Nested loop sequences : towards efficient loop structures in automatic parallelization

Zbigniew Chamski

▶ **To cite this version:**

Zbigniew Chamski. Nested loop sequences : towards efficient loop structures in automatic parallelization. [Research Report] RR-2094, INRIA. 1993. inria-00074578

HAL Id: inria-00074578
https://inria.hal.science/inria-00074578

Submitted on 24 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Nested Loop Sequences: Towards Efficient Loop Structures in Automatic Parallelization

Zbigniew Chamski

N° 2094

Octobre 1993

PROGRAMME 1

Architectures parallèles,
bases de données,
réseaux et systèmes distribués

1993

ISSN 0249-6399

Nested Loop Sequences: Towards Efficient Loop Structures in Automatic Parallelization

Zbigniew Chamski

Programme 1 — Architectures parallèles, bases de données, réseaux et systèmes distribués
Projet API

Rapport de recherche n° 2094 — Octobre 1993 — 16 pages

Abstract: An important problem in automatic parallelization of scientific programs is to generate loops from an algebraic description of the iteration domain. The usual technique is to produce a perfectly nested set of loops, whose bounds consist in maxima and minima of several affine functions. However, perfect loop nests suffer from the run-time overhead of evaluating bound expressions and do not allow to scan non-convex domains efficiently. In this paper we study a candidate loop structure for overcoming these problems. This structure, called *nested loop sequence* (NLS,) is defined as a sequence of DO loops whose bodies are nonempty sequences of DO loops. We propose an algorithm to compute a NLS scanning a given convex polyhedron, which overcomes the run-time overhead problem. The algorithm produces a loop structure in which the bounds of every loop consist each in a single affine function.

Key-words: loop generation, program transformation, linear programming, optimization.

(Résumé : tsvp)

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : (33) 99 84 71 00 – Télécopie : (33) 99 38 38 32

Séquences imbriquées de boucles : vers des structures de boucles efficaces pour la parallélisation automatique

Résumé : La restructuration automatique de programmes scientifiques nécessite la génération de nouvelles structures de boucles pour le programme final. Celle-ci se fait à partir d'une description algébrique du domaine d'itération. Habituellement, le compilateur génère un nid de boucles parfaitement imbriquées dans lequel les bornes sont des extrema de plusieurs fonctions affines. Cette approche conduit à des surcoûts importants à l'exécution, et ne permet pas le parcours de domaines d'itération non convexes. Dans ce document nous étudions une structure de boucles permettant de résoudre ces deux problèmes, appelée *séquence imbriquée de boucles.* Nous proposons un algorithme permettant de calculer une telle structure énumérant les points d'un domaine convexe, ce qui permet de résoudre le premier problème : l'algorithme produit une arborescence de boucles dans laquelle chacune des bornes consiste en une seule fonction affine.

Mots-clé : génération de boucles, transformation de programmes, programmation linéaire, optimisation.

1 Introduction

One of the challenging problems in automatic program parallelization is the generation of loop structures for the output program. The input program usually contains a *perfect nest* of loops scanning a convex set $\mathcal{D} = \{x \mid Ax + b \geq 0\}$ of integer points. During the parallelization process this set may be modified by some affine transformation, leading to a new iteration domain $\mathcal{D}' = \{x \mid A'x + b' \geq 0\}$, thus requiring the determination of a new loop structure. This problem arises also when generating parallel programs from formal specifications, where iteration domains are *originally* represented as unions of convex polyhedra.

Usually, the computation of the loop structure is performed using the Fourier-Motzkin pairwise elimination method as proposed by Irigoin in [9, 1] and by Wolf and Lam in [14], or its derivatives such as the Omega test ([11].) However, the former approach suffers from both the complexity of the Fourier-Motzkin elimination itself. Also, both approaches produce loop structures with significant run-time overhead due to the complexity of resulting bound expressions. While the former difficulty can be fairly reduced by using other linear programming algorithms (see [7, 3]), the latter is inherent to the perfect loop nest model.

The run-time overhead is mainly due to the fact that in a perfect loop nest, the bounds are given as extrema of several affine functions. All these functions must be evaluated each time the loop is entered, each function evaluation being equivalent to a dot product. While this is not significant for some applications, there are many others for which the bound computation overhead becomes critical: imaging, digital signal processing, distributed-memory transfer codes etc. Overheads close to 90% have been reported in codes produced by automatic parallelizers.

