A comparison of nested loops parallelization algorithms.

Alain Darte, Frédéric Vivien

A comparison of nested loops parallelization algorithms

Alain Darte and Frédéric Vivien

May 1995

Research Report N^{O} 95-11

A comparison of nested loops parallelization algorithms

Alain Darte and Frédéric Vivien

May 1995

Abstract

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

Keywords: automatic parallelization, dependence analysis, linear programming

Résumé

Dans ce rapport, nous comparons trois algorithmes de parallélisation automatique de boucles imbriquées (les algorithmes de Kennedy et Allen, de Wolf et Lam et de Darte et Vivien) qui utilisent des représentations différentes des vecteurs de distance. Nous identifions les concepts qui leur sont communs et ceux qui les différencient. Nous étudions l'optimalité de chacun des algorithmes par rapport à l'analyse de dépendance qu'il utilise. Nous illustrons sa puissance et ses limitations par des exemples bien choisis. Cette étude permet finalement d'identifier quel algorithme est le mieux adapté a une analyse de dépendance donnée.

Mots-clés: parallélisation automatique, analyse de dépendances, programmation linéaire

A comparison of nested loops parallelization algorithms

Alain Darte and Frédéric Vivien

May 1995

Abstract

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

Contents

List of Figures

1 Introduction

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

However, detecting parallelism (i.e. transforming **DO** loops into **DOALL** loops), and understanding parallelism (i.e. detecting which dependences are responsible for the sequentiality in the code) is independent of the target architecture. It only depends on the structure of the sequential code to be parallelized. This is certainly one of the reasons why a large amount of algorithms have been proposed for detecting **DOALL** loops, as a first step in the parallelization process. First, one studies the problem of parallelization on an ideal machine (a PRAM for example), and then, further optimizations are taken into account (depending on the machine for which the code is to be compiled) such as the choice of granularity, the data distribution, the optimization of communications, ...This two-step approach is the most often used and not only in the field of automatic nested loops parallelization: this is also the case, among others, for general task scheduling or software pipelining.

This paper studies different **parallelism detection algorithms** based on:

i. a simple decomposition of the dependence graph into its strongly connected components such as Allen and Kennedy's algorithm [AK87].

ii. unimodular loop transformations, either ad-hoc transformations such as Banerjee's algorithm [Ban90], or generated automatically such as Wolf and Lam's algorithm [WL91].

iii. schedules, either mono-dimensional schedules [KMW67, DKR91, Fea92a] (a particular case being the hyperplane method [Lam74]) or multi-dimensional schedules [DV94, Fea92b].

These algorithms seem very different not only by the techniques they use (graph algorithms for (i), matrix computations for (ii), linear programming for (iii)), but also by the description of dependences they work with (graph description and level of dependences for (i), direction vectors for (ii), description of dependences by polyhedra for (iii)). Nevertheless, we try to identify the concepts that make these algorithms different or similar and we discuss their respective power and limitations.

Our main result is that all parallelizing algorithms, that use information only on distance vectors, can be subsumed by a general algorithm, based on an algorithm first proposed by Karp, Miller and Winograd [KMW67] in the context of uniform recurrence equations. This algorithm has three main properties:

- it can be adapted to all usual representations of distance vectors;
- it can be proven optimal with respect to the representation of dependences it works with;
- it points out exactly which dependences are responsible for a loss of parallelism.

Furthermore, we show that Allen and Kennedy's algorithm and Wolf and Lam's algorithm are particular implementations of this algorithm for less accurate dependence representations. As a consequence, they can also be proven optimal with respect to the dependence representation they use.

This study permits to characterize exactly 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 can not use the precision of its result. Conversely, no need to use a sophisticated parallelization algorithm if the dependence representation is not precise enough.

2 Input and output of parallelization algorithms

Nested **DO** loops are one of the code structures that permit to describe a set of computations, whose size is not proportional to the code size. For example, n nested loops whose loop counters describe a n-cube of size N, correspond to 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. sets of independent computations of size $\Omega(N^r)$ for $r \geq 1$).

This aspect 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 only depend on the size of the sequential code and not on the number of computations it describes. The input of parallelization algorithms is a description of the dependences that link the different computations generated by the loop nest, the output is a description of an equivalent code with explicit parallelism.

