A Linear Algebra Framework for Static HPF Code Distribution*

Technical report A-278-CRI

Corinne ANCOURT, Fabien COELHO, François IRIGOIN, Ronan KERYELL †

Corresponding author: Fabien COELHO.

Centre de Recherche en Informatique,
École Nationale Supérieure des Mines de Paris,
35, rue Saint-Honoré, F–77305 Fontainebleau cedex, FRANCE.
Phone: +33 1 64 69 47 08. Fax: + 33 1 64 69 47 09.
URL: `http://www.cri.ensmp.fr/pips`,
ftp: `anonymous@ftp.cri.ensmp.fr`

November 24, 1995

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 logical 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 powerful framework to encode HPF directives and to synthesize distributed code with space-efficient array allocation, tight loop bounds and vectorized communications for `INDEPENDENT` loops. The generated code includes traditional optimizations such as guard elimination, message vectorization and aggregation, overlap analysis... The systematic use of an affine framework makes it possible to prove the compilation scheme correct.

*An early version of this paper was presented at the Fourth International Workshop on Compilers for Parallel Computers held in Delft, the Netherlands, December 1993.

†{ancourt,coelho,irigoin,keryell}@cri.ensmp.fr, http://www.cri.ensmp.fr/~{...}

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 at the right time 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 parallel machines.

Manufacturers and research laboratories, led 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 [38, 47, 81] 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 initial definition of the new language, HPF, was frozen in May 1993, and corrections were added in November 1994 [36]. Prototype compilers incorporating some HPF features are available [18, 19, 26, 81, 88, 14]. Commercial compilers from APR [64, 65], DEC [71, 17], IBM [42] and PGI [34, 68] are also being developed or are already available. These compilers implement part or all of the HPF Subset, which only allows static distribution of data and prohibits dynamic redistributions.

This paper deals with this HPF static subset and shows how 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-computes 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. HPF's `FORALL` statements or constructs, as well as a possible future `ON` extension to advise the compiler about the distribution of iterations onto the processors, can be translated into a set of independent loops by introducing a temporary array mapped as required to store the intermediate results. These translations are briefly outlined in Figures 1 and 2. The resulting code is a pair of loops which can be compiled by our scheme, following the *owner-computes rule*, if the `ON` clause is put into the affine framework. The `FORALL` translation requires a temporary array due to its SIMD-like semantics. However, if the assigned array is not referenced in the *rhs*, the `FORALL` loop is independent and should be tagged as such to fit directly our scheme. Such *necessary* temporary arrays are not expected to cost much, both on the compilation and execution point of view: The allocated memory is reusable (it may be allocated on the stack), and the copy assignment on local data should be quite fast.

This compilation scheme directly generates optimized code which includes techniques such as guard elimination [38], message vectorization and aggregation [47, 81].

```
! non-independent
!   A in the rhs may
!   induce RW dependences...
FORALL (i=1:n, j=1:m, MASK(i,j))
  A(i,j) = f(A, ...)
```

```
! array TMP declared and mapped as A
! initial copy of A into TMP
!   because of potential RW dependences
INDEPENDENT(j,i)
do j=1, m
  do i=1, n
    TMP(i,j) = A(i,j)
  enddo
enddo
! possible synchro...
INDEPENDENT(j,i)
do j=1, m
  do i=1, n
    if (MASK(i,j)) A(i,j) = f(TMP, ...)
  enddo
enddo
```

Figure 1: Masked `FORALL` to `INDEPENDENT` loops

```
INDEPENDENT(j,i), ON(...)
do j=1, m
  do i=1, n
    A(i,j) = f(...)
  enddo
enddo
```

```
! array TMP(iteration space)
!   mapped as ON(iteration space)
INDEPENDENT(j,i)
do j=1, m
  do i=1, n
    TMP(i,j) = f(...)
  enddo
enddo
! possible synchro...
INDEPENDENT(j,i)
do j=1, m
  do i=1, n
    A(i,j) = TMP(i,j)
  enddo
enddo
```

Figure 2: `ON` to `INDEPENDENT` loops

It is compatible with overlap analysis [38]. There are no restrictions neither on the kind of distribution (general cyclic distributions are handled), nor on the rank of array references (the dimension of the referenced space: for instance rank of `A(i,i)` is 1). The memory allocation part, whether based on overlap extensions, or dealing with temporary arrays needed to store both remote and local elements, 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 [36] and optimization techniques for HPF [38, 81].

The paper is organized as follow. Section 2 shows how HPF directives can be expressed as affine constraints and normalized to simplify the compilation process and its description. 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 iterations 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 method is compared with previous work in Section 6.

2 HPF directives

The basic idea of this work was to show that the HPF compilation problem can be put into a linear form, including both equalities and inequalities, then to show how to use polyhedron manipulation and scanning techniques to compile an HPF program from this linear form. The linear representations of the array and HPF declarations, the data mapping and the loop nest accesses are presented in this section.

HPF specifies data mappings in two steps. First, the array elements are *aligned* with a *template*, which is an abstract grid used as a convenient way to relate different arrays together. Each array element is assigned to at least one template cell thru the `ALIGN` directive. Second, the *template* is *distributed* onto the *processors*, which is an array of virtual processors. Each template cell is assigned to one and only one processor thru the `DISTRIBUTE` directive. The *template* and *processors* are declared with the `TEMPLATE` and `PROCESSORS` directives respectively.

Elements of arrays aligned on the same template cell are allocated on the same elementary processor. Expressions using these elements can be evaluated locally, without inter-processor communications. Thus the alignment step mainly depends on the algorithm. The template elements 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 expressiveness 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 Notations

Throughout this paper, a lower case letter as v denotes a vector of integers, which may be variables and constants. $v_i, (i \geq 0)$ the ith element or variable of vector v. Subscript 0, as in v_0, denotes a constant integer vector. As a convention, a denotes the variables which describe the elements of an array, t is used for *templates* and p for *processors*. An upper case letter as A denotes a constant integer matrix. Constants are implicitly expanded to the required number of dimensions. For instance 1 may denote a vector of 1. $|A|$ denotes the determinant of matrix A.

2.2 Declarations

The data arrays, the *templates* and the *processors* are declared as Cartesian grids in HPF. If a is the vector of variables describing the dimensions of array `A(3:8,-2:5,7)`, then the following linear constraints are induced on a:

$$3 \leq a_1 \leq 8, \quad -2 \leq a_2 \leq 5, \quad 1 \leq a_3 \leq 7$$

These may be translated into the matrix form $D_{\mathtt{A}} a \leq d$ where:

$$D_{\mathtt{A}} = \begin{pmatrix} 1 & 0 & 0 \\ -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & -1 \end{pmatrix}, \quad a = \begin{pmatrix} a_1 \\ a_2 \\ a_3 \end{pmatrix}, \quad d = \begin{pmatrix} 8 \\ -3 \\ 5 \\ 2 \\ 7 \\ -1 \end{pmatrix}$$

Any valid array element must verify the linear constraints, *i.e.* `A(`a_1`,`a_2`,`a_3`)` is a valid array element if Equation (1) is verified by vector a. In the remaining of the paper it is assumed without loss of generality that the dimension lower bounds are equal to 0. This assumption simplifies the formula by deleting a constant offset at the origin. Thus the declaration constraints for `A(0:5,0:7,0:6)` can be written intuitively as

$$0 \leq a < D.1 \tag{1}$$

where

$$D.1 = \begin{pmatrix} 6 & 0 & 0 \\ 0 & 8 & 0 \\ 0 & 0 & 7 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 6 \\ 8 \\ 7 \end{pmatrix}$$

D is a diagonal matrix composed of the sizes of `A`. The rational for storing the sizes in a diagonal matrix instead of a vector is that this form is needed for the distribution formula (see Section 2.4). Likewise the template declaration constraints are $0 \leq t < T.1$ and the processors $0 \leq p < P.1$.

2.3 Alignment

The `ALIGN` directive is used to specify the alignment relation between one object and one or many other objects through an affine expression. 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 array index can be used at most once in a template subscript expression in any given alignment, and each subscript expression cannot contain more than one index [36]. Let us consider the following HPF alignment directive, where the first dimension of the array is collapsed and the most general alignment subscripts are used for the other dimensions:

```
align A(*,i,j) with T(2*j-1,-i+7)
```

it induces the following constraints between the dimensions of `A` and `T` represented respectively by vector a and t:

$$t_1 = 2a_3 - 1, \quad t_2 = -a_2 + 7$$

Thus the relation between `A` and `T` is given by $t = Aa + s_0$ where

$$t = \begin{pmatrix} t_1 \\ t_2 \end{pmatrix}, \; A = \begin{pmatrix} 0 & 0 & 2 \\ 0 & -1 & 0 \end{pmatrix}, \; a = \begin{pmatrix} a_1 \\ a_2 \\ a_3 \end{pmatrix}, \; s_0 = \begin{pmatrix} -1 \\ 7 \end{pmatrix}$$

```
template T(0:99), T'(0:99)
processors P(0:4)
distribute T(block(20)), T'(cyclic(1)) onto P
```

Figure 3: Example 1

A (for alignment) is a matrix with at most one non-zero integer element per column (dummy variables must appear at most once in the alignment subscripts) and per row (no more than one dummy variable in an alignment subscript), and s_0 is a constant vector that denotes the constant shift associated to the alignment. Note that since the first dimension of the array is collapsed the first column of A is null, and that the remaining columns constitute an anti-diagonal matrix since `i` and `j` are used in reverse order in the subscript expressions.

However this representation cannot express replication of elements on a dimension of the template, as allowed by HPF. The former relation is generalized by adding a projection matrix R (for replication) which selects the dimensions of the template which are actually aligned. Thus the following example:

```
align A(i) with T(2*i-1,*)
```

is translated into

$$Rt = Aa + s_0 \tag{2}$$

where $R = \begin{pmatrix} 1 & 0 \end{pmatrix}$, $t = \begin{pmatrix} t_1 \\ t_2 \end{pmatrix}$, $A = \begin{pmatrix} 2 \end{pmatrix}$, $a = \begin{pmatrix} a_1 \end{pmatrix}$, $s_0 = \begin{pmatrix} -1 \end{pmatrix}$

The same formalism can also deal with a chain of alignment relations (`A` is aligned to `B` which is aligned to `C...`) by composing the alignments. To summarize, a "`*`" in an array reference induces a zero column in A and a "`*`" in a template reference a zero column in R. When no replication is specified, R is the identity matrix. The number of rows of R and A corresponds to the number of template dimensions that are *actually* aligned, thus leading to equations linking template and array dimensions. Template dimensions with no matching array dimensions are removed through R.