To reduce this overhead it is necessary to either simplify the bound expressions, or change the loop structure model used in loop generation. The former depends strongly on the shape of the domains and may become a strong limitation for several classes of problems, such as the generation of transfer codes in data-parallel programs. In this paper, we follow the latter approach, which opens new possibilities in terms of expressive power of the code that can be generated.

The perfect loop nest being too restrictive, we propose to use a more versatile construct, called *nested loop sequence* (NLS for short.) A NLS will give us the opportunity to refine the bound expressions in such a way that only a minimal set of bound expressions will be evaluated at a time. Also, this structure can potentially describe non-convex iteration domains, which is important in code generation from formal specifications.

The rest of the paper is organized as follows. We first give an example of the problems introduced by perfect loop nests (section 2), then we give several definitions (section 3.) Section 4 sets the guidelines of our algorithm and describes the Parametric Integer Programming algorithm that forms the base of our approach. In section 5 we devise an algorithm for computing the NLS

corresponding to a given convex iteration set. We conclude with an overview of other approaches to the loop generation problem (section 6) and several guidelines for future work, in particular an extension of the algorithm to non-convex domains (section 7.)

2 Example

To introduce our approach, let us consider an iteration set resulting from the hexagonal tiling of a domain (equation 1 and figure 1.)

$$\begin{array}{lcl} i & \geq & 0 \\ -i & \geq & 4 \\ j & \geq & 0 \\ -j & \geq & 4 \\ i-j & \geq & 2 \\ -i+j & \geq & 2 \end{array} \tag{1}$$

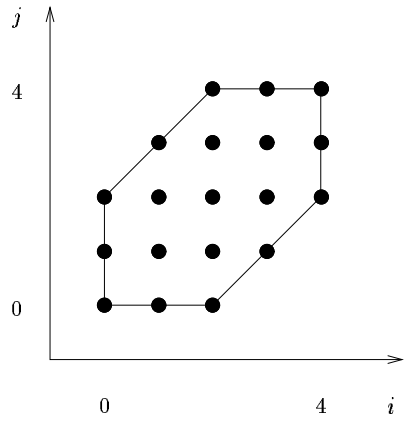


Figure 1: Iteration set example: a hexagonal tile

If one wants to scan the points of this set by first enumerating all successive values of the j index for each i, then the corresponding loop nest will have the following shape:

```
      DO 999 I = 0,4
        DO 998 J = MAX(0,I-2), MIN(4,I+2)
C         elementary computation
 998    CONTINUE
 999  CONTINUE
```

One can notice that for each value of `I`, two expressions must be evaluated in order to determine each of the bounds of the loop on `J`.

However, it is also easy to notice that if the loop on `I` is separated into two loops, say `DO I=0,1` and `DO I=2,4`, then the bounds of the `J` index simplify to `J=0,I+2` and `J=I-2,4`, respectively. This reduces the loop computation overhead by one half. The resulting loop structure is

```
      DO 9999 I = 0,1
        DO 9997 J=0,I+2...
C         elementary computation
 9997   CONTINUE
 9999 CONTINUE
      DO 9998 I = 2,4
        DO 9996 J=I-2,4
C         elementary computation
 9996   CONTINUE
 9998 CONTINUE
```

While the separation of the loop here is straigthforward, it is not the case in general. In the following, we propose a general method of loop splitting at any level, our objective being to obtain loop structures with bound expressions reduced each to a single affine function.

Before presenting the algorithm, let us give several definitions.

3 Definitions

It is assumed that reader is familiar with basic concepts of linear programming and linear algebra in general. For any definition not appearing below, please refer to any introductory book on linear algebra, such as [12].

A loop is 4-tuple $<i, l, u, b>$ where i is the loop index, l and u are lower and upper **bounds**, respectively, and b is a nonempty sequence of statements called loop **body**.

The loop body is executed once for each integer value comprised between the lower and the upper bound of the loop.

If the loop L appears in the body of another loop L', it is said to be an **inner** loop and its bounds may depend on the actual value of the index of the outer loop L'.

