Automatic Generation of Data Parallel Code

Jean-François Collard* Paul Feautrier†

Abstract

The literature on automatic parallelization generally focuses on data dependency analysis but seldom on code generation. The generated code may be MIMD-like code with explicit message passing, or a higher-level data-parallel one. This report describes a scheme to automatically generate the latter kind of codes, with a focus on data-parallel extensions of FORTRAN such as the High Performance Fortran (HPF) language.

Keywords: automatic parallelization, code generation, data-parallel languages, HPF

Résumé

La littérature sur la parallélisation automatique est généralement centrée sur l'analyse des dépendances et l'ordonnancement, mais s'intéresse rarement à la génération de code. Le code généré peut être du style MIMD avec transmission explicite des messages, ou de plus haut niveau, par exemple data-parallèle. Ce rapport présente un schéma de production de ce dernier type de code, avec une attention particlulière pour les extensions data-parallèles de FORTRAN telles que le langage High Performance Fortran (HPF).

*LIP, ENS Lyon, 46 Allée d'Italie, F-69364 LYON CEDEX 07. jfcollar@lip.ens-lyon.fr

†PRISM, Université de Versailles, 45 Avenue des Etats-Unis, F-78035 VERSAILLES CEDEX Paul.Feautrier@prism.uvsq.fr

```
@INPROCEEDINGS{CF:93,
        AUTHOR = {J.-F. Collard and P. Feautrier},
        TITLE = {Automatic Generation of Data Parallel Code},
        BOOKTITLE = {Proceedings of the Fourth International Workshop
          on Compilers for Parallel Computers},
        ADDRESS = {Delft, The Netherlands},
        EDITOR = {H.J. Sips},
        MONTH = Dec,
        YEAR = 1993,
        PAGES = {321--332}
}
```

1 Introduction

When Lamport invented the hyperplane method [Lam74], he introduced so fruitful an idea that current automatic parallelization research still heavily relies on it: operations spanned by a loop body are clustered in waves (or "hyperplanes", or "fronts"), one wave being executed at a time. Time is here a logical schedule and sums up, in theory, the inherent sequentiality of the source program — or, rather, what the compiler thinks is the inherent sequentiality of the source program. This logical time may be implicit, as in the loop transformation framework, or explicit, as in methods inherited from systolic design.

The PAF (*Parallélisеur Automatique pour* FORTRAN) automatic parallelizer designed by Feautrier [RWFt91] uses explicit schedules. The first step is dataflow analysis [Fea91] from which one may deduce a single-assignment form of the source program. One then obtains an affine schedule which gives the execution date of every operation. This schedule may be one-dimensional [Fea92b], or possibly multi-dimensional [Fea92a]. A space mapping is also computed, yielding a virtual geometry or "template" on which operations are to be executed [Fea93]. This space mapping is again an affine function in the original loop counters. These elements compose a space-time mapping[1].

Once a space-time mapping is chosen, the corresponding parallel code has to be generated; this is the central theme of this paper. A space-time mapping is in fact a basis change, where the new basis is chosen such that fronts are parallel to the space axes. The generated code has to enumerate, in the new basis, every integer point of the iteration space [AI91]. One should note that space-time mappings have to be invertible. However, it is not always possible to get unimodular space-time mappings. Thus, the logical space-time may have "holes", i.e. instants and places at which no useful work is done.

In the target program, iteration domains are scanned along the spatial dimensions using a data-parallel statement or a new nest of parallel `forall` loops. This parallelization method is thus particularly well suited to SIMD or SPMD distributed-memory machines. These `forall` loops are surrounded by one or more sequential `do` loops scanning time coordinates. To conserve the semantics of the source program, all the references to arrays must be correctly reindexed.

This paper does not deal with finding a suitable basis change, but with the code generation phase of the parallelizer. An overview of the problem of scanning polyhedra after non-unimodular transformations is presented in Section 2. Statement reindexation is presented in Section 3. Section 4 explains how loops are derived in a way that solves the problems above, along with examples of object code. Then we conclude in Section 5, and sum up what remains to be done.

2 Non-Unimodular Transformations

To get an idea of the problems to be faced, consider the following depth-2 perfect loop nest, parametrized by n:

```
for i= 0 , n
  for j= 0 , n
```

[1] Some functional language compilers towards distributed memory architectures follow exactly the same approach, determining *temporal* and *spatial* parts of *index domains* [CCL91].

```
      S
    endfor
  endfor
```

and suppose that statement S is scheduled at $\theta(i,j) = 2i + 4j + 1$, due to dependences. S's execution dates will be 1,2,4, and so on... We say that S's schedule has *period* 2. The *scheduling vector* is $(2,4)^T$, and is non primitive. So, the space-time transformation cannot be unimodular. The temporal dimension to be scanned in the generated program will thus have holes. However, to get a "dense" spatial dimension, we may divide the scheduling vector by the gcd of its components. In our example, $(2,4)^T / \gcd(2,4) \mapsto (1,2)^T$. We *then* complete the scheduling vector to a unimodular matrix, e.g.

