Parallel Processing Letters
© World Scientific Publishing Company

PARALLELIZING NESTED LOOPS WITH APPROXIMATIONS OF DISTANCE VECTORS: A SURVEY*

ALAIN DARTE and FRÉDÉRIC VIVIEN
Laboratoire LIP, École Normale Supérieure de Lyon, France.
e-mail: [Alain.Darte,Frederic.Vivien]@lip.ens-lyon.fr

Received (received date)
Revised (revised date)
Communicated by (Name of Editor)

ABSTRACT

In this paper, we compare three nested loops parallelization algorithms (Allen and Kennedy's algorithm, Wolf and Lam's algorithm and Darte and Vivien's algorithm) that use different representations of distance vectors as input. We study the optimality of each with respect to the dependence analysis it uses. We propose well-chosen examples that illustrate the power and limitations of the three algorithms. This study identifies which algorithm is the most suitable for a given representation of distance vectors.

Keywords: automatic parallelization, loop nest, dependence analysis, reduced dependence graph, distance vector.

1 Introduction

Loop transformations have been shown to be useful for extracting parallelism from regular nested loops for a large class of machines. Of course, a different optimized code corresponds to each type of machine: depending on the memory hierarchy, the granularity of the final code must be carefully chosen to optimize memory accesses. Fine-grain parallelism may be efficient for vector machines, whereas for distributed memory machines coarse-grain parallelism (obtained by tiling or blocking techniques) is preferable and allows the reduction of inter-processor communications.

However, detecting parallelism in loops (i.e. transforming DO loops into DOALL loops), and understanding parallelism (i.e. detecting the dependences responsible for the sequentiality in the code) is independent of the target architecture. It only depends on the structure of the sequential code to be parallelized. This is one of the reasons why a large number of algorithms have been proposed for detecting DOALL loops, as a first step of the parallelization process. First, one studies the problem of parallelization on an ideal machine, and then, further machine-dependent optimizations are taken into account such as the choice of granularity, the data distribution, the optimization of communications. This two-step approach

*Supported by the CNRS-INRIA project ReMaP

is used not only in the field of automatic nested loops parallelization, but also, among others, for general task scheduling and software pipelining.

This paper studies the three main parallelism detection algorithms that work with a description of *distance vectors* [a], the algorithms of Allen and Kennedy [1], Wolf and Lam [20], and Darte and Vivien [9]. These algorithms seem very different not only by the techniques they use (computations of strongly connected components, computations of unimodular matrices, and resolution of linear programs, respectively), but also by the description of dependences they work with (approximation of distance vectors by dependence level, direction vectors, and polyhedra, respectively). Nevertheless, we try to identify the concepts that make these algorithms different or similar and we discuss their respective power and limitations. In [23], the problem of determining the minimal dependence abstraction needed to apply a given loop transformation has been addressed. Our study answers the dual question: it characterizes which algorithm is the most suitable for a given representation of dependences. In Section 2, we define the framework in which the three algorithms are discussed. In Section 3, we present, for each algorithm, its main concepts and its optimality result. We summarize our results in Section 4.

2 Input and output of parallelization algorithms

2.1 Nested loops

For the sake of clarity, we restrict ourselves to the case of perfectly nested DO loops with affine loop bounds, even if some of the algorithms can handle more complicated nested loops. This restriction enables to identify, as usually, the iterations of n nested loops (n is the *depth* of the loop nest) with vectors in $\mathbb{Z}^n$ (the *iteration vectors*) contained in a finite convex polyhedron defined by the loop bounds (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 loop. In the sequential code, the iterations are therefore executed in the lexicographic order of the iteration vectors. We denote by P the polyhedral iteration domain, by I and J n-dimensional iteration vectors in P, 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 $I = J$.

