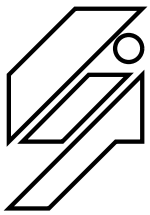
Code generation in bouclettes.

Pierre Boulet, Michèle Dion



Laboratoire de l'Informatique du Parallélisme

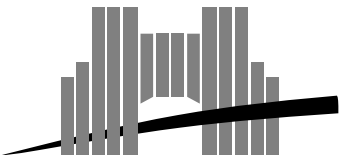
Ecole Normale Supérieure de Lyon
Unité de recherche associée au CNRS n°1398

Code generation in Bouclettes

Pierre BOULET
Michèle DION

November 1995

Research Report N^{O} 95-43



Ecole Normale Supérieure de Lyon
46 Allée d'Italie, 69364 Lyon Cedex 07, France
Téléphone : (+33) 72.72.80.00 Télécopieur : (+33) 72.72.80.80
Adresse électronique : lip@lip.ens−lyon.fr

Code generation in Bouclettes

Pierre BOULET
Michèle DION

November 1995

Abstract

Bouclettes is a source to source loop nest parallelizer. It takes as input Fortran uniform, perfectly nested loops and gives as output a HPF (High Performance Fortran) program with data distribution and parallel (`$HPF! INDEPENDENT`) loops. This paper explains how the HPF program is built from some scheduling and allocation functions automatically generated by Bouclettes.

Keywords: automatic parallelization, loop nest, HPF, compiler, code generation

Résumé

Bouclettes est un paralléliseur source à source de nids de boucles. Il prend en entrée des boucles Fortran uniformes et parfaitement imbriquées et retourne en sortie un programme HPF (High Performance Fortran) avec une distribution des données et des boucles parallèles (`$HPF! INDEPENDENT`). Ce papier explique comment le programme HPF est construit à partir des fonctions d'ordonnancement et d'allocation générées automatiquement par Bouclettes.

Mots-clés: parallélisation automatique, nid de boucles, HPF, compilation, génération de code

Code generation in Bouclettes

Pierre BOULET <Pierre.Boulet@lip.ens-lyon.fr>
Michèle DION <Michele.Dion@lip.ens-lyon.fr>

November 20, 1995

1 Introduction

1.1 What is Bouclettes?

Bouclettes is a source to source loop nest parallelizer. It takes Fortran 77 loops as input and returns an equivalent parallel program in HPF (High Performance Fortran).

Bouclettes has been written to validate some scheduling and mapping techniques based on the hyperplane method. These techniques are briefly sketched in section 2. The goal persued when building Bouclettes was to have a completely automatic parallelization tool. This goal has been reached and the input of the user is only required to choose the parallelization methodologies to be applied.

We have chosen HPF as the output language because we believe it can become a standard for parallel programming. Furthermore, data parallelism is a programmation paradigm that provides a simple way of describing data distributions and of managing the communications induced by the computations. It thus relieves the programmer (or the parallelization tool) of generating the low-level communications.

This paper is organized as follows: after the introduction, we recall in section 2 how the parallelism is extracted from the input program; and in the following two sections, we explain how the code is rewritten in HPF. We first describe the issues of rewriting a program scheduled with a linear schedule in section 3, and then propose a technique to rewrite a program scheduled with a shifted linear schedule in section 4. We then present a detailed example in section 5. Finally we conclude in section 6.

1.2 Related work

Automatic parallelization has been studied by many researchers and some tools for automatic parallelization have been written: SUIF [7], at Stanford University, California, PIPS [11, 6] at the École Nationale Supérieure des Mines de Paris, France, the Omega Library [10] at the University of Maryland, Maryland, LooPo [8] at the University of Passau, Germany, and PAF [12] at the University of Versailles, France, among others.

The particularities of Bouclettes in regards of these other tools are the methodologies employed and the output language. Indeed, Bouclettes takes as input perfectly nested loops with translations as array access functions and uses more complicated techniques to parallelize this kind of loops into HPF programs.

2 Data analysis and parallelism extraction

The parallelization process can be decomposed into several inter-dependent tasks. See figure 2.

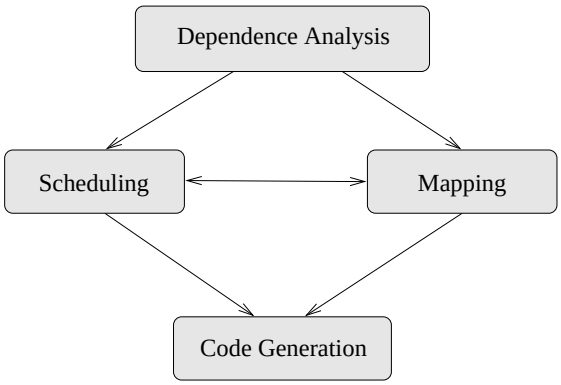


Figure 1: The parallelization stages

The dependence analysis consists in building a graph representing the constraints on the execution order of the instances of the statements. The scheduling uses the dependences to build a function that associates an execution time to an instance of a statement. The mapping stage maps the data arrays and the instances of the statements to a virtual mesh of processors. The two previous stages (the scheduling and the mapping) are inter-dependent: we want the global transformation of the original loop nest to respect data dependences. The last stage is the code generation. We generate here code with parallel loops (`INDEPENDENT` loops) and a data allocation (`DISTRIBUTE` and `ALIGN` directives).

The Bouclettes system is organized as a succession of stages:

1. the input program is analyzed and translated into an internal representation,
2. this representation is used to compute the data dependences, in our case, data dependences are uniform, so a simple custom dependence analyzer is enough to get the exact data dependences,
3. from these data dependences, a linear or shifted linear schedule is computed,
4. the schedule and the internal representation are used to compute a mapping compatible with the schedule,

5. finally, the HPF code is generated following the previously computed transformation.

