Chapter 5. Loop Parallelization Algorithms

Alain Darte, Yves Robert, and Frédéric Vivien

LIP, Ecole Normale Supérieure de Lyon, F - 69364 LYON Cedex 07, France
`[Alain.Darte,Yves.Robert,Frederic.Vivien]@lip.ens-lyon.fr`

Summary. This chapter is devoted to a comparative survey of loop parallelization algorithms. Various algorithms have been presented in the literature, such as those introduced by Allen and Kennedy, Wolf and Lam, Darte and Vivien, and Feautrier. These algorithms make use of different mathematical tools. Also, they do not rely on the same representation of data dependences. In this chapter, we survey each of these algorithms, and we assess their power and limitations, both through examples and by stating "optimality" results. An important contribution of this chapter is to characterize which algorithm is the most suitable for a given representation of dependences. This result is of practical interest, as it provides guidance for a compiler-parallelizer: given the dependence analysis that is available, the simplest and cheapest parallelization algorithm that remains optimal should be selected.

1. Introduction

Loop parallelization algorithms are useful source to source program transformations. They are particularly appealing as they can be applied *without* any knowledge of the target architecture. They can be viewed as a first – *machine-independent* – step in the code generation process. Loop parallelization will detect parallelism (transforming **DO** loops into **DOALL** loops) and will expose those dependences that are responsible for the intrinsic sequentiality of some operations in the original program.

Of course, a second step in code generation will have to take machine parameters into account. Determining a good granularity generally is a key to efficient performance. Also, data distribution and communication optimization are important issues to be considered. But all these problems will be addressed on a later stage. Such a two-step approach is typical in the field of parallelizing compilers (other examples are general task graph scheduling and software pipelining).

This chapter is devoted to the study of various **parallelism detection algorithms** based on:

1. A simple decomposition of the dependence graph into its strongly connected components such as Allen and Kennedy's algorithm [2].
2. Unimodular loop transformations, either ad-hoc transformations such as Banerjee's algorithm [3], or generated automatically such as Wolf and Lam's algorithm [31].
3. Schedules, either mono-dimensional schedules [10, 12, 19] (a particular case being the hyperplane method [26]) or multi-dimensional schedules [15, 20].

These loop parallelization algorithms are very different for a number of reasons. First, they make use of various mathematical techniques: graph algorithms for (1), matrix computations for (2), and linear programming for (3). Second, they take a different description of data dependences as input: graph description and dependence levels for (1), direction vectors for (2), and description of dependences by polyhedra or affine expressions for (3). For each of these algorithms, we identify the key concepts that underline them, and we discuss their respective power and limitations, both through examples and by stating "optimality" results.

An important contribution of this chapter is to characterize which algorithm is the most suitable for a given representation of dependences. No need to use a sophisticated dependence analysis algorithm if the parallelization algorithm cannot take advantage of the precision of its result. Conversely, no need to use a sophisticated parallelization algorithm if the dependence representation is not precise enough.

The rest of this chapter is organized as follows. Section 2 is devoted to a brief summary of what loop parallelization algorithms are all about. In Section 3, we review major dependences abstractions: dependence levels, directions vectors, and dependence polyhedra. Allen and Kennedy's algorithm [2] is presented in Section 4 and Wolf and Lam's algorithm [31] is presented in Section 5. It is shown that both algorithms are "optimal" in the class of those parallelization algorithms that use the same dependence abstraction as their input, i.e. dependence levels for Allen and Kennedy and direction vectors for Wolf and Lam. In Section 6 we move to a new algorithm that subsumes both previous algorithms. This algorithm is based on a generalization of direction vectors, the dependence polyhedra. In Section 7 we briefly survey Feautrier's algorithm, which relies on exact affine dependences. Finally, we state some conclusions in Section 8.

2. Input and Output of Parallelization Algorithms

Nested **DO** loops enable to describe a set of computations, whose size is much larger than the corresponding program size. For example, consider n nested loops whose loop counters describe a n-cube of size N: these loops encapsulate a set of computations of size N^n. Furthermore, it often happens that such loop nests contain a non trivial **degree of parallelism**, i.e. a set of independent computations of size $\Omega(N^r)$ for $r \geq 1$.

This makes the parallelization of nested loops a very challenging problem: a compiler-parallelizer must be able to detect, if possible, a non trivial degree of parallelism with a compilation time *not* proportional to the sequential execution time of the loops. To make this possible, efficient parallelization algorithms must be proposed with a *complexity*, an *input size* and an *output size* that depend only on n but certainly not on N, i.e. that depend on the size of the sequential code but not on the number of computations described.

The input of parallelization algorithms is a description of the dependences which link the different computations. The output is a description of an equivalent code with explicit parallelism.

2.1 Input: Dependence Graph

Each statement of the loop nest is surrounded by several loops. Each iteration of these loops defines a particular execution of the statement, called an **operation**. The dependences between the operations are represented by a directed acyclic graph: the **expanded dependence graph** (EDG). There are as many vertices in the EDG as operations in the loop nest. Executing the operations of the loop nest while respecting the partial order specified by the EDG guarantees that the correct result of the loop nest is preserved. Detecting parallelism in the loop nest amounts to detecting anti-chains in the EDG. We illustrate the notion of "expanded dependence graph" with the Example 21 below. The EDG corresponding to this code is depicted on Figure 2.1.

Example 21.

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

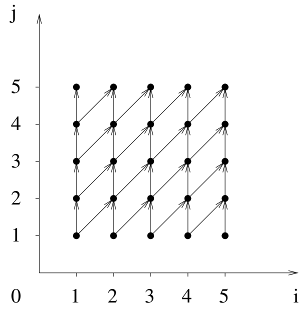

Fig. 2.1. Example 21 and its EDG.

Unfortunately, the EDG cannot be used as input for parallelization algorithms, since it is usually too large and may not be described exactly at compile-time. Therefore the **reduced dependence graph** (RDG) is used instead. The RDG is a condensed and approximated representation of the EDG. This approximation must be a superset of the EDG, in order to preserve the dependence relations. The RDG has one vertex per statement in the loop nest and its edges are labeled according to the chosen approximation of dependences (see Section 3 for details). Figure 2.2 presents two possible RDGs for Example 21, corresponding to two different approximations of the dependences.

Since its input is a RDG and not an EDG, a parallelization algorithm is not able to distinguish between two different EDGs which have the same RDG. Hence, the parallelism that can be detected is the parallelism contained

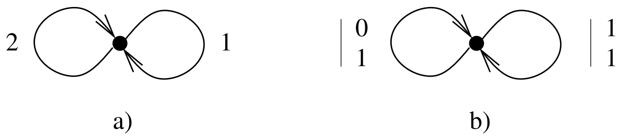

Fig. 2.2. RDG: a) with dependence levels; b) with direction vectors.

in the RDG. Thus, the quality of a parallelization algorithm must be studied *with respect to* the dependence analysis.