2.1 Input: reduced dependence graph

Each iteration of the loops that surround a statement corresponds to a particular execution of the statement, that we call an **operation**. The dependences between operations are represented by a directed acyclic graph that has as many vertices as operations: the **expanded dependence graph** (EDG). Executing the operations of the loop nest while respecting the partial order specified by the EDG guarantees that the result of the loop nest is preserved. Detecting parallelism in the loop nest means detecting anti-chains in the EDG.

Unfortunately, in general, the EDG can not be used as an input for parallelization algorithms, since it is too large (it has as many vertices as operations described by the loop nest) and may not be described exactly at compile-time. One prefers to manipulate the **reduced dependence graph** (RDG) which is a representation, in a condensed form, of an approximate EDG. This approximation must be a superset of the EDG so that dependence relations are preserved. The RDG has as one vertex per statement in the loop nest and its edges are labelled in a way depending on the chosen approximation (we will recall how in section 2.3). See [ZC90] for a survey on dependence tests such as gcd test, power test, omega test, lambda test, and [Fea91] for more details on exact dependence analysis.

Since its input is the RDG and not the EDG, a parallelization algorithm is not able to distinguish between two different EDGs which have the same RDG. The parallelism that can be detected is then the parallelism contained in the RDG. Thus, the quality of a parallelization algorithm must be studied *with respect to* the dependence analysis.

2.2 Output: nested loops

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

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

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

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

- to 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);
- to apply a linear change of basis on the iteration domain, i.e. to apply a unimodular transformation on the iteration vectors (as in Wolf and Lam's algorithm).
- to define a d-dimensional schedule, i.e. to apply an affine transformation from $\mathbb{Z}^n$ to $\mathbb{Z}^d$ and to interpret the transformation as a multi-dimensional timing function. Each component will correspond to a sequential loop, the missing $(n-d)$ dimensions will correspond to **DOALL** loops (as in Feautrier's algorithm and Darte and Vivien's algorithm).

These three transformation schemes can be described by loop nests after more or less complicated rewriting processes (see [WL91, DR94, Xue94, CFR94, Col94]). We will not discuss them here. We will rather study the link between the loops transformations involved (the output) and the dependences representation (the input), our goal being to characterize, for a given dependences representation, which algorithm is optimal, i.e. exhibits the maximal number of parallel loops.

2.3 Representations of dependences

In all dependence analysis methods, dependence relations between operations are defined by Bernstein's conditions [Ber66]. Briefly speaking, two operations are considered dependent if both access the same memory location and if at least one access is a write. Furthermore, this dependence is directed according to the sequential order. Depending on the order of write(s) and/or read, this dependence corresponds to the 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 EDG [2]. Note that $(J - I)$ is always lexicographically non negative when $S_i(I) \Longrightarrow S_j(J)$.

The RDG is a compression of the EDG. In the RDG, two statements S_i and S_j are said dependent (we write $S_i \to S_j$) if there exists at least one pair (I, J) such that $S_i(I) \Longrightarrow S_j(J)$.

[1]These loops can be arbitrary 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.

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

Furthermore, the dependence $S_i \to S_j$ is labelled by the set $\{(I,J) \in P^2 \mid S_i(I) \Longrightarrow S_j(J)\}$, or by an approximation that contains this set. The precision and representation of this approximation makes the power of the dependence analysis.

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

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

In most dependence analysis algorithms however, rather than the set of pairs (I,J), one computes the set of values $(J-I)$. This latter is called the set of **distance vectors**, or **dependence vectors**. 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.

We give below usual representations of the set of distance vectors (by decreasing precision).

Rays and vertices A dependence analysis algorithm such as [IT87] provides a description of a **dependence polyhedron** by its vertices and rays [3]. A dependence polyhedron with no vertices (or whose vertices have been converted to rays) is called a **dependence cone**. Very often, the dependence polyhedron has a single vertex but many rays.

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 all possible i-th components of distance vectors: it is equal to $z+$ (resp. $z-$) if all i-th components are greater than (resp. smaller than) or equal to z. It is equal to $*$ if the i-th component takes 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. $(-1)-$). Note that a direction vector can always be decomposed into several lexicographically non negative direction vectors. For example, the direction vector $(0+,*)$ is decomposed into $(+,*)$ and $(0,0+)$ since the distance vectors $(0,-)$ do not exist. In the rest of the paper, we will thus assume that all direction vectors are lexicographically non negative.

