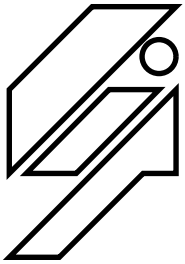
# Loop parallelization algorithms : from parallelism extraction to code generation.

Pierre Boulet, Alain Darte, Georges-Andre Silber, Frédéric Vivien




*Laboratoire de l'Informatique du Parallélisme*

Ecole Normale Supérieure de Lyon
Unité de recherche associée au CNRS n°1398

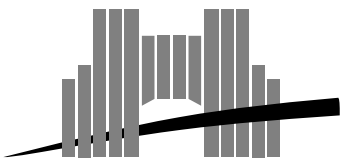

# *Loop parallelization algorithms: from parallelism extraction to code generation*


Pierre Boulet
Alain Darte
Georges-André Silber
Frédéric Vivien



June 1997

Research Report N$^{O}$ 97-17


**Ecole Normale Supérieure de Lyon**
46 Allée d'Italie, 69364 Lyon Cedex 07, France
Téléphone : (+33) (0)4.72.72.80.00 Télécopieur : (+33) (0)4.72.72.80.80
Adresse électronique : lip@lip.ens−lyon.fr

# Loop parallelization algorithms: from parallelism extraction to code generation

Pierre Boulet
Alain Darte
Georges-André Silber
Frédéric Vivien

June 1997

**Abstract**

In this paper, we survey loop parallelization algorithms, analyzing the dependence representations they use, the loop transformations they generate, the code generation schemes they require, and their ability to incorporate various optimizing criteria such as maximal parallelism detection, permutable loops detection, minimization of synchronizations, easiness of code generation, etc. We complete the discussion by presenting new results related to code generation and loop fusion for a particular class of multi-dimensional schedules, called shifted linear schedules. We demonstrate that algorithms based on such schedules, while generally considered as too complex, can indeed lead to simple codes.

**Keywords:** automatic parallelization, nested loops, parallelization algorithms, loop fusion, synchronizations, code generation.

**Résumé**

Dans ce rapport, nous présentons divers algorithmes de parallélisation, en prenant en compte la représentation des dépendances qu'ils utilisent, les transformations de boucle qu'ils génèrent, les techniques de génération de code dont ils ont besoin, et enfin, leur capacité à incorporer divers critères d'optimisation tels que la détection du parallélisme maximal, la détection de boucles permutables, la minimisation des synchronisations, la simplicité de la génération de code, etc... Nous complétons notre discussion par la présentation de nouveaux résultats liés à la génération de code et à la fusion de boucles pour une classe particulière d'ordonnancements multi-dimensionnels appelés ordonnancements linéaires décalés. Nous montrons que des algorithmes qui se fondent sur de tels ordonnancements, souvent considérés comme trop complexes, peuvent néanmoins générer des codes simples.

**Mots-clés:** Parallélisation automatique, boucles imbriquées, algorithmes de parallélisation, fusion de boucles, synchronisations, génération de code.

# Loop parallelization algorithms: from parallelism extraction to code generation

Pierre Boulet Alain Darte Georges-André Silber
Frédéric Vivien
E-mail: FirstName.LastName@lip.ens-lyon.fr

August 26, 1997

**Abstract**

In this paper, we survey loop parallelization algorithms, analyzing the dependence representations they use, the loop transformations they generate, the code generation schemes they require, and their ability to incorporate various optimizing criteria such as maximal parallelism detection, permutable loops detection, minimization of synchronizations, easiness of code generation, etc. We complete the discussion by presenting new results related to code generation and loop fusion for a particular class of multi-dimensional schedules, called shifted linear schedules. We demonstrate that algorithms based on such schedules, while generally considered as too complex, can indeed lead to simple codes.
**Keywords:** automatic parallelization, nested loops, parallelization algorithms, loop fusion, synchronizations, code generation.

# 1 Introduction

Loop transformations have been shown useful for extracting parallelism from regular nested loops for a large class of machines, from vector machines and VLIW machines to multiprocessor architectures. Several surveys have already presented in details the tremendous list of possible loop transformations (see for example the survey by Bacon, Graham and Sharp [4], or Wolfe's book [38]), and their particular use. Two additional surveys have presented the link between loop parallelization algorithms and dependence analysis: in [40], Yang, Ancourt and Irigoin characterize, for each loop transformation used to reveal parallelism, the minimal dependence abstraction needed to check its validity; in [15], a complementary study is proposed that answers the dual question: for a given dependence abstraction, what is the simplest algorithm that detects maximal parallelism?

Loop parallelization algorithms consist in finding a "good" loop transformation that reveals parallelism. But it is only a step in the compilation process: further optimizations must be taken into account (depending on the machine for which the code is to be compiled) such as the choice of the granularity of the parallel program, the data distribution,

the optimization of communications, etc. Thus, to generate efficient parallel codes, a loop parallelization algorithm must be able, either to consider optimization criteria that are more accurate than the simple detection of parallel loops, or to generate an intermediate abstract parallel code that is simple enough so that further optimizations can still be performed.

In this paper, we survey loop parallelization algorithms with this compilation process in mind. In Section 2, we explain why detecting parallel loops is not sufficient to generate parallel codes, taking the example of the compilation of High Performance Fortran. In Section 3, we present different loop parallelization algorithms proposed in the literature, recalling the dependence representations they use, the loop transformations they generate, and their capabilities to incorporate various optimizing criteria such as maximal parallelism detection, permutable loops detection, minimization of synchronizations, easiness of code generation, etc. In Section 4, we present the code generation techniques involved by these loop transformations.

The rest of the paper is devoted to a more accurate description of two particular optimization problems: how to generate codes that are as simple as possible (Section 5) and how to handle loop fusion (for example to minimize synchronizations) in loop parallelization algorithms (Section 6). These last two sections present new results. Finally, we give some conclusions in Section 7.

# 2 Compilation of parallel loops

## 2.1 Abstract parallel code

A loop is parallel if there are no *dependences* between different iterations of the loop or, in other words, if there are no dependences *carried* by the loop. Consequently, all iterations of the loop can be executed concurrently on different processors. Many languages and compilers offer means for the programmer to express that a loop is parallel and to map data among processors. The INDEPENDENT directive of HPF [21] asserts to the compiler that the iterations in the do loop that follows the directive may be executed concurrently without changing the semantics of the program. The programmer (or a parallelizing algorithm) asserts that no iteration can interfere with any other iteration.

```
do i = 1,3
   S_a : A[i] = B[i]
   S_b : C[i] = D[i]
enddo
```

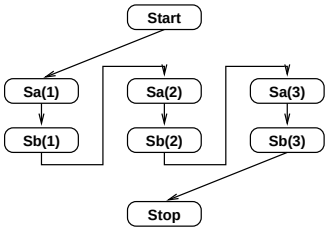



Figure 1: A sequential do loop and its precedence graph.

The precedence graph of Figure 1 represents the execution order of the statements in a sequential loop. The edges of the graph imply an order for the execution of the statements. If there is an edge from $S_a(k)$ to $S_b(l)$, it means that $S_a(k)$ must be executed *before* $S_b(l)$[1]. This

[1] $S_a(k)$ means execution of the statement $S_a$ for the loop index value $k$.

order is imposed by the semantic of do loops. The precedence graph of Figure 2 for the same code but with the INDEPENDENT directive is simpler: all the edges between two different iterations have been removed. It means that the computer may execute the iterations in parallel, if the arrays are well mapped.

```
!HPF$ INDEPENDENT
do i = 1,3
   S_a : A[i] = B[i]
   S_b : C[i] = D[i]
enddo
```

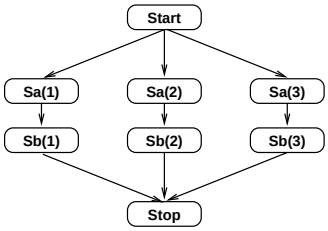



Figure 2: A simple independent do loop and its precedence graph.

Actually, in the example of Figure 1, there are no *data* dependences between $S_a$ and $S_b$ since the operations do not use the same memory locations. Thus, an iteration $i$ of the loop may be executed either as: $S_a(i)$ before $S_b(i)$, $S_b(i)$ before $S_a(i)$, or as $S_a(i)$ in parallel with $S_b(i)$. More precisely, we have the precedence graph of Figure 3: with this type of code, the compiler has a high degree of liberty to produce the executable parallel code.

```
!HPF$ INDEPENDENT
do i = 1,3
   S_a : A[i] = B[i]
   S_b : C[i] = D[i]
enddo
```

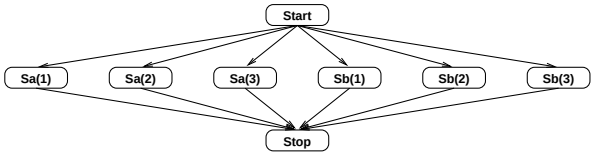



Figure 3: A simple independent do loop and its precedence graph (actual graph).

However, suppose now that we replace statement $S_b$ by $S'_b$ : $C[i-1] = A[i]$. $S'_b(i)$ uses data computed by $S_a(i)$ (the array reference $A[i]$). This time, we must enforce the precedence graph of the INDEPENDENT loop (Figure 2), i.e. we must execute $S_a(i)$ before $S'_b(i)$. There is a *loop independent* dependence, a dependence that lies inside the loop body, independent of the iteration. The compiler has a smaller degree of liberty to produce the code. Furthermore, the mapping of the arrays is important: the *owner computes rule* implies that the processor that *owns* the left hand side of the computation *computes* it. If $A[i]$ and $C[i-1]$ are not on the same processor, some communications and synchronizations may be generated in the loop itself.

This brief discussion shows that even with an INDEPENDENT directive, the actual generation of the parallel code has a variable degree of difficulty. The question is: given a parallel loop, how to produce the most efficient executable parallel code? And even more, how to produce any parallel code? We are going to deal with these questions in the next section.

## 2.2 Executable parallel code

Consider again the example of Figure 3. An HPF compiler may produce two types of code, following the owner computes rule. The first type is the simplest expression of the owner

computes rule on the entire iteration space. For each element of computation, the code tests if this element of computation is owned by the executor. This gives a parallel code like the one in Figure 4(a). To make this code more efficient, communications can be moved outside of the loop if array accesses are known at compile-time. Nevertheless, it remains inefficient since each processor spans the entire iteration space.

```
do i = 1, N
   if is_local(B[i]) send(B[i])
   if is_local(A[i]) A[i] = receive(B[i])

   if is_local(D[i]) send(D[i])
   if is_local(C[i]) C[i] = receive(D[i])
enddo
```

(a)

```
(Communication : get slice of B)
(Communication : get slice of D)
do i = my_A_slice_start, my_A_slice_stop
   A[i] = B[i]
enddo
do i = my_C_slice_start, my_C_slice_stop
   C[i] = D[i]
enddo
```

(b)

Figure 4: Two types of parallel executable codes for the code of Figure 3.