2.4 Distribution

Once optional alignments are defined in HPF, objects or aligned objects can be distributed on Cartesian processor arrangements, declared like arrays and templates, and represented by the inequality $0 \leq p < P.1$. Each dimension can be distributed by block, or in a cyclic way, on the processor set or even collapsed on the processors. Let us first consider the Examples 1 and 2 in Figures 3 and 5:

Example 1

In Example 1, the distributions of the templates creates a link between the templates elements t and the processors p, so that each processor owns some of the template elements. These links can be translated into linear formulae with the addition of variables. For the `block` distribution of template `T`, assuming the local offset ℓ within the size 20 block, $0 \leq \ell < 20$, then the formula simply is:

$$t = 20p + \ell$$

For a fixed processor p and the allowed range of offsets within a block ℓ, the formula gives the corresponding template elements that are mapped on that processor. There is no constant in the equality due to the assumption that template and processor dimensions start from 0. For the `cyclic` distribution of template `T'`, a cycle variable c counts the number of cycles over the processors for a given template element. Thus the formula is:

$$t' = 5c + p$$

The cycle number c generates an initial offset on the template for each cycle over the processors. Then the p translation associates the processor's owned template element for that cycle. The general `cyclic(`n`)` multi-dimensional case necessitates both cycle c and local offset ℓ variable vectors, and can be formulated with a matrix expression to deal with all dimensions at once.

General case

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. P is a square matrix with the size of the processor dimensions on the diagonal. Such a distribution is not linear according to its definition [36] but may be written as a linear relation between the processor coordinate p, the template coordinate t and two additional variables, ℓ and c:

$$\Pi t = CPc + Cp + \ell \tag{3}$$

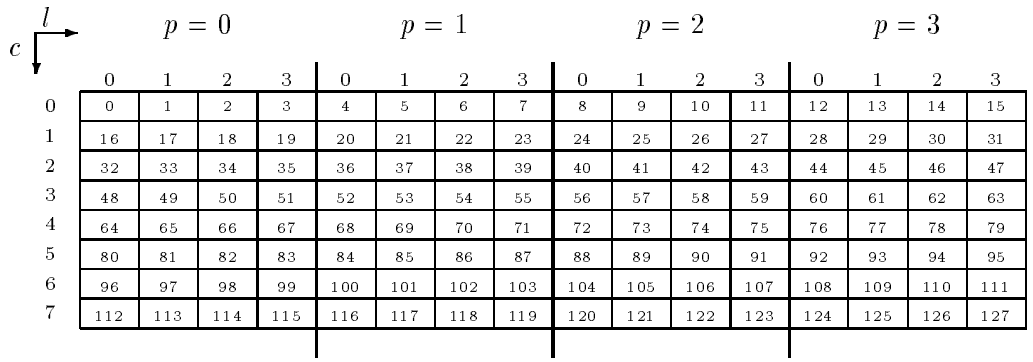
Vector ℓ represents the offset within one block in one processor and vector c represents the number of wraparound (the sizes of which are CP) that must be performed to allocate blocks cyclically on processors as shown on Figure 4 for the example in Figure 7.

The projection matrix Π is needed when some dimensions are collapsed on processors, that means that all the elements on the dimension are on a same processor. These dimensions are thus orthogonal to the processor parallelism and can be eliminated. The usual modulo and integer division operators dealing with the block size are replaced by predefined constraints on p and additional constraints on ℓ. Vector c is implicitly constrained by array declarations and/or (depending on the replication) by the `TEMPLATE` declaration.

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 (3) can be simplified by zeroing the c coordinate for this dimension. This additional equation reduces the number of free variables and, hence, removes a loop in the generated code. `CYCLIC` is equivalent to `CYCLIC(1)` which is translated into an additional equation: $\ell = 0$. `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. Since block distributions do not cycle around the processors, $c = 0$ can be added.

The variables are bounded by:

$$0 \leq p < P.1, \quad 0 \leq t < T.1, \quad 0 \leq \ell < C.1 \tag{4}$$



c \ l	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
2	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
3	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
4	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
5	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
6	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
7	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127

```
chpf$ processors P(0:3)
chpf$ template T(0:127)
chpf$ distribute T(cyclic(4)) onto P
```

Figure 4: Example of a template distribution (from [25]).

```
template T(0:99,0:99,0:99)
processors P(0:9,0:9)
distribute T(*,cyclic(4),block(13)) onto P
```

Figure 5: Example 2

Example 2

Let us consider the second, more general example, in Figure 5. These directives can be translated into the following matrix form:

$$\begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} t = \begin{pmatrix} 40 & 0 \\ 0 & 130 \end{pmatrix} c + \begin{pmatrix} 4 & 0 \\ 0 & 13 \end{pmatrix} p + \ell, \ \begin{pmatrix} 0 \\ 0 \end{pmatrix} \leq \ell < \begin{pmatrix} 4 \\ 13 \end{pmatrix}$$

where

$$t = \begin{pmatrix} t_1 \\ t_2 \\ t_3 \end{pmatrix}, \ p = \begin{pmatrix} p_1 \\ p_2 \end{pmatrix}, \ c = \begin{pmatrix} c_1 \\ c_2 \end{pmatrix}, \ \ell = \begin{pmatrix} \ell_1 \\ \ell_2 \end{pmatrix}, \ c_2 = 0$$

and

$$\Pi = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \ C = \begin{pmatrix} 4 & 0 \\ 0 & 13 \end{pmatrix}, \ P = \begin{pmatrix} 10 & 0 \\ 0 & 10 \end{pmatrix}$$

In this example, the distribution is of type `BLOCK` in the last dimension, thus the added $c_2 = 0$. 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 is collapsed on the processors. Thus it can be discarded from the equations. It means that it can be assumed that $\Pi = I$ in Equation (3) if these useless dimensions are removed by a normalization process.

2.5 Iteration domains and subscript functions

Although iteration domains and subscript functions do not concern directly the placement and the distribution of the data in HPF, they must be considered for the code generation. Since loops are assumed `INDEPENDENT` with affine bounds, they can be represented by a set of inequalities on the iteration vectors i: The original enumeration order is not kept by this representation, but the independence of the loop means that the result of the computation does not depend on this very order. The array reference links the iteration vector to the array dimensions.

```
INDEPENDENT(i2,i1)                    ! normalized (stride 1) version
do j=1, n1, 3                         do i2=0, (n1-1)/3
   do i1=j+2, n2+1                       do i1=3*i2+3, n2+1
      A(2*i1+1, n1-j+i1) = ...              A(2*i1+1, n1-3*i2+i1-1) = ...
   enddo                                 enddo
enddo                                 enddo
```

Figure 6: Loop example

Let us consider for instance the example in Figure 6. The iteration domain can be translated into a parametric (n_1 and n_2 may not be known at compile time) form, where the constant vector is a function over the parameters. Thus the set of constraints for the loop nest iterations, with the additional change of variable $j = 3i_2 + 1$ to normalize the loop [87], is:

$$\begin{pmatrix} 0 & -1 \\ 0 & 3 \\ 1 & 0 \\ -1 & 3 \end{pmatrix} \begin{pmatrix} i_1 \\ i_2 \end{pmatrix} \leq \begin{pmatrix} 0 \\ n_1 - 1 \\ n_2 + 1 \\ -3 \end{pmatrix}$$

Moreover the array reference can also be translated into a set of linear equalities:

$$a = \begin{pmatrix} a_1 \\ a_2 \end{pmatrix} = \begin{pmatrix} 2 & 0 \\ 1 & -3 \end{pmatrix} \begin{pmatrix} i_1 \\ i_2 \end{pmatrix} + \begin{pmatrix} 1 \\ n_1 - 1 \end{pmatrix}$$

In the general case, a set of constraints is derived for the iteration domain of the loop nest (Equation (5), where n stands for a vector of parameters for the loop nest) and for the array references (Equation (6)).

$$Li \leq b_0(n) \tag{5}$$

$$a = Si + a_0(n) \tag{6}$$

2.6 Putting it all together

Let us now consider the example in Figure 7. According to the previous Sections, all the declaration of distributions, alignments, arrays, processors and templates, iterations and references can be rewritten as

```
      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)) onto P
      A(0:U:3) = ...
```

Figure 7: Example in CHATTERJEE *et al.* [25]

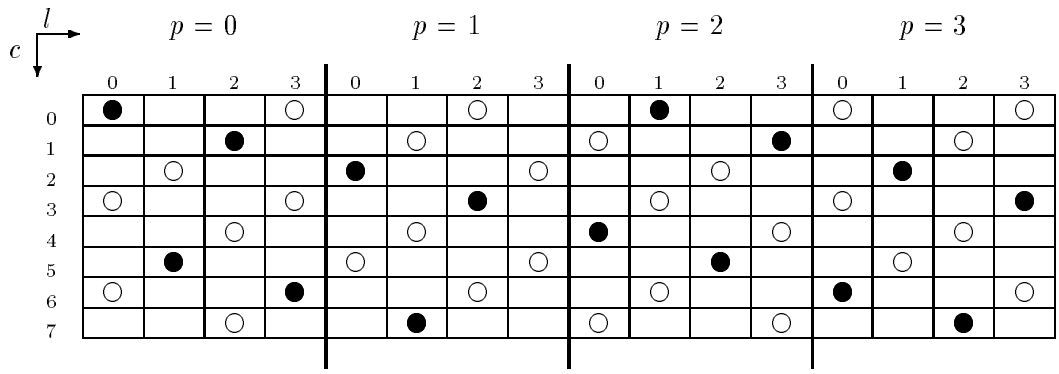


Figure 8: Example of an array mapping and writing (from [25]).

$$
\begin{aligned}
0 &\leq \ell < C.1,\\
0 &\leq a < D.1,\\
0 &\leq p < P.1,\\
0 &\leq t < T.1,\\
Rt &= Aa + s_0,\\
t &= CPc + Cp + \ell,\\
Li &\leq b_0(n),\\
a &= Si + a_0(n)
\end{aligned}
$$

where

$$D = (43),\ T = (128),\ A = (3),\ s_0 = (0),\ R = (1),\ C = (4),\ P = (4),$$

$$L = \begin{pmatrix} 3 \\ -3 \end{pmatrix},\ b_0(n) = \begin{pmatrix} \mathtt{U} \\ 0 \end{pmatrix},\ S = (3),\ a_0(n) = (0),$$

The array mapping of `A` is represented by the "$\circ$" or "$\bullet$" on Figure 8 and the written elements (if `U` = 42) of `A` with "$\bullet$". As can be seen on this 2D representation of a 3D space, the mapping and the access are done according to a regular pattern, namely a lattice. This is exploited to translate the compilation problem in a linear algebra framework.