For example, Example 21 and Example 22 have the same RDG with dependence levels (Figure 2.2 (a)). Thus, a parallelization algorithm which takes as input RDGs with dependence levels, cannot distinguish between the two codes. However, Example 21 contains one degree of parallelism whereas Example 22 is intrinsically sequential.

Example 22.

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

2.2 Output: Nested Loops

The size of the parallelized code, as noticed before, should not depend on the number of operations that are described. This is the reason why the output of a parallelization algorithm must always be described by a set of loops [1].

There are at least three ways to define a new order on the operations of a given loop nest (i.e. three ways to define the output of the parallelization algorithm), in terms of nested loops:

1. Use elementary loop transformations as basic steps for the algorithm, such as loop distribution (as in Allen and Kennedy's algorithm), or loop interchange and loop skewing (as in Banerjee's algorithm);
2. Apply a linear change of basis on the iteration domain, i.e. apply a unimodular transformation on the iteration vectors (as in Wolf and Lam's algorithm).
3. Define a d-dimensional schedule, i.e. apply an affine transformation from $\mathbb{Z}^n$ to $\mathbb{Z}^d$ and interpret the transformation as a multi-dimensional timing function. Each component will correspond to a sequential loop, and

[1] These loops can be arbitrarily complicated, as long as their complexity only depends on the size of the initial code. Obviously, the simpler the result, the better. But, in this context, the meaning of "simple" is not clear: it depends on the optimizations that may follow. We consider that structural simplicity is preferable, but this can be discussed.

the missing $(n-d)$ dimensions will correspond to **DOALL** loops (as in Feautrier's algorithm and Darte and Vivien's algorithm).

The output of these three transformation schemes can indeed be described as loop nests, after a more or less complicated rewriting processes (see [8, 9, 11, 31, 36]). We do not discuss the rewriting process here. Rather, we focus on the link between the representation of dependences (the input) and the loop transformations involved in the parallelization algorithm (the output). Our goal is to characterize which algorithm is optimal for a given representation of dependences. Here, "optimal" means that the algorithm succeeds in exhibiting the maximal number of parallel loops.

3. Dependence Abstractions

For the sake of clarity, we restrict ourselves to the case of perfectly nested **DO** loops with affine loop bounds. This restriction permits to identify the iterations of the 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**) bounded 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, where $1 \leq i \leq s$. 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 3.1 recalls the different concepts of dependence graphs introduced in the informal discussion of Section 2.1: expanded dependence graphs (EDG), reduced dependence graphs (RDG), apparent dependence graphs (ADG), and the notion of distance sets. In Section 3.2, 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 3.3, we show how the model of PRDG generalizes classical dependence abstractions of distance sets such as dependence levels and direction vectors.

3.1 Dependence Graphs and Distance Sets

Dependence relations between operations are defined by Bernstein's conditions [4]. 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 one. Depending on the order of write(s) and/or read, the dependence corresponds to a so called

flow dependence, **anti dependence** or **output dependence**. 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 nonnegative when $S_i(I) \Longrightarrow S_j(J)$.

In general, the EDG cannot 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 (see [35, 37] for a survey on dependence tests such as the gcd test, the power test, the omega test, the lambda test, and [18] for more details on exact dependence analysis). Instead, dependences are captured through a smaller cyclic directed graph, with s vertices (as many as statements), 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 $e : S_i \rightarrow S_j$) if there exists at least one pair (I, J) such that $S_i(I) \Longrightarrow S_j(J)$. Furthermore, the [2] edge e from S_i to S_j in the RDG 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 the representation of this approximation make 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:

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

For a certain class of nested loops, it is possible to express exactly this set of pairs (I, J) (see [18]): I is given as an affine function (in some particular cases, involving floor or ceiling functions) $f_{i,j}$ of J where J varies in a polyhedron $\mathcal{P}_{i,j}$:

$$\{(I, J) \in \mathcal{D}^2 \mid S_i(I) \Longrightarrow S_j(J)\} = \{(f_{i,j}(J), J) \mid J \in \mathcal{P}_{i,j} \subset \mathcal{D}\} \tag{3.1}$$

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 3.1 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

[2] Actually, there is such an edge for each pair of memory accesses that induces a dependence between S_i and S_j.

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 3.1), since the information concerning the **location** in the EDG of such a distance vector is lost. This may even cause some loss of parallelism, as will be seen in Example 64. 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 Lamport [26] and by Wolfe in [32, 33], then used in Wolf and Lam's parallelizing algorithm [31].
– **dependence polyhedron**, introduced in [22] and used in Irigoin and Triolet's supernode partitioning algorithm [23]. We refer to the PIPS software [21] for more details on dependence polyhedra.

We now formally define reduced dependence graphs whose edges are labeled by dependence polyhedra. Then we show that this representation subsumes the two other representations, namely dependence levels and direction vectors.

3.2 Polyhedral Reduced Dependence Graphs

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

Definition 31 (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 polytope and of a polyhedral cone (for more details see [30]). 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 nonnegative combination of its rays and of any combination of its lines.

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

$$p = \sum_{i=1}^{\omega} \mu_i v_i + \sum_{i=1}^{\rho} \nu_i r_i + \sum_{i=1}^{\lambda} \xi_i l_i \tag{3.2}$$

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 are interested only in integral vectors that belong to the dependence polyhedra, since dependence distance are indeed integral vectors.

Definition 32. *A* **polyhedral reduced dependence graph (PRDG)** *is a RDG, where 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 $(J - I) \in P(e)$.*

We explore in Section 6 this representation of dependences. At first sight, the reader can see dependence polyhedra as a generalization of direction vectors.

3.3 Definition and Simulation of Classical Dependence Representations

We 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 dependence levels 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 unique distance vector is called a **uniform dependence vector**.

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

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

Indeed, consider an edge e labeled by a direction vector d and denote by I^+, I^- and I^* the sets of components of d which are respectively equal to

$z+$ (for some integer z), $z-$, and $*$. Finally, denote by d_z the n-dimensional vector whose i-th component is equal to z if the i-th component of d is equal to z, $z+$ or $z-$, and to 0 otherwise.

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

$$p = d_z + \sum_{i \in I^+} \nu_i e_i - \sum_{i \in I^-} \nu'_i e_i + \sum_{i \in I^*} \xi_i e_i \tag{3.3}$$

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

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

Dependence levels The representation by level is the less accurate dependence abstraction. In a loop nest with n nested loops, the set of distance vectors is approximated by an integer l, in $[1, n] \cup \{\infty\}$, 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**. A reduced dependence graph whose edges are labeled by dependence levels is called a Reduced Leveled Dependence Graph (RLDG).

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 information on the remaining components. Therefore, an edge of level $l < \infty$ is equivalent to the direction vector: $(\overbrace{0, \ldots, 0}^{l-1}, 1+, \overbrace{*, \ldots, *}^{n-l})$ and an edge of level ∞ corresponds to the null dependence vector. As any direction vector admits an equivalent polyhedron, so does a representation by level. For example, a level 2 dependence in a 3-dimensional loop nest, means a 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. Allen and Kennedy's Algorithm

Allen and Kennedy's algorithm [2] has first been designed to vectorizing loops. Then, it has been extended so as to maximize the number of parallel loops and to minimize the number of synchronizations in the transformed code. The input of this algorithm is a RLDG.