If the code to compile is simple enough and if the compiler is smart enough, a second approach is possible in which each processor computes only the slice of the array it owns, as in the code of Figure 4(b). Note that the *distribution* of the code in two loops is not needed if both slices are the same, for example if both arrays are mapped the same way.

Now, consider the case where there is a loop independent dependence as in the example modified with statement $S'_b$. An HPF compiler could generate a code like in Figure 5(a). Here, the communication for $A$ cannot be just moved outside of the loop as before, since we must communicate something that is computed *inside* the loop. A possibility is to distribute the loop to obtain the general scheme: a parallel loop, a global synchronization, a parallel loop. Another possibility is to choose a good mapping such that $A[i]$ and $C[i-1]$ are owned by the same processor, so that the loop independent dependence takes place inside a processor. In this case, we can even generate a code such as in Figure 5(b).

```
do i = 1, N
   if is_local(B[i]) send(B[i])
   if is_local(A[i])
      A(i) = receive(B[i])
      send(A[i])
   endif
   if (is_local(C[i − 1]) C[i − 1] = receive(A[i])
enddo
```

(a)

```
(Communication : get slice of B)
do i = my_A_slice_start, my_A_slice_stop
   A[i] = B[i]
   C[i − 1] = A[i]
enddo
```

(b)

Figure 5: Two types of parallel executable codes for the code with $S'_b$.

If $A$ and $C$ are not mapped to the same processor, these is a last possibility. Indeed, some compilers offer the ON HOME directive[2], modifying the owner computes rule. In the

[2]This directive is an approved extension of HPF 2.0 and some compilers have already implemented it, like ADAPTOR [7], an HPF compiler by Thomas Brandes.

previous example, we can force the compiler to produce a code that computes $C[i-1]$ at the same place as $A[i]$, by generating temporary arrays or by duplicating some computations with the help of some overlap areas (as ADAPTOR does). This kind of code can be much more difficult to produce, because the compiler needs to have a precise knowledge of array accesses in order to produce the communications.

As we have just mentioned, compiling parallel loops with loop independent dependences is quite difficult. One could argue that, in this case, a compiler should always compile a parallel loop with multiple statements as a succession of parallel loops with a single statement, interleaved with some communications/synchronizations. In other words, why not always implement loop distribution rather than loop fusion? Why trying to produce codes with "large" loop bodies?

The advantages of loop fusion are well-known. First, synchronization is a costly operation. Therefore, minimizing the number of synchronizations is important. Second, even if loop fusion increases the size of the loop, which can have a negative impact on cache and register performance, it can improve data reuse by moving references closer together in time, making them more likely to still reside in cache or registers [38, 31]. This is of major importance with the development of cache memory hierarchies. Reuse provided by fusion can even be made explicit by using scalar replacement to place array references in a register. Furthermore, fusion decreases loop overhead, increasing the granularity of parallelism, and allowing easier scalar optimizations, such as subexpression elimination.

The examples above illustrate that even if an abstract code is parallel, it can be difficult for a compiler to produce an efficient executable code. This fundamental aspect of automatic parallelization has to be taken into account: generating parallel loops is not sufficient for generating efficient parallel executable programs. When designing parallel loop detection algorithms, we must consider various criteria: of course the maximization of the degree of parallelism, but also the feasibility of the code generation, the minimization of synchronizations, the flexibility of the algorithm, the possibility of loop fusion, etc. Indeed, the detection of parallelism is a first step in the compilation scheme: it should not produce codes that are so complex that no further optimizations are possible.

# 3 Loop parallelization algorithms

The structure of nested loops allows the programmer to describe parameterized sets of computations as an enumeration, but in a particular order, called the sequential order. Many loop transformations have been proposed to change the enumeration order so as to increase the efficiency of the code, see for example the survey by Bacon, Graham and Sharp [4]. However, most of these transformations are still applied in an ad-hoc fashion, through heuristics, and only a few of them are generated fully automatically by loop parallelization algorithms.

In this section, we give a quick summary of the loop transformations that are captured by these loop parallelization algorithms (Section 3.2). Before, in Section 3.1, we recall the dependence abstractions used to check the validity of the transformations used by these algorithms. Finally, in Section 3.3, we list the main loop parallelization algorithms that have been proposed in the literature, with a survey of their main characteristics (see Table 1).

All these algorithms apply to a particular type of codes: nested loops, possibly non perfectly nested, but in which the control can be statically defined, in other words loops with no jumps and no conditionals (except conditionals that can be captured statically, for example when control dependences can be converted to data dependences, or when the conditional restricts statically the range of the loop counters). Classically, loop bounds are supposed to be affine functions of some parameters and of surrounding loop counters, with unit steps, so that the computations associated to a given statement $S$ can be described by a subset (actually the integral points of a polyhedron) $D_S$ of $\mathbb{Z}^{n_S}$, where $n_S$ is the number of loops surrounding $S$. $D_S$ is called the *iteration domain* of $S$, and the integral vectors in $D_S$ the *iteration vectors*. The $i$-th component of an iteration vector is the value of the counter of the $i$-th loop surrounding $S$, counting from the outermost to the innermost loop. To each $I \in D_S$ corresponds a particular execution of $S$ denoted by $S(I)$. In the sequential order, all computations $S(I)$ are executed following the lexicographical order defined on the iteration vectors. If $I$ and $J$ are two vectors, we write $I \prec_{lex} J$ if $I$ is lexicographically strictly smaller than $J$, and $I \preceq_{lex} J$ if $I \prec_{lex} J$ or $I = J$.

## 3.1 Dependence abstractions

Data dependence relations between operations are defined by Bernstein's conditions [5]. Two operations are 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. We write $S(I) \longrightarrow S'(J)$ if the statement $S'$ at iteration $J$ depends on the statement $S$ at iteration $I$. Dependences are captured through a directed acyclic graph, called the *reduced dependence graph* (RDG), or statement level dependence graph. Each vertex of the RDG is identified with a statement of the loop nest, and there is an edge from $S$ to $S'$ if there exists at least one pair $(I, J) \in D_S \times D_{S'}$ such that $S(I) \longrightarrow S'(J)$. An edge between $S$ and $S'$ is labeled using various *dependence abstractions* or *dependence approximations*, depending on the dependence analysis and on the input needed by the loop parallelization algorithm. Except for affine dependences (see below), a dependence $S(I) \longrightarrow S'(J)$ is represented by an approximation of the *distance vector* $J - I$. If $S$ and $S'$ do not have the same domain, only the components of the vector $J - I$, that correspond to the $n_{S,S'}$ loops surrounding both statements, are defined. Classical representations of distance vectors (by increasing precision) are:

**Dependence level:** introduced by Allen and Kennedy in [1, 2]. A distance vector $J - I$ is approximated by an element $l$ (the *level*) in $[1, n_{S,S'}] \cup \{\infty\}$, defined as $\infty$ if $J - I = 0$, or as the largest integer such that the $l - 1$ first components of the distance vector are zero. When $l = \infty$, the dependence is said *loop independent*, and *loop carried* otherwise.

**Direction vector:** first described by Lamport in [28], then by Wolfe in [37]. A set of distance vectors between $S$ and $S'$ is represented by a $n_{S,S'}$-dimensional vector, called the *direction vector*, whose components belong to $\mathbb{Z} \cup \{*\} \cup (\mathbb{Z} \times \{+, -\})$. Its $i$-th component is an approximation of the $i$-th component of the distance vectors: $z+$ means $\geq z$, $z-$ means $\leq z$, and $*$ means any value.

**Dependence polyhedron:** introduced by Irigoin and Triolet [26]. A set of distance vectors between $S$ and $S'$ is approximated by a subset of $\mathbb{Z}^{n_{S,S'}}$, defined as the integral points of a polyhedron. This is an extension of the direction vector abstraction.

**Affine dependences:** used by Feautrier [18] to express dependence relations when exact dependence analysis is feasible. A set of dependences $S(I) \longrightarrow S'(J)$ can be represented by an affine function $f$ that expresses $I$ in terms of $J$ ($I = f(J)$) or the converse, subject to affine inequalities that restrict the range of validity of the dependence.

## 3.2 Loop transformations

We only focus here on the transformations that are captured by the loop parallelization algorithms presented in Section 3.3.

**Statement reordering:** the order of statements in a loop body is modified. Statement reordering is valid if and only if loop independent dependences are preserved.

**Loop distribution:** a loop, surrounding several statements, is split into several identical loops, each surrounding a subset of the original statements. The validity of loop distribution is related to the construction of the strongly connected components of the RDG (without considering dependences carried by an outer loop).

**Unimodular loop transformations:** a unimodular loop transformation is a change of basis (in $\mathbb{Z}^{n_S}$) applied on the iteration domain $D_S$. The computations are described through a new iteration vector $I' = UI$ where $U$ is an integral matrix of determinant 1 or $-1$. Unimodular loop transformations are combinations of loop interchange, loop reversal, and loop skewing. A unimodular transformation $U$ is valid if and only if $Ud \succ_{lex} 0$ for each non null distance vector $d$.

**Affine transformations:** a general affine transformation defines a new iteration vector $I'$ for each statement $S$ by an affine function $I' = M_S I + \rho_S$. $M_S$ is a non parameterized non singular square integral matrix of size $n_S$, and $\rho_S$ is a possibly parameterized vector. The linear part may be unimodular or not. Such a transformation is valid if and only if $S(I) \longrightarrow S'(J) \Rightarrow M_S I + \rho_S \prec_{lex} M_{S'} J + \rho_{S'}$.

**Tiling:** this transformation consists in rewriting a set of $n$ loops into $2n$ loops, by defining *tiles* of size $(t_1, \ldots, t_n)$: $I = (i_1, \ldots, i_n)$ is transformed into $I' = (i_1 \div t_1, \ldots, i_n \div t_n, i_1 \bmod t_1, \ldots, i_n \bmod t_n)$. A sufficient condition for tiling is that the $n$ original loops are fully permutable.

## 3.3 Parallelization algorithms

In the following, the optimality of an algorithm has to be understood with respect to the dependence abstraction it uses. For example, the fact that Allen and Kennedy's algorithm is

optimal for maximal parallelism detection means that a parallelization algorithm which takes as input the same information as Allen and Kennedy's algorithm, namely a representation of dependences by dependence level, cannot find more parallelism that Allen and Kennedy's algorithm does.

