Scanning polyhedra without Do-loops

Pierre BOULET
LIFL, USTL
Bâtiment M3, Cité scientifique
59655 Villeneuve d'Ascq Cedex, France
Pierre.Boulet@lifl.fr

Paul FEAUTRIER
Laboratoire PR*i*SM
Université de Versailles-St Quentin en Yvelines
Bâtiment Descartes, 45, av. des États-Unis
78035 Versailles Cedex, France
Paul.Feautrier@prism.uvsq.fr

Abstract

We study in this paper the problem of polyhedron scanning which appears for example when generating code for transformed loop nests in automatic parallelization. After a review of related works, we detail our method to scan affine images of polyhedra. After some experimental results we show how our method applies to unions of affine images of polyhedra.

We have taken the option to generate low level code without loops. This has allowed us to have a completely general and fully parameterized method without losing efficiency.

1. Introduction

In the field of automatic parallelization, efforts have been focused on parallelizing loops because they concentrate most of the computing time in a small number of statements. The polytope model has been devised in order to fully analyze loops. In this model, the iteration domain (the set of operations) of a statement is described as a parameterized polyhedron. Each operation is associated to an iteration vector whose components are the values of the surrounding loop counters. Translating the loop bounds into inequalities gives a polyhedron which the iteration vector must belong to.

In order to exhibit parallelism, one applies transformations to the iteration space. These transformations are usually deduced from attempts to optimize the efficiency of the transformed program. Scheduling, for instance, aims at minimizing the running time on a parallel computer, while mapping aims at minimizing communications. All in all, these transformations usually are affine transformations. They map the original iteration domain of each statement to a new iteration domain defined as the set of images of the points of the original domain. We will consider here only integral affine transformations, which means that the image domain is a set of integer points.

The program which results from a given transformation is obtained by writing code for enumerating, or scanning in a given order the resulting iteration domains. This problem has been the subject of a large number of papers, starting with Irigoin's thesis [9] and the seminal paper [1]. The order in which the points are enumerated is relevant since one of the constraints which are imposed on the transformations is that the image set must be enumerated in lexicographic order. In fact some dimensions of the image iteration space are time dimensions which are sequential while some others are space dimensions which are parallel. The originality of our method is that we do not try to generate a new loop nest to be compiled later, but rather to directly generate low level code. This gives us more freedom in the structure of the generated code while retaining good efficiency. Observe that the usual invective against `GOTO`'s does not apply in our case: the code we generate is not for human consumption. It is just an intermediate representation for the use of a compiler backend.

The organization of the paper is as follows: in Sect. 2, we present previous solutions to the same problem; in Sect. 3 we study the case of a single linearly bounded lattice, which we illustrate with some experiments in Sect. 4. We then show in section 5 how the method presented before can be extended to handle the general case of the union of linearly bounded lattices and we conclude in Sect. 6.

2. Related work

The problem of scanning polyhedra first arose in relation with the loop inversion transform. While it is evident that the "inverse" of

```
do i = 1,n          do j = 1,m
  do j = 1,m    is    do i = 1,n
    S                   S
  end do              end do
end do              end do
```

it requires some thought to see that the inverse of

```
do i = 1,n          do j = 1,n
  do j = i,n    is    do i = 1,j
    S                   S
  end do              end do
end do              end do
```

This kind of problem occurs either when the polyhedron to be scanned is given without any reference to a loop nest (for instance when one use specification languages like ALPHA) or when the loop nest is submitted to a unimodular transform. This situation is characterized by the fact that *all* integer points in the given polyhedron are to be visited. In this form, the problem was first solved by Irigoin in [9]; see also [8, 1, 5].

However, not all parallelizing transformations are unimodular. They may even be singular: this situation occurs when constructing communication loops [11]. The solution is to write the transformation $T = HU$ where U is unimodular and H is a Hermite normal form of T. One first scans the image of the iteration space by U, as above, then apply H, which has the property of being monotone, increasing with respect to lexicographic order. See [6, 12, 13].

