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

Alain Darte and Frédéric Vivien*
Laboratoire LIP, URA CNRS 1398
École Normale Supérieure de Lyon
F - 69364 LYON Cedex 07

July 9, 1997

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, this algorithm is optimal for the classical approximation by 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. It can also be viewed as a combination of both Allen and Kennedy's algorithm and Wolf and Lam's algorithm.

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

1 Introduction

Loop transformations are useful source to source transformations for improving the performance of programs, for detecting parallelism, and for improving data locality. Year after year, a catalog of such transformations has been developed (see for example the survey paper [3]) but how to combine these transformations to achieve some particular goal remains a research topic. In the past ten years, many researchers have proposed various algorithms and frameworks that unify (some of) these code transformations: Allen and Kennedy's algorithm [2] for handling loop distribution and loop fusion, Banerjee's framework [4] and Wolf and Lam's algorithm [39] for handling unimodular transformations, i.e. combinations of loop reversal, loop skewing and loop interchange, Kelly and Pugh's framework [31] and Feautrier's algorithm for handling general affine transformations [22].

Concerning this last framework, it is interesting to notice that fundamental differences exist between Kelly and Pugh's approach and Feautrier's approach: the first approach is a "try and test" heuristic, where transformations are selected and analyzed in terms of granularity, data locality, code complexity, while the second approach is a completely specified algorithm whose only goal is to build as many parallel inner loops as possible. Which approach is the best? Since none of them is implemented in a real compiler, we can only give theoretical motivations. The strength of Feautrier's approach is twofold: first, in a unified framework, it gives an upper bound on the maximal theoretical parallelism that can be detected by affine transformations, second its result can be reproduced by anybody since the algorithm is fully specified. This second advantage is also its weakness: it is not flexible enough to handle other objectives. For example, there is no guarantee concerning the simplicity of the generated code, and no clear way is given for increasing the granularity of

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

the parallelized code. On the other hand, even though these criteria can be taken into account by Kelly and Pugh, they can miss a good solution found by Feautrier's algorithm because of their heuristic strategy.

In fact, Kelly and Pugh prefer a heuristic approach because parallelism detection is not the only issue. Among other criteria, simple code generation also is fundamental, and we believe that it is important to control the complexity of the transformations generated by a parallelizing algorithm. With this idea in mind, we present a parallelization algorithm, simpler than Feautrier's algorithm, which captures simultaneously loop distribution, index-shift method, and unimodular transformations. It combines techniques related to system of uniform recurrence equations introduced by Karp, Miller and Winograd [29], with the techniques involved in Allen and Kennedy's algorithm (Allen-Kennedy for short) and Wolf and Lam's algorithm (Wolf-Lam for short). Our algorithm is in the "approximated class" as it takes as input an approximation of the dependences, called dependence polyhedra. Dependence polyhedra are a generalization of direction vectors: roughly speaking, dependence polyhedra are sort of direction vectors whose "directions" are not necessarily parallel to the axis of the basis. Thus, our algorithm can also work with classical direction vectors. In fact, each time we write "dependence polyhedra", the reader can read "direction vectors". We present here the more general version of our algorithm, the one which works on polyhedral reduced dependence graphs, as the generalization induces no overhead.

Our algorithm has the following properties:

- It does not require an exact dependence analysis. It is optimal [1] for dependence graphs 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-Kennedy when dependences are expressed by dependence levels. Furthermore, when dependences are expressed by direction vectors, it generalizes Wolf-Lam to the case of loop nests with multiple statements (Wolf-Lam 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 property allows to better understand the link between the maximal degree of parallelism that can be detected and the accuracy of dependence abstractions. See [15] for a complete study of this topic. This property also enables to use efficient techniques to remove disturbing false dependences (see [6]).

- 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 is, in practice, a way to exploit medium-grain parallelism (for a complete study see [13]).

- It produces schedules as regular as possible in order to generate codes as simple as possible. Indeed, our algorithm rewrites the codes using affine schedules, but, unlike Feautrier's algorithm, these affine schedules are chosen such as as many statements as possible have the same linear part: the code generation can then be viewed as a sequence of partial unimodular transformations and loop distributions. As a result, our output codes are guaranteed to be simpler than Feautrier's codes. Moreover, the complexity of the algorithm itself is lower as demonstrated by the comparison study independently done at the university of Passau [35].

- It is based on the resolutions of linear programs that are similar to those solved in Feautrier's algorithm. The only (through fundamental) difference is that we look for less general affine transformations. Despite this difference, our optimality result gives some hints concerning the optimality cases of Feautrier's algorithm that was first presented as a "greedy heuristic".

The rest of the paper is organized as follows. In Section 2, after a brief survey of existing parallelizing algorithms, we motivate our work by showing very simple (and real) code fragments for which neither Allen and Kennedy's algorithm, nor Wolf and Lam's algorithm, is able to detect some parallelism. In Section 3, we illustrate the use of a polyhedral approximation of the dependences. Then, we formally define polyhedral reduced dependence graphs, and we demonstrate the expressive power of this dependence abstraction. In

[1] All the mentioned optimality results are true *with respect to the dependence analysis*. This means that the algorithm is able to find all the parallelism contained in the representation of the dependences it takes as input.

Section 4, we give an overview of the different steps of the parallelization algorithm, for perfectly nested loops. We illustrate its capabilities on an ad-hoc example. The proofs of correctness and optimality of the algorithm are detailed in the next two sections, which form the heart of the paper. Section 5 is devoted to the problem of computability of the dependence graph and the study of its properties, Section 6 addresses the scheduling problem and the efficiency of our solution. These two sections are fundamental to completely explain why the algorithm actually works, and to make the paper self-contained. However, they can be skipped at first reading. In Section 7, we discuss some implementation strategies that enables to reduce the complexity of the algorithm. We run our algorithm on a few examples to illustrate its power compared to existing algorithms. Finally, we briefly show how it can be extended to non-perfect loop nests, even though some work remains to be done in this area. We conclude in Section 8.

2 Motivation

We are interested in compilers which automatically translate sequential programs into parallel programs. We especially look at algorithms that parallelize loop nests. These algorithms follow two different approaches. In the first approach, called "exact", the algorithms work with an exact representation of the dependences. In the second approach, called "approximated", the algorithms work with a conservative approximation of these dependences. We first survey these two approaches, exhibiting their advantages and drawbacks.

2.1 The exact approach

In this first approach, the parallelization algorithms take as input an exact representation of the dependences. The main algorithms are those of Lamport, of Darte and Robert, of Feautrier, and of Lim and Lam:

Lamport [33] The hyperplane method proposed by Lamport [33] applies to a set of perfectly nested loops whose dependences are all uniform (a perfect uniform loop nest). Such a loop nest with d loops always contains $d-1$ degrees of parallelism. The parallelized code contains one outer sequential loop and $d-1$ inner parallel loops.

Darte and Robert [11, 12] Darte and Robert look for an affine schedule for each statement in the loop nest. All dependences need to be uniform, and a quite large linear system (obtained by the duality theorem of linear programming) has to be solved. The schedule is selected among all possible affine schedules. This algorithm is optimal in terms of parallelism extraction: the latency of the selected schedule and the latency of the best existing schedule are guaranteed to be asymptotically equivalent [10, 9] for strongly connected dependence graphs. The technique can be extended to affine dependences (i.e. dependences expressed as affine functions of loop counters) but the optimality is not guaranteed any longer.