3 Overview of the compilation scheme

The compilation scheme is based on the HPF declarations, the *owner computes 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 left hand side `X` with some right hand side expression f on variables `Y`, `Z`,...(Figure 9). The loop is normalized to use unit steps as explained in the previous section. Since the loop is independent, $S_{\mathtt{X}}$ must be injective on the iteration domain for an independent parallel loop to be deterministic. The n models external runtime parameters that may be involved in some subexpressions, namely the loop bounds and the reference shifts.

The HPF standard does not specify the relationship between the HPF processors and the actual physical processors. Thus, efficiently compiling loop nests involving arrays distributed onto different processor sets without further specification is beyond the scope of this paper. But if the processor virtualization process can be represented by affine functions, which would be probably the case, we can encompass these different processor sets in our framework by remaining on the physical processors. In the following we assume for clarity that the interfering distributed arrays in a loop nest are distributed on the same processors set.

`INDEPENDENT(`i`)`
`forall(`$Li \leq b_0(n)$`)`
`  X(`$S_{\mathtt{X}} i + a_{\mathtt{X}0}(n)$`) = f(Y(`$S_{\mathtt{Y}} i + a_{\mathtt{Y}0}(n)$`), Z(`$S_{\mathtt{Z}} i + a_{\mathtt{Z}0}(n)$`))`

Figure 9: Generic loop

3.1 Own Set

The subset of `X` that is allocated on processor p according to HPF directives must be mapped onto an array of the output program. 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 tradeoff 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:

$$\begin{aligned} Own_{\mathtt{X}}(p) = \{a \mid \exists t, \exists c, \exists \ell,\ s.t. \quad & R_{\mathtt{X}} t = A_{\mathtt{X}} a + s_{\mathtt{X}0} \\ \wedge\ & t = C_{\mathtt{X}} P c + C_{\mathtt{X}} p + \ell_{\mathtt{X}} \\ \wedge\ & 0 \leq a < D_{\mathtt{X}}.1 \\ \wedge\ & 0 \leq p < P.1 \\ \wedge\ & 0 \leq \ell < C_{\mathtt{X}}.1 \\ \wedge\ & 0 \leq t < T_{\mathtt{X}}.1\} \end{aligned}$$

Subset $Own_{\mathtt{X}}(p)$ can be mapped onto a Fortran array by projection on c and ℓ, or on any equivalent set of variables, *i.e.* up to an injective mapping. Although Own is not a dense polyhedron as defined here, it can be seen as such in the higher dimensional (a, c, ℓ) space. Thus this view is used in the following, although our interest it just to

represent the array elements a. Note that for a given distributed dimension a_i there is one and only one corresponding (p_j, c_j, ℓ_j) triplet.

3.2 Compute set

Using the *owner computes rule*, the set of iterations local to p, $Compute(p)$, is directly derived from the previous set, $Own_{\mathtt{X}}(p)$. The iterations to be computed by a given processor are those of the loop nest for which the *lhs* are owned by the processor:

$$Compute(p) = \{i \mid S_{\mathtt{X}} i + a_{\mathtt{X}0}(n) \in Own_{\mathtt{X}}(p) \wedge Li \leq b_0(n)\}$$

Note that $Compute(p)$ and $Own_{\mathtt{X}}(p)$ are equivalent sets if the reference is direct ($S_{\mathtt{X}} = I$, $a_{\mathtt{X}0} = 0$) and if the iteration and array spaces conform. This is used in Section 4.3 to study the allocation of local arrays as a special case of local iterations. According to this definition of *Compute*, when `X` is replicated, its new values are computed on all processors having a copy. Depending on a tradeoff between communication and computation speed, the optimal choice may be to broadcast the data computed once in parallel rather than computing each value locally.

3.3 View set

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

$$View_{\mathtt{Y}}(p) = \{a \mid \exists i \in Compute(p) \; s.t. \;\; a = S_{\mathtt{Y}} i + a_{\mathtt{Y}0}(n)\}$$

Note that unlike $S_{\mathtt{X}}$, matrix $S_{\mathtt{Y}}$ is not constrained and cannot always be inverted.

If the intersection of this set with $Own_{\mathtt{Y}}(p)$ is non-empty, some elements needed by processor p for the computation are fortunately on the same processor p. Then, in order to simplify the access to $View_{\mathtt{Y}}(p)$ without having to care about dealing differently with remote and local elements in the computation part, the local copy $\mathtt{Y}'$ may be enlarged so as to include its neighborhood, including $View_{\mathtt{Y}}(p)$. The neighborhood is usually considered not too large when it is bounded by a numerical *small* constant, which is typically the case if `X` and `Y` are identically aligned and accessed at least on one dimension up to a translation, such as encountered in numerical finite difference schemes. This optimization is known as *overlap analysis*[38]. Once remote values are copied into the overlapping $\mathtt{Y}'$, all elements of $View_{\mathtt{Y}}(p)$ can be accessed uniformly in $\mathtt{Y}'$ with no overhead. However such an allocation extension may be considered as very rough and expensive in some cases (*e.g.* a matrix multiplication), because of the induced memory consumption. A reusable temporary array might be preferred, and locally available array elements must be copied. Such tradeoffs to be considered in the decision process are not discussed in this paper, but present the techniques for implementing both solutions.

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 connexion in the dependence graph [5]. 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 area in the array and two different local copies should be allocated to reduce the total amount of memory allocated: a unique copy would 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 distributed array, its local elements must be taken into account as a special reference and be accessed with (p, c, ℓ) instead of i.

The definition of *View* is thus altered to take into account array *regions*. These regions are the result of a precise program analysis which is presented in [80, 50, 9, 11, 10, 33, 32, 31]. An array region is a set of array elements described by equalities and inequalities defining a convex polyhedron. This polyhedron may be parameterized by program variables. Each array dimension is described by a variable. The equations due to subscript expression $S_{\mathtt{Y}}$ are replaced by the array region, a parametric polyhedral subset of Y which can be automatically computed. For instance, the set of references to Y performed in the loop body of:

```
do i2 = 1, N
   do i1 = 1, N
      X(i1,i2) = Y(i1,i2) + Y(i1-1,i2) + Y(i1,i2-1)
   enddo
enddo
```

is automatically summarized by the parametric region on $a_{\mathtt{Y}} = (y_1, y_2)$ for the Y reference in the loop body of the example above as:

$$\mathtt{Y}(y_1, y_2) : \{(y_1, y_2) \mid y_1 \leq i_1 \ \wedge \ y_2 \leq i_2 \ \wedge \ i_1 + i_2 \leq y_1 + y_2 + 1\}$$

Subset $View_{\mathtt{Y}}$ 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 with regular holes [4]. This might be useful for red-black SOR.

As mentioned before, if Y is a distributed array and its region includes local elements, it might be desired to simply extend the local allocation so as to simplify the addressing and to allow a uniform way of accessing these array elements, for a given processor $p_{\mathtt{Y}}$ and cycle $c_{\mathtt{Y}}$. The required extension is computed simply by mixing the distribution and alignment *equations* to the $View_{\mathtt{Y}}$ description. For the example above, assuming that X and Y are aligned and thus distributed the same way, the overlap is expressed on the block offset $\ell_{\mathtt{Y}}$ with:

$$\begin{aligned} R_{\mathtt{Y}} t &= A_{\mathtt{Y}} a_{\mathtt{Y}} + s_{\mathtt{Y}_0} \\ t &= CPc_{\mathtt{Y}} + Cp_{\mathtt{Y}} + \ell_{\mathtt{Y}} \\ a_{\mathtt{Y}} &\in View_{\mathtt{Y}}(p) \end{aligned}$$

Vectors $c_{\mathtt{Y}}$ and $p_{\mathtt{Y}}$ are constrained as usual by the loop, the processor declaration and the template declaration of X but vector $\ell_{\mathtt{Y}}$ is no more constrained in the block size (C). It is indirectly constrained by $a_{\mathtt{Y}}$ and the Y region. This leads to a $-1 \leq \ell_{\mathtt{Y}}$ lower bound instead of the initial $0 \leq \ell_{\mathtt{Y}}$, expressing the size 1 overlap on the left.

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:

$$\begin{aligned} Send_{\mathtt{Y}}(p,p') &= Own_{\mathtt{Y}}(p) \cap View_{\mathtt{Y}}(p') \\ Receive_{\mathtt{Y}}(p,p') &= View_{\mathtt{Y}}(p) \cap Own_{\mathtt{Y}}(p') \end{aligned}$$

These two sets cannot always be used to derive the data exchange code. Firstly, a processor 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[1]. Secondly, replicated arrays would lead to useless communication: each owner would try to send data to each viewer. An additional constraint should be added to restrict the sets of communications to needed ones. Different techniques can be used to address this issue: (1) replication allows broadcasts and/or load-balance, what is simply translated into linear constraints as described in [29]. (2) The affectation of owners to viewers can also be optimized in order to reduce the distance between communicating processors. For instance, the cost function could be the minimal Manhattan distance[2] between p and p' or the lexicographically minimal vector[3] $p'-p$ if the interconnection network is a $|p|$-dimension grid[4]. 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 [35].

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 $\mathtt{Y}'$, the local part of $\mathtt{Y}$, the allocation and addressing of which are discussed in Section 4.3. *Send* and *Receive* are polyhedral sets and algorithms in [4] 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 the local processor p, is shown in Figure 10. `U` represents nay local array generated according to the dependence graph. 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.

This code presented in Figure 10 is quite different from the one suggested in [58, 57]. Firstly, the set of local iterations $Compute(p)$ is no longer expressed in the i frame but in the (c,ℓ) frame: There is no *actual* need to know about the initial index values in

[1] 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, because the loop over p' is an outermost loop.

[2] The Manhattan norm of a vector is the sum of the absolute values of its coordinates, *i.e.* the l_1 norm.

[3] *i.e.* the closest on the first dimension, and then the closest on the second dimension, and so on, dimension per dimension, till one processor is determined.

