A Linear Algebra Framework for Static HPF Code Distribution

Corinne Ancourt Fabien Coelho
François Irigoin Ronan Keryell
Centre de Recherche en Informatique, Ecole Nationale Supérieure des Mines de Paris*
35, rue Saint Honoré, 77305 Fontainebleau cedex, France

Abstract

High Performance Fortran (hpf) was developed to support data parallel programming for simd and mimd machines with distributed memory. The programmer is provided a familiar uniform logic address space and specifies the data distribution by directives. The compiler then exploits these directives to allocate arrays in the local memories, to assign computations to elementary processors and to migrate data between processors when required. We show here that linear algebra is a convenient framework to encode hpf directives and to synthesize distributed code with space-efficient array allocation, tight loop bounds and minimal communications for **INDEPENDENT** loops. The systematic use of an affine framework makes it easy to prove the compilation scheme correct. The generated code includes traditional optimizations such as guard elimination, message vectorization, and aggregation, overlap analysis,...

1 Introduction

Distributed memory multiprocessors can be used efficiently if each local memory contains the right pieces of data, if local computations use local data and if missing pieces of data are quickly moved between processors.

Macro packages and libraries are available to ease the programmer's burden but the level of details still required transforms the simplest algorithm, e.g. a matrix multiply, into hundreds of lines of code. This fact decreases programmer productivity and jeopardizes portability, as well as the economical survival of distributed memory machines.

Manufacturers and research laboratories, lead by Digital and Rice University, decided in 1991 to shift part of the burden onto compilers by providing the programmer a uniform address space to allocate objects and a (mainly) implicit way to express parallelism. Numerous research projects [15, 19, 35] and a few commercial products had shown that this goal could be achieved and the High Performance Fortran Forum was set up to select the most useful functionalities and to standardize the syntax. The definition of the new language, hpf, was frozen in May 1993 [13] and the first compilers are already available [7, 8, 11, 35, 24]. This quick delivery was made possible by defining a subset of the language which only allows static distribution of data and prohibits dynamic redistribution.

This paper deals with this hpf static subset and shows how simple changes of basis and affine constraints can be used to relate the global memory and computation spaces seen by the programmer to the local memory and computation spaces allocated to each elementary processor. These relations, which depend on hpf directives added by the programmer, are used to allocate local parts of global arrays and temporary copies which are necessary when non-local data is used by local computations. These constraints are also used in combination with the *owner-compute rule* to decide which computations are local to a processor, and to derive loop bounds. Finally they are used to generate send and receive statements required to access non-local data.

These three steps, local memory allocation, local iteration enumeration and data communication, are put together as a general compilation scheme for parallel loops, known as **INDEPENDENT** in hpf, with affine bounds and subscript expressions. This compilation scheme directly generates optimized code which includes techniques such as guard elimination [15], message vectorization and aggregation [35], and overlap analysis [15]. There are no restrictions neither on the kind of distribution, block or cyclic, nor on the rank of array references.

The memory allocation part is independent of parallel loops and can always be used. The relations between the global programmer space and the local processor spaces can also be used to translate sequential loops with a run-time resolution mechanism or with some optimizations.

The reader is assumed knowledgeable in hpf directives [13] and optimization techniques for hpf [15, 35].

The paper is organized as follows. Section 2

*{ancourt,coelho,irigoin,keryell}@cri.ensmp.fr.

shows how HPF directives can be expressed as affine constraints and normalized to simplify the compilation process. Section 3 presents an overview of the compilation scheme and introduces the basic sets *own*, *compute*, *send* and *receive* that are used to allocate local parts of HPF arrays and temporaries, to enumerate local iteration and to generate data exchanges between processors. Section 4 refines these sets to minimize the amount of memory space allocated, to reduce the number of loops whenever possible and to improve the communication pattern. This is achieved by using different coordinates to enumerate the same sets. Examples are shown in Section 5 and the results are compared with previous work in Section 6.

2 HPF directives

In HPF, data distribution is specified in two steps:

1. array elements are mapped onto virtual processor arrays called *templates* by an `ALIGN` directive;
2. templates, and implicitly all array elements mapped on them, are distributed onto HPF processors.

Array elements aligned on the same template are allocated on the same elementary processor. Expressions using these elements can be evaluated locally, without inter-processor communications. This alignment step mainly depends on the algorithm.

These templates are packed in blocks to reduce communication and scheduling overheads without increasing load imbalance too much. The block sizes depend on the target machine, while the load imbalance stems from the algorithm.

Templates can be bypassed by aligning an array on another array, and by distributing array directly on processors. This does not increase the expressive power of the language but implies additional check on HPF declarations. Templates are systematically used in this paper to simplify algorithm descriptions.

Our framework deals with both stages and could easily tackle direct alignment and direct distribution. The next sections show that any HPF directive can be expressed as a set of affine constraints.

2.1 Alignment

The `ALIGN` directive is used to specify the alignment relation between one object and one or many other objects through an affine expression. To avoid denoting constant terms in affine expressions, homogeneous coordinates are used when necessary. The cartesian domains of FORTRAN array and HPF template declarations are expressed by linear homogeneous[1] inequalities such as

$$\tilde{D}a \leq 0, \quad \tilde{T}t \leq 0 \tag{1}$$

where a and t describe the array and template elements respectively.

The alignment of an array `A` on a template `T` is an affine mapping from a to t. Alignments are specified dimension-wise with integer affine expressions as template subscript expressions. each time *only one* index of the template and once is used in an affine expression it can *no longer* be used in another expression of the same alignment [13]. Each array index can be used at most once in a subscript expression in any given alignment [13], and each subscript expression cannot contain more than one index. Thus the relation between `A` and `T` is given by

$$t = \tilde{A}a \tag{2}$$

where $\tilde{A}$ is a matrix with at most one non-zero integer element per column and per row (without considering the last homogeneous column).

Equation (2) cannot express replication of elements of `A` on dimensions of `T` although it is allowed by HPF. A projection matrix $\tilde{R}$ is added to leave unspecified template dimensions unconstrained and Equation (2) becomes:

$$\tilde{R}t = \tilde{A}a \tag{3}$$

The same formalism can also deal with a chain of alignment relations (`A` is aligned on `B` which is aligned on `C`,...) as allowed in HPF.

The relation $\tilde{T}t \leq 0$ is redundant with this last equation and FORTRAN array declaration inequalities because, according to HPF semantics, an array element cannot be aligned outside a template.

These notations are illustrated with four examples in a seismic 2D benchmark translated from FORTRAN 77 to HPF. The declaration

```
      real V(3:200,2:100)
!HPF$ template T(3:200,2:100)
!HPF$ align V(I,J) with T(I,J)
```

means a perfect alignment translated in

$$\tilde{D} = \tilde{T} = \begin{pmatrix} -1 & 0 & 3 \\ 1 & 0 & -200 \\ 0 & -1 & 2 \\ 0 & 1 & -100 \end{pmatrix}$$
$$\tilde{R} = \tilde{A} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$$

with

[1] Homogeneous matrices are labelled with "$\tilde{\ }$".

$$t = \begin{pmatrix} t_1 \\ t_2 \\ 1 \end{pmatrix}, \quad a = \begin{pmatrix} a_1 \\ a_2 \\ 1 \end{pmatrix}$$

in the homogeneous coordinates. The second declaration

```
      real UINT_W(10:40)
!HPF$ align UINT_W(I) with T(5*I-7,2)
```

is represented by

$$\widetilde{R} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}, \widetilde{A} = \begin{pmatrix} 5 & -7 \\ 0 & 2 \end{pmatrix}, \widetilde{D} = \begin{pmatrix} -1 & 10 \\ 1 & -40 \end{pmatrix}$$

The third one

```
      real U(198,0:197,3)
!HPF$ align U(I,J,*) with T(I,J)
```

leads to

$$\widetilde{R} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}, \quad \widetilde{A} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$
$$\widetilde{D} = \begin{pmatrix} -1 & 0 & 0 & 1 \\ 1 & 0 & 0 & -198 \\ 0 & -1 & 0 & 0 \\ 0 & 1 & 0 & -197 \\ 0 & 0 & -1 & 1 \\ 0 & 0 & 1 & -3 \end{pmatrix}$$

since the third dimension of the array is collapsed on the template.

Finally the last example uses replications along the lines of the template and the second dimension of the array is collapsed:

```
      real SISMO(60,1000)
!HPF$ align SISMO(I,*) with T(*,2*I+50)
```

which is represented by

$$\widetilde{R} = \begin{pmatrix} 0 & 1 & 0 \end{pmatrix}, \quad \widetilde{A} = \begin{pmatrix} 2 & 0 & 50 \end{pmatrix}$$
$$\widetilde{D} = \begin{pmatrix} -1 & 0 & 1 \\ 1 & 0 & -60 \\ 0 & -1 & 1 \\ 0 & 1 & -1000 \end{pmatrix}$$

To summarize, "*" in an array reference induces a zero column in $\widetilde{A}$ and "*" in a template reference a zero column in R. When no replication is specified, R is the identity matrix.