Level of dependence The coarsest representation of dependences is the representation by level. The set of distance vectors is represented by an integer p, in $[1 \ldots (n+1)]$, defined as the largest integer such that the $p-1$ first components of the distance vectors are zero. A dependence at level $p \le n$ means that the dependence occurs at depth p of the loop nest, i.e. at a given iteration of the $p-1$ outermost loops. In this case, one says that the dependence is a **loop carried dependence** at level p or that the dependence is carried at level p. If $p = n+1$, the dependence occurs inside the loop body, but between two different statements.

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 is lost. This may even be the cause of a loss of parallelism (see section 3.3.3). However, this representation remains important, especially when exact dependence analysis is either too expensive or not feasible.

[3] In fact, one could argue that the polyhedron is always bounded and thus has no rays. However, since loops are very often parametrized, some parametrized vertices are converted to non parametrized vertices and rays.

3 A study of different loops parallelization algorithms

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

3.1 Allen and Kennedy's algorithm

Allen and Kennedy's algorithm [AK87] is based on the following facts:

i. An outermost loop is parallel if it has no loop carried dependence, i.e. if there is no dependence with level 1.

ii. 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 RDG from S_2 to S_1.

Property (i) permits to mark a loop as a **DOALL** or a **DOSEQ** loop, whereas property (ii) suggests that the parallelism detection can be done independently in each strongly connected component of the RDG. The input of the algorithm is a description of the RDG whose edges are labelled by the levels of dependences. Parallelism extraction is done by loop distribution.

For a dependence graph G, we denote by $G(k)$ the subgraph of G in which all dependences at level 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(RDG, 1).

ALLEN-KENNEDY(G, k)

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

Example 1

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

The dependence graph $G = G(1)$ drawn on figure 1 has only one strongly connected component (and at least one edge at level 1), thus the first call has no effect. However, at level 2 (the edge at level 1 is not considered), $G(2)$ has two strongly connected components: all computations on array b can be carried out before any computation on array a. With a loop distribution at level 2 and 3, we get:

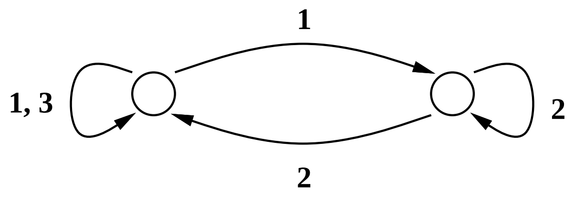

Figure 1: Reduced Dependence Graph for Example 1 (with level of dependences)

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

Property 1 *Algorithm ALLEN-KENNEDY is optimal among all parallelism detection algorithms whose input is a RDG labelled by the level of dependences.*

Proof: The proof is based on the fact that algorithm ALLEN-KENNEDY has the same behaviour as Darte and Vivien's algorithm [DV94] for the particular case of a RDG labelled by the level of dependences, if all **DOALL** loops are made innermost. The optimality of algorithm ALLEN-KENNEDY is then a consequence of the optimality of Darte and Vivien's algorithm in the general case precised by property 3. □

Property 1 shows that algorithm ALLEN-KENNEDY is well adapted to a representation of dependences by level of dependences. Therefore, to detect more parallelism than found by algorithm ALLEN-KENNEDY, is possible only if more precision is given on the dependences. A classic example for which it is possible to overcome algorithm ALLEN-KENNEDY is an example where a simple interchange (example 2) or a simple skew and an interchange (example 3) reveal parallelism (see dependence graphs on figure 2).

Examples 2 **and** 3

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

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

3.2 Wolf and Lam's algorithm

Examples 2 and 3 contain parallelism. However, as shown by property 1, this parallelism can not be extracted if the dependences are represented by level of dependences only. To remedy this

Figure 2: Reduced Dependence Graphs for Examples 2 and 3

limitation, Wolf and Lam [WL91] proposed an algorithm that uses direction vectors as input. Their work unified all previous algorithms based on elementary matrix operations such as loop skewing, loop interchange, loop reversal, in a unique framework, the framework of **valid unimodular transformations**.

Looking for unimodular transformations is of practical interest since they are (1) linear, (2) invertible in $\mathbb{Z}^n$. Given a unimodular transformation T, property (1) permits to check if T is valid (T is valid if $Td >_l 0$ for all non zero distance vectors d) and property (2) permits to rewrite easily the code (simple change of basis in $\mathbb{Z}^n$). In general, since $Td >_l 0$ can not be checked for all *distance* vectors, 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 that we can thus assume lexicographically positive (see section 2.3).