Allen and Kennedy's algorithm is based on the following facts:

1. A loop is parallel if it has no loop carried dependence, i.e. if there is no dependence, whose level is equal to the depth of the loop, that concerns a statement surrounded by the loop.
2. All iterations of a statement S_1 can be carried out before any iteration of a statement S_2 if there is no dependence in the RLDG from S_2 to S_1.

Property (1) allows to mark a loop as a **DOALL** or a **DOSEQ** loop, whereas property (2) suggests that parallelism detection can be independently conducted in each strongly connected component of the RLDG. Parallelism extraction is done by loop distribution.

4.1 Algorithm

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

ALLEN-KENNEDY*(G, k).*

- If $k > n$, stop.
- Decompose $G(k)$ into its strongly connected components G_i and sort them topologically.
- Rewrite code so that each G_i belongs to a different loop nest (at level k) and the order on the G_i is preserved (distribution of loops at level $\geq k$).
- For each G_i, mark the loop at level k as a **DOALL** loop if G_i has no edge at level k. Otherwise mark the loop as a **DOSEQ** loop.
- For each G_i, call ALLEN-KENNEDY(G_i, $k+1$).

We illustrate Allen and Kennedy's algorithm on the code below:

Example 41.

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

The dependence graph $G = G(1)$, drawn on Figure 4.1, has only one strongly connected component and at least one edge at level 1, thus the first call finds that the outermost loop is sequential. However, at level 2 (the edge at level 1 is no longer considered), $G(2)$ has two strongly connected components: all iterations of statement S_2 can be carried out before any

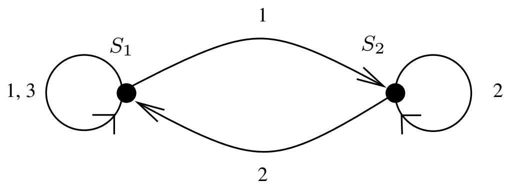

Fig. 4.1. RLDG for Example 41.

iteration of statement S_1. A loop distribution is performed. The strongly connected component including S_1 contains no edge at level 2 but one edge at level 3. Thus the second loop surrounding S_1 is marked DOSEQ and the third one DOALL. The strongly connected component including S_2 contains an edge at level 2 but no edge at level 3. Thus the second loop surrounding S_1 is marked DOALL and the third one DOSEQ. Finally, we get:

```
DOSEQ i=1,n
   DOSEQ j=1,n
      DOALL k=1,n
         $S_2$: b(i, j, k) = b(i, j-1, k+j) + a(i-1, j, k)
      ENDDO
   ENDDO
   DOALL j=1,n
      DOSEQ k=1,n
         $S_1$: a(i, j, k) = a(i-1, j+i, k) + a(i, j, k-1) + b(i, j-1, k)
      ENDDO
   ENDDO
ENDDO
```

4.2 Power and Limitations

It has been shown in [6] that for each statement of the initial code, as many surrounding loops as possible are detected as parallel loops by Allen and Kennedy's algorithm. More precisely, consider a statement S of the initial code and L_i one of the surrounding loops. Then L_i will be marked as parallel if and only if there is no dependence at level i between two instances of S. This result proves only that the algorithm is optimal among all parallelization algorithms that describe, in the transformed code, the instances of S with exactly the same loops as in the initial code. In fact a much stronger result has been proved in [17]:

Theorem 41. *Algorithm* ALLEN-KENNEDY *is optimal among all parallelism detection algorithms whose input is a Reduced Leveled Dependence Graph (RLDG).*

It is proved in [17] that for any loop nest $\mathcal{N}_1$, there exists a loop nest $\mathcal{N}_2$, which has the same RLDG, and such that for any statement S of $\mathcal{N}_1$ surrounded after parallelization by d_S sequential loops, there exists in the exact dependence graph of $\mathcal{N}_2$ a dependence path which includes $\Omega(N^{d_S})$ instances of statement S. In other words, Allen and Kennedy's algorithm cannot distinguishes $\mathcal{N}_1$ from $\mathcal{N}_2$ as they have the same RLDG, and the parallelization algorithm is optimal in the strongest sense on $\mathcal{N}_2$ as it reaches on each statement the upper bound on the parallelism defined by the longest dependence paths in the EDG.

This proves that, as long as the only information available is the RLDG, it is not possible to find more parallelism than found by Allen and Kennedy's algorithm. In other words, algorithm Allen-Kennedy is well adapted to a representation of dependences by dependence levels. Therefore, to detect more parallelism than found by algorithm Allen-Kennedy, more information on the dependences is required. Classical examples for which it is possible to overcome algorithm Allen-Kennedy are Example 42 where a simple interchange (Figure 4.2) reveals parallelism and Example 43 where a simple skew and interchange (Figure 4.3) are sufficient.

Example 42.

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

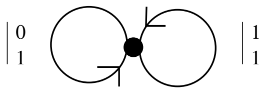
$\begin{vmatrix}0\\1\end{vmatrix}$
$\begin{vmatrix}1\\1\end{vmatrix}$

Fig. 4.2. Example 42: code and RDG.

Example 43.

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

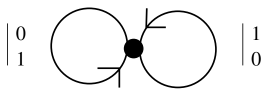
$\begin{vmatrix}0\\1\end{vmatrix}$
$\begin{vmatrix}1\\0\end{vmatrix}$

Fig. 4.3. Example 43: code and RDG.

5. Wolf and Lam's Algorithm

Examples 42 and 43 contain some parallelism, that can not be detected by Allen and Kennedy's algorithm. Therefore, as shown by Theorem 41, this parallelism can not be extracted if the dependences are represented by dependence levels. To overcome this limitation, Wolf and Lam [31] proposed an algorithm that uses direction vectors as input. Their work unifies all previous algorithms based on elementary matrix operations such as loop skewing, loop interchange, loop reversal, into a unique framework: the framework of valid **unimodular transformations**.

5.1 Purpose

Wolf and Lam aim at building sets of fully permutable loop nests. Fully permutable loops are the basis of all tiling techniques [5, 23, 29, 31]. Tiling is used to expose medium-grain and coarse-grain parallelism. Furthermore, a set of d fully permutable loops can be rewritten as a single sequential loop and $d-1$ parallel loops. Thus, this method can also be used to express fine grain parallelism.

Wolf and Lam's algorithm builds the largest set of outermost fully permutable[3] loops. Then it looks recursively at the remaining dimensions and at the dependences not satisfied by these loops. The version presented in [31] builds the set of loops via a case analysis of simple examples, and relies on a heuristic for loop nests of depth greater than or equal to six. In the rest of this section, we explain their algorithm from a theoretical perspective, and we provide a general version of this algorithm.