In the general case, there are several statements which may be subjected to different transformations. One has to scan the union of the images of several polyhedra, not necessarily of the same dimension. All solutions which have been proposed [3, 10, 4] are compromises between code size and performance, with no clear way of selecting an optimum.

In all cases, the result is obtained by combining several algorithms: Hermite normal form construction, Fourier-Motzkin elimination, various methods for splitting domains. Our aim here is to give just one algorithm for handling all cases. In the interest of clarity, we will nevertheless split the presentation in two parts: firstly, the case of a linearly bounded lattice (LBL), and secondly the case of a union of linearly bounded lattices. Scanning a polyhedron is just a particular case of scanning an LBL.

3. Scanning Linearly Bounded Lattices

We first describe in this section the problem we consider (see section 3.1), then the algorithm used to solve it (see section 3.2) and finally the code generation scheme used by this algorithm (see section 3.3).

3.1. Problem specification

We consider here the problem of scanning a *linearly bounded lattice* in *lexicographic order*.

Definition 3.1 (Linearly Bounded Lattice). A linearly bounded lattice $\mathcal{L}$ is a set of integer points verifying a system of affine inequalities as follows:

$$\mathcal{L} = \{x \in \mathbb{Z}^n \mid \exists y \in \mathbb{Z}^m, Ax + By \leq c\}$$
$$\text{where } n, m, p \in \mathbb{N}^*, A \in \mathbb{Z}^p \times \mathbb{Z}^n,$$
$$B \in \mathbb{Z}^p \times \mathbb{Z}^m \text{ and } c \in \mathbb{Z}^p \ .$$

We will in fact handle parameterized linearly bounded lattices:

$$\mathcal{L}(z) = \{x \in \mathbb{Z}^n \mid \exists y \in \mathbb{Z}^m, Ax + By + Cz \leq d\}$$
$$\text{where } n, m, p, q \in \mathbb{N}^*, A \in \mathbb{Z}^p \times \mathbb{Z}^n,$$
$$B \in \mathbb{Z}^p \times \mathbb{Z}^m, C \in \mathbb{Z}^p \times \mathbb{Z}^q \text{ and } d \in \mathbb{Z}^p \ .$$

Here z is a vector of parameters whose values are in a polyhedron $\mathcal{D}$ defined as:

$$\mathcal{D} = \{z \in \mathbb{Z}^q \mid Ez \leq f\} \ .$$

In the following, all the LBLs we will encounter will be parameterized LBLs.

Definition 3.2 (lexicographic order).
The lexicographic order $\prec$ over $\mathbb{Z}^m$ can be expressed as:

$$(x_1, \ldots, x_m) \prec (y_1, \ldots, y_m) \iff$$
$$\exists k \in \{0, \ldots, m-1\}$$
$$\mid x_1 = y_1, \ldots, x_k = y_k, x_{k+1} < y_{k+1} \ .$$

We note $\min_\prec$ the lexicographic minimum.

3.2. Resolution method

The basic idea for the enumeration of the LBL $\mathcal{L}(z)$ is to build a function, "next", which, given a point in $\mathcal{L}(z)$, returns the next higher point in $\mathcal{L}(z)$ according to lexicographic order.

To initialize this process, we have to build a constant, "first", which is the lexicographic minimum of $\mathcal{L}(z)$. "first" is defined as:

$$\text{first} = \min_\prec \{y \in \mathcal{L}(z)\} \ .$$

Note. This kind of problem is a parameterized linear program that can be solved using a tool such as PIP [7]. PIP computes lexicographic minima of domains defined by integer linear inequations. The solutions PIP returns are in the form of a *quasi-affine selection tree* (QUAST).