Denote by $t(1), \ldots, t(n)$, the rows of T. 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 **timing vector** or **separating hyperplane**. Such a timing vector exists if and only if Γ, the closure of the cone generated by all direction vectors, is pointed. 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 [Sch86] 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 in the heart of the hyperplane method and its variants (see [Lam74, DKR91]), which are particularly interesting for exposing fine-grain parallelism, whereas the notion of fully permutable loops is the base of all tiling techniques [IT88, SD90, BDRR94, WL91], which are used for exposing coarse-grain parallelism. As said before, both formulations are equivalent when reasoning on 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 for transforming the $n - r$ innermost loops, by considering the direction vectors not already carried by at least one of the r outermost loops (i.e that belong to the lineality space of Γ). This is the general idea of Wolf and Lam's algorithm even if it is not explicitely described in these terms in [WL91]. This can be summarized by 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').

Now, 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 non negative, lower triangular and Q is unimodular.
- Q^{-1} is the desired transformation matrix (since $pQ^{-1}D = HT_1D$).

Remark: This algorithm is not exactly the original Wolf and Lam's algorithm, but the general principle is similar. Wolf and Lam build the matrix T, step by step, during the algorithm, as a product of unimodular matrices. Furthermore, they do not compute exactly Γ^+ but they propose heuristics and special algorithms for some particular cases.

Example 4

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

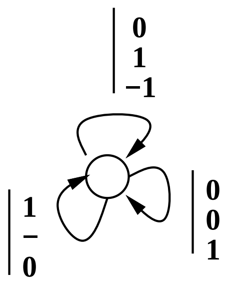

Figure 3: Reduced Dependence Graph for Example 4 (with direction vectors)

The set of direction vectors is $D = \{(1, -, 0), (0, 0, 1), (0, 1, -1)\}$ (see figure 3). 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 $E_2 = \{(0, 2, 1), (0, 1, 0)\}$. This corresponds in terms of loops transformations to skew the loop k by factor 2 and then to interchange loops j and k:

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

Wolf and Lam showed that this methodology is optimal (Theorem B.6. in [WL91]): "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 such a theorem. However, once again, this theorem has to be understood with respect to the dependence analysis that is used: here, direction vectors but with no information on the structure of the dependence graph. A correct formulation is the following:

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

Proof: Once again, we use the optimality of Darte and Vivien's algorithm: on a loop nest whose body has only one statement, and whose dependences are represented by direction vectors, Darte and Vivien's algorithm has the same behaviour as algorithm WOLF-LAM. □

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 in terms of strongly connected components is not exploited results in a loss of parallelism. For example, algorithm WOLF-LAM finds no parallelism in example 1 (whose RDG is given by figure 4) because of the typical structure of the direction vectors $(1, -, 0), (0, 1, -), (0, 0, 1)$.

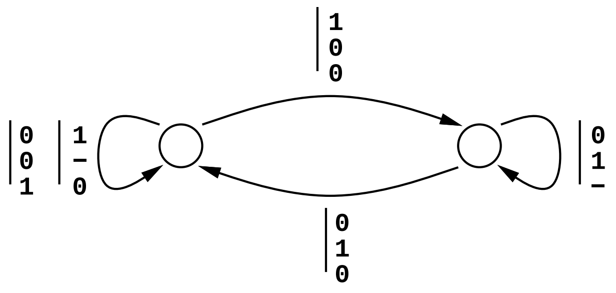

Figure 4: Reduced Dependence Graph for Example 1 (with direction vectors)

3.3 Darte and Vivien's algorithm

One can imagine to combine algorithms WOLF-LAM and ALLEN-KENNEDY, so as to exploit simultaneously 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 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 apply the same technique, recursively on each component.

Such a strategy permits to expose more parallelism by combining unimodular transformations *and* loop distribution. However, it is not optimal as example 5 illustrates. We will indeed see that the key concept is not the cone generated by the direction vectors (i.e. the weights of the edges of the RDG), but the cone generated by the *weights of the cycles* of the RDG. This remark leads to the multi-dimensional scheduling algorithm of Darte and Vivien [DV94] that can be seen as a combination of unimodular tranformations, loop distribution, *and* index-shift method.