Feautrier [21, 22] 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. However, Feautrier's algorithm is more general since it is able to derive multi-dimensional affine schedules when no one-dimensional schedule exists. So far, Feautrier's algorithm is indeed the most powerful algorithm for inner parallelism detection in nested loops, even if, until now, it can be applied only to static control programs (see [20] for a definition). There are some attempts to generalize this approach by weakening the constraints on the input, using a Fuzzy Array Dataflow Analysis [7]. The only known results [10, 9] concerning the efficiency of this algorithm are the optimality cases of Darte and Robert's algorithm. However, there are no known results for the multi-dimensional case. Actually, what we call "Feautrier's algorithm" is named "greedy heuristic" by its author [22].

Lim and Lam [34] Lim and Lam's algorithm is a very recent extension of Feautrier's algorithm which finds fully permutable loops and external parallel loops (if any). The technique involves the affine form of Farkas' lemma and a variation of Fourier-Motzkin elimination. This algorithm is also devoted to programs where the array accesses are affine functions, but unlike the three algorithms above, the goal is not to look for the transformation which finds the maximal degree of parallelism, but to look

for the affine transformation, if any, which finds a given degree of parallelism while minimizing the degree of synchronization. Their algorithm is said optimal under some hypotheses (but optimality has a different meaning).

We call "exact" the approach used in the four algorithms above because they all rely on an exact dependence analysis and an exact representation of the dependences. The main advantage of this approach is that the loss of parallelism is never due to the computation of the dependences. However, some parallelism can be lost because of the parallelization algorithms themselves, and therefore their efficiency must be considered. Darte and Robert's algorithm, and in some cases Feautrier's algorithm, are optimal. But they are only optimal for the classes of programs they are able to process! All these algorithms require that exact dependence analysis is feasible and that dependences are affine. This implies very strong hypotheses on the original codes. Only programs with static control flow and affine array accesses can be processed. And not all algorithms given above will process all of them. The limitation on the algorithms' inputs is the main drawback of this approach.

The exact approach is sometimes said to have another disadvantage: its cost. Indeed, the exact computation of the dependences is often said to be expensive. This can be discussed. What is sure however is that the execution time of Feautrier's algorithm, for example, is long as it requires the resolution of large linear programs (see for example [35] for timing reports). Finally and, in our view, more important, the code generated by Feautrier's algorithm can be very complicated. Feautrier builds one multi-dimensional affine schedule per statement in the loop nest. This enables him to find schedules with theoretically efficient latencies ... but the price to pay is the complexity of the code that may induce a loss of efficiency at run-time.

In conclusion, with the "exact" approach, no parallelism is lost because of the dependences representation. But there are strong requirements on the program to be parallelized. The efficiency of these parallelization algorithms is not known in the most general cases. Furthermore, most of them are computation expensive and the generated codes are complicated. Thus, the dream is to find a parallelization algorithm which handles more general programs and produces quickly simpler outputs, while having sufficient guarantees on its efficiency.

2.2 The approximated approach

In this approach, the parallelization algorithms take as input a conservative approximation of the dependences. The two main algorithms in this category are those of Allen and Kennedy, and of Wolf and Lam:

Allen and Kennedy [2] The basic technique of this algorithm is loop distribution which is done by an analysis of the strongly connected components of the reduced dependence graph (RDG). Dependences are represented by levels, which is a very rough representation, although often sufficient on real codes. Allen and Kennedy's algorithm is known to be optimal [18] with respect to its input. This means that a parallelization algorithm, which takes as input a reduced dependence graph where the dependences are represented by levels, cannot find more parallelism than Allen and Kennedy's algorithm does.

Wolf and Lam [39] Their algorithm is based on a cone separation technique, adapted to the case of direction vectors, and used to detect fully permutable loops. A set of d fully permutable loops can always be rewritten as one sequential and $d-1$ parallel loops. The transformations used in Wolf-Lam are unimodular transformations. The representation of the dependences, by direction vectors, is sharper than in Allen-Kennedy. However, the graph structure of the dependences is not taken into account. Wolf-Lam is optimal [39] among unimodular transformations which take direction vectors as input.

Unlike the "exact" approach, the "approximated" approach is able to process more general loop nests. This is its main advantage. For example, the parallelization algorithms in this category are able to handle codes containing array accesses with indirections: $A[B[I]]$. Another advantage is that they involve simpler techniques than the algorithms in the "exact" approach. Consequently, they are easier to implement, they are quicker and they generate simpler outputs. Furthermore, they have been proved optimal with respect to the representation of the dependences they used (see [18, 16]).

The problem of this approach is the cost, in terms of loss of parallelism, of an approximation of the dependences. Indeed, the dependences are conservatively approximated (over-approximated): the chosen

dependences abstraction may represent more dependences than there are in the code. The additional dependences induce a greater apparent sequentiality of the original code, and this may cause a loss of parallelism. The problem is to determine whether this loss of parallelism is affordable. The fundamental question is: if the parallelization algorithm I used finds no parallelism in the loop nest, would another technique find some? Right now, there is no complete answer to this general question. However, a comparison of Allen-Kennedy and Wolf-Lam gives some hints.

2.3 Allen-Kennedy *vs* Wolf-Lam

None of these two algorithms subsumes the other and both are sometimes unable to find parallelism in cases where there is obviously some. We illustrate these points with three real fragments of code, borrowed from [23] and [19]. We also explain why combining loop distribution with the techniques involved in Wolf-Lam is not straight-forward.

Example 1 (where Wolf-Lam beats Allen-Kennedy) Example 1, depicted in Figure 1, is the Gauss-Seidel iteration program [23]. There is in the dependence graph, Figure 2, a dependence cycle at depth 1 and at depth 2. Thus Allen-Kennedy finds no parallelism in this code. However, the dependences are uniform equal to $(1,0)$ and $(0,1)$. Thus, Wolf-Lam finds that the two loops are fully permutable and that this loop nest contains some parallelism. This is the typical example where an algorithm such as Allen-Kennedy fails.

```
DO j = 1,m
  DO i = 1,n
    g(i, j) = (f(i,j)+g(i-1,j)+g(i+1,j)+g(i,j-1)+g(i,j+1))/4
  ENDDO
ENDDO
```

Figure 1: Ex. 1, Gauss-Seidel.

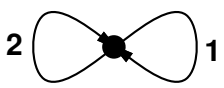

Figure 2: RDG, Ex. 1.