5.2 Theoretical Interpretation

Unimodular transformations have two main advantages: linearity and invertibility. Given a unimodular transformation T, the linearity allows to easily check whether T is a valid transformation. Indeed, T is valid if and only if $Td >_l 0$ for all non zero distance vectors d. The invertibility enables to rewrite easily the code as the transformation is a simple change of basis in $\mathbb{Z}^n$.

In general, $Td >_l 0$ cannot be checked for all *distance* vectors, as there are two many of them. Thus, one tries to guarantee $Td >_l 0$ for all non-zero *direction* vectors, with the usual arithmetic conventions in $\mathbb{Z} \cup \{*\} \cup (\mathbb{Z} \times \{+, -\})$. In the following, we consider only non-zero direction vectors, which are known to be lexicographically positive (see Section 3.1).

Denote by $t(1), \ldots, t(n)$, the rows of T. Let Γ be the closure of the cone generated by all direction vectors. For a direction vector d:

$$Td >_l 0 \Leftrightarrow \exists k_d,\ 1 \leq k_d \leq n \mid \forall i,\ 1 \leq i < k_d,\ t(i).d = 0 \text{ and } t(k_d).d > 0.$$

This means that the dependences represented by d are carried at loop level k_d. If $k_d = 1$ for all direction vectors d, then all dependences are carried by the first loop, and all inner loops are **DOALL** loops. $t(1)$ is then called a

[3] The i-th and $(i+1)$-th loops are permutable if and only if the i-th and $(i+1)$-th components of any distance vector of depth $\geq i$ are nonnegative.

timing vector or **separating hyperplane**. Such a timing vector exists if and only if Γ is pointed, i.e. if and only if Γ contains no linear space. This is also equivalent to the fact that the cone Γ^+ – defined by $\Gamma^+ = \{y \mid \forall x \in \Gamma,\ y.x \geq 0\}$ – is full-dimensional (see [30] for more details on cones and related notions). Building T from n linearly independent vectors of Γ^+ permits to transform the loops into n fully permutable loops.

The notion of timing vector is at the heart of the hyperplane method and its variants (see [10, 26]), which are particularly interesting for exposing fine-grain parallelism, whereas the notion of fully permutable loops is the basis of all tiling techniques. As said before, both formulations are strongly linked by Γ^+.

When the cone Γ is not pointed, Γ^+ has a dimension r, $1 \leq r < n$, $r = n - s$ where s is the dimension of the lineality space of Γ. With r linearly independent vectors of Γ^+, one can transform the loop nest so that the r outermost loops are fully permutable. Then, one can recursively apply the same technique to transform the $n - r$ innermost loops, by considering the direction vectors not already carried by one of the r outermost loops, i.e by considering the direction vectors included in the lineality space of Γ. This is the general idea of Wolf and Lam's algorithm even if they obviously did not express it in such terms in [31].

5.3 The General Algorithm

Our discussion can be summarized by the algorithm Wolf-Lam given below. Algorithm Wolf-Lam takes as input a set of direction vectors D and a sequence of linearly independent vectors E (initialized to void) from which the transformation matrix is built:

Wolf-Lam*(D, E).*

- Define Γ as the closure of the cone generated by the direction vectors of D.
- Define $\Gamma^+ = \{y \mid \forall x \in \Gamma,\ y.x \geq 0\}$ and let r be the dimension of Γ^+.
- Complete E into a set E' of r linearly independent vectors of Γ^+ (by construction, $E \subset \Gamma^+$).
- Let D' be the subset of D defined by $d \in D' \Leftrightarrow \forall v \in E',\ v.d = 0$ (i.e. $D' = D \cap E'^{\perp} = D \cap \text{lin.space}(\Gamma)$).
- Call Wolf-Lam(D', E').

Actually, the above process may lead to a non unimodular matrix. Building the desired unimodular matrix T can be done as follows:

- Let D be the set of direction vectors. Set $E = \emptyset$ and call Wolf-Lam(D, E).
- Build a non singular matrix T_1 whose first rows are the vectors of E (in the same order). Let $T_2 = pT_1^{-1}$ where p is chosen so that T_2 is an integral matrix.
- Compute the left Hermite form of T_2, $T_2 = QH$, where H is nonnegative, lower triangular and Q is unimodular.
- Q^{-1} is the desired transformation matrix (since $pQ^{-1}D = HT_1D$).

We illustrate this algorithm with the following example:

Example 51.

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

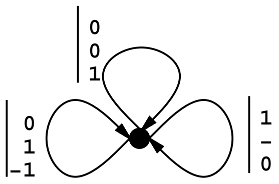

Fig. 5.1. Example 51: code and RDG.

The set of direction vectors is $D = \{(1, -, 0), (0, 0, 1), (0, 1, -1)\}$ (see Figure 5.1). The lineality space of $\Gamma(D)$ is two-dimensional (generated by $(0, 1, 0)$ and $(0, 0, 1)$). Thus, $\Gamma^+(D)$ is one dimensional and generated by $E_1 = \{(1, 0, 0)\}$. Then $D' = \{(0, 0, 1), (0, 1, -1)\}$ and $\Gamma(D')$ is pointed. We complete E_1 by two vectors of $\Gamma^+(D')$, for example by $E_2 = \{(0, 1, 0), (0, 1, 1)\}$. In this particular example, the transformation matrix whose rows are E_1, E_2 is already unimodular and corresponds to a simple loop skewing. For exposing **DOALL** loops, we choose the first vector of E_2 in the relative interior of Γ^+, for example $E_2 = \{(0, 2, 1), (0, 1, 0)\}$. In terms of loops transformations, this amounts to skewing the loop k by factor 2 and then to interchanging loops j and k:

```
DOSEQ i=1,n
  DOSEQ k=3,3×n
    DOALL j=max(1, $\lceil \frac{k-n}{2} \rceil$), min($n$, $\lfloor \frac{k-1}{2} \rfloor$)
      a(i, j, k-2×j) = a(i-1, j+i, k-2×j) + a(i, j, k-2×j-1) + a(i, j-1, k-2×j+1)
    ENDDO
  ENDDO
ENDDO
```

5.4 Power and Limitations

Wolf and Lam showed that this methodology is optimal (Theorem B.6. in [31]): “an algorithm that finds the maximum coarse grain parallelism, and then recursively calls itself on the inner loops, produces the maximum degree of parallelism possible”. Strangely, they gave no hypothesis for this theorem. However, once again, this theorem has to be understood with respect to the dependence analysis that is used: namely, direction vectors, but without any information on the structure of the dependence graph. A correct formulation is the following:

Theorem 51. *Algorithm* Wolf-Lam *is optimal among all parallelism detection algorithms whose input is a set of direction vectors (implicitly, one considers that the loop nest has only one statement or that all statements form an atomic block).*

Therefore, as for algorithm Allen-Kennedy, the sub-optimality of algorithm Wolf-Lam in the general case has to be found, not in the algorithm methodology, but in the weakness of its input: the fact that the structure of the RDG is not exploited may result in a loss of parallelism. For example, contrarily to algorithm Allen-Kennedy, algorithm Wolf-Lam finds no parallelism in Example 41 (whose RDG is given by Figure 5.2) because of the typical structure of the direction vectors $(1, -, 0), (0, 1, -), (0, 0, 1)$.

