ELSEVIER

Parallel Computing 24 (1998) 421–444

PARALLEL
COMPUTING

Loop parallelization algorithms: From parallelism extraction to code generation

Pierre Boulet *, Alain Darte [1], Georges-André Silber [2], Frédéric Vivien [3]

Laboratoire LIP, URA CNRS 1398, École Normale Supérieure de Lyon, F-69364 Lyon Cedex 07, France

Received 10 April 1997; revised 2 September 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 loop 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 multidimensional schedules called shifted linear schedules. We demonstrate that algorithms based on such schedules lead to simple codes. © 1998 Elsevier Science B.V. All rights reserved.

Keywords: Automatic parallelization; Nested loops; Parallelization algorithms; Loop fusion; 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 et al. [1] or Wolfe's book [2]), and their particular use. Two additional surveys have presented the link between loop parallelization algorithms and dependence analysis: in

* Corresponding author. E-mail: pierre.boulet@lip.ens-lyon.fr
[1] E-mail: alain.darte@lip.ens-lyon.fr
[2] E-mail: georges-andre.silber@lip.ens-lyon.fr
[3] E-mail: frederic.vivien@lip.ens-lyon.fr

0167-8191/98/$19.00 © 1998 Elsevier Science B.V. All rights reserved.
PII S0167-8191(98)00020-9

Ref. [3], Yang et al. characterize for each loop transformation used to reveal parallelism, the minimal dependence abstraction needed to check its validity; in Ref. [4], 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 we must keep in mind that generating parallel loops is not sufficient for generating efficient parallel executable programs. When designing parallel loop detection algorithms, we must consider many other optimizations related to the granularity of the parallel program, to the data distribution, to the communications, etc. For that, the algorithm must be *flexible enough* to incorporate criteria that are more accurate than the simple detection of parallel loops. Furthermore, the parallelizing algorithm is only a step in the whole compilation scheme: it should thus generate an intermediate abstract parallel code that is *simple enough* so that further optimizations can still be performed. For example, an algorithm for parallel loop detection can be integrated in a parallelization platform (as schematized in Fig. 1) that transforms (automatically or semi-automatically) sequential Fortran to parallel Fortran via a language such as High Performance Fortran (HPF) [5].

The aim of this paper is to survey loop parallelization algorithms with such a compilation scheme in mind. Sections 2 and 3 are devoted to a brief state of the art: in Section 2, we survey the different algorithms proposed in the literature, recalling the dependence representations they use, the loop transformations they generate, and their ability to incorporate optimizing criteria such as maximal parallelism detection, permutable loops detection, minimization of synchronizations, etc. In Section 3, we present the code generation techniques involved by these loop transformations. Sections 4 and 5 present in more details some new contributions dealing with two particular optimization problems: how to generate codes that are as simple as possible (Section 4) and how to handle loop fusion (for example to minimize synchronizations) in loop parallelization algorithms (Section 5). Finally, we give some conclusions in Section 6.

Before that, we take the example of the compilation of High Performance Fortran to introduce some concepts and vocabulary, and to illustrate the kind of optimizations that remain to be done once parallel loops have been detected.

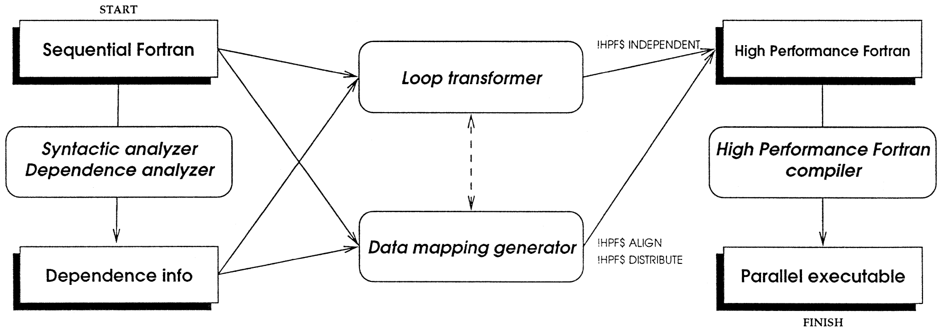

Fig. 1. Parallelizing process.

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

(a)

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

(b)

Fig. 2. Two abstract parallel codes.

Consider the codes of Fig. 2. Both have a single do loop, marked by the INDEPENDENT directive of HPF that asserts to the compiler that the iterations may be executed concurrently without changing the semantics of the program. There are no *dependences* between the different iterations of the loop or, in other words, there are no dependences *carried* by the loop. However, in the code (b) there is a *loop-independent* dependence, from S_a to S_b for a given iteration of the loop.

Starting from the code in Fig. 2a, a HPF compiler may produce two types of code, following the *owner-computes rule* (the processor that *owns* the left-hand side of the computation *computes* it). 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 Fig. 3a. To make this code more efficient, communications can be moved outside of the loop if array accesses are known at compile-time, or if the data have been already sent for a previous set of loops. Nevertheless, it remains inefficient since each processor spans the entire iteration space. If the code to compile is simple enough (as here) 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 Fig. 3b. 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 code of Fig. 2b. A code such as the code in Fig. 3a could also be generated. However, the communication of array *A* needed to compute array *C* cannot just be 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 choice is to select a good mapping

```
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)

Fig. 3. Two types of parallel executable codes for the code of Fig. 2a.

```
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)

Fig. 4. Two types of parallel executable codes for the code of Fig. 2b.

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 Fig. 4b.

If A and C are not mapped to the same processor, these is a last possibility. Indeed, some compilers offer the ON HOME directive [4] modifying the owner-computes rule. In the 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. 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. However, loop fusion (opposed to loop distribution) offers some other advantages (see Section 2) and it may be desirable to produce codes with large loop bodies.