Indeed, depending on the values of the parameters, a solution may have different values. It has been shown that these values can be expressed as linear expressions of the parameters in polyhedral subdomains. Thus the QUAST structure is a selection (`if then else`) tree describing these subdomains and the associated solutions. Two branches of this tree are separated by a linear predicate defining an hyperplane, thus delimiting two subdomains of the parameter domain. The leaves of this tree are linear expressions of the searched minimum. We note $\perp$ when there is no solution for a given subdomain. Some problems may require the presence of some integer modulos at some depth in the selection tree. These modulos add complexity to the generated code.

Once we know how to compute "first", "next" can be described in the following way:

$$\begin{aligned} \mathrm{next} : \mathcal{L}(z) &\to \mathcal{L}(z) \cup \{\perp\} \\ x &\mapsto \min_{\prec}\{y \in \mathcal{L}(z) \mid x \prec y\} \quad . \end{aligned}$$

So, we have to solve the following problem:

find the minimum x' subject to the constraints:

$$\left\{ \begin{array}{l} z \in \mathcal{D} \\ x \in \mathcal{L}(z) \\ x' \in \mathcal{L}(z) \\ x \prec x' \end{array} \right. \iff \left\{ \begin{array}{l} Ez \leq f \\ Ax + By + Cz \leq d \\ Ax' + By' + Cz \leq d \\ x \prec x' \end{array} \right.$$

This problem is non-linear due to the constraint $x \prec x'$. We have to decompose it into smaller linear problems and to compose their results. To build these smaller problems, we use definition 3.2 to decompose the constraint $x \prec x'$ into the disjunction:

$$\begin{array}{ll} & x_1 < x'_1 \\ \text{or} & x_1 = x'_1, x_2 < x'_2 \\ & \vdots \\ \text{or} & x_1 = x'_1, \ldots, x_{m-1} = x'_{m-1}, x_m < x'_m \end{array}$$

We can now build the linear subproblems $\mathcal{P}^1, \ldots, \mathcal{P}^m$. Here is the general form of $\mathcal{P}^k$:

minimize x' subject to the constraints:

$$\left\{ \begin{array}{l} Ez \leq f \\ Ax + By + Cz \leq d \\ Ax' + By' + Cz \leq d \\ x_1 = x'_1 \\ \vdots \\ x_{k-1} = x'_{k-1} \\ x_k < x'_k \end{array} \right.$$

Using the results of these parameterized linear problems, we can now generate the iteration code.

The abstract program looks like:

```
    x = first
1   if x = ⊥ then goto 2
    loop body
    x = next(x)
    goto 1
2
```

This prescription is sufficient when one just has to write sequential code, as for instance scatter/gather code in communication routines. But one may wonder how parallelism will be expressed in this framework. In the standard scheme, those loops whose counter is not one of the components of time are flagged as parallel. One then relies on a low level parallel compiler to write the actual parallel code.

In our context, the solution is to select some of the variables as virtual processor names, and to scan the remaining variables with the virtual processor names as parameters. This will naturally generate SPMD code. If p is the virtual processor name, and if we want to use blocking in the virtual processor space, we just add the constraints: $a \leq p \leq b$, a and b being new parameters. Our system then automatically writes the virtualization loop.

The next step is to insert synchronization primitives. In the case of a distributed memory machine, synchronization is a byproduct of message passing. For a global memory machine, one has to use a new synchronization primitive: `synch(t)`.

Each processor has its own virtual clock in shared memory. The `synch` primitive first sets this clock to `t`, then enters a busy waiting loop. At each iteration, the processor computes (redundantly) the minimum of all clocks. It leaves the waiting loop iff its clock is equal to the said minimum.

Note. The above is just a definition of the semantics of `synch`. Its performance can be enhanced in various ways: relinquishing the processor in coarse grained situations, logarithmic update of the minimum, and others.

One may extend this definition to multidimensional time. This being done, one just has to insert a `synch` in the above code at each point where a component of time is modified. Since the `synch` primitive has the ability of "jumping ahead" in time, our parallel program will be at least as efficient as an implementation with barriers, in which there are as many synchronization operations as there are time steps.