A **loop nest** is a set of loops such that some loops appear in the body of other loops. If for each loop in a nest, its body consists of either another loop or a sequence of statements, none of which is a loop, then such a nest is said to be **perfect**. Otherwise,

Definition 1 *If the body of any loop in a loop nest $\mathcal{L}$ is either a sequence of loops or contains no loops at all, then $\mathcal{L}$ is called a* **nested loop sequence** *(see fig. 2.)*

```
      DO 999 I=1,M
        DO 998 J=1,N
          DO 997 K=1,P
C           no loops here
            C = C + A(I,K)
                    * B(K,J)
 997      CONTINUE
 998    CONTINUE
 999  CONTINUE
```

a) a perfect loop nest

```
      DO 999 I=1,M1
        DO 9981 J=1,N1
          DO 9971 K=1,P1
C           no loops here
            C = C + A(I,K)
                    * B(K,J)
 9971     CONTINUE
          DO 9972 K=P1+1,P
C           no loops here
            C = C + A(I,K)
                    * B(K,J)
 9972     CONTINUE
 9981   CONTINUE
        DO 9982 J=N1+1,N
          ...
 9982   CONTINUE
 999  CONTINUE
      ...
```

b) a nested loop sequence

Figure 2: Examples of loop nests.

The **iteration space** of a loop nest is the set of integer points enumerated by its innermost loops, that is, the set of vectors consisting of the actual index values of all the loops for each execution of innermost loop bodies. E.g., for the loop nest from page 2, the iteration set is the polyhedron defined in equation 1 and depicted in fig. 1.

An important property of the execution of a perfect loop nest is that the points of its iteration space are enumerated in **lexicographical** order. The weak lexicographical ordering is a total order. Therefore, if a parallel enumeration of points has to be done (meaning a parallel execution of several loops,) the ordering must be restricted to a subset of point coordinates, called **time coordinates**, as opposed to the remaining ones, called **spatial coordinates**.

In the following, we will only deal with time coordinates. Spatial coordinates will eventually appear as problem parameters, leading to a parametric solution (that is, a program in which loop bounds depend on the coordinates of the processor it runs on.)

Any loop nest can be represented as a tree whose nodes are loop bodies (sequences of statements) labeled with indices and bounds. The tree representation for loop nests from figure 2 is given in figure 3.

4 The idea of the NLS construction algorithm

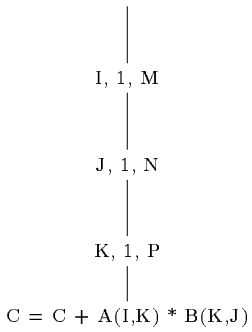
We have already seen with the example from section 2 that, in order to simplify bound expressions of a given loop, we need to separate an outer loop into an appropriate *sequence* of loops.

Intuitively, the separation must take place at the points where the actual bound function changes: if an upper bound is defined as `MIN(4, I+2)`, the regions to be distinguished are the one where `4` $\geq$ `I+2` and the one where `4` $<$ `i+2`. In other words, one must solve a problem of the form "find all x such that $\min(a(x), b(x), \ldots) = a(x)$" where a, b etc. are affine functions.

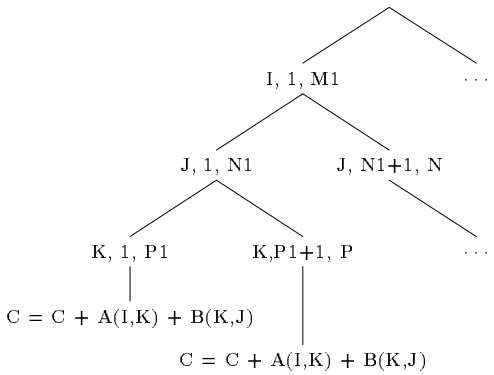
While this can be done be done by examining all the combinations of functions appearing in a given bound expression, it will be much more efficient to determine directly the conditional form of the extremum: "*if* $(b-a)(x) \geq 0$ *then* $a(x)$ *else* $b(x)$". This is best done by using the Parametric Integer Programming algorithm by Feautrier ([6, 8].)

4.1 Parametric Integer Programming (PIP)

The aim of the Parametric Integer Programming (PIP) algorithm is to compute the lexicographical extremum (min or max) of a convex polyhedron $\mathcal{F}(z) = \{x \mid Ax + Bz + c \geq 0,\ Cz + d \geq 0\}$. The results are given as selection trees whose nodes are guarded by affine functions of the parameters. The constraint system $Cz + d \geq 0$, applicable to parameters only, is called the **context** of the problem. It can be used to restrict the solution space when necessary.



a) a perfect loop nest



b) nested loop sequence

Figure 3: Tree representation of the trees from fig. 2.