Example 2 (where Allen-Kennedy beats Wolf-Lam) Example 2, depicted in Figure 3, is part of a linear equation solver based on matrix-vector operations, written by Dongarra and Eisenstat [19]. Wolf-Lam algorithm, as described in [39], is only able to process perfectly nested loop nests. Thus, we first transform the code into a perfect one, before applying Wolf-Lam. There are many different solutions to this "perfectization" problem. One of them is given on Figure 4.

```
DO k = 1, n
  S1 : xk = x(k) × a(k, k)
  DO i = k + 1, n
    S2 : x(i) = x(i) − a(i, k) × xk
  ENDDO
  S3 : x(k) = xk
ENDDO
```

Figure 3: Ex. 2, Dongarra-Eisenstat.

```
DO k = 1, n
  DO i = k, n
    S'1 : if (i == k)      then xk = x(k) × a(k, k) endif
    S'2 : if (i >= k + 1)  then x(i) = x(i) − a(i, k) × xk endif
    S'3 : if (i == n)      then x(k) = xk endif
  ENDDO
ENDDO
```

Figure 4: A perfectly nested version of Ex. 2.

The corresponding reduced dependence graph, computed by Petit [30] (but in which redundant direction vectors have been removed), is drawn on Figure 5. Because of the direction vectors of weight $(1,0-)$ and $(0,+)$, among others, Wolf-Lam is not able to detect any parallelism. However, there is clearly no cycle of level 2 in the dependence graph, thus, with a simple loop distribution at depth 2, Allen-Kennedy finds

that the second loop can be made parallel. Actually, Allen-Kennedy is able to retrieve the original non-perfect version, with a parallel i loop: running Allen-Kennedy on the original code would lead to the same parallelized code.

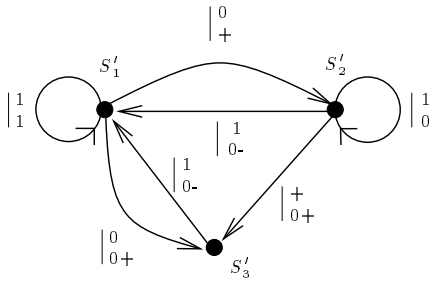

Figure 5: Dependence graph of Dongarra and Eisenstat's perfectly nested version (Ex. 2).

In this example, Wolf-Lam can be easily combined with Allen-Kennedy as follows. Instead of completely ignoring the structure of the dependence graph, we can apply a loop distribution before each recursive call to Wolf-Lam. We can look for the first set of fully permutable loops, then apply loop distribution, then call Wolf-Lam again on the remaining dependences, etc. On this example, such a strategy enables us to find some parallelism when the original Wolf-Lam algorithm fails. However, there are examples where this simple modification is not sufficient to detect maximal parallelism (see Example 6 in Section 7.2).

Example 3 (where Wolf-Lam needs a perfect loop nest) For parallelizing Example 2, we first transformed the original code into a perfect code. One could argue that Wolf-Lam was not able to detect parallelism because of this transformation, and that Wolf-Lam should be apply on non-perfectly nested loops "set of perfectly nested loops" by "set of perfectly nested loops". Indeed, on Example 2, this modified version of Wolf-Lam finds some parallelism. The first set of loops only includes the k loop, which is detected as sequential. Then, the second set of loops only includes the i loop and the statement S_2. The i loop contains no dependences and then is found parallel. Thus, in this case, it was *not* a good idea to transform the original non-perfect loop nest into a perfect one, before applying Wolf-Lam. However, Example 3 below shows that this is sometimes a good idea.

```
DO k = n − 1, 0 step −1
  DO i = k + 1, n
    b(i) = b(i)/(x(i) − x(i − k − 1))
  ENDDO
  DO i = k, n − 1
    b(i) = b(i) − b(i + 1)
  ENDDO
ENDDO
```

Figure 6: Ex. 3, Vandermonde kernel.

```
DO k = n − 1, 0 step −1
  DO i = k + 1, n
    b(i) = b(i)/(x(i) − x(i − k − 1))
    b(i − 1) = b(i − 1) − b(i)
  ENDDO
ENDDO
```

Figure 7: A perfectly nested version of Ex 3..

Example 3 (see Figure 6) is part of an algorithm to solve the Vandermonde system: $Vz = b$ [23]. Let us try to apply Wolf-Lam "set of perfectly nested loops" by "set of perfectly nested loops". The first set of perfectly nested loops only includes the k loop. This loop is obviously sequential as $b(i)$ is written at several different k iterations. For the second level, we have two sets of perfectly nested loops to look at. The first i loop is parallel, as it contains no dependences. The second i loop is sequential as the new value of $b(i)$ depends upon the old values of $b(i+1)$. Thus the program parallelized this way still contains two sequential nested loops as in the original program. Allen-Kennedy finds the same parallelized code. However, consider a perfectly nested version of this code as shown in Figure 7. The direction vectors are $(+,0)$, $(0,1)$,

$(1,0)$, $(+,1)$, $(+,-1)$, and $(1,-1)$. They are of course lexicographically positive. Furthermore their second component is always constant. Consequently, Wolf-Lam is able to find two permutable loops and, thus, one degree of parallelism. Therefore, on this example, transforming the code into perfectly nested loops and applying Wolf-Lam enables us to find more parallelism.

Conclusion These three examples show that sometimes Wolf-Lam algorithm finds some parallelism when Allen-Kennedy only finds sequentiality, and vice versa. These examples also demonstrate that the basic Wolf-Lam algorithm needs to be upgraded to handle loop distribution. We proposed two new versions of Wolf-Lam in which we tried to combine loop distribution techniques as in Allen-Kennedy. We have now four different algorithms and no one subsumes the others. This confusing situation illustrates that combining unimodular transformations and loop distribution is not straightforward.

2.4 Our goal

Schematically, we want an algorithm which works with an approximation of the dependences as we need to be able to process general loop nests; we want to use techniques powerful enough to extract all the parallelism described by the chosen representation of the dependences; we want to generate codes as "simple" as possible. We would like this algorithm to be able to detect some parallelism at least when Allen-Kennedy or Wolf-Lam does. Of course, such an algorithm exists: the algorithm which tries Allen-Kennedy and the different versions of Wolf-Lam and takes the "best" answer! The way this algorithm is built shows its main weakness: it uses either the dependence graph structure (Allen-Kennedy) or direction vectors (Wolf-Lam), but never knowledge of both types at the same time. For example, the simple Wolf-Lam extensions that we proposed above use the dependence graph structure before or after the computation of a maximal set of fully permutable loops, but never during this computation. The aim of Wolf and Lam [39] was to "combine the mathematical rigor in the matrix transformation model with the generality of the vectorizing and concurrentizing compiler approach". We have the same goal, except that we would also like to exploit the structure of the dependence graph.

There is a large gap between the complexity of the algorithms in the "approximated" and "exact" approaches, both in terms of dependence abstraction and in terms of execution time. One of our goals is to fill this gap and to propose an intermediate algorithm, thus of medium complexity, but still optimal for all classical approximations of dependences. Moreover, we would like to generate "simple" codes, not to loose at running time what we theoretically gained using a more powerful parallelization algorithm. Thus, we want to use transformation techniques intermediate between those used by Allen-Kennedy and Wolf-Lam, which generate very simple codes, and those used by Feautrier's algorithm, which are far more expressive but can generate complicated codes.

3 Polyhedral approximations of dependences

3.1 A motivating example

We first introduce the notion of polyhedral reduced dependence graph (PRDG) on an ad-hoc example. PRDGs will be formally defined in Section 3.2. We show on this example that no parallelism can be detected if dependences are approximated by levels or direction vectors, but that an exact representation of dependences is not needed: an approximation of distance vectors by polyhedra is sufficient. We illustrate the underlying idea of our parallelizing algorithm, the "uniformization" principle, that enables us to capture PRDGs as if they were uniform dependence graphs.