This brief discussion shows that even with an INDEPENDENT directive, the actual generation of the parallel code has a variable degree of difficulty. The way parallel loops are exposed (fused or not fused for example) interferes with the different optimizations that are done (or that are not done!) before or after the generation of parallel loops (choice of the data mapping, generation of communications, of synchronizations, etc.). We will not describe these optimizations here. We refer to the survey in the same issue for a description of data mapping techniques.

2. 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.

[4] This directive is an approved extension of HPF 2.0 and some compilers have already implemented it, like ADAPTOR [6], an HPF compiler by Thomas Brandes.

Table 1
A comparison of various loop parallelizing algorithms

Algorithms	Dependence abstraction	Loop transformation	Maximal degree of parallelization	Synchronization through fusion	Code generation	Tiling
Allen and Kennedy [7]	Dependence level; multiple statements; nonperfect	Distribution	Optimal	Yes	Very easy	No
Wolf and Lam [8]	Direction vectors; one statement; perfect	Unimodular	Optimal	No	Easy	Yes
Darte and Vivien [9]	Polyhedra; multiple statements; nonperfect	Affine	Optimal	No	Complicated	No
Feautrier [10]	Affine (exact); multiple statements; nonperfect	Affine	Suboptimal	No	Complicated	No
Lim and Lam [11]	Affine (exact); multiple statements; nonperfect	Affine	Suboptimal	?	?	Yes

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 et al. [1]. 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 2.2). Before, in Section 2.1, we recall the dependence abstractions used to check the validity of the transformations. Finally, in Section 2.3, 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 code: nested loops, possibly nonperfectly 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 statically restricts 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 $\boldsymbol{I} \in D_S$ corresponds a particular execution of S denoted by $S(\boldsymbol{I})$. In the sequential order, all computations $S(\boldsymbol{I})$ are executed following the lexicographical order defined on the iteration vectors. If $\boldsymbol{I}$ and $\boldsymbol{J}$ are two vectors, we write $\boldsymbol{I} \prec_{\text{lex}} \boldsymbol{J}$ if $\boldsymbol{I}$ is lexicographically strictly smaller than $\boldsymbol{J}$, and $\boldsymbol{I} \preceq_{\text{lex}} \boldsymbol{J}$ if $\boldsymbol{I} \prec_{\text{lex}} \boldsymbol{J}$ or $\boldsymbol{I} = \boldsymbol{J}$.

2.1. Dependence abstractions

Data dependence relations between operations are defined by Bernstein's conditions [12]. Two operations are dependent if both operations access the same memory location and if at least one access is a write. The dependence is directed according to the sequential order, from the first executed operation to the last one. We write $S(\boldsymbol{I}) \rightarrow S'(\boldsymbol{J})$ if the statement S' at iteration $\boldsymbol{J}$ depends on the statement S at iteration $\boldsymbol{I}$. 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 $(\boldsymbol{I}, \boldsymbol{J}) \in D_S \times D_{S'}$ such that $S(\boldsymbol{I}) \rightarrow S'(\boldsymbol{J})$. 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 $S(\boldsymbol{I}) \rightarrow S'(\boldsymbol{J})$ is represented by an approximation of the *distance vector* $\boldsymbol{J} - \boldsymbol{I}$. If S and S' do not have the same domain, only the components of the vector $\boldsymbol{J} - \boldsymbol{I}$, that correspond to the $n_{S,S'}$ loops surrounding both statements, are defined. The four classical representations of distance vectors (by increasing precision) are enumerated below.

2.1.1. Dependence level

The dependence level was introduced by Allen and Kennedy in Refs. [13,7]. A distance vector $\boldsymbol{J}-\boldsymbol{I}$ is approximated by an element l (the level) in $[1,n_{S,S'}]\cup\{\infty\}$, defined as ∞ if $\boldsymbol{J}-\boldsymbol{I}=0$, 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 to be *loop-independent* and *loop-carried*, otherwise.

2.1.2. Direction vector

The direction vector was first described by Lamport in Ref. [14], then by Wolfe in Ref. [15]. 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.

2.1.3. Dependence polyhedron

The dependence polyhedron was introduced by Irigoin and Triolet [16]. 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.

2.1.4. Affine dependences

The affine dependences were used by Feautrier [17] to express dependence relations when exact dependence analysis is feasible. A set of dependences $S(\boldsymbol{I})\to S'(\boldsymbol{J})$ can be represented by an affine function f that expresses $\boldsymbol{I}$ in terms of $\boldsymbol{J}$ ($\boldsymbol{I}=f(\boldsymbol{J})$) or the converse, subject to affine inequalities that restrict the range of validity of the dependence.

2.2. Loop transformations

We only focus here on the transformations that are captured by the loop parallelization algorithms presented in Section 2.3.

2.2.1. 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.

2.2.2. 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).

2.2.3. 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

$\boldsymbol{I}' = \mathbf{U}\boldsymbol{I}$ where $\mathbf{U}$ 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 $\mathbf{U}$ is valid if and only if $\mathbf{U}\boldsymbol{d} \succ_{\text{lex}} 0$ for each nonzero distance vector $\boldsymbol{d}$.

2.2.4. Affine transformations

A general affine transformation defines a new iteration vector $\boldsymbol{I}'$ for each statement S by an affine function $\boldsymbol{I}' = \mathbf{M}_S\boldsymbol{I} + \boldsymbol{\rho}_S$. $\mathbf{M}_S$ is a nonparameterized nonsingular square integral matrix of size n_S, and $\boldsymbol{\rho}_S$ is a possibly parameterized vector. The linear part may be unimodular or not. Such a transformation is valid if and only if $S(\boldsymbol{I}) \to S'(\boldsymbol{J}) \Rightarrow \mathbf{M}_S\boldsymbol{I} + \boldsymbol{\rho}_S \prec_{\text{lex}} \mathbf{M}_{S'}\boldsymbol{J} + \boldsymbol{\rho}_{S'}$.

2.2.5. Tiling

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

2.3. 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. However, it does not mean that more parallelism cannot be detected if more information is exploited.