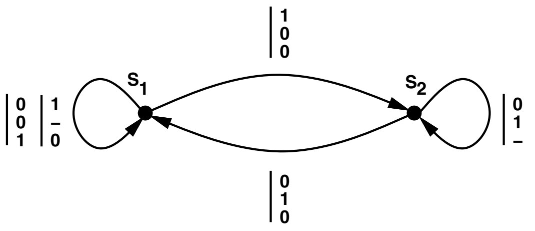

Fig. 5.2. Reduced Dependence Graph with direction vectors for Example 41.

6. Darte and Vivien's Algorithm

In this section, we introduce a third parallelization algorithm, that takes as input polyhedral reduced dependence graphs. We first explain our motivation (Section 6.1), then we proceed to a step-by-step presentation of the algorithm. We work out several examples.

6.1 Another Algorithm Is Needed

We have seen two parallelization algorithms so far. Each algorithm may output a pure sequential code for examples where the other algorithm does find some parallelism. This motivates the search for a new algorithm subsuming algorithms Wolf-Lam and Allen-Kennedy. To reach this goal, one can imagine to combine these algorithms, so as to simultaneously exploit the structure of the RDG and the structure of the direction vectors: first, compute the cone generated by the direction vectors and transform the loop nest so as to expose the largest outermost fully permutable loop nest; then, consider the subgraph of the RDG, formed by the direction vectors that are not carried by the outermost loops, and compute its strongly connected components; finally, apply a loop distribution in order to separate these components, and recursively apply the same technique on each component.

Such a strategy enables to expose more parallelism by combining unimodular transformations *and* loop distribution. However, it is not optimal as Example 61 (Figure 6.1) illustrates. Indeed, on this example, combining algorithms Allen-Kennedy and Wolf-Lam as proposed above enables to find only one degree of parallelism, since at the second phase the RDG remains strongly connected. This is not better than the basic algorithm Allen-Kennedy. However, one can find two degrees of parallelism in Example 61 by scheduling $S_1(i, j, k)$ at time-step $4i-2k$ and $S_2(i, j, k)$ at time-step $4i-2k+3$.

Example 61.

```
DO i=1,n
 DO j=1,n
  DO k=1,n
   $S_1$: a(i, j, k) = b(i-1, j+i, k) + b(i, j-1, k+2)
   $S_2$: b(i, j, k) = a(i, j-1, k+j) + a(i, j, k-1)
  ENDDO
 ENDDO
ENDDO
```

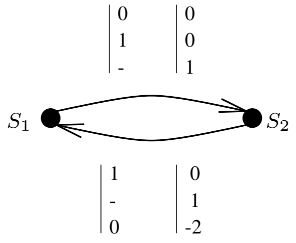

Fig. 6.1. Example 61: code and RDG.

Consequently, we would like to have a single parallelization algorithm which finds some parallelism at least when Allen-Kennedy or Wolf-Lam does. The obvious solution would be to try Allen-Kennedy, then Wolf-Lam (and even a combination of both algorithms) and to report the best answer. But such a naive approach is not powerful enough, because it uses either the dependence graph structure (Allen-Kennedy) or direction vectors (Wolf-Lam), but never benefits from both knowledges at the same step. For example, the proposed combination of both algorithms would use the dependence graph structure before or after the computation of a maximal set of fully permutable loops, but never during this computation. We claim that information on both the graph structure and the direction vectors **must** be used simultaneously. This is because the key concept when scheduling RDGs is not the cone generated by the direction vectors (i.e. the weights of the edges of the RDG), but turns out to be the cone generated by the *weights of the cycles* of the RDG.

This is the motivation for the multi-dimensional scheduling algorithm presented below. It can be seen as a combination of unimodular transformations, loop distribution, *and* index-shift method. This algorithm subsumes algorithms Allen-Kennedy and Wolf-Lam. Beforehand we motivate the choice of the representation of the dependences that the algorithm works with.

6.2 Polyhedral Dependences: A Motivating Example

In this section we present an example which contains some parallelism that cannot be detected if the dependences are represented by levels or direction vectors. However, there is no need to use an exact representation of the dependences to find some parallelism in this loop nest. Indeed, a representation of the dependences with dependence polyhedra enables us to parallelize this code.

Example 62.

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

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

Fig. 6.2. Example 62: source code and exact dependence relations

Consider Example 62 of Figure 6.2. Its exact dependences are listed on the same figure, and Figure 6.3 shows the corresponding (reduced) dependence graphs when dependence edges are labeled respectively with levels and direction vectors. What is the output of our favorite parallelization algorithms?

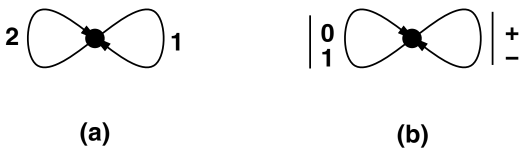

Fig. 6.3. RDG for Example 62: (a) by levels, (b) by direction vectors.

- **Allen-Kennedy**. Here, the levels of the three dependences are respectively 2, 1, and 1. There is a dependence cycle at depth 1 and at depth 2. Therefore, no parallelism is detected.
- **Wolf-Lam**. Here, 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.

– **Feautrier**. This algorithm will be described in Section 7. It takes as input the exact dependences. It leads to the valid schedule $T(i,j) = 2i + j - 3$. 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[4] parametric linear programs (as in Feautrier's algorithm), enables to reveal one degree of parallelism. The corresponding parallelized code is:

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

However, in Example 62, 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 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:

$$Xu \ge 1 \qquad Xv \ge 1 \qquad Xr \ge 0$$

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

which leads, as for Feautrier, 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, but 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 we do not need to use the affine form of Farkas' lemma anymore as in Feautrier's algorithm

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

(see Section 7). 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 and 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 6.4: we introduce a new node S' that enables to simulate ϕ and a null-weight edge to go from S' back to the initial node T. This "uniformization" principle is the underlying idea of the loop parallelization algorithm described in this section.

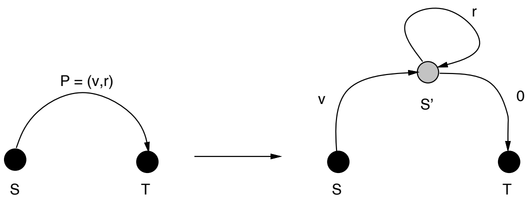

Fig. 6.4. Simulation of an edge labeled by a polyhedron with one vertex and one ray.

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 (u, v, and r). 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 [24].
– 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 by the parallelization algorithm.