2.2 Distribution

Once optional alignments are defined in HPF, objects or aligned objects can be distributed on cartesian processor sets, declared like arrays and templates, and represented by inequality:

$$\widetilde{P}p \leq 0 \tag{4}$$

Each dimension can be distributed by block, or in a cyclic way on the processor set or even collapsed on the processors. HPF allows a different block size for each dimension. The extents (n in `BLOCK(`n`)` or `CYCLIC(`n`)`) are stored in a diagonal matrix, C.

Such a distribution may be written as a linear relation between the processor coordinate p, the template coordinate t and two additional variables, ℓ and c. Vector ℓ represents the offset within one block in one processor. Vector c represents the number of wrap-arounds that must be performed to allocate blocks cyclically on processors.

$$\widetilde{\Pi}t = CPc + Cp + \ell \tag{5}$$

The projection matrix $\widetilde{\Pi}$ is needed when some dimensions are collapsed on processors. These dimensions are thus orthogonal to the processor parallelism and can be eliminated. This matrix and the vector t are the only homogeneous objects in Equation (5).

The usual modulo and integer division operators are replaced by predefined constraints on p and additional constraints on ℓ. Vector c is implictly constrained by array declarations and, redundantly, by the `TEMPLATE` declaration.

The diagonal matrix P describes the processors geometry and each element of its diagonal describes the number of processors in each dimension. It is related to the homogeneous matrix $\widetilde{P}$ which declares the processors geometry and offset. Since P is used instead of $\widetilde{P}$ in Equation (5), the lower bounds are taken into account at the level of $\widetilde{\Pi}$.

Specifying `BLOCK(`n`)` in a distribution instead of `CYCLIC(`n`)` is equivalent but tells the compiler that the distribution will not wrap around and the Equation (5) can be simplified by zeroing the c coordinate for this dimension. This additional equation reduces the number of free variables and, hence, remove a loop in the generated code. `CYCLIC` is equivalent to `CYCLIC(1)` and `BLOCK` without parameter is equivalent to `BLOCK(`$\lceil \frac{e_t}{e_p} \rceil$`)` where e_t and e_p are the template and processor extents in the matching dimensions. As `BLOCK`, the `CYCLIC` declaration is translated into an additional equation, $\ell = 0$. The variables are bounded by:

$$\widetilde{P}p \leq 0, \; \widetilde{T}t \leq 0, \; 0 \leq C^{-1}\ell < 1$$

For example, the HPF declaration

```
!HPF processors PE(5:20,6:10)
!HPF distribute T(block,block) onto PE
```

with the same template as in the first example de-

```
      real A(0:42)
!HPF$ template T(0:127)
!HPF$ processors P(0:3)
!HPF$ align A(i) with T(3*i)
!HPF$ distribute T(cyclic(4))
      A(0:42:3) = ...
```

Figure 1: Example in CHATTERJEE *et al.* [10]

fines the matrices

$$\widetilde{P} = \begin{pmatrix} -1 & 0 & 5 \\ 1 & 0 & -20 \\ 0 & -1 & 6 \\ 0 & 1 & -10 \end{pmatrix}, \quad \widetilde{\Pi} = \begin{pmatrix} 1 & 0 & 3 \\ 0 & 1 & 4 \end{pmatrix}$$
$$C = \begin{pmatrix} 13 & 0 \\ 0 & 20 \end{pmatrix}, \quad P = \begin{pmatrix} 16 & 0 \\ 0 & 5 \end{pmatrix}$$

since $\lceil \frac{200-3+1}{20-5+1} \rceil = 13$ and $\lceil \frac{100-2+1}{10-6+1} \rceil = 20$. The constant terms of Π are chosen to exactly align the lower bounds of the processors and the templates.

In this example, the distribution is of type `BLOCK` in every dimension, the CPc is useless and may be discarded. But the addition of equation $c = 0$ is preferred because it leads to a unique algorithm for any kind of distribution.

2.3 Normalization

To clarify the exposition of our compilation scheme, HPF declarations are normalized. Every declaration of array, processor set or template is translated to have 0 as lower bound. This obviously implies no loss of generality for the method. This normalization lets us get rid of homogeneous matrices and replace them by diagonal matrices.

Another normalization is performed. If a dimension of a template is collapsed on the processors, the corresponding coordinate is useless because even if it is used to distribute the dimension of an array, this dimension will be collapsed on the processors. Thus it can be discarded from the equations. These two normalization steps simplify the inequalities used by our algorithms and allow removing the $\widetilde{\Pi} = I$ matrix from Equation (5).

All the declaration of distributions, arrays, processors and templates can thus be rewritten as

$$\begin{array}{lcl} 0 \leq C^{-1}\ell < 1 & , & 0 \leq D^{-1}a < 1 \\ 0 \leq P^{-1}p < 1 & , & 0 \leq T^{-1}t < 1 \\ Rt = Aa + t_0 & , & t = CPc + Cp + \ell \end{array}$$

where t_0 is the coordinate in the template of the first element of the aligned array. Strict rational inequalities are in fact replaced by non-strict integer ones by multiplying them by the determinant Δ of the non-inverted matrix. The non-strict upper bound is $\Delta - 1$. For instance the example given in [10] (Figure 1) is already normalized and can be translated in the following set of affine constraints $D = (43), T = (128), A = (3), t_0 = (0), R = (1), C = (4), P = (4)$.

```
!HPF$ independent
      forall(i,0 ≤ L⁻¹i < 1)
        X(S_X i + X_0)=f(Y(S_Y i + Y_0),Z(S_Z i + Z_0),...)
```

Figure 2: Generic Loop

3 Overview of the compilation scheme

The compilation scheme is based on the HPF declarations, the owner-compute rule and the SPMD paradigm, and deals with `INDEPENDENT` loops. Loop bound expressions and array subscript expressions are assumed affine. Such a parallel loop independently assigns elements of some lhs `X` with some rhs expression f on variables `Y`, `Z`,... (Figure 2).

The loop is normalized to start with iteration 0 and to use unit steps. The upper bounds are encoded in matrix L. Non-unit steps are transferred into the subscript matrices $S_{\mathtt{X}}, S_{\mathtt{Y}}, S_{\mathtt{Z}}, \ldots$ Since the loop is independent, $S_{\mathtt{X}}$ must be injective.

3.1 Own Set

The subset of `X` allocated on processor p according to HPF directives must be declared in the output program, up to a one-to-one mapping. Let $own_{\mathtt{X}}(p)$ be this subset of `X` and `X'` the local array mapping this subset. Each element of $own_{\mathtt{X}}(p)$ must be mapped to one element of `X'` but some elements of `X'` may not be used. Finding a good mapping involves a trade-off between the memory space usage and the access function complexity. This is studied in Section 4.3. Subset $own_{\mathtt{X}}(p)$ is derived from HPF declarations, expressed in an affine formalism (see Section 2), as:

$$\begin{array}{rl} own_{\mathtt{X}}(p) = \{x \mid \exists t\, \exists c \;\; s.t. & Rt = Ax \\ \wedge & t = CPc + Cp + \ell \\ \wedge & 0 \leq D_{\mathtt{X}}^{-1}x < 1 \\ \wedge & 0 \leq P^{-1}p < 1 \\ \wedge & 0 \leq C^{-1}\ell < 1\} \end{array}$$

Subset $own_{\mathtt{X}}(p)$ is a bounded polyhedral set. It can be mapped onto a FORTRAN array by projection on c and ℓ, or on any equivalent set of variables, *i.e.* up to a one-to-one mapping.

3.2 Compute set

Using the owner-compute rule, the set of iterations local to p, $compute(p)$, is directly derived from the previous set, $own_{\texttt{X}}(p)$:

$$compute(p) = \{i \mid S_{\texttt{X}} i \in own_{\texttt{X}}(p)\}$$

Note that $compute(p)$ and $own_{\texttt{X}}(p)$ are equivalent sets if $S_{\texttt{X}}$ it the identity function. This is used in Section 4.3 to study the allocation of local arrays as a special case of local iterations.

Since i is linked to p, c and ℓ, local iterations can be enumerated by scanning the i polyhedral set with `DO` loops over p, c and ℓ or any equivalent set of variables, i.e. up to a unimodular change of basis. When `X` is replicated, its new values are computed on all processors having a copy. If the number of physical processors is reasonably large, this is faster than a combination of non-redundant computations followed by broadcasts.

3.3 View set

The subset of `Y` (or `Z`,...) that is used by the loop is induced by the set of local iterations:

$$view_{\texttt{Y}}(p) = \{y \mid \exists i \in compute(p) \ s.t.\ y = S_{\texttt{Y}} i\}$$

Note that matrix $S_{\texttt{Y}}$ is not constrained and cannot always be inverted. If the intersection of this set with $own_{\texttt{Y}}(p)$ is non-empty, the usual optimization is to enlarge the local copy $\texttt{Y}'$ so as to include its neighborhood, when it is not too large[2]. This is known as *overlap analysis* [15].