3.3. Code generation algorithm

Let us now consider the iteration code. Ideally, this code should be as efficient as a loop would be, in the case when a loop can be designed to iterate over the same set of points $\mathcal{L}(z)$.

First we can observe that any solution of the problem $\mathcal{P}^k$ is lexicographically greater than any solution of the subproblem $\mathcal{P}^l$ when $k < l$. This comes from the definition of these problems. Indeed, in any solutions x'^k of $\mathcal{P}^k$ and x'^l of $\mathcal{P}^l$, $\forall i < k, x'^k_i = x_i$, $x'^k_k > x_k$ and $\forall i < l, x'^l_i = x_i$. As $k < l$, $\forall i < k, x'^k_i = x'^l_i$ and $x'^k_k > x'^l_k$, which means $x'^k \succ x'^l$. This allows us to seek the next point in the domain by first looking for solutions of $\mathcal{P}^m$, then, if there is no solution, of $\mathcal{P}^{m-1}$, and so on until $\mathcal{P}^1$. If this last problem has no solution, then we have finished scanning the set $\mathcal{L}(z)$.

In the simple example where the given polyhedron is a cube, $\{(i,j,k) \mid 1 \leq i,j,k \leq n\}$, the scanning code is as follows.

```
        i = 1         \
        j = 1         | first
        k = 1         /
2       CONTINUE
c   --- loop body ---
        if (k.le.n-1) then    \
          k = 1+k             |
          GOTO 2              |
        endif                 |
        if (j.le.n-1) then    |
          k = 1               | next
          j = 1+j             |
          GOTO 2              |
        endif                 |
        if (i.le.n-1) then    |
          k = 1               |
          j = 1               |
          i = 1+i             |
          GOTO 2              |
        endif                 /
1       CONTINUE
```

Remark. The storage of the new values of `i`, `j` and `k` — which represent x_1, x_2 and x_3— is done in reverse order to avoid using temporary variables. This is not useful on this particular example but the advantage of this method can be seen in more complicated examples. We have also suppressed useless storage statements like `i = i`.

An other important speed factor in loops is that loop bounds are only evaluated once at the beginning of the loop. We can refine our code generation scheme to do the same here. In each predicate of a conditional, one has just to compute the invariant part beforehand and store it in some variable. Here is what our example finally looks like:

```
        i = 1
        j = 1
        k = 1
        b3 = -1+n
4       b2 = -1+n
3       b1 = -1+n
2       CONTINUE
c   --- loop body ---
        if (k.le.b1) then
          k = 1+k
          GOTO 2
        endif
        if (j.le.b2) then
          k = 1
          j = 1+j
          GOTO 3
        endif
        if (i.le.b3) then
          k = 1
          j = 1
          i = 1+i
          GOTO 4
        endif
1       CONTINUE
```

Note. If `i` is the time variable, one just has to insert a `synch(i)` just after statement 4.

The generated code has the following general form:

```
        x⃗ = first
        computation of level 1 constants
label_1 computation of level 2 constants
label_2 computation of level 3 constants
         ...
label_m-1 computation of level m constants
label_m loop body
        next_m   ⇀ goto label_m
        next_m-1 ⇀ goto label_m-1
         ...
        next_1   ⇀ goto label_1
```

where "$\text{next}_i \rightharpoonup$ `goto` label_i" corresponds to a selection tree (`if then else`) which translates the QUAST next_i. At each leaf of this selection tree, there are, first, the storage of the new values of the indices ($\vec{x}$), and next, a "`goto` label_i" statement to start the next iteration.

4. Experimental results

We study here how the code we generate compares with loop methods in terms of compactness and speed.