Lamport's algorithm [14] 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 Ref. [18]) and to multidimensional schedules.

Allen and Kennedy's algorithm [7] is based on the decomposition of the reduced dependence graph into strongly connected components. It uses dependences represented by levels, and transforms codes by loop distribution and statement reordering. It is optimal for maximal parallelism detection (see Ref. [19]). The minimization of synchronizations is considered through loop fusion (see Section 5.1). However, it is not really adapted (because of the inaccuracy of dependence levels) to the detection of outer parallelism and permutable loops.

Wolf and Lam's algorithm [8] 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 dependences are captured by direction vectors (and no dependence graph structure).

Feautrier's algorithm [20,10] produces a general affine transformation. It can handle perfectly nested loops as well as nonperfectly nested loops as long as exact dependence analysis is feasible. It relies on affine dependences. The affine transformation is built as a solution of linear programs obtained by the affine form of Farkas' lemma [21] 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, currently, to the detection of outer parallelism and permutable loops.

Darte and Vivien's algorithm [9] 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 4.1). 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 Ref. [22]), and the minimization of synchronizations through loop fusion (see Section 5.2). Furthermore, the code generation is simpler (see Section 4). 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's algorithm [11] 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.

3. 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 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. The problem can be viewed as generating some iteration code for polyhedra, affine images of polyhedra and unions of affine images of polyhedra. We follow this hierarchy in our discussion.

3.1. 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 second 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), this is easy.

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

3.1.1. Fourier–Motzkin elimination

Ancourt and Irigoin first proposed this technique in Ref. [23] and it has then been used in several prototype compilers [24–26]. 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 elimination, some inequalities are added to the system to build a 'triangular' system where 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 a combination of loop skewing and loop interchange $\begin{pmatrix} i'_1 \\ i'_2 \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} i_1 \\ i_2 \end{pmatrix}$ (see Fig. 5). The system describing the transformed polyhedron and its transformation by the Fourier–Motzkin elimination of i'_2 are:

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

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 elimina-

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

```
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
```

$$\xrightarrow{\begin{pmatrix} i'_1 \\ i'_2 \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} i_1 \\ i_2 \end{pmatrix}}$$

Fig. 5. Unimodular example.

tion or on the simplex algorithm [26]. If some redundant inequalities are not removed, empty iterations may appear in the resulting loop nest, causing overhead. Although the Fourier–Motzkin elimination has super-exponential complexity for big problems, it remains fast for small problems, and works well in practice.

3.1.2. 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 et al. in Ref. [27] and has been used in at least three experimental parallelizers [28,27,29].

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 Ref. [27] that this resolution can be done using PIP [30], 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 Fig. 5, 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 [31] addresses the problem of control overhead by replacing extreme operations by conditionals at the expense of code duplication.

3.2. Non-unimodular linear transformations

When dealing with non-unimodular transformations, the classical approach [32] is to decompose the transformation matrix into its Hermite normal form [21] 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** $(=\mathbf{T})$ 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 nonzero entries are shifts. Fig. 6 shows the example of the transformation of a 2-D loop nest by $\begin{pmatrix} 2 & 0 \\ 1 & 3 \end{pmatrix}$.

```
do $i_1 = lb_{i_1}, ub_{i_1}$
    do $i_2 = lb_{i_2}, ub_{i_2}$
        ...
    enddo
enddo
```

```
do $i'_1 = 2lb_{i_1}, 2ub_{i_1}, 2$
    do $i'_2 = \frac{i'_1}{2} + 3lb_{i_2}, \frac{i'_1}{2} + 3ub_{i_2}, 3$
        ...
    enddo
enddo
```

Fig. 6. Non-unimodular example.

Xue presented in Ref. [33] another method to deal with non-unimodular transformations. It is based on the Fourier–Motzkin elimination to compute the loop bounds and on Hermite decomposition to compute the steps and shifting constants.

3.3. Extensions

3.3.1. Perfect loop nests with one-dimensional shifted linear schedules

In Ref. [28], 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.

3.3.2. Nonperfect loop nests with one-dimensional schedules

Collard presents in Ref. [34] a method to produce code when each statement of a nonperfect loop nest has been assigned an affine one-dimensional schedule.

3.3.3. General affine case

Kelly, Pugh and Rosser present in Ref. [35] a method to generate code for the general affine case: a nonperfect loop nest transformed with a possibly different affine transformation for each statement. Their transformation is based on the Presburger arithmetic implemented in the Omega library [36]. The problem is to be able to scan an arbitrary union of polyhedra. In Section 4, we present a simpler code generation scheme for the case of shifted linear schedules.

4. Code generation for shifted linear schedules

In most real codes, the loop transformations needed to exploit parallelism are quite simple (combinations of distribution and interchange for example). When they are derived and applied with the help of the human (i.e., by hand or semi-automatically), particular cases can be identified and complicated cases can be avoided when possible. However, when the loop transformations are derived and applied by the parallelizing compiler itself (i.e., automatically), all cases of transformations have to be considered in the code generation step, even if the complicated cases will never occur: the consequence is that the resulting code may be unreadable (which is a drawback if the user wants to check what the compiler is doing) or just too complicated for further optimizations. Therefore, we believe that it is important in a parallelizer:

- to generate simple transformations when possible (optimization in the scheduling step).
- to develop simpler code generation schemes for particular types of transformations (optimization in the code generation step).

In this section, we address these two points (the second point first) by studying in more details a particular class of loop transformations that we call *multidimensional shifted linear schedules*. We will illustrate our results with the example below.

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 multidimensional 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 Fig. 7. The code is not really simple, involving complex guards (with modulo operations), non-unit loop steps, complicated loop bounds and array access functions. Nevertheless, it is already a bit 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 but redundant conditionals such as $t_1 \leq n-1$ for statement S' have been removed.