When `Y` (or `X`, or `Z`,...) is referenced many times in the input loop or in the input program, these references must be clustered according to their connexity in the dependence graph [4]. Input dependencies are taken into account as well as usual ones (flow-, anti- and output-dependencies). If two references are independent, they access two distant regions and two different local copies should be allocated to reduce the total amount of memory allocated. A unique copy should be as large as the convex hull of these distant regions. If the two references are dependent, only one local copy should be allocated to avoid any consistency problem between copies. If `Y` is a HPF array, its local elements must be taken into account as a special reference.

The definition of *view* is thus altered to take into account *regions* [34, 20] of arrays. The equations due to subscript expression $S_{\texttt{Y}}$ are replaced by a parametric polyhedral subset of `Y` which can be automatically computed. For instance, the set of references to `Y` in:

```
do I1 = 1, N
   do I2 = 1, N
      X(I1,I2) = Y(I1-1,I2) + Y(I1+1,I2)
               + Y(I1,I2-1) + Y(I1,I2+1)
   enddo
enddo
```

is summarized by the parametric region on $y = (y_1, y_2)$:

$$\begin{array}{lrcl}\{y \mid & y_1 + y_2 & \leq & i_1 + i_2 + 1 \\ \wedge & y_2 + i_1 & \leq & y_1 + i_2 + 1 \\ \wedge & i_1 + i_2 & \leq & y_1 + y_2 + 1 \\ \wedge & y_1 + i_2 & \leq & y_2 + i_1 + 1\}\end{array}$$

Subset $view_{\texttt{Y}}$ is still polyhedral and array bounds can be derived by projection, up to a change of basis.

If regions are not precise enough because convex hulls are used to summarize multiple references, it is possible to use additional parameters to exactly express a set of references [2]. This might be useful for red-black SOR.

If the region of `Y` includes HPF local elements, additional equations must be added to follow the standard allocation scheme and to access all elements in a unique way from processor p at cycle c:

$$\begin{array}{rcl} t' & = & Ay \\ t' & = & CPc + Cp + \ell' \end{array}$$

Vectors c and p are constrained as usual but vector ℓ' is no more constrained by C. It is indirectly constrained by y and the `Y` region. New variables t' and ℓ' are added to avoid clashes with t and ℓ which appear in $compute(p)$.

3.4 Send and Receive Sets

The set of elements that are owned by p and used by other processors and the set of elements that are used by p but owned by other processors are both intersections of previously defined sets:

$$send_{\texttt{Y}}(p) = \{(p', y) \mid y \in own_{\texttt{Y}}(p) \cap view_{\texttt{Y}}(p')\}$$

$$receive_{\texttt{Y}}(p) = \{(p', y) \mid y \in view_{\texttt{Y}}(p) \cap own_{\texttt{Y}}(p')\}$$

These two sets cannot always be used to derive the data exchange code. First of all p does not need to exchange data with itself. This cannot be expressed directly as an affine constraint and must be added as a guard in the output code[3]. Secondly, replicated arrays would lead to useless communication: each owner would try to send data to each user. An additional constraint should be added to

[2] The neighborhood is usually considered not too large when it is bounded by a numerical constant.

[3] This could be avoided by exchanging data first with processors p' such that $p' < p$ and then with processors such that $p' > p$, using the lexicographic order. But this additional optimization is not likely to decrease the execution time much.

```
real X′((c,ℓ) ∈ view_X(p)),
      Y′((c,ℓ) ∈ view_Y(p)),
      Z′((c,ℓ) ∈ view_Z(p))
forall(U′ ∈ (Y′,Z′,...))
   if send_U(p) ≠ ∅
      forall((p′,ℓ,c) ∈ send_U(p))
         if p ≠ p′
            send(p′, U′(ℓ,c))
   if receive_U(p) ≠ ∅
      forall((p′,ℓ,c) ∈ receive_U(p))
         if p ≠ p′
            U′(ℓ,c) = receive(p′)
if compute(p) ≠ ∅
   forall ((ℓ,c) ∈ compute(p))
      X′(S_X′(ℓ,c)) = f(S_Y′(ℓ,c),
                          S_Z′(ℓ,c), ...)
```

Figure 3: The output SPMD code.

restrict the set of communications to needed ones. For instance, the cost function could be the minimal Manhattan distance between p and p' or the lexicographically minimal vector $p' - p$ if the interconnection network is a $|p|$-dimension grid. A combined cost function might even be better by taking advantage of the Manhattan distance to minimize the number of hops and of the lexicographical minimum to insure uniqueness. These problems can be cast as linear parametric problems and solved [12].

When no replication occurs, elementary data communications implied by $send_{\mathtt{Y}}$ and $receive_{\mathtt{Y}}$ can be parametrically enumerated in basis (p', u), where u is a basis for Y, the local part of Y. *send* and *receive* are polyhedral sets and algorithms in [2] can be used. If the last component of u is allocated contiguously, vector messages can be generated.

3.5 Output SPMD code

The generic output SPMD code, parametric on p the local processor, is shown in Figure 3.

Communications between processors can be executed in parallel when asynchronous send/receive calls and/or parallel hardware is available as in TRANSPUTER-based machines or in the PARAGON. The guards $send_{\mathtt{U}}$,... can be omitted if unused HPF processors are not able to free physical processors for other useful tasks. The loop bound correctness ensures that no spurious iterations or communications occur.

The parallel loop on U′ can be exchanged with the parallel inner loop on p' included in the `forall`. This other ordering makes message aggregation possible.

Note that the set of local iterations $compute(p)$ is no longer expressed in the i frame but in the (c, ℓ) frame. Additional changes of basis or changes of coordinates must be performed to minimize the space allocated in local memory and to generate a minimal number of loops. Each change must exactly preserve integer points, whether they represent array elements or iterations. This is detailed in the next section.

4 Refinement

The pseudo-code shown in Figure 3 is still far from Fortran. Additional changes of coordinates are needed to generate proper Fortran declarations and loops.

4.1 Iterations enumeration

A general method to enumerate local iterations is described below. It is based on searching a *good* lattice basis to scan them in an appropriate order, and on using unimodular transformations to switch from the user visible frame to the local frame.

The subscript function S is supposed to be affine. All object declarations are supposed normalized with 0 as lower bound. The HPF array mapping is also normalized: $\Pi = I$ (Section 2.3). Furthermore, the algorithm is presented when no replication is requested ($R = I$) to make it clearer.

Under these assumptions and according to Section 2, HPF declarations and Fortran references are represented by the following set of equations:

$$\left.\begin{array}{rl} \text{alignment} & t = Aa + t_0 \\ \text{distribution} & t = CPc + Cp + \ell \\ \text{affine reference} & a = Si + a_0 \end{array}\right\} \qquad (6)$$

where p is the processor vector, ℓ the local offset vector, c the cycle vector, a the accessed element and i the iteration. In order to simplify the following discussion, it is assumed that every dimension of the template has a matching array dimension which is aligned to that dimension, so a is decomposed as $a_|$ for the aligned dimensions, and a_+ for not aligned ones (corresponding to null columns of A). Let $|v|$ denotes the number of variables of vector v. S is of rank $|i|$, since the iterations are independent. Under these assumptions, we have $|t| = |c| = |p| = |\ell| = |a_||$.

Let x be the following set of variables:

$$x = (p, \ell, c, a_+, a_|, t, i) \qquad (7)$$

They are needed to describe the access. Equation (6) can be rewritten as $Fx = f_0$. The set of interest is the lattice $\mathcal{F} = \{x | Fx = f_0\}$. The goal is to find a parametric solution for each component of x, where p would remain unconstrained. This allows each processor to scan its part of $\mathcal{F}$ with minimal control overhead.

Let x' be the subset of x:

$$x' = (p, \ell, c, a_+) \tag{8}$$

x' is a set of free variables in $\mathcal{F}$. Indeed, t can be computed with the distribution equation and x'. Then $a_|$ is derived with the alignment equation, if it is non-empty. Finally $a_|$ and a_+ and the affine subscript expression define i. Moreover these variables are independent because F is of rank $2|t| + |i|$ (F is non-degenerated), with $5|t| + |a_+| + |i|$ variables, so there are at most $3|t| + |a_+| = |x'|$ free variables.

