HAL
archives-ouvertes.fr

Static domain analysis for compiling commutative loop nests

Marc Le Fur, Jean-Louis Pazat, Françoise André

▶ To cite this version:

Marc Le Fur, Jean-Louis Pazat, Françoise André. Static domain analysis for compiling commutative loop nests. [Research Report] RR-2067, INRIA. 1993. <inria-00074605>

HAL Id: inria-00074605
https://hal.inria.fr/inria-00074605

Submitted on 24 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Static Domain Analysis for Compiling Commutative Loop Nests

Marc Le Fur, Jean-Louis Pazat, Françoise André

N° 2067

Octobre 1993

PROGRAMME 1

Architectures parallèles,
bases de données,
réseaux et systèmes distribués

ISSN 0249-6399

Static Domain Analysis for Compiling Commutative Loop Nests

Marc Le Fur, Jean-Louis Pazat, Françoise André *

Programme 1 — Architectures parallèles, bases de données, réseaux
et systèmes distribués
Projet PAMPA

Rapport de recherche n° 2067 — Octobre 1993 — 14 pages

Abstract: In the field of scientific computation, many users wish to use the sequential programming model, even though they aim to execute their program on a Distributed Memory Parallel Computer (DMPC). To meet this demand, some prototypes of compilers have been designed to "distribute" sequential programs onto DMPCs. In this paper, we present a static domain analysis which leads to the generation of efficient code for these machines. This analysis relies on the enumeration of the points of a polyhedron which is based on linear and integer programming.

Key-words: Distributed memory parallel computers, compilation, loop distribution, domain analysis, linear and integer programming.

(Résumé : tsvp)

*mlefur@irisa.fr, pazat@irisa.fr, fandre@irisa.fr

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : (33) 99 84 71 00 – Télécopie : (33) 99 38 38 32

Analyse statique de domaines pour la compilation des nids de boucles commutatifs

Résumé : Dans le domaine du calcul scientifique, beaucoup d'utilisateurs souhaitent programmer en utilisant un modèle séquentiel, même lorsqu'ils visent une exécution sur une Machine Parallèle à Mémoire Distribuée (MPMD). Pour satisfaire cette demande, quelques prototypes de compilateurs ont été développés pour "distribuer" des programmes séquentiels sur des MPMD. Dans ce rapport, nous présentons une analyse statique de domaines permettant la production d'un code efficace pour ces machines. Cette analyse est basée sur l'énumération des points d'un polyèdre qui trouve ses fondements dans la programmation linéaire et entière.

Mots-clé : Machines parallèles à mémoire distribuée, compilation, distribution de boucles, analyse de domaines, programmation linéaire et entière.

1 Introduction

Parallel computers are widely used in the field of scientific computation. A new generation of parallel computers, Distributed Memory Parallel Computers (DMPCs), are now coming into view. DMPCs can be defined as a network of "nodes", each node being a processor tightly connected to its local memory. The number of nodes of these parallel machines can be greater than 1000. The main difference between DMPCs and "conventional" parallel computers comes from the lack of global memory in DMPCs.

Programming this kind of parallel machines is difficult because of many low-level problems due to a programming model based on parallel communicating processes. Even though this model is well known, many users of DMPCs want to use a sequential programming model. They often find more natural to turn their formulas into a sequential program (like FORTRAN *FORmula TRANslator*).

Some prototypes of compilers have been designed to "distribute" a sequential program onto a DMPC. Because this task is very difficult, these compilers rely on a user-defined data decomposition which is used as a guideline to generate communicating processes. For this purpose, new constructs have been added to existing sequential languages. However, no significant efforts have been made to define completely new languages.

2 Compiling techniques

Due to this very particular way of compiling programs, new problems have to be tackled in the field of compiling techniques:

- Because of the size of the machine (number of processors), code generation relies on the SPMD model. Each processor will execute the same code on different parts of the data,
- in order to take into account the data distribution specified in the program, these compilers use a rule to link the SPMD code generation with the data distribution. The rule usually used is called the *owner write rule*.