Example 4 (use of polyhedral approximations) Example 4 is given in Figure 8. The exact dependences are listed in Figure 9. Figure 10 shows the corresponding (reduced) dependence graphs (RDGs) when dependence edges are labeled respectively (a) with levels and (b) direction vectors. Let us try to parallelize Example 4 with the algorithms described in Section 2: the algorithms of Allen and Kennedy, of Wolf and Lam, of Darte and Robert, and of Feautrier.

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

Figure 8: Source code.

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

Figure 9: Exact dependence relations.

Allen-Kennedy 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-Lam The dependence vectors are respectively $(0,1)$, $(+,-)$, and $(+,-)$. In the second dimension, the "1" and the "$-$" prevent to detect two fully permutable loops. Therefore, the code remains unchanged, and no parallelism is detected.

Darte-Robert One level of parallelism is detected with the valid linear schedule $T(i,j) = 2i + j$.

Feautrier This algorithm finds the same schedule than Darte-Robert (up to a constant): $T(i,j) = 2i+j-3$. One level of parallelism is detected.

In this particular example, the representation of the dependences by levels or by direction vectors is not accurate enough to reveal parallelism. This is the reason why Allen-Kennedy and Wolf-Lam are not able to detect any parallelism. Exact dependence analysis, associated to linear programming methods that require to solve large[2] parametric linear programs, enables to reveal one degree of parallelism. The corresponding parallelized code is give in Figure 11.

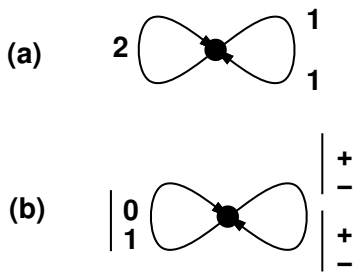

Figure 10: RDG for Ex. 4.

```
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
```

Figure 11: Example 4, parallelized version.

However, in Example 4, an exact representation of the dependences is not necessary to reveal some parallelism. Indeed, one can notice that there is one uniform dependence $u = (0,1)$ and a set of distance vectors $\{(j-i, i-j) = (j-i)(1,-1) \mid 1 \le j-i \le n-1\}$ that can be (over)-approximated by the set $P = \{(1,-1) + \lambda(1,-1) \mid \lambda \ge 0\}$. P is a polyhedron with one vertex $v = (1,-1)$ and one ray $r = (1,-1)$. Now, suppose that, as in the "exact" algorithms, we are looking for a linear schedule $T(i,j) = x_1 i + x_2 j$. Let $X = (x_1, x_2)$. For T to be a valid schedule, we look for X such that $Xd \ge 1$ for any dependence vector d. Thus, $X(0,1) \ge 1$ and $Xp \ge 1$ for all $p \in P$. The latter inequality is equal to: $X(1,-1) + \lambda X(1,-1) \ge 1$ with $\lambda \ge 0$, which is equivalent to: $X(1,-1) \ge 1$ and $X(1,-1) \ge 0$, i.e. $Xv \ge 1$ and $Xr \ge 0$. Therefore, one has just to solve the three following inequalities:

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

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

which leads, as above, to $X = (2, 1)$. Thus, for this example, an approximation of the dependences by levels or even direction vectors is not sufficient for the detection of parallelism. However, with an approximation of the dependences by polyhedra, we find the same parallelism as with exact dependence analysis, by solving a simpler set of inequalities.

What is important here is the "uniformization" which enables us to go from the inequality on the set P to uniform inequalities on v and r. Thanks to this uniformization, the affine constraints disappear and unlike Feautrier's algorithm, we do not need to use Farkas' lemma. To better understand the "uniformization" principle, think in terms of dependence path. The idea is to consider an edge e, from statement S to statement T, 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. This simulation is summarized in Figure 12: we introduce a new node S' that enables to simulate ϕ and a zero weight edge to go from S' back to the initial node T. This "uniformization" principle is the underlying idea of the loop parallelization algorithm proposed in this paper.

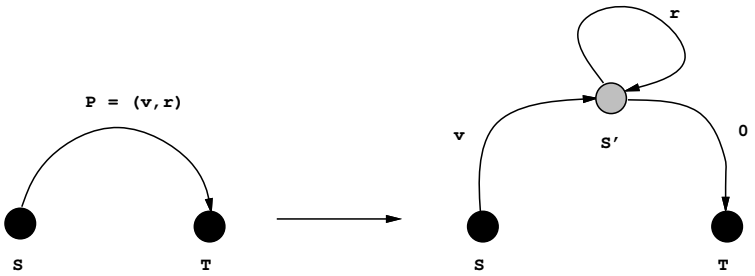

Figure 12: Simulation of an edge labeled by a polyhedron with one vertex and one ray.

By uniformizing the dependences, we have in fact "uniformized" the constraints and transformed the underlying affine scheduling problem into a simple scheduling problem where all dependences are uniform (0, v, and r in Figure 12). However, there are two fundamental differences between this framework and the classical framework of uniform loop nests:

- The uniform dependence vectors are not necessarily lexico-positive (for example, a ray can be equal to $(0, -1)$). Therefore, the scheduling problem is more difficult. However, it can be solved by techniques similar to those used to solve the problem of computability of systems of uniform recurrence equations [29, 16].
- The constraint imposed on a ray r is weaker than the classical constraint: the constraint is indeed $Xr \geq 0$ instead of $Xr \geq 1$. This freedom must be taken into account in the parallelization algorithm.

3.2 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 7.3. This restriction allows us 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 (called the **iteration domain**) defined by the loop bounds. 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 3.2.1 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 3.2.2, 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 3.2.3, we show how the model of PRDG generalizes classical dependence abstractions of distance sets.

3.2.1 Dependence graphs and distance sets

Dependence relations between operations are defined by Bernstein's conditions [5]. 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**.[3] 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 (see [20]), it is possible to express exactly this set of pairs (I, J): in this case, 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. 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 [1, 2] for Allen and Kennedy's parallelizing algorithm.
- **direction vector**, introduced by Wolfe in [40, 41] and used in Wolf and Lam's algorithm [39].
- **dependence polyhedron** (and dependence cones), introduced in [26] and used in Irigoin and Triolet's supernode partitioning algorithm [28]. We refer to the PIPS [25] software for more details on dependence polyhedra. Dependence cones are particular cases of dependence polyhedra.

We first define formally reduced dependence graphs whose edges are labeled by dependence polyhedra and we show the expressive power of this model.

[3] In some cases, output and anti dependences can be removed by data renaming and data expansion. See for example [20].

3.2.2 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.*

A polyhedron can always be decomposed as the sum of a (convex) polytope and of a polyhedral cone (for more details see [38]). 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:

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

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 distances are always integral vectors.

