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
{darte,fvivien}@lip.ens-lyon.fr

Abstract

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

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 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.

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

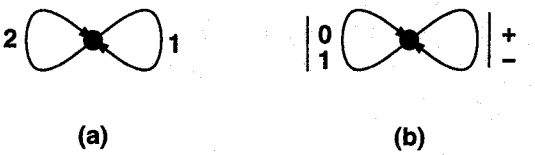

Figure 1. Reduced Dependence Graph for Example 1: (a) by levels, (b) by direction vectors.

Allen and Kennedy [1] 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 [17] 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 [3, 4] 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 [10, 11] 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

1089-795X/96 $5.00 © 1996 IEEE
Proceedings of PACT '96

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, reveals one degree of parallelism. The corresponding parallelized code is:

```
DO j = 3, 3n
  DOPAR i = max(1, ⌈(j-n)/2⌉), min(n, ⌊(j-1)/2⌋)
    a(i, j - 2i) = a(j - 2i, i) + 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:

$$Xd \geq 1 \quad Xv \geq 1 \quad Xr \geq 0$$

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.

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 simulates ϕ and a null-weight edge to come back to the initial node S.

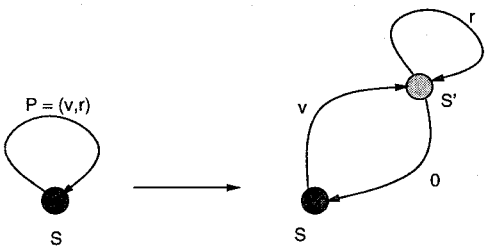

Figure 2. Simulation of an edge labeled by a polyhedron with one vertex and one ray.

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 [17] 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 enlightens the link between the maximal

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

degree of parallelism that can be detected and the accuracy of dependence abstractions. See [5] 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. Unfortunately, because of a lack of space, we can not give the full proofs of correctness and optimality of our algorithm. We refer to [8] in which all proofs are detailed.

Nevertheless, in Section 4, we summarize our results by showing how they are related to techniques developed for systems of uniform recurrence equations [6]. We illustrate the algorithm on a quite complicated example.

Finally, in Section 6, we discuss some implementation strategies that permit to reduce the complexity of the algorithm and to clean up the solution for code generation. Then, we briefly show how extending the algorithm to non perfect loop nests and 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 6.2. With this restriction, we can 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 [2]. 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 $S_i \stackrel{e}{\rightarrow} S_j$ 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:

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

For a certain class of nested loops, it is possible to express exactly this set of pairs (I, J) (see [9]): I is given as an affine function $f_{i,j}$ of J where J varies in a polyhedron $\mathcal{P}_{i,j}$:

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

[2] In some cases, output and anti dependences can be removed by data renaming and/or expansion. See for example [9].

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. 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 for Allen and Kennedy's parallelizing algorithm [1].
- **direction vector**, introduced by Wolfe [18] and used in Wolf and Lam's parallelizing algorithm [17].
- **dependence polyhedron**, introduced in [12] and used in Irigoin and Triolet's supernode partitioning algorithm [13].

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

We first recall the mathematical definition of a polyhedron and its decomposition 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.

A polyhedron can always be decomposed as the sum of a (convex) polytope and of a polyhedral cone (for more details see [16]). 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:

$$p = \sum_{i=1}^{\omega} \mu_i v_i + \sum_{i=1}^{\rho} \nu_i r_i + \sum_{i=1}^{\lambda} \xi_i l_i \qquad (2)$$

with $\mu_i \in \mathbb{Q}^+$, $\nu_i \in \mathbb{Q}^+$, $\xi_i \in \mathbb{Q}$, and $\sum_{i=1}^{\omega} \mu_i = 1$.

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 *A **polyhedral reduced dependence graph (PRDG)** is a RDG, for which each edge e 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, 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** and the second one **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: z if the dependence is uniform in this dimension with unique value z, $z+$ (resp. $z-$) if all i-th components are greater

(resp. smaller) than or equal to z, and $*$ if the i-th component may take any value. In general, + (resp. -) is used as shorthand for 1+ (resp. (-1)-).

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. 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 [7]) 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 null, or ∞ if all components are null.

A dependence level is also a representation by a polyhedron. 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

Our 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 [19].

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.

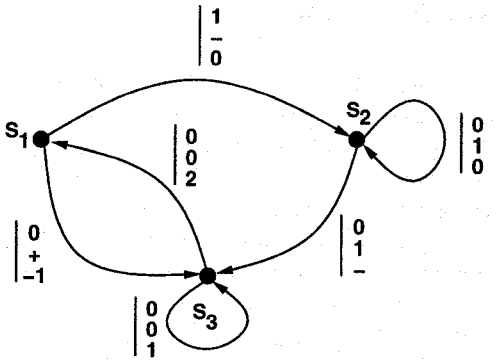

Figure 3. Reduced dependence graph with direction vectors, for Example 2.

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 has been made so as to remove "if" tests). For each statement, one level of parallelism has been detected.

This code has been generated by the the procedure "codegen" of the Omega Calculator delivered with Petit [15], thanks 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.

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.

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 $e = (x_e, y_e) \in E$ do
  1. If $\rho = 0$ and $\lambda = 0$ (the polyhedron is reduced to a polytope)
     - Add to F ω edges of weights v_1, v_2, ..., v_ω directed from x_e to y_e.
  2. 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, ..., v_ω 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.