[4] For a vector v, let $|v|$ denotes its dimension.

```
real X'((c,ℓ) ∈ Own_X(p)),
     Y'((c,ℓ) ∈ View_Y(p)),
     Z'((c,ℓ) ∈ View_Z(p))
forall(U ∈ {Y',Z',...})
      forall((p,p'),p ≠ p',Send_U(p,p') ≠ ∅ )
            forall((ℓ,c) ∈ Send_U(p,p'))
                  send(p', U(ℓ,c))
forall(U ∈ {Y',Z',...})
      forall((p,p'),p ≠ p',Receive_U(p,p') ≠ ∅ )
            forall((ℓ,c) ∈ Receive_U(p,p'))
                  U(ℓ,c) = receive(p')
if Compute(p) ≠ ∅
   forall((ℓ,c) ∈ Compute(p))
         X'(S_X'(ℓ,c)) = f(Y'(S_Y'(ℓ,c)),
                           Z'(S_Z'(ℓ,c)),...)
```

Figure 10: The output SPMD code.

the original code. Secondly, no difference is made between local and non-local iteration computations, at the price of possible communication and computation overlaps, but at the benefice of an homogeneous addressing: Only one reference and addressing should be generated for each original reference in the computation expression, thus avoiding costly code duplications or runtime guards to deal with each reference of an expression that may or may not be locally available at every iteration. The temporary array management and addressing issues are discussed further in the next sections. Thirdly, non-local iterations computed as a set difference is not likely to produce an easily manageable set for code generation, since it should generally be non convex. Thirdly, the parametric description of the communications allows to enumerate a subset of active processors.

Additional changes of basis or changes of coordinates must be performed to reduce the space allocated in local memory and to generate a minimal number of more efficient scanning 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 10 is still far from Fortran. Additional changes of coordinates are needed to generate proper Fortran declarations and loops.

4.1 Enumeration of iterations

A general method to enumerate local iterations is described below. It is based on (1) solving HPF and loop Equations, both equalities (2, 3, 6) and inequalities (4, 5), on (2) searching a *good* lattice basis to scan the local iterations in an appropriate order using p as parameter since each processor knows its own identity and (3) on using linear transformations to switch from the user visible frame to the local frame.

In this section, a change of frame is computed based on the available equalities to find a dense polyhedron that can be scanned efficiently with standard techniques. The change of frame computation uses two HERMITE forms to preserve the order of the (ℓ, c) variables which are related to the allocation scheme, for benefiting from cache effects.

Simplifying formalism

The subscript function S and loop bounds are affine. All object declarations are normalized with 0 as lower bound. The HPF array mapping is also normalized: $\Pi = I$. Under these assumptions and according to Section 2, HPF declarations and Fortran references are represented by the following set of equations:

$$\left.\begin{array}{rrcl} \text{alignment (2)} & Rt & = & Aa + s_0 \\ \text{distribution (3)} & t & = & CPc + Cp + \ell \\ \text{affine reference (6)} & a & = & Si + a_0(n) \end{array}\right\} \tag{7}$$

where p is the processor vector, ℓ the local offset vector, c the cycle vector, a the accessed element i the iteration and n the parameters.

Problem description

Let x be the following set of variables:

$$x = (\ell, c, a, t, i) \tag{8}$$

They are needed to describe array references and iterations. The order of the ℓ and c variables chosen here should be reflected by the local allocation in order to benefit from cache effects. This order suits best cyclic distributions, as discussed in [83], because there should be more cycles (along c) than block elements (along ℓ) in such cases. Putting the block offset after the cycle number for a given dimension would lead to larger inner loops for *more block* distributions. Note that the derivations discussed in this section are independent of this order. Equation (7) can be rewritten as

$$Fx = f_0(n, p) \tag{9}$$

with

$$F = \begin{pmatrix} 0 & 0 & A & -R & 0 \\ I & CP & 0 & -I & 0 \\ 0 & 0 & -I & 0 & S \end{pmatrix} \text{ and } f_0(n,p) = \begin{pmatrix} s_0 \\ -Cp \\ a_0(n) \end{pmatrix} \quad (10)$$

For instance in the Chatterjee *et al.* example in Figure 7, Equation 10 is:

$$\begin{pmatrix} 0 & 0 & 1 & -1 & 0 \\ 1 & 16 & 0 & -1 & 0 \\ 0 & 0 & -1 & 0 & 3 \end{pmatrix} \begin{pmatrix} \ell \\ c \\ a \\ t \\ i \end{pmatrix} = \begin{pmatrix} 0 \\ -4p \\ 0 \end{pmatrix}$$

The set of interest is the lattice $\mathcal{F} = \{x | Fx = f_0(n,p)\}$. The goal of this section is to find a parametric solution for each component of x. This allows each processor p to scan its part of $\mathcal{F}$ with minimal control overhead.

It is important to note that F is of full row-rank. The rows contain distinct variables that insure their independence one from the other: Firstly, the alignment equations are composed of independent equalities (they differ from a variables); This is also the case for the distribution and reference equations because of the ℓ and t variables. Secondly the equalities composing the distribution equations are the only to contain ℓ variables, thus they are independent from the alignment and reference. Thirdly, the alignment and reference equations are separated by the template variables thru R. Since all equalities are independent, F is a full row-rank. Note that HPF restrictions about alignment are not exploited.

If additional equalities are taken into account, such as those arising from simple block ($c_i = 0$) or cyclic ($\ell_i = 0$) distributions, they can be used to remove the variables from the equation and do not actually change this property. Other degenerated cases may arise (for instance, there is only one processor on a given dimension...), but they are not actually discussed here. Our general scheme can directly take advantage of such extra information to optimize the generated code by including it as additional equalities and inequalities.

Parametric solution of equations

Lattice $\mathcal{F}$ is implicitly defined but a parametric definition is needed to enumerate its elements for a given processor p. There are two kinds of parameters that must be set apart. First the constant unknown parameters n and local processor id p the value of which are known at runtime on each processor. Second, the parameters we are interested in, that have to be enumerated or instantiated on each processor to scan the integer solutions to the HPF equations, namely the variables in vector x.

An HERMITE form [76] of integer matrix F is used to find the parameters. This form associates to F (an $n \times m$ matrix with $m \geq n$) three matrices H, P and Q, such that $H = PFQ$. P is a permutation (a square $n \times n$ matrix), H an $n \times m$ lower triangular integer matrix and Q an $m \times m$ unimodular change of basis. Since F is of full row-rank, no permutation is needed: $P = I$ and $H = FQ$ (a). By definition, H is a lower triangular matrix, and thus can be decomposed as $H = (H_L\ 0)$, where H_L is an $n \times n$ integer triangular square matrix. We know that $|H_L| \in \{-1, 1\}$. Indeed, H_L is of full rank (as F) and the column combinations performed by the HERMITE form

computation puts unit coefficients on H_L diagonal. This is insured since independent unit coefficients appear in each row of F. Thus H_L is an integer triangular unimodular matrix, and has an integral inverse.

Now we can use Q as a change of basis between new variables v and x, with $v = Q^{-1}x$ (b). Vector v can also be decomposed like H in two components: $v = (v_0, v')$, where $|v_0|$ is the rank of H. Using (a) and (b) Equation (9) can be rewritten as:

$$Fx = FQQ^{-1}x = Hv = (H_L 0)(v_0, v') = H_L v_0 = f_0(n,p)$$

v_0 is a parametric solution at the origin which depends of the runtime value of the n and p parameters. Thus we have $v_0(n,p) = H_L^{-1} f_0(n,p)$. By construction, H does not constrain v' and Q can also be decomposed like v as $Q = (Q_0\ F')$. Lattice $\mathcal{F}$ can be expressed in a parametric linear way:

$$x = Qv = (Q_0\ F')(v_0(n,p), v') = Q_0 v_0(n,p) + F'v'$$

and with $x_0(n,p) = Q_0 v_0(n,p) = Q_0 H_L^{-1} f_0(n,p)$:

$$\mathcal{F} = \{x | \exists v'\ \ s.t.\ \ x = x_0(n,p) + F'v'\} \tag{11}$$

We have switched from an implicit (9) description of a lattice on x to an explicit (11) one through F' and v'. Note that F' is of full column-rank.

Cache-friendly order

As mentionned earlier, the allocation is based somehow on the (ℓ, c) variables. The order used for the enumeration should reflect as much as possible the one used for the allocation, so as to benefit from memory locality. Equation (11) is a parametric definition of x . However the new variables v' are not necessarily ordered as the variables we are interested in. The aim of this paragraph is to reorder the variables as desired, by computing a new transformation based on the HERMITE form of F'. Let $H' = P'F'Q'$ be this form. Let Q' define a new basis:

$$u = Q'^{-1}v' \tag{12}$$

$$x - x_0(n,p) = F'v' = P'^{-1}H'Q^{-1}v' = P'^{-1}H'u \tag{13}$$

If $P' = I$, the new generating system of $\mathcal{F}$ is based on a triangular transformation between x and u (13). Since H' is a lower triangular integer matrix, the variables of u and of x simply correspond one to the other. Knowing this correspondance allows to order the variable u components to preserve locality of accesses. If $P' \neq I$, the variables are shuffled and some cache effect may be lost. However we have never encountered such an example.

Code generation

Variable u defining x in lattice $\mathcal{F}$ and polyhedron $\mathcal{K}$ are easy to scan with `DO` loops. The constraints defining polyhedron $\mathcal{K}$ are coming from declarations, HPF directives and normalized loop bounds. They are:

$$0 \leq \ell < C.1,\ \ 0 \leq t < T.1,\ \ 0 \leq a < D.1,\ \ Li \leq b_0(n)$$

Let $Kx \leq k_0(n)$ be these constraints on x. Using (13) the constraints on u can be written $K(x_0(n,p) + P'^{-1}H'u) \leq k_0(n)$, that is $K'u \leq k'_0(n,p)$, where $K' = KP'^{-a}H'$ and $k'_0(n,p) = k_0(n) - Kx_0(n,p)$.

Algorithms presented in [4] or others [35, 30, 53, 2, 24, 23, 62, 59, 54, 86] 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. Otherwise, the quality of the control overhead depends on the accuracy of integer projections [73] but the correctness does not.

Correctness

The correctness of this enumeration scheme stems from (1) the exact solution of integer equations using the HERMITE form to generate a parametric enumeration, from (2) the unimodularity of the transformation used to obtain a *triangular* enumeration, and from (3) the independent parallelism of the loop which allows any enumeration order.