**Lamport's algorithm [28]** considers perfectly nested loops whose distance vectors are supposed to be uniform (constant) except for some fixed components. It produces a set of vectors, best known as "Lamport's hyperplanes", that form a unimodular matrix. Lamport proposed an extension of this algorithm to handle statement reordering, extension which can also schedule independently the left- and right-hand sides of assignments. Lamport's algorithm is related to linear schedules (see [12]) and to multi-dimensional schedules.

**Allen and Kennedy's algorithm [2]** is based on the decomposition of the reduced dependence graph into strongly connected components. It uses dependences represented by levels, and transforms programs by loop distribution and statement reordering. It is optimal for maximal parallelism detection (see [16]). The minimization of synchronizations is considered through loop fusion (see Section 6.1). However, it is not really adapted (because of the poor dependence abstraction) to the detection of outer parallelism and permutable loops.

**Wolf and Lam's algorithm [36]** is a reformulation of Lamport's algorithm to the case of direction vectors. It produces a unimodular transformation that reveals fully permutable loops in a set of perfectly nested loops. As a set of $d$ fully permutable loops can be rewritten as one sequential loop and $d-1$ parallel loops, it can also detect parallel loops. The dependence abstraction it uses is sharper than the one in Allen and Kennedy's algorithm. However, the structure of the RDG is not considered. It is optimal for maximal parallelism detection if the only information on direction vectors with no knowledge of the dependence graph structure.

**Feautrier's algorithm [19, 20]** produces a general affine transformation. It can handle perfectly nested loops as well as non perfectly nested loops as long as exact dependence analysis is feasible. It relies on affine dependences. The affine transformation is build as a solution of linear programs obtained by the affine form of Farkas' lemma [35] applied to dependence constraint equations. Although Feautrier's algorithm is the most powerful algorithm for detecting innermost parallelism in loops with affine dependences, it is not optimal since it turns out that affine transformations are not sufficient. Moreover, it is not adapted, a priori, to the detection of outer parallelism and permutable loops.

**Darte and Vivien's algorithm [14]** is a simplification of Feautrier's algorithm to the case of dependences represented by dependence polyhedra (an example being direction vectors). It also produces an affine transformation, but of a restricted form, called shifter linear schedule (see Section 5.1). It is optimal for maximal parallelism detection if dependences are approximated by dependence polyhedra. Since it is simpler than Feautrier's algorithm, more optimizing criteria can be handled: the detection of permutable loops and outer parallelism

(see [13]), and the minimization of synchronizations through loop fusion (see Section 6.2). Furthermore, the code generation is simpler (see Section 5). However, it may find less parallelism than Feautrier's algorithm when exact dependence analysis is feasible because of its restricted choice of transformations.

**Lim and Lam [30]** is an extension of Feautrier's algorithm whose goal is to detect fully permutable loops and outer parallel loops. As Feautrier's algorithm, it relies on a description of dependences as affine dependences. It uses the affine form of Farkas' lemma and the Fourier-Motzkin elimination. Lim and Lam's algorithm has the same qualities and weaknesses as Feautrier's algorithm: it is, in theory, very powerful, but no guarantee is given concerning the easiness of code generation. Indeed, many solutions are equivalent relatively to the criteria they optimize: choosing the simplest solution is not explained in Lim and Lam's algorithm, and code generation is not addressed.

# 4 Code generation

Once the program has been analyzed and some loop transformation has been found, it remains to generate the code corresponding to the transformed program. In the current section, we review the techniques that currently exist to handle this last problem. We go from the simplest transformations to the most complicated ones. We skip in the discussion loop distribution and statement reordering for which code generation is straightforward.

## 4.1 Unimodular transformations

Unimodular transformations apply to perfect loop nests whose iteration domains are convex polyhedra. They are important for code generation because they guarantee that, if the original iteration domain is a convex polyhedron, the iteration domain of the transformed loop nest is also a convex polyhedron. It means that the code generation problem simplifies to lexicographically scanning the integer points of a convex polyhedron.

The other part of the code generation is to express the array access functions with respect to the new loop indices. Since a unimodular transformation is invertible (with integral inverse), it is easy to express the array access functions with respect to the new loop indices. Let $U$ be the matrix of the unimodular transformation, $I$ the iteration vector of the original loop nest, $I' = UI$ the iteration vector of the transformed loop nest, we just have to replace everywhere in the loop nest body $I$ by $U^{-1}I'$.

We present now the two classical approaches for the polyhedron scanning problem: the Fourier-Motzkin pairwise elimination and the simplex algorithm.

**Fourier-Motzkin elimination.** Ancourt and Irigoin first proposed this technique in [3] and it has then been used in several prototype compilers [25, 33, 22]. The idea is to use a projection algorithm to find the loop bounds for each dimension. The polyhedron is represented as usually by a system of inequalities. At each step of the Fourier-Motzkin pairwise elimination algorithm, some inequalities are added to the system to build a "triangular"

| Algorithms | Dependence abstraction | Loop transformations | Maximal degree of // | Synchro. Fusion | Code generation | Tiling |
|---|---|---|---|---|---|---|
| **Allen-Kennedy [2]** | Dependence level<br>Multiple statements<br>Non perfect | Distribution | Optimal | Yes | Very easy | No |
| **Wolf-Lam [36]** | Direction vectors<br>One statement<br>Perfect | Unimodular | Optimal | No | Easy | Yes |
| **Darte-Vivien [14]** | Polyhedra<br>Multiple statements<br>Perfect | Shifted linear | Optimal | Partial yes | Quite easy | Yes |
| **Feautrier [20]** | Affine (exact)<br>Multiple statements<br>Non perfect | Affine | Sub-optimal | No | Complicated | No |
| **Lim-Lam [30]** | Affine (exact)<br>Multiple statements<br>Non perfect | Affine | Sub-optimal | ? | ? | Yes |

Table 1: A comparison of various loop parallelizing algorithms.

system were each loop index depends only on the previous loop indices and on parameters. As many inequalities can define a loop bound, we have to take the maximum of the lower loop bounds and the minimum of the upper loop bounds. Let us take an example: a square domain transformed by combination of loop skewing and loop interchange $\binom{i'_1}{i'_2} = \left(\begin{smallmatrix}1 & 1\\ 1 & 0\end{smallmatrix}\right)\binom{i_1}{i_2}$ (see Figure 6). The system describing the transformed polyhedron and its transformation by the Fourier-Motzkin elimination of $i'_2$ are:

$$\begin{cases} 1 \le i'_2 \le n \\ 1 \le i'_1 - i'_2 \le n \end{cases} \xrightarrow[\text{elimination of } i'_2]{\text{Fourier-Motzkin}} \begin{cases} 2 \le i'_1 \le 2n \\ 1 \le i'_2 \le n \\ i'_1 - n \le i'_2 \le i'_1 - 1 \end{cases}$$

```
do i1 = 1, n                                   do i'1 = 2, 2n
    do i2 = 1, n        (i'1 i'2)=(1 1 1 0)(i1 i2)      do i'2 = max(1, i'1 - n), min(n, i'1 - 1)
        ...             ------------------->               ...
    enddo                                          enddo
enddo                                          enddo
```

Figure 6: Unimodular example.

The main drawback of this algorithm is that it can generate redundant inequalities. Their elimination requires a powerful test also based on the Fourier-Motzkin elimination or on the simplex algorithm [22]. If some redundant inequalities are not removed, some iterations may be empty in the resulting loop nest, causing overhead. The better the elimination, the fewer empty iterations. Although the Fourier-Motzkin elimination has super-exponential complexity for big problems, it remains fast for small problems, and works well in practice.

**Simplex algorithm.** The second approach to compute the loop bounds uses an extended version of the simplex algorithm: indeed one has to be able to solve parametric integer linear problems in rational numbers. This method has been proposed by Collard, Feautrier and Risset in [11] and has been used in at least three experimental parallelizers [6, 11, 24].

The basic idea is to build a polyhedra $D_k$ for each loop index $i_k$ in which outer indices are considered as parameters and to search for the extrema of $i_k$ in $D_k$ so as to find the loop bounds. It has been shown in [11] that this resolution can be done using PIP [17], a parametric dual simplex implementation, and that the result is expressed as the ceiling of the maximum of affine expressions for the lower bound and the floor of the minimum of affine expressions for the upper one. On the example of Figure 6, the result is the same.

This algorithm produces no empty iterations but may introduce floor and ceiling operations. The complexity of the simplex algorithm is exponential in the worst case but polynomial on the average and so also works well in practice. Chamski [9] addresses the problem of control overhead by replacing extrema operations by conditionals at the expense of code duplication.

## 4.2 Non-unimodular linear transformations

When dealing with non-unimodular transformations, the classical approach [29] is to decompose the transformation matrix into its Hermite normal form [35] to get back to the unimodular case. An algorithm based on column operations on an integral nonsingular matrix $T$ transforms it into the product $HU$ $(= T)$ where $U$ is unimodular and $H$ is a nonsingular, lower triangular, nonnegative matrix, in which each row has a unique maximum entry, which is its diagonal entry. Once the transformation $U$ has been considered, it is easy to handle the matrix $H$: each diagonal element corresponds to a multiplication of the loop counter and can be coded with steps in the loop indices, and the other non-zero entries are shifts. Figure 7 shows the example of the transformation of a 2-D loop nest by $\left(\begin{smallmatrix} 2 & 0 \\ 1 & 3 \end{smallmatrix}\right)$.

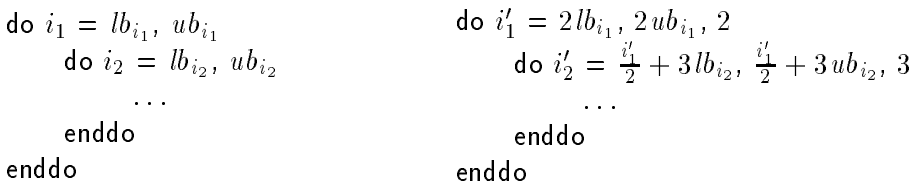



Figure 7: Non-unimodular example.

Xue presented in [39] another method to deal with non-unimodular transformations. It is based on Fourier-Motzkin elimination to compute the loop bounds and on Hermite decomposition to compute the steps and shifting constants.