We detail below the analysis and parallelism extraction phases. A complete example illustrating these phases (and the rewriting one) is given in section 5.

2.1 Dependence analysis

The dependence analysis consists in finding the constraints on the execution order of the iterations of the statements. The dependence analysis is quite simple in the restricted context we have here. It basically consists in finding all the data dependences between the inner statements. The three kinds of dependences (direct, anti and output dependences) can be computed in the same way: the dependence vectors are differences between two data access functions that address the same array and reciprocally, all the differences between two data access functions that address the same array are dependence vectors.

2.2 Scheduling

Darte and Robert have presented techniques to compute schedules for a given uniform loop nest [2, 4]. These techniques are part of the theoretical basis of Bouclettes.

Currently, the user has the choice between the linear schedule and the shifted linear schedule. A schedule is a function that associates to each computation point (each iteration of a statement) the time when it is computed in the parallel program.

the linear schedule is a linear function that associates a time t to an iteration point $\vec{i}$ ($\vec{i} = (i, j, k)$ if the loop nest is three dimensional) as follows:

$$t = \left\lfloor \frac{p}{q} \pi . \vec{i} \right\rfloor$$

where p, q are mutually prime integers and π is a vector of integers of dimension d (the depth of the loop nest) whose components are relatively prime.

the shifted linear schedule is an extension of the linear schedule where each statement of the loop nest body has its own scheduling function. All these functions share the same linear part and some (possibly different) shifting constant are added for each statement. The time t for statement k is computed as follows:

$$t = \left\lfloor \frac{p}{q} \pi . \vec{i} + \frac{c_k}{q} \right\rfloor$$

where p, q, c_k are integers and π is a vector of integers of dimension d (the depth of the loop nest) whose components are relatively prime.

The computation of these schedules is done by techniques which guarantee that the result is optimal in the considered class of schedules. Here "optimal" means that the total latency is minimized.

2.3 Mapping

Darte and Robert have presented a technique to build a mapping of data and computation on a virtual processors grid [3]. It is this technique that is used in Bouclettes.

A mapping function is a function that associates to each computation point the processor which will do the computation and to each array element the processor whose memory it will be stored in.

Based on the computation of the so called "communication graph", a structure that represents all the communications that can occur in the given loop nest, a projection M and some shifting constants are computed. The base idea is to project the arrays (and the computations) on a virtual processor grid of dimension $d-1$. Then, the arrays and the computations are aligned (by the shifting constants) to suppress some computations.

More precisely, M, the projection matrix, is a $(d-1) \times d$ full ranked matrix of integers and the constants α_x are vectors of integers and of dimension $d-1$. To each array or statement x is then associated an allocation function defined by:

$$\text{alloc}_x(\vec{i}) = M\vec{i} + \alpha_x$$

As the considered loop nests are uniform, choosing a different matrix for different arrays or statements would not improve the mapping. The schedule has to be taken into account to choose the matrix M. Effectively, the transformed loop nest will have as iteration domain, the image of the initial iteration domain by the transformation:

$$\vec{i} \mapsto \begin{bmatrix} \pi \\ M \end{bmatrix} \vec{i}$$

It is mandatory to have this iteration domain mapped onto $\mathbf{N}^d$, as otherwise we would need rationally indexed processors. As the choice of M does not have a high impact on the number of communications that remain, $\left[\begin{smallmatrix} \pi \\ M \end{smallmatrix}\right]$ is just computed as the unimodular completion of vector π.

Once M has been computed, the alignment constants are determined in order to minimize the number of computations. Here the user can choose if he wants to respect the owner computes rule (as in HPF) or not. If he chooses not to respect this rule, some temporary arrays may be generated in the next stage to take it into account.

3 Coding the linear schedule

3.1 Loop rewriting

Bouclettes uses techniques presented by Collard, Feautrier and Risset in [1] to rewrite the loops after reindexation. Reindexation yields a new iteration space, which is a convex integer polyhedron defined by a set of affine constraints. Rewriting the nested loop needs to scan all the integer points of this convex and the algorithm relies on a parameterized version of the Dual Simplex, PIP (see [5]).

Consider the initial perfect loop nest (see program 1), where $\vec{z}$ is a vector of structure parameters (see [1]):

Program 1 Initial perfect loop nest

```
do i1=i1^l(z), i1^u(z)
   do i2=i2^l(i1, z), i2^u(i1, z)
          ...
         do id=id^l(i1, ..., id-1, z), id^u(i1, ..., id-1, z)
            S1(i1, i2, ..., id, z)
             ...
            Sk(i1, i2, ..., id, z)
         enddo
    ...
   enddo
enddo
```

We rewrite the loop nest after an unimodular linear transformation U. The vector coordinates $\vec{i} = (i_1, \ldots, i_d)^T$ and $\vec{j} = (j_1, \ldots, j_d)^T$, respectively in the old and new basis, are related by:

$$\vec{j} = U\vec{i}$$

where U is a $d \times d$ unimodular matrix ($\det(U) = \pm 1$). Since we are dealing with perfect loop nests, the iteration spaces are finite convex polyhedra of $\mathbf{Z}^d$ that can be defined by a set of inequalities such as:

$$D(\vec{z}) = \{\vec{i} | \vec{i} \in \mathbf{Z}^d, C\vec{i} + C'\vec{z} + \vec{b} \geq 0\}$$

where C, C' are constraints matrix and $\vec{b}$ is a constant vector. In the new iteration space, the polyhedra can be defined as :

$$D(\vec{z}) = \{\vec{j} | \vec{j} \in \mathbf{Z}^d, CU^{-1}\vec{j} + C'\vec{z} + \vec{b} \geq 0\}.$$

Collard, Feautrier and Risset have proved that the initial loop nest (program 1) can be rewritten in the new iteration space in the form given in program 2 where the loop bounds j_n^l and j_n^u, $1 \leq n \leq d$ are simple enough (for an HPF compiler) expressions of the structure parameters and of the surrounding indices $j_1, \ldots, j_{n-1}$. For a perfect loop nest of depth d, the new loop bounds are obtained after $2d$ successive calls to PIP.