Definition 2 *A **polyhedral reduced dependence graph (PRDG)** is a RDG, for which each edge $e : S_i \rightarrow S_j$ 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 $(J - I) \in P(e)$.*

The notion of dependence polyhedron is very close to the notion of *dependence convex hull* [27]. Indeed, a dependence convex hull is a particular dependence polyhedron: the convex hull of the distance vectors. Here, we make no assumptions on the dependence polyhedra we handle or on the way they were computed.

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**.

3.2.3 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.

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 zero except the i-th component which is 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 the vertex $(2, 0, -1, 3)$, the two rays $(1, 0, 0, 0)$ and $(0, 0, -1, 0)$, and the line $(0, 1, 0, 0)$.

Levels of dependence The representation by levels is the less accurate (but still useful on real applications) 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 $(0, \ldots, 0, 1+, *, \ldots, *)$ (with $l-1$ leading zeros) and an edge of level ∞ corresponds to the zero dependence vector. As any direction vector admits an equivalent polyhedron, so does a representation by levels. 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 $(0, 0, 1)$.

4 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 5 and 6. An optimized version of the algorithm is presented in Section 7. Our parallelization algorithm consists of two main steps, a "uniformization" step presented in Section 4.1 and a "scheduling" step summarized in Section 4.2. We illustrate both steps with synthetic Example 5 below.

Example 5 The code is given in Figure 13. We assume that the edges of the reduced dependence graph (see Figure 14) are labeled by direction vectors. This graph was built by the dependence analyzer Tiny [42].

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 Figure 15 (but in which no effort, such as loop peeling, has been made so as to remove "if" tests). Thus, for each statement, one level of parallelism can be detected.

This code has been generated with the help of the procedure "codegen" of the Omega Calculator delivered with Petit [30]. The Omega Calculator, developed by Bill Pugh's team, is a framework to compute dependences, to check the validity of program transformations, and to transform programs, once the transformation is given. It does not propose the transformations to be applied. We plan to integrate our algorithm in Petit as a transformations generator. We point out that the code proposed above is a "virtual" code in the sense that it only reveals hidden parallelism. We do not claim that it must be executed as such.

```
DO i = 1, n
  DO j = 1, n
    DO k=1, j
      S1: a(i, j, k) = c(i, j, k - 1) + 1
      S2: b(i, j, k) = a(i - 1, j + i, k) + b(i, j - 1, k)
      S3: c(i, j, k + 1) = c(i, j, k) + b(i, j - 1, k + i)
            + a(i, j - k, k + 1)
    ENDDO
  ENDDO
ENDDO
```

Figure 13: Code for Example 5.

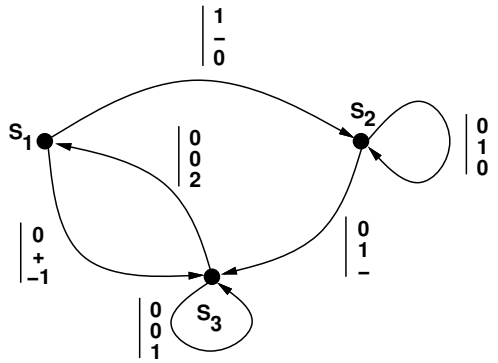

Figure 14: RDG with direction vectors, Ex. 5.

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

Figure 15: Example 5, parallelized version.

4.1 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 and 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 3.2.2: 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 $e : x_e \rightarrow y_e \in E$ 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 zero weight edge directed from n_e to y_e.

Back to Example 5 The PRDG of Example 5 was depicted in Figure 14. Figure 16 shows the uniform dependence graph associated to it. It has three new nodes (i.e. virtual nodes) that correspond to the symbol "+" and the two symbols "−" in the initial direction vectors.

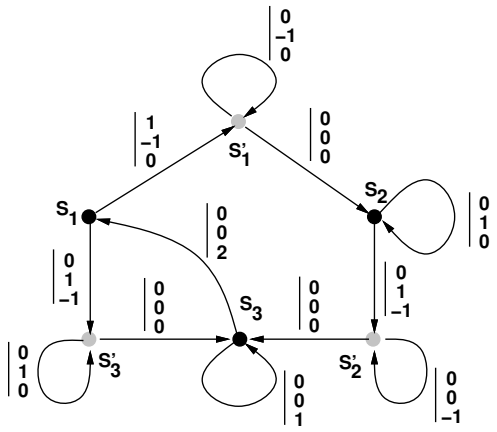

Figure 16: Translated uniform reduced dependence graph for Ex. 5.

4.2 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 is called for each strongly connected component of G_u).

This is a recursive algorithm. Each step of the recursion builds a particular subgraph G' of the current graph G being processed. 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). **Intuitively, the principle of the algorithm is to define a new order of computations (i.e. a loop transformation) such that as many dependence vectors as possible are carried by the outermost loop after transformation, then as many remaining dependence vectors as possible are carried by the second outermost loop, and so on.**

G' is defined as the subgraph of G generated by all the edges of G that belong to at least one multi-cycle of zero weight. A multi-cycle is a union of cycles, not necessarily connected, and the weight of a union of cycles is the sum of the weights of its constitutive cycles. G' is built by the resolution of a linear program (see Section 6.1 for details).

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

DARTE-VIVIEN(G, k) /* builds the k-th component of the schedule */

i. Build G' the subgraph of G generated by all edges that belong to at least one zero 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:

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

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 6). In this case, the two steps, Step (i) and Step (iii), can be solved simultaneously by the resolution of a single linear program.
- 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 7 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 [13]).

Back to Example 5 Consider the uniform dependence graph of Figure 16. 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 zero weight. The subgraph G' is depicted in Figure 17.

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 $\rho_{S_2} = 0$, for example if we minimize x.

In G', there remain four strongly connected components, and two of them are not considered since they only have virtual nodes. The two other components have no zero weight multi-cycles. 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 the beginning of Section 4, 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 [16].

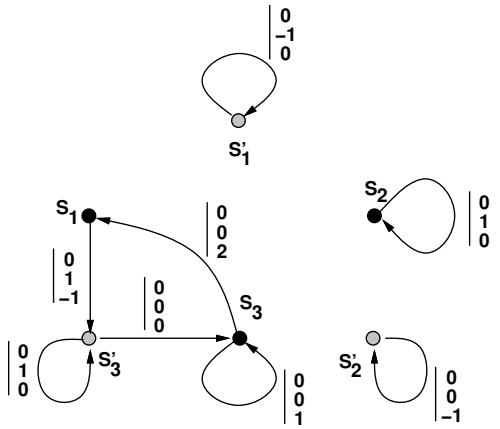

Figure 17: Subgraph of zero weight multi-cycles for Example 5.

5 Computability of a PRDG

In this section, we study the theoretical properties of PRDGs, in terms of computability, with the same spirit as in Karp, Miller and Winograd's paper [29]. Section 6 will show how computable PRDGs can be scheduled.