$$\begin{bmatrix} 1 & 2 \\ 1 & 1 \end{bmatrix}.$$

We of course go back to the real scheduling vector, getting a non-unimodular matrix

$$\begin{bmatrix} 2 & 4 \\ 1 & 1 \end{bmatrix}$$

whose associated transformation maps convex polyhedra to domains having no holes along the second dimension. The purpose of such a transformation is of course to have step-1 `forall`s enumerating processor coordinates. However, this cannot be immediately extended to multi-dimensional schedules, since steps or periods along these dimensions are not simply the gcd of their corresponding scheduling vector entries. Given the matrix of the transformation, the steps are in fact the diagonal coefficients of the Hermite form of this matrix [Dar93].

We will here restrict ourselves to one-dimensional schedules but give multi-dimensional generalizations when appropriate. Even though the bulk of our algorithms would not change, taking multi-dimensional schedules into account needs more comprehensive linear algebra techniques.

3 Reindexation

3.1 Problem Definition

The transformation which allows one to express statements in a new basis is well known [RWFt91, GMrQtS89, DR93] and is usually called *reindexation* . Note, however, that when loop nests are imperfect, the reindexation we have to deal with is more complex than the one in [LP92] or [Ban90].

To see what reindexation consists in, let S_a and S_b be two statements in the original program, nested in possibly different loop nests.

S_b : $\mathtt{b}[\vec{\imath}_b] = ...$
S_a : $\mathtt{a}[\vec{\imath}_a] = ...\mathtt{b}[\mathcal{L}(\vec{\imath}_a)]...$

The former statement initializes an array which is read by the latter. More precisely, for a given value of S_a's iteration vector $\vec{\imath}_a$, operation $(S_a, \vec{\imath}_a)$ reads the array element $\mathtt{b}[\mathcal{L}(\vec{\imath}_a)]$. $\vec{\imath}_a$ of course belongs to S_a's iteration space, denoted by $\mathcal{I}(S_a)$. $\mathcal{L}$ is an index

function which maps $\mathcal{I}(S_a)$ onto $\mathcal{I}(S_b)$. Let T_a and T_b be S_a's and S_b's space-time mappings, respectively:

$$\vec{c}_a = T_a(\vec{\imath}_a), \vec{c}_b = T_b(\vec{\imath}_b)$$

Reindexation could be done in two different ways.

- Since space-time mappings are chosen invertible, the k^{th} counter $\vec{\imath}_x[k]$ in statement S_x is replaced by ${T_x}^{-1}(\vec{c}_x)[k]$. This yields:

  S_b : b$[T_b^{-1}(\vec{c}_b)]$ = ...
  S_a : a$[{T_a}^{-1}(\vec{c}_a)]$ = ...b$[\mathcal{L}({T_a}^{-1}(\vec{c}_a))]$..

  Array subscripts may be quite complex, making physical mapping intricate and regular communication patterns difficult to detect.

- We aim at having left-hand side arrays subscripted by the new space-time components. Arrays a and b are replaced by two new arrays a' and b':

  S_b : b'$[\vec{c}_b]$ = ...
  S_a : a'$[\vec{c}_a]$ = ...b'$[\vec{c'}_b]$..

  where

  $$\vec{c'}_b = T_b \circ \mathcal{L} \circ {T_a}^{-1}(\vec{c}_a). \quad (1)$$

  The first entries in vector $\vec{c'}_b$ are exactly S_b's time components, and the others are exactly virtual processor coordinates.

The second solution has the following advantages. Suppose vector $\vec{c}_a$ is equal to the two-dimensional vector $(t, \vec{p}_a)^T$, where t is the one-dimensional global time and $\vec{p}_a$ is the coordinate vector of the executing processor. Moreover, suppose that vector $T_b \circ \mathcal{L} \circ {T_a}^{-1}(\vec{c}_a)$ is equal to $(t-d, \vec{p'})$ where d is a positive numerical constant. Then each row in array b' can safely be deallocated, as far as statement S_a is concerned, d time steps after its definition. If all references to b' are of the above form, and if D is the maximum of all d's, then only $D+1$ rows of b' need be allocated along the temporal dimension, provided the references are "wrapped" modulo $D+1$. In some cases, a more elaborate analysis may reduce the factor of $D+1$ to D, see [Cha93] for details. In this way, the main drawback of single-assignment form, memory expansion, is greatly reduced.

Similarly, if the new access vector is of the form $(t', \vec{p}_a + \vec{x})$ where $\vec{x}$ is a constant vector, then communication between S_a's and S_b's processors is regular. Communication patterns can be deduced from the value of $\vec{x}$ in a straightforward way.

3.2 An example

Let us take the following example:

S_b : b$[i, j]$ = ...
S_a : a$[i', j']$ = ...b$[2j', 2i' + j' + 1]$..