4.1. Simple 3D example: *permutation*

The domain we consider here is the affine image of the polyhedron:

$$\begin{cases} 1 \leq i \leq n \\ 1 \leq j \leq n \\ 1 \leq k \leq i+j \end{cases}$$

by the permutation

$$\begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}.$$

Any (good) loop transformation method would produce the following code:

```
        do x = 1, 2*n
          do y = max(1,x-n), n
            do z = max(1,x-y), n
c   --- loop body ---
            enddo
          enddo
        enddo
```

Here is the code we produce:

```
        x = 1
        y = 1
        z = 1
        b9 = -1+2*n
        b8 = 1-n
8       b7 = -1+n
        b5 = -2+x
        b6 = -1+x
7       b4 = -1+n
6       CONTINUE
c   --- loop body ---
        if (z.le.b4) then
          z = 1+z
          GOTO 6
        endif
        if (y.le.b7) then
          if (y.ge.b5) then
            z = 1
            y = 1+y
            GOTO 7
          else
            z = b6-y
            y = 1+y
            GOTO 7
          endif
        endif
        if ((n-x).ge.0) then
          z = x
          y = 1
          x = 1+x
          GOTO 8
        else
          if (x.le.b9) then
            z = n
            y = b8+x
            x = 1+x
            GOTO 8
          endif
        endif
5       CONTINUE
```

The two codes share a common structure. Hence, the execution times are similar with a slight advantage to the second one. The running times and code sizes are shown in Tab. 1.

time (s) / size (bytes)	loops	gotos
without opt.	1.87 / 2,192	1.75 / 3,300
with opt.	0.14 / 1,416	0.12 / 1,668

Table 1. Execution times and object code sizes for the permutation example

All the execution times have been measured on a SUN SPARCstation 20. All the programs have been compiled with the SUN f77 FORTRAN compiler. We have compiled our programs both without any compiler optimization and with the `-O` option.

These results confirm that the two codes share a common structure, but as we generate lower level code, we are able to have slightly better execution times.

Remark. Further optimizations such as replacing non strict inequalities by strict ones to remove some intermediate variables or merging some intermediate variables with the same value may reduce the non optimized running time but do not improve the optimized running time. This is due to the good optimizations done by the compiler. These optimizations can even increase the optimized running time because they reduce the number of variables, which reduces opportunities to allocate variables to registers.

Although the Fortran codes have different lengths, the object codes generated have nearly the same size with a slight advantage to the loop version.

4.2. Lefur's Example P1

This more complex example is detailed in an extended version of this article [2]. In this case, the goto code is slightly slower than the do-loop code.

5. Extension to unions of linearly bounded lattices

We study here how the method presented in Sect. 3 can be extended to deal with unions of LBLs, which appear when generating code for non perfect loop nests where each statement may be subjected to a different affine transformation: each LBL is the image of the iteration domain of some statement by an affine transformation.

5.1. Preliminary remarks

The basic idea is to apply the previous method to each LBL and to combine the results. There are however some issues to deal with:

LBLs of different dimensions. To be able to speak about union of sets, we must deal with sets of the same dimension. If one set has less dimensions than the others, one can just complete the missing dimensions with a constant. The choice of this constant is arbitrary; one can set it to 0.

Reducing the complexity of the computation. We have remarked that computing a minimum of several QUASTs can be costly, as much during code generation as during the execution of this code. So one of our aims when developing the code generation algorithm for unions of LBLs has been to avoid computing these minima.

Code duplication. Code duplication is also an issue when dealing with this kind of transformation. Though it can be costly, we have not focused on its complete elimination.

5.2. Formal method

Considering the previous remarks, we have designed a method to iterate over unions of LBLs.

Let us consider that we have p LBLs $\mathcal{L}_1(z), \ldots, \mathcal{L}_p(z)$, depending on some parameters $z \in \mathcal{D}$. These LBLs are defined by:

$$\mathcal{L}_i(z) = \{x \in \mathbb{Z}^n \mid \exists y \in \mathbb{Z}^{m_i}, A_i x + B_i y + C_i z \leq d_i\} \ .$$

We note $\mathcal{U}(z) = \bigcup_{1 \leq i \leq p} \mathcal{L}_i(z)$.