There are at least three ways to define a new order on the operations of a loop nest (i.e. to define the output of the parallelization algorithm) that can be expressed by nested loops. We can:

- use elementary loop transformations, such as loop distribution (Allen and Kennedy [1]), or loop interchange and loop skewing (Banerjee [2]);
- apply a linear change of basis on the iteration domain, i.e. apply a unimodular transformation on the iteration vectors (Wolf and Lam [20]).
- define a d-dimensional schedule, i.e. an affine transformation from $\mathbb{Z}^n$ to $\mathbb{Z}^d$ interpreted as a multi-dimensional timing function. Each component will

[a] We do not consider Feautrier's algorithm [13] in this classification as it works with an exact description of dependences, and not with an approximation. In this sense, it is more powerful. However discussing its optimality is much more complicated. See Section 3.4 for some hints.

correspond to a sequential loop, the missing $(n-d)$ dimensions will correspond to DOALL loops (Feautrier [13], Darte and Vivien [9]).

Here, we do not discuss the rewriting process needed to obtain some loop nests from these three transformation schemes (see [20,22,6,5]), but we discuss the link between the loops transformations (the output) and the dependence representation (the input). We want to characterize, for a given dependence representation, which algorithm is optimal, i.e. exhibits the maximal number of parallel loops.

2.2 Representations of dependences

To each iteration of the loops that surround a statement corresponds a particular execution of the statement, called an *operation*. Dependence relations between operations are defined by Bernstein's conditions [3]: two operations are dependent if both access the same memory location and if at least one access is a write. The dependence is directed according to the sequential order. 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 $S_i(I) \Longrightarrow S_j(J)$. Executing the operations while respecting the partial order specified by the EDG guarantees that the result of the loop nest is preserved. Generally, the EDG cannot be used, as it is too large or may not be computable exactly at compile-time. Thus, one prefers to manipulate the *reduced dependence graph* (RDG), a compression of the EDG.

The RDG has one vertex per statement in the loop nest. Two statements S_i and S_j of the RDG are said dependent (we write $S_i \rightarrow S_j$) if there exists one pair (I,J) such that $S_i(I) \Longrightarrow S_j(J)$. The dependence $S_i \rightarrow S_j$ is labeled by the set $\{(I,J) \in P^2 \mid S_i(I) \Longrightarrow S_j(J)\}$, or by an approximation that contains this set. The precision of this approximation makes the power of the dependence analysis. See [24] for a survey on dependence tests, and [12] for more details on exact dependence analysis. Since its input is the RDG and not the EDG, a parallelization algorithm cannot distinguish between two different EDGs which have the same RDG: only the parallelism contained in the RDG can be detected. Thus, the quality of a parallelization algorithm must be studied *with respect to* the dependence analysis.

For a certain class of loop nests, one can exactly express the set of pairs (I,J) (see [12]): I is given as an affine function $f_{i,j}$ of J where J varies in a polyhedron $\mathcal{P}_{i,j}$: $\{(I,J) \in P^2 \mid S_i(I) \Longrightarrow S_j(J)\} = \{(f_{i,j}(J), J) \mid J \in \mathcal{P}_{i,j} \subset P\}$. In most dependence analysis algorithms however, rather than the set of pairs (I,J), one computes the set of *distance vectors* or *dependence vectors* $(J-I)$. When exact dependence analysis is feasible, the above equation 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). The representation by distance vectors is not equivalent to the representation by pairs, since the information concerning the *location* in the EDG of such a distance is lost. This lack of information may prevent detecting parallelism (see Section 3.4). Nevertheless, the representation of dependences by approximation of distance vectors remains important, especially

when exact dependence analysis is either too expensive or not feasible. Classical representations of distance vectors (by increasing precision) are:

Level of dependence: introduced by Allen and Kennedy in [1]. A distance vector $J - I$ is approximated by an element l (the *level*) in $[1, n] \cup \{\infty\}$, defined as ∞ if $J - I = 0$, or as the largest integer such that the $l - 1$ first components of the distance vector are zero. When $l = \infty$, the dependence is said *loop independent*, and *loop carried* otherwise.