## 4.3 Extensions

**Perfect loop nests with one-dimensional shifted linear schedules.** In [6], Boulet and Dion explain how to deal with affine transformations which share the same linear part and shift the first transformed loop index with constants, one constant for each statement in the body of the original loop nest. Moreover, they handle the case of rational schedules, transformations whose first dimension may have rational entries. The basic idea is to decompose the transformation into a unimodular one and then handle the rational part and the shifting constants with a two-dimensional time combined with loop splitting to avoid control overhead.

**Non perfect loop nests with one-dimensional schedules.** Collard presents in [10] a method to produce code when each statement of a non perfect loop nest has been assigned an affine one-dimensional schedule. His scheme is to build a global outer time loop and to add guards around the statements to compute them only when necessary. The goal is to reduce control overhead.

**General affine case.** Kelly, Pugh and Rosset present in [27] a method to generate code for the general affine case, that is a non perfect loop nest transformed with a possibly different affine transformation for each statement. Their transformation is based on Presburger

arithmetic implemented in the Omega library [34]. They also address the problem of control overhead versus code duplication. The problem is to be able to scan an arbitrary union of polyhedra. In the next section, we present a simpler code generation scheme for the case of shifted linear schedules.

# 5 Code generation for shifted linear schedules

General affine schedules are necessary to capture and manipulate non perfectly nested loops. However, applying such a transformation is not straightforward in the general case, and may lead to complex codes with nasty guards, loop steps, loop bounds and array access functions, as illustrated by the following example.

**Example 1:** Consider two statements $S$ and $S'$ with the same iteration domain $\{1 \leq i, j, k \leq N\}$. Assume that $S(i,j,k)$ and $S'(i,j,k)$ are mapped, using a multi-dimensional schedule, respectively to $(i-j, i+j, i+j+2k)$ and $(i-j+1, i+j+2, k)$. A possibility is to generate the code of Figure 8. Notice that this code is already optimized: a more naive approach would have kept both $S$ and $S'$ inside the same $t_3$ loop and all conditionals in this innermost loop. Furthermore, some natural conditionals such as $t_1 \leq n-1$ for statement $S'$ have been removed since they are redundant.

```
do t1 = -n + 1, n
  do t2 = max(t1 + 2, -t1 + 2), min(t1 + 2n + 1, -t1 + 2n + 3)
    if t1 + t2 mod 2 and t1 + t2 ≤ 2n
      do t3 = t2 + 2, t2 + 2n, 2
        S((t1 + t2)/2, (t2 - t1)/2, (t3 - t2)/2)
      enddo
    endif
    if t1 + t2 + 1 mod 2 and t2 + t1 ≥ 5
      do t3 = 1, n
        S'((t1 + t2 - 3)/2, (t2 - t1 - 1)/2, t3)
      enddo
    endif
  enddo
enddo
```

Figure 8: A code with modulos.

The complexity of this type of code generation is mainly due to the two following facts:

- In a non perfect loop nest, the statements may have different iteration domains. Therefore, in order to schedule them simultaneously and to write the corresponding code, we need to be able to scan (i.e. describe by loops) an arbitrary union of polyhedra, even if all statements have the same multi-dimensional schedule.

- A multi-dimensional affine schedule can be viewed as a change of basis applied to the iteration domain. Therefore, as soon as the schedules of two statements are different, we need once again to be able to scan an union of polyhedra, even if all statements have the same iteration domain.

The goal of this section is to present a new code generation scheme that leads to clean, simple, and easily understandable codes. This is made possible because we restrict the set of schedules that we consider, so as to circumvent the two difficulties stated above. We consider only particular multi-dimensional affine schedules, called *multi-dimensional shifted linear* schedules. We show that if all iteration domains are the same in the original code, up to a translation (a typical example is when the original code is perfectly nested), then the code generation for such schedules is a lot simpler, and leads to cleaner codes. Indeed, the code generation can be seen as a hierarchical combination of loop distribution, loop bumping (i.e. adding a constant to a loop counter), and matrix transformation, where each statement can be considered independently (thus avoiding the complicated problem of overlapping different polyhedra).

Furthermore, we show that, given a shifted linear schedule $\sigma$, it is possible to build an equivalent shifted linear schedule $\sigma'$, equivalent in the sense that the nature of the loops (sequential, parallel or permutable) in the transformed code is preserved, and such that the code generation for $\sigma'$ involves only unimodular transformations and loop bumping.

## 5.1 Shifted linear schedules

We use the following notations. If $M$ is a matrix, $[M]_k$ denotes the $k$-th row of $M$, $[M][k]$ denotes the matrix whose rows are the first $k$ rows of $M$, and $[M][i,j]$ denotes the square sub-matrix of $M$, intersection of the rows and columns of $M$ from $i$ to $j$.

To make the discussion simpler, we consider that all iteration domains have the same dimension $n$ (i.e. for any statement $S$, $n_S = n$) so that all iteration vectors and all matrices have the same size (otherwise we can complete the iteration vectors with ending 0). As recalled in Section 3.2, a multi-dimensional affine schedule $\sigma$ is defined for each statement $S$ by an integral square nonsingular matrix $M_S$ of size $n$ and an integral vector $\rho_S$ of size $n$. We write $\sigma = (M_S, \rho_S)$. The computation $S(I)$ associated to the iteration vector $I$ before transformation is associated to the iteration vector $M_S I + \rho_S$ after transformation. A multi-dimensional function $\sigma = (M_S, \rho_S)$ is a valid schedule if and only if:

$$S(I) \longrightarrow S'(J) \Rightarrow M_S I + \rho_S \prec_{lex} M_{S'} J + \rho_{S'} \tag{1}$$

We say that a dependence $S(I) \longrightarrow S'(J)$ is *satisfied by $\sigma$ at level $k$* if:

$$[M_S I + \rho_S][k-1] = [M_{S'} J + \rho_{S'}][k-1] \text{ and } [M_S I + \rho_S]_k < [M_{S'} J + \rho_{S'}]_k$$

Equation 1 guarantees that $k$ is always well-defined: any dependence is satisfied at some level $k$, and for a unique $k$. We denote by $k_{S,S'}$ the maximal level at which some dependence between $S$ and $S'$ is satisfied, and by $c_{S,S'}$ the maximal level $c$ such that $[M_S][c] = [M_{S'}][c]$.

**Definition 1.** A multi-dimensional affine schedule $\sigma = (M_S, \rho_S)$ is *shifted linear* if, for any statements $S$ and $S'$, $[M_S][k_{S,S'}] = [M_{S'}][k_{S,S'}]$, i.e. if $k_{S,S'} \leq c_{S,S'}$.

In other words, a multi-dimensional affine schedule is shifted linear if $S$ and $S'$ have the same linear part for the outermost levels as long as there exists a dependence between $S$ and $S'$ not yet satisfied. Note that, as recalled in Section 3.3, looking for such schedules is not penalizing if dependences are approximated by a polyhedral approximation of distance vectors (for example direction vectors) and if the main objective is the detection of the maximal degree of parallelism.

## 5.2 Code generation scheme

Remember the code generation scheme for a single statement $S$. If the matrix $M_S$ is not unimodular, we use the Hermite form $M_S = H_S U_S$ where $U_S$ is unimodular, $H_S$ non negative, lower triangular, and each non diagonal component of $H_S$ is strictly smaller than the diagonal component of same row. Then, we transform the code first using $U_S$, then using the loop skewing $H_S$. Here, we have multiple statements, different matrices $M_S$, and different constants $\rho_S$, therefore we must apply a different unimodular transformation, a different loop skewing, and a different loop bumping for each statement. Fortunately, by construction of the Hermite form [35], we have:

$$[M_S][k] = [M_{S'}][k] \quad \Rightarrow \quad [H_S][k] = [H_{S'}][k] \text{ and } [U_S][k] = [U_{S'}][k]$$

Therefore, while all dependences between two statements $S$ and $S'$ are not satisfied, all loops that surround $S$ and $S'$ are the same (up to a constant): we just have to generate the codes for $S$ and $S'$ separately, and to fuse the two codes into a single one until level $k_{S,S'}$. Then, for the remaining dimensions, since there are no more dependences between $S$ and $S'$, the two codes do not need to be perfectly matched: one can just write them one above the other, and the resulting code remains correct. In other words, in this restricted case, there is no need for a complicated algorithm for scanning a union of polyhedra.

The code generation process is the following. We write $M_S = H_S U_S$ and $\rho_S = H_S q_S + r_S$ where $q_S$ and $r_S$ are integral vectors where each component of $r_S$ is non negative and strictly smaller than the corresponding diagonal element of $H_S$ (this decomposition is unique). Then, we decompose the transformation $\sigma_S : I \to M_S I + \rho_S$ in four steps, $\sigma_S^1 : I \to U_S I$, $\sigma_S^2 : I \to I + q_S$, $\sigma_S^3 : I \to H_S I$, $\sigma_S^4 : I \to I + r_S$: $\sigma_S^1$ is a unimodular transformation, $\sigma_S^2$ is a loop bumping, $\sigma_S^3$ is a loop skewing, and $\sigma_S^4$ consists simply in writing the code in the $r_S$-th position in the loop body (because the code at this point is a code with loop steps). Note that when $H_S$ is diagonal, there is no need to really multiply the counter by the diagonal component: we can keep the original counter and avoid steps and floor functions. We will use this technique in Section 5.3.

**Back to Example 1:** We find the two unimodular transformations $U_S : (i,j,k) \to (i - j, j, j + k)$ and $U_{S'} : (i,j,k) \to (i - j, j, k)$, and the two skewing transformations $H_S : (i,j,k) \to (i, i+2j, i+2k)$ and $H_{S'} : (i,j,k) \to (i, i+2j, k)$. Furthermore $q_S = r_S = (0,0,0)$, and $q_{S'} = (1,0,0)$, $r_{S'} = (0,1,0)$. We find the two intermediate codes of Figure 9 by applying the transformations $U_S$ and $U_{S'}$, and the loop bumping by $q_S$ and $q_{S'}$.