These techniques are used in many projects such as the FortranD Compiler [HKT92], the Vienna Fortran Compilation System [BCZ92], and in our project Pandore II [Che93].

Scalars and array elements are mapped onto the memories of the DMPC processors according to the user-defined data distribution. For sake of simplicity, we assume that each array is partitioned into blocks, each block being assigned to exactly one processor (there are no duplicates). Scalar elements are replicated on all processors.

The principles of the *owner write rule* are the following:

- If a statement modifies a scalar or an array element, the statement will be executed by the processor where the data is mapped (we say that the processor *owns* the data element),
- if the processor that is to perform a statement needs to fetch a *distant* data element, it will need a cooperation with the *owner* of the data element.

This rule can be specified for each statement of the source language for any sequential language. We use two special instructions in the intermediate SPMD code produced by the compiler: $Refresh$ and $Exec$. $Refresh(v, P)$ updates the local copy of the array element v on the processor P and $Exec(S, P)$ executes the statement S on the processor P. A straightforward (but naïve) run-time implementation of these functions is shown below:

$$\begin{aligned} Refresh(v,P) \quad &\equiv \quad \textbf{if}\,(myself = \pi(v)) \ \wedge \ (P \neq myself)\ \textbf{then send}(P, v) \\ &\qquad\quad \textbf{if}\,(myself = P) \ \wedge \ (\pi(v) \neq myself)\ \textbf{then recv}(\pi(v), v) \end{aligned}$$

$$Exec(S,P) \quad \equiv \quad \textbf{if}\, myself \ = \ P\ \textbf{then}\ S$$

where $myself$ is the name of the current processor, $\pi(v)$ denotes the owner of the array element v. Communication between processors is FIFO and no messages are lost; $send(Q, w)$ sends the value of the array element w to the processor Q (this send is non–blocking), $recv(Q, w)$ waits for receiving the value of the element w from processor Q.

This translation scheme has been proved correct, i.e. there exists some equivalence between the behaviour of the distributed program and the sequential one [BCJT93].

3 Inefficiency of the basic scheme for compiling loops

The compiling scheme is defined at the statement level and thus implies a very large overhead at run-time. Since most scientific codes are based on loops working among large arrays, a compiler should be able to generate efficient code for loops. An interesting class of loops are perfectly nested loops. Among them, we focus on what we call "commutative loop nests"; in these loops, the body is restricted to an assignment and iterations can be executed in any order. Parallel nested loops and reduction operations encountered in scientific programs generally belong to this class; in addition, systematic transformations of sequential programs produce such parallel loops [FM90].

When applied to the following commutative loop nest:

```
for i = 1 , 1000
    for j = i , 2*i+1
        S(i,j) :  A[ i , j-i ] := B[ j , 2*i-2 ]
```

the basic translation scheme leads to:

for $i = 1\ ,\ 1000$
  for $j = i\ ,\ 2{*}i{+}1$
    $S1(i,j) : Refresh(B[\,j\ ,\ 2{*}i{-}2\,],\ \pi(A[\,i\ ,\ j{-}i\,]))$
    $S2(i,j) : Exec(A[\,i\ ,\ j{-}i\,] := B[\,j\ ,\ 2{*}i{-}2\,],\ \pi(A[\,i\ ,\ j{-}i\,]))$

This code cannot be efficient because each $Refresh$ and $Exec$ statement contains guards (**if**) which are computed at run-time for each iteration by each processor. Moreover, the basic scheme does not take into account the potential parallelism of loops in order to separate the generated code execution into a communication phase and a computation phase.

The best way to tackle this problem seems to analyze data access patterns at compile time. In this field, some limited optimization techniques have been proposed in the FortranD compiler and in VFCS. For both compilers, arrays involved in the parallel loop must be mapped in such a way that each processor only owns one block of an array. In the Vienna Fortran Compilation System, the loop bounds and the array access functions are affine functions of only one enclosing index. In the Fortran D compiler, the restrictions are the following: the loop bounds must be constant, the array access functions are in `i+c` form and only one dimension of an array can be distributed. Furthermore, both compilers perform a domain analysis for each processor, leading to a compile time depending on the number of processors.