The ILP mechanism underlying PIP is the parametric Dual Simplex algorithm, with a lexicographical objective function which ensures the termination of the computation and the uniqueness of the solution. Whenever the definition of the extremum depends on a constraint on the parameters, two subproblems are instantiated and explored, and are represented in the solution as a selection tree guarded by the corresponding constraint. For example, for the hexagonal tile from section 2, the (lexicographical) maximum of j as a function of i generated by PIP is "if $i - 2 \geq 0$ then 4 else $i + 2$".

The reader interested in the internals of the algorithm is invited to refer to the initial paper on PIP ([6].) Also, the robustness of the algorithm and its suitability for loop bound generation are discussed in [3] and [5].

5 The NLS construction algorithm

Our algorithm operates on a tree representation of loops. In the following, we will often identify a tree with the corresponding loop nest.

Let us first outline the algorithm. We will then describe in detail each of its subparts. Next, we will give a procedural description and correctness and termination proofs.

The loop tree is built starting from the outermost loops to the innermost ones. The input of the algorithm is the constraint system $Ax + b \geq 0$ defining the p-dimensional convex polyhedron to scan.

The exploration of each non-terminal node in the tree starts with the computation of bound expressions of its descendant using PIP. The context of the PIP computations is the set of bounds of all outer loops. The two selection trees obtained in this way are stored as attributes of the node.

The node is then refined as long as at least one of its bound expressions contains a conditional. The refinement consists in replacing a subtree containing the current node with a sequence of two subtrees scanning the same iteration domain, and such that each branch of the conditional holds on exactly one subtree. The level at which the duplication is to take place is defined by the shape of the constraint of the conditional.

Then, the exploration continues with the first node which is either unexplored or contains conditional bounds. The order in which nodes are selected (breadth-first or depth-first) is not meaningful. The computation stops when there is no unexplored nodes at any level from 0th (root) to the $p - 1^{st}$ level in the tree.

5.1 Algorithm components

We mentioned above two components of the algorithm: bound expression determination and node refinement (decomposition.) Let us discuss them in a more detailed way.

5.1.1 Determination of bound expressions

The bound expressions are extracted from the results of two PIP calls. Let q denote the nesting level of the current node. The bounds computed at this level are those of the $q + 1^{st}$ index expressed as functions of the q outer indices. Let $x_{i:j}$ denote the vector formed by the coordinates i through j of the vector x.

The problems submitted to PIP are formed as follows: given the domain definition $Ax + b \geq 0$ and the context $Cy + d \geq 0$ induced by outer loops, the matrix A is split into two submatrices A_P and A_V, consisting of columns $A_{*,1}$ to $A_{*,q}$ and $A_{*,q+1}$ to $A_{*,p}$, respectively. Let us split vector x into a parameter subvector $x_P = x_{1:q}$ and a variable subvector $x_V = x_{q+1,p}$. The queries consist

then in computing the lexicographical extrema of the polyhedron

$$\mathcal{D}(x_P) = \{x_V \mid A_V x_V + A_P x_P + b \geq 0,\ C x_P + d \geq 0\}.$$

The results returned by PIP express the extrema of the *vector* x_V. However, only the first coordinate of the result (that is, x_{q+1}) is kept.

5.1.2 Node refinement

As mentioned above, the refinement of a node is performed at a level defined by the shape of the condition used for the refinement. The last non-null coefficient in the constraint (except the constant) determines the innermost index that affects the value of the constraint. Hence, if r is the position of this index, the decomposition must be applied to the ancestor of the current node located at the rth level.

The nodes replacing the decomposed one are identical to that node except for:

- the upper bound of the first (leftmost) node and the lower bound of the second (rightmost) node, which are derived from the decomposition constraint,
- the conditional term used for the decomposition, which is replaced by its appropriate subterm.