Then, we apply the loop skewings $H_S$ and $H_{S'}$, and the displacements by $r_S$ and $r_{S'}$. We get the two codes of Figure 10. The displacements are visualized by `nop` operations. Finally,

```
do t1 = −n + 1, n − 1
   do t2 = max(1, −t1 + 1),
           min(n, −t1 + n)
     do t3 = t2 + 1, t2 + n
       S(t1 + t2, t2, t3 − t2)
     enddo
   enddo
enddo
```

```
do t1 = −n + 2, n
   do t2 = max(1, −t1 + 2)
           min(n, −t1 + n + 1)
     do t3 = 1, n
       S′(t1 + t2 − 1, t2, t3)
     enddo
   enddo
enddo
```

Figure 9: Separate codes after unimodular transformation.

```
do t1 = −n + 1, n − 1
   do t2 = max(t1 + 2, −t1 + 2),
           min(t1 + 2n, −t1 + 2n), 2
     do t3 = t2 + 2, t2 + 2n, 2
       S((t1+t2)/2, (t2−t1)/2, (t3−t2)/2)
     enddo
     nop
   enddo
enddo
```

```
do t1 = −n + 2, n
   do t2 = max(t1 + 2, −t1 + 4),
           min(t1 + 2n, −t1 + 2n + 2), 2
     nop
     do t3 = 1, n
       S′((t1+t2)/2 − 1, (t2−t1)/2, t3)
     enddo
   enddo
enddo
```

Figure 10: Separate codes after loop skewing.

merging the two codes, we get the code of Figure 11. Loop peeling could be used to remove as many conditionals as possible if this turns out to be more efficient. Notice also that the two conditionals $t_1 \leq n-1$ and $t_2 \geq -n+2$ could be removed since they are redundant with the other constraints.

## 5.3 Equivalent schedules and unimodularity

In terms of parallelism extraction, multi-dimensional schedules may be used for detecting either *parallel* and *sequential* loops, or *permutable* loops as a first step before tiling. We say that two schedules $\sigma$ and $\sigma'$ are *equivalent* if, in both transformed codes, the nature of the loops (parallel or sequential in the first case, permutable in the second case) is the same.

Let us analyze how dependences in the original code are transformed if we use the code generation process described in Section 5.2. If $S(I) \longrightarrow S'(J)$ is satisfied at level $k$:

$$\begin{array}{l} [U_S I + q_S]\,[k-1] = [U_{S'} J + q_{S'}]\,[k-1] \text{ and } [r_S]\,[k-1] = [r_{S'}]\,[k-1] \\ [U_S I + q_S]_k \leq [U_{S'} J + q_{S'}]_k \\ [U_S I + q_S]_k = [U_{S'} J + q_{S'}]_k \Rightarrow [r_S]_k < [r_{S'}]_k \end{array} \quad (2)$$

In other words, a dependence satisfied at level $k$ is either *loop carried* at level $k$, when $[U_S I + q_S]_k < [U_{S'} J + q_{S'}]_k$, or *loop independent* at level $k$, and in this latter case $r_S < r_{S'}$. A loop at level $k$ is then parallel in the transformed code, if there is no dependence carried at level $k$ between any two statements surrounded by this loop. We point out that the nature

```
do t1 = -n + 1, n
  do t2 = max(t1 + 2, -t1 + 2), min(t1 + 2n, -t1 + 2n + 2), 2
    if t1 ≤ n - 1 and t2 ≤ -t1 + 2n
      do t3 = t2 + 2, t2 + 2n, 2
        S((t1 + t2)/2, (t2 - t1)/2, (t3 - t2)/2)
      enddo
    endif
    if t1 ≥ -n + 2 and t2 ≥ -t1 + 4
      do t3 = 1, n
        S'((t1 + t2)/2 - 1, (t2 - t1)/2, t3)
      enddo
    endif
  enddo
enddo
```

Figure 11: Combination of the two codes.

of a dependence (loop carried or loop independent) is not fully specified by the schedule $\sigma = (M_S, \rho_S)$ itself, but depends on the way we write the code. For example, handling the constants $\rho_S$ differently may change the nature of a dependence (but not the level at which it is satisfied). Therefore, the equivalence of two schedules has to be understood with respect to the code generation we use. We have the following result:

**Theorem 1.** *For any shifted linear schedule $\sigma = (M_S, \rho_S)$, there exists a shifted linear schedule $\sigma' = (M'_S, \rho'_S)$, equivalent for parallel and sequential loops, and such that $M'_S = H'_S U'_S$ where $H'_S$ is non-negative diagonal and $U'_S$ unimodular.*

*Proof.* See the extended proof in the appendix A. We build $\sigma'$ as follows. We write $H_S = K_S H'_S$ where $H'_S$ is the diagonal matrix such that $H'_S$ and $H_S$ have the same diagonal. Then $\sigma' = (K_S^{-1} M_S, K_S^{-1} H_S q_S + r_S) = (H'_S U_S, H'_S q_S + r_S)$ is a shifted linear schedule equivalent for parallel and sequential loops. □

We now consider schedules used for detecting maximal blocks of permutable loops. A maximal block of nested loops, from level $i$ to level $j$, is permutable in the transformed code if for all statements $S$ and $S'$ surrounded by these loops, for any dependence $S(I) \longrightarrow S'(J)$ satisfied at a level between $i$ and $j$, the dependence distance is non negative, i.e.:

$$[M_S I + \rho_S][i-1] = [M_{S'} J + \rho_{S'}][i-1] \Rightarrow [M_S I + \rho_S][j] \leq [M_{S'} J + \rho_{S'}][j] \qquad (3)$$

Once again, since we address only shifted linear schedules, we consider only blocks, surrounding statements $S$ and $S'$, whose maximal level $j$ is smaller than $c_{S,S'}$.

**Theorem 2.** *For any shifted linear schedule $\sigma = (M_S, \rho_S)$, there exists a shifted linear schedule $\sigma' = (M'_S, \rho'_S)$, equivalent for permutable loops, and such that $M'_S = H'_S U'_S$ where $U'_S$ is unimodular and $H'_S$ is positive diagonal, with all entries equal to 1 except possibly for each last level of a block of permutable loops containing $S$.*

*Proof.* The technique is to define, for each statement $S$, a well chosen loop skewing $G_S$ (see the construction in the appendix B) such that $G_S M_S = H'_S U'_S$. Then $\sigma' = (G_S M_S, \lfloor G_S \rho_S \rfloor)$ is a shifted linear schedule equivalent for permutable loops. □

When generating code for revealing permutable loops, we may want that permutable loops are perfectly nested. This is not the case with the code generation scheme proposed in Section 5.2 because of the constants $r_S$. Each time two statements have different values of $r_S$ for the same loop, the resulting code is non perfectly nested (except of course at the innermost level). Therefore, for general affine schedules, we may need to enforce loops to be perfectly nested by not decomposing the constants $\rho_S$ into $q_S$ and $r_S$. However, the resulting code would be much more complicated. This is the reason why we impose in Theorem 2 that the components of $H'_S$ are equal to 1, except for the last level of a block of permutable loops. Then, the code is easy to generate for the outermost block. To make sure that it is also simple for inner blocks, we impose that $[r_S]\,[i_{S,S'} - 1] = [r_{S'}]\,[i_{S,S'} - 1]$ where $i_{S,S'}$ is the first level of the innermost block of permutable loops surrounding $S$ and $S'$. Fortunately, this technical condition is true for shifted linear schedules that are built by the algorithm proposed in [13] for which we developed these simplification techniques.

**Back to Example 1:** We assume that the first two dimensions correspond to a block of permutable loops. Following the proof of Theorem 2, we find the two loop skewing transformations $G_S : (i,j,k) \to (i, i+j, -i+k)$ and $G_{S'} : i,j,k \to (i, i+j, k)$ which lead to the schedule $(i-j, 2i, 2j+2k)$ for $S$ and $(i-j+1, 2i+3, k)$ for $S'$. We get the final equivalent code of Figure 12 which is quite simpler.

```
do t1 = -n + 1, n
   do t2 = max(t1 + 1, 1), min(t1 + n, n + 1)
     if t1 ≤ n − 1 and t2 ≤ n
       do t3 = t2 − t1 + 1, t2 − t1 + n
         S(t2, t2 − t1, t3 − t2 + t1)
       enddo
     endif
     if t1 ≥ −n + 2 and t2 ≥ 2
       do t3 = 1, n
         S'(t2 − 1, t2 − t1, t3)
       enddo
     endif
   enddo
enddo
```

Figure 12: Equivalent code for Example 1.

It is shown in the appendices A and B that the nature of each loop (not only permutable, but also parallel and sequential) is preserved. As noticed in Section 5.2, all loop steps are unit steps, and there is no use of floor or ceiling functions, even if the transformation is

non unimodular. This is because the loop skewings (the Hermite forms of the schedule) are diagonal, and because there is no need to really multiply the loop counter by the diagonal component. This demonstrates that shifted linear schedules have nice properties, for maximal parallelism detection as well as for code generation.

# 6 Reducing the number of synchronizations

In this section, we recall how the fusion of parallel loops is handled in Allen and Kennedy's algorithm so as to reduce the number of synchronizations (see [8]). We show that the problem becomes much more difficult if loop bumping and loop fusion are combined. We show the NP-completeness of the problem, even in the simple case of uniform dependences, and we propose an integer linear programming method to solve it.

## 6.1 Fusion of parallel loops

Consider a piece of code only composed of parallel loops. Consider the dependences that take place *inside* the code (in other words, if the code is surrounded by some loops, do not consider dependences carried by these loops), and in particular dependences between different loops (inter-dependences). The fusion of two parallel loops is valid and gives one parallel loop if there is no inter-dependence between these two loops, or if all inter-dependences become loop independent after fusion. Otherwise, the semantics of the code is not preserved or the loop produced is sequential. An inter-dependence that is not loop independent after fusion is called *fusion preventing*.

The technique to minimize the number of parallel loops after fusion is the following. The goal is to assign to each statement $S$ a nonnegative integer $p(S)$ that indicates which parallel loop contains $S$ after fusion. Let $e$ be a dependence from statement $S$ to statement $S'$. Then, after fusion, the loop containing $S$ must appear, in the new loop nest, before the loop containing $S'$: $p(S) \leq p(S')$. Furthermore, if this dependence is fusion preventing, $S$ and $S'$ cannot be in the same loop after fusion: $p(S) + 1 \leq p(S')$. To minimize the total number of parallel loops after fusion, we just have to minimize the label of the last loop, $\max_S p(S)$. To obtain the desired loop nest, we place in the same parallel loop all statements with the same value $p$, and parallel loops are ordered by increasing $p$. The formulation is thus:

$$\begin{cases} \max_S p(S) \text{ s.t. } p(S) \geq 0 \text{ and} \\ \forall\ S \xrightarrow{e} S' \text{ s.t. } e \text{ is not fusion preventing}, & p(S) \leq p(S') \\ \forall\ S \xrightarrow{e} S' \text{ s.t. } e \text{ is fusion preventing}, & p(S) + 1 \leq p(S') \end{cases}$$

The reader can recognize a classical scheduling problem: $p(S)$ is the maximal weight of a dependence path ending in $S$, where the weight of an edge is defined as 1 if the edge is fusion preventing, and 0 otherwise. Therefore, a greedy algorithm is optimal and polynomial.