Such complicated codes may occur when trying to scan (i.e., describe by loops) an arbitrary union of polyhedra, either when scheduling simultaneously several statements whose iteration domains are different (nonperfectly nested loops) or when scheduling several statements with different multidimensional schedules. However, when restricting to multidimensional shifted linear schedules, we can avoid this general scanning problem. We will show that, if all iteration domains are the same 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 and more readable codes. Indeed, the code generation can be seen as a hierarchical combination of loop distributions, loop bumpings (i.e., adding a constant to a loop counter), and matrix transformations, where each statement can be considered independently (thus avoiding the complicated problem of overlapping different polyhedra). Furthermore, we will show that given

```
do $t_1 = -n+1, n$
  do $t_2 = \max(t_1+2, -t_1+2), \min(t_1+2n+1, -t_1+2n+3)$
    if $t_1+t_2 = 0 \bmod 2$ and $t_1+t_2 \leq 2n$
      do $t_3 = t_2+2, t_2+2n, 2$
        $S(\frac{t_1+t_2}{2}, \frac{t_2-t_1}{2}, \frac{t_3-t_2}{2})$
      enddo
    endif
    if $t_1+t_2 = 1 \bmod 2$ and $t_2+t_1 \geq 5$
      do $t_3 = 1, n$
        $S'(\frac{t_1+t_2-3}{2}, \frac{t_2-t_1-1}{2}, t_3)$
      enddo
    endif
  enddo
enddo
```

Fig. 7. A code with modulo operations.

a shifted linear schedule σ, it is always possible to build an equivalent shifted linear schedule σ', equivalent in the sense that the nature of the loops (sequential, parallel or permutable) in the transformed code is preserved, and such that the matrix transformations involved for σ' are only unimodular transformations.

4.1. Shifted linear schedules

To make the discussion simpler, we consider that all iteration domains have the same dimension n (i.e., with the notations of Section 2, $n_S = n$ for all statements S) so that all iteration vectors and all matrices have the same size. As recalled in Section 2.2, a multidimensional affine schedule σ is defined for each statement S by an integral square nonsingular matrix $\mathbf{M}_S$ of size n and an integral vector $\boldsymbol{\rho}_S$ of size n. We write $\sigma = (\mathbf{M}_S, \boldsymbol{\rho}_S)$. The computation $S(\boldsymbol{I})$ associated to the iteration vector $\boldsymbol{I}$ before transformation is associated to the iteration vector $\mathbf{M}_S \boldsymbol{I} + \boldsymbol{\rho}_S$ after transformation. A multidimensional function $\sigma = (\mathbf{M}_S, \boldsymbol{\rho}_S)$ is a valid schedule if and only if:

$$S(\boldsymbol{I}) \to S'(\boldsymbol{J}) \Rightarrow \mathbf{M}_S \boldsymbol{I} + \boldsymbol{\rho}_S \prec_{\text{lex}} \mathbf{M}_{S'} \boldsymbol{J} + \boldsymbol{\rho}_{S'} \tag{1}$$

We say that a dependence $S(\boldsymbol{I}) \to S'(\boldsymbol{J})$ is satisfied by σ at level k if:

$$[\mathbf{M}_S \boldsymbol{I} + \boldsymbol{\rho}_S][k-1] = [\mathbf{M}_{S'} \boldsymbol{J} + \boldsymbol{\rho}_{S'}][k-1] \text{ and } [\mathbf{M}_S \boldsymbol{I} + \boldsymbol{\rho}_S]_k < [\mathbf{M}_{S'} \boldsymbol{J} + \boldsymbol{\rho}_{S'}]_k$$

where the notation $[\mathbf{M}]_k$ denotes the k-th row of a matrix $\mathbf{M}$, and the notation $[\mathbf{M}][k]$ denotes the matrix whose rows are the first k rows of $\mathbf{M}$.

Eq. (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 $[\mathbf{M}_S][c] = [\mathbf{M}_{S'}][c]$.

Definition 1. A multidimensional affine schedule $\sigma = (\mathbf{M}_S, \boldsymbol{\rho}_S)$ is *shifted linear* if, for any statements S and S', $[\mathbf{M}_S][k_{S,S'}] = [\mathbf{M}_{S'}][k_{S,S'}]$, i.e., if $k_{S,S'} \le c_{S,S'}$. In other words, a multidimensional 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 2.3, looking for such schedules is not penalizing if dependences are captured 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.

4.2. Code generation scheme

Remember the code generation scheme for a single statement S. If the matrix $\mathbf{M}_S$ is not unimodular, we use the Hermite form $\mathbf{M}_S = \mathbf{H}_S \mathbf{U}_S$ where $\mathbf{U}_S$ is unimodular, $\mathbf{H}_S$

nonnegative, lower triangular, and each nondiagonal component of $\mathbf{H}_S$ is strictly smaller than the diagonal component of same row. Then, we transform the code first using $\mathbf{U}_S$, then using the loop skewing $\mathbf{H}_S$. Here, we have multiple statements, different matrices $\mathbf{M}_S$, and different constants $\boldsymbol{\rho}_S$, 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 [21], it can be shown that:

$$[\mathbf{M}_S][k] = [\mathbf{M}_{S'}][k] \Rightarrow [\mathbf{H}_S][k] = [\mathbf{H}_{S'}][k] \text{ and } [\mathbf{U}_S][k] = [\mathbf{U}_{S'}][k]$$

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 an arbitrary union of polyhedra.