To maintain the semantics of the initial (non-decomposed) loop, the order in which points of the iteration domain of the *sequence* of new loops "L_1; L_2" are scanned must be the same as in the initial loop L. Therefore, if the rth index is scanned in ascending order, the upper bound $u_1(x_{1:r-1})$ of L_1 and the lower bound $l_2(x_{1:r-1})$ of L_2 must satisfy

$$u_1(x_{1:r-1}) < l_2(x_{1:r-1}) \tag{2}$$

and

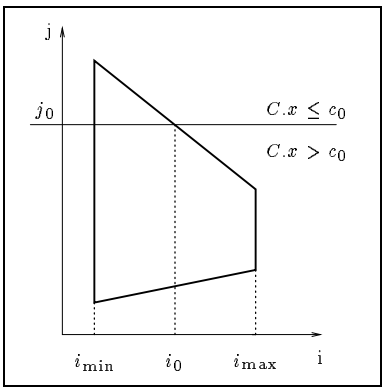
$$u_2(x_{1:r-1}) \geq l_1(x_{1:r-1}) - 1. \tag{3}$$

The former ensures the disjunction of the iteration domains of L_1 and L_2, whereas the latter ensures that no integer points are lost in the transformation.

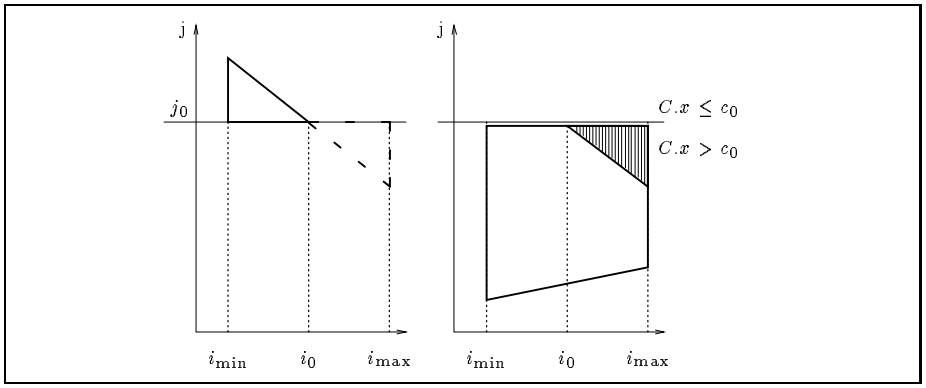
In practice, either of the bounds is directly given by the decomposition constraint which always has integer coefficients. The other bound is then obtained by adding ± 1 to its value, following the constraint type (maximizing/minimizing.) E.g., let the decomposition constraint be $x_r \geq C.x_{1:r-1} + c_0$. The lower bound of the second loop will be then equal to $C.x_{1:r-1} + c_0$, whereas the upper bound of the first loop will be $C.x_{1:r-1} + c_0 - 1$.

5.1.3 Validity of a constraint

The straight decomposition of a node whose bounds are single affine functions is not always legal. Consider the decomposition shown in figure 4. After the transformation, the union of the new iteration domains is actually larger that the initial domain (the hatched region has been included in the union.) It is therefore necessary to restrict the decomposition to the region where it is valid (that is, $i \leq i_0$.)



a) before decomposition



b) after decomposition along $C.x \leq c_0$

Figure 4: An invalid decomposition of a loop

Again, this is determined using the PIP algorithm. Given a node n at level r and a constraint $C(x) \geq 0$ applicable to this node, let inf_C and sup_C denote

the lower and upper bound functions implied by C. Let $lower_n$ and $upper_n$ be the lower and upper bound functions of n.

Definition 2 *The* **validity region** *of a constraint $C(x) \geq 0$ at node n is the set of points $x_{1:r-1}$ satisfying the current context and such that*

- *(upper bound validity condition)*

$$upper_n(x_{1:r-1}) - sup_C(x_{1:r-1}) + 1 \;\geq\; 0,$$

- *(lower bound validity condition)*

$$inf_C(x_{1:r-1}) - lower_n(x_{1:r-1}) + 1 \;\geq\; 0,$$

- *both $C(x) \geq 0$ and $-C(x) - 1 \geq 0$ are feasible in the current context.*

If the first condition fails or $sup_C(x_{1:r-1}) - x_r \geq 0$ is not feasible in the current context, the constraint is said to be **invalid for upper bound substitution**. *If the second condition fails or $x_r - inf_C(x_{1:r-1}) \geq 0$ is not feasible in the current context, the constraint is said to be* **invalid for lower bound substitution**.