Back to Example 2 The uniform dependence graph associated to the PRDG of Example 2 (see Figure 3) is depicted in Figure 4. It has three new nodes (i.e. virtual nodes) that correspond to the symbol "+" and the two symbols "−" in the initial direction vectors.

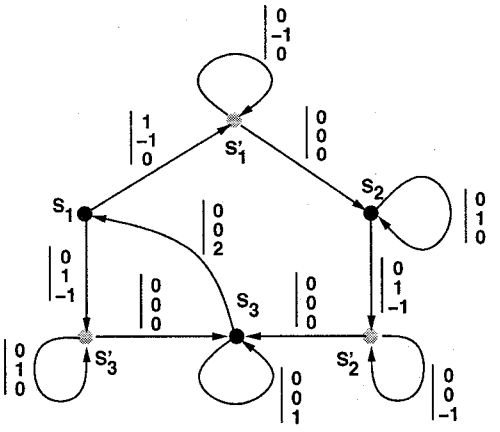

Figure 4. Translated uniform reduced dependence graph

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.

G' can be built by one linear programming resolution. Indeed, it has been shown (see [6]) that the edges of G' are exactly the edges e for which $v_e = 0$ in any optimal solution of the following linear program:

$$\min \{ \sum_e v_e \mid q \geq 0,\ v \geq 0,\ q + v \geq 1,\ Cq = 0,\ Wq = 0 \} \quad (3)$$

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).

Once G' is built, 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 following 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, ..., $X_S^{d_S}$ and

a sequence of constants $\rho_S^1, \ldots, \rho_S^{d_S}$ that define a valid multi-dimensional schedule.

DARTE-VIVIEN(G, k)

1. Build G' the subgraph of null weight multi-cycles of G.
2. 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.
3. Select a vector X and, for each node S in G, a constant ρ_S such that:

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

   For all actual nodes S of G, let $\rho_S^k = \rho_S$ and $X_S^k = X$.
4. If G' is empty or has only virtual nodes, return.
5. 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.
6. 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

- Step (2) is necessary only for general PRDGs: for example, it could be removed for RDGs labeled by direction vectors. In this case, Steps (1) and (3) can be solved by a single linear program resolution.
- In Step (3), we do not specify, on purpose, how the vector X and the constants ρ are selected, so as to allow various selection criteria, depending if fine or medium grain parallelism is desired. For example, a maximal set of linearly independent vectors X can be selected if the goal is to derive fully permutable loops (see [5]).

Back to Example 2 Consider the uniform dependence graph of Figure 4. 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 5.

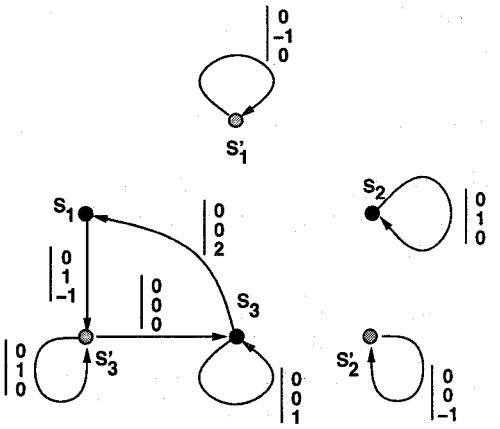

Figure 5. Subgraph of null weight multi-cycles for Example 2.

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.

4. Properties of the parallelization algorithm

Note the particular structure of G_u: it is a graph with edges labeled by integral vectors, i.e. a uniform dependence graph. However, the weights of the edges are not necessarily lexicographically positive which makes a huge difference with classical uniform dependence graphs of nested loops.

Actually, G_u is very similar to a reduced dependence graph associated to a system of uniform recurrence equations. The only difference is that some nodes of G_u are virtual nodes. This difference is small and this is why we can still use (with slight modifications) all techniques we previously developed for systems of uniform recurrence equations [6].

The correctness and optimality of our algorithm comes from this strong link between the "uniformized" graph G_u and systems of uniform recurrence equations. In particular,

we have the following results, whose proofs are detailed in [6] and [8].

4.1. Correctness

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

Furthermore, G_u has no cycle of null weight containing an actual node if and only if DA(G_u) = TRUE, where DA is the decomposition algorithm given below. Therefore, a PRDG G_o is computable if and only if DA(G_u) = TRUE.

Algorithm DA is a modified version of the seminal decomposition of Karp, Miller, and Winograd [14].

$\bigwedge$ denotes the logical AND.

Boolean DA(G)

1. Build G' the subgraph of null weight multi-cycles of G.
2. 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 $\bigwedge_{i=1}^{s} \text{DA}(G'_i)$.

This decomposition is related to the computability problem. However, considering at each step the dual of linear program 3 establishes the link with the scheduling problem. This is, without entering the details, what makes correct our scheduling algorithm. Algorithm DA is indeed the skeleton of the algorithm of Section 3.3.

4.2. Optimality

In the scheduling algorithm, each statement S is scheduled by a d_S-multi-dimensional schedule. d_S is called the depth of S. It is equal to the number of recursive calls (counting the first one) needed to remove S from the graph, except if S do not belong to a cycle, in which case $d_S = 0$. d, the depth of the graph, is the maximal d_S.

For systems of uniform recurrence equations, the depth of a graph is a measure of the degree of parallelism it describes. This result still holds for PRDGs and the depth d defined above. Indeed, we have the following results:

Theorem 2 *If $G_o = (V, E)$ is computable, the multi-dimensional scheduling function T:*

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

defines a valid multi-dimensional schedule for G_o.

Furthermore, this schedule is optimal, in the sense 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.

More precisely:

Theorem 3 *Assume that the iteration domain $\mathcal{D}$ is contained in a n-dimensional cube of size $O(N)$ and contains a n-dimensional cube of size $\Omega(N)$. Then, 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.*

5. Yet another example

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. After parallelization, S_1 is surrounding by a single sequential loop and S_2 by two.

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 6 has been found by the dependence analyzer Tiny [19]. The uniformization step transforms G_o into G_u which is depicted in Figure 7.

There is a multi-cycle of null weight generated by all edges whose weight is orthogonal to $(1, 0, 0)$ (see Figure 7). 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.

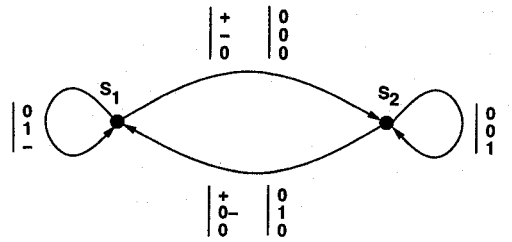

Figure 6. Reduced dependence graph with direction vectors, for Example 3.

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 8.

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 8 how this 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
```