4.2 Symbolic resolution

The previous method can be applied in a symbolic way, if the dimensions are not coupled and thus can be dealt with independently, as array sections in [25, 43, 78]. Equations (7) then become for a given dimension:

$$\left.\begin{array}{rl} \text{alignment} & t = \alpha a + t_0 \\ \text{distribution} & t = \pi\gamma c + \gamma p + \ell \\ \text{affine reference} & a = \sigma i + a_0 \end{array}\right\} \tag{14}$$

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

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

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

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 two rightmost columns of Q noted F' ($H' = P'F'Q'$) is such that $x - x_0 = H'u$ with:

$$H' = \begin{pmatrix} -g & 0 \\ \mu & \frac{\alpha\sigma}{g} \\ \omega & \frac{\pi\gamma}{g} \end{pmatrix},\ x_0 = \begin{pmatrix} \alpha a_0 + t_0 - \gamma p \\ 0 \\ 0 \end{pmatrix},\ P' = I \text{ and } Q' = \begin{pmatrix} \mu & \frac{\alpha\sigma}{g} \\ \omega & \frac{\pi\gamma}{g} \end{pmatrix}$$

This links the two unconstrained variables u to the elements x of the original lattice $\mathcal{F}$. Variables a and t can be retrieved using Equations (14).

The translation of constraints 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

$$
\begin{array}{l}
\forall p \in [0 \ldots \pi - 1] \\
\texttt{do } u_1 = \frac{\alpha a_0 + t_0 - \gamma p + \gamma + g - 2}{g}, \frac{\alpha a_0 + t_0 - \gamma p}{g} \\
\quad \texttt{do } u_2 = \frac{-\omega u_1 + \frac{\pi\gamma}{g} - 1}{\frac{\pi\gamma}{g}}, \frac{\beta - 1 - \omega u_1}{\frac{\pi\gamma}{g}} \\
\quad\quad x = H'u + x_0(p)
\end{array}
$$

Figure 11: Generated code

are rectangular[5] and using the constraints in K:

$$0 \leq \ell < \gamma, \quad 0 \leq a < s, \quad 0 \leq i < \beta$$

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

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

and can be translated as constraints on u:

$$\begin{pmatrix} g & 0 \\ -g & 0 \\ -\omega & -\frac{\pi\gamma}{g} \\ \omega & \frac{\pi\gamma}{g} \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \end{pmatrix} \leq \begin{pmatrix} \alpha a_0 + t_0 - \gamma p \\ -\alpha a_0 - t_0 + \gamma p + \gamma - 1 \\ 0 \\ \beta - 1 \end{pmatrix}$$

The resulting generic SPMD code for an array section is shown in Figure 11. As expected, the loop nest is parameterized by p, the processor identity. Integer divisions with positive remainders are used in the loop bounds.

4.3 HPF array allocation

The previous two sections can also be used to allocate local parts of HPF distributed arrays. A loop nest referencing a whole array through an identity subscript function ($S = I, a_0 = 0$) serves as a basis for the allocation. The dense polyhedron obtained by the changes of bases for the enumeration purpose can be used to store the required elements, since 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 based on Cartesian sets and are not as general as Fortran loop bounds. General methods have been proposed to allocate polyhedra [85]. For or particular case, it is possible to add another change of frame to fit Fortran array declaration constraints better and to reduce the amount of allocated memory at the expense of the access function complexity.

[5] This assumption is, of course, not necessary for the general algorithm described in Section 4.1, but met by array sections.

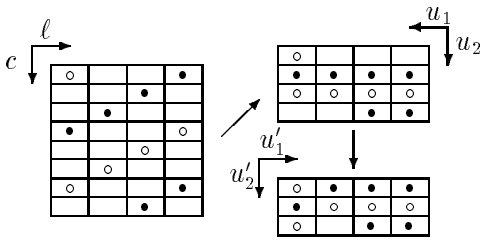


Figure 12: Packing of elements

The local array elements are packed onto local memories dimension by dimension. The geometric intuition of the packing scheme for the [25] example is shown in Figure 12. The basic idea is to remove the regular holes due to the alignment stride by allocating the dense u space on each processor. The "$\circ$" and "$\bullet$" are just used to support the geometrical intuition of the change of frame. 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 transformation from x to u. The third one is the final packed form, which could be used if no space must be wasted, but at the expense of a complex access function.

An array dimension can be collapsed or distributed. If the dimension is collapsed 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. Of course, a dual method is to pack first along the rows, then removing the empty ones. This last method is less efficient for the example on Figure 12 since it would require 16 (8×2) elements instead of 12 (3×4). The two packing schemes can be chosen according to this criterion. Other issues of interest are the induced effects for the cache behavior and the enumeration costs. Formulae are derived below to perform these packing schemes.

Packing of the symbolic resolution

Let us 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 the triangular part of H', i.e. its first rows. If $H'' = (I\,0)H'$ is the selected sub-matrix, we have $x' - x'_0 = H''u$, *i.e.*:

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

Variables σ and a_0 were substituted by their values in the initial definition of H'.

Variables (u_1, u_2) could be used to define an efficient packing, since holes are removed by the first change of basis (Figure 12). In order to match simply and closely the ℓ, c space, the sign of u_1 (linked to ℓ) can be changed, and the vector should be shifted so that the first local array element is mapped near $(0, 0)$.

$$0 \leq p \leq \pi - 1,\ 0 \leq \ell \leq \gamma - 1$$
$$\left\lfloor \frac{t_0}{\gamma\pi} \right\rfloor \leq c \leq \left\lfloor \frac{\alpha(s-1)+t_0}{\gamma\pi} \right\rfloor$$

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

Figure 13: Local new declaration

Some space may be wasted at the beginning and end of the allocated space. For a contrived example (with very few cycles) the wasted space can represent an arbitrary amount on each dimension. Let us assume that two third of the space is wasted for a given dimension. Thus the memory actually used for an array with 3 of these dimensions is 1/27 and 26/27 of the allocated memory is wasted...If such a case occurs, the allocation may be skewed to match a rectangle as closely as possible. This may be done if space is at a premium and if more complex, non-affine access functions are acceptable. The improved and more costly scheme is described in the next section.

Allocation basis

Let M be the positive diagonal integer matrix composed of the absolute value of the diagonal coefficients of H''.

$$u' = \text{alloc}(u) = \lfloor M^{-1}(x' - x'_0) \rfloor = \lfloor M^{-1} H'' u \rfloor \tag{15}$$

M provides the right parameters to perform the proposed packing scheme. To every u a vector u' is associated through Formula 15. This formula introduces an integer division. Let's show why u' is correct and induces a better mapping of array elements on local memories than u. Since H'' is lower triangular, Formula 15 can be rewritten:

$$\forall i \in [1 \ldots |u|], u'_i = \frac{h_{i,i}}{|h_{i,i}|} u_i + \frac{\sum_{j=1}^{i-1} h_{i,j}\, u_j}{|h_{i,i}|} \tag{16}$$

Function alloc(u) is bijective: alloc(u) is injective: if u^a and u^b are different vectors, and i is the first dimension for which they differ, Formula 16 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.

Array declaration

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 M, are used to declare the local part of the HPF array. Figure 13 shows the resulting declaration for the local part of the array, in the

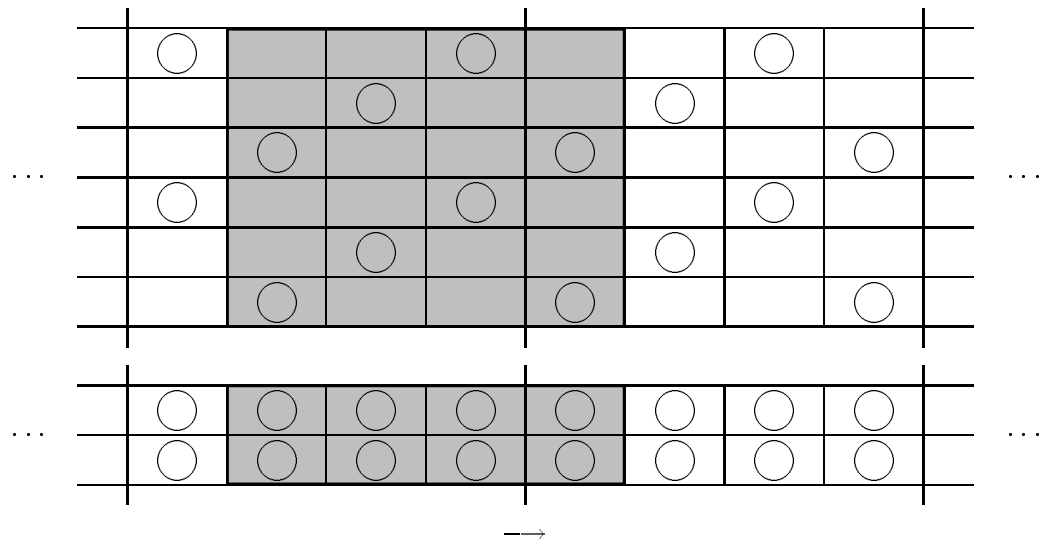


Figure 14: Overlap before and after packing.

general symbolic case, for one dimension of the array. Bounds $\min(c)$ and $\max(c)$ are computed by FOURIER projection.

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 16. The row packing scheme would have been obtained by choosing (c, ℓ) instead of (ℓ, c) for x'. Different choices can be made for each dimension. The access function requires an integer division for the reduced memory allocation. Techniques have been suggested to handle divisions by invariant integers efficiently [41] that could help reduce this cost. Also, because contiguity is preserved, only one division per column is required to compute the base location. These packing schemes define two parameters (u'_1, u'_2) 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.

4.4 Properties

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

According to the access function, the iteration enumeration order and the packing scheme in Figure 11 can be reversed via loop u_2 direction in order that accesses to the local array are contiguous. Thus the local cache and/or prefetch mechanisms, if any, are efficiently used.

The packing scheme is also compatible with overlap analysis techniques [38]. 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

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

$\forall p \in [0 \ldots 3]$
do u'_1 = 0, 3
  $lb_2 = \frac{16p+4u'_1+8}{9}, ub_2 = \frac{16p+4u'_1+7}{9}$
  $lb'_2 = \frac{9lb_2-4u'_1-16p}{3}$
  do u'_2 = lb'_2, $lb'_2 + (ub_2 - lb_2)$
    ...

Figure 15: Optimized code.

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.

For example unsing the HPF array mapping of Figure 7:

```
align B with A
A(1:42) = B(0:41)
```

has a $View_{\mathtt{B}}$ area represented in grey on Figure 14 in the unpacked template space and the local packed array space. The local array B′ can be extended by overlap to contain the grey area. Thus, constraint $0 \leq \ell \leq 3$ in *Own* becomes $-3 \leq \ell \leq 3$, expressing the size 3 overlap on the left.