We use here the same method as in Sect. 3.2, by building a constant, "first", and a function, "next". "first" is defined as: $\mathrm{first} = \min_{\prec}\{y \in \mathcal{U}(z)\}$, and as before, this is an integer programming problem directly solvable by PIP. Function "next" is defined as:

$$\begin{aligned} \mathrm{next} : \mathcal{U}(z) &\to \mathcal{U}(z) \cup \{\bot\} \\ x &\mapsto \min_{\prec}\{y \in \mathcal{U}(z) \mid x \prec y\} \ . \end{aligned}$$

To compute this function, we decompose it into linear integer programming problems.

If we suppose that, for a given point x of $\mathcal{U}(z)$, we know which $\mathcal{L}_i(z)$ it belongs to, $\mathrm{next}(x)$ can be expressed as: $\mathrm{next}_i(x) = \min_{\prec_{j \in \mathbb{N}_p^*}} \mathrm{next}_{i,j}(x)$ for any i such that $x \in \mathcal{L}_i(z)$ and where $\mathrm{next}_{i,j}$ is defined by:

$$\begin{aligned} \mathrm{next}_{i,j} : \mathcal{L}_i(z) &\to \mathcal{L}_j(z) \cup \{\bot\} \\ x &\mapsto \min_{\prec}\{y \in \mathcal{L}_j(z) \mid x \prec y\} \ . \end{aligned}$$

As in Sect. 3.2, we decompose the constraint $x \prec y$ into k linear constraints to build the linear problems $\mathcal{P}^k_{i,j}, 1 \leq i, j \leq p, 1 \leq k \leq n$ whose solutions are the following point in $\mathcal{L}_j$ whose first different coordinate is the k-th, considering that the current point is in $\mathcal{L}_i$. The general form of $\mathcal{P}^k_{i,j}$ is:

minimize x' subject to the constraints:

$$\begin{cases} Ez \leq f \\ A_i x + B_i y + C_i z \leq d_i \\ A_j x' + B_j y' + C_j z \leq d_j \\ x_1 = x'_1 \\ \vdots \\ x_{k-1} = x'_{k-1} \\ x_k < x'_k \end{cases}$$

We then combine the results of these problems to find the following point, $\mathrm{next}^k_i(x)$, in $\mathcal{U}(z)$ whose first different coordinate is the k-th, considering that the current point x is in a selected $\mathcal{L}_i(z)$: it is the minimum of the solutions of the $\mathcal{P}^k_{i,j}$ for all j. Such a minimum of QUASTs is a QUAST. We label each branch of this minimum QUAST by the indices j of the QUASTs it comes from, i.e. the statements which should be executed at the corresponding point. This branch labeling allows us to know which $\mathcal{L}_j(z)$ the following point belongs to.

Taking the minimum of the $\mathrm{next}^k_i(x)$ gives us $\mathrm{next}_i(x)$. These functions, next^k_i, are sufficient to compute function next in all cases. In fact, let us suppose that we know, for the current point x, the set $\mathcal{I}_x(z) = \{i \mid x \in \mathcal{L}_i(z)\}$. The following point is one of the $\mathrm{next}_i(x)$ for $i \in \mathcal{I}_x(z)$. Indeed, as the problems $\mathcal{P}^k_{i,j}$ differ only by their context for different $i \in \mathcal{I}_x(z)$, all their solutions are correct. So all the $\mathrm{next}_i(x)$ with $i \in \mathcal{I}_x(z)$ are the point following x in $\mathcal{U}(z)$. And, by reading the labels of the QUAST branches, we can now

identify $\mathcal{I}_{\mathrm{next}(x)}(z)$. By induction, we can iterate all the points of $\mathcal{U}(z)$ given the computation of all the $\mathcal{P}_{i,j}^k$.

We only lack the knowledge of $\mathcal{I}_{\mathrm{first}}(z)$. The solution is to decompose the linear problem $\min_{\prec}\{y \in \mathcal{U}(z)\}$ into $\min_{\prec}\{\min_{\prec}\{y \in \mathcal{L}_i(z)\} \mid i \in \mathbb{Z}_p^*\}$ and to label the branches as above.