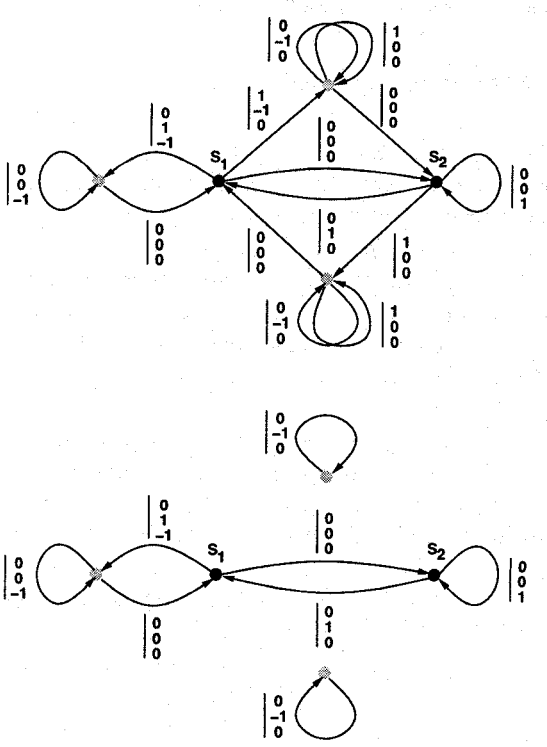

Figure 7. Translated uniform dependence graph for Example 3 and its corresponding G'.