S_b's iteration vector is $\vec{\imath}_b = (i,j)^T$. Suppose its schedule is:

$$\theta_b(i,j) = 2i + 4j + 1.$$

If i's and j's lower bounds are both 0, then the execution date set is {1,3,5,...}. Let the space mapping be:

$$\pi_b(i,j) = i + j.$$

The space-time mapping is thus:

$$T_b : \begin{bmatrix} t \\ p_b \end{bmatrix} = \begin{bmatrix} 2 & 4 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix}.$$

Let us now consider S_a, whose iteration vector is $\vec{\imath}_a = (i',j')^T$. Suppose S_a's schedule is:

$$\theta_a(i',j') = 3j'.$$

The execution date set for statement S_a is then {0,3,6,...}. S_a's and S_b's executions thus follow the time diagram in Fig. 1.

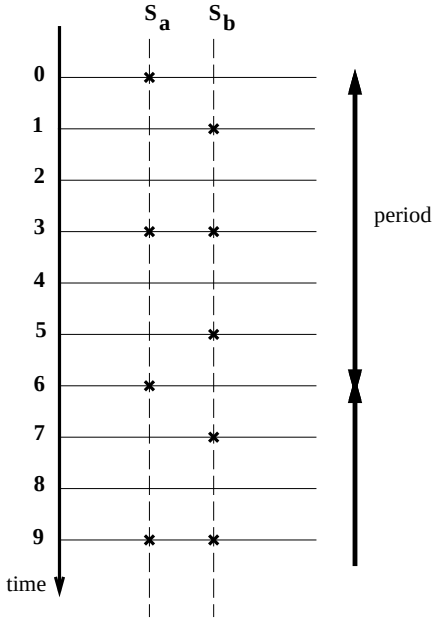

Figure 1: Time diagram for schedules $3j$ and $2i + 4j + 1$

Suppose S_a's space mapping is:

$$\pi_a(i',j') = (i' + j').$$

That is:

$$T_a = \begin{bmatrix} t \\ p_a \end{bmatrix} = \begin{bmatrix} 0 & 3 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} i' \\ j' \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

Then, b's reindexation in S_a will be:

$$\begin{bmatrix} t_b \\ p_b \end{bmatrix} = \begin{bmatrix} 0 & 8 \\ 1/3 & 2 \end{bmatrix} \begin{bmatrix} t_a \\ p_a \end{bmatrix} + \begin{bmatrix} 5 \\ 1 \end{bmatrix}.$$

4 Nest Generation

4.1 Time loop generation

Two problems appear when constructing the transformed loop nest:

- Bounds of the resulting iteration space may depend on structure parameters (whose value may not be known at compile time). Thus, one needs precise symbolic bounds computation. Care must be taken so as not to "forget" points (i.e., operations!) of the original iteration space, and not to create spurious ones. To find the loop bounds of a reindexed loop nest, we use a parametrized version of the Dual Simplex method, implemented by Feautrier in the PIP software (Parameter Integer Programming [Fea88]). Much more details are given in [CFR93]. As a by-product, this procedure yields bounds on the logical time, i.e. the latency of the algorithm, and on processors numbers, which give the size of the underlying geometry or template.

- Since transformations may not be unimodular, some statements may not be executed at each time step, possibly yielding a complicated activation pattern.

As far as the last problem is concerned, let:

$$\theta(\vec{\imath}) = \sum_{j=k}^{n} a_k \vec{\imath}_k + a_0$$

be the schedule of a given statement. Let ϖ be the the greatest common divisor of $a_1 \ldots a_n$. Obviously, the possible values of $\theta(\vec{\imath}) - a_0$ are multiples of ϖ, and the activation pattern of the distinguished statement is periodic with a period of ϖ. In the case where the nest contains several statements with periods ϖ_a, ϖ_b, etc., it is easy to see that the period of the activation pattern will be the least common multiple of ϖ_a, ϖ_b, etc.

In our running example, schedules are $2i + 4j + 1$ and $3j$, thus periods are 2 and 3, and the period of the nest is 6.

A first code generation scheme is to unroll the loop is such a way that the period is in evidence. In the new loop body of our example, statement S_b would appear three times but S_a only twice. The program skeleton is now:

```
for t = 0 to ... step 6
  if min(θ_a) ≤ t ≤ max(θ_a) then S_a(t)
  if min(θ_b) ≤ t + 1 ≤ max(θ_b) then S_b(t + 1)
  if min(θ_a) ≤ t + 3 ≤ max(θ_a) then S_a(t + 3)
  if min(θ_b) ≤ t + 3 ≤ max(θ_b) then S_b(t + 3)
  if min(θ_b) ≤ t + 5 ≤ max(θ_b) then S_b(t + 5)
endfor
```

Since this method would have created heavy overheads, we chose a different implementation. For every statement, the sets of possible schedule values (e.g. {1,3,5,..} and {0,3,6,..} for S_b and S_a respectively), are scanned by local time variables (say, t_a and t_b respectively). These local times are initialized to their lower bounds, are independently

incremented by their periods each time the corresponding statement is executed, and then set to an illegal value (e.g. -1) when execution must end. The code for our running example has thus the following skeleton. (Note that `b'`'s reindexing in S_a has been found in Section 3.2.)