Example 5

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

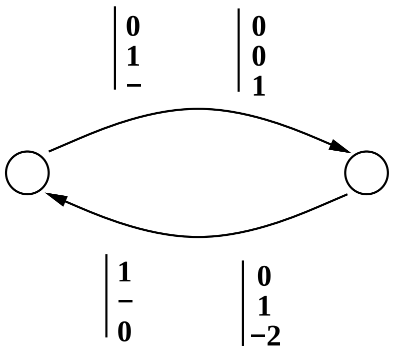

Figure 5: Reduced Dependence Graph for Example 5 (with direction vectors)

On this example (whose RDG is given on figure 5), combining algorithms ALLEN-KENNEDY and WOLF-LAM, as proposed above, finds 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 5 (see below).

Darte and Vivien's first motivation was to find an algorithm:

- that is flexible enough to support all representations of distance vectors based on a polyhedral representation.
- that detects the maximal degree of parallelism contained in the RDG.

3.3.1 Canonical representation of the RDG

The first point is that any RDG, whose edges are labelled by a polyhedral representation of the distance vectors, can be simulated by a RDG, whose edges are labelled by dependence vectors.

Consider the particular case of a dependence between two statements S_1 and S_2 whose associated distance vectors are represented by a polyhedron with a single vertex w and a single ray r. This means that, in the RDG, one considers that all distance vectors of the form $w + \lambda r$ (with $\lambda \geq 0$) exist, and that, in the EDG, there is a dependence path of length 1, from $S_1(I)$ to $S_2(I + w + \lambda r)$, for all $\lambda \geq 0$ and for all I in P (the iteration domain) such that $(I + w + \lambda r)$ belongs to P.

Thus, the situation is the same as if there were a *virtual* statement V, with a uniform self-dependence r, and two uniform dependences, w from S_1 to V and 0 from V to S_2. For simulating the distance vector $w+\lambda r$, use once the edge from S_1 to V, then turn λ times around V, and finally go to S_2. However, this simulation corresponds to a dependence path of length $\lambda+2$ instead of 1. To suppress this difference, one assigns a **delay** to each edge, 1 to the edge labelled by w and 0 to the others. The length of a simulated path is then the sum of the delays along the edges it uses.

This simulation is the base of Darte and Vivien's algorithm whose first phase consists in transforming a RDG, whose edges are labelled by polyhedra, into an equivalent RDG, whose edges are labelled by weights (dependence vectors) and delays (0 or 1), and whose vertices form two classes: the actual vertices and the virtual vertices. This phase is done by the algorithm TRANSFORM:

TRANSFORM(G)

- Mark all vertices of G as actual vertices.
- For all edges $e=(x_e, y_e)$ of G, create a virtual vertex V_e.
- If e is labelled by a polyhedron with vertices $v_1, \ldots, v_i$, rays $r_1, \ldots, r_j$ and lines $l_1, \ldots, l_k$,
  - suppress the edge e.
  - create i edges from x_e to V_e labelled by $v_1, \ldots, v_i$, with a delay 1.
  - create j self-loops around V_e labelled by $r_1, \ldots, r_j$, with a delay 0.
  - create $2k$ self-loops around V_e labelled by $l_1, \ldots, l_k$ and $-l_1, \ldots, -l_k$, with a delay 0.
  - create one edge from V_e to y_e labelled by the null vector 0, with a delay 0.
- Return the transformed graph.

Remark: when the polyhedron that labels an edge e has neither rays, nor lines, it is not necessary to create a virtual vertex. One can create edges directly from x_e to y_e.

For example, a representation of dependences by level correspond to a particular representation by direction vectors: a dependence at level $p \leq n$ is equivalent to the direction vector $(\overbrace{0, \ldots, 0}^{p-1}, 1, \overbrace{*, \ldots, *}^{n-p})$. A representation by direction vectors is equivalent to a representation with uniform dependences and virtual vertices. For example, the direction vector $(0, 1, 0-)$ corresponds to a polyhedron with one vertex $(0, 1, 0)$ and one ray $(0, 0, -1)$, whereas the polyhedron that corresponds to $(1, 2+, *)$ has one vertex $(1, 2, 0)$, one ray $(0, 1, 0)$ and one line $(0, 0, 1)$.

3.3.2 Scheduling a system of uniform recurrence equations

Note that a RDG built by the algorithm TRANSFORM does not always correspond to the RDG of a loop nest since dependence vectors are not anymore lexicographically non negative. In fact, (if one forgets that some vertices are virtual and that some edges have delay 0), this is the RDG of a system of uniform recurrence equations (SURE), introduced, in a seminal paper, by Karp, Miller and Winograd [KMW67].

Karp, Miller and Winograd studied the problem of computability of a SURE: they showed that it is linked to the problem of detecting cycles of null weight in the reduced dependence graph G, and that it can be solved by a recursive decomposition of the graph, based on the detection of multi-cycles (i.e. union of 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.