Set $\mathcal{F}$ is implicitly defined but a parametric definition is needed to enumerate its elements. Parameters can be found by computing a HERMITE form [32] of integer matrix F. This form associates to F three matrices, H, P and Q, such that $H = PFQ$. P is a permutation, H a lower triangular integer matrix and Q a unimodular change of basis. Since F is non-degenerated, no permutation is needed and $P = I$. By definition, H is a lower triangular matrix which can be decomposed as $(H_L\ 0)$. Let $v = Q^{-1}x$ be a parameter vector in a new basis. Vector v can also be decomposed in two components like H: $v = (v_0, v')$, where $|v'| = |x'|$. A solution x_0 of $Fx = f_0$ is obtained by computing $v_0 = H_L^{-1} f_0$ and $x_0 = Q_0 v_0$. By construction, H does not constrain v' and Q can also be decomposed like v as $Q = (Q_0\ F')$. The parametric solutions are $x = x_0 + F'v'$ and set $\mathcal{F}$ can be expressed in a parametric way:

$$\mathcal{F} = \{x | \exists v'\ s.t.\ x = x_0 + F'v'\} \tag{9}$$

However the generated code has to be parametrized by the processor vector p. This can be achieved by another change of basis. F' is non-degenerated since it links the free variables of v to x. Under this condition, its HERMITE form $H' = P'F'Q'$ is obtained without permutations and $P' = I$. Let $u = Q'^{-1}v'$ define a new basis, then

$$x - x_0 = F'v' = H'u \tag{10}$$

The new generating system of $\mathcal{F}$ is based on a triangular transformation between x and u (Equation (10)). Since H' is a lower triangular integer matrix, and processor vector p is the first set of variables of x, p is only defined by the first $|p|$ variables of u.

This is designed to parametrize the scanning of $\mathcal{F}$ for each processor p_0. The first $|p|$ variables of u are functions of p_0 through the upper left $|p| \times |p|$ submatrix. The remaining part of H' is then used to generate the elements of $\mathcal{F}$ that are on p_0, by enumerating u values.

Let's explain why the u's defining x in $\mathcal{F}$ are easy to scan with `DO` loops. Let $Kx \leq k$ be the polyhedron that constrains x. These constraints are coming from declarations, HPF directives and normalized loop bounds. They are:

$$\begin{array}{lcl} 0 \leq P^{-1}p < 1 & , & 0 \leq C^{-1}\ell < 1 \\ 0 \leq T^{-1}t < 1 & , & 0 \leq D^{-1}a < 1 \\ 0 \leq L^{-1}i < 1 & & \end{array}$$

Using (10) the constraints on u can be written $K(x_0 + H'u) \leq k$, that is $K'u \leq k'$, where $K' = KH'$ and $k' = k - Kx_0$.

Algorithms presented in [3] can be used to generate the loop nest enumerating the local iterations. When S is of rank $|a|$, optimal code is generated because no projections are required. Else, the quality of the control overhead depends on the accuracy of integer projections [29], but the correctness does not.

The correctness of this enumeration scheme stems from the unimodularity which preserves the one-to-one relationship between integer points and iterations, from the independent parallelism of the loop which allows any enumeration order, and from the choice of p as first component of x, which parametrizes the solution by the processor identity.

Replication is handled by a similar algorithm. The method is applied on aligned template dimensions. Then the generated SPMD code is not parametrized by the processor identity on the replicated dimension, thus every such processor also scans its local data, replicated or not. If arrays are mapped on one element along a template dimension, guards have to be inserted to select the proper processors.

4.2 Symbolic resolution

The previous method can be applied in a symbolic way, when the components of the variables are not coupled and thus can be dealt with independently. This is the case when array sections are used as in [10, 18, 33]. Equations (6) then become for a given dimension:

$$\left.\begin{array}{rl} \text{alignment} & t = \alpha a + t_0 \\ \text{distribution} & t = \pi\gamma c + \gamma p + \ell \\ \text{affine reference} & a = \sigma i + a_0 \end{array}\right\} \tag{11}$$

where π is the number of processors (a diagonal coefficient of P), and γ the block size (*idem* C). In order to simplify the symbolic resolution, variables a and t are eliminated. The matrix form of the system is then $f_0 = (\alpha a_0 + t_0)$, $x = (p, \ell, c, i)$ and $F = (\ \gamma \quad 1 \quad \pi\gamma \quad -\alpha\sigma\ )$.

$\forall p_0 \in [0 \ldots \pi - 1],\ u_1 = p_0$

do u_2 = $\frac{-b_u-\gamma p_0+g-1}{g}$, $\frac{-b_l-\gamma p_0}{g}$

  do u_3 = $\frac{-\omega u_2+\frac{\pi\gamma}{g}-1}{\frac{\pi\gamma}{g}}$, $\frac{\beta-1-\omega u_2}{\frac{\pi\gamma}{g}}$

    $x = H'u + x_0$

Figure 4: Generated code

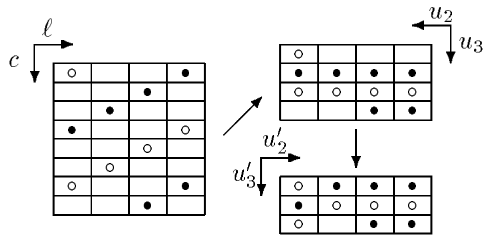

Figure 5: Packing of elements

The HERMITE normal form is $H = \begin{pmatrix} 1 & 0 & 0 & 0 \end{pmatrix} = PFQ$, with $P = I$ and:

$$Q = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & -\gamma & -\pi\gamma & \alpha\sigma \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Let g, μ and ω be such that g is $\gcd(\pi\gamma, \alpha\sigma)$ and $g = \pi\gamma\mu - \alpha\sigma\omega$ is the BEZOUT identity. The HERMITE form H' of the three rightmost columns of Q is such that $x - x_0 = H'u$ with:

$$H' = \begin{pmatrix} 1 & 0 & 0 \\ -\gamma & -g & 0 \\ 0 & \mu & \frac{\alpha\sigma}{g} \\ 0 & \omega & \frac{\pi\gamma}{g} \end{pmatrix}, x_0 = \begin{pmatrix} 0 \\ \alpha a_0 + t_0 \\ 0 \\ 0 \end{pmatrix}$$

This links the three unconstrained variables u to the elements x of the original lattice $\mathcal{F}$. Variables a and t can be retrieved using Equations (11).

The translation of conses traints on x to u gives a way to generate a loop nest to scan the polyhedron. Under the assumption $\alpha > 0$ and $\sigma > 0$, assuming that loop bounds are rectangular[4], and using the constraints in K:

$$0 \le p \le \pi - 1 \quad , \quad 0 \le \ell \le \gamma - 1$$
$$0 \le a \le s - 1 \quad , \quad 0 \le i \le \beta - 1$$

the constraints on x (the a one is redundant if the code is correct) are:

$$\begin{pmatrix} -1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & -1 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} p \\ \ell \\ c \\ i \end{pmatrix} \le \begin{pmatrix} 0 \\ \pi - 1 \\ 0 \\ \gamma - 1 \\ 0 \\ \beta - 1 \end{pmatrix}$$

and can be translated as constraints on u:

$$\begin{pmatrix} -1 & 0 & 0 \\ 1 & 0 & 0 \\ \gamma & g & 0 \\ -\gamma & -g & 0 \\ 0 & -\omega & -\frac{\pi\gamma}{g} \\ 0 & \omega & \frac{\pi\gamma}{g} \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ u_3 \end{pmatrix} \le \begin{pmatrix} 0 \\ \pi - 1 \\ -b_l \\ b_u \\ 0 \\ \beta - 1 \end{pmatrix}$$

where $b_l = -\alpha a_0 - t_0$, $b_u = \gamma - 1 + b_l$.

The resulting generic SPMD code for an array section is shown in Figure 4. As expected, the loop nest is parametrized by p_0, the processor identity. Integer divisions with positive remainders are used in the loop bounds.

4.3 HPF array allocation

The previous two sections can be used to allocate parts of HPF arrays. If the subscript function S is the identity function, local iterations and local elements are strictly equivalent. Thus the constraints on local iterations can be reused as constraints on local elements. However, Fortran array declarations are not as general as Fortran loop bounds, and it is interesting to add another change of frame to fit Fortran specifications for array declarations and to reduce the amount of allocated memory.

The local array elements are packed onto local memories dimension by dimension. The geometric intuition of the packing scheme for one dimension, in the example in [10] displayed Figure 1, is shown in Figure 5 for the first processor of the dimension. The first grid is the part of the template local to the processor, in the cycle and offset coordinate system. The second grid shows the same grid through unimodular transformation from x to u. The third one is the final packed form.

The components in x' (Definition (8)) related to the current dimension are considered. If the coordinate is in a_+, the dimension is collapsed, and no packing is needed, so the initial declaration is preserved. If the dimension is aligned, there are three corresponding coordinates (p, ℓ, c). For every processor p, local (ℓ, c) pairs have to be packed onto a smaller area. This is done by first packing up the elements along the columns, then by removing the empty ones. A dual method is to pack first along the lines, then removing the empty ones. Formulae are derived below to perform this task.