$lb_b = \theta'_b s\, lower\, bound$
$ub_b = \theta'_b s\, upper\, bound$
$t_b = lb_b$
$lb_a = \theta'_a s\, lower\, bound$
$ub_a = \theta'_a s\, upper\, bound$
$t_a = lb_a$
`for` t `=` ... `step` 1
  `if` $t == t_b$ `then`
    `forall` $p \in b$'s topology
      `b'`$[t, p]$ `=` ...
    `endforall`
    $t_b = t_b + \varpi_b$
    `if` $t_b > ub_b$ `then` $t_b = -1$
  `endif`
  `if` $t == t_a$ `then`
    `forall` $p \in a$'s topology
      `a'`$[t, p]$ `=`
      ..`b'`$[8 \times p + 5, 1/3 \times t + 2 \times p + 1]$..
    `endforall`
    $t_a = t_a + \varpi_a$
    `if` $t_a > ub_a$ `then` $t_a = -1$
  `endif`
`endfor`

The first advantage of this scheme is that control overhead, compared to the sequential case, is reduced to the equality tests between global and local times. Moreover, this scheme obviously avoids code duplication. However, spurious array allocation is still a problem (e.g. even entries of array `b'` are uselessly allocated). To circumvent this difficulty, we note that, for every statement S_x:

$$\forall \vec{\imath}, \exists \tau \ : \ t = \theta_x(\vec{\imath}) = \tau \times \varpi_x + l_x$$

where $l_x = \min_{\vec{\imath}}(\theta_x(\vec{\imath}))$ [2].

Thus, τ is a non-negative integer whose initial value is 0, and is incremented every time S'_b's body is entered. Indexing `b'` by τ implies that there is a one-to-one correspondence between `b'` cells and S'_b body executions. For this reason, we call τ a "minor time". Using minor times as subscripts has the added advantage of avoiding division in the transformed subscript expression.

The generated code for our example is then:

$\tau_b = 0$
$\tau_a = 0$
/* Other initializations do not change */
`for` t `=` ... `step` 1
  `if` $t == t_b$ `then`
    `forall` $p \in b$'s topology

[2] Note that this can be immediately extended to multi-dimensional schedules, when the ϖ_x are given by the transformation's Hermite matrix.

```
      b'[τ_b, p] = ...
    endforall
    t_b = t_b + ϖ_b
    τ_b = τ_b + 1
    if t_b > ub_b then t_b = −1 endif
  endif
  if t == t_a then
    forall p ∈ a's topology
      a'[τ_a, p] =
      ..b'[4 × p + 2, τ_a + 2 × p + 1]..
    endforall
    t_a = t_a + ϖ_a
    τ_a = τ_a + 1
    if t_a > ub_a then t_a = −1 endif
  endif
endfor
```

4.2 Memory allocation

In Section 3.1, we noted that arrays, when accessed in a right hand side by time translations such as

$$(t - d, \vec{p}_a + \vec{x})^T,$$

could be slimmed to $d + 1$ rows along the time dimension. What happens if we use minor times?

Firstly, note that a necessary condition for the time component to be a translation is that S_a's and S_b's schedules have the same period ϖ. Then, we can write

$$\forall t,\ \exists \tau_b\ :\ t - d = \tau_a \varpi + l_a - d = \tau_b \varpi + l_b$$

$$\Leftrightarrow d' = \tau_a - \tau_b = \frac{l_b - l_a + d}{\varpi}.$$

Thus $l_b - l_a + d$ is a multiple of ϖ. Moreover, since d' may be negative, the number of allocated rows becomes $\mid d' \mid +1$.

4.3 Generating Fortran

Due to the single-assignment form, some expressions in right-hand sides may be conditionals. Typically, such expressions appear in statement such as:

```
if t == t_1 then
  forall p ∈ b's topology bounds
    b[τ_1, p] =if C(t, p) then y else z
  endforall
  t_1 = t_1 + ϖ_1
  τ_1 = τ_1 + 1
  if t_1 > ub_1 then t_1 = −1 endif
endif
```

Since conditional expressions are not included in Fortran, they must be discarded. The only way to do this is to split such statements in two pieces. One piece will be executed when $C(t, p)$ holds, and the other when $C(t, p)$ does not hold. These two statement

pieces will have differing date sets, whose extrema are computed by adding $C(t,p)$ (resp. $\neg C(t,p)$) to the inequalities which define the iteration domain.