In the region where a constraint is invalid for a bound, the corresponding conditional can be directly replaced by its appropriate branch. If the decomposition constraint is a minimizing one, i.e., is of the form $x_r - inf_C(x_{1:r-1}) \geq 0$, then

- if it is invalid for lower bound substitution, the conditional will be replaced by its "else" part,
- if it is invalid for upper bound substitution, the conditional will be replaced by its "then" part.

It is convenient to represent the validity information as a conditional term. If $C_L(x) \geq 0$ and $C_U(x) \geq 0$ are lower and upper bound validity conditions, respectively, then the conditional term will be

if $C_L(x) \geq 0$
  then **if** $C_U(x) \geq 0$
    then decompose
    else keep only the case where condition
      $x_r \leq sup_C(x_{1:r-1})$ holds
  else keep only the case where condition
    $x_r \geq inf_C(x_{1:r-1})$ holds

We can now give the complete algorithm. The loop tree is modified by side-effects and it is designated by its root node *root*. Each node is given three attributes: its bound expressions *lower* and *upper*, and a validity condition called *pending*. We don't describe formally the functions used by the algorithm since they are strongly implementation- and programming style dependent.

```
begin
  crnt := root;
  while crnt ≠ nil
    if never-explored(crnt) then
      (lower(crnt), upper(crnt)) :=
        compute-bounds(crnt) endif;
    if cond-free(lower(crnt)) and
      cond-free(upper(crnt)) and
      pending(crnt) = nil
    then
      adopt-node(crnt, lower(crnt), upper(crnt));
      crnt := next-to-explore(crnt);
    else
      cond :=
        select-cond(lower(crnt),
                    upper(crnt),
                    pending(crnt));
      valid := validity-conditions(crnt, cond);
      if valid = true then
        decompose-node(crnt, cond);
        crnt := next-to-explore(crnt);
      else
        pending(crnt) := valid;
        old := crnt;
        crnt = find-ancestor(crnt, cond);
        pending(crnt) :=
          remote-cond(position-in(old, crnt), valid);
      endif
    endif
  endwhile
end
```

Notes:

- we omitted the handling of the context, which can be directly obtained from the bounds of the ancestors of the current node;
- **adopt-node** creates a node whose bounds are given by the parameters and whose *pending* attribute is equal to *nil* ; the node is inserted as the only descendant of the node passed as the first parameter;
- **next-to-explore** returns *nil* if there is no further nodes to explore; a node will be explored if either some of its attributes contain conditionals, or it has no descendants and is located on a non-terminal level;

- **decompose-node** maintains the semantics of the loop it decomposes (see section 5.1.2;) if the current node is the root of a subtree, the entire subtree is replicated; the conditional chosen by **select-cond** is replaced by its appropriate branches in both new subtrees;
- if the conditional is of the form "remote-cond(*position, conditional*)", then the replacement is performed on the subtrees designated by *position*;
- **find-ancestor(***node, cond***)** returns the ancestor located at the level corresponding to the last non-null index coefficient in *cond* (see section 5.1.2.)

We can now sketch the correctness and termination proofs of the algorithm.

Correctness proof: for an unexplored node, the definition of a PIP problem solution ensures that the set of points satisfying the conditional bounds built by PIP is identical to the original iteration space (see [5].) Next, each decomposition is applied only if the decomposition constraint is valid, so the iteration space is not altered by a decomposition operation. Finally, the invalidity problem can be reduced to a loop decomposition at an outer level.

The algorithm stops when no node contains a conditional attribute and all nodes have their descendants determined. Therefore, the resulting loop structure contains no conditional bounds and scans the initial iteration space.

Termination proof: In [6], Feautrier shows that the PIP algorithm always terminates. On the other hand, the decomposition makes strictly decrease a metrics $\mathcal{M}(z)$ of the node z, consisting in the total number of conditions appearing in its attributes. In the case where the decomposition constraint is invalid, after a finite number of steps the node will be replaced by a sequence of nodes z'_i on which either one of the branches of the corresponding conditional is invalid (so $\forall i\ \mathcal{M}(z'_i) < \mathcal{M}(z)$), or the decompisition is valid and then z'_i can be replaced by a sequence of two nodes z''_1 and z''_2 such that $\mathcal{M}(z''_1) < \mathcal{M}(z)$ and $\mathcal{M}(z''_2) < \mathcal{M}(z)$.

The number of nodes added in each step being finite, the algorithm always terminates.