An extension of this technique has been proposed in [31] to handle both the fusion of parallel loops and the fusion of sequential loops. It consists in two steps. First the fusion of parallel loops is performed as above, except that additional fusion preventing edges are added each time there is a dependence path between two statements that goes through a

sequential loop. Then, the similar technique is used for sequential loops. As noticed by Mc Kinley and Kennedy, the total number of loops may not be minimal, but the number of parallel loops is and, therefore, the number of synchronizations.

## 6.2 Fusion of parallel loops and shifted linear schedules

We now consider the particular case of the generation of parallel loops with shifted linear schedules (see Section 5.1). We suppose that $k$ loops have already been generated, and that all the dependences are satisfied by these loops, except some dependences that form an acyclic graph $G_a$.

The code generation technique proposed in Section 5.2 would generate the $n-k$ remaining loops, by placing each statement in a separate set of nested parallel loops so that the dependences of the acyclic graph are satisfied at level $k$ as loop independent. Here, we want to do better. We want to generate as few parallel loops as possible and no sequential loops, in order to have, once again, the maximal parallelism while minimizing synchronizations.

We consider the case where one loop remains to be built ($k = n - 1$): the general case is similar if we decide that two statements share all or none of their surrounding parallel loops. Once again, we try to place in the same parallel loop only statements for which the schedule is defined by the same linear part (shifted linear schedule). Practically, we are given a vector $X$ that will be used to generate the last loop, and we try to generate constants $\rho_S$ so as to fully define the schedule. If $S$ and $S'$ are to be placed in the same parallel loop, we must find two constants $\rho_S$ and $\rho_{S'}$ such that, for each dependence $e : S(I) \rightarrow S'(J)$, $X(J - I) + \rho_{S'} - \rho_S = 0$ so that the dependence becomes loop independent. To make the link with Section 6.1, here we try to fuse more parallel loops using in addition loop bumping. This gives more freedom, but makes the optimal solution more difficult to find.

We assume that the dependences in $G_a$ are uniform. We denote by $w(e)$ the quantity $X(J - I)$ associated with the edge $e$. Remark that when $G_a$ is acyclic, if considered as an undirected graph, one can always choose the constants $\rho_S$ so that all statements can be placed in the same parallel loop. On the other hand, if $G_a$ has an (undirected) cycle, this may not be possible. Indeed, consider an undirected cycle in $G_a$, $C = \sum_{e \in C} \delta_e e$ where $\delta_e \in \{-1, 1\}$, i.e. a cycle that can use an edge backwards ($\delta_e = -1$) or forwards ($\delta_e = 1$). Define the weight of $C$ as $w(C) = \sum_{e \in C} \delta_e w(e)$. If all dependences of $C$ are transformed into loop independent dependences, then $w(C) = 0$. Conversely, if $w(C) \neq 0$, then for at least one edge $e = (S_e, S'_e)$ of the cycle, $S'_e$ has to be placed in a parallel loop after the parallel loop that surrounds $S_e$. This remark leads to the following integer linear program

$$\begin{cases} \max_S p(S) \text{ s.t. } p(S) \geq 0 \text{ and} \\ \forall\ e = (S_e, S'_e),\ p(S_e) \leq p(S'_e) \\ w(C) \neq 0 \Rightarrow \sum_{e \in C} p(S'_e) \geq 1 + \sum_{e \in C} p(S_e) \\ \text{for each undirected elementary cycle } C. \end{cases}$$

that solves the problem: $p(S)$ is the label of the parallel loop in which $S$ should be placed. Indeed, by construction, the subgraph $G'_a$ of $G_a$ formed by the edges $e = (S_e, S'_e)$ for which $p(S_e) = p(S'_e)$ only contains undirected cycles $C$ such that $w(C) = 0$. Therefore, one can build the desired constants $\rho_S$ such that for all edge $e \in G'_a$, $w(e) + \rho_{S'_e} - \rho_{S_e} = 0$.

We point out that solving the linear program above is exponential for two reasons: first, the number of undirected elementary cycles can be exponential, and second, we use integer linear programming. Nevertheless, in practice, $G_a$ is usually very small, thus the program is solvable in reasonable time. However, in theory, the problem is NP-complete, as stated by the following theorem.

**Theorem 3.** *Let $G_a$ be an acyclic directed graph where each edge $e$ has a weight $w(e) \in \mathbb{Z}$. Given an integer $\rho_S$ for each vertex $S$, we define the quantity $w_\rho(e)$ for each edge $e = (S_e, S'_e)$ by $w_\rho(e) = 0$ if $w(e) + \rho_{S'_e} - \rho_{S_e} = 0$, and $w_\rho(e) = 1$ otherwise. The weight of a path is defined as the sum of the weights $w_\rho(e)$ of its edges. Then, finding values for $\rho_S$ which minimize the maximal weight of a path in $G_a$ is NP-complete.*

*Proof.* The proof is by reduction of the fusion problem from the UET-UCT scheduling problem (Unitary Execution Time - Unitary Communication Time), see the appendix. □

We illustrate the methods described in Sections 6.1 and 6.2 with the program of Figure 13. Because of the non zero dependences the simple fusion builds three different parallel loops (see Figure 14) when the fusion with loop bumping only builds two parallel loops.

```
do i = 1, n
    a[i] = 1
    b[i] = 1
    c[i] = a[i − 1] + b[i]
    d[i] = a[i] + b[i − 1]
    e[i] = d[i] + d[i − 1]
enddo
```

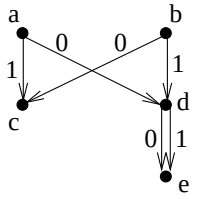



Figure 13: Original code and its dependence graph.

```
dopar i = 1, n
    a[i] = 1
    b[i] = 1
enddo
dopar i = 1, n
    c[i] = a[i − 1] + b[i]
    d[i] = a[i] + b[i − 1]
enddo
dopar i = 1, n
    e[i] = d[i] + d[i − 1]
enddo
```

```
dopar i = 1, n + 1
    if i ≤ n then a[i] = 1
    if 1 < i then b[i − 1] = 1
    if i ≤ n then d[i] = a[i] + b[i − 1]
enddo
dopar i = 1, n
    c[i] = a[i − 1] + b[i]
    e[i] = d[i] + d[i − 1]
enddo
```

Figure 14: Optimal solution for simple fusion, and for fusion with loop bumping.

# 7 Conclusion

We have proposed a comparative study of loop parallelization algorithms, insisting on the program transformations they produce, on the code generation scheme they need, and on their capabilities to incorporate various optimization criteria such as the detection of parallel loops, the detection of permutable loops, the minimization of synchronizations through loop fusion, and the easiness of code generation.

The simplest algorithm (Allen and Kennedy's algorithm) is of course not able to detect as much parallelism as the most complex algorithm (Feautrier's algorithm and its variants or extensions). However, the code generation it involves is straightforward and sharp optimizations such as the maximal fusion of parallel loops can be taken into account. For more complex algorithms, the loop transformations are obtained as solutions of linear programs, minimizing one criterion: no guarantee is given concerning the simplicity of the solution, or its quality with respect to a second optimization criterion. In other words, for complex algorithms, it remains to demonstrate that generating a "clean" solution is feasible. We gave some hints in this direction. We showed that, for algorithms based on shifted linear schedules, code generation is guaranteed to be simple, and that loop fusion can be handled (even if it can be expensive in theory).

A fundamental problem remains to be solved in the future: the link between parallelism detection and data mapping. Indeed, a parallel loop can be efficiently executed only if an adequate data mapping is proposed. This question is related to complex problems such as automatic alignment and distribution, scalar and array privatization, duplication of computations, etc.

# A Parallel loops and shifted linear schedules

We first prove some properties of loop skewing related to parallel and sequential loops.

**Lemma 1.** *Let $\sigma = (M_S, \rho_S)$ be a shifted linear schedule, where $M_S = H_S U_S$, $\rho_S = H_S q_S + r_S$. Suppose given, for each statement $S$, a rational matrix $G_S$, lower triangular, with diagonal greater than 1, such that $G_S M_S$ is integral and for any statements $S$ and $S'$, $[G_S]\,[k_{S,S'}] = [G_{S'}]\,[k_{S,S'}]$. Then, $\sigma' = (G_S M_S, G_S H_S q_S + r_S)$ is a shifted linear schedule, and the level at which a dependence is satisfied and its nature are the same for $\sigma$ and for $\sigma'$.*

*Proof.* Note first that $\sigma'$ is integral since $G_S H_S$ is integral if and only if $G_S M_S$ is integral. Furthermore, $[G_S M_S]\,[k_{S,S'}] = [G_{S'} M_{S'}]\,[k_{S,S'}]$. Let us compute the Hermite form of $G_S M_S$. We have $G_S M_S = G_S H_S U_S$, and $G_S H_S$ is lower triangular, but not necessarily non negative. Actually, the Hermite form of $G_S H_S$ is $G_S H_S = H'_S T_S$ where $T_S$ is lower triangular (since $G_S H_S$ is lower triangular) and unimodular. Furthermore, by construction of the Hermite form, it can be checked that $[T_S]\,[k_{S,S'}] = [T_{S'}]\,[k_{S,S'}]$. Thus the Hermite form of $G_S M_S$ is $G_S M_S = H'_S U'_S$ where $H'_S = G_S H_S T_S^{-1}$ and $U'_S = T_S U_S$. Finally, since the diagonal of $G_S$ is greater than 1, $G_S H_S q_S + r_S = H'_S T_S q_S + r_S$ is the desired unique decomposition of $G_S H_S q_S + r_S$ into $H'_S q'_S + r'_S$ with $q'_S = T_S q_S$ and $r'_S = r_S$.