3.2 When the OCR is not respected

HPF compilers respect the *owner computes rule* (OCR): each processor computes its own data only. In the mapping process of Bouclettes, the user can chose not to respect the OCR. Let us consider an array element $a(\vec{i} + c_a)$ computed in a statement $S(\vec{i})$. The mapping process can return allocation functions such that: $(M(\vec{i} + c_a) + \alpha_a) \neq M\vec{i} + \alpha_S$. To make both the allocation functions and the OCR compatible, we need to add "temporary" arrays during the code generation stage.

Program 2 Rewritten perfect loop nest

```
do j1=j1^l(z⃗), j1^u(z⃗)
   do j2=j2^l(j1, z⃗), j2^u(j1, z⃗)
        ...
        do jd=jd^l(j1, ..., jd−1, z⃗), jd^u(j1, ..., jd−1, z⃗)
           S1(U^{-1} j⃗, z⃗)
            ...
           Sk(U^{-1} j⃗, z⃗)
        enddo
    ...
   enddo
enddo
```

Program 3 Initial loop nest which does not verify the OCR

```
real a(n,n)

do i⃗
      S(i⃗)    a(i⃗ + c_a) = expr
enddo
```

Program 4 Loop nest which verifies the OCR after the addition of temporary arrays

```
real a(n, n)
real a_tmp(n, n)

do i⃗
      S1(i⃗)   a_tmp(i⃗ + c_a) = expr
      S2(i⃗)   a(i⃗ + c_a) = a_tmp(i⃗ + c_a)
enddo
```

Hence, the loop nest of program 3, where *expr* is an expression of other array elements of the program is transformed by adding temporary arrays into the loop nest of program 4.

The new allocation functions are deduced from the initial functions to respect the OCR:

$$\begin{aligned} \text{alloc}_a(\vec{i}) &= M\vec{i} + \alpha_a \\ \text{alloc}_{S_1}(\vec{i}) &= M\vec{i} + \alpha_s \\ \text{alloc}_{a_{tmp}}(\vec{i}) &= M\vec{i} + \alpha_S - Mc_a \\ \text{alloc}_{S_2}(\vec{i}) &= M\vec{i} + Mc_a + \alpha_a. \end{aligned}$$

3.3 Array alignment

In HPF, the programmer can specify the data mapping at two levels. First the arrays are aligned with respect to one another with the directive `ALIGN`. Then, the aligned data are distributed to the processors with the directive `DISTRIBUTE` [9].

The general alignment statement is:

```
!HPF$ ALIGN array WITH target
```

This specifies to the compiler that the `array` should be aligned with the `target`. The `target` can be either another array of the program or a `TEMPLATE` (a virtual array). An example of alignement statement is:

```
!HPF$ ALIGN A(i,j) WITH B(j+1,i+1)
```

The general distribution statement is:

```
!HPF$ PROCESSORS proc
!HPF$ DISTRIBUTE arrays [ONTO proc]
```

The programmer can also specify the way to distribute the arrays on the processors (`BLOCK`, `CYCLIC` or `BLOCK_CYCLIC`). Bouclettes can generate any type of distribution. The best one would certainly be a `BLOCK_CYCLIC` one where the size of the block would depend on the target machine. As in current HPF compilers, only `BLOCK` distributions are implemented, we have chosen to make Bouclettes generate `BLOCK` distributions.

According to the projection matrix M, we adopt two different strategies in the code generation to align the data:

- if the projection is along one axis of the iteration, we are able to align directly the arrays as we explain in the following,
- otherwise, we need to "redistribute" the arrays before aligning them (see Section 3.4).

When the projection is along one direction of the iteration space D
Let a_k, $(1 \leq k \leq n)$ be the arrays of a loop nest of depth d. Let $\text{alloc}_{a_k}(\vec{i}) = M\vec{i} + \alpha_{a_k}$ be the allocation function for array a_k. Let π be the linear scheduling vector for the loop nest. Let $\vec{i} \in D$ and $\vec{j} = \vec{i} + \begin{pmatrix} \pi \\ M \end{pmatrix}^{-1} \begin{pmatrix} 0 \\ \alpha_k \end{pmatrix}$. We have

$$M\vec{j} = M\vec{i} + M \begin{pmatrix} \pi \\ M \end{pmatrix}^{-1} \begin{pmatrix} 0 \\ \alpha_k \end{pmatrix}.$$

Let $\begin{pmatrix} \pi \\ M \end{pmatrix}^{-1} = (X_1\ X_2)$, we have

$$\begin{pmatrix} \pi \\ M \end{pmatrix} (X_1\ X_2) = \begin{pmatrix} \pi X_1 & \pi X_2 \\ MX_1 & MX_2 \end{pmatrix} = Id.$$

Hence, $MX_2 = Id$ and

$$M\vec{j} = M\vec{i} + M(X_1X_2) \begin{pmatrix} 0 \\ \alpha_k \end{pmatrix}$$
$$M\vec{j} = M\vec{i} + \alpha_k$$

Let $p = M\vec{i} + \alpha_k$ (the processor p receives the value $a_k(\vec{i})$). p and the image of $\vec{j}$ by M correspond to the same point in the virtual processor space. To align all arrays a_k with respect to one another, one possibility is to declare a template `BCLT_template` of dimension d and to align each array with the following directive:

$$\texttt{!HPF\$ ALIGN } a_k(\vec{i}) \texttt{ WITH BCLT\_template} \left(\vec{i} + \begin{pmatrix} \pi \\ M \end{pmatrix}^{-1} \begin{pmatrix} 0 \\ \alpha_k \end{pmatrix} \right)$$

The distribution of the aligned data onto the processors is then specified with the directive:

```
!HPF$ DISTRIBUTE BCLT_template (BLOCK,...,BLOCK,*,BLOCK,...,BLOCK)
```

The "*" corresponds to the direction of the projection. Let us notice that this is possible only because the projection is parallel to one direction of the iteration space.