The generic code proposed in Figure 11 can be greatly improved in many cases. Integer division may be simplified, or performed efficiently with shifts, or even removed by strength reduction. 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 [25], such transformations produce the code shown in Figure 15.

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 in x' 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 inferred from $Compute_L(p)$ and from subscript function $S_{\mathtt{X}}$. For instance, references to array Y in Figure 18 require local copies. Because of Y's distribution, overlap analysis is fine but another algorithm is necessary for other cases.

Let us consider the generic loop in Figure 9 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 $S_{\mathtt{Y}}$ has the same

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

initial system: $f^0 = f$
for $i = 1 \ldots (|w| - 1)$
  $g_i = \gcd(\{f_j^{i-1}, j > i\})$
  $T_i = \{t | t = \sum_{j \leq i} f_j^{i-1} w_j \wedge W w_j \leq 0\}$
  $s_i = \max_{t \in T_i} t - \min_{t \in T_i} t$
  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 16: Heuristic to reduce allocated space

rank as $S_{\mathtt{X}}$, temporaries can be stored as $\mathtt{X}$. If furthermore the access function $S_{\mathtt{X}}$ uses 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 via an HERMITE transformation, then to use this new basis for declaration, or the compressed form to reduce further the allocated space.

The set of local iterations, $Compute(p)$, is now defined by a new basis and new constraints, such that $x - x_0 = P'^{-1} H' u$ (13) and $K' u \leq k'$. Some rows of $P'^{-1} H'$ define iteration vector i, which is part of x (8). 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 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 $v' = \Pi v$ contributes 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:

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

and by substituting i:

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

It is sometime possible to use x' instead of v'. 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 x' 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 patterns 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 suggested.

Each dimension, i.e. component of v'', is treated independently. Let $F^i_{\mathtt{Y}}$ be a line of $F_{\mathtt{Y}}$ and y the corresponding component of v'': $y = fv''$. A change of basis G (not related to the previous alloc operation) 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:

$$f = (\ 16 \quad 3 \quad -4 \quad -16 \quad 0\ )$$

is replaced by:

$$\begin{array}{rcl} f & = & f'G \\ G & = & \begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & -1 & 0 \end{pmatrix} \\ f' & = & (\ 3 \quad 4 \quad 16\ ) \end{array}$$

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 heuristic shown in Figure 16. It reduces the extent by $\frac{s_i}{g_i}$ at each step, if the constraints W show 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.

This scheme is correct if and only if 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 [40, 67].

4.6 Data movements

The relationships between the bases and frames defined in the previous sections are shown in Figure 17. 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 local 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

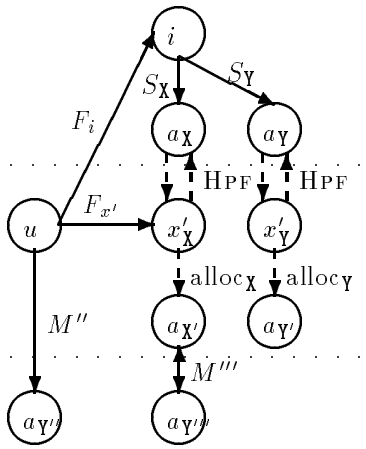


Figure 17: Relationships between frames

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 us consider array X in Figure 17. 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_X, by construction[6] of u. To each a_X corresponds only one a'_X on each processor[7]. As a result, data exchanges controlled 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 Y'' and Y''' are more difficult to initialize because there is no such identity between their allocation function as local part of an HPF array, alloc_Y, and their allocation functions as temporaries, $\text{t_alloc}_{Y''}$ and $\text{t_alloc}_{Y'''}$. However, basis u can still be used.

When the temporary is allocated exactly like the master array, e.g. Y''' and X', any enumeration of elements u in $\mathcal{U}$ enumerates every element a'''_Y once and only once because the input loop is assumed independent. On the receiver side, a'''_Y is directly derived from u. On the sender side, a_Y is computed as $S_Y F_i u$ and a non-affine function is used to obtain x'_Y and a'_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. Otherwise, calls to packing and unpacking routines can be generated.

[6] As explained in Section 4.3, allocation is handled as a special case by choosing the identity for S_X.

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

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 17. 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:

$$u = M^t(MM^t)^{-1}a''_{\mathtt{Y}} \tag{17}$$

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 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 (17) 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 $a''_{\mathtt{Y}}$.

```
      implicit integer (a-z)
      real X(0:24,0:18), Y(0:24,0:18)
!HPF$ template T(0:80,0:18)
!HPF$ align X(I,J) with T(3*I,J)
!HPF$ align Y with X
!HPF$ processors P(0:3,0:2)
!HPF$ distribute T(cyclic(4),block) onto P

      read *, m, n

!HPF$ independent(I,J)
      do I = m, 3*m
         do J = 0, 2*I
            X(2*I,J) = Y(2*I,J) + Y(2*I+1,J) + I - J
         enddo
      enddo

!HPF$ independent(I)
      do I = 0, n
         Y(I,I) = I
      enddo
```

Figure 18: Code example.

5 Examples

The different algorithms presented in the previous section were used to distribute the contrived piece of code of Figure 18, using functions of a linear algebra library developed at École des mines de Paris.

This is an extension of the example in [25] 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 `cyclic(4)` and `block` 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'`. Furthermore, the values of the `I` and `J` are used in the computation, the iteration domain is non rectangular and is parameterized with `m`.

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.

The output code is too long to be printed here. Interesting excerpts are shown in figures 19 and 20 and commented below.

5.1 HPF Declarations

HPF declarations are already normalized. Templates are all used thus $\Pi = I$. Other key parameters are:

$$P = \begin{pmatrix} 4 & 0 \\ 0 & 3 \end{pmatrix}, \; T = \begin{pmatrix} 80 & 0 \\ 0 & 18 \end{pmatrix}, \; C = \begin{pmatrix} 4 & 0 \\ 0 & 7 \end{pmatrix}$$

```
      implicit integer (a-z)
c     Local declaration of X and Y(0:42,0:42):
      real X_hpf(0:6, 0:1, 0:4), Y_hpf(0:6, 0:2, 0:4)

      read *, m, n

c     For each processor P(p1,p2)

c     Send view_Y to compute_X:
      do dest_p1 = 0, 3
       do dest_p2 = 0, 2
        if (dest_p1.ne.p1.or.dest_p2.ne.p2) then
         do u6 = max(-15, -3 - 4*p1), min(0, -4*p1)
          do u7 = max(0, divide(15 - 5*u6, 16)),
     &          min(6, divide(24 - 5*u6, 16))
           do u8 = 7*p2, min(20, 7*p2)
            do u9 = 0, 4
             do u11 = max(0, divide(16*u7 + 5*u6,2), m, (2 + 2*dest_p1 + 8*u9)/3),
     &             min(12, divide(16*u7 + 5*u6,2), 3*m, (2*dest_p1 + 8*u9)/3)
              do u12 = max(0, divide(2*u11 - u8, 21), divide(7*dest_p2 - u8, 21)),
     &              min(0, divide(20 + 7*dest_p2 - u8, 21))
               w3 = 3*u7 + u6
               w5 = (-u6 - 4*p1)/3
               w6 = u8 - 7*p2
               send(P(dest_p1,dest_p2), Y_hpf(w6,w5,w3))
              enddo
             enddo
            enddo
           enddo
          enddo
         enddo
        endif
       enddo
      enddo

c     receive_Y
c     ...

c     compute_X:
      do u4 = 0, 4
       do u6 = max(0, m, (2 + 2*p1 + 8*u4)/3),
     &      min(12, 3*m, (1 + 2*p1 + 8*u4)/3)
        do u7 = max(0, 7*p2), min(18, 2*u6, 6 + 7*p2)
         w3 = u4
         w5 = (6*u6 - 16*u4 - 4*p1)/3
         w6 = u7 - 7*p2
         i = u6
         j = u7
         X_hpf(w6, w5, w3) = Y_hpf(w6, w5, w3)
     &        + i - j + Y_hpf(w6, w5 + 1, w3)
        enddo
       enddo
      enddo
```