Let's consider the result of the above symbolic resolution, when the subscript expression is the identity ($\sigma = 1, a_0 = 0$). The equation between u and the free variables of x is obtained by selecting

[4] This assumption is, of course, not necessary for the general algorithm described in Section 4.1.

$$0 \leq p \leq \pi - 1, \;\; 0 \leq \ell \leq \gamma - 1$$

$$\left\lfloor \frac{t_0}{\gamma\pi} \right\rfloor \leq c \leq \left\lfloor \frac{\alpha(s-1)+t_0}{\gamma\pi} \right\rfloor$$

$$\texttt{array A'(0:}\left\lceil \frac{c_{up}-c_{low}+1}{\frac{\alpha}{g}} \right\rceil\texttt{,0:}\left\lceil \frac{\gamma}{g} \right\rceil\texttt{)}$$

Figure 6: Local new declaration

the first $|x'|$ lines of H'. If $H'' = (I\,0)H'$ is the selected matrix, we have $x' - x'_0 = H''u$, *i.e.*:

$$\begin{pmatrix} p \\ \ell - t_0 \\ c \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ -\gamma & -g & 0 \\ 0 & \mu & \frac{\alpha}{g} \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ u_3 \end{pmatrix}$$

Variables (u_2, u_3) could be used to define an efficient packing, since holes are removed by the change of basis (Figure 5) but the non diagonal terms of H'' induce poor array declaration bounds. The space has to be skewed in order to match a rectangle as closely as possible, if space is at a premium, and if more complex, non-affine access functions are acceptable. Let G be the positive diagonal integer matrix composed of the absolute value of the diagonal coefficients of H''.

$$\text{alloc} \;:\; u' = \lfloor G^{-1}(x' - x'_0) \rfloor = \lfloor G^{-1}H''u \rfloor \quad (12)$$

G provides the right parameters to perform the proposed packing scheme. To every u a vector u' is associated through formula 12. This formula introduces an integer division. Let's show why u' induces a better mapping of array elements on local memories than u. Since H'' is lower triangular, formula 12 can be rewritten:

$$\forall i \in [1 \ldots |u|], u'_i = \pm u_i + \frac{\sum_{j=1}^{i-1} h_{i,j} u_j}{|h_{i,i}|} \quad (13)$$

This shows that alloc(u) is bijective. If u^a and u^b are different vectors, and i is the first dimension for which they differ, formula 13 shows that u'^a_i and u'^b_i will also differ. The function is also surjective, since the property allows to construct a vector that matches any u' by induction on i.

Two of the three components of x', namely p and ℓ, are explicitly bounded in K. Implicit bounds for the third component, c, are obtained by projecting K on c. These three pairs of bounds, divided by G, are used to declare the local part of the HPF array. Figure 6 shows the resulting declaration for the local part of the array, in the general symbolic case, for one dimension of the array.

The packing scheme induced by u' is better than the one induced by u because there are no non-diagonal coefficients between u' and x' that would introduce a waste of space, and u' is as packed as u because contiguity is preserved by formula 13. The row packing scheme would have been obtained by choosing $x' = (p, c, \ell)$ instead of $x' = (p, \ell, c)$.

These packing schemes define two parameters (u'_2, u'_3) to map one element of one dimension of the HPF array to the processor's local part. The local array declaration can be linearized with the u'_3 dimension first, if the Fortran limit of 7 dimensions is exceeded.

$\alpha = 3, t_0 = 0, \sigma = 3, a_0 = 0, \gamma = 4, \pi = 4, \beta = 15$

$\forall p_0 \in [0 \ldots 3]$
do u'_2 = 0, 3
  $lb_3 = \frac{16p_0+4u'_2+8}{9}$, $ub_3 = \frac{16p_0+4u'_2+7}{9}$
  $lb'_3 = \frac{9lb_3-4u'_2-16p_0}{3}$
  do u'_3 = lb'_3, $lb'_3 + (ub_3 - lb_3)$
    ...

Figure 7: Simplified code example

4.4 Properties

The proposed iteration enumeration and packing scheme has several properties that should be emphasized. This scheme is compatible with efficient cache exploitation and overlap analysis. Moreover, some improvements can statically enhance the generated code.

If loop u_2 in Figure 4 is reversed, the iteration enumeration order and the packing scheme are such that accesses are contiguous. Thus the local cache, if any, is efficiently used.

The packing scheme is also compatible with overlap analysis techniques [15]. Local array declarations are extended to provide space for border elements that are owned by neighbor processors, and to simplify accesses to non-local elements. The overlap is induced by relaxing constraints on ℓ, which is transformed through the scheme as relaxed constraints on u'_2. This allows overlaps to be simply considered by the scheme. Moreover a translated access in the original loop leading to an overlap is transformed into a translated access in the local SPMD generated code.

The generic code proposed in Figure 4 can be greatly improved in many cases. Integer division may be simplified, or performed efficiently with shifts, or even removed. Node splitting and loop invariant code motion should be used to reduce the control overhead. Constraints may also be simplified, for instance if the concerned elements just match a cycle. Moreover, it is possible to generate the loop nest directly on u', when u is not used in the loop body. For the main example in [10], such transformations produce the code shown in Figure 7.

In the general resolution (Section 4.1) the cycle variables c were put after the local offsets ℓ. The induced inner loop nest is then on c. It may be interesting to exchange ℓ and c orders when the block size is larger than the number of cycles: the loop with the larger range would then be the inner one. This may be useful when elementary processors have some vector capability.

4.5 Allocation of temporaries

Temporary space must be allocated to hold non-local array elements accessed by local iterations. For each loop L and array `X`, this set is infered from $compute_L(p)$ and from subscript function $S_{\mathtt{X}}$. For instance, references to array `Y` and `Z` in Figure 10 require local copies. Because of `Y`'s distribution, overlap analysis is fine but another algorithm is necessary for `Z`.

Let's consider the generic loop in Figure 2, and assume that local elements of `Y` cannot efficiently be stored using overlap analysis techniques. First of all, if all or most of local elements of the lhs reference are defined by the loop nest, and if $S_{\mathtt{Y}}$ is of rank $|i|$, temporaries can be stored as `X`. If $S_{\mathtt{Y}}$ is of rank $|i|$, and if the access function $S_{\mathtt{X}}$ use one and only one index in each dimension, the resulting access pattern is still HPF like, so the result of Section 4.3 can be used to allocate the temporary array. Otherwise, another multi-stage change of coordinates is required to allocate a minimal area.

The general idea of the temporary allocation scheme is first to reduce the number of necessary dimensions for the temporary array, then to use this new basis for declaration, or a related one through a new change of coordinates if it is possible and desired to reduce the allocated space.

The set of local iterations, $compute(p)$, is now defined by a new basis and new constraints, such that $x - x_0 = H'u$ (Equation (10)) and $K'u \leq k'$. Some lines of H' define iteration vector i, which is part of x (Equation (7)). Let H'_i be this sub-matrix: $i = H'_i u$.

Let $H_{\mathtt{Y}} = P_{\mathtt{Y}} S_{\mathtt{Y}} Q_{\mathtt{Y}}$ be $S_{\mathtt{Y}}$'s HERMITE normal form. Let $v = Q_{\mathtt{Y}}^{-1} i$ be the new parameters, then $S_{\mathtt{Y}} i = P_{\mathtt{Y}}^{-1} H_{\mathtt{Y}} v$. Vector v can be decomposed as (v', v'') where $v' = \Pi v$ contribute to the computation of i and v'' belongs to $H_{\mathtt{Y}}$'s kernel. If $H_{\mathtt{Y}L}$ is a selection of the non-zero columns of $H_{\mathtt{Y}} = (H_{\mathtt{Y}L} 0)$, then we have:

$$\begin{aligned} a_{\mathtt{Y}} - a_{\mathtt{Y}0} &= P_{\mathtt{Y}}^{-1} H_{\mathtt{Y}L}\, v' \\ v' &= \Pi Q_{\mathtt{Y}}^{-1}\, i \end{aligned}$$

and by substituting i:

$$v' = \Pi Q_{\mathtt{Y}}^{-1} H'_i u$$

input: $\{y = f.w, Ww \leq 0\}$
output: $\{y' = f'.w\}$

initial system: $f^0 = f$
for $i = 1 \ldots (|w| - 1)$
  $g_i = \gcd(\{f_j^{i-1}, j > i\})$
  $s_i = \text{extent}(\sum_{j \leq i} f_j^{i-1} w_j, W)$
  if $(s_i \geq g_i)$
    then $f^i = f^{i-1}$
    else $\forall j \leq i, f_j^i = f_j^{i-1}$ and $\forall j > i, f_j^i = \frac{f_j^{i-1}}{g_i} s_i$
end for
$f' = f^{|w|-1}$

Figure 8: Heuristic to reduce allocated space

It is sometime possible to use x' instead of u'. For instance, if the alignments and the subscript expressions are translations, i is an affine function of x' and v' simply depends on x'. The allocation algorithm is based on u but can also be applied when vxp is used instead.

Since the local allocation does not depend on the processor identity, the first $|p|$ components of u should not appear in the access function. This is achieved by decomposing u as (u', u'') with $|u'| = |p|$ and $\Pi Q_{\mathtt{Y}}^{-1} H'_i$ in $(F_P, F_{\mathtt{Y}})$ such that:

$$v' = F_P u' + F_{\mathtt{Y}} u''$$

and the local part v'' is introduced as:

$$v'' = F_{\mathtt{Y}} u''$$

Then the first solution is to compute the amount of memory to allocate by projecting constraints K onto v''. This is always correct but may lead to a waste of space because periodic pattern of accesses are not recognized and *holes* are allocated. In order to reduce the amount of allocated memory, a heuristic, based on large coefficients in $F_{\mathtt{Y}}$ and constraints K is used.