Direction vector: first introduced by Lamport in [17], then by Wolfe in [21]. A set of distance vectors between S and S' is 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 component of the distance vectors: $z+$ means $\geq z$, $z-$ means $\leq z$, and $*$ means any value.

Dependence polyhedron: introduced by Irigoin and Triolet [14]. A set of distance vectors between S and S' is approximated by a subset of $\mathbb{Z}^n$, defined as the integral points of a polyhedron.

A dependence at level p and the direction vector $(\overbrace{0, \ldots, 0}^{p-1}, 1+, \overbrace{*, \ldots, *}^{n-p})$ are equivalent, and direction vectors are particular polyhedra (e.g. the direction vector $(1, 2+, *)$ corresponds to a polyhedron with one vertex $(1, 2, 0)$, one ray $(0, 1, 0)$ and one line $(0, 0, 1)$). Thus, the representation by polyhedra is the most general one.

3 A classification of different loops parallelization algorithms

In this section, we present the main ideas of Allen and Kennedy's algorithm, Wolf and Lam's algorithm, and Darte and Vivien's algorithm. For each algorithm, we give an example that illustrates its power and an example that illustrates its limitations.

3.1 Allen and Kennedy's algorithm

Allen and Kennedy's algorithm [1] is based on two facts: 1) the outermost loop is parallel if it has no loop carried dependences, i.e. if there is no dependence with level 1; 2) all iterations of statement S_i can be carried out before any iteration of statement S_j if there is no dependence in the RDG from S_j to S_i. The first property allows to mark a loop as DOALL or DOSEQ, whereas the second suggests that parallelism detection can be done independently in each strongly connected component of the RDG. The input of the algorithm is a RDG whose edges are labeled by dependence levels. Parallelism extraction is done by loop distribution.

For a dependence graph G, we denote by $G(k)$ the subgraph of G in which all dependences at level $< k$ have been removed. Here is a sketch of the algorithm in its most basic formulation. The initial call is ALLEN-KENNEDY(RDG, 1).

Allen-Kennedy(G, k)

- If $k > n$, stop.
- Decompose $G(k)$ into its strongly connected components G_i and sort them topologically.

- Rewrite code so that each G_i belongs to a different loop nest (at level k) and the order on the G_i is preserved (distribution of loops at level $\geq k$).
- For each G_i, mark the loop at level k as a **DOALL** loop if G_i has no edge at level k. Otherwise mark the loop as a **DOSEQ** loop.
- For each G_i, call ALLEN-KENNEDY(G_i, $k+1$).

Example 1

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

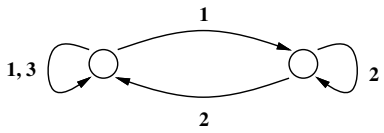

Fig. 1: Example 1 and its Reduced Dependence Graph (with dependence levels)

The dependence graph $G = G(1)$ of Example 1, see Figure 1, has one edge at level 1 and only one strongly connected component, thus the first call simply marks the outer loop as DOSEQ. At level 2 (the edge at level 1 is not considered), $G(2)$ has two strongly connected components: all computations on array b can be carried out before any computation on array a. With a loop distribution at levels 2 and 3, we get the code:

```
  DOSEQ 1 i=1,n
    DOSEQ 2 j=1,n
      DOALL 2 k=1,n
        b(i,j,k) = b(i,j-1,k+j) + a(i-1,j,k)
2   CONTINUE
    DOALL 3 j=1,n
      DOSEQ 3 k=1,n
        a(i,j,k) = a(i-1,j+i,k) + a(i,j,k-1) + b(i,j-1,k)
3   CONTINUE
1 CONTINUE
```

Property 1 *Algorithm* ALLEN-KENNEDY *is optimal among all parallelism detection algorithms whose input is a RDG labeled by dependence levels.*