Darte and Vivien showed that G' can be efficiently built by the resolution of a simple linear program (program 2 or its dual program 3). This resolution permits to design a parallelization algorithm, whose principle is dual to Karp, Miller and Winograd's algorithm.

$$\min\left\{ \textstyle\sum_e v_e \quad | \quad q \geq 0, \quad v \geq 0, \quad w \geq 0, \quad q + v = 1 + w, \quad Bq = 0 \right\} \tag{2}$$

$$\max\left\{ \textstyle\sum_e z_e \quad | \quad z \geq 0, \quad 0 \leq z_e \leq 1, \quad Xw(e) + \rho_{y_e} - \rho_{x_e} \geq z_e \right\} \tag{3}$$

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 3), and equivalently, for which $v_e = 0$ (resp. $v_e = 1$) in the primal (program 2). When summing inequations $Xw(e) + \rho_{y_e} - \rho_{x_e} \geq z_e$ 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 length of the cycle C).

In other words and 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 lin.space(Γ) of Γ and X is a vector of the relative interior of Γ^+. However, there is no need to build Γ effectively for building G'. This is one of the interest of linear programs 2 and 3.

These are the main ideas of Darte and Vivien's algorithm. The rest are technical modifications that are needed to distinguish between virtual and actual vertices, to take into account the delay of the edges and the nature of the edges (vertices, rays or lines of a dependence polyhedron). The general principle of Darte and Vivien's algorithm is the following:

- Apply a global loop distribution for separating the different strongly connected components G_i of the RDG G.

- For each component G_i that has at least one edge, compute its transformed graph H_i = TRANSFORM(G_i) and call DARTE-VIVIEN(H_i, 1).

Algorithm DARTE-VIVIEN is given below. It takes as input a transformed RDG, strongly connected, with at least one edge, and it returns a so called multi-dimensional schedule, i.e, for each actual vertex v, a set of d_v n-dimensional vectors $X_1^v, \ldots, X_{d_v}^v$ and d_v constants $\rho_1^v, \ldots, \rho_{d_v}^v$, such that computing the iteration I of the statement S_v at the multi-dimensional step $(X_1^v.I + \rho_1^v, \ldots, X_{d_v}^v.I + \rho_{d_v}^v)$ leads to a valid schedule (if these steps are lexicographically ordered).

DARTE-VIVIEN(G, k)

- Build G', the subgraph of G generated by the edges that belong to a multi-cycle of null weight.

- For a given dependence polyhedron, add in G' all the edges that simulate this polyhedron, if at least one of the edges that correspond to its vertices is already in G' (technical modification).

- Find a vector X and constants ρ_v, such that:

$$\begin{cases} Xw(e) + \rho_{y_e} - \rho_{x_e} \geq 0 \text{ for all edges } e = (x_e, y_e) \in G' \\ Xw(e) + \rho_{y_e} - \rho_{x_e} \geq d_e \text{ for all edges } e = (x_e, y_e) \notin G' \text{ with delay } d_e \end{cases}$$

  For all actual vertices v of G, let $\rho_k^v = \rho_v$ and $X_k^v = X$.

- If G' is empty, return.

- If G' is strongly connected and has at least one actual vertex, G is not computable (and the initial RDG is not consistent).
- Otherwise, decompose G' into its strongly connected components G_i and for each G_i that has at least one actual vertex, call DARTE-VIVIEN(G_i, $k+1$).