4 Pandore II optimized compilation scheme

The method presented here can handle a rather large class of regular problems. It applies on commutative loop nests where:

- Array references and loop bounds are affine functions of the enclosing indexes,
- each loop step is equal to 1.

The example given in 3 satisfies these restrictions. As regards data distribution, we make the following assumptions:

- An array can either be replicated on all processors (the array is owned by each processor) or distributed, that is partitioned into rectangular blocks, each block being assigned to exactly one processor,
- scalar elements are systematicaly replicated.

The last assumptions concern the network: it is fully connected, the communication between processors is FIFO and no message is lost.

In the following, we briefly present the main principles of the compilation method and a tool which is at the root of the code generation. Then, we use an example to highlight the compilation technique that is described more formally in 4.4. The last part will show that this method also enables to compile parameterized loops.

4.1 Main Principles

The compilation scheme takes the commutativity of loops into account to separate the generated SPMD code into a communication part and a computation part. The generation of each part relies on a domain analysis that takes advantage of the decomposition into blocks of the arrays. This domain analysis allows to describe sequences of send statements between two processors (leading to a message vectorization between these processors) and sequences of local assignments. Furthermore, this domain analysis is symbolic, i.e. independent of the number of processors.

Because we assume that the loop bounds and the array access functions are affine, the data access domains can be characterized by polyhedrons and the generated code execution will consist in scanning these polyhedrons. Indeed, with any non empty bounded polyhedron can be associated a nested loop performing the enumeration of its points. Two methods are generally used to compute this enumeration code. The most employed method is based on the Fourier-Motzkin pairwise elimination [Sch86, Duf74] and the other one on the parametric integer programming [Fea89]. An algorithm using the first method is given in [IA91] and the enumeration code production using the parametric integer programming is detailed in [CFR93]. Both methods allow to synthesize an enumeration code which is a perfectly nested loop whose lower (resp. upper) bounds are maxima (resp. minima) of quasi-affine expressions (the euclidian division of an affine expression with integer coefficients by a positive integer). For example, the points (i, j, k) of the polyhedron defined by the following inequality system:

$$\begin{cases} i \geq 0 \\ -i + 7 \geq 0 \\ j - 1 \geq 0 \\ -j + 1000 \geq 0 \\ -j + k \geq 0 \\ 2 * j - k + 1 \geq 0 \\ -500 * i + j + k \geq 0 \\ 500 * i - j - k + 499 \geq 0 \end{cases}$$

can be enumerated by the nested loop:

for $i = 0$, 6
  for $j =$ **max**$(1$, **div**$(500{*}i{+}1, 3))$, **min**$(1000$, **div**$(500{*}i{+}499, 2))$
    for $k =$ **max**$(j$, $500{*}i{-}j)$, **min**$(2{*}j{+}1$, $500{*}i{-}j{+}499)$

4.2 Scanning polyhedrons using Fourier-Motzkin pairwise elimination

The method can be briefly described as follows. Let $P = \{x = (x_1, \ldots, x_n) \in \mathbb{Z}^n \ / \ Ax^T + b \geq 0\}$ (A is an integer matrix and c an integer vector) be a non empty bounded polyhedron. Using Fourier-Motzkin elimination, the polyhedron P is successively projected along the $x_n, x_{n-1}, \ldots, x_1$ axis, in order to determine for each x_r:

- A set of *minimizing* constraints: $\mathcal{MI}(x_r) = \{a_{pr}x_r + f_p((x_j)_{j<r}) \geq 0\}_{p \in P_r}$ where $\forall p \in P_r \ a_{pr}$ is a positive integer and f_p an affine function with integer coefficients,
- and a set of *maximizing* constraints: $\mathcal{MA}(x_r) = \{a_{qr}x_r + f_q((x_j)_{j<r}) \geq 0\}_{q \in Q_r}$ with $\forall q \in Q_r \ a_{qr}$ a negative integer and f_q an affine function with integer coefficients.