6.3 Illustrating Example

We work out the following example, assuming that in the reduced dependence graph, edges are labeled by direction vectors. The dependence graph, depicted in Figure 6.5, was built by the dependence analyzer Tiny [34].

The reader can check that neither ALLEN-KENNEDY, nor WOLF-LAM, 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

Example 63.

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

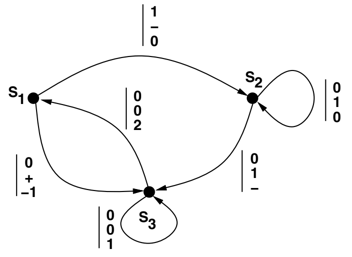

Fig. 6.5. Example 63: code and RDG.

and $(2i+1, 2k+3)$ for the third statement. This schedule corresponds to the code with explicit parallelism given below (but in which no effort, such as loop peeling, has been made so as to remove "if" tests). Thus, for each statement, one level of parallelism can be detected.

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

This code has been generated, from the schedule given above, by the procedure "codegen" of the Omega Calculator[5] delivered with Petit [25]. 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 implemented as such.

[5] The Omega Calculator is a framework to compute dependences, to check the validity of program transformations, and to transform programs, once the transformation is given.

6.4 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 between the vertices of a dependence graph and the vertices of a dependence polyhedron, we call the first one **nodes** instead of **vertices**. Furthermore, 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 the polyhedron $P(e)$. 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: ω, ρ, and λ denote the number of vertices v_i, of rays r_i, and of 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
  – Add to $W$ a new virtual node $n_e$,
  – Add to $F$ $\omega$ edges of weights $v_1, v_2, \ldots, v_\omega$ directed from $x_e$ to $n_e$,
  – Add to $F$ $\rho$ self-loops around $n_e$ of weights $r_1, r_2, \ldots, r_\rho$,
  – Add to $F$ $\lambda$ self-loops around $n_e$ of weights $l_1, l_2, \ldots, l_\lambda$,
  – Add to $F$ $\lambda$ self-loops around $n_e$ of weights $-l_1, -l_2, \ldots, -l_\lambda$,
  – Add to $F$ a null weight edge directed from $n_e$ to $y_e$.
```

Back to Example 63. The PRDG of Example 63 is drawn in Figure 6.5. Figure 6.6 shows the uniform dependence graph associated to it. It has three new nodes in gray (i.e. virtual nodes) that correspond to the symbol "+" and the two symbols "−" in the initial direction vectors.

6.5 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

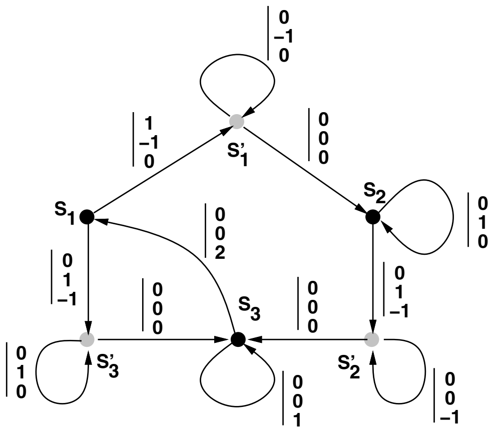

Fig. 6.6. Translated uniform reduced dependence graph.

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).

G' is defined as the subgraph of G generated by all the edges of G that belong to at least one multi-cycle of null 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 6.6).

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

DARTE-VIVIEN(G, k).

1. Build G' the subgraph of G generated by all edges that belong to at least one null weight multi-cycle of G.
2. Add in G', all edges from x_e to y_e and all self-loops on y_e if $e = (x_e, y_e)$ is an edge already in G', from an actual node x_e to a virtual node y_e.
3. Select a vector X, and a constant ρ_S for each node S in G, 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 $X_S^k = X$.

4. If G' is empty or has only virtual nodes, return.
5. If G' is strongly connected and has at least one actual node, G is not computable (and the initial PRDG G_o is not consistent), return.
6. Otherwise, decompose G' into its strongly connected components G_i and call DARTE-VIVIEN(G_i, $k+1$) for each subgraph G_i that has at least one actual node.

Remarks

– Step (2) is necessary only for general PRDGs: for example, it could be removed for RDGs labeled by direction vectors (for details see [16]). In this case, the resolution of a single linear program can simultaneously solve Step (1) and Step (3).
– In Step (3), we do not specify, on purpose, how the vector X and the constants ρ are selected, so as to allow various selection criteria. For example, a maximal set of linearly independent vectors X can be selected if the goal is to derive fully permutable loops (see [13] for details).

Back to Example 63 Consider the uniform dependence graph of Figure 6.6. There are two elementary cycles of weights $(1,0,1)$ and $(0,1,1)$, and five self-loops of weights $(0,0,1)$, $(0,0,-1)$, $(0,1,0)$ (twice) and $(0,-1,0)$. Therefore, all edges (except the edges that only belong to the cycle of weight $(1,0,1)$) belong to a multi-cycle of null weight. The subgraph G' is drawn in Figure 6.7.

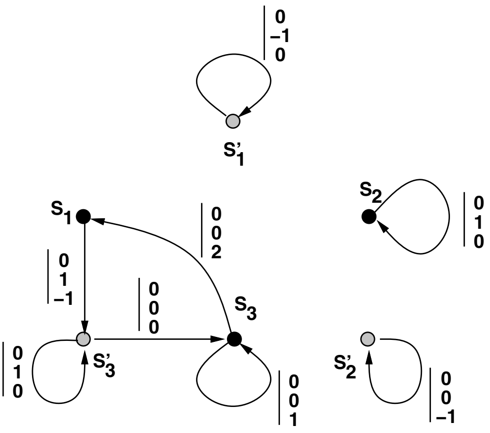

Fig. 6.7. Subgraph of null weight multi-cycles for Example 63.

The constraints coming from edges in G' impose that $X=(x,y,z)$ must be orthogonal to the weight of all cycles of G'. Therefore, $y=z=0$. Finally, considering the other constraints, we find the solution $X=(2,0,0)$, $\rho_{S_1}=\rho_{S_3}=1$ and $\rho_{S_2}=0$. In G', there remain four strongly connected components, and two of them are not considered since they only have virtual nodes. The two other components have no null weight multi-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 Section 6.3, the 2-dimensional schedules: $(2i, j)$ for S_2, $(2i+1, 2k)$ for S_1 and $(2i+1, 2k+3)$ for S_3.

6.6 Schematic Explanations

G_u does not always correspond to the RDG of a loop nest since its dependence vectors are not necessarily lexicographically nonnegative. In fact, if one forgets that some nodes are virtual, G_u is nothing but the reduced dependence graph of a System of Uniform Recurrence Equations (SURE), introduced by Karp, Miller and Winograd [24]. Karp, Miller and Winograd study the problem of computability of a SURE: they show that its computability is linked to the problem of detecting cycles of null weight in its RDG G, which can be done by a recursive decomposition of the graph, based on the detection of multi-cycles of null weight. The key structure of their algorithm is G', the subgraph of G generated by the edges that belong to a multi-cycle of null weight.

G' can efficiently be built by the resolution of a simple linear program (program 6.1 or its dual program 6.2). This resolution enables to design a parallelization algorithm, whose principle is dual to Karp, Miller and Winograd's algorithm:

$$\min \left\{ \sum_e v_e \quad | \quad q \geq 0, \quad v \geq 0, \quad w \geq 0, \quad q + v = 1 + w, \quad Bq = 0 \right\} \tag{6.1}$$

$$\max \left\{ \sum_e z_e \quad | \quad z \geq 0, \quad 0 \leq z_e \leq 1, \quad Xw(e) + \rho_{y_e} - \rho_{x_e} \geq z_e \right\} \tag{6.2}$$

where $w(e)$ is the dependence vector associated to the edge e, $B = [CW]^t$, C is the connection matrix and W the matrix of dependence vectors.

Without entering the details, X is a n-dimensional vector and there is one variable ρ per vertex of the RDG and one variable z per edge of the RDG. The edges of G' (resp. $G \setminus G'$) are the edges $e = (x_e, y_e)$ for which $z_e = 0$ (resp. $z_e = 1$) in the optimal solution of the dual (program 6.2), and equivalently, for which $v_e = 0$ (resp. $v_e = 1$) in the primal (program 6.1). When summing inequalities $Xw(e) + \rho_{y_e} - \rho_{x_e} \geq z_e$ on a cycle C of G, one finds that $Xw(C) = 0$ if C is a cycle of G' and $Xw(C) \geq l(C) > 0$ otherwise ($l(C)$ is the number of edges of C not in G').

To see the link with algorithm Wolf-Lam, when considering the cone Γ generated by the *weights of the cycles* (and not the weights of the edges), G'

is the subgraph whose cycle weights generate the lineality space of Γ and X is a vector of the relative interior of Γ^+. However, there is no need to build Γ effectively to build G'. This is one of the interest of the linear programs 6.1 and 6.2.

We have outlined the main ideas of algorithm DARTE-VIVIEN [15]. Some technical modifications are needed to distinguish between virtual and actual nodes, and to take into account the nature of the edges (vertices, rays or lines of a dependence polyhedron): see [16] for full details.

6.7 Power and Limitations

Now that we have a multi-dimensional schedule T, we can prove its optimality in terms of degree of parallelism. We can show [14,16] that for each statement S (i.e. for each node of G_o), the number of instances of S that have been sequentialized by T is of the same order as the number of instances of S that are inherently sequentialized by the dependences.

Theorem 61. *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 (the number of nested recursive calls) of the 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 considered inherently sequential because of the dependence analysis.*

Once again, this algorithm is optimal with respect to the dependence analysis. Consider the example in Figure 6.8.

Example 64.