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 18(a) is computable (even if the direction vector $(0, *)$ could be refined in the case of nested loops) whereas the PRDG of Figure 18(b) is not computable, i.e. can not be scheduled. This comes from

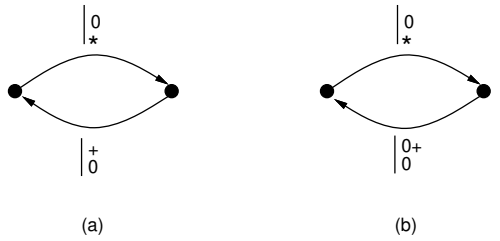

Figure 18: Two different PRDGs: (a) computable, (b) not computable.

the fact that there is a dependence cycle of zero 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 5.1, we explore the links between G_o and its translated G_u. Following the results of this study, we are able 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 5.2 is devoted to the computability problem. Section 5.3 focuses on the length of the longest paths described by G_u. We give lower bounds on the length of these paths. These bounds are really important since lower bounds on the length of dependence paths are upper bounds on the parallelism in the dependence graphs (see [18] for a sharp definition of degree of parallelism).

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.

5.1 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.

5.1.1 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. Furthermore, 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, given by the following lemma and corollary:

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

Proof: The proof is straightforward, by construction of G_u. A detailed proof is given in [17]. □

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.*

5.1.2 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 and/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: The proof is not difficult. One has just to decompose each polyhedral edge on the vertices, rays and lines of the corresponding polyhedron. If some components are not integral, we multiply (i.e. we use several times the cycle) all the components by a suitably large integer so that they all become integral. Details can be found in [17]. □

5.1.3 Computability condition

In the case of bounded iteration domains, a reduced dependence graph is computable if and only if the apparent dependence graph 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 zero 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 zero weight cycle with at least one actual node.*

5.2 Computability of the translated graph G_u

In the previous section, we moved the computability problem from G_o to G_u. Now, checking the existence of zero 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 [29] and [16]), with a slight modification so as to take into account the fact that some nodes in G_u 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 [29, 36, 37, 32, 16], the first one being Karp, Miller, and Winograd's decomposition [29]. However, we suggest the reader to refer to the algorithm formulation and proofs proposed in [16] 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 zero weight cycles in a uniform reduced dependence graph G. It is a recursive algorithm, based on the search for zero weight multi-cycles (i.e. union of cycles, not necessarily connected). However, in our case (cf Theorem 1), we are not interested in all zero weight cycles, but only in zero 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 is a direct consequence of the correctness of Karp, Miller, and Winograd's decomposition.

5.2.1 Decomposition algorithm: DA

$\bigwedge$ denotes the logical AND.

Boolean DA(G)

i. Build G' the subgraph of G generated by all the edges that belong to a zero 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 $\bigwedge_{i=1}^{s} \mathrm{DA}(G'_i)$.

Then, G_u has no cycle of zero weight containing an actual node if and only if $\mathrm{DA}(G_u) = \mathrm{TRUE}$. Therefore, a PRDG G_o is computable if and only if $\mathrm{DA}(G_u) = \mathrm{TRUE}$.

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

For systems of uniform recurrence equations, the depth of the decomposition algorithm is a measure of the degree of parallelism described by its dependence graph [16]. We will see that this result still holds for PRDGs and the depth d defined for algorithm DA.

5.2.2 Dependence polyhedra considered to be atomic: algorithm DA*

For technical reasons that will become clear in the scheduling section (Section 6), we need to slightly modify the decomposition algorithm DA.

Remember how G_u was built (refer to the translation algorithm in Section 4.1). 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 zero 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 $\bigwedge_{i=1}^{s} \text{DA*}(G'_i)$.

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 zero weight multi-cycles is also unchanged since we add only edges that do not belong to a zero weight multi-cycle. Therefore, both algorithms DA* and DA have exactly the same behavior (in terms of nodes).

5.3 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 [10], 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 [9], 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 [14]. 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 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. □

6 Scheduling a PRDG

In this section, we detail the scheduling algorithm for PRDGs introduced in Section 4.2. In particular, we show how to find the vector X and the constants ρ_S. We also prove that our algorithm is able to find all the parallelism contained in a PRDG. In Section 6.1, we recall how G' can be computed and the link with "shifted-linear" schedules. In Section 6.2, we show that the scheduling algorithm is correct, i.e. that it produces a schedule that respect all dependences. Finally, in Section 6.3, we show the optimality of the scheduling algorithm.

6.1 Construction of G' and related properties

In this section, we recall how to build G', the subgraph of zero weight multi-cycles. We also recall why it is closely related to the construction of separating hyperplanes and, 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 the resolution of a single linear program. Indeed, it has been shown (see [16]) 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 [13]). Edges of G' are characterized by the following lemma:

Lemma 3

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

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 enables to build a "shifted linear" schedule.

6.2 Correctness of the scheduling algorithm

We are now ready to prove that the scheduling algorithm proposed in Section 4.2 is correct.

Note first that the skeleton of Algorithm DARTE-VIVIEN is nothing but Algorithm DA*, whose correctness has already been studied in Section 5.2.2. Therefore, it remains to prove two things:

- that Step (iii) is feasible, i.e. that it is possible to define a vector X and some constants ρ_S satisfying to the desired inequalities.
- that, as claimed in Section 4.2, the sequence of vectors $X_S^1, \ldots, X_S^{d_S}$ and the sequence of constants $\rho_S^1, \ldots, \rho_S^{d_S}$ 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:

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

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:*

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

defines a valid multi-dimensional schedule for G_o.

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 3.2). Actually, we prove a bit stronger property: there exists an integer $k \in [1 \ldots d]$ such that

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

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

Remember that all possible values of $w(e)$ are given by Equation 2: $w(e) = \sum_{j=1}^{j=\omega} \mu_j v_j + \sum_{j=1}^{j=\rho} \nu_j r_j + \sum_{j=1}^{j=\lambda} \xi_j l_j$ with $\mu_i \in \mathbb{Q}^+$, $\nu_i \in \mathbb{Q}^+$, $\xi_i \in \mathbb{Q}$ and $\sum_{j=1}^{j=\omega} \mu_j = 1$. We built G_u the following way. For each edge $e = (x_e, y_e)$, we introduced a new node n_e. 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 zero 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 zero 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:

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

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

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

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 enables 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 [38] – 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 enables to relax constraints for the next levels. This is the case for example for direction vectors. The decomposition proposed in Section 3.2.3 is indeed a decomposition on a Hilbert basis.

6.3 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 are 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. Therefore, 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 the iteration domain contains (resp. is contained in) a full dimensional cube of size $\Omega(N)$ (resp. $O(N)$), and if d is the depth of the decomposition algorithm, 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.*

7 Implementation, examples, and extensions

7.1 Implementation strategies

The most time consuming steps of the scheduling algorithm presented in Section 4.2 are the resolutions of the different linear programs of Steps (i) and (iii). In order to speed up the algorithm, we 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 the resolution of a linear program.

This remark would enable 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 may not be always reasonable. To avoid this problem, we use a **basis of cycles**, instead of considering all cycles. This technique enables to reduce both the number of constraints and the number of variables in the linear program of Step (iii). We only give below an idea of this technique. Details can be found in [17].

Definition 4 *A basis of cycles $\mathcal{BC}$ of G is a family of independent cycles* [4] $C_1, \ldots, C_m$ *such that for each cycle C in G, there exists some rationals $\lambda_1, \ldots, \lambda_m$ such that: $C = \sum_{i=1}^{m} \lambda_i C_i$.*

We refer to [24] for a more precise definition of basis of cycles and cyclomatic number.