Proof. The optimality result originally presented by Allen and Kennedy concerns mainly the minimization of synchronization barriers. In terms of parallelism detection, it has only been proved that ALLEN-KENNEDY detects the maximum number of parallel loops if the way loops are nested remains unchanged, i.e. if loop distribution/fusion is the only possible transformation. In [11] a stronger result has been proved: for any set of loops L, whose dependences are described by a RDG G labeled by dependence levels, there exists a set of loops L' with the same dependence graph G and that contains exactly the same degree of parallelism detected by ALLEN-KENNEDY for L. Since L and L' can not be distinguished in terms of RDG, this proves the optimality among all parallelizing algorithms. □

Property 1 shows that ALLEN-KENNEDY is well adapted to a representation of dependences by dependence levels. Therefore, detecting more parallelism than

found by ALLEN-KENNEDY is possible only with more accurate dependence approximations. For example, it is possible to overcome ALLEN-KENNEDY when a simple interchange (Example 2) or a simple skew and an interchange (Example 3) reveal parallelism (see dependence graphs on Figure 2).

Examples 2 and 3

```
DO i=1,n                          DO i=1,n
   DO j=1,n                          DO j=1,n
      a(i,j) = a(i-1,j-1) + a(i,j-1)    a(i,j) = a(i-1,j) + a(i,j-1)
CONTINUE                          CONTINUE
```

Fig. 2: Reduced Dependence Graphs for Examples 2 and 3

3.2 Wolf and Lam's algorithm

Examples 2 and 3 contain parallelism which cannot be extracted if the dependences are represented by dependence levels. To remedy this limitation, Wolf and Lam [20] proposed an algorithm that uses direction vectors. Their work unified all previous algorithms based on elementary matrix operations such as loop skewing, loop interchange, loop reversal, in a unique framework, the framework of *valid unimodular transformations*.

Looking for unimodular transformations is of practical interest since they are (1) linear, (2) invertible in $\mathbb{Z}^n$. Given a unimodular transformation T, property (1) enables to check if T is valid (T is valid if $Td >_l 0$ for all non zero distance vectors d) and property (2) enables to rewrite easily the code (simple change of basis in $\mathbb{Z}^n$). In general, since $Td >_l 0$ cannot be checked for all *distance* vectors, one guarantees $Td >_l 0$ for all non zero *direction* vectors, with the usual arithmetic conventions in $\mathbb{Z} \cup \{*\} \cup (\mathbb{Z} \times \{+,-\})$. In WOLF-LAM, all non zero direction vectors are assumed to be lexicographically positive (see Section 2.2). Due to a lack of space, we refer to the original paper [20] for full details. Here, we just point out its fundamental concept: the dependence cone Γ, defined below.

Denote by $t_1, \ldots, t_n$, the rows of T. For a direction vector d: $Td >_l 0$ if and only if $\exists k_d$, $1 \leq k_d \leq n$ such that $\forall i, 1 \leq i < k_d$, $t_i d = 0$ and $t_{k_d} d > 0$: the dependences represented by d are transformed by T into dependences of level k_d. If $k_d = 1$ for all direction vectors d, all dependences are carried by the first loop, and all inner loops are DOALL loops. t_1 is then called a *timing vector* or *separating hyperplane (see also [17])*. Such a timing vector exists if and only if Γ, the closure of the cone generated by all direction vectors, is pointed. This is equivalent to the fact that the cone Γ^+ - defined by $\Gamma^+ = \{y \mid \forall x \in \Gamma,\ y.x \geq 0\}$ - is full-dimensional (see [19] for details on cones and related notions). Building T from n linearly independent vectors of Γ^+ enables to transform the loops into n fully *permutable* loops. When Γ is not pointed, Γ^+ has a dimension r, $1 \leq r < n$, where $n - r$ is the dimension

of the lineality space of Γ. With r linearly independent vectors of Γ^+, one can transform the loop nest so that the r outermost loops are fully permutable. Then, one can recursively apply the same technique for transforming the $n-r$ innermost loops, considering the direction vectors not already carried by at least one of the r outermost loops (i.e that belong to the lineality space of Γ). This is the idea of Wolf and Lam's algorithm even if it is not explicitly described in these terms in [20].