Figure 19: Output code (excerpts, part 1).

```
c     compute_Y:
      do u4 = 0, 3
       do u5 = max(0, 7*p2, (2 + 4*p1 + 16*u4)/3),
     &      min(18, 6 + 7*p2, n, (3 + 4*p1 + 16*u4)/3)
         w3 = u4
         w5 = (3*u5 - 16*u4 - 4*p1)/3
         w6 = u5 - 7*p2
         i = u5
         Y_hpf(w6, w5, w3) = i
       enddo
      enddo

      end

c     Integer divide with positive remainder. Assumes j>0.
      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 20: Output code (excerpts, part 2).

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

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.

$$\begin{aligned} Own_{\mathtt{X}}(p) = \{ \quad & (x_1, x_2) \mid t_1 = 3x_1,\ t_2 = x_2 \\ & t_1 = 16c_1 + 4p_1 + \ell_1,\ t_2 = 21c_2 + 7p_2 + \ell_2 \\ & 0 \le x_1 \le 24,\ 0 \le x_2 \le 18,\ 0 \le p_1 \le 3,\ 0 \le p_2 \le 2 \\ & 0 \le \ell_1 \le 3,\ 0 \le \ell_2 \le 6,\ 0 \le t_1 \le 80,\ 0 \le t_2 \le 18 \quad \} \end{aligned}$$

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

$$\begin{aligned} & u_1 = p_1, \quad u_2 = p_2, \quad u_3 = c_1, \quad 3u_5 - 16u_3 - 4u_1 = \ell_1 \\ & u_4 = c_2, \quad u_5 = x_1, \quad u_6 = x_2, \quad u_6 - 21u_4 - 7u_2 = \ell_2 \end{aligned}$$

Then, Frame u is replaced by Frame u' (represented with the w variables in the Fortran code) to avoid non-diagonal terms in the constraints. Frame u' is linked to the user basis x by:

$$u'_1 = p_1, \quad u'_2 = p_2, \quad u'_3 = c_1, \quad u'_4 = c_2, \quad 3u'_5 = \ell_1, \quad u'_6 = \ell_2$$

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

$$\begin{aligned} & 0 \le u'_1 \le 3, \quad 0 \le u'_2 \le 2, \quad 0 \le u'_3 \le 4 \\ & 0 \le u'_4 \le 0, \quad 0 \le u'_5 \le 1, \quad 0 \le u'_6 \le 6 \end{aligned}$$

Coordinates u'_1 and u'_2 are the processor coordinates. The other coordinates, u'_3 to u'_6, are the local array coordinates. Note that since we have a block distribution in the second dimension, $u'_4 = 0$. The Fortran declaration of `X'` is shown in Figure 19 as `X_hpf`. 10×7 elements are allocated while 7×7 would be enough. The efficiency factor may be better in the (p, ℓ, c) basis. It would also be better with less contrived alignments 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 `dest_p1` and `dest_p2` loop is not as tight as it could. This suggests that central and edge processors should be handled differently when overlap is observed. A linearized view of the processors could also help to fix this issue.

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 `u7` can be simplified as a modulo operation. Finally `w5` and `w6` can also be evaluated iteratively.

5.4 Local iterations for the second loop

As for the first 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:

$$\begin{aligned} \mathit{Compute}_{\mathtt{Y}}(p) = \{ \quad & (y_1, y_2) \mid y_1 = i,\ y_2 = i,\ t_1 = 3y_1,\ t_2 = y_2 \\ & t_1 = 16c_1 + 4p_1 + \ell_1,\ t_2 = 21c_2 + 7p_2 + \ell_2 \\ & 0 \le i \le n,\ 0 \le y_1 \le 24,\ 0 \le y_2 \le 18,\ 0 \le p_1 \le 3,\ 0 \le p_2 \le 2 \\ & 0 \le \ell_1 \le 3,\ 0 \le \ell_2 \le 6,\ 0 \le t_1 \le 80,\ 0 \le t_2 \le 18 \quad \} \end{aligned}$$

The loop bounds were retrieved by projection on u. It is probably 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 may as well be hidden in the inner loop bounds. Experiments are needed to find out the best approach.

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 [4]. Further work is needed in this area.

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. Nevertheless, if we can prove the dividend is positive, we can use the Fortran division.

6 Related work

Techniques to generate distributed code from sequential or parallel code using a uniform memory space have been extensively studied since 1988 [22, 70, 89]. Techniques and prototypes have been developed based on Fortran [38, 39, 47, 18, 69, 88, 19, 20], C [8, 63, 6, 60, 7, 61] or others languages [74, 75, 58, 66, 57].

The most obvious, most general and safest technique is called run-time resolution [22, 70, 74]. Each instruction is guarded by a condition which is only 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 [26] 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 [22].

Many optimization techniques have been introduced to handle specific cases. GERNDT introduced overlap analysis in [38] 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 [81] 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.

Recent publications tackle any alignment and distribution but often restrict references to array sections. Each dimension is independent of the others as was assumed in Section 4.2.

In PAALVAST *et al.* [72, 83] a technique based on the resolution of the associated Diophantine equations is presented. Row- and column-wise allocation and addressing schemes are discussed. BENKNER *et al.* [16, 15] present similar techniques.

CHATTERJEE *et al.* [25] developed a finite state machine approach to enumerate local elements. No memory space is wasted and local array elements are ordered by Fortran lexicographic 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 in Figure 11 may be used to compute the FSM. In fact the lower iteration of the innermost loop is computed by the algorithm that builds the FSM. KENNEDY *et al.* [55, 56, 48, 46] and others [79] have suggested improvements to this technique, essentially to compute faster at run-time the automaton transition map. Also multi-dimensional cases need many transition maps to be handled.

Papers by STICHNOTH *et al.* [78, 77] on the one hand and GUPTA *et al.* [43, 44, 52] on the other hand present two similar methods based on closed forms for this 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 virtualization to handle cyclic distributions. Arrays are densely allocated as in [25] and the initial order is preserved but no formulae are given. STICHNOTH uses the dual method for array allocation as in [26], that is blocks are first compressed, and the cycle number is used as a second argument.

In [3, 5] polyhedron-based techniques are presented to generate transfer code for machines with a distributed memory. In [2, 82] advanced analyses are used as an input to a code generation phase for distributed memory machines. Polyhedron scanning techniques are used for generating the code. Two family of techniques have been suggested for that purpose. First, FOURIER elimination based techniques [49, 4, 53, 2, 62, 59, 54, 86], and second, parametric integer programming based methods [35, 30, 24, 23]. In [12], a two-fold HERMITE transformation is also used to remove modulo indexing from a loop nest. First, variables are added to explicit the modulo computation, then the HERMITE computations are used to regenerate simply new loop bounds. While the aim is different, the transformations are very similar to those presented here.

7 Conclusion

The generation of efficient SPMD code from an HPF program is not a simple task and, up to now, many attempts have provided partial solutions and many techniques. A translation of HPF directives in affine constraints, avoiding integer division and modulo, was presented in Section 2 to provide a unique and powerful framework for HPF optimizations. Homogeneous notations are used to succinctly represent user specifications and a normalization step is then applied to reduce the number of objects used in the compilation scheme overviewed in Section 3. New algorithms are presented to (1) enumerate local iterations, to (2) allocate local parts of distributed arrays, to (3) generate send and receive blocks and to (4) allocate temporaries implied by rhs references. They are based on changes of basis and enumeration schemes. It is shown in Section 4 that problem (2) can be casted as special case of problem (1) by using an identity subscript function but that constraints on array bounds are more difficult to efficiently satisfy than loop bounds. Problem (3) is an extension of problem (1): a unique reference to an array is replaced by a set of references and the equation used to express the reference is replaced by a set of inequalities. Problem (4) is an extension of problem (2). The set of elements to allocate is no longer the image of a rectangle but the image of an iteration set which can have any polyhedral shape. This shows that all these problems are closely linked.

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 generate efficient codes for classical block distributions, similar to the ones produced by classical techniques. 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.

Our scheme can also be extended to cope with processor virtualization if the virtualization scheme is expressed with affine constraints. Such a scheme could reuse HPF distribution to map HPF processors on physical processors.

Many partial optimization techniques are integrated in our direct synthesis approach: message vectorization, and aggregation [47], overlap analysis [38]. A new storage management scheme is also proposed. Moreover other optimizations techniques may be applied to the generated code such as vectorization [87], loop invariant code motion [1] and software pipelining [37, 84].

This technique uses algorithms, directly or indirectly, that may be costly, such as FOURIER elimination or the simplex algorithm, which have exponential worst-case behaviors. They are used for array region analysis, in the set manipulations and in the code generation for polyhedron scanning. However our experience with such algorithms is that they remain practical for our purpose: Polyhedron-based techniques are widely implemented in the PIPS framework [51] where HPFC, our prototype HPF compiler, is developped. Firstly, for a given loop nest, the number of equalities and inequalities is quite low, typically a dozen or less. Moreover these systems tend to be composed of independent subsystems on a dimension per dimension basis, resulting in a more efficient practical behavior. Secondly efficient and highly tuned versions of such algorithms are available, for instance in the Omega library. Thirdly, potentially less precise but faster program analysis [21, 13, 45] can also be used in place of the region analysis.

Polyhedron-based techniques are already implemented in HPFC, our prototype HPF compiler [27] to deal with I/O communications in a host/nodes model [28] and also to deal with dynamic remappings [29] (`realign` and `redistribute` directives). For instance, the code generation times for arbitrary remappings are in 0.1–2s range. Future work includes the implementation of our scheme in HPFC, experiments, extensions to optimize sequential loops, to overlap communication and computation, and to handle indirections.

Acknowledgments

We are thankful to Béatrice CREUSILLET for her many questions, Pierre JOUVELOT for his careful reading and many suggestions, William PUGH for helpful comments and debugging, William (Jingling) XUE for the many remarks which helped improve the paper and to the referees for their precise and constructive comments.

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, 1986.

[2] Saman P. Amarasinghe and Monica S. Lam. Communication Optimization and Code Generation for Distributed Memory Machines. In *ACM SIGPLAN International Conference on Programming Language Design and Implementation*, June 1993.

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

[4] Corinne Ancourt and François Irigoin. Scanning Polyhedra with DO Loops. In *Symposium on Principles and Practice of Parallel Programming*, April 1991.

[5] Corinne Ancourt and François Irigoin. Automatic code distribution. In *Third Workshop on Compilers for Parallel Computers*, July 1992.

[6] Françoise André, Olivier Chéron, and Jean-Louis Pazat. Compiling sequential programs for distributed memory parallel computers with Pandore II. Technical report, IRISA, May 1992.

[7] Françoise André, Marc Le Fur, Yves Mahéo, and Jean-Louis Pazat. Parallelization of a Wave Propagation Application using a Data Parallel Compiler. Publication interne 868, IRISA, October 1994.

[8] Françoise André, Jean-Louis Pazat, and Henry Thomas. Pandore: A system to manage data distribution. Publication Interne 519, IRISA, February 1990.

[9] Béatrice Apvrille. Calcul de régions de tableaux exactes. In *Rencontres Francophones du Parallélisme*, pages 65–68, June 1994.