The code generation process is the following. We write $\mathbf{M}_S = \mathbf{H}_S\mathbf{U}_S$ and $\boldsymbol{\rho}_S = \mathbf{H}_S\boldsymbol{q}_S + \boldsymbol{r}_S$ where $\boldsymbol{q}_S$ and $\boldsymbol{r}_S$ are integral vectors where each component of $\boldsymbol{r}_S$ is nonnegative and strictly smaller than the corresponding diagonal element of $\mathbf{H}_S$ (this decomposition is unique). Then, we decompose the transformation $\sigma_S{:}\boldsymbol{I} \to \mathbf{M}_S\boldsymbol{I} + \boldsymbol{\rho}_S$ in four steps, $\sigma_S^1{:}\boldsymbol{I} \to \mathbf{U}_S\boldsymbol{I}$, $\sigma_S^2{:}\boldsymbol{I} \to \boldsymbol{I} + \boldsymbol{q}_S$, $\sigma_S^3{:}\boldsymbol{I} \to \mathbf{H}_S\boldsymbol{I}$, $\sigma_S^4{:}\boldsymbol{I} \to \boldsymbol{I} + \boldsymbol{r}_S{:}\sigma_S^1$ is a unimodular transformation, σ_S^2 is a loop bumping, σ_S^3 is a loop skewing (and each loop step is equal to the corresponding diagonal component of $\mathbf{H}_S$), and σ_S^4 consists simply in writing the code in the $\boldsymbol{r}_S$-th position in the loop body. We point out that when $\mathbf{H}_S$ is diagonal, there is no need to really multiply the counter by the diagonal component: we can keep the original counter and avoid loop steps and floor functions. We will use this remark in Section 4.3.

Back to Example 1. We find the two unimodular transformations $\mathbf{U}_S{:}(i,j,k) \to (i-j,j,j+k)$ and $\mathbf{U}_{S'}{:}(i,j,k) \to (i-j,j,k)$, and the two skewing transformations $\mathbf{H}_S{:}(i,j,k) \to (i,i+2j,i+2k)$ and $\mathbf{H}_{S'}{:}(i,j,k) \to (i,i+2j,k)$. Furthermore $\boldsymbol{q}_S = \boldsymbol{r}_S = (0,0,0)$, and $\boldsymbol{q}_{S'} = (1,0,0)$, $\boldsymbol{r}_{S'} = (0,1,0)$. We first find the two intermediate codes of Fig. 8 by applying the transformations $\mathbf{U}_S$ and $\mathbf{U}_{S'}$, and the loop bumping by $\boldsymbol{q}_S$ and $\boldsymbol{q}_{S'}$.

```
do $t_1 = -n+1, n-1$
   do $t_2 = \max(1, -t_1+1)$,
             $\min(n, -t_1+n)$
      do $t_3 = t_2+1, t_2+n$
         $S(t_1+t_2, t_2, t_3-t_2)$
      enddo
   enddo
enddo
```

```
do $t_1 = -n+2, n$
   do $t_2 = \max(1, -t_1+2)$,
             $\min(n, -t_1+n+1)$
      do $t_3 = 1, n$
         $S'(t_1+t_2-1, t_2, t_3)$
      enddo
   enddo
enddo
```

Fig. 8. Separate codes after unimodular transformation.

```
do $t_1 = -n+1, n-1$
  do $t_2 = \max(t_1+2, -t_1+2),$
            $\min(t_1+2n, -t_1+2n), 2$
    do $t_3 = t_2+2, t_2+2n, 2$
      $S(\frac{t_1+t_2}{2}, \frac{t_2-t_1}{2}, \frac{t_3-t_2}{2})$
    enddo
    nop
  enddo
enddo
```

```
do $t_1 = -n+2, n$
  do $t_2 = \max(t_1+2, -t_1+4),$
            $\min(t_1+2n, -t_1+2n+2), 2$
    nop
    do $t_3 = 1, n$
      $S'(\frac{t_1+t_2}{2}-1, \frac{t_2-t_1}{2}, t_3)$
    enddo
  enddo
enddo
```

Fig. 9. Separate codes after loop skewing.

Then, we apply the loop skewings $\mathbf{H}_S$ and $\mathbf{H}_{S'}$, and the displacements by $\boldsymbol{r}_S$ and $\boldsymbol{r}_{S'}$. We get the two codes of Fig. 9. The displacements are visualized by non-operations.

Finally, merging the two codes, we get the code of Fig. 10. The 'apparent holes' due to the non-unimodularity (and that appear as modulo operations in the code of Fig. 7) have been clearly identified, thanks to the displacements $\boldsymbol{r}_S$ and $\boldsymbol{r}_{S'}$: the code of Fig. 10 is now more efficient. Notice also that the two conditionals $t_1 \le n-1$ and $t_1 \ge -n+2$ could be removed since they are redundant with the other constraints. We just kept them to make the technique more understandable.

4.3. Equivalent schedules and unimodularity

In terms of parallelism extraction, multidimensional 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 σ and σ' 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.

```
do $t_1 = -n+1, n$
  do $t_2 = \max(t_1+2, -t_1+2), \min(t_1+2n, -t_1+2n+2), 2$
    if $t_1 \le n-1$ and $t_2 \le -t_1+2n$
      do $t_3 = t_2+2, t_2+2n, 2$
        $S(\frac{t_1+t_2}{2}, \frac{t_2-t_1}{2}, \frac{t_3-t_2}{2})$
      enddo
    endif
    if $t_1 \ge -n+2$ and $t_2 \ge -t_1+4$
      do $t_3 = 1, n$
        $S'(\frac{t_1+t_2}{2}-1, \frac{t_2-t_1}{2}, t_3)$
      enddo
    endif
  enddo
enddo
```

Fig. 10. Combination of the two codes.