Moreover, note that these pieces are writes to the same array (the array initially written by the split statement). To preserve single-assignment, minor times must be shared among the statement pieces, even if their local times are different. Thus, the initial statement is rewritten as:

/* t_{11} and t_{12} are initialized to their lower */
/* bounds, ie lb_{11} and lb_{12} respectively. */
flag = false
if $t == t_{11}$ **then**
  forall $p \in b$'s topology, s.t. $C(t,p)$ holds
    $b[\tau_1, p]$ = y
  endforall
  $t_{11} = t_{11} + \varpi_1$
  flag = true
  if $t_{11} > ub_{11}$ **then** $t_{11} = -1$ **endif**
endif
if $t == t_{12}$ **then**
  forall $p \in b$'s topology, s.t. $\neg C(t,p)$ holds
    $b[\tau_1, p]$ = z
  endforall
  $t_{12} = t_{12} + \varpi_1$
  flag = true
  if $t_{12} > ub_{12}$ **then** $t_{12} = -1$ **endif**
endif
/* if one of the two pieces has been
executed, increment minor time τ_1 */
if flag == true **then**
  $\tau_1 = \tau_1 + 1$
endif
flag = false

4.4 Implementation & Example

This scheme has been implemented so as to generate C or Fortran code augmented with `forall`, a data-parallel extension of `for`. New variables are of six kinds: local times (`t`), lower and upper schedule bounds (`lb` and `ub` respectively), schedule periods (`per`), minor times (`m`) and execution flags (`exe`). Variable names are prefixed by their corresponding statement names, and suffixed by time dimensions.

As a real-life example, take the Gaussian elimination written as:

```
    program gauss
    real a(n,n), x(n)
    real s, f
    integer i, j, k, n

    do i=1, n-1
     do j=i+1, n
1      f=a(j,i)/a(i,i)
       do k=i+1,n
2        a(j,k)=a(j,k)-f*a(i,k)
      enddo
     enddo
```

```
    enddo
    do i = 1, n
3     s = 0.
      do j = 1, i-1
4       s = s + a(n-i+1, n-j+1)*x(n-j+1)
      enddo
5     x(n-i+1) = (a(n-i+1, n+1) - s)
   & /a(n-i+1, n-i+1)
    enddo
    end
```

A Schedule for this example has been computed in [Fea92a] and appears in the third column of Table 1.

Statement	Counters	Schedule
S_1	i, j	$2i-2$
S_2	i, j, k	$2i-1$
S_3	i	0
S_4	i, j	$2j+2n-2$
S_5	i	$2i+2n-3$

Table 1: A schedule for `gauss`

Statement	Mapping
S_1	$n+1-j, n-1$
S_2	$n+1-j, k+n-1$
S_3	$i, 2n$
S_4	$i, 2n$
S_5	$0, 2n$

Table 2: A space mapping

On the other hand, a two dimensional space mapping is given in Table 2. When reindexation has been done, we notice that all arrays S_1, .., S_5 are accessed with constant delays 1,2,1,2 and 1, respectively. For instance, Statement S_1 is wrapped modulo 1, i.e. only one row is needed along its first dimension instead of $n-2$ without wrapping and $2n-4$ without using minor times.

The following MasPar Fortran code is generated[3]. Note that predicates such as $C(t,p)$ do not appear explicitly in the final code, but are taken into account when computing the bounds on local times and processor coordinates.

```
  PROGRAM gauss

  real S_1(0:0 , 1:n-1 , n-1:n-1)
  real S_2(0:1 , 1:n-1 , n+1:2*n-1)
  real S_3(0:0 , 1:n   , 2*n:2*n)
C ...
```