7.1.1 Computation of the vector X

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 constraints to solve are:

$$\{ \;\; x \geq 0, \quad X\mathcal{W} = (x + \delta)Q \;\; \} \tag{8}$$

Actually, the system above 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:

$$\min \{ \;\; \sum_{j=1}^{n} Y_j \quad | \quad x \geq 0, \quad X\mathcal{W} = (x + \delta)Q, \quad X \leq Y, \quad -X \leq Y \;\; \} \tag{9}$$

In the following, we assume that we have a solution (X, x) of the linear program 8.

7.1.2 Computation of the constants ρ

To compute the constants ρ, we use the Bellman-Ford algorithm [8] 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 $\mathcal{C}$ 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 satisfy the conditions of Step (iii) of the scheduling algorithm.*

[4] Actually, in such "cycles", edges can be traversed backwards. Except in this definition, all cycles used in the rest of the paper are 'true" cycles, i.e. cycles for which edges are always traversed forwards.

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 enables to guarantee that the constants ρ are simple if the vector X is simple. This property is highly desirable for code generation. Furthermore, the constraints on X expressed with a basis of cycles are useful to build the set of independent vectors needed for tiling [13].

7.2 Examples

We illustrate our algorithm on two synthetic examples so as to illustrate the different cases that, at least in theory, may occur. Example 6 can be scheduled with a single sequential loop, what Allen-Kennedy and Wolf-Lam fail to find. This example is presented with direction vectors. Example 7 is an example that cannot be parallelized successfully if dependences are approximated by direction vectors.

Example 6 (with direction vectors) Example 6 is the code of Figure 19. Its reduced dependence graph, with direction vectors, is given in Figure 20. In this example, the extension of Wolf-Lam with loop

```
DO i=1,n
  DO j=1,n
    DO k=1,n
      S1: a(i, j, k) = b(i-1, j+i, k) + b(i, j-1, k+2)
      S2: b(i, j, k) = a(i, j-1, k+j) + a(i, j, k-1)
    ENDDO
  ENDDO
ENDDO
```

Figure 19: Code of Example 6.

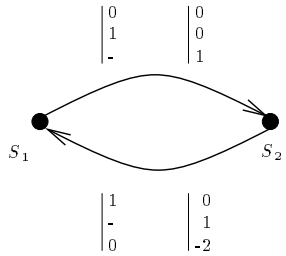

Figure 20: RDG for Example 6.

distribution proposed in Section 2 only finds one degree of parallelism, when one can find two degrees. Indeed, for Wolf-Lam, because of the two dependences with a component equal to "-", the first set of fully permutable loops only contains the i loop. Only one dependence is removed from the RDG and the graph remains strongly connected. The second set of fully permutable loops only contains the j loop. Two dependences are removed and the graph is no more strongly connected. A loop distribution can be applied. Then the third loop is found to be parallel. Thus, Wolf-Lam finds one degree of parallelism.

We now apply our algorithm. The transformed RDG is given in Figure 22. It has 4 vertices (two of them are virtual). The weights of elementary cycles are $(0, 0, -1)$ and $(0, -1, 0)$ for the self-loops and $(1, 0, -1)$, $(1, -1, 1)$, $(0, 2, -3)$, and $(0, 1, -1)$ for the other elementary cycles. Therefore, one can find a one-dimensional schedule, for example $X = (4, 0, -2)$, $\rho_1 = 0$ and $\rho_2 = 3$. Two degrees of parallelism can be exposed: the resulting code is given in Figure 21.

Example 7 (with dependence polyhedra) We now illustrate our technique with the example of Figure 23 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. Furthermore, both Allen-Kennedy and Wolf-Lam find that at least one of the statements must be fully sequentialized. With dependence polyhedra and our algorithm, the resulting S_1-latency is $O(N)$, the S_2-latency is $O(N^2)$, for a loop nest of depth 3.

The graph G_o depicted in Figure 24 has been found by the dependence analyzer Tiny [42].

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

Figure 21: Ex. 6, parallelized version.

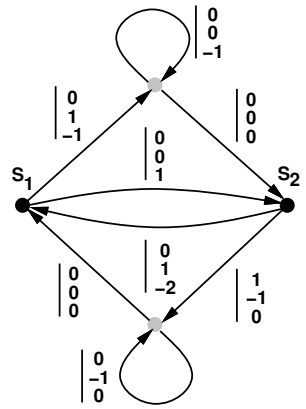

Figure 22: Transformed RDG, Ex. 6.

```
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
```

Figure 23: Code of Example 7.

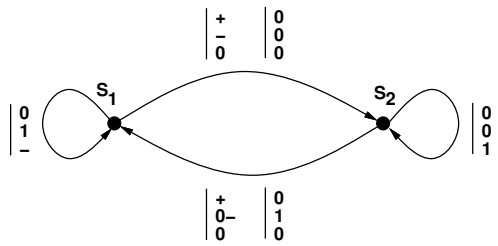

Figure 24: RDG with direction vectors for Ex. 7.

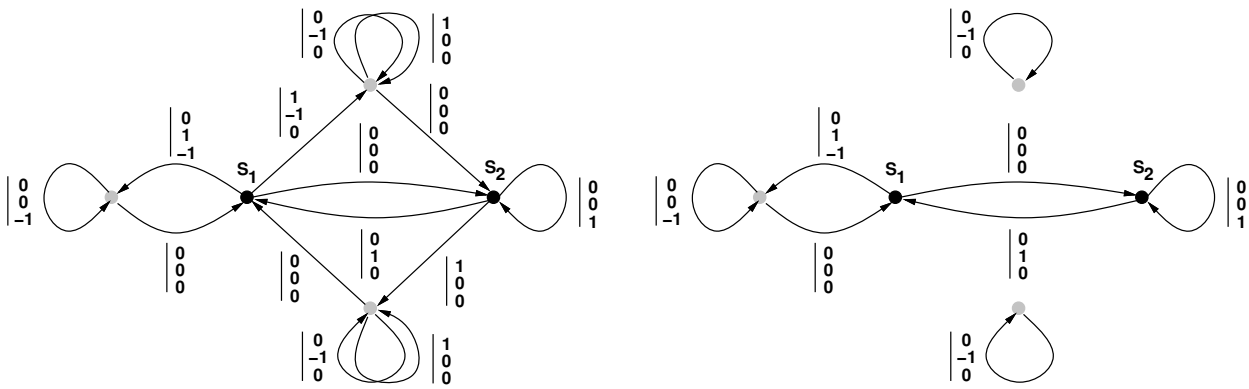

Figure 25: Translated uniform dependence graph for Example 7 and its corresponding G'.

The uniformization step transforms G_o into G_u which is depicted in Figure 25.

There is a multi-cycle of zero weight generated by all edges whose weight is orthogonal to $(1,0,0)$ (see Figure 25). In G', the strongly connected component that contains S_1 and S_2 still has a multi-cycle of zero weight that visits an actual node (S_2). S_1 is removed at depth 2 but S_2 is only removed at depth 3. Thus, 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 code given Figure 26.

```
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
```

Figure 26: Example 7, first parallelized version.