Each dimension, i.e. component of v'', is treated independently. Let $F_{\mathtt{Y}}^i$ be a line of $F_{\mathtt{Y}}$ and y the corresponding component of v'': $y = fv''$. A change of basis G is applied to v'' to reduce f to a simpler equivalent linear form f' where each coefficient appears only once and where coefficients are sorted by increasing order. For instance:

$$f = \begin{pmatrix} 16 & 3 & -4 & -16 & 0 \end{pmatrix}$$

is replaced by:

$$\begin{aligned} f &= f'G \\ G &= \begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & -1 & 0 \end{pmatrix} \\ f' &= \begin{pmatrix} 3 & 4 & 16 \end{pmatrix} \end{aligned}$$

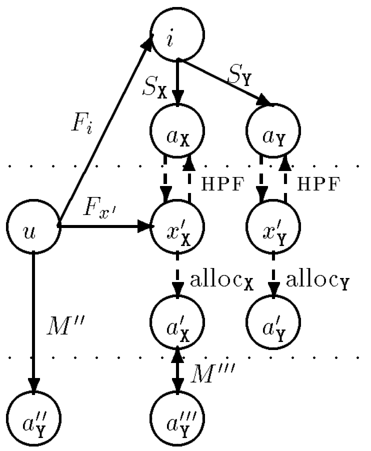

Figure 9: Relationships between frames

and v'' is replaced by $w = Gv''$. Constraints K on u are rewritten as constraints W on w by removing the constraints due to processors and by using projections and G.

Linear form f is then processed by the second part of the heuristic, shown in figure 8. It reduces the extent by $\frac{s_i}{g_i}$ at each step, if the constraints W shows that there is a hole. A hole exists when g_i is larger than the extent of the partial linear form being built, under constraints W. Linearity of access to temporary elements is preserved.

The correctness of the scheme is shown by proving that a unique location y' is associated to each y. Further insight on this problem, the minimal covering of a set by interval congruences, can be found in Granger [17] and Masdupuy [26].

4.6 Data movements

The relationships between the bases and frames defined in the previous sections are shown in Figure 9. Three areas are distinguished. The top one contains user level bases for iterations, i, and array elements, $a_{\mathtt{X}}$, $a_{\mathtt{Y}}$,... The middle area contains the bases and frames used by the compiler to enumerate iterations, u, and to allocate local parts of HPF arrays, $a'_{\mathtt{X}}$, $a'_{\mathtt{Y}}$,... as well as the universal bases, $x'_{\mathtt{X}}$ and $x'_{\mathtt{Y}}$, used to define the lattices of interest, $\mathcal{F}$ and $\mathcal{F}'$. The bottom area shows new bases used to allocate temporary arrays, $a''_{\mathtt{Y}}$ and $a'''_{\mathtt{Y}}$, as defined in Section 4.5.

Solid arrows denote affine functions between different spaces and dashed arrows denote possibly non-affine functions built with integer divide and modulo operators. A quick look at these arrows shows that u, which was defined in Section 4.1 is the central basis of the scheme, because every other coordinates can be derived from u, along one or more paths.

Two kinds of data exchanges must be generated: updates of overlap areas, and initializations of temporary copies. Overlap areas are easier to handle because the local parts are mapped in the same way on each processor, using the same **alloc** function. Let's consider array X in Figure 9. The send and receive statements are enumerated from the same polyhedron, up to a permutation of p and p', with the same basis u. To each u corresponds only one element in the user space, $a_{\mathtt{X}}$, by construction[5] of u. To each $a_{\mathtt{X}}$ corresponds only one $a'_{\mathtt{X}}$ on each processor[6]. As a result, data exchanges controled by loop on u enumerate the same element at the same iteration.

Because the allocation scheme is the same on each processor, the inner loop may be transformed into a vector message if the corresponding dimension is contiguous and/or the send/receive library supports constant but non-unit strides. Block copies of larger areas also are possible when **alloc** is affine, which is not the general case from a theoretical point of view but should very often happen in real applications.

Temporary arrays like $\mathtt{Y}''$ and $\mathtt{Y}'''$ are more difficult to initialize because there is no such identity between their allocation function as local part of an HPF array, $\mathrm{alloc}_{\mathtt{Y}}$, and their allocation functions as temporaries, $t\_\mathrm{alloc}_{\mathtt{Y}''}$ and $t\_\mathrm{alloc}_{\mathtt{Y}'''}$. However, basis u can still be used. When the temporary is allocated exactly like the master array, e.g. $\mathtt{Y}'''$ and $\mathtt{X}'$, any enumeration of elements u in U enumerates every element $a'''_{\mathtt{Y}}$ once and only one because the input loop is assumed independent. On the receiver side, $a'''_{\mathtt{Y}}$ is directly derived from u. On the sender side, $a_{\mathtt{Y}}$ is computed as $S_{\mathtt{Y}}F_i u$ and a non-affine function is used to obtain $x'_{\mathtt{Y}}$ and $a'_{\mathtt{Y}}$. Vector messages can be used when every function is affine, because a constant stride is transformed into another constant stride, and when such transfers are supported by the target machine. Else, calls to packing and unpacking routines call be generated.

When the temporary is allocated with its own allocation function M, it is more difficult to find a set of u enumerating exactly once its elements. This is the case for copy $\mathtt{Y}''$ in Figure 9. Elements of $\mathtt{Y}''$, $a''_{\mathtt{Y}}$, must be used to compute one related u among many possible one. This can be achieved by using a pseudo-inverse of the access matrix M:

$$u = M^t(MM^t)^{-1}a''_{\mathtt{Y}} \qquad (14)$$

Vector u is the rational solution of equation $a''_{\mathtt{Y}} = Mu$ which has a minimum norm. It may well not belong to $\mathcal{U}$, the polyhedron of meaningful u which

[5] As explained in Section 4.3, allocation is handled as a special case by choosing the identity for $S_{\mathtt{X}}$.

[6] Elements in overlap area are allocated more than once, but on different processors.

```
      real X(0:42,0:42),
     &     Y(0:42,0:42), Z(0:42)

!HPF$ template T(0:150,0:150)
!HPF$ align X(I,J) with T(3*I,3*J)
!HPF$ align Y with X
!HPF$ align Z(I) with T(0,0)
!HPF$ processors P(0:3,0:3)
!HPF$ distribute                      &
!HPF$   T(cyclic(4),cyclic(4)) onto P

!HPF$ independent(I,J)
      do I = 1, 14
         do J = 0, 14
            X(3*I,3*J) = Y(3*I,3*J) +
     &        Y(3*I-1,3*J) + Z(3*I)
         enddo
      enddo
!HPF$ independent(I)
      do I = 0, 14
         Y(I,I) = 1.0
      enddo
```

Figure 10: Code example.

are linked to a user iteration. However, M was built to have the same kernel as $S_{\mathtt{Y}}F_i$. Thus the same element $a_{\mathtt{Y}}$ is accessed as it would be by a regular u. Since only linear transformations are applied, these steps can be performed with integer computations by multiplying Equation (14) with the determinant of MM^t and by dividing the results by the same quantity. The local address, $a'_{\mathtt{Y}}$, is then derived as above. This proves that sends and receives can be enumerated by scanning elements $a''_{\mathtt{Y}}$.

5 Examples

The different algorithms presented in the previous section can be used to distribute the contrived piece of code of Figure 10.

This is an extension of an example given in [10] showing that allocation of HPF arrays may be non-trivial. The reference to `X` in the first loop requires an allocation of `X`$'$ and the computation of new loop bounds. It is not a simple case because the subscript function is not the identity and because a block-cyclic distribution is specified. The two references to `Y` in the first loop are similar but they imply some data exchange between processors and the allocation of an overlap area in `Y`$'$. The reference to `Z` implies the allocation of a temporary array, `Z`$'$, which is independent from the local part of `Z`. The second loop shows that the $compute(p)$ set may have fewer dimensions than the array referenced and that fewer loops have to be generated.

```
c     Local declaration of Y(0:42,0:42):
      real X_hpf(0:7,0:1,0:7,0:1)
      real Y_hpf(0:7,-1:1,0:7,0:1)
c     For each processor (0<=p1<=3,0<=p2<=0) :
c     do I = 1, 14
c        do J = 1, 14
c     ...
c     Send view(Y) to compute(X):
      do u3 = 0, min(p1+1, 3)
        if(u3.ne.p1) then
          do c1 = 0, 7
            do c2 = 0, 7
              do u6 = max(2, divide(2+4*p1+16*c1,3)),
     &                min(42, divide(3+4*p1+16*c1,3))
                if (divide(u6+2,3).le.divide(u6+1,3)) then
                  do u7 = max(0,divide(8+4*p2+16*c2,9)),
     &                    min(14, divide(3+4*p2+16*c2,9))
                    l1 = 3 * u6 - 16 * c1 - 4 * p1
                    l2 = 9 * u7 - 16 * c2 - 4 * p2
                    send(u3, p2, Y_hpf(c1, l1, c2, l2))
                  enddo
                endif
              enddo
            enddo
          enddo
        endif
      enddo
c     ...
c     do I=0,14 Y(I,I)=1.0
      do u3 = 0, 2
        do u4 = 0, 2
          do u5 = max(max(0,divide(2 + 4*p1 + 16*u3,3)),
     &                divide(2 + 4*p2 + 16*u4,3)),
     &            min(min(14,divide(3 + 4*p1 + 16*u3,3)),
     &                divide(3 + 4*p2 +16*u4,3))
            w5 = divide(3*u5 - 4*p1 - 16*u3,3)
            Y_hpf(u3, w5, u4, u6) = 1.0
          enddo
        enddo
      enddo
      end

c     Integer Fortran divide has not the good semantics:
      integer function divide(i,j)
      integer i,j
      if(i.ge.0) then
         divide = i/j
      else
         divide = -(-i+j-1)/j
      endif
      end
```

Figure 11: Output code (excerpts).