5.2 Example

Let us illustrate the operation of this algorithm on the introductory hexagonal tile example. The computation starts with the determination of the bounds of the first index i (with no parameters.) The lower and upper bounds produced by PIP are 0 and 4, respectively. As they contain no conditionals, the corresponding node can be directly instantiated and added to the tree as the only descendant of the root.

The loop tree can therefore be represented as

```
      DO 999 I = 0,4
C       further exploration needed
C       context: 0<=i<=4
 999 CONTINUE
```

The next node to explore is the node introduced above. The computation of the bounds of its descendants leaves two conditional expressions, "if $i-2 \geq 0$ then $i-2$ else 0" for the lower bound, and "if $i-2 \geq 0$ then 4 else $i+2$" for the upper bound. Let us start with the lower bound. The condition $i-2 \geq 0$ is valid (polyhedra $\{i \mid i-2 \geq 0,\ -i-1 \geq 0\}$ for the lower bound, and $\{i \mid -i+1 \geq 0,\ i-4 \geq 0\}$ for the upper bound are both empty, and both $i-2 \geq 0$ and $-i+1 \geq 0$ are feasible constraints in the current context.) Hence, the current node can be decomposed into a sequence of two nodes: $0 \leq i \leq 1$ and $2 \leq i \leq 4$. The loop tree becomes

```
     DO 998 I = 0,1
C      lower bound:  0
C      upper bound:  "if i-2 >= 0 then 4
C                                 else i+2"
C      context: 0<=i<=1
 998 CONTINUE
     DO 999 I = 2,4
C      lower bound:  i-2
C      upper bound:  "if i-2 >= 0 then 4
C                                 else i+2"
C      context: 2<=i<=4
 999 CONTINUE
```

Let us explore the first of the two nodes. The condition to eliminate is $i-2 \geq 0$. However, this condition is not feasible in the current context (the polyhedron $\{i \mid i-2 \geq 0,\ -i+1 \geq 0,\ i \geq 0\}$ is empty,) so it is invalid for lower bound substitution. Therefore, only the branch "else $i+2$" is to be kept. The node contains now no conditionals, so it can be instantiated leading to the tree

```
     DO 998 I = 0,1
       DO 997 J = 0, I+2
C        elementary computation comes here
 997   CONTINUE
 998 CONTINUE
     DO 999 I = 2,4
C      lower bound:  i-2
C      upper bound:  "if i-2 >= 0 then 4
C                                 else i+2"
C      context: 2<=i<=4
 999 CONTINUE
```

Following the above approach, we observe that in the second loop on i, the decomposition constraint is again invalid, but this time for the upper bound.

Therefore, the conditional bound will be directly replaced by its "then" part, leading to the final shape of the loop tree:

```
     DO 998 I = 0,1
       DO 997 J = 0, I+2
C        elementary computation comes here
 997   CONTINUE
 998 CONTINUE
     DO 999 I = 2,4
       DO 996 J = I-2, 4
C        elementary computation comes here
 996   CONTINUE
 999 CONTINUE
```

6 Related work

We mentioned already in the introduction the work on the use of the Fourier-Motzkin elimination method by Irigoin and Ancourt ([9, 1],) and by Wolf and Lam ([14].) Several algorithms initially designed for dependence analysis have been used to generate the bounds of a perfect loop nest, such as the Omega test of Pugh ([11].) However, as Maydan *et al.* point out in [10], this particular class of algorithms is designed to solve exactly integer programming problems on small sets of data. The scalability of such algorithms in terms of coefficient values and problem size has therefore to be shown.

The PIP algorithm has been used to generate perfect loop nests ([7, 3, 5].) In these solutions, the conditional extrema representations produced by PIP are either used directly as they come ([6],) or are collapsed into usual min and max expressions ([3, 5, 4].) [3] contains a quantitative study of two PIP-based approaches proposed in [7], and [4] compares their performance with that of the Fourier-Motzkin elimination from [1]. The results achieved in the tests show that the PIP algorithm offers indeed good scalability and allows to solve fairly large problems.

In her PhD thesis ([13],) N. Tawbi deals with the counting of points in an iteration domain and devises an algorithm similar to ours. However, her algorithm differs in that it uses a combinatorial scheme to build the subdomains, and therefore allows unnecessary decompositions of the domain leading to an additional overhead which contradicts our hypotheses.