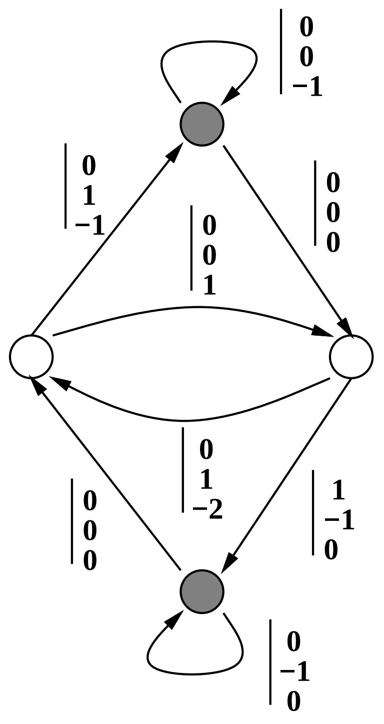

Figure 6: Transformed Reduced Dependence Graph for Example 5

We now go back to example 5. The transformed RDG is given on figure 6. 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 $(1,0,-1)$, $(1,-1,1)$, $(0,2,-3)$, $(0,1,-1)$ for the other elementary cycles. Therefore, Γ is pointed and one can find a one-dimensional schedule, for example given by $X=(4,0,-2)$, $\rho_a=0$ and $\rho_b=3$. Two degrees of parallelism can be exposed and the resulting code is then:

```
   $k = 2 - n$
   DOALL 1 $i = \max(1, \lceil\frac{k+1}{2}\rceil), \min(n, \lfloor\frac{n+k}{2}\rfloor)$
     DOALL 1 $j = 1, n$
       a($i$, $j$, $-k+2*i$) = b($i-1$, $j+i$, $-k+2*i$) + b($i$, $j-1$, $-k+2*i+2$)
1    CONTINUE

   DOSEQ 2 $k = 3-n, 2*n-1$
     DOALL 3 $i = \max(1, \lceil\frac{k+1}{2}\rceil), \min(n, \lfloor\frac{n+k}{2}\rfloor)$
       DOALL 3 $j = 1, n$
         a($i$, $j$, $-k+2*i$) = b($i-1$, $j+i$, $-k+2*i$) + b($i$, $j-1$, $-k+2*i+2$)
3    CONTINUE
     DOALL 4 $i = \max(1, \lceil\frac{k}{2}\rceil), \min(n, \lfloor\frac{n+k-1}{2}\rfloor)$
       DOALL 4 $j = 1, n$
         b($i$, $j$, $-k+2*i+1$) = a($i$, $j-1$, $-k+2*i+j+1$) + a($i$, $j$, $-k+2*i$)
4    CONTINUE
```

```
2 CONTINUE

  $k = 2 * n$
  DOALL 5 $i = \max(1, \lceil \frac{k}{2} \rceil), \min(n, \lfloor \frac{n+k-1}{2} \rfloor)$
    DOALL 5 $j = 1, n$
      $b(i, j, -k + 2 * i + 1) = a(i, j - 1, -k + 2 * i + j + 1) + a(i, j, -k + 2 * i)$
5 CONTINUE
```

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

Proof: Consider a loop nest whose reduced dependence graph is G. Let $H = \text{TRANSFORM}(G)$ and $d = \max\{d_v \mid v \text{ actual vertex of } H\}$ where d_v has been given by algorithm DARTE-VIVIEN for each actual vertex of H, thus for each vertex of G. d is the recursion depth of algorithm DARTE-VIVIEN. The transformed code contains at most d nested sequential loops ($(n - d)$ degrees of parallelism are exposed). Furthermore, for a loop nest whose iteration domain contains (resp. is contained in) a n-dimensional cube of size N (resp. λN for some $\lambda \geq 1$), one can build a dependence path of length $\Omega(N^d)$ in the EDG that corresponds to G (this is the difficult part of the proof). Therefore, any parallelization algorithm would expose a sequentiality of $\Omega(N^d)$. Since the sequentiality exposed by algorithm DARTE-VIVIEN is $O(N^d)$, it is optimal. $\square$

Studying the transformed RDG of examples 1 to 4 permits to better understand why parallelism were (or were not) found by the previous algorithms. The dependences that are responsible for the inherent sequentiality of the loop nest are exactly those that correspond to edges of G'. This has two consequences:

- If G' has only flow dependences, there is no need to transform the code into single assignment form since this would not increase the degree of parallelism in the code.
- If the dependence analysis is not exact, G' shows which edges deserve a more accurate dependence analysis for detecting more parallelism. There is no need to give a more precise description of edges in $G \setminus G'$ since they are not responsible for the loss of parallelism.

3.3.3 Limitations of Darte and Vivien's algorithm

Darte and Vivien's algorithm is optimal for any polyhedral representation of *distance vectors* (Property 3). However, it may not be optimal if more information is given on the *pairs of iteration vectors* that induce a dependence. This comes from the fact that the set of distance vectors $\{(J - I) \mid S_1(I) \Rightarrow S_2(J)\}$ is the *projection* of the set $\{(J - I, J) \mid S_1(I) \Rightarrow S_2(J)\}$ (which is as precise as the set of pairs $\{(I, J) \mid S_1(I) \Rightarrow S_2(J)\}$). Therefore, the projection makes us believe that the distance vectors can take place anywhere in the iteration domain even if this is not true. This loss of precision may be the cause of a loss of parallelism as example 6 illustrates.