The notion of timing vectors is in the heart of the hyperplane method and its variants [17,7], which are particularly interesting for exposing fine-grain parallelism, whereas the notion of fully permutable loops is the base of all tiling techniques [15,18,4,20,8], which are used for exposing coarse-grain parallelism. As said before, both formulations are equivalent when reasoning on Γ^+.

Example 4

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

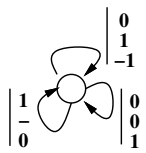

Fig. 3: Example 4 and its Reduced Dependence Graph (with direction vectors)

We illustrate WOLF-LAM with the example of Figure 3. The set of direction vectors is $D = \{(1,-,0),(0,0,1),(0,1,-1)\}$. The lineality space of $\Gamma(D)$ is two-dimensional, generated by $(0,1,0)$ and $(0,0,1)$. Thus, $\Gamma^+(D)$ is one dimensional and generated by $X_1 = (1,0,0)$ and the outermost loop is sequential. It remains to consider $D' = \{(0,0,1),(0,1,-1)\}$. $\Gamma(D')$ is pointed. We can expose two permutable loops by defining two vectors of $\Gamma^+(D')$, e.g. $X_2 = (0,1,0)$ and $X_3 = (0,1,1)$. The transformation matrix whose rows are X_1, X_2, X_3 is unimodular and corresponds to a simple loop skewing. For exposing DOALL loops, we choose X_2 in the relative interior of Γ^+, for example $X_2 = (0,2,1)$, and we can take $X_3 = (0,1,0)$. This means skewing the loop k by factor 2 and then interchanging loops j and k:

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

Wolf and Lam showed that their methodology is optimal [20]: "An algorithm that finds the maximum coarse grain parallelism, and then recursively calls itself on the inner loops, produces the maximum degree of parallelism possible". Actually, they just proved that their algorithm is optimal among all algorithms that use unimodular transformations. A stronger optimality result can be established, derived from the optimality of Darte and Vivien's algorithm. Indeed, on a loop nest whose body has only one statement, and whose dependences are represented by direction vectors, DARTE-VIVIEN behaves as WOLF-LAM. Thus:

Property 2 WOLF-LAM *is optimal among all parallelism detection algorithms whose input is a set of direction vectors (implicitly, one thus considers that the loop nest has only one statement or that all statements form an atomic block).*

Therefore, as for ALLEN-KENNEDY, the sub-optimality of WOLF-LAM in the general case has to be found, not in the algorithm methodology, but in the weakness of its input: the fact that the structure of the RDG in terms of strongly connected components is not exploited results in a loss of parallelism. For example, WOLF-LAM finds no parallelism in Example 1 (whose RDG with direction vectors is given in Figure 4) because of the typical structure of the direction vectors $(1,-,0),(0,1,-),(0,0,1)$. With such vectors, $r = 1$ at each level of the recursive construction, and no permutable loops are detected.

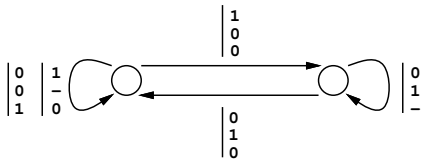

Fig. 4: Reduced Dependence Graph for Example 1 (with direction vectors)

3.3 Darte and Vivien's algorithm

The idea of Darte and Vivien's algorithm is to combine in an optimal way ALLEN-KENNEDY and WOLF-LAM so as to exploit both the structure of the RDG and the structure of the direction vectors. The key concept is now the cone generated by the *weights of the cycles* of the RDG (instead of the weights of the edges). It leads to a multi-dimensional scheduling algorithm [9] which is a combination of unimodular transformation, loop distribution, and index-shift method. This algorithm works for RDG whose edges are labeled by a polyhedral representation of distance vectors. There are two steps: first the polyhedral RDG is uniformized into a uniform RDG, and then is scheduled with a variant of Karp, Miller and Winograd's decomposition for system of uniform recurrence equations [16]. The algorithm is fully described in [9]. We illustrate how it works through the following example:

Example 5

DO $i = 1, n$
  DO $j = 1, n$
    DO $k = 1, n$
      a(i, j, k) = b($i-1$, $j+i$, k) + b(i, $j-1$, $k+2$)
      b(i, j, k) = a(i, $j-1$, $k+j$) + a(i, j, $k-1$)
CONTINUE

The RDG with direction vectors is depicted in Figure 5. At levels 1 and 2, the RDG remains strongly connected, but is no more connected at level 3: therefore ALLEN-KENNEDY detects one degree of parallelism for both statements. WOLF-LAM, because of the vectors $(1,-,0)$, $(0,1,-)$ and $(0,0,1)$, detects no parallelism.

A description of distance vectors by polyhedra can be translated into a description by uniform dependences, by introducing one virtual node for each polyhedron, simulating vertices as in-coming edges, and rays and lines by self-dependences on this virtual node. In Example 5, we simulate the vector (1,-,0) by a polyhedron with

one vertex (1,-1,0) and one ray (0,-1,0). We introduce a virtual node and add an edge (1,-1,0), and a self-edge (0,-1,0) to this virtual node. This way, we uniformize the RDG. The result is shown on Figure 5 (virtual nodes are in gray).

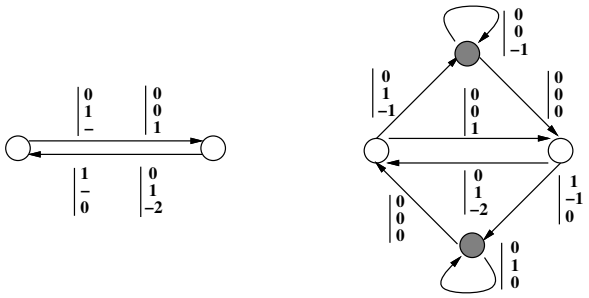

Fig. 5: Example 5: RDG for with direction vectors and uniformized RDG.

A RDG uniformized this way cannot be the RDG of a loop nest since dependence vectors are no more necessarily lexicographically non negative. In fact, forgetting that some vertices are virtual, this is the RDG of a system of uniform recurrence equations, introduced by Karp, Miller and Winograd [16]. The scheduling problem of such a RDG is dual to the problem of its computability and is linked to the detection of cycles of null weight, which can be solved by a recursive decomposition of the graph, based on the detection of multi-cycles (union of cycles) of null weight. The key structure is G', the subgraph of G generated by the edges that belong to a multi-cycle of null weight. Edges that do not belong to G' can be scheduled by a shifted-linear schedule (i.e. a linear schedule associated to a shift of the indices). Other edges are recursively satisfied in the remaining dimensions. To see the link with Wolf-Lam, G' is the subgraph whose cycle weights generate the lineality space of Γ, where Γ is now the cone generated by the *weights of the cycles*. Choosing a vector in Γ^+ enables to define a scheduling vector for all cycles not in G' and to define a shifted-linear schedule that satisfies all edges not in G'.

We come back to Example 5. The transformed RDG has 4 vertices (two are virtual). The weights of elementary cycles are $(0,0,-1)$ and $(0,-1,0)$ for the self-loops and $(1,0,-1)$, $(1,-1,1)$, $(0,2,-3)$, $(0,1,-1)$ for the other cycles. Thus, Γ is pointed and one finds a one-dimensional schedule, for example $X = (4,0,-2)$, $\rho_a = 0$, $\rho_b = 3$. Two degrees of parallelism are exposed and the resulting code is:

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

For Example 5, we considered all elementary cycles of the RDG. However, smarter linear programming resolutions enables to keep a polynomial complexity and enables to avoid computing all elementary cycles (which may be exponential in the number of edges for an arbitrary graph). We have the following optimality result:

Property 3 *Algorithm* DARTE-VIVIEN *is optimal among all parallelism detection algorithms whose input is a graph whose edges are labeled by a polyhedral representation of distance vectors.*

Proof. We just give the scheme of the proof. All details are provided in [10]. As DARTE-VIVIEN is recursive, one can associate to each statement S, the number d_S of recursive steps needed to satisfy all dependences concerning S. In the parallelized code, statement S will be surrounded by d_S sequential loops. Furthermore, for a loop nest whose iteration domain contains (resp. is contained in) a n-dimensional cube of size $O(N)$ (resp. $\Omega(N)$), one can build a dependence path in the EDG obtained by unrolling the RDG, that visits $\Omega(N^{d_S})$ times statement S. Therefore, any parallelization algorithm would expose a sequentiality of $\Omega(N^{d_S})$. Since the sequentiality exposed by DARTE-VIVIEN is $O(N^{d_S})$, it is, in this sense, optimal. □

3.4 Limitations of the representation by distance vectors

DARTE-VIVIEN is optimal for any polyhedral representation of *distance vectors*. However, it may not be optimal if more information is given on the *pair of iteration vectors* that induce a dependence. Indeed, the set of distance vectors $\{(J-I) \mid S_1(I) \Rightarrow S_2(J)\}$ is the *projection* of the set $\{(J-I, J) \mid S_1(I) \Rightarrow S_2(J)\}$ (which is as precise as the set of pairs $\{(I, J) \mid S_1(I) \Rightarrow S_2(J)\}$), projection which makes us believe that the distance vectors can take place anywhere in the iteration domain. Example 6 (Figure 6) illustrates this fact: when the dependences are described by distance vectors, we obtain the RDG of Figure 6. Therefore, there exists a multi-cycle of null weight and the two actual vertices belong to G'. The depth of DARTE-VIVIEN is 2 and no parallelism can be found. However, computing iteration (i, j) of the first statement (resp. the second statement) at step $2i + j$ (resp. $i + j$), leads to a valid schedule that exposes one degree of parallelism.

To build such a schedule one needs exact dependence analysis and must look for multi-dimensional schedules whose linear parts are different for different statements *even if they belong to the same strongly connected component.* These are the base of Feautrier's algorithm [13] whose fundamental mathematical tool is the affine form of Farkas' lemma [19]. Property 3 shows that there is no need to look for different linear parts (whose construction is more expensive and lead to more complicated rewriting processes) in a given strongly connected component of the current subgraph G', as long as dependences are given by distances vectors. On the other hand, Example 6 shows that it can be useful when a more accurate dependence analysis is available. The reason is that, as in Example 6, it may happen that the sum of the approximations of distance vectors along a cycle is not equal to the approximation

of the sum of these distance vectors. In other words, approximating the distance sets $\{(I,J) \mid S_i(I) \Longrightarrow S_j(J)\}$ is less accurate than approximating directly the sets $\{(I,J) \mid S_i(I) \stackrel{*}{\Longrightarrow} S_i(J)\}$ where $\stackrel{*}{\Longrightarrow}$ denotes the transitive closure of $\Longrightarrow$.

Example 6

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

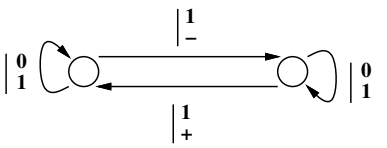

Fig. 6: Example 6 and its RDG with direction vectors.

4 Conclusion

Our study offers a classification of loop parallelization algorithms. Allen and Kennedy's algorithm is optimal for a representation of dependences by dependence levels and Wolf and Lam's algorithm is optimal for a representation by direction vectors (but for a loop nest with only one statement). Neither of them subsumes the other, since each uses information that can not be exploited by the other (graph structure for the first one, direction vectors for the second one). Both algorithms are subsumed by Darte and Vivien's algorithm which is optimal for any polyhedral representation of distance vectors. Feautrier's algorithm is an extension of the latter, but the characterization of its optimality remains open. This classification enables a compiler-parallelizer to choose, depending on the dependence analysis at its disposal, the simplest and cheapest parallelization algorithm that remains optimal, i.e the algorithm that is the most appropriate to the available representation of dependences. Future work will try to answer the open question of optimality of Feautrier's algorithm.