Consider now a dependence $S(I) \longrightarrow S'(J)$ satisfied at level $k$ for $\sigma$. It remains to check whether Equation 2 is satisfied the same way with $U'_S$, $q'_S$ and $r'_S$. $[U_{S'} J + q_{S'}]\,[k] - [U_S I + q_S]\,[k]$ is a non negative vector of size $k$ whose first $k-1$ first components are null. Thus, multiplying by $T_S$, we find that $[U'_{S'} J + q'_{S'}]\,[k] - [U'_S I + q'_S]\,[k]$ and $[U_{S'} J + q_{S'}]\,[k] - [U_S I + q_S]\,[k]$ are equal since $[T_S]\,[k] = [T_{S'}]\,[k]$ and all diagonal components of $T_S$ are 1. Furthermore, by construction, $r_S = r'_S$. This proves that the dependence is also satisfied at level $k$ in $\sigma'$ and has the same nature (loop carried or loop independent). □

Lemma 1 is the key property to simplify schedules used for detecting sequential and parallel loops. This leads to the following result:

**Theorem 1.** *For any shifted linear schedule $\sigma = (M_S, \rho_S)$, there exists a shifted linear schedule $\sigma' = (M'_S, \rho'_S)$, equivalent for parallel and sequential loops, and such that $M'_S = H'_S U'_S$ where $H'_S$ is positive diagonal and $U'_S$ unimodular.*

*Proof.* We write $H_S = K_S D_S$ where $D_S$ is the diagonal matrix such that $D_S$ and $H_S$ have the same diagonal. Then, $K_S^{-1}$ is lower triangular with diagonal equal to 1, and $\left[K_S^{-1}\right]\,[c_{S,S'}] = \left[K_{S'}^{-1}\right]\,[c_{S,S'}]$ since $[K_S]\,[c_{S,S'}] = [K_{S'}]\,[c_{S,S'}]$ and since $K_S$ and $K_{S'}$ are lower triangular. Moreover, $K_S^{-1} M_S = D_S H_S^{-1} H_S U_S = D_S U_S$ which is integral. Finally, by Lemma 1, $\sigma' = (K_S^{-1} M_S, K_S^{-1} H_S q_S + r_S) = (D_S U_S, D_S q_S + r_S)$ is an equivalent schedule. □

# B Permutable loops and shifted linear schedules

We denote by $j_{S,S'}$ the maximal level $\leq c_{S,S'}$ of a block of permutable loops that surround both $S$ and $S'$ and by $i_{S,S'}$ the first level of this innermost block. We first need a lemma.

**Lemma 2.** *Let $\sigma = (M_S, \rho_S)$ be a shifted linear schedule. Suppose given, for each statement $S$, a rational matrix $G_S$, lower triangular, such that $G_S M_S$ is integral, and $[G_S][j_{S,S'}] = [G_{S'}][j_{S,S'}]$ for any statements $S$ and $S'$. If, for any maximal block of permutable loops from level $i$ to level $j$ surrounding $S$, $[G_S][i,j]$ is non negative and all components of the last row of $[G_S][i,j]$ are greater than 1, then, $\sigma' = (G_S M_S, \lfloor G_S \rho_S \rfloor)$ is a shifted linear schedule, and permutable loops for $\sigma$ are permutable loops for $\sigma'$.*

*Proof.* Consider a block of permutable loops from level $i$ to level $j$, surrounding two statements $S$ and $S'$, and a dependence $S(I) \longrightarrow S'(J)$ satisfied for $\sigma$ at level $k$, with $i \leq k \leq j$. We evaluate the vector $D = G_{S'}(M_{S'}J + \rho_{S'}) - G_S(M_S I + \rho_S)$. Since $[G_S][j_{S,S'}] = [G_{S'}][j_{S,S'}]$ and since $[G_S][i,j]$ is non negative and lower triangular, $[D][k-1] = 0$ and $[D][j] \geq 0$. Furthermore, all components of the last row of $[G_S][i,j] = [G_{S'}][i,j]$ are greater than 1, thus $[D]_j \geq 1$. Therefore, the evaluation of $\lfloor D \rfloor$ shows that the dependence is carried at level $k'$ with $k \leq k' \leq j$. Furthermore, the components of the distance vector $D$ are non negative at least until level $j$. This proves that the structure of the blocks of permutable loops are the same for $\sigma$ and for $\sigma'$. □

We now can prove the desired property for schedules used for detecting permutable loops:

**Theorem 2.** *For any shifted linear schedule $\sigma = (M_S, \rho_S)$, there exists a shifted linear schedule $\sigma' = (M'_S, \rho'_S)$, equivalent for permutable loops, and such that $M'_S = H'_S U'_S$ where $U'_S$ is unimodular, and $H'_S$ is positive diagonal, with all entries equal to 1, except possibly for each last level of a block of permutable loops containing $S$.*

*Proof.* The construction of Theorem 1 is not sufficient to guarantee that permutable loops are preserved: the matrix $G_S$ in Lemma 2 needs additional properties. We start from $H_S^{-1}$, and for each block of permutable loops from level $i$ to level $j$ (starting from the innermost block), we use a process similar to the (left) Hermite decomposition of $\left[H_S^{-1}\right][i,j]$, multiplying on the left by a unimodular matrix: we add a multiple of the $(j-1)$-th row of $H_S^{-1}$ to the $j$-th row so that the $(j-1)$-th component of the $j$-th row is non negative and smaller than the diagonal component of the same column. Then, we repeat the operation for the $(j-2)$-th column, manipulating the $j$-th and $(j-1)$-th rows with the $(j-2)$-th row, and so on, until the $i$-th row. Actually, we slightly modify this process for each step, so that all components of the last row of $\left[H_S^{-1}\right][i,j]$ are greater than the diagonal component of the same row (which is not necessarily true if we only simulate the Hermite form).

With this process, we end up with a decomposition $H_S^{-1} = T_S R_S$ where $T_S$ is unimodular and $R_S$ has the properties mentioned above. Moreover, by construction and since $\left[H_S^{-1}\right][j_{S,S'}] = \left[H_{S'}^{-1}\right][j_{S,S'}]$, we have $[T_S][j_{S,S'}] = [T_{S'}][j_{S,S'}]$ and $[R_S][j_{S,S'}] = [R_{S'}][j_{S,S'}]$. Let $H'_S$ be the diagonal matrix whose components are the diagonal components of $H_S$, except those, equal to 1, that do not correspond to the last level of a block of permutable loops containing $S$. Then, $G_S = H'_S R_S$ satisfies the conditions of Lemma 2, and thus $\sigma' = (G_S M_S, \lfloor G_S \rho_S \rfloor)$ is a shifted linear schedule, equivalent to $\sigma$ for permutable loops. Furthermore, $G_S M_S = H'_S T_S^{-1} U_S$: $\sigma'$ has the desired form. □

# C Minimization of synchronizations with loop fusion and loop bumping

We prove that the minimization of synchronizations with loop fusion and loop bumping is NP-complete. As stated in Section 6, we need to prove the following:

**Theorem 3.** *Let $G_a$ be an acyclic directed graph where each edge $e$ has a weight $w(e) \in \mathbb{Z}$. Given an integer $\rho_S$ for each vertex $S$, we define the quantity $w_\rho(e)$ for each edge $e = (S_e, S'_e)$ by $w_\rho(e) = 0$ if $w(e) + \rho_{S'_e} - \rho_{S_e} = 0$, and $w_\rho(e) = 1$ otherwise. The weight of a path is defined as the sum of the weights $w_\rho(e)$ of its edges. Then, finding values for $\rho_S$ which minimize the maximal weight of a path in $G_a$ is NP-complete.*

*Proof.* The decision problem associated to the fusion problem is: given a non negative constant $D$, can we find a value for each $\rho_S$ such that the maximal weight $w_\rho(P)$ of a path $P$ in $G_a$ is less than $D$? This problem is obviously in NP: given some values for the $\rho_S$, the maximal path weight in $G_a$ can be computed in linear time, and compared to $D$.

Now, to prove the NP-completeness, we use a polynomial reduction from the UET-UCT problem[32]. The UET-UCT problem (Unitary Execution Time, Unitary Communication Time) is a scheduling problem with unbounded number of processors, and with execution and communication delays. It is defined by a DAG (Directed Acyclic Graph) $G = (V, E)$ where each vertex $v \in V$ has a weight equal to 1 (the duration of the task $v$), and each edge $e \in E$ has a weight (the communication time) equal to 1 if both extremities of $e$ are not mapped to the same processor, and 0 otherwise. The goal is to define for each task $v$ a processor $\pi(v)$ and an execution date $t_\pi(v)$ such that each processor computes only one task at a time, and such that the total execution time $1 + \max_{v \in V} t_\pi(v)$ is minimized. We point out that the DAG to be scheduled can be assumed with no multi-edge and no transitive edge since such edges correspond to redundant constraints for the scheduling problem. We will make this (classical) assumption in the rest of the proof.

It has been proved in[32] that the UET-UCT problem is NP-complete. Moreover, the study in[23] shows that the optimal solution can be searched among solutions whose mapping $\pi$ is a *linear clustering*, i.e. a mapping such that $\pi(u) = \pi(v)$ may occur only if there is a directed path from $u$ to $v$, or from $v$ to $u$. Restricting to linear clusterings makes the problem simpler to formulate (even if, of course, it remains NP-complete). Indeed, given a linear clustering $\pi$, the problem of determining the execution date $t_\pi$ is a classical DAG scheduling problem. The constraints for the execution times $t_\pi(v)$ are: for each edge $e = (u, v)$,

$$\begin{cases} t_\pi(v) \geq t_\pi(u) + 1 \text{ if } \pi(u) = \pi(v) & \text{(execution of } u\text{)} \\ t_\pi(v) \geq t_\pi(u) + 2 \text{ if } \pi(u) \neq \pi(v) & \text{(execution of } u \text{ plus communication to } v\text{)} \end{cases} \tag{4}$$

In other words, each task $v$ can be scheduled as soon as possible: its execution ends at time $u_\pi(v) = t_\pi(v) + 1$, and $u_\pi(v)$ can be defined as the maximal weight (counting the weights of both edges and vertices) of a path ending at $v$.

We now reduce this UET-UCT linear clustering problem to our fusion problem by the following polynomial construction. We transform each DAG $G_s$ for the scheduling problem (as said above with no multi-edges and no transitive edges) into a DAG $G_a$ (with some

multi-edges) for the fusion problem. For each edge $e$ in $G_s$, we define in $G_a$ two vertices $h_e$ ($h$ for head) and $t_e$ ($t$ for tail) and an edge from $h_e$ to $t_e$ of weight $w = 1$. For each pair of edges $e$ and $f$ ($e \neq f$) in $G_s$ that both leave (or both enter) the same vertex, we define in $G_a$ an edge from $h_e$ to $t_f$, and an edge from $h_f$ to $t_e$ both with weight $w = 0$. This first part of the construction is illustrated in Figure 15. For each pair of edges $e$ and $f$ in $G_s$ such that $e$ enters the vertex that $f$ leaves, we define two edges in $G_a$ from $t_e$ to $h_f$ with respective weights $w = 0$ and $w = 1$. This second part of the construction is illustrated in Figure 16. We extend this construction to a vertex $v$ with no predecessors (resp. no successors) by defining an additional vertex $t_v$ (resp. $h_v$), and two edges from $t_v$ to $h_e$ (resp. from $t_e$ to $h_v$) with respective weights $w = 0$ and $w = 1$, for each edge $e$ that leaves $v$ (resp. enters $v$). For example, the graph $G_s$ of Figure 17 is transformed into the graph $G_a$ of Figure 18 (dotted circles indicate vertices that correspond to a given vertex of the initial graph). Note that, by construction, all undirected elementary cycles of $G_a$ have a non zero weight.