3.4 Array rotation

When this projection is not along one axis, we need to "redistribute" the arrays to write the HPF directives.

Let $U = \begin{pmatrix} \pi \\ M \end{pmatrix}$. For each array a_k of the loop nest, we define the new array $a_{k,\text{rot}}$ such that

$$a_{k,\text{rot}}(\vec{i}) = a_k(U^{-1}(\vec{i})).$$

We compute the new allocation function for array $a_{k,\text{rot}}$ from the allocation function for array a_k. Let $\text{alloc}_{a_k}(\vec{i}) = M\vec{i} + \alpha_k$, we choose $\text{alloc}_{a_{k,\text{rot}}}(\vec{i}) = MU^{-1}\vec{i} + \alpha_k$ ($a_{k,\text{rot}}(\vec{i})$ and $a_k(U^{-1}(\vec{i}))$ are in the memory of the same processor).

As in Section 3.3, let $U^{-1} = (X_1\ X_2)$. Hence, we have

$$MU^{-1} = (MX_1\ MX_2)$$

and

$$UU^{-1} = \begin{pmatrix} \pi X_1 & \pi X_2 \\ MX_1 & MX_2 \end{pmatrix} = \mathrm{Id}.$$

Hence,

$$MU^{-1} = \left(\begin{array}{c|c} 0 & \\ \vdots & \mathrm{Id} \\ 0 & \end{array}\right).$$

The projection matrix for the new "rotated" arrays defines a projection parallel to the first dimension of the processor space.

Besides, after rewriting, the access to array a_k in the new loop nest is $a_k(U^{-1}\vec{j} + c_{a_k})$. We have

$$\begin{aligned} a_k(U^{-1}\vec{j} + c_{a_k}) &= a_{k,\mathrm{rot}}(U(U^{-1}\vec{j} + c_{a_k})) \\ &= a_{k,\mathrm{rot}}(\vec{j} + Uc_{a_k}). \end{aligned}$$

If we replace in the new loop nest, all the occurrences of array a_k by the corresponding occurrences of array $a_{k,\mathrm{rot}}$, we obtain again a rewritten loop nest with uniform access to arrays.

3.5 Summary

To summarize our approach, the strategy to generate the code after a linear scheduling is:

1. verify if the data and computation mapping is compatible with the OCR, if not insert temporary arrays,
2. rewrite the loop nest,
3. verify if the projection matrix corresponds to a projection along one dimension of the iteration space, if not replace the initial arrays by rotated arrays, generate the parallel loops at the beginning and at the end of the program to respectively initialize with the correct values the rotated values and to copy the values of the rotated arrays in the initial arrays,
4. generate the alignment directives to align each array of the loop nest with respect to a template,
5. generate the distribute directive to map the template to virtual processors.

Program 5 Aloop nest after the linear transformation

```
     do t=t_l , t_u
$HPF! INDEPENDENT
        do pr^1=pr_l^1(t), pr_u^1(t)
           ...
$HPF! INDEPENDENT
           do pr^d=pr_l^d(t), pr_u^d(t)
              S_1(t, pr^1, ... , pr^d)
              ...
              S_k(t, pr^1, ... , pr^d)
           enddo
         ...
        enddo
     enddo
```

4 Coding shifted linear schedules

4.1 From linear scheduling to shifted linear scheduling

As explained before, we have rewritten the initial loop nest taking into account only the linear part of the schedule. This transformed loop nest looks like program 5.

Let us consider the following schedule:

$$\text{schedule}(I) = \left\lfloor \frac{1}{q}(pLI + c) \right\rfloor \tag{1}$$

for a given statement.

The previous transformations uses $t = LI$. So the execution time following the shifted linear schedule (equation 1) can be rewritten as

$$\text{exe}(t) = \left\lfloor \frac{1}{q}(pt + c) \right\rfloor . \tag{2}$$

In the above transformed code, the processors on which the computations are executed depend only on the execution time t (and on the parameters of the program). So, if we can "inverse" the exe function, we will be able to rewrite the loop nest with a new time variable corresponding to the time given by the schedule.

We now show that this inverse is:

$$\text{lin}(T) = \left\lceil \frac{1}{p}(qT - c) \right\rceil . \tag{3}$$

We have to prove the following proposition.