In [2], Carr and Kennedy use the index space splitting transformation to generate blocked versions of several numerical algorithms. This transformation can be seen as the application of elementary steps of the NLS generation algorithm at specific points in a loop structure. Therefore, it appears that our algorithm can be used to further refine the blocking of loops in their approach.

7 Future extensions

One can note that an NLS can be used to scan non-convex domains. This property is extremely important in automatic generation of parallel programs ([3],) where several non-disjoint convex domains must be scanned using a single loop structure. Since existing methods allow only to scan convex domains, they may potentially lead to a very inefficient code.

Possible solutions to this problem include the scanning of the convex hull of (or the smallest rectangle parallelepiped containing) the union of elementary domains, using appropriate membership tests. However, this may imply the systematic evaluation of *all* constraints defining any of the domains, thus leading to a prohibitive overhead.

Our NLS generation algorithm appears as a good candidate for the non-convex domain scanning. We are currently investigating its actual applicability, following a conjecture formulated in [3].

Space-efficiency problems due to code replication are another direction of research. Since loop decompositions are more expensive at the outermost levels, a proposed approach is to explore a hybrid loop structure in which outer loops will use extremum-based bounds, whereas inner loops will be decomposed in the usual way.

8 Conclusion

In this paper we introduced a new loop structure, called nested loop sequence, allowing to generate more time-efficient code for parallelized applications. We also presented an algorithm for generating such loop structures. Our algorithm produces a nested loop sequence in which all loop bounds are reduced to a single affine function. We are currently studying the quantitative properties of this algorithm and its use in scanning non-convex polyhedral domains.

References

[1] C. Ancourt and F. Irigoin. Scanning polyhedra with DO loops. In *ACM/SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pages 39–50, ACM, June 1991.

[2] S. Carr and K. Kennedy. Compiler blockability of numerical algorithms. In *Supercomputing '92*, pages 114–124, IEEE, Minneapolis, USA, November 1992.

[3] Z. Chamski. *Environnement logiciel de programmation d'un accélérateur de calcul parallèle*. PhD thesis, Université de Rennes I, February 1993.

[4] Z. Chamski. Fast and efficient generation of loop bounds. Proceedings of *ParCo '93*, Elsevier Science Publishers (North Holland), to appear.

[5] J.-F. Collard, P. Feautrier, and T. Risset. *Construction of DO Loops from Systems of Affine Constraints.* Research Report 93-15, École Normale Supérieure de Lyon, Lyon, France, May 1993.

[6] P. Feautrier. Parametric integer programming. *Recherche Opérationnelle/Operations Research*, 22(3):243–268, 1988.

[7] P. Feautrier. Semantical analysis and mathematical programming. Application to parallelization and vectorization. In M. Cosnard et al., editors, *Parallel and Distributed Algorithms*, pages 309–320, Elsevier Science Publishers B. V. (North-Holland), 1989.

[8] P. Feautrier and N. Tawbi. *Résolution de systèmes d'inéquations linéaires ; mode d'emploi du logiciel PIP.* Technical Report, Laboratoire MASI, Univ. Pierre et Marie Curie, Paris, France, December 1989.

[9] F. Irigoin. *Partitionnement des boucles imbriquées. Une technique d'optimisation pour les programmes scientifiques.* PhD thesis, Univ. Pierre et Marie Curie, Paris, France, June 1987.

[10] D. E. Maydan, J. L. Hennessy, and M. S. Lam. Efficient and exact data dependence analysis. In *SIGPLAN 1991 Conference on Programming Language Design and Implementation*, pages 1–14, ACM, Toronto, Canada, June 1991.

[11] W. Pugh. A practical algorithm for exact array dependence analysis. *Communications of the ACM*, 35(8):102–114, August 1992.

[12] A. Schrijver. *Theory of Linear and Integer Programming. Wiley Interscience Series in Discrete Mathematics*, John Wiley and Sons, 1986.

[13] N. Tawbi. *Parallélisation automatique : estimation des durées d'exécution et allocation statique de processeurs.* PhD thesis, Université Paris VI, September 1991.

[14] M. J. Wolf and M. S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):452–471, October 1991.

Unité de recherche INRIA Lorraine, Technôpole de Nancy-Brabois, Campus scientifique,
615 rue de Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 46 avenue Félix Viallet, 38031 GRENOBLE Cedex 1
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105,
78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS
Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
(France)
ISSN 0249-6399