Let us analyze how dependences in the original code are transformed if we use the code generation process described in Section 4.2. If $S(\boldsymbol{I}) \to S'(\boldsymbol{J})$ is satisfied at level k:

$$\begin{aligned}
&[\mathbf{U}_S\boldsymbol{I}+\boldsymbol{q}_S][k-1]=[\mathbf{U}_{S'}\boldsymbol{J}+\boldsymbol{q}_{S'}][k-1] \text{ and } [\boldsymbol{r}_S][k-1]=[\boldsymbol{r}_{S'}][k-1]\\
&[\mathbf{U}_S\boldsymbol{I}+\boldsymbol{q}_S]_k \le [\mathbf{U}_{S'}\boldsymbol{J}+\boldsymbol{q}_{S'}]_k\\
&[\mathbf{U}_S\boldsymbol{I}+\boldsymbol{q}_S]_k = [\mathbf{U}_{S'}\boldsymbol{J}+\boldsymbol{q}_{S'}]_k \Rightarrow [\boldsymbol{r}_S]_k < [\boldsymbol{r}_{S'}]_k
\end{aligned} \qquad (2)$$

In other words, a dependence satisfied at level k is either *loop-carried* at level k (when $[\mathbf{U}_S\boldsymbol{I}+\boldsymbol{q}_S]_k < [\mathbf{U}_{S'}\boldsymbol{J}+\boldsymbol{q}_{S'}]_k$, or *loop-independent* at level k, and in this latter case $\boldsymbol{r}_S < \boldsymbol{r}_{S'}$. 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 of a dependence (loop-carried or loop-independent) is not fully specified by the schedule $\sigma=(\mathbf{M}_S,\boldsymbol{\rho}_S)$ itself, but depends on the way we write the code. For example, handling the constants $\boldsymbol{\rho}_S$ differently may change the nature of a dependence (but not the level at which it is *satisfied* which is fully specified by σ). 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 $\sigma=(\boldsymbol{M}_S,\boldsymbol{\rho}_S)$, there exists a shifted linear schedule $\sigma'=(\boldsymbol{M}'_S,\boldsymbol{\rho}'_S)$, equivalent for parallel and sequential loops, and such that $\boldsymbol{M}'_S=\boldsymbol{H}'_S\boldsymbol{U}'_S$ where $\boldsymbol{H}'_S$ is nonnegative diagonal and $\boldsymbol{U}'_S$ unimodular.*

Proof. See the extended proof in Ref. [37]. We build σ' as follows. We write $\mathbf{H}_S=\mathbf{K}_S\mathbf{H}'_S$ where $\mathbf{H}'_S$ is the diagonal matrix such that $\mathbf{H}'_S$ and $\mathbf{H}_S$ have the same diagonal. Then $\sigma'=(\mathbf{K}_S^{-1}\mathbf{M}_S,\mathbf{K}_S^{-1}\mathbf{H}_S\boldsymbol{q}_S+\boldsymbol{r}_S)=(\mathbf{H}'_S\mathbf{U}_S,\mathbf{H}'_S\boldsymbol{q}_S+\boldsymbol{r}_S)$ 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 $S(\boldsymbol{I}) \to S'(\boldsymbol{J})$ satisfied at a level between i and j, the dependence distance is nonnegative, i.e.,

$$\begin{aligned}
&[\mathbf{M}_S\boldsymbol{I}+\boldsymbol{\rho}_S][i-1]=[\mathbf{M}_{S'}\boldsymbol{J}+\boldsymbol{\rho}_{S'}][i-1]\\
&\quad\Rightarrow [\mathbf{M}_S\boldsymbol{I}+\boldsymbol{\rho}_S][j] \le [\mathbf{M}_{S'}\boldsymbol{J}+\boldsymbol{\rho}_{S'}][j]
\end{aligned} \qquad (3)$$

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 $\sigma=(\boldsymbol{M}_S,\boldsymbol{\rho}_S)$, there exists a shifted linear schedule $\sigma'=(\boldsymbol{M}'_S,\boldsymbol{\rho}'_S)$, equivalent for permutable loops, and such that $\boldsymbol{M}'_S=\boldsymbol{H}'_S\boldsymbol{U}'_S$ where $\boldsymbol{U}'_S$ is unimodular and $\boldsymbol{H}'_S$ 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 $\mathbf{G}_S$ (see the construction in Ref. [37]) such that $\mathbf{G}_S\mathbf{M}_S=\mathbf{H}'_S\mathbf{U}'_S$. Then $\sigma'=(\mathbf{G}_S\mathbf{M}_S,\lfloor\mathbf{G}_S\,\boldsymbol{\rho}_S\rfloor)$ 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 4.2 because of the constants $\boldsymbol{r}_S$. Each time two statements have different values of $\boldsymbol{r}_S$ for the same loop, the resulting code is nonperfectly 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 $\boldsymbol{\rho}_S$ into $\boldsymbol{q}_S$ and $\boldsymbol{r}_S$. However, the resulting code would be much more complicated. This is the reason why we impose in Theorem 2 that the components of $\mathbf{H}'_S$ are equal to 1, except for the last level of a block of permutable loops. Then, the code is easy to generate.

Back to Example 1. We assume that the first two dimensions correspond to a block of permutable loops. Applying our ‘unimodularization’ technique to the initial schedule—$(i-j, i+j, i+j+2k)$ for S and $(i-j+1, i+j+2, k)$ for S'—we find the two loop skewing transformations $\mathbf{G}_S:(i,j,k) \to (i, i+j, -i+k)$ and $\mathbf{G}_{S'}:(i,j,k) \to (i, i+j, k)$ which lead to the schedule $(i-j, 2i, 2j+2k)$ for S and $(i-j+1, 2i+3, k)$ for S'. By construction, this schedule is equivalent for permutable loops. Actually, it is also equivalent for parallel and sequential loops. We get the final equivalent code of Fig. 11: it is simpler and moreover, it reveals exactly the same amount of parallelism (in terms of loops).

As noticed in Section 4.2, 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: there is no need to really multiply the loop counter by the diagonal component.

This study demonstrates that shifted linear schedules have nice properties: we can derive schedules that are guaranteed to be simple (thanks to the ‘unimodularization’ process described above) and we can generate simple codes for them. We point out that this ‘unimodularization’ property is not true for arbitrary multidimensional schedules.

```
do $t_1 = -n+1, n$
  do $t_2 = \max(t_1+1, 1), \min(t_1+n, n+1)$
    if $t_1 \leq n-1$ and $t_2 \leq n$ /* note: first constraint could be removed */
      do $t_3 = t_2 - t_1 + 1, t_2 - t_1 + n$
        $S(t_2, t_2 - t_1, t_1 - t_2 + t_3)$
      enddo
    endif
    if $t_1 \geq -n+2$ and $t_2 \geq 2$ /* note: first constraint could be removed */
      do $t_3 = 1, n$
        $S'(t_2 - 1, t_2 - t_1, t_3)$
      enddo
    endif
  enddo
enddo
```

Fig. 11. Equivalent code for Example 1.