Proposition 1

$$\forall t \in \mathbf{Z}, (t \in [\text{lin}(T), \text{lin}(T+1)[\Leftrightarrow \text{exe}(t) = T)$$

Proof Let $t \in \mathbf{Z}$ such that:

$$\mathrm{lin}(T) \leq t < \mathrm{lin}(T+1)$$

We can successively deduce:

$$\left\lceil \frac{1}{p}(qT - c) \right\rceil \leq t < \left\lceil \frac{1}{p}(q(T+1) - c) \right\rceil \tag{4}$$

and

$$\frac{1}{q}\left(p\left\lceil \frac{1}{p}(qT-c)\right\rceil + c\right) \leq \frac{1}{q}(pt+c) < \frac{1}{q}\left(p\left\lceil \frac{1}{p}(q(T+1)-c)\right\rceil + c\right).$$

Let $f(T) = \frac{1}{q}\left(p\left\lceil \frac{1}{p}(qT-c)\right\rceil + c\right)$, we then have:

$$f(T) \leq \frac{1}{q}(pt+c) < f(T+1).$$

There always exist α and β such that

$$qT - c = p\alpha + \beta, 0 \leq \beta < p, \alpha \in \mathbf{Z} \tag{5}$$

Let us compute $f(T)$:

$$\begin{aligned} f(T) &= \frac{1}{q}\left(p\left\lceil \frac{1}{p}(qT-c)\right\rceil + c\right) \\ &= \frac{1}{q}\left(p\left\lceil \alpha + \frac{\beta}{p}\right\rceil + c\right) \\ &= \frac{1}{q}\left((p\alpha + c) + p\left\lceil \frac{\beta}{p}\right\rceil\right) \\ &= \frac{1}{q}\left((qT - \beta) + p\left\lceil \frac{\beta}{p}\right\rceil\right) \\ &= T + \frac{1}{q}\left(p\left\lceil \frac{\beta}{p}\right\rceil - \beta\right). \end{aligned}$$

If we proved that $\lfloor f(T) \rfloor = T$, we would obtain

$$T \leq \left\lfloor \frac{1}{q}(pt+c)\right\rfloor < T+1$$

which is equivalent to $\mathrm{exe}(t) = T$, thereby establishing the proof.

Let us discuss the value of

$$\lfloor f(T) \rfloor = \left\lfloor T + \frac{1}{q}\left(p\left\lceil \frac{\beta}{p}\right\rceil - \beta\right)\right\rfloor$$

in function of β.

- If $\beta = 0$ then $f(T) = T$ and $\lfloor f(T) \rfloor = T$.

- Otherwise $0 < \beta < p$ and so $\left\lceil \frac{\beta}{p} \right\rceil = 1$. We would like to have $\frac{1}{q}(p-\beta) < 1$. This is equivalent to prove

$$1 < \frac{q+\beta}{p}. \tag{6}$$

  - If $p < q$ then equation 6 is verified.
  - Else, as $p \wedge q = 1$, $p > q$. Remember the hypothesis (equation 4):

$$\left\lceil \frac{1}{p}(qT - c) \right\rceil \leq t < \left\lceil \frac{1}{p}(q(T+1) - c) \right\rceil.$$

    This implies that

$$\left\lceil \frac{1}{p}(qT - c) \right\rceil < \left\lceil \frac{1}{p}(q(T+1) - c) \right\rceil.$$

    Introducing α and β (equation 5) in this equation leads to

$$\left\lceil \alpha + \frac{\beta}{p} \right\rceil < \left\lceil \alpha + \frac{\beta+q}{p} \right\rceil.$$

    As α is a integer, we have

$$\left\lceil \frac{\beta}{p} \right\rceil < \left\lceil \frac{\beta+q}{p} \right\rceil.$$

    As $\left\lceil \frac{\beta}{p} \right\rceil = 1$, we have

$$1 < \left\lceil \frac{\beta+q}{p} \right\rceil$$

    which implies equation 6.

We have proven that in all cases, $\lfloor f(T) \rfloor = T$, which implies $\mathrm{exe}(t) = T$ and

$$\forall t \in \mathbf{Z}, (t \in [\mathrm{lin}(T), \mathrm{lin}(T+1)[\ \Rightarrow \mathrm{exe}(t) = T).$$

We know have to prove the opposite.
It is immediate to see that

$$\bigcup_{T \in \mathbf{Z}} [\mathrm{lin}(T), \mathrm{lin}(T+1)[= \mathbf{Z}.$$

So, let $t \in \mathbf{Z}$ and $T = \mathrm{exe}(t)$. The previous equation implies that

$$\exists T', \mathrm{lin}(T') \leq t < \mathrm{lin}(T'+1).$$

We have just proven that this implies that $\mathrm{exe}(t) = T'$. We can now conclude that $T' = T$ and that

$$\forall t \in \mathbf{Z}, (T = \mathrm{exe}(t) \ \Rightarrow t \in [\mathrm{lin}(T), \mathrm{lin}(T+1)[).$$

□

4.2 The formal transformation

4.2.1 The time boundaries

After the linear transformation, the time t varies from t_l to t_u. We have to find the boundaries of the final time T which corresponds to the interval $[t_l, t_u]$. We take as execution time:

$$\text{exe}_i(t_l) \leq T \leq \text{exe}_i(t_u) \tag{7}$$

where $\text{exe}_i(t) = \left\lfloor \frac{1}{q}(pt + c_i) \right\rfloor$.

Using proposition 1, for statement S_i, considering $\text{lin}_i(t) = \left\lceil \frac{1}{p}(qt - c_i) \right\rceil$, we have the code of program 6:

Program 6 Code for statement S_i with shifted linear schedule

```
     do T=exe_i(t_l), exe_i(t_u)
$HPF! INDEPENDENT
        do t=lin_i(T), lin_i(T+1) - 1
$HPF! INDEPENDENT
           do pr^1=pr_l^1(t), pr_u^1(t)
            ...
$HPF! INDEPENDENT
              do pr^d=pr_l^d(t), pr_u^d(t)
                 S_i(t,pr^1,... ,pr^d)
              enddo
            ...
           enddo
        enddo
     enddo
```

Let us verify that this transformation is the one we were looking for: does the `T` loop index defined above corresponds to the schedule? Consider index point I. Statement $S_i(I)$ should be executed at time $T = \text{exe}_i(LI)$. Proposition 1 proves that LI is in the definition interval of t. So statement $S_i(I)$ is indeed executed at the required time on the processors computed in the mapping phase.