Let us clarify this on a simple example :

Example 5.1 (Simple 2D example).
Let us consider two statements ① and ② whose iteration domains are $D_1 = D_2 = \{(a,b) \mid 1 \le a \le n, 1 \le b \le n\}$ and that are transformed by, respectively:

$$f_1 : (a,b) \mapsto (3a,b) \text{ and } f_2 : (a,b) \mapsto (3a+1,b).$$

k	i	next_i^k
2	1	**if** $b \le n-1$ **then** $a, b+1\,\{①\}$ **else** $\perp$
	2	**if** $b \le n-1$ **then** $a, b+1\,\{②\}$ **else** $\perp$
1	1	**if** $a \le 3n-3$ **then** $a+1, 1\,\{②\}$ **else** $\perp$
	2	**if** $a \le 3n-2$ **then** $a+2, 1\,\{①\}$ **else** $\perp$

Table 2. next_i^k**.**

Table 2 shows the next_i^k computed for this particular case. The symbols ① and ② label the leaves of next_i^k, indicating which statement has to be executed at the corresponding point.

$\min_{\prec} f_1(D_1)$	$\min_{\prec} f_2(D_2)$	start
$3,1$	$4,1$	$3,1\,\{①\}$

Table 3. Starting point.

To determine the starting point, we have to solve the minima of $f_1(D_1)$ and $f_2(D_2)$ and their minimum. See Tab. 3 for these computations.

We have now computed all the information needed to scan the union $\mathcal{L}_1(n) \cup \mathcal{L}_2(n)$ lexicographically.

5.3. Code generation

Let us now explore the details of the iteration code generation. We have the same constraints as in Sect. 3.3: avoid as much as possible unneeded computations. The solution is also the same as before: we will try to avoid computing several times the same expressions.

Let us remark that the labels of the branches of the QUASTs are sets of statements (the $\mathcal{I}_x(z)$). So, to be able to do a complete optimization, we have to consider these sets and to generate a separate piece of code for each of them. Let us label these sets $\mathcal{I}^1, \mathcal{I}^2, \ldots, \mathcal{I}^S$. The abstract coding scheme is described below.

```
        x, s = first
        if x = ⊥ then goto endlabel
        goto label_s
label_1 statements of set I^1
        x, s = next_{i∈I^1}(x)
        if x = ⊥ then goto endlabel
        goto label_s
         ...
label_S statements of set I^S
        x, s = next_{i∈I^1}(x)
        if x = ⊥ then goto endlabel
        goto label_s
endlabel
```

To generate such a code while avoiding computing minima of QUASTs, the base functions we use are the next_i^k. We can now extract from the QUASTs all constant expressions with respect to their nesting level. Reusing the same structure as in Sect. 3.3, we would like to branch directly to the deepest nesting level, thus avoiding the computation of outer level constants.

We must be careful when doing this because some statements appear in several sets $\mathcal{I}^s$ and so we have to devise a mean to compute the constants at the right time. One solution is to use variables indicating whether these computations have to be done.

Let us clarify this: let $\mathrm{init}_{i,k}$ = `true` mean that constants for statement i at dimension k have to be computed. These variables are all initially `true`. They are set to `false` each time they are computed and to `true` whenever there is a change in the iteration dimension. We summarize this below with the coding scheme corresponding to the abstract code of a given set $\mathcal{I}^s$.

```
label_{s,1} foreach i ∈ I^s do
              if init_{i,1} then
                 computation of level 1
                  constants for statement i
                 init_{i,1} = false
              endif
            enddo
label_{s,2} foreach i ∈ I^s do
                if init_{i,2} then
                computation of level 2
                 constants for statement i
                init_{i,2} = false
                endif
```

```
        enddo
         ...
label_{s,d} foreach i ∈ J^s do
              if init_{i,d} then
            computation of level d
             constants for statement i
            init_{i,d} = false
              endif
        enddo
        computation of the statements
         of set J^s
        next_{i,d} ⤚ goto label_{j,d}
        do j = 1, n
              init_{j,d} = true
        enddo
        next_{i,d-1} ⤚ goto label_{j,d-1}
        do j = 1, n
              init_{j,d-1} = true
        enddo
         ...
        next_{i,1} ⤚ goto label_{j,1}
        goto endlabel
```