[3] Variables are named according to the following convention: variables belonging to part number a of Statement S_b are labeled with `S_`b`n`a. For space reason, the generated code has been truncated after S_3.

```
C .. mapping directives for the MP-1:

CMPF MAP  S_1 ( MEMORY,XBITS,YBITS )
CMPF MAP  S_2 ( MEMORY,XBITS,YBITS )
CMPF MAP  S_3 ( MEMORY,XBITS,YBITS )
C ...

C .. declarations ..

  S_1n0per0=2
  S_1n0lb0=2
  S_1n0ub0=2*n-4
  S_1n0t0=S_1n0lb0
  S_1n1per0=2
  S_1n1lb0=0
  S_1n1ub0=0
  S_1n1t0=S_1n1lb0
  S_1m0=0
  S_1exe=0

  S_2n0per0=2
  S_2n0lb0=3
  S_2n0ub0=2*n-3
  S_2n0t0=S_2n0lb0
  S_2n1per0=2
  S_2n1lb0=1
  S_2n1ub0=1
  S_2n1t0=S_2n1lb0
  S_2m0=0
  S_2exe=0

  S_3n0per0=0
  S_3n0lb0=0
  S_3n0ub0=0
  S_3n0t0=S_3n0lb0
  S_3m0=0
  S_3exe=0

  DO time0 = 0,max((4*n-3),(2*n)),1

C The first statement is split in two pieces...

    IF (S_1n0t0.EQ.time0) THEN
    FORALL (p0 = 1:n-(S_1m0+1))
  -   S_1(0,p0,n-1)=
  -   S_2( mod((S_1m0-1),2), p0, n+S_1m0)
  -   /S_2( mod((S_1m0-1),2), n-S_1m0, n+S_1m0)
    S_1exe=1
    S_1n0t0=S_1n0t0+S_1n0per0
    IF (S_1n0t0.GT.S_1n0ub0) THEN
    S_1n0t0=-1
    END IF
    END IF
    IF (S_1n1t0.EQ.time0) THEN
    FORALL (p0 = 1:n-1)
  -   S_1(0,p0,n-1) =
```

```
-    a((n+1)-p0,S_1m0+1)
-    / a(S_1m0+1,S_1m0+1)
     S_1exe=1
     S_1n1t0=S_1n1t0+S_1n1per0
     IF (S_1n1t0.GT.S_1n1ub0) THEN
     S_1n1t0=-1
     END IF
     END IF

C Has one of them been executed?...

     IF (S_1exe.EQ.1) THEN
     S_1m0=S_1m0+1
     END IF
     S_1exe=0


     IF (S_2n0t0.EQ.time0) THEN
     FORALL (p0 = 1:n-(S_2m0+1),
-         p1 = n+S_2m0+1:2*n-1)
-    S_2(mod(S_2m0,2),p0,p1)=
-     S_2(mod((S_2m0-1),2),p0,p1)
-     - S_1(0,p0,n-1)
-     * S_2(mod((S_2m0-1),2),n-S_2m0,p1)
     S_2exe=1
     S_2n0t0=S_2n0t0+S_2n0per0
     IF (S_2n0t0.GT.S_2n0ub0) THEN
     S_2n0t0=-1
     END IF
     END IF

     IF (S_2n1t0.EQ.time0) THEN
     FORALL(p0 = 1:n-1, p1 = n+1:2*n-1)
-    S_2(mod(S_2m0,2),p0,p1) =
-     a((n+1)-p0,(p1+1)-n)
-     - S_1(0,p0,n-1)
-     * a(S_2m0+1,(p1+1)-n)
     S_2exe=1
     S_2n1t0=S_2n1t0+S_2n1per0
     IF (S_2n1t0.GT.S_2n1ub0) THEN
     S_2n1t0=-1
     END IF
     END IF

     IF (S_2exe.EQ.1) THEN
     S_2m0=S_2m0+1
     END IF
     S_2exe=0

C Statement S3 is not split:

     IF (S_3n0t0.EQ.time0) THEN
     FORALL (p0 = 1:n)  S_3(0,p0,2*n)=0.
     S_3m0=S_3m0+1
     S_3n0t0=S_3n0t0+S_3n0per0
     IF (S_3n0t0.GT.S_3n0ub0) THEN
     S_3n0t0=-1
```

```
    END IF
    END IF

C ... the other statements...

    END DO
   END
```

More details are given in [Col94].

4.5 Expressing Parallelism using FORALLs

FORALLs seem to be perfectly adequate to express the parallelism extracted by PAF. However, their current implementation suffer from various restrictions which complicate the code generation process. Many common patterns, appearing in the transformed code, are not accepted by current commercial compilers. We list below the main problems.

FORALL indices. They should appear inside the statement in the same order as in the FORALL header. Moreover, every FORALL header index should appear exactly once in the array references. For instance, the line

```
FORALL(j=1:N,i=1:N) A(j,i)=B(j)
```

is rejected because "array reference does not contain all FORALL indices". Note that this syntax is explicitly accepted in the HPF proposal [For93, Lov93], and this phenomenon occurs in many statements in our `gauss` example.

Non-conformable arrays. Suppose a one-dimensional array is to be copied in every column of a two-dimensional one. Consider the following code:

```
REAL A(N,N), B(N)
FORALL (j=1:N) A(:,j) = B
```

The MasPar-Fortran compiler warns that the statement will be executed serially and complains that "functions of FORALL indices [are] not allowed". Expliciting all the indices does not help, since we would go back to the previous error. The only way we are aware of to implement such a specification in MPF is to use array sections and the `SPREAD` function.

Scalar reference to an array. The compiler complains that

```
FORALL(I=1:N) B(I) = PIV(J)
```

is a "front-end array reference with FORALL subscript(s)". Adding the directive `CMPF ONDPU PIV` to put the array on the DPU does help, though. Since `I` does not index `PIV`, this is then an "array reference without FORALL indices", just a special case of the first error. In conclusion, this kind of scalar diffusion is forbidden in current FORALL implementations. We would have to store `PIV(J)` in a temporary scalar. Since scalars are stored on the front-end, this would yield a so-called scalar "slosh" (i.e. a data movement from the front-end to the processor array, or conversely), but would avoid the pitfalls above.