From $\mathcal{MI}(x_r)$, a set of lower bounds for x_r can be deduced:

$$\mathcal{LB}(x_r) = \{\mathbf{div}(-f_p((x_j)_{j<r}) + a_{pr} - 1, a_{pr})\}_{p \in P_r}$$

whereas the set $\mathcal{MA}(x_r)$ provides upper bounds for x_r:

$$\mathcal{UB}(x_r) = \{\mathbf{div}(f_q((x_j)_{j<r}), -a_{qr})\}_{q \in Q_r}$$

And so, the nested loop associated with polyhedron P is:

$$\begin{array}{l} \mathbf{for}\ x_1 = \mathbf{max}\ \mathcal{LB}(x_1),\ \mathbf{min}\ \mathcal{UB}(x_1) \\ \quad \ddots \\ \qquad \mathbf{for}\ x_n = \mathbf{max}\ \mathcal{LB}(x_n),\ \mathbf{min}\ \mathcal{UB}(x_n) \end{array}$$

Unfortunately, the projection of a polyhedron along an axis generates a lot of redundant inequalities [Sch86, Che87]. This results in many useless bounds in the nested loop scanning the polyhedron and so, in an important run-time overhead. Therefore, these redundant inequalities must be eliminated. One must take care of two crucial points here:

- The redundancy test used,
- the way how redundant inequalities are eliminated.

In the tool that we developed for our compiler, the redundancy test used is a simplex [Sch86]. Once the sets $\mathcal{MI}(x_r)$ and $\mathcal{MA}(x_r)$ have been computed for each x_r, redundant inequalities are eliminated, leading to a new set of minimizing constraints: $\mathcal{MI}'(x_r)$ and a new set of maximizing constraints: $\mathcal{MA}'(x_r)$ for each x_r. Unlike the algorithm given in [IA91], the sets $\mathcal{MI}'(x_r)$ and $\mathcal{MA}'(x_r)$ are computed in the following way:

2. The enumeration code of polyhedron $\mathcal{E}$ can be computed by one of the methods cited above:

   for $k_A = 0\ ,\ 2$
   for $k_B = \mathbf{max}(0\ ,\ \mathbf{div}(500{*}k_A{-}1, 250))\ ,\ \mathbf{min}(3\ ,\ \mathbf{div}(250{*}k_A{+}249, 125))$
   for $i = \mathbf{max}(250{*}k_B{+}1\ ,\ 500{*}k_A)\ ,\ \mathbf{min}(\mathbf{div}(500{*}k_B{+}501, 2)\ ,\ 500{*}k_A{+}499)$
   for $j = i\ ,\ 2{*}i{+}1$

   The first two loops of this enumeration code show that some pairs of blocks $(k_A, k_B) \in 0..7 \times 0..7$ are not reached during the computation.

3. We then insert two guards in this enumeration code in order to generate the SPMD send code:

   for $k_A = 0\ ,\ 2$
   if *myself* $\neq$ owner of block k_A of A **then**
   for $k_B = \mathbf{max}(0\ ,\ \mathbf{div}(500{*}k_A{-}1, 250))\ ,\ \mathbf{min}(3\ ,\ \mathbf{div}(250{*}k_A{+}249, 125))$
   if *myself* = owner of block k_B of B **then**
   for $i = \mathbf{max}(250{*}k_B{+}1\ ,\ 500{*}k_A)\ ,\ \mathbf{min}(\mathbf{div}(500{*}k_B{+}501, 2)\ ,\ 500{*}k_A{+}499)$
   for $j = i\ ,\ 2{*}i{+}1$
   send $B[j\ ,\ 2{*}i{-}2\,]$ to the owner of block k_A of A

   In practice, it should be noted that the array elements are not sent elementwise but are combined into vectors. The SPMD receive code is then generated in the same way.