```
DO i=1,n
 DO j=i,n
  $S_1$ a(i, j) = b(i-1, j+i) + a(i, j-1)
  $S_2$ b(i, j) = a(i-1, j-i) + b(i, j-1)
 ENDDO
ENDDO
```

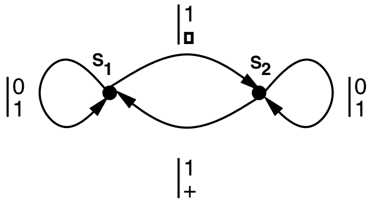

Fig. 6.8. Example 64: code and RDG.

If the dependences are described by distance vectors, the RDG has two self-dependences $(0, 1)$ and two edges labeled by polyhedra, both with one vertex and one ray (respectively $(0, 1)$ and $(0, -1)$). Therefore, there exists a multi-cycle of null weight. Furthermore, the two actual vertices belong to G'. Thus, the depth of algorithm DARTE-VIVIEN is 2 and no parallelism can be found.

However, computing iteration (i, j) of the first statement (resp. the second statement) at step $2i + j$ (resp. $i + j$), leads to a valid schedule that exposes one degree of parallelism [6]. DARTE-VIVIEN was not able to find parallelism in this example because the approximation of the dependences had already lost all the parallelism.

The technique we used here to detect parallel loops consists in looking for multi-dimensional schedules whose linear parts (the vectors X) may be different for different statements *even if they belong to the same strongly connected component.* This is the base of Feautrier's algorithm [20] whose fundamental mathematical tool is the affine form of Farkas' lemma. Theorem 61 however shows that there is no need to look for different linear parts (whose construction is more expensive and lead to more complicated rewriting processes) in a given strongly connected component of the current subgraph G', as long as dependences are given by distances vectors. On the other hand, Example 64 shows that such a refinement may be useful only when a more accurate dependence analysis is available.

7. Feautrier's Algorithm

In [20], Paul Feautrier proposed an algorithm to schedule static control programs with affine dependences. This algorithm makes use of an exact dependence analysis, which is always feasible for such programs [18]. This is to be contrasted with the previous three algorithms (ALLEN-KENNEDY, WOLF-LAM, and DARTE-VIVIEN) which work with an approximation of the dependences.

Feautrier's algorithm takes as input a reduced dependence graph G in which an edge $e : S_i \rightarrow S_j$ is labeled by the set of pairs (I, J) such that $S_j(J)$ depends on $S_i(I)$. This algorithm builds recursively a multi-dimensional affine schedule for each statement of the loop nest:

```
FEAUTRIER($G$).
– Decompose $G$ into its strongly connected component $G_i$ and sort them
  topologically.
– For each strongly connected component $G_i$:
    – Find an affine schedule by statement which induces a non-negative delay
      on all dependences and satisfies as many dependences as possible.
    – Build the set $G'_i$ of unsatisfied edges. If $G'_i \neq \emptyset$, call FEAUTRIER($G'_i$).
```

This algorithm is similar to DARTE-VIVIEN because of its structure and output, and because both use linear programs to build affine schedules. Here are the main points for a comparison of the two algorithms:

[6] The schedules $\lfloor \frac{3}{2}i + j + \frac{1}{2} \rfloor$ and $\lfloor \frac{1}{2}i + j \rfloor$ minimize the latency but the code is more complicated to write.

- Darte-Vivien is able to schedule [7] programs even if dependence analysis is not feasible, given a RDG with polyhedral dependences. Feautrier is only able to process static control programs with affine dependences. In this sense, the first algorithm is more powerful. Note however that there are some attempts to generalize Feautrier's approach by weakening the constraints on its input, using a Fuzzy Array Dataflow Analysis [7].
- When dependence analysis is feasible, Feautrier is much more powerful. This algorithm is able to process any set of loops that describe polyhedra, even if the loops are not perfectly nested. Darte-Vivien can also process non perfectly nested loops, either by considering each block of perfectly nested loops separately, or by fusing artificially the non perfectly nested loops. In theory however, this is less natural and less powerful.
- Darte-Vivien is based on the resolution of linear programs that are similar to those solved by Feautrier. The only (through fundamental) difference is that the former looks for less general affine transformations. Therefore, on static control programs with affine dependences, Feautrier always find more parallelism than Darte-Vivien (cf. Example 64). However, despite this difference, the optimality result for Darte-Vivien gives some hints concerning the optimality cases of Feautrier that was first presented as a "greedy heuristic".
- Feautrier needs to use the affine form of Farkas' lemma to obtain its linear programs, which Darte-Vivien avoids thanks to its uniformization scheme. Therefore, Feautrier's linear programs are more complex.
- Both algorithms were extended from fine grain to medium grain parallelism detection through a search for fully permutable loops. Darte et al. [13] proposed an extension of Darte-Vivien which is a mere generalization of Wolf-Lam. Lim and Lam [27] proposed an extension of Feautrier which finds maximal sets of fully permutable loops while minimizing the amount of synchronizations required in the parallelized code.
- Darte-Vivien produces schedules as regular as possible in order to generate codes as simple as possible. Indeed, this algorithm rewrites the codes using affine schedules, but, unlike Feautrier, these affine schedules are chosen such that 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, the output codes are guaranteed to be simpler than Feautrier's codes.

A small comparison study was conducted in [28]. It used only four examples. As expected, the complexity of Darte-Vivien was (much) lower than that of Feautrier. More surprisingly, both algorithms output the same result on each of the four examples considered. Obviously, more real examples should be processed to reach a conclusion. At least we can say that more complex techniques do not always provide better results!

[7] We did not write "is able to rewrite"...

Finally, here is a code (Example 71) which obviously contain some parallelism, but which cannot be parallelized by any of the four parallelization algorithms surveyed in this paper:

Example 71.

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

