, pages 319–329, San Diego, CA, January 1988.

[KMP+95] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and D. Wonnacott. *New user interface for Petit and other interfaces: user guide.* University of Maryland, June 1995.

[KMW67] R.M. Karp, R.E. Miller, and S. Winograd. The organization of computations for uniform recurrence equations. *Journal of the ACM*, 14(3):563–590, July 1967.

[KS88] S. Rao Kosaraju and Gregory F. Sullivan. Detecting cycles in dynamic graphs in polynomial time (preliminary version). In ACM Press, editor, *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*, pages 398–406, May 1988.

[Rao85] Sailesh K. Rao. *Regular Iterative Algorithms and their Implementations on Processor Arrays.* PhD thesis, Stanford University, October 1985.

[Roy88] Vwani P. Roychowdhury. *Derivation, Extensions and Parallel Implementation of Regular Iterative Algorithms.* PhD thesis, Stanford University, December 1988.

[Sch86] Alexander Schrijver. *Theory of Linear and Integer Programming.* John Wiley and Sons, New York, 1986.

[WL91] Michael E. Wolf and Monica S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Trans. Parallel Distributed Systems*, 2(4):452–471, October 1991.

[Wol82] M. Wolfe. *Optimizing Supercompilers for Supercomputers.* PhD thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, October 1982.

[Wol89] Michael Wolfe. *Optimizing Supercompilers for Supercomputers.* MIT Press, Cambridge MA, 1989.

[Wol90] Michael Wolfe. *TINY, a loop restructuring research tool.* Oregon Graduate Institute of Science and Technology, December 1990.