Computation code generation

1. As before, the set of points (k_A, i, j) where $k_A \in 0..7$ is a block of A, (i, j) an iteration vector such that $S(i, j)$ writes in k_A, can be characterized by the polyhedron $\mathcal{C}$ defined by the following system:

$$\begin{cases} 0 \leq k_A \leq 7 \\ 1 \leq i \leq 1000 \\ i \leq j \leq 2 * i + 1 \\ 500 * k_A \leq i \leq 500 * k_A + 499 \end{cases}$$

2. The points of $\mathcal{C}$ can be enumerated by the nested loop:

   for $k_A = 0\ ,\ 2$
   for $i = \mathbf{max}(500{*}k_A, 1)\ ,\ \mathbf{min}(500{*}k_A{+}499, 1000)$
   for $j = i\ ,\ 2{*}i{+}1$

which shows that the blocks 3..7 of A are not written in during the computation.

3. Finally, a guard is inserted to produce the SPMD computation code:

for $k_A = 0\ ,\ 2$
  if *myself* = owner of block k_A of A **then**
    for $i = \mathbf{max}(500{*}k_A, 1)\ ,\ \mathbf{min}(500{*}k_A{+}499, 1000)$
      for $j = i\ ,\ 2{*}i{+}1$
        $A[\,i\ ,\ j{-}i\,] := B[\,j\ ,\ 2{*}i{-}2\,]$

4.4 Compilation scheme description

Given a parameterized polyhedron $P(y) = \{x \in \mathbb{Z}^n \ / \ Ax^T + By^T + c \geq 0\}$ (A, B are integer matrices, c is an integer vector) and a point $x_0 \in \mathbb{Z}^n$, $x_0 \in P(y)$ will denote the inequality system $Ax_0^T + By^T + c \geq 0$.

If $x = (x_1, \ldots, x_n)$ and $y = (y_1, \ldots, y_p)$, (x, y) will designate the point $(x_1, \ldots, x_n, y_1, \ldots, y_p)$. This notation will be extended to an arbitrary number of points.

for I **in** $\mathcal{D}$ will denote a nested loop, I standing for the iteration vector and $\mathcal{D}$ for the iteration domain. Because of the restrictions mentioned in the beginning of the section, it should be noted that the domain $\mathcal{D}$ is a polyhedron.

With this last notation, a commutative loop nest has the following form:

for I **in** $\mathcal{D}$
  $lref \ := \ Exp\,(\mathcal{D}ist \ \biguplus \ \mathcal{R}epl)$

where *lref* is the left hand side (lhs) reference of the assignment, $\mathcal{D}ist$ the set of references to distributed arrays in the expression Exp and $\mathcal{R}epl$ the set of references to replicated variables in Exp.

4.4.1 Definitions

Block family of a distributed array For sake of simplicity, we assume that each lower bound of an array is 0. Let X be a m-dimensional distributed array.

- s_i^X (resp. H_i^X) $(i \in 0..m{-}1)$ will denote the block size (resp. the number of elements) of array X in the i^{th} dimension,
- $\mathcal{DD}(X) = \{i \in 0..m{-}1 \ / \ s_i^X < H_i^X\}$ will stand for the set of distributed dimensions of X.

With these notations, the set of blocks of array X can be characterized by the family:

$$\mathcal{BF}(X) = \{Block(X, (j_i)_{i \in \mathcal{DD}(X)}) \ / \ \forall i \in \mathcal{DD}(X) \quad 0 \leq j_i \leq \lceil H_i^X / s_i^X \rceil - 1\}$$

where $Block(X, (j_i)_{i \in \mathcal{DD}(X)})$ is the block of array X indexed by $(j_i)_{i \in \mathcal{DD}(X)}$; it is the parameterized rectangular tile $lbnd_0 .. ubnd_0 \times \ldots \times lbnd_{m-1} .. ubnd_{m-1}$ with

$$\forall i \in \mathcal{DD}(X) \begin{cases} lbnd_i = j_i\, s_i \\ ubnd_i = min(j_i\, s_i + s_i - 1, H_i - 1) \end{cases}$$

$$\forall i \notin \mathcal{DD}(X) \begin{cases} lbnd_i = 0 \\ ubnd_i = H_i - 1 \end{cases}$$