5. Fusion of parallel loops

The advantages of loop fusion are well-known. First, synchronization is a costly operation. Therefore, minimizing the number of synchronizations is important. Fusing parallel loops is part of the answer as one synchronization is required after each parallel loop. Second, even if loop fusion increases the size of the loop, which can have a negative impact on cache and register performances, it can improve data reuse by moving references closer in time, making them more likely to still reside in cache or registers [2,38]. 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.

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 Ref. [39]). 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.

5.1. Fusion of parallel loops in Allen and Kennedy's algorithm

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 (interdependences). The fusion of two parallel loops is valid and gives one parallel loop if there is no interdependence between these two loops, or if all interdependences become loop-independent after fusion. Otherwise, the semantics of the code is not preserved or the loop produced is sequential. An interdependence 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 $\pi(S)$ 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': $\pi(S) \leq \pi(S')$. Furthermore, if this dependence is fusion preventing, S and S' cannot be in the same loop after fusion: $\pi(S)+1 \leq \pi(S')$. To minimize the total number of parallel loops after fusion, we just have to minimize the label of the last loop, $\max_S \pi(S)$. To obtain the desired loop nest, we place in the same parallel loop all statements with the same value π, and parallel loops are ordered by increasing π. The formulation is thus:

$$\begin{cases} \max_S \pi(S) \ \text{ s.t. } \ \pi(S) \geq 0 \text{ and} \\ \forall S \overset{e}{\rightarrow} S' \ \text{ s.t. } \ e \text{ is not fusion–preventing}, \qquad \pi(S) \leq \pi(S') \\ \forall S \overset{e}{\rightarrow} S' \ \text{ s.t. } \ e \text{ is fusion–preventing}, \qquad \pi(S)+1 \leq \pi(S') \end{cases}$$

The reader can recognize a classical scheduling problem: $\pi(S)$ 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 Ref. [38] 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 McKinley and Kennedy, the total number of loops may not be minimal, but the number of parallel loops is and, therefore, the number of synchronizations.

5.2. 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 4.1). 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 4.2 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 $\boldsymbol{X}$ that will be used to generate the last loop, and we try to generate constants $\boldsymbol{\rho}_S$ 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 $\boldsymbol{\rho}_S$ and $\boldsymbol{\rho}_{S'}$ such that, for each dependence $e{:}S(\boldsymbol{I}) \rightarrow S'(\boldsymbol{J})$, $\boldsymbol{X}(\boldsymbol{J}-\boldsymbol{I})+\boldsymbol{\rho}_{S'}-\boldsymbol{\rho}_S=0$, i.e., so that the dependence becomes loop-independent. To make the link with Section 5.1, 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 $\boldsymbol{X}(\boldsymbol{J}-\boldsymbol{I})$ associated with the edge e. Remark that when G_a is acyclic, if considered as an undirected graph, one can always choose the constants $\boldsymbol{\rho}_S$ 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

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

that solves the problem: $\pi(S)$ 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 $\pi(S_e) = \pi(S'_e)$ only contains undirected cycles C such that $w(C) = 0$. Therefore, one can build the desired constants $\boldsymbol{\rho}_S$ such that for all edge $e \in G'_{\mathrm{a}}$, $w(e) + \boldsymbol{\rho}_{S'_e} - \boldsymbol{\rho}_{S_e} = 0$.

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 $\boldsymbol{\rho}_S$ for each vertex S, we define the quantity $w_{\boldsymbol{\rho}}(e)$ for each edge $e = (S_e, S'_e)$ by $w_{\boldsymbol{\rho}}(e) = 0$ if $w(e) + \boldsymbol{\rho}_{S'_e} - \boldsymbol{\rho}_{S_e} = 0$, and $w_{\boldsymbol{\rho}}(e) = 1$ otherwise. The weight of a path is defined as the sum of the weights $w_{\boldsymbol{\rho}}(e)$ of its edges. Then, finding values for $\boldsymbol{\rho}_S$ which minimize the maximal weight $w_{\boldsymbol{\rho}}$ 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 details in Ref. [37]. □

We illustrate the methods described in Sections 5.1 and 5.2 with the program of Fig. 12. Because of the nonzero dependences the simple fusion builds three different parallel loops (see Fig. 13) 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
```

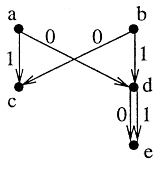

Fig. 12. Original code and its dependence graph.

```
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 $i \leq n$ then $a[i] = 1$
    if $1 < i$ then $b[i-1] = 1$
    if $i \leq n$ 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
```