4.2.2 When there are several instructions

It would be interesting here to have a construct to express control parallelism in HPF. For the moment —the current specification is HPF 1 and can be found in [9]— such a construct does not exist, but at the time of writing, it is discussed into the HPF Forum to consider its inclusion in the HPF 2 specification.

Indeed, we would like to execute in parallel all the statements that have the same schedule. As it is not currently possible, we just execute them sequentially —it is likely that they would be sequentialized by the compiler anyway.

The last problem we have to solve is how to deal with different shifting constants for different statements. It is not very difficult:

- first, let the sequential time T go from the lower bound of the time given by the smallest constant to the upper bound of the time given by the

largest constant: T varies from T_l to T_u defined as

$$T_l = \left\lfloor \frac{1}{q} \left(pt_l + \min_i c_i \right) \right\rfloor$$
$$T_u = \left\lfloor \frac{1}{q} \left(pt_u + \max_i c_i \right) \right\rfloor$$

- and then verify for each instruction that the corresponding linear time t does not exceed its definition interval $[t_l, t_u]$. This can be done by letting t vary from $\max(t_l, \mathrm{lin}_i(T))$ to $\min(t_u, \mathrm{lin}_i(T+1) - 1)$.

  Note: this style of program writing is correct because in Fortran, if a `do` loop has its lower bound greater than its upper bound, its body is not executed.

4.2.3 Some last optimizations

To avoid as much as possible the computation of min and max functions, we can distinguish three stages in the execution of the parallelized program:

1. The initial stage when at each time some statements may not execute: we compute the lower bound as $\max(t_l, \mathrm{lin}_i(T))$ but let the upper bound without the min: $\mathrm{lin}_i(T+1) - 1$. This is for values of T varying form

   $$\left\lfloor \frac{1}{q} (pt_l + \min_i c_i) \right\rfloor .$$

   to

   $$\left\lceil \frac{1}{q} (pt_l + \max_i c_i) \right\rceil - 1$$

2. The steady-state phase when all statements always execute: there are no min and no max. This is for values of T varying form

   $$\left\lceil \frac{1}{q} (pt_l + \max_i c_i) \right\rceil$$

   to

   $$\left\lfloor \frac{1}{q} (pt_u + \min_i c_i) \right\rfloor - 1.$$

3. The final stage when some statements may have finished to execute: we let the lower bound without the max: $\mathrm{lin}_i(T)$ and compute the upper bound as $\min(\mathrm{lin}_i(T+1) - 1)$. This is for values of T varying form

   $$\left\lfloor \frac{1}{q} (pt_u + \min_i c_i) \right\rfloor$$

   to

   $$\left\lfloor \frac{1}{q} (pt_u + \max_i c_i) \right\rfloor .$$

We suppose here that n the size parameter of the program is large enough.

We have also worked on the symbolic simplification of loop bounds. The module that work on symbolic expressions puts affine expressions in a "normalized form" (each symbolic constant appears only once) then the floor and ceil functions are deleted if the expression on which they apply is a integer expression and the number of division is reduced by factorizing the common denominators of fractions. This leads to more readable code and less expensive computations to determine loop bounds.

A third optimization addresses the case of non executed loops because their lower bounds is greater than their upper bound. If it is always the case for a loop, this loop is discarded. And if a loop has the same lower and upper bounds, the do statement is replaced by an affectation of the common bounds value to the loop index. These simplifications give a more readable form for a human being, which is also easier to analyze for a compiler program.

5 A detailed example

We will study here the whole rewriting scheme on an example.

5.1 The input program

The example that we consider (see program 7) is a two dimensional loop nest with two inner statements. This is not a real world code but it has been designed to show the difficulty of rewriting (and even to find the parallelism).

Program 7 Example: input program

```
      parameter (n=100)

      integer i,j
      real a(n,n)
      real b(n,n)

      do i= 2, n-5
         do j= 6, n-3
c   Statement 1
            a(i,j)=a(i,j-5)+b(i-1,j+3)-a(i+5,j-4)
c   Statement 2
            b(i+1,j-2)=a(i,j-1)
         enddo
      enddo
```

There are four data dependences which are:

From statement	To statement	Dependence vector
2	1	$[2,-5]$
1	2	$[0,1]$
1	1	$[5,-4]$
1	1	$[0,5]$

The first line means that the data item produced by the instance $[i, j]$ of statement 2 is used by the instance $[i + 2, j - 5]$ of statement 1.

5.2 Linear scheduling without redistribution

The optimal linear scheduling vector is $[3, 1]$. The projection matrix is $[1, 0]$ and the alignment constants are:

array	shift	statement	shift
a	0	1	0
b	1	2	0

We can easily check that the owner computes rule is respected here. Actually forcing to respect this rule here gives another equivalent mapping. As the owner computes rule is respected, we will not see any temporary arrays in the produced code to enforce this rule. As the projection matrix is $[1, 0]$, the projection of the two-dimensional arrays is done on the first dimension following the second direction. Hence the redistribution is not necessary.