Polyhedrons for code generation Let $X[\,h(I)\,]$ et $X'[\,h'(I)\,]$ be two references to distributed arrays in the nest assignment. For the generation of data exchange and computation codes, we use the following polyhedrons:

- $\mathcal{P}_1(X[\,h(I)\,], X'[\,h'(I)\,])$ defined as the set of points $((j_i^X)_{i \in \mathcal{DD}(X)},\ (j_i^{X'})_{i \in \mathcal{DD}(X')},\ I)$ where iteration vector I is such that references $h(I)$ and $h'(I)$ respectively reach the block of X indexed by $(j_i^X)_{i \in \mathcal{DD}(X)}$ and the block of X' indexed by $(j_i^{X'})_{i \in \mathcal{DD}(X')}$. More formally:

$$\begin{aligned} \mathcal{P}_1(X[\,h(I)\,], X'[\,h'(I)\,]) \;=\; \{ \quad & ((j_i^X)_{i \in \mathcal{DD}(X)},\ (j_i^{X'})_{i \in \mathcal{DD}(X')},\ I) \;\; / \\ & \forall i \in \mathcal{DD}(X) \quad 0 \le j_i^X \le \lceil H_i^X / s_i^X \rceil - 1, \\ & \forall i \in \mathcal{DD}(X') \quad 0 \le j_i^{X'} \le \lceil H_i^{X'} / s_i^{X'} \rceil - 1, \\ & I \in \mathcal{D}, \\ & h(I) \in Block(X, (j_i^X)_{i \in \mathcal{DD}(X)}), \\ & h'(I) \in Block(X', (j_i^{X'})_{i \in \mathcal{DD}(X')}) \\ \} & \end{aligned}$$

- $\mathcal{P}_2(X[\,h(I)\,])$ defined as the set of points $((j_i^X)_{i \in \mathcal{DD}(X)},\ I)$ where iteration vector I is such that reference $h(I)$ reaches the block of X indexed by $(j_i^X)_{i \in \mathcal{DD}(X)}$:

$$\begin{aligned} \mathcal{P}_2(X[\,h(I)\,]) \;=\; \{ \quad & ((j_i^X)_{i \in \mathcal{DD}(X)},\ I) \;\; / \\ & \forall i \in \mathcal{DD}(X) \quad 0 \le j_i^X \le \lceil H_i^X / s_i^X \rceil - 1, \\ & I \in \mathcal{D}, \\ & h(I) \in Block(X, (j_i^X)_{i \in \mathcal{DD}(X)}) \\ \} & \end{aligned}$$

4.4.2 Compilation scheme when *lref* refers to a distributed array

Let us note $A[\,f(I)\,]$ the lhs reference and $\mathcal{C}om = \mathcal{D}ist - \{lref\}$ the set of references in $\mathcal{D}ist$ that *may* generate communications between processors (it is clear indeed that, if *lref* belongs to $\mathcal{D}ist$, this reference does not lead to any interprocessor communication). It should be noted that the set $\mathcal{C}om$ can be even more reduced if some references in $\mathcal{D}ist$ are aligned (in a High Performance Fortran [For93] manner) with $A[\,f(I)\,]$.

Data exchange code generation For each $B_k[\,g_k(I)\,]$ in $\mathcal{C}om$:

1. compute the enumeration code of polyhedron $\mathcal{P}_1(A[\,f(I)\,], B_k[\,g_k(I)\,])$ by one of the methods mentioned in 4.1 and 4.2:

   for $(j_i^A)_{i\in\mathcal{DD}(A)}$ **in** $\mathcal{D}_1$

   for $(j_i^{B_k})_{i\in\mathcal{DD}(B_k)}$ **in** $\mathcal{D}_2$

   for I **in** $\mathcal{D}_3$

2. produce the SPMD send code:

   for $(j_i^A)_{i\in\mathcal{DD}(A)}$ **in** $\mathcal{D}_1$

   if *myself* $\neq$ owner of $Block(A, (j_i^A)_{i\in\mathcal{DD}(A)})$ **then**

   for $(j_i^{B_k})_{i\in\mathcal{DD}(B_k)}$ **in** $\mathcal{D}_2$

   if *myself* $=$ owner of $Block(B_k, (j_i^{B_k})_{i\in\mathcal{DD}(B_k)})$ **then**