The output code is too long to be printed here. Interesting excerpts are shown in figure 11 and commented below.

5.1 HPF Declarations

HPF declarations already are normalized. Templates are fully utilized and $\Pi = I$. Other key parameters are:

$$P = \begin{pmatrix} 4 & 0 \\ 0 & 4 \end{pmatrix}, T = \begin{pmatrix} 150 & 0 \\ 0 & 150 \end{pmatrix}, C = \begin{pmatrix} 4 & 0 \\ 0 & 4 \end{pmatrix}$$

$$A_X = A_Y = \begin{pmatrix} 3 & 0 \\ 0 & 3 \end{pmatrix}, R_X = R_Y = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

$$t_{0X} = t_{0Y} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}, A_Z = \begin{pmatrix} 0 \\ 0 \end{pmatrix}, t_{0Z} = \begin{pmatrix} 0 \\ 0 \end{pmatrix},$$

5.2 Allocation of X′

Array `X` is referenced only once, as a lhs. Thus its local part is directly derived from the set $own_{\mathtt{X}}(p)$ using the algorithm described in section 4.3.

$$own_X(p) = \{ \quad \begin{array}{l} t_1 = 3x_1, t_2 = 3x_2 \\ t_1 = 16c_1 + 4p_1 + \ell_1 \\ t_2 = 16c_2 + 4p_2 + \ell_2 \\ 0 \leq x_1, x_2 \leq 42, \quad 0 \leq p_1, p_2 \leq 3 \\ 0 \leq \ell_1, \ell_2 \leq 3, \quad 0 \leq t_1, t_2 \leq 150 \quad \} \end{array}$$

The equations are parametrically solved and x is replaced by u in the constraints according to the following change of basis:

$$\begin{array}{ll} u_1 = p_1, & 3u_5 - 16u_3 - 4u_1 = \ell_1 \\ u_2 = p_2, & 3u_6 - 16u_4 - 4u_2 = \ell_2 \\ u_3 = c_1, & u_5 = x_1 \\ u_4 = c_2, & u_6 = x_2 \end{array}$$

Then, frame u is replaced by frame u' to avoid non-diagonal terms in the constraints. Frame u' is linked to the user basis x by:

$$\begin{array}{lll} u'_1 = p_1, & u'_2 = p_2, & u'_3 = c_1 \\ u'_4 = c_2, & 3u'_5 = \ell_1, & 3u'_6 = \ell_2 \end{array}$$

and constraints K are projected on each component of u' to derive the declaration bounds of `X′`:

$$\begin{array}{lll} 0 \leq u'_1 \leq 3, & 0 \leq u'_2 \leq 3, & 0 \leq u'_3 \leq 7 \\ 0 \leq u'_4 \leq 7, & 0 \leq u'_5 \leq 1, & 0 \leq u'_6 \leq 1 \end{array}$$

Coordinates u'_1 and u'_2 are the processor coordinates. The other coordinates, u'_3 to u'_6, are the local array coordinates. The FORTRAN declaration of `X′` is shown in Figure 11.

16×16 elements are allocated while 11×11 would be enough. The efficiency factor would be better in the (p, ℓ, c) basis. It would also be better with less contrived aligments and subscript expressions and/or larger blocks.

5.3 Sending Y′

Before the first loop nest is executed, each processor (p_1, p_2) must send some elements of `Y` to its right neighbor $(p_1 + 1, p_2)$, with a wrap-around for the rightmost processors, $(3, p_2)$. The pairs of communicating processors are not accurately represented by convex constraints and the `u3` loop is not as tight as it could. This suggests that central and edge processors should be handled differently when overlap is observed.

The bounds for the inner loops are complicated but they could mostly be evaluated iteratively if node splitting and loop invariant code motion are used. Also, the test on `u6` can be simplified as a modulo operation. Finally `l1` and `l2` can also be evaluated iteratively. Note that the dimensions of `Y′` should be permuted to reduce the stride for a possible vector message along `u7`.

5.4 Local iterations for the second loop

The second loop cannot be represented as an array section assignment, which makes it harder to handle than the first one. The set of local iteration is defined by:

$$compute_{\mathtt{Y}}(p) = \{ \quad \begin{array}{l} y_1 = i, \quad y_2 = i \\ t_1 = 3y_1, \quad t_2 = 3y_2 \\ t_1 = 16c_1 + 4p_1 + \ell_1 \\ t_2 = 16c_2 + 4p_2 + \ell_2 \\ 0 \leq i \leq 14, \quad 0 \leq y_1, y_2 \leq 42 \\ 0 \leq p_1, p_2 \leq 3, \quad 0 \leq \ell_1, \ell_2 \leq 3 \\ 0 \leq t_1, t_2 \leq 150 \quad \} \end{array}$$

The loop bounds were retrieved by projection on u. It was found useless to generate a guard on the processor identity because non-involved processor have nothing to do and because this does not delay the other ones. The guard is hidden in the inner loop bounds.

Note that an extra-loop is generated. `Y` diagonal can be enumerated with only two loops and three are generated. This is due to the use of an imprecise projection algorithm but does not endanger correctness [2]. Further work is needed in this area.

5.5 Integer divide

One implementation of the integer divide is finally shown. The divider is assumed strictly positive, as is the case in all call sites. It necessary because FORTRAN remainder is not positive for negative numbers. It was added to insure the semantics of the output code.

6 Related work

Techniques to generate distributed code from sequential or parallel code using a uniform memory space have been extensively studied since 1988 [9, 28, 39, 30, 15, 22, 16, 6, 25, 2, 21, 27, 31, 23, 5, 19, 4, 8, 11, 38].

The most obvious, most general and safest technique is called run-time resolution[9, 28, 30]. Each instruction is guarded by a condition that is true for processors that must execute it. Each memory address is checked before it is referenced to decide whether the address is local and the reference is executed, whether it is remote, and a receive is executed, or whether it is remotely accessed and a send is executed. This rewriting scheme is easy to implement [11] but very inefficient at run-time because guards, tests, sends and receives are pure overhead. Moreover every processor has to execute the whole control flow of the program, and even for parallel loop, communications may sequentialize the program at run-time [9].

Many optimization techniques have been introduced to handle specific cases. GERNDT introduced overlap analysis in [15] for block distributions. When local array parts are allocated with the necessary overlap and when parallel loops are translated, the instruction guards can very often be moved in the loop bounds and the send/receive statements are globally executed before the local iterations, *i.e.* the loop nest with run-time resolution is distributed into two loop nests, one for communications and one for computations. The communication loops can be rearranged to generate vector messages.

TSENG [35] presents lots of additional techniques (message aggregation and coalescing, message and vector message pipelining, computation replication, collective communication...). He assumes affine loop bounds and array subscripts to perform most optimizations. He only handles `block` and `cyclic(1)` distributions and the alignment coefficient must be 1.

Three recent publications tackle any alignment and distribution but restrict references to array sections. Each dimension is independent of the others as was assumed in Section 4.2. CHATTERJEE *et al.* [10] developed a finite state machine approach to enumerate local elements. No memory space is wasted and local array elements are ordered by FORTRAN lexico-graphic order exactly like user array elements. They are sequentially accessed by `while` loops executing the FSM's, which may be a problem if vector units are available. Moreover, accesses to an auxiliary data structure, the FSM transition map, add to the overhead. Note that the code generated Figure 4 may be used to compute the FSM. In fact the lower iteration of the innermost loop is computed by the algorithm that constructs the FSM.

Two papers by STICHNOTH [33] on one hand and GUPTA *et al.* [18] on the other present two similar methods to solve the same problem. They use array sections but compute some of the coefficients at run-time. GUPTA *et al.* solve the block distribution case and use processor virtualisation to handle cyclic distributions. Arrays are densely allocated as in [10] and the initial order is preserved but no formulae are given. STICHNOTH uses the dual method for array allocation as in [11], that is blocks are first compressed, and the cycle number is used as a second argument.