```
DOPAR i=1,⌊n/2⌋
 a(i) = 1 + a(n-i)
ENDDO
DOPAR i=⌊n/2⌋ + 1,n
 a(i) = 1 + a(n-i)
ENDDO
```

Fig. 7.1. Example 71: original code and parallelized version.

8. Conclusion

Our study provides a classification of loop parallelization algorithms. The main results are the following: Allen and Kennedy's algorithm is optimal for a representation of dependences by levels, and Wolf and Lam's algorithm is optimal for a representation by direction vectors (but for a loop nest with only one statement). Neither one subsumes the other, since each uses information that cannot be exploited by the other (graph structure for the first one, direction vectors for the second one). However, both are subsumed by Darte and Vivien's algorithm which is optimal for any polyhedral representation of distance vectors. Feautrier's algorithm subsumes Darte and Vivien's algorithm when dependences can be represented as affine dependences, but the characterization of its optimality remains open.

We believe this classification of loop parallelization algorithms to be of practical interest. It provides guidance for a compiler-parallelizer in order to choose the most suitable algorithm: given the dependence analysis that is available, the simplest and cheapest parallelization algorithm that remains optimal should be selected. Indeed, this is the algorithm that is the most appropriate to the available representation of dependences.

References

1. J.R. Allen and K. Kennedy. PFC: a program to convert programs to parallel form. Technical report, Dept. of Math. Sciences, Rice University, TX, March 1982.
2. J.R. Allen and K. Kennedy. Automatic translations of Fortran programs to vector form. *ACM Toplas*, 9:491–542, 1987.
3. Utpal Banerjee. A theory of loop permutations. In Gelernter, Nicolau, and Padua, editors, *Languages and Compilers for Parallel Computing*. MIT Press, 1990.

4. A. J. Bernstein. Analysis of programs for parallel processing. In *IEEE Trans. on El. Computers, EC-15*, 1966.
5. Pierre Boulet, Alain Darte, Tanguy Risset, and Yves Robert. (pen)-ultimate tiling? *Integration, the VLSI Journal*, 17:33–51, 1994.
6. D. Callahan. *A Global Approach to Detection of Parallelism.* PhD thesis, Dept. of Computer Science, Rice University, Houston, TX, 1987.
7. J.-F. Collard, D. Barthou, and P. 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. Jean-François Collard. Code generation in automatic parallelizers. In Claude Girault, editor, *Proc. Int. Conf. on Application in Parallel and Distributed Computing. IFIP WG 10.3*, pages 185–194. North Holland, April 1994.
9. Jean-François Collard, Paul Feautrier, and Tanguy Risset. Construction of DO loops from systems of affine constraints. *Parallel Processing Letters*, 5(3):421–436, September 1995.
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. Mapping uniform loop nests onto distributed memory architectures. *Parallel Computing*, 20:679–710, 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. Technical Report 96-34, LIP, ENS-Lyon, France, November 1996.
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. Optimal fine and medium grain parallelism detection in polyhedral reduced dependence graphs. In *Proceedings of PACT'96*, Boston, MA, October 1996. IEEE Computer Society Press.
16. 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.
17. Alain Darte and Frédéric Vivien. On the optimality of Allen and Kennedy's algorithm for parallelism extraction in nested loops. *Journal of Parallel Algorithms and Applications*, 96. Special issue on Optimizing Compilers for Parallel Languages.
18. Paul Feautrier. Dataflow analysis of array and scalar references. *Int. J. Parallel Programming*, 20(1):23–51, 1991.
19. 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.
20. 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.
21. F. Irigoin, P. Jouvelot, and R. Triolet. Semantical interprocedural parallelization: an overview of the PIPS project. In *Proceedings of the 1991 ACM International Conference on Supercomputing*, Cologne, Germany, June 1991.
22. F. Irigoin and R. Triolet. Computing dependence direction vectors and dependence cones with linear systems. Technical Report ENSMP-CAI-87-E94, Ecole des Mines de Paris, Fontainebleau (France), 1987.

23. F. Irigoin and R. Triolet. Supernode partitioning. In *Proc. 15th Annual ACM Symp. Principles of Programming Languages*, pages 319–329, San Diego, CA, January 1988.
24. 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.
25. 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.
26. Leslie Lamport. The parallel execution of DO loops. *Communications of the ACM*, 17(2):83–93, February 1974.
27. 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.
28. 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`.
29. R. Schreiber and Jack J. Dongarra. Automatic blocking of nested loops. Technical Report 90-38, The University of Tennessee, Knoxville, TN, August 1990.
30. Alexander Schrijver. *Theory of Linear and Integer Programming.* John Wiley and Sons, New York, 1986.
31. 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.
32. M. Wolfe. *Optimizing Supercompilers for Supercomputers.* PhD thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, October 1982.
33. Michael Wolfe. *Optimizing Supercompilers for Supercomputers.* MIT Press, Cambridge MA, 1989.
34. Michael Wolfe. *TINY, a loop restructuring research tool.* Oregon Graduate Institute of Science and Technology, December 1990.
35. Michael Wolfe. *High Performance Compilers For Parallel Computing.* Addison-Wesley Publishing Company, 1996.
36. Jingling Xue. Automatic non-unimodular transformations of loop nests. *Parallel Computing*, 20(5):711–728, May 1994.
37. Hans Zima and Barbara Chapman. *Supercompilers for Parallel and Vector Computers.* ACM Press, 1990.