   for I **in** $\mathcal{D}_3$

   send $B_k[\,g_k(I)\,]$ to the owner of $Block(A, (j_i^A)_{i\in\mathcal{DD}(A)})$

3. produce the dual SPMD receive code:

   for $(j_i^A)_{i\in\mathcal{DD}(A)}$ **in** $\mathcal{D}_1$

   if *myself* $=$ owner of $Block(A, (j_i^A)_{i\in\mathcal{DD}(A)})$ **then**

   for $(j_i^{B_k})_{i\in\mathcal{DD}(B_k)}$ **in** $\mathcal{D}_2$

   if *myself* $\neq$ owner of $Block(B_k, (j_i^{B_k})_{i\in\mathcal{DD}(B_k)})$ **then**

for I **in** $\mathcal{D}_3$

receive $B_k[\,g_k(I)\,]$ from the owner of $Bloc(B_k, (j_i^{B_k})_{i \in \mathcal{DD}(B_k)})$

For sake of simplicity, we have described here the communications elementwise. In practice, the array elements are combined into vectors before being sent.

Computation code generation

1. compute the enumeration code of polyhedron $\mathcal{P}_2(A[\,f(I)\,])$:

   for $(j_i^A)_{i \in \mathcal{DD}(A)}$ **in** $\mathcal{D}_4$

   for I **in** $\mathcal{D}_5$

2. produce the SPMD computation code:

   for $(j_i^A)_{i \in \mathcal{DD}(A)}$ **in** $\mathcal{D}_4$

   if *myself* = owner of $Block(A, (j_i^A)_{i \in \mathcal{DD}(A)})$ **then**

   for I **in** $\mathcal{D}_5$

   $A[\,f(I)\,] \;:=\; Exp\,(\mathcal{D}ist \uplus \mathcal{R}epl)$

4.4.3 Compilation scheme when *lref* refers to a replicated variable

Data exchange code generation For each $B_k[\,g_k(I)\,]$ in $\mathcal{D}ist$:

1. compute the enumeration code of polyhedron $\mathcal{P}_2(B_k[\,g_k(I)\,])$:

   for $(j_i^{B_k})_{i \in \mathcal{DD}(B_k)}$ **in** $\mathcal{D}_1$

   for I **in** $\mathcal{D}_2$

2. produce the SPMD send code:

   for $(j_i^{B_k})_{i \in \mathcal{DD}(B_k)}$ **in** $\mathcal{D}_1$

   if *myself* = owner of $Block(B_k, (j_i^{B_k})_{i \in \mathcal{DD}(B_k)})$ **then**

   for I **in** $\mathcal{D}_2$

   broadcast$(B_k[\,g_k(I)\,])$

3. produce the dual SPMD receive code:

for $(j_i^{B_k})_{i \in \mathcal{DD}(B_k)}$ **in** $\mathcal{D}_1$

if *myself* $\neq$ owner of $Block(B_k, (j_i^{B_k})_{i \in \mathcal{DD}(B_k)})$ **then**

for I **in** $\mathcal{D}_2$

receive$(B_k[\, g_k(I)\,])$

The broadcast described here is performed elementwise; in practice, vectors of consecutive elements (in the enumeration) are broadcasted.

Computation code generation The nested loop is replicated on all the processors:

for I **in** $\mathcal{D}$
  $lref \;:=\; Exp\,(\mathcal{D}ist \uplus \mathcal{R}epl)$

4.5 Compiling parameterized loops

The method previously presented allows to compile "parameterized" commutative loop nests, that is to say commutative loop nests located in the body of one or many surrounding loops. If we note K (resp. $\mathcal{C}$) the iteration vector (resp. iteration domain, that we assume to be a polyhedron) associated with these enclosing loops, a parameterized commutative loop nest has the following form:

for I **in** $\mathcal{D}(K)$
  $lref \;:=\; Exp\,(\mathcal{D}