7 Conclusion

Although the usual affine assumptions are made for loop bounds and subscript expressions, our compilation scheme simultaneously lifts several restrictions. Array references are not restricted to array sections. General HPF alignments and distributions are supported, and the same algorithms also provide optimal code for classical block distributions. Memory allocation is almost 100 % efficient on large blocks and performs quite well on small ones when strange alignments are used. We believe that this slight memory waste is more than compensated by the stride-1 vector load, store, send and receive which can be performed on the copies and which are necessary for machines including vector units. These contiguous accesses also perform well with a cache. The overlap analysis and allocation is integrated to the basic allocation scheme. Finally, most computations are performed at compile-time and no auxiliary data structures are used.

Many partial optimization techniques are integrated in our direct synthesis approach: message vectorization, and aggregation [19], overlap analysis [15]. A new storage management scheme is also proposed. Moreover other optimizations techniques may be applied to the generated code such as vectorization [37], loop invariant code motion [1] and software pipelining [14, 36].

Future work includes an implementation in our HPF compiler [11], extensions to optimize sequential loops, including I/O loops, to overlap communication and computation, and to handle indirections.

Acknowledgments

We are thankful to Pierre JOUVELOT for his careful reading and the many improvements he suggested.

References

[1] Alfred V. AHO, Ravi SETHI, and Jeffrey D. ULLMAN. *Compilers Principles, Techniques, and Tools.* Addison-Wesley Publishing Company, 1986.

[2] Corinne ANCOURT. *Génération automatique de codes de transfert pour multiprocesseurs à mémoires locales.* PhD thesis, Université Paris VI, March 1991.

[3] Corinne ANCOURT and François IRIGOIN. Scanning Polyhedra with DO Loops. In *Symposium on Principles and Practice of Parallel Programming*, April 1991.

[4] Corinne ANCOURT and François IRIGOIN. Automatic Code Distribution. In *Third Workshop on Compilers for Parallel Computers*, July 1992.

[5] Françoise ANDRÉ, Olivier CHÉRON, and Jean-Louis PAZAT. Compiling Sequential Programs for Distributed Memory Parallel Computers with Pandore II. Technical report, IRISA, May 1992.

[6] Françoise ANDRÉ, Jean-Louis PAZAT, and Henry THOMAS. Pandore: A System to Manage Data Dis-

tribution. Publication Interne 519, IRISA, February 1990.

[7] Thomas BRANDES. Efficient Data Parallel Programming without Explicit Message Passing for Distributed Memory Multiprocessors. Internal Report AHR-92 4, High Performance Computing Center, German National Research Institute for Computer Science, September 1992. ftp `ftp.gmd.de`, `gmd/adaptor`.

[8] Thomas BRANDES. Adaptor: A Compilation System for Data Parallel Fortran Programs. Technical report, High Performance Computing Center, German National Research Institute for Computer Science, August 1993. ftp `ftp.gmd.de`, `gmd/adaptor/docs`.

[9] David CALLAHAN and Ken KENNEDY. Compiling Programs for Distributed-Memory Multiprocessors. *The Journal of Supercomputing*, 2:151–169, 1988.

[10] Siddhartha CHATTERJEE, John R. GILBERT, Fred J. E. LONG, Robert SCHREIBER, and Shang-Hua TENG. Generating Local Addresses and Communication Sets for Data-Parallel Programs. In *Symposium on Principles and Practice of Parallel Programming*, 1993.

[11] Fabien COELHO. Étude de la Compilation du *High Performance Fortran*. Master's thesis, Université Paris VI, September 1993. Rapport de DEA Systèmes Informatiques, stage effectué au CRI, ENSMP. Rapport EMP-CRI E/178.

[12] Paul FEAUTRIER. Parametric Integer Programming. *RAIRO Recherche Opérationnelle*, 22:243–268, September 1988.

[13] High Performance Fortran FORUM. High Performance Fortran Language Specification. Version 1.0 CRPC-TR 92225, Center for Research on Parallel Computation, Rice University, Houston, USA, May 1993.

[14] Franco GASPERONI and Uwe SCHEIEGELSHOHN. Scheduling Loop on Parallel Processors: A Simple Algorithm with Close to Optimum Performance. Lecture Note, INRIA, 1992.

[15] Hans Michael GERNDT. *Automatic Parallelization for Distributed-Memory Multiprocessing Systems.* PhD thesis, University of Vienna, 1989.

[16] Hans Michael GERNDT and Hans Peter ZIMA. Optimizing Communication in Superb. In *CONPAR90*, pages 300–311, 1990.

[17] Philippe GRANGER. *Analyses Sémantiques de Congruence.* PhD thesis, École Polytechnique, July 1991.

[18] S. K. S. GUPTA, S. D. KAUSHIK, S. MUFTI, S. SHARMA, C.-H. HUANG, and P. SADAYAPPAN. On compiling Array Expressions for Efficient Execution on Distributed-Memory Machines. In *International Conference on Parallel Processing*, pages II–301–II–305, August 1993.

[19] Seema HIRANANDANI, Ken KENNEDY, and Chau-Wen TSENG. Compiling Fortran D for MIMD Distributed-Memory Machines. *Communications of the ACM*, 35(8):66–80, August 1992.

[20] François IRIGOIN. Interprocedural Analyses for Programming Environment. In Jack J. DONGARA and Bernard TOURANCHEAU, editors, *Environments and Tools for Parallel Scientific Computing*, pages 333–350, Saint-Hilaire-du-Touvet, September 1992. North-Holland, Amsterdam, NSF-CNRS.

[21] Charles KOELBEL and Piyush MEHROTRA. Compiling Global Name-Space Parallel Loops for Distributed Execution. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):440–451, October 1991.

[22] Charles KOELBEL, Piyush MEHROTRA, and John VAN ROSENDALE. Supporting Shared Data Structures on Distributed Memory Architectures. Technical Report ASD 915, Purdue University, January 1990.

[23] Oded LEMPEL, Shlomit S. PINTER, and Eli TURIEL. Parallelizing a C Dialect for Distributed Memory MIMD Machines. In *Language and Compilers for Parallel Computing*, August 1992.

[24] John LEVESQUE. *FORGE 90 and High Performance Fortran.* Applied Parallel Research, Inc., 1992. `xHPF77` presentation.

[25] J. LI and Marina CHEN. Compiling communication-efficient programs for massively parallel machines. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):361–376, July 1991.

[26] François MASDUPUY. *Array Indices Relational Semantic Analysis using Rational Cosets and Trapezoids.* PhD thesis, École Polytechnique, November 1993. Technical Report EMP-CRI A/247. *To be defended.*

[27] John MERLIN. Techniques for the Automatic Parallelisation of 'Distributed Fortran 90'. SNARC 92 02, University of Southampton, 1992.

[28] Ravi MIRCHANDANEY, Joel S. SALTZ, Roger M. SMITH, David M. NICOL, and Kay CROWLEY. Principles of Runtime Support for Parallel Processors. In *International Conference on Supercomputing*, pages 140–152, July 1988.

[29] William PUGH. A Pratical Algorithm for Exact Array Dependence Analysis. *CACM*, 35(8):102–114, August 1992.

[30] Anne ROGERS and Keshav PINGALI. Process Decomposition Through Locality of Reference. In *ACM SIGPLAN International Conference on Programming Language Design and Implementation*, June 1989.

[31] Anne ROGERS and Keshav PINGALI. Compiling for distributed memory architectures, June 1992.

[32] Alexander SCHRIJVER. *Theory of linear and integer programming.* Wiley, New-York, 1986.

[33] James M. STICHNOTH. Efficient Compilation of Array Statements for Private Memory Multicomputers. CMU-CS-93 109, School of Computer Science, Carnegie Mellon University, February 1993.

[34] Rémi TRIOLET, Paul FEAUTRIER, and François IRIGOIN. Direct Parallelization of Call Statements. In *Proceedings of the ACM Symposium on Compiler Construction*, 1986.

[35] Chau-Wen TSENG. *An Optimising Fortran D Compiler for MIMD Distributed Memory Machines.* PhD thesis, Rice University, Houston, Texas, January 1993.

[36] Jian WANG and Christine EISENBEIS. Decomposed Software Pipelining: A New Approach to Exploit Instruction Level Parallelism for Loop Programs. In *Working Conference on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism*, January 1993.

[37] Hans ZIMA and Barbara CHAPMAN. *Supercompilers for Parallel and Vector Computers.* ACM Press, 1990.

[38] Hans ZIMA and Barbara Mary CHAPMAN. Compiling for Distributed-Memory Systems. *Proceedings of the IEEE*, February 1993.

[39] Hans Peter ZIMA, H. J. BAST, and Hans Michael GERNDT. SUPERB : A Tool for Semi-Automatic MIMD/SIMD Parallelization. *Parallel Computing*, 6:1–18, 1988.