Non-rectangular iteration domains. The following syntax is accepted by HPF:

```
      FORALL(I=1:N,J=1:N,I>J)
&       A(I,J) = 0.
```

The mask is here a scalar expression. But in MPF, the masking expression must be conformable with the masked array, e.g. with `A` in the statement above. Thus, we have to create an auxiliary boolean array to that purpose:

```
      REAL A(N,N)
      LOGICAL B(N,N)
      FORALL(I=1:N,J=1:N)
&       B(I,J) = J.GT.I
      FORALL(I=1:N,J=1:N,B(I,J))
&       A(I,J) = 0.
```

4.6 Physical Mapping Directives

To achieve the best speed, programs should use at all time as many physical processors as possible. Using the mapping directives to tune processor allocation is thus vital. In our case, we obviously want the arrays' first dimensions to be stored in the same processor memories. The remaining dimensions have to be spread across the physical topology.

For instance, the MasPar MP-1 is a two-dimensional grid. A one-dimensional array is supposed to be mapped in a so-called "raster-scan" way, i.e. by linearizing the grid and then storing the array elements on it, with a possible wrap-around if the array size exceeds the number of processors. This method guarantees that all the processors are used with good load balancing. To enforce this mapping, we generate the `ALLBITS` directive for a 2-D array's second dimension. The last two dimensions of a three-dimensional array are laid across the processors using two kinds of directives:

- `XBITS` and `YBITS`, successively, if both dimensions extents depend on size parameters;
- If a dimension extent is known at compile time, and if this extent is lower than the order of the physical grid (32 in our MP-1), then this dimension should, according to our first experiments, get the `MEMORY` directive. Otherwise, this dimension would use one grid dimension without filling it. The other dimension should then be laid across all the processors using `ALLBITS`.

4.7 Performance

Timing studies have been done on a MasPar MP-1, equipped with 32×32 processors, each having 16 Kbytes of local memory. This machine's peak performance in register-to-register floating point computations has been measured to be 80Mflop/s.

The first test compared an automatically parallelized F77 matrix addition program and its F90 counterpart (whose main statement just takes one line: `A = B + C`.) This experiment allows us to have an idea of the cost of generated code control overhead. Results are shown in Fig.2. The generated code is slightly slower than the hand-coded

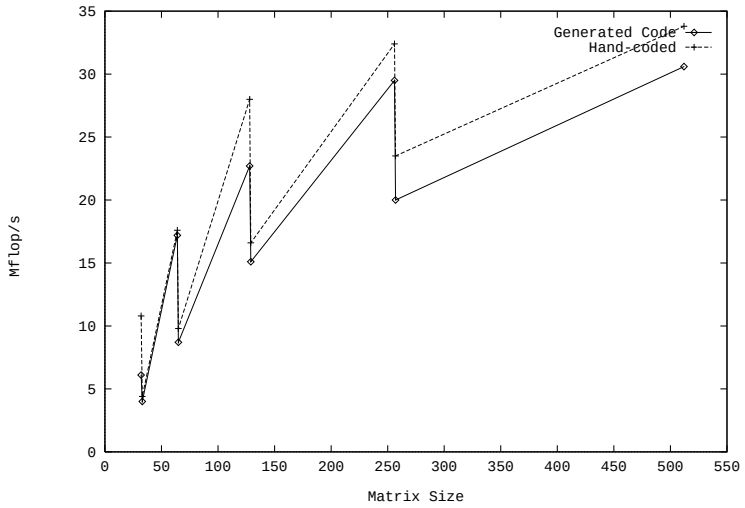

Figure 2: Performance achieved by automatically generated matrix-addition code and its hand-coded counterpart.

one, mainly because of the added statements. Its Mflop/s performance is only 10% lower than the hand-coded program's one on 512×512 matrices.

A second test has been done on a matrix-vector multiplication ($Y = AX$) F77 code. Its automatically parallelized version competed with the hand-written equivalent below.

```
real a(n,n), x(n), y(n), t(n,n)
t = SPREAD(x, dim=1, ncopies=n)
t = a * t
y = SUM(t,dim=2)
```

Using a temporary array `t` exhausted PE memory for $n = 576$ and above. The performances are compared in Fig.3.

5 Conclusion

This software has been written in Le_Lisp, and amounts to about 1500 lines of code. We propose a code generation scheme where control overhead is kept small, where communication patterns can easily be mapped to multi-dimensional grids, and which can be expressed in actual data-parallel languages. Moreover, we have seen that two phenomena imply memory over-allocation: non-unimodular schedules and single-assignment. We showed how both can be dealt with thanks to so-called "minor times" and wrapping modulo constant delays, respectively. These features are essential when target machine nodes have little memory.

6 Acknowledgments

The first author is partly supported by the french CNRS Coordinated Research Program on Concurrency, Communication and Cooperation C^3, PRC/MRE contract ParaDigme and DRET contract 91/1180. The second author has been partially supported by project LHPC-C^3.