Finally, using the same optimizations as in Sect. 3.3, we are now able to generate the iteration code for a union of LBLs. The generated code for our simple example is available in an extended version of this paper[2].

6. Conclusion

We have presented a general method for scanning linearly bounded lattices and unions of linearly bounded lattices. The power of this method comes from the fact that it accepts any number of parameters. Indeed, as all the underlying computations are done with PIP which is parameterized, one can for example parameterize the code generation by the number of the target processor and the size of the domain.

This method distinguishes itself from the other existing ones by producing low level code and not using Do-loops. We have however tried to keep the efficiency of the loop structure and the experimental results we have obtained show that we have achieved our goal. We should note that the efficiency of the produced code depends greatly on the optimizations done afterwards by the native compiler.

The codes we generate may look very complex. One however must keep three points in mind:

- Firstly, the generated code is to be compiled without human intervention. Hence, the only relevant figure of merit is execution time. From this point of view, our codes qualify.
- Next, we generate directly low level code. Low level code equivalent to loops such as in Sect. 4.2 would probably look as complicated as our own.
- Lastly, in many cases, the scanning code is inherently complex. The constraint that the scanning code must be simple and elegant should be taken care of when selecting code transformations. How to express this constraint and solve the associated optimization problem is unknown at present.

Further experimentation of the general case will be done with the inclusion of the prototype in a complete parallelizer.

References

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

[2] P. Boulet and P. Feautrier. Scanning polyhedra without do-loops. Technical report, Laboratoire PR*i*SM, Université de Versailles-St Quentin en Yvelines, France, 1998. available at `http://www.lifl.fr/~boulet/publi/polyscanRR.ps.gz`.

[3] Z. Chamski. Scanning polyhedra with do loop sequences. In *Workshop on Parallel Algorithms '92*, Sofia, 1992.

[4] J.-F. Collard. Code generation in automatic parallelizers. In C. Girault, editor, *Proc. Int. Conf. on Application in Parallel and Distributed Computing. IFIP WG 10.3*, pages 185–194. North Holland, Apr. 1994.

[5] J.-F. Collard, P. Feautrier, and T. Risset. Construction of DO loops from systems of affine constraints. *Parallel Processing Letters*, 5(3):421–436, Sept. 1995.

[6] A. Darte. *Techniques de parallélisation automatique de nids de boucles*. PhD thesis, Ecole Normale Supérieure de Lyon, 1993.

[7] P. Feautrier. Parametric integer programming. *RAIRO Recherche Opèrationnelle*, 22:243–268, Sept. 1988.

[8] P. Feautrier. Semantical analysis and mathematical programming. In M. C. et al., editor, *Parallel and distributed algorithms*, pages 309–320, North Holland, 1989. Elsevier Science Publishers B.V.

[9] F. Irigoin. *Partitionnement des boucles imbriquées, une technique d'optimisation pour les programmes scientifiques*. PhD thesis, Ecole Nationale Supérieure des Mines de Paris, June 1987.

[10] W. Kelly, W. Pugh, and E. Rosser. Code generation for multiple mappings. In *The 5th Symposium on Frontiers of Massively Parallel Computation*, pages 332–341, McLean, Virginia, Feb. 1995.

[11] G.-R. Perrin and C. Reffay. Communication code generation in systems of affine recurrence equations. *Integration: the VLSI Journal*, 20:63–83, 1995.

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

[13] J. Xue. Unimodular transformations of non-perfectly nested loops. *Parallel Computing*, 22:1621–1645, Feb. 1997.