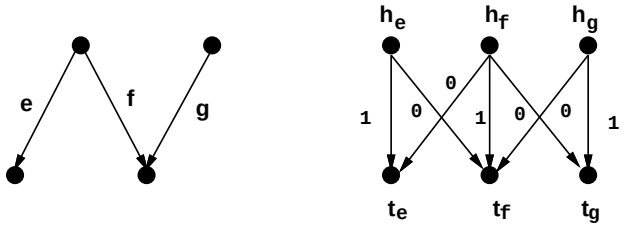



Figure 15: First part of the translation.

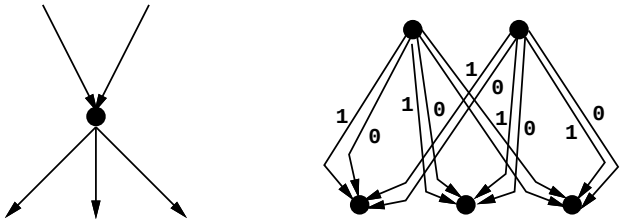



Figure 16: Second part of the translation.

Following the notations we used previously, for a linear clustering $\pi$, we denote by $u_\pi(v)$ the maximal weight of a path (sum of vertex and edge weights) in $G_s$ ending at $v$: $u_\pi(v)$ can be interpreted as the time step at which $v$ completes execution. For a loop bumping $\rho$, we denote by $p_\rho(v)$ the maximal weight of a path (sum of edge weights $w_\rho(e)$) in $G_a$ ending at $v$: $p_\rho(v)$ can be interpreted as the label (counting from 0) of the parallel loop in which $v$ should be placed. We now explain the correspondence between $p_\rho$ and $t_\pi$.

We first point out that, when the DAG $G_s$ has no multi-edge, and no transitive edge (as we assumed), a mapping $\pi$ is a linear clustering if and only if, for each vertex $v$, at most one of the successors (resp. predecessors) of $v$ is mapped to the same processor as $v$. We use this local characterization hereafter.

Each vertex $v$ in $G_s$ corresponds in $G_a$ to a complete bipartite subgraph of $G_a$ with vertices of the form $t_e$ corresponding to edges that enter $v$, and vertices of the form $h_f$

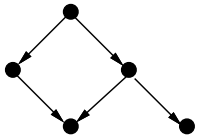


Figure 17: UET-UCT graph.

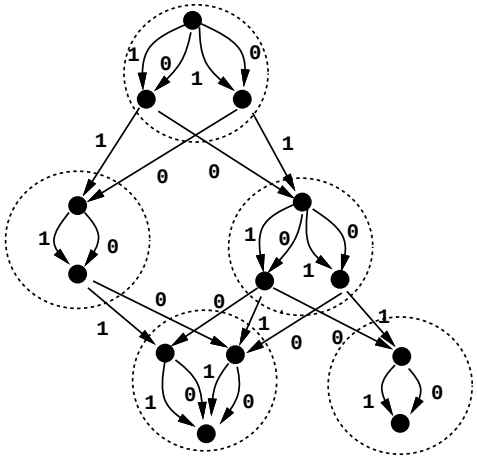


Figure 18: Transformed graph.

corresponding to edges that leave $v$ (see for example the dotted circles in Figure 18). Each $h_f$ is a successor of each $t_e$, and $h_f$ (resp. $t_e$) has no other predecessor (resp. successor). Thus, whatever the loop bumping $\rho$, all $p_\rho(h_f)$, for the edges $f$ that leave $v$, are equal (more precisely equal to 1 plus the maximal value among the $p_\rho(t_e)$ where $e$ enters $v$): we denote this common value by $\tilde{p}_\rho(v)$. We have $\tilde{p}_\rho(v) \geq \tilde{p}_\rho(u)+1$: traversing a "dotted circle" costs 1. Furthermore, because of the undirected cycles of non zero weight such as those in Figure 15, $\tilde{p}_\rho(v) = \tilde{p}_\rho(u) + 1$ is possible for only one predecessor $u$ (resp. successor $v$) of $v$ (resp. $u$). Therefore, we can define a linear clustering $\pi$ such that $\pi(u) = \pi(v)$ if and only if there is an edge from $u$ to $v$ with $\tilde{p}_\rho(v) = \tilde{p}_\rho(u) + 1$. Then, $t_\pi(v) = u_\pi(v) - 1 = \tilde{p}_\rho(v) - 1$ defines a valid schedule and $\max_{v \in G_s} u_\pi(v) = \max_{h \in G_a} p_\rho(h)$.

Conversely, suppose given a linear clustering $\pi$, and the corresponding $u_\pi$. We show that it is possible to define a loop bumping $\rho$ and the corresponding $p_\rho$'s such that $\max_{v \in G_s} u_\pi(v) = \max_{h \in G_a} p_\rho(h)$. We define $\rho$ as follows: for each edge $e = (u, v)$ in $G_s$ such that $\pi(u) = \pi(v)$, we let $\rho_{h_e} = 1$ and for all edges $f = (w, v)$, $f \neq e$, we let $\rho_{t_f} = 1$. For all other vertices in $G_a$, the value of $\rho$ is 0. Since $G_s$ has no multi-edge and no transitive edge, and since $\pi$ is a linear clustering, the value of $\rho$ for all vertices is uniquely defined. Furthermore, with this particular choice for $\rho$, the weights of paths in $G_a$ and $G_s$ are strongly related. If $\pi(u) = \pi(v)$, then all edges from a vertex $h_e$ corresponding to $u$ to a vertex $t_f$ corresponding to $v$ have a null weight, therefore the constraint that links $\tilde{p}_\rho(u)$ and $\tilde{p}_\rho(v)$ is $\tilde{p}_\rho(v) \geq \tilde{p}_\rho(u) + 1$. If $\pi(u) \neq \pi(v)$, then the edge from $h_e$ to $t_e$ remains of weight $w_\rho = 1$ (since its weight $1 + \rho_{t_e} - \rho_{h_e}$ is either 1 or 2 but never 0). Therefore the constraint that links $\tilde{p}_\rho(u)$ and $\tilde{p}_\rho(v)$ is $\tilde{p}_\rho(v) \geq \tilde{p}_\rho(u) + 2$. Therefore, for all vertices $u \in G_s$, $u_\pi(u) = \tilde{p}_\rho(u)$. This proves that the maximal weight of a path is the same in $G_a$ and in $G_s$. The NP-completeness of our loop fusion problem follows. □

# References


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

[2] John R. Allen and Ken Kennedy. Automatic translation of Fortran programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, October 1987.

[3] Corinne Ancourt and François Irigoin. Scanning polyhedra with DO loops. In *Proc. of the 3rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 39–50, April 1991.

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

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

[6] Pierre Boulet and Michèle Dion. Code generation in Bouclettes. In *Proceedings of the Fifth Euromicro Workshop on Parallel and Distributed Processing*, pages 273–280, London, UK, January 1997. IEEE Computer Society Press.

[7] Thomas Brandes. *ADAPTOR Programmer's Guide - Version 4.0.* German National Research Institute for Computer Science, March 1996.

[8] David Callahan. *A Global Approach to Detection of Parallelism.* PhD thesis, Dept. of Computer Science, Rice University, March 1987.

[9] Zbigniew Chamski. *Environnement logiciel de programmation d'un accélérateur de calcul parallèle.* PhD thesis, Université de Rennes, Rennes, France, 1993.

[10] 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.

[11] 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.

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

[13] Alain Darte, Georges-André Silber, and Frédéric Vivien. Combining retiming and scheduling techniques for loop parallelization and loop tiling. *Parallel Processing Letters*, 1997. Special issue, to appear.

[14] 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.

[15] Alain Darte and Frédéric Vivien. A comparison of nested loops parallelization algorithms. *Parallel Processing Letters*, 1997. To appear, a short version is available in the proceedings of ETFA'95.

[16] 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.

[17] Paul Feautrier. Parametric integer programming. *RAIRO Recherche Opèrationnelle*, 22:243–268, September 1988.

[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] High Performance Fortran Forum. High Performance Fortran Language Specification. Technical Report 2.0, Rice University, January 1997.

[22] Marc Le Fur, Jean-Louis Pazat, and Françoise André. Commutative loop nest distribution. In H.J. Sips, editor, *Proc. of the Fourth Int. Workshop on Compilers for Parallel Computers*, pages 345–350, Delft, The Netherlands, December 1993.

[23] Apostolos Gerasoulis and Tao Yang. On the granularity and clustering of directed acyclic task graphs. *IEEE Trans. Parallel and Distributed Systems*, 4(6):686–701, 1993.

[24] The group of Pr. Lengauer. The loopo project. World Wide Web document, URL: `http://brahms.fmi.uni-passau.de/cl/loopo/index.html`.

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

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

[27] Wayne Kelly, William Pugh, and Evan Rosser. Code generation for multiple mappings. In *The 5th Symposium on Frontiers of Massively Parallel Computation*, pages 332–341, McLean, Virginia, February 1995.

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

[29] Wei Li and Keshav Pingali. A singular loop transformation framework based on non-singular matrices. In *5th Workshop on Languages and Compilers for Parallel Computing*, pages 249–260, Yale University, August 1992.

[30] 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.

[31] Kathryn S. McKinley and Ken Kennedy. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *The Sixth Annual Languages and Compiler for Parallelism Workshop*, number 768 in Lecture Notes in Computer Science, pages 301–320. Springer-Verlag, 1993.

[32] C. Picouleau. Two new NP-complete scheduling problems with communication delays and unlimited number of processors. Technical Report 91-24, IBP, Université Pierre et Marie Curie, France, April 1991.

[33] William Pugh. A practical algorithm for exact array dependence analysis. *Communications of the ACM*, 8:27–47, August 1992.

[34] William Pugh and the Omega Team. World Wide Web document, URL: `http://www.cs.umd.edu/projects/omega/`.

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

[36] 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.

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

[38] Michael Wolfe. *High Performance Compilers For Parallel Computing*. Addison-Wesley Publishing Company, 1996.

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

[40] Yi-Qing Yang, Corinne Ancourt, and François Irigoin. Minimal data dependence abstractions for loop transformations. *International Journal of Parallel Programming*, 23(4):359–388, August 1995.