```
$a[1] = 1$
$d[1] = a[1] + b[0]$
dopar $i = 2, n$
    $a[i] = 1$
    $b[i-1] = 1$
    $d[i] = a[i] + b[i-1]$
enddo
$b[n] = 1$
dopar $i = 1, n$
    $c[i] = a[i-1] + b[i]$
    $e[i] = d[i] + d[i-1]$
enddo
```

Fig. 13. One optimal solution for simple fusion, and two for fusion with loop bumping.

We show on this example (third code on Fig. 13) that, using loop peeling, one can remove the ‘if’ tests introduced by loop fusion. In fact this is always possible and easy as the tests we introduced are always simple functions of the loop bounds. The removal of ‘if’ tests is especially useful when the iteration domains are large.

6. 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.

References

[1] D.F. Bacon, S.L. Graham, O.J. Sharp, Compiler transformations for high-performance computing, ACM Comput. Surveys 26 (4) (1994) .
[2] Michael Wolfe, High Performance Compilers For Parallel Computing, Addison-Wesley Publishing, 1996.
[3] Y.-Q. Yang, C. Ancourt, F. Irigoin, Minimal data dependence abstractions for loop transformations, Int. J. Parallel Programming 23 (4) (1995) 359–388.
[4] Alain Darte, Frédéric Vivien, Parallelizing nested loops with approximation of distance vectors: a survey, Parallel Process. Lett. (1997).
[5] High performance Fortran forum, high performance Fortran language specification, Technical Report 2.0, Rice University, January 1997.
[6] Thomas Brandes, ADAPTOR Programmer's Guide—Version 4.0, German National Research Institute for Computer Science, March 1996.
[7] J.R. Allen, K. Kennedy, Automatic translation of Fortran programs to vector form, ACM Trans. Programming Languages Syst. 9 (4) (1987) 491–542.
[8] M.E. Wolf, M.S. Lam, A loop transformation theory and an algorithm to maximize parallelism, IEEE Trans. Parallel Distributed Syst. 2 (4) (1991) 452–471.
[9] Alain Darte, Frédéric Vivien, Optimal fine and medium grain parallelism detection in polyhedral reduced dependence graphs, Proceedings of PACT '96, IEEE Computer Society Press, Boston, MA, October 1996.
[10] P. Feautrier, Some efficient solutions to the affine scheduling problem: Part II. Multidimensional time, Int. J. Parallel Programming 21 (6) (1992) 389–420.
[11] Amy W. Lim, Monica S. Lam, Maximizing parallelism and minimizing synchronization with affine transforms, Proceedings of the 24th Annual ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages, January 1997.
[12] A.J. Bernstein, Analysis of programs for parallel processing, IEEE Trans. Electron. Comput. 15 (1966) 757–762.
[13] John R. Allen, Ken Kennedy, PFC: a program to convert Fortran to parallel form, Technical Report MASC-TR82-6, Rice University, Houston, TX, USA, 1982.
[14] L. Lamport, The parallel execution of DO loops, Commun. ACM 17 (2) (1974) 83–93.
[15] Michael Wolfe, Optimizing Supercompilers for Supercomputers, MIT Press, Cambridge, MA, 1989.
[16] François Irigoin, 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.
[17] P. Feautrier, Dataflow analysis of array and scalar references, Int. J. Parallel Programming 20 (1) (1991) 23–51.
[18] A. Darte, L. Khachiyan, Y. Robert, Linear scheduling is nearly optimal, Parallel Process. Lett. 1 (2) (1991) 73–81.
[19] A. Darte, F. Vivien, On the optimality of Allen and Kennedy's algorithm for parallelism extraction in nested loops, J. Parallel Algorithms Applicat. 12 (1–3) (1997) 83–112, Special issue on Optimizing Compilers for Parallel Languages.
[20] P. Feautrier, Some efficient solutions to the affine scheduling problem: Part I. One-dimensional time, Int. J. Parallel Programming 21 (5) (1992) 313–348.
[21] Alexander Schrijver, Theory of Linear and Integer Programming, Wiley, New York, 1986.
[22] Alain Darte, Georges-André Silber, Frédéric Vivien, Combining retiming and scheduling techniques for loop parallelization and loop tiling, Parallel Process. Lett. (1997) Special issue, to appear. Also available as Tech. Rep. LIP, ENS-Lyon, RR96-34.
[23] Corinne Ancourt, François Irigoin, Scanning polyhedra with DO loops, Proceedings of the 3rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, April 1991, pp. 39–50.
[24] François Irigoin, Pierre Jouvelot, Rémy Triolet, Semantical interprocedural parallelization: an overview of the PIPS project, Proceedings of the 1991 ACM International Conference on Supercomputing, Cologne, Germany, June 1991.
[25] W. Pugh, A practical algorithm for exact array dependence analysis, Commun. ACM 8 (1992) 27–47.

[26] Marc Le Fur, Jean-Louis Pazat, Françoise André, Commutative loop nest distribution, in: H.J. Sips (Ed.), Proc. of the Fourth Int. Workshop on Compilers for Parallel Computers, Delft, The Netherlands, December 1993, pp. 345–350.

[27] J.-F. Collard, P. Feautrier, T. Risset, Construction of DO loops from systems of affine constraints, Parallel Process. Lett. 5 (3) (1995) 421–436.

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

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

[30] P. Feautrier, Parametric integer programming, RAIRO Recherche Opèrationnelle 22 (1988) 243–268.

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

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

[33] J. Xue, Automatic non-unimodular transformations of loop nests, Parallel Comput. 20 (5) (1994) 711–728.

[34] Jean-François Collard, Code generation in automatic parallelizers, in: Claude Girault (Ed.), Proc. Int. Conf. on Application in Parallel and Distributed Computing, IFIP WG 10.3, North Holland, April 1994, pp. 185–194.

[35] Wayne Kelly, William Pugh, Evan Rosser, Code generation for multiple mappings, The 5th Symposium on Frontiers of Massively Parallel Computation, McLean, VA, February 1995, pp. 332–341.

[36] William Pugh, the Omega Team, World Wide Web document, url:http://www.cs.umd.edu/projects/omega/.

[37] Pierre Boulet, Alain Darte, Georges-André Silber, Frédéric Vivien, Loop parallelization algorithms: from parallelism extraction to code generation, Technical Report 97-17, LIP, ENS-Lyon, France, June 1997.

[38] Kathryn S. McKinley, Ken Kennedy, Maximizing loop parallelism and improving data locality via loop fusion and distribution, in: U. Banerjee, D. Gelernter, A. Nicolau, D. Padua, (Eds.), The Sixth Annual Languages and Compiler for Parallelism Workshop, Number 768 in Lecture Notes in Computer Science, Springer-Verlag, 1993, pp. 301–320.

[39] David Callahan, A global approach to detection of parallelism, PhD Thesis, Dept. of Computer Science, Rice University, March 1987.