Example 6

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

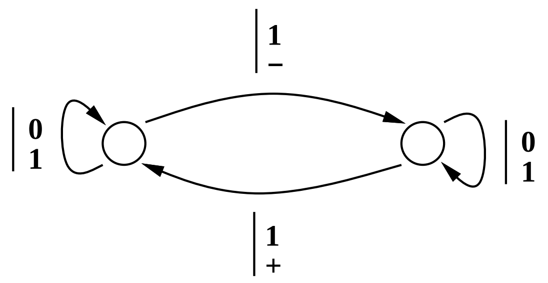

Figure 7: Reduced Dependence Graph for Example 6 (with direction vectors)

If the dependences are described by distance vectors, the RDG (see figure 7) has two self-dependences $(0, 1)$ and two edges labelled 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 [4].

The technique used here 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 [Fea92b] whose fundamental mathematical tool is the affine form of Farkas lemma. Property 3 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 6 shows that it can be useful when a more accurate dependence analysis is available. Now, the only remaining open question concerns the optimality of Feautrier's algorithm: for which representation of the dependences is Feautrier's algorithm optimal?

4 Conclusion

Our study offers a classification of loops parallelization algorithms. Our main results are the following: Allen and Kennedy's algorithm is optimal for a representation of dependences by level and Wolf and Lam's algorithm is optimal for a representation by direction vectors (but for a loop nest with only one statement). Neither of them subsumes the other one, since each uses information that can not be exploited by the other (graph structure for the first one, direction vectors structure for the second one). However, both are subsumed by Darte and Vivien's algorithm that is optimal for any polyhedral representation of distance vectors. Feautrier's algorithm is an extension of this latter, but the characterization of its optimality remains open.

We believe this classification of practical interest, since it permits a compiler-parallelizer to choose, depending on the dependence analysis at its disposal, the simplest and cheapest parallelization algorithm that remains optimal, i.e the algorithm that is the most appropriate to the available representation of dependences. Future work will try to answer the remaining open question concerning the optimality of Feautrier's algorithm.

[4]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.

References

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

[Ban90] Utpal Banerjee. A theory of loop permutations. In Gelernter, Nicolau, and Padua, editors, *Languages and Compilers for Parallel Computing*. The MIT Press, Cambridge, Massachusetts, 1990.

[BDRR94] Pierre Boulet, Alain Darte, Tanguy Risset, and Yves Robert. (pen)-ultimate tiling ? *Integration, the VLSI Journal*, 17:33–51, 1994.

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

[CFR94] Jean-François Collard, Paul Feautrier, and Tanguy Risset. Construction of do loops from systems of affine constraints. *Parallel Processing Letters*, 1994. to appear.

[Col94] 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.

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

[DR94] Alain Darte and Yves Robert. Mapping uniform loop nests onto distributed memory architectures. *Parallel Computing*, 20:679–710, 1994.

[DV94] Alain Darte and Frédéric Vivien. Automatic parallelization based on multi-dimensional scheduling. Technical Report 94-24, Laboratoire de l'Informatique du Parallélisme, Ecole Normale Supérieure de Lyon, France, September 1994.

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

[Fea92a] Paul Feautrier. Some efficient solutions to the affine scheduling problem, part I, one-dimensional time. *Int. J. Parallel Programming*, 21(5):313–348, October 1992. Available as Technical Report 92-28, Laboratoire MASI, Université Pierre et Marie Curie, Paris, May 1992.

[Fea92b] Paul Feautrier. Some efficient solutions to the affine scheduling problem, part II, multi-dimensional time. *Int. J. Parallel Programming*, 21(6):389–420, December 1992. Available as Technical Report 92-78, Laboratoire MASI, Université Pierre et Marie Curie, Paris, October 1992.

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

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

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

[Lam74] Leslie Lamport. The parallel execution of DO loops. *Communications of the ACM*, 17(2):83–93, February 1974.

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

[SD90] R. Schreiber and Jack J. Dongarra. Automatic blocking of nested loops. Technical Report 90-38, The University of Tennessee, Knoxville, TN, August 1990.

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

[Xue94] Jingling Xue. Automatic non-unimodular transformations of loop nests. *Parallel Computing*, 20(5):711–728, May 1994.

[ZC90] Hans Zima and Barbara Chapman. *Supercompilers for Parallel and Vector Computers.* ACM Press, 1990.