The resulting code is program 8.[1]

Program 8 Example: HPF program with linear schedule and no redistribution

```
      PROGRAM boucle

      INTEGER P1
      INTEGER T
      PARAMETER (n = 100)
      REAL a(n,n)
      REAL b(n,n)

!HPF$ TEMPLATE BCLT_0_template(n+1,n+3)
!HPF$ DISTRIBUTE BCLT_0_template(BLOCK,*)
!HPF$ ALIGN a(i1,i2) WITH BCLT_0_template(i1,i2+3)
!HPF$ ALIGN b(i1,i2) WITH BCLT_0_template(i1+1,i2)
      DO T = 12, 4*n-18
!HPF$ INDEPENDENT
         DO P1 = ceiling(max((-n+T)/3.0+1,2)), floor(min((T)/3.0-2,n-5))
            a(P1,T-3*P1) = (a(P1,T-3*P1-5)+(b(P1-1,T-3*P1+3)-a(P1+5,T-3*P1-4)))
            b(P1+1,T-3*P1-2) = a(P1,T-3*P1-1)
         END DO
      END DO
      END
```

Following the declarations, there are the distribution and alignment directives. They are generated from the mapping projection and the shifting constants. The `DISTRIBUTE` directive uses a `BLOCK` strategy to map the template on the processors. As mentioned before, any strategy could be used here and a block-cyclic approach with a block size depending on the target machine would probably be a better solution.

[1] The variables have been renamed for improved readability

The loop nest consists in a sequential loop (index `T`) surrounding a parallel loop (index `P1`). `T` and `P1` are obtained from the initial loop indices as:

$$\begin{pmatrix} \texttt{T} \\ \texttt{P1} \end{pmatrix} = \begin{pmatrix} 3 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} \texttt{i} \\ \texttt{j} \end{pmatrix}.$$

5.3 Linear scheduling with redistribution

The user can choose to enforce a redistribution. In this case, the data arrays are copied into temporary arrays for which the resulting loop nest may be simpler to analyze by an HPF compiler. This is because when doing this redistribution, the complicated array access functions are moved out of the main loop nest to the surrounding `FORALL` loops that realize the redistribution. Thus, the array access functions become translations that are better optimized by HPF compilers. The resulting HPF program is program 9.

The `FORALL` statements express the redistribution before and after the computation of the transformed loop nest using the temporary arrays. Note that the array access functions are translations in this case and are more complicated without the redistribution.

5.4 Shifted linear scheduling case

We study here shifted linear scheduling without redistribution. It is possible to redistribute the arrays as in the previous section but it would not show anything new and would only complicate the code here.

The shifted linear scheduling functions[2] are:

$$\begin{cases} \text{schedule}_1(I) = \left\lfloor \frac{1}{5}([7,1].I) \right\rfloor \\ \text{schedule}_2(I) = \left\lfloor \frac{1}{5}([7,1].I + 4) \right\rfloor \end{cases}$$

The mapping of the arrays (and the computations) is the same as the one obtained for linear scheduling and the resulting code is decomposed in three stages:

1. The initial stage (see program 10) is limited to one unit of time (`T = 4`) and we can see the `max` function in the lower bound of the loops over the virtual time `VT`. This function ensures that the computations start at the right time considering the time shifting constants. Each statement is inside a loop nest of depth two: the `VT` index iterates over the instances of the statement that are scheduled at the same time (here `T = 4`), and the `P1` index iterates over the (virtual) processors.

2. The steady-state stage (see program 11) is the main stage when there is no time boundary problem and every thing is regular. Once again we have the two parallel loop nests inside the sequential loop over the time.

3. The final stage (see program 12) matches the initial stage to deal with the end of the computations with respect to the shifting constants.

[2] $\text{schedule}_i(I)$ is the scheduling function of statement i

Program 9 Example: HPF program with linear schedule and redistribution

```
      PROGRAM boucle

      INTEGER I1
      INTEGER I0
      INTEGER P1
      INTEGER T
      PARAMETER (n = 100)
      REAL a(n,n)
      REAL b(n,n)
      REAL ROTa(4*n-3,n)
      REAL ROTb(4*n-3,n)

!HPF$ TEMPLATE BCLT_0_template(4*n,n+10)
!HPF$ DISTRIBUTE BCLT_0_template(*,BLOCK)
!HPF$ ALIGN ROTa(i1,i2) WITH BCLT_0_template(i1+3,i2)
!HPF$ ALIGN ROTb(i1,i2) WITH BCLT_0_template(i1,i2+10)
      FORALL (I0 = 1:4*n-3,I1 = 1:n)
         ROTa(I0,I1) = 0
      END FORALL
      FORALL (I0 = 1:4*n-3,I1 = 1:n)
         ROTb(I0,I1) = 0
      END FORALL
      FORALL (I0 = 1:n,I1 = 1:n)
         ROTa(3*I0+I1-3,I0) = a(I0,I1)
      END FORALL
      FORALL (I0 = 1:n,I1 = 1:n)
         ROTb(3*I0+I1-3,I0) = b(I0,I1)
      END FORALL
      DO T = 12, 4*n-18
!HPF$ INDEPENDENT
         DO P1 = ceiling(max((-n+T)/3.0+1,2)), floor(min((T)/3.0-2,n-5))
            ROTa(T-3,P1) = (ROTa(T-8,P1)+(ROTb(T-3,P1-1)-ROTa(T+8,P1+5)))
            ROTb(T-2,P1+1) = ROTa(T-4,P1)
         END DO
      END DO
      FORALL (I0 = 1:n,I1 = 1:n)
         a(I0,I1) = ROTa(3*I0+I1-3,I0)
      END FORALL
      FORALL (I0 = 1:n,I1 = 1:n)
         b(I0,I1) = ROTb(3*I0+I1-3,I0)
      END FORALL
      END
```