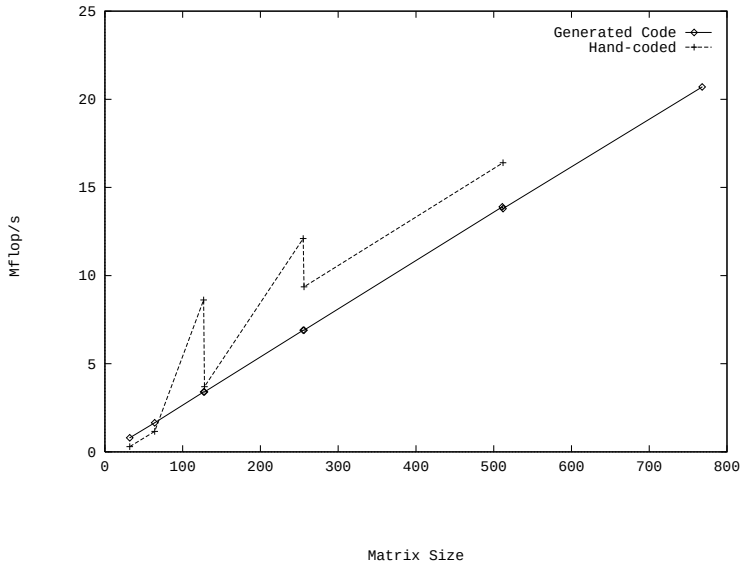

Figure 3: Performance achieved by automatically generated matrix-vector multiplication code and its hand-coded counterpart.

References

[AI91] C. Ancourt and F. Irigoin. Scanning polyhedra with DO loops. In *Proc. ACM SIGPLAN '91*, pages 39–50, June 1991.

[Ban90] U. Banerjee. Unimodular transformations of double loops. Technical Report CSRD Rpt. No. 1036, University of Illinois, August 1990.

[CCL91] M. Chen, Y. Choo, and J. Li. Crystal: Theory and pragmatics of generating efficient parallel code. In B. K. Szymanski, editor, *Parallel Functional Languages and Compilers*. ACM Press Frontier Series, 1991.

[CFR93] J.-F. Collard, P. Feautrier, and T. Risset. Construction of DO loops from systems of affine constraints. Technical Report 93-15, LIP, ENS Lyon, France, 1993.

[Cha93] Z. Chamski. *Environnement logiciel de programmation d'un accélérateur de calcul parallèle*. PhD thesis, Univ. Rennes I, Rennes, February 1993.

[Col94] J.-F. Collard. Code generation in automatic parallelizers. In *Proc. of the Int. Conf. on Applications in Parallel and Distributed Computing, IFIP W.G 10.3*, Caracas, Venezuela, April 1994. North Holland. To appear.

[Dar93] A. Darte. *Techniques de parallélisation automatique de nids de boucles*. PhD thesis, LIP, ENS Lyon, France, 1993.

[DR93] A. Darte and Y. Robert. Mapping uniform loop nests onto distributed memory architectures. Technical Report 93-03, LIP, ENS Lyon, France, January 1993. ftp: `lip.ens-lyon.fr`.

[Fea88] P. Feautrier. Parametric integer programming. *RAIRO Recherche Opérationnelle*, 22:243–268, September 1988.

[Fea91] P. Feautrier. Dataflow analysis of scalar and array references. *Int. Journal of Parallel Programming*, 20(1):23–53, February 1991.

[Fea92a] P. Feautrier. Some efficient solution to the affine scheduling problem, part II, multidimensional time. *Int. J. of Parallel Programming*, 21(6), December 1992.

[Fea92b] P. Feautrier. Some efficient solutions to the affine scheduling problem, part I, one dimensional time. *Int. J. of Parallel Programming*, 21(5):313–348, October 1992.

[Fea93] P. Feautrier. Toward automatic partitioning of arrays on distributed memory computers. In *ACM ICS'93*, Tokyo, July 1993. also available as IBP/report 92.95.

[For93] High Performance Fortran Forum. High performance fortran language specification. Technical report, January 1993. Version 1.0 Draft.

[GMrQtS89] P. Gachet, Ch. Mauras, P. Quinton, and Y. Saouter. A language for the design of regular parallel algorithms. In F. Andre and J.P. Verjus, editors, *First European Workshop on Hypercube and Distributed Computers*, pages 189–202, Rennes, France, October 1989. North-Holland.

[Lam74] L. Lamport. The parallel execution of do loops. *CACM*, 17:83–93, February 1974.

[Lov93] D. B. Loveman. High performance fortran. *IEEE Parallel & Distributed Technology*, pages 25–42, February 1993.

[LP92] W. Li and K. Pingali. A singular loop transformation framework based on non-singular matrices. In *Fifth Annual Workshop on Language and Compilers for Parallelism*, New Haven, August, 1992.

[RWFt91] M. Raji-Werth and P. Feautrier. On parallel program generation for massively parallel architectures. In M. Durand and F. El Dabaghi, editors, *High Performance Computing II*. North-Holland, October 1991.

Reference [CFR93] can be found on anonymous ftp `lip.ens-lyon.fr`.