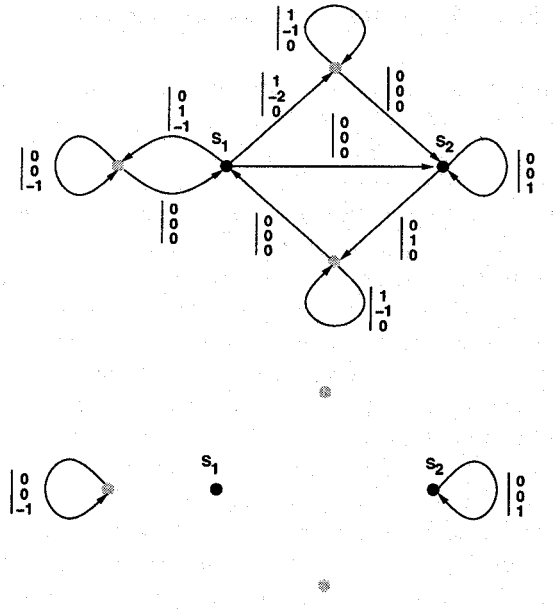

Figure 8. Polyhedral reduced dependence graph for Example 3 and its corresponding G'.

6. Implementation strategies

6.1. Reasoning on cycles

Consider the constraints of Step (3). It can be shown 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 by X, 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 suggests to adopt a two steps strategy: compute X first by a linear programming approach, and then compute the constants ρ by a graph-based technique. 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, which simulate all constraints on cycles with only $|E| - |V| + 1$ cycles. This technique permits us to keep small the number of constraints and variables in the linear programs to solve. Furthermore, it guarantees that the constants ρ are simple if the vector X is simple. This property is highly desirable for code generation. Full details can be found in [8].

6.2. 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 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 exploits 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 [15].

Example 4

```
DO i = 2, n
  s(i) = 0
  DO j = 1, i-1
    s(i) = s(i) + a(j, i) b(j)
  ENDDO
  b(i) = b(i) - s(i)
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
```

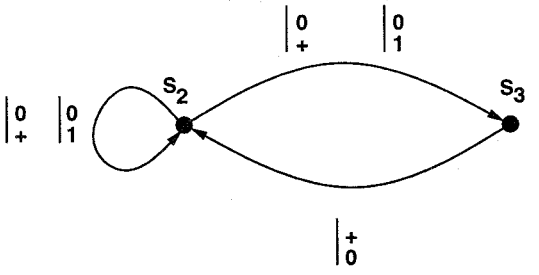

Figure 9. Reduced dependence graph with direction vectors, for Example 4.

The reduced dependence graph, with direction vectors, for the two last statements is depicted in Figure 9. It is easy

to see that the corresponding uniformized dependence graph has no multi-cycle of null weight. Therefore, there is some parallelism. Indeed, applying our scheduling algorithm, we find that the vector $X = (x, y)$ has to satisfy the constraints $y \geq 1, x+y \geq 2, x \geq 0$, 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, 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 the practical efficiency of our algorithm 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 to do for handling non perfectly nested loops. Our algorithm is indeed 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 automatized.

References

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

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

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

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

[5] A. Darte and F. 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.

[6] A. Darte and F. Vivien. Revisiting the decomposition of Karp, Miller, and Winograd. *Parallel Processing Letters*, 5(4):551–562, Dec. 1995.

[7] A. Darte and F. Vivien. On the optimality of Allen and Kennedy's algorithm for parallelism extraction in nested loops. Technical Report 96-05, LIP, ENS-Lyon, France, Feb. 1996. Extended version of Europar'96.

[8] A. Darte and F. Vivien. Optimal fine and medium grain parallelism in polyhedral reduced dependence graphs. Technical Report 96-06, LIP, ENS-Lyon, France, Apr. 1996.

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

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

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

[12] 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.

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

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

[15] 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.

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

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

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

[19] M. Wolfe. *TINY, a loop restructuring research tool*. Oregon Graduate Institute of Science and Technology, Dec. 1990.