References

1. John R. Allen and Ken Kennedy. Automatic translation of Fortran programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, October 1987.
2. Utpal Banerjee. A theory of loop permutations. In Gelernter, Nicolau, and Padua, editors, *Languages and Compilers for Parallel Computing*. MIT Press, 1990.
3. Arthur J. Bernstein. Analysis of programs for parallel processing. *IEEE Transactions on Electronic Computers*, 15:757–762, October 1966.
4. Pierre Boulet, Alain Darte, Tanguy Risset, and Yves Robert. (pen)-ultimate tiling? *Integration, the VLSI Journal*, 17:33–51, 1994.
5. Jean-François Collard. Code generation in automatic parallelizers. In Claude Girault, editor, *Proc. Int. Conf. on Application in Parallel and Distributed Computing. IFIP WG 10.3*, pages 185–194. North Holland, April 1994.
6. Jean-François Collard, Paul Feautrier, and Tanguy Risset. Construction of DO loops from systems of affine constraints. *Parallel Processing Letters*, 5(3):421–436, September 1995.

7. Alain Darte, Leonid Khachiyan, and Yves Robert. Linear scheduling is nearly optimal. *Parallel Processing Letters*, 1(2):73–81, 1991.
8. Alain Darte, Georges-André Silber, and Frédéric Vivien. Combining retiming and scheduling techniques for loop parallelization and loop tiling. Technical Report 96-34, LIP, ENS-Lyon, France, November 1996.
9. Alain Darte and Frédéric Vivien. Optimal fine and medium grain parallelism detection in polyhedral reduced dependence graphs. In *Proceedings of PACT'96*, Boston, MA, October 1996. IEEE Computer Society Press.
10. Alain Darte and Frédéric Vivien. Optimal fine and medium grain parallelism detection in polyhedral reduced dependence graphs. Technical Report 96-06, LIP, ENS-Lyon, France, April 1996.
11. Alain Darte and Frédéric Vivien. On the optimality of Allen and Kennedy's algorithm for parallelism extraction in nested loops. *Journal of Parallel Algorithms and Applications*, 96. Special issue on Optimizing Compilers for Parallel Languages.
12. Paul Feautrier. Dataflow analysis of array and scalar references. *Int. J. Parallel Programming*, 20(1):23–51, 1991.
13. 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.
14. François Irigoin and Rémy 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.
15. François Irigoin and Rémy Triolet. Supernode partitioning. In *Proc. 15th Annual ACM Symp. Principles of Programming Languages*, pages 319–329, San Diego, CA, January 1988.
16. 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.
17. Leslie Lamport. The parallel execution of DO loops. *Communications of the ACM*, 17(2):83–93, February 1974.
18. Robert Schreiber and Jack J. Dongarra. Automatic blocking of nested loops. Technical Report 90-38, The University of Tennessee, Knoxville, TN, August 1990.
19. Alexander Schrijver. *Theory of Linear and Integer Programming*. John Wiley and Sons, New York, 1986.
20. 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.
21. Michael Wolfe. *Optimizing Supercompilers for Supercomputers*. MIT Press, Cambridge MA, 1989.
22. Jingling Xue. Automatic non-unimodular transformations of loop nests. *Parallel Computing*, 20(5):711–728, May 1994.
23. Yi-Qing Yang, Corinne Ancourt, and François Irigoin. Minimal data dependence abstractions for loop transformations. *International Journal of Parallel Programming*, 23(4):359–388, August 1995.
24. Hans Zima and Barbara Chapman. *Supercompilers for Parallel and Vector Computers*. ACM Press, 1990.