[10] Béatrice Apvrille-Creusillet. Régions exactes et privatisation de tableaux (exact array region analysis and array privatization). Master's thesis, Université Paris VI, France, September 1994. Available via `http://www.cri.ensmp.fr/~creusil`.

[11] Béatrice Apvrille-Creusillet. Calcul de régions de tableaux exactes. *TSI, Numéro spécial RenPar'6*, mai 1995.

[12] Florin Balasa, Frank H. M. Fransen, Francky V. M. Catthoor, and Hugo J. De Man. Transformation on nested loops with modulo indexing to affine recurrences. *Parallel Processing Letters*, 4(3):271–280, March 1994.

[13] V. Balasundaram and Ken Kennedy. A technique for summarizing data access and its use in parallelism enhancing transformations. In *ACM SIGPLAN International Conference on Programming Language Design and Implementation*, June 1989.

[14] Prithviraj Banerjee, John A. Chandy, Manish Gupta, Eugene W. Hodges IV, John G. Holm, Antonio Lain, Daniel J. Palermo, Shankar Ramaswamy, and Ernesto Su. An Overview of the PARADIGME Ccompiler for Distributed-Memory Multicomputers. *IEEE Computer*, 28(10), October 1995.

[15] Siegfried Benkner. Handling Block-Cyclic Distributed Arrays in Vienna Fortran. Technical Report 9, Institute for Software Technology and Parallel Systems, University of Vienna, November 1994.

[16] Siegfried Benkner, Peter Brezany, and Hans Zima. Processing Array Statements and Procedure Interfaces in the Prepare High Performance Fortran Compiler. In *5th International Conference on Compiler Construction*, April 1994. Springer-Verlag LNCS vol. 786, pages 324–338.

[17] John A. Bircsak, M. Regina Bolduc, Jill Ann Diewald, Israel Gale, Jonathan Harris, Neil W. Johnson, Shin Lee, C. Alexander Nelson, and Carl D. Offner. Compiling High Performance Fortran for Distributed-Memory Systems. Report, Digital Equipment Corp., October 1995. To be published in the *Digital Technical Journal*.

[18] 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.

[19] 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.

[20] Thomas Brandes. Evaluation of High Performance Fortran on some Real Applications. In *High-Performance Computing and Networking, Springer-Verlag LNCS 797*, pages 417–422, April 1994.

[21] D. Callahan and Ken Kennedy. Analysis of interprocedural side effects in a parallel programming environment. *Jounal of Parallel and Distributed Computing*, 5:517–550, 1988.

[22] David Callahan and Ken Kennedy. Compiling programs for distributed-memory multiprocessors. *The Journal of Supercomputing*, 2:151–169, 1988.

[23] Zbigniew Chamski. Fast and efficient generation of loop bounds. Research Report 2095, INRIA, October 1993.

[24] Zbigniew Chamski. Nested loop sequences: Towards efficient loop structures in automatic parallelization. Research Report 2094, INRIA, October 1993. In Proceedings of the 27th Annual Hawaii Int. Conf. on System Sciences, 1994, p. 14–22.

[25] 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. *Jounal of Parallel and Distributed Computing*, 26(1):72–84, April 1995. Also appeared in PPoPP'93.

[26] Fabien Coelho. Étude de la Compilation du High Performance Fortran. Master's thesis, Université Paris VI, September 1993. Rapport de DEA Systèmes Informatiques. TR EMP E/178/CRI.

[27] Fabien Coelho. Experiments with HPF Compilation for a Network of Workstations. In *High-Performance Computing and Networking, Springer-Verlag LNCS 797*, pages 423–428, April 1994.

[28] Fabien Coelho. Compilation of I/O Communications for HPF. In *5th Symposium on the Frontiers of Massively Parallel Computation*, pages 102–109, February 1995.

[29] Fabien Coelho and Corinne Ancourt. Optimal Compilation of HPF Remappings. Technical Report A 277, CRI, École des mines de Paris, October 1995.

[30] Jean-François Collard, Paul Feautrier, and Tanguy Risset. Construction of DO loops from Systems of Affine Constraints. LIP RR93 15, ENS-Lyon, May 1993.

[31] Béatrice Creusillet. Analyse de flot de données : Régions de tableaux IN et OUT. In *Rencontres Francophones du Parallélisme*, mai-juin 1995.

[32] Béatrice Creusillet. IN and OUT array region analyses. In *Workshop on Compilers for Parallel Computers*, June 1995.

[33] Béatrice Creusillet and François Irigoin. Interprocedural Array Region Analyses. In *Language and Compilers for Parallel Computing*, August 1995.

[34] Leonardo Dagum, Larry Meadows, and Douglas Miles. Data Parallel Direct Simulation Monte Carlo in High Performance Fortran. *Scientific Programming*, xx(xx):xx, June 1995. To appear.

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

[36] High Performance Fortran Forum. *High Performance Fortran Language Specification.* Rice University, Houston, Texas, November 1994. *Version 1.1.*

[37] Franco Gasperoni and Uwe Scheiegelshohn. Scheduling loop on parallel processors: A simple algorithm with close to optimum performance. Lecture Note, INRIA, 1992.

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

[39] Hans Michael Gerndt and Hans Peter Zima. Optimizing Communication in Superb. In *CONPAR90*, pages 300–311, 1990.

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

[41] Torbjörn Granlund and Peter L. Montgomery. Division by invariant integers using multiplication. In *ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 61–72, June 1994.

[42] Manish Gupta, Sam Midkiff, Edith Schonberg, Ven Seshadri, David Shields, Ko-Yang Wang, Wai-Mee Ching, and Ton Ngo. An HPF Compiler for the IBM SP2. In *Workshop on Compilers for Parallel Computers, Malaga*, pages 22–39, June 1995.

[43] 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.

[44] S.K.S. Gupta, S. D. Kaushik, C.-H. Huang, and P. Sadayappan. On compiling array expressions for efficient execution on distributed-memory machines. Technical Report 19, Department of Computer and Information Science, The Ohio State University, 1994.

[45] Paul Havlak and Ken Kennedy. An implementation of interprocedural bounded regular section analysis. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):350–360, July 1991.

[46] Seema Hiranandani, Ken Kennedy, John Mellor-Crummey, and Ajay Sethi. Compilation techniques for block-cyclic distributions. In *ACM International Conference on Supercomputing*, July 1994.

[47] 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.

[48] Semma Hirannandani, Ken Kennedy, John Mellor-Crummey, and Ajay Sethi. Advanced Compilation Techniques for Fortran D. CRPC-TR 93338, Center for Research on Parallel Computation, Rice University, October 1993.

[49] François Irigoin. Code generation for the hyperplane method and for loop interchange. ENSMP-CAI-88 E102/CAI/I, CRI, École des mines de Paris, October 1988.

[50] 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.

[51] François Irigoin, Pierre Jouvelot, and Rémi Triolet. Semantical interprocedural parallelization: An overview of the PIPS project. In *ACM International Conference on Supercomputing*, June 1991.

[52] S. D. Kaushik, C.-H. Huang, and P. Sadayappan. Compiling Array Statements for Efficient Execution on Distributed-Memory Machines: Two-level Mappings. In *Language and Compilers for Parallel Computing*, pages 14.1–14.15, August 1995.

[53] Wayne Kelly and William Pugh. A framework for unifying reordering transformations. UMIACS-TR-93 134, Institute for Advanced Computer Studies, University of Maryland, April 1993.

[54] Wayne Kelly, William Pugh, and Evan Rosser. Code generation for multiple mappings. In *5th Symposium on the Frontiers of Massively Parallel Computation*, pages 332–341, February 1995.

[55] Ken Kennedy, Nenad Nedeljković, and Ajay Sethi. A Linear Time Algorithm for Computing the Memory Access Sequence in Data-Parallel Programs. In *Symposium on Principles and Practice of Parallel Programming*, 1995. Sigplan Notices, 30(8):102–111.

[56] Ken Kennedy, Nenad Nedeljković, and Ajay Sethi. Efficient address generation for block-cyclic distributions. In *ACM International Conference on Supercomputing*, pages 180–184, June 1995.

[57] 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.

[58] Charles Koelbel, Piyush Mehrotra, and John Van Rosendale. Supporting shared data structures on distributed memory architectures. Technical Report ASD 915, Purdue University, January 1990.

[59] Marc Le Fur. Scanning Parametrized Polyhedron using Fourier-Motzkin Elimination. Publication interne 858, IRISA, September 1994.

[60] Marc Le Fur, Jean-Louis Pazat, and Françoise André. Commutative loop nests distribution. In *Workshop on Compilers for Parallel Computers, Delft*, pages 345–350, December 1993.

[61] Marc Le Fur, Jean-Louis Pazat, and Françoise André. An Array Partitioning Analysis for Parallel Loop Distribution. In *Euro-Par'95, Stockholm, Sweden*, pages 351–364, August 1995. Springer Verlag, LNCS 966.

[62] Hervé Le Verge, Vincent Van Dongen, and Doran K. Wilde. Loop nest synthesis using the polyhedral library. Publication Interne 830, IRISA, May 1994.

[63] 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.

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

[65] John M. Levesque. Applied Parallel Research's xHPF system. *IEEE Parallel and Distributed Technologies*, page 71, Fall 1994.

[66] 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.

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

[68] Larry F. Meadows, Douglas Miles, Clifford Walinsky, Mark Young, and Roy Touzeau. The Intel Paragon HPF Compiler. Technical report, Portland Group Inc., June 1995.

[69] John Merlin. Techniques for the automatic parallelisation of 'Distributed Fortran 90'. SNARC 92 02, University of Southampton, 1992.

[70] Ravi Mirchandaney, Joel S. Saltz, Roger M. Smith, David M. Nicol, and Kay Crowley. Principles of Runtime Support for Parallel Processors. In *ACM International Conference on Supercomputing*, pages 140–152, July 1988.

[71] Carl D. Offner. Digital's HPF Compiler: Meeting the Challenge of Generating Efficient Code on a Workstation Farm. In *NASA Ames Workshop*, 1993.

[72] Edwin M. Paalvast, Henk J. Sips, and A.J. van Gemund. Automatic parallel program generation and optimization from data decompositions. In *1991 International Conference on Parallel Processing — Volume II : Software*, pages 124–131, June 1991.

[73] William Pugh. A pratical algorithm for exact array dependence analysis. *CACM*, 35(8):102–114, August 1992.

[74] Anne Rogers and Keshav Pingali. Process decomposition through locality of reference. In *ACM SIGPLAN International Conference on Programming Language Design and Implementation*, June 1989.

[75] Anne Rogers and Keshav Pingali. Compiling for distributed memory architectures, June 1992. Sent by the first author, when asked for her PhD.

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

[77] J. Stichnoth, O'Hallaron D., and T. Gross. Generating communication for array statements: Design, implementation and evaluation. *Jounal of Parallel and Distributed Computing*, 21(1):150–159, April 1994.

[78] 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.

[79] Ashwath Thirumalai and J. Ramanujam. Fast Address Sequence Generation for Data-Parallel Programs using Integer Lattices. In *Language and Compilers for Parallel Computing*, pages 13.1–13.19, August 1995.

[80] Rémi Triolet, Paul Feautrier, and François Irigoin. Direct parallelization of call statements. In *Proceedings of the ACM Symposium on Compiler Construction*, 1986.

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

[82] Vincent Van Dongen. Compiling distributed loops onto SPMD code. *Parallel Processing Letters*, 4(3):301–312, March 1994.

[83] C. van Reuuwijk, H. J. Sips, W. Denissen, and E. M. Paalvast. Implementing HPF distributed arrays on a message-passing parallel computer system. Computational Physics Report Series, CP-95 006, Delft University of Technology, November 1994.

[84] 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.

[85] Doran K. Wilde and Sanjay Rajopadhye. Allocating memory arrays for polyhedra. Research Report 2059, INRIA, July 1993.

[86] Jingling Xue. Constructing do loops for non-convex iteration spaces in compiling for parallel machines. In *International Parallel Processing Symposium*, April 1995.

[87] Hans Zima and Barbara Chapman. *Supercompilers for Parallel and Vector Computers*. ACM Press, 1990.

[88] Hans Zima and Barbara Chapman. Compiling for distributed-memory systems. *Proceedings of the IEEE*, February 1993.

[89] 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.