Program 10 Example: shifted linear schedule, the initialization stage

```
      PROGRAM boucle

      INTEGER T
      INTEGER VT
      INTEGER P1
      PARAMETER (n = 100)
      REAL a(n,n)
      REAL b(n,n)

!HPF$ TEMPLATE BCLT_0_template(n+1,n+7)
!HPF$ DISTRIBUTE BCLT_0_template(BLOCK,*)
!HPF$ ALIGN a(i1,i2) WITH BCLT_0_template(i1,i2+7)
!HPF$ ALIGN b(i1,i2) WITH BCLT_0_template(i1+1,i2)
      T = 4
!HPF$ INDEPENDENT
      DO VT = max(20,5*T), 5*T+4
!HPF$ INDEPENDENT
         DO P1 = ceiling(max((-n+VT+3)/7.0,2)), floor(min((VT-6)/7.0,n-5))
            a(P1,VT-7*P1) = (a(P1,VT-7*P1-5)+
     & (b(P1-1,VT-7*P1+3)-a(P1+5,VT-7*P1-4)))
         END DO
      END DO
!HPF$ INDEPENDENT
      DO VT = max(20,5*T-4), 5*T
!HPF$ INDEPENDENT
         DO P1 = ceiling(max((-n+VT+3)/7.0,2)), floor(min((VT-6)/7.0,n-5))
            b(P1+1,VT-7*P1-2) = a(P1,VT-7*P1-1)
         END DO
      END DO
```

Program 11 Example: shifted linear schedule, the steady-state stage

```
      DO T = 5, (floor((8*n-38)/5.0)-1)
!HPF$ INDEPENDENT
         DO VT = 5*T, 5*T+4
!HPF$ INDEPENDENT
            DO P1 = ceiling(max((-n+VT+3)/7.0,2)), floor(min((VT-6)/7.0,n-5))
               a(P1,VT-7*P1) = (a(P1,VT-7*P1-5)+
     & (b(P1-1,VT-7*P1+3)-a(P1+5,VT-7*P1-4)))
            END DO
         END DO
!HPF$ INDEPENDENT
         DO VT = 5*T-4, 5*T
!HPF$ INDEPENDENT
            DO P1 = ceiling(max((-n+VT+3)/7.0,2)), floor(min((VT-6)/7.0,n-5))
               b(P1+1,VT-7*P1-2) = a(P1,VT-7*P1-1)
            END DO
         END DO
      END DO
```

Program 12 Example: shifted linear schedule, the final stage

```
      DO T = floor((8*n-38)/5.0), floor((8*n-34)/5.0)
!HPF$ INDEPENDENT
         DO VT = 5*T, min(8*n-38,5*T+4)
!HPF$ INDEPENDENT
            DO P1 = ceiling(max((-n+VT+3)/7.0,2)), floor(min((VT-6)/7.0,n-5))
               a(P1,VT-7*P1) = (a(P1,VT-7*P1-5)+
     & (b(P1-1,VT-7*P1+3)-a(P1+5,VT-7*P1-4)))
            END DO
         END DO
!HPF$ INDEPENDENT
         DO VT = 5*T-4, min(8*n-38,5*T)
!HPF$ INDEPENDENT
            DO P1 = ceiling(max((-n+VT+3)/7.0,2)), floor(min((VT-6)/7.0,n-5))
               b(P1+1,VT-7*P1-2) = a(P1,VT-7*P1-1)
            END DO
         END DO
      END DO
      END
```

6 Conclusion

We have presented in this paper the problems to solve for code generation in the Bouclettes tool and the solutions that have been implemented. We have chosen HPF as output language and this choice has proven critical for the code generation. Indeed, the use of HPF has relieved us from generating all the low level communications in the output parallel program. On the other hand, some complications arise from some current limitations of HPF:

- the fact that HPF respects the owner computes rule has forced us to generate some temporary arrays when the mapping is not compatible with this rule. It should be pointed out that the user can select an option in Bouclettes that force the mapping to respect the owner computes rule.
- the data distributions allowed in HPF are not always powerful enough to express the mapping. We have then developed a redistribution scheme to deal with all our mappings.
- the pure data parallelism of HPF does not allow control parallelism that would be necessary to express all the parallelism exposed by some shifted linear schedules.

References

[1] Jean-François Collard, Paul Feautrier, and Tanguy Risset. Construction of do loops from systems of affine constraints. Technical Report 93-15, Laboratoire de l'Informatique du Parallélisme, may 1993.

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

[3] Alain Darte and Yves Robert. The alignment problem for perfect uniform loop nest: Np-completeness and heuristics. In J.J. Dongarra and B. Tourancheau eds, editors, *Environments and Tools for Parallel Scientific Computing II*, SIAM Press, pages 33–42, 1994.

[4] Alain Darte and Yves Robert. Constructive methods for scheduling uniform loop nests. *IEEE Trans. Parallel Distributed Systems*, 5(8):814–822, 1994.

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

[6] Paul Feautrier and Nadia Tawbi. Résolution de systèmes d'inéquations linéaires; mode d'emploi du logiciel PIP. Technical Report 90-2, Institut Blaise Pascal, Laboratoire MASI (Paris), January 1990.

[7] Stanford Compiler Group. Suif compiler system. World Wide Web document, URL:
`http://suif.stanford.edu/suif/suif.html`.

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

[9] Charles H. Koelbel, David B. Loveman, Robert S. Schreiber, Guy L. Steele Jr., and Mary E. Zosel. *The High Performance Fortran Handbook*. The MIT Press, 1994.

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

[11] PIPS Team. Pips (interprocedural parallelizer for scientific programs). World Wide Web document, URL:
`http://www.cri.ensmp.fr/~pips/index.html`.

[12] PRiSM SCPDP Team. Systematic construction of parallel and distributed programs. World Wide Web document, URL:
`http://www.prism.uvsq.fr/english/parallel/paf/autom_us.html`.