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 27. 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 than the flow dependence $(1,-1,0)$.

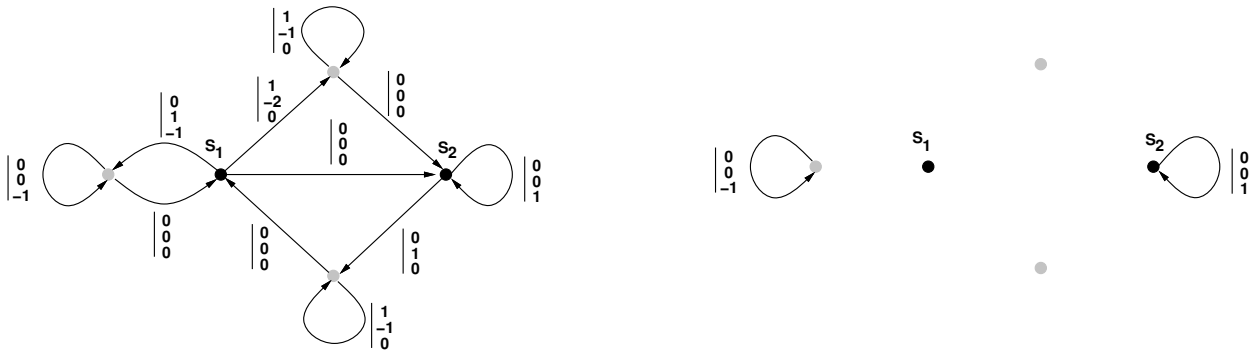

Figure 27: The PRDG G_u, for Ex. 7, and the corresponding subgraph of zero weight multi-cycles G'_u.

Note in Figure 27 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 given in Figure 28.

7.3 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

```
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 28: Example 7, second parallelized version.

dimensions. In other words, at a given depth of the algorithm, we truncate all vectors to the same dimension[5] and we apply on the truncated vectors the same technique as for perfectly nested loops. Finally, we complete each derived vector X with ending zeros so that it fits 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 enables to exploit the information on non common dimensions, and to benefit from the power of our algorithm for perfectly nested loops. As we did in Example 3, 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. Hereafter is an example, borrowed from the examples proposed in Petit [30], where the code is non-perfect at each level. Nevertheless, we point out that this technique is not general enough yet, and finding the adequate "perfectization" is not straightforward, if feasible...

Example 8 (non perfect loop nest)

```
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 Example 8, 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 the code of Figure 29.

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 code of Figure 30 (once again without any effort to remove "if" tests).

8 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.

[5] All vectors are truncated to the smallest dimension of a distance vector.

```
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 29: Perfectly nested version of Ex 8.

```
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
```

Figure 30: Ex. 8, parallelized version.

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.

Our algorithm, like Feautrier's algorithm, was implemented at the Passau University, Germany, in the LooPo project. They compared the two algorithms on four codes [35]. Both parallelization algorithms output the same result, but our algorithm was quicker, and especially on the more complex codes (at least 10 times quicker on the three more complex codes). This experimentation does not prove that "the Darte-Vivien scheduler clearly outperforms the Feautrier scheduler on complex input programs", as it is written in [35]. However, it shows that techniques as complex as Feautrier's ones are not always necessary to parallelize real codes.

Some work remains for handling non-perfect loop nests. 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.

Some work remains also to be done for exploiting the parallelism that has been detected. Indeed, a parallelizing algorithm only enhances operations that can be carried concurrently. It does not precise *where* these operations can be efficiently performed: automatic data mapping remains a crucial issue.

Acknowledgments

We would like to thank the anonymous referees for their careful reading and helpful comments. We also wish to give special thanks to Paul Feautrier for many fruitful discussions that made clearer the link between the different parallelization algorithms, and to François Irigoin for his wise advice and his always pertinent remarks.

References

[1] John R. Allen and Ken Kennedy. PFC: a program to convert Fortran to parallel form. Technical Report MASC-TR82-6, Rice University, Houston, TX, USA, 1982.

[2] 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.

[3] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26(4), 1994.

[4] Utpal Banerjee. A theory of loop permutations. In Gelernter, Nicolau, and Padua, editors, *Languages and Compilers for Parallel Computing*. MIT Press, 1990.

[5] Arthur J. Bernstein. Analysis of programs for parallel processing. *IEEE Transactions on Electronic Computers*, 15:757–762, October 1966.

[6] Pierre-Yves Calland, Alain Darte, Yves Robert, and Frédéric Vivien. Plugging anti and output dependence removal techniques into loop parallelization algorithms. *Parallel Computing*, 23(1-2):251–266, 1997.

[7] Jean-François Collard, Denis Barthou, and Paul Feautrier. Fuzzy Array Dataflow Analysis. In *Proceedings of 5th ACM SIGPLAN Symp. on Principles and practice of Parallel Programming*, Santa Barbara, CA, July 1995.

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

[9] Patrick Le Gouëslier d'Argence. An asymptotically optimal affine schedule on bounded convex polyhedric domains. In *Proceedings of Euro-Par'96 Parallel Processing*, volume 1124 of *LNCS*. Springer Verlag, August 1996.

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

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

[12] 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.

[13] Alain Darte, Georges-André Silber, and Frédéric Vivien. Combining retiming and scheduling techniques for loop parallelization and loop tiling. *Parallel Processing Letters*, 1997. Special issue, to appear. Also available as Tech. Rep. LIP, ENS-Lyon, RR96-34.

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

[15] 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. Will also appear in PPL, Special Issue, 1997.

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

[17] 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.

[18] 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*, 12(1-3):83–112, 1997. Special issue on Optimizing Compilers for Parallel Languages.

[19] Jack J. Dongarra and Stanley C. Eisenstat. lud. World Wide Web document, URL: `http://netlib.bell-labs.com/netlib/benchmark/index.html`.

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

[21] 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.

[22] 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.

[23] Gene H. Golub and Charles F. Van Loan. *Matrix computations*. Johns Hopkins, 2 edition, 1989.

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

[25] François Irigoin, Pierre Jouvelot, and Rémy Triolet. Semantical interprocedural parallelization: an overview of the PIPS project. In *Proceedings of the 1991 ACM International Conference on Supercomputing*, Cologne, Germany, June 1991.

[26] 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.

[27] François Irigoin and Rémy Triolet. Dependence approximation and global parallel code generation for nested loops. In *Proceedings of the International Workshop on Parallel and Distributed Algorithms*, October 1988.

[28] 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.

[29] 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.

[30] 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.

[31] Wayne Kelly and William Pugh. A framework for unifying reordering transformations. Technical Report CS-TR-3193, University of Maryland, April 1993.

[32] 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.

[33] Leslie Lamport. The parallel execution of DO loops. *Communications of the ACM*, 17(2):83–93, February 1974.

[34] Amy W. Lim and Monica S. Lam. Maximizing parallelism and minimizing synchronization with affine transforms. In *Proceedings of the 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 1997.

[35] Wolfgang Meisl. Practical methods for scheduling and allocation in the polytope model. World Wide Web document, URL: `http://brahms.fmi.uni-passau.de/cl/loopo/doc`.

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

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

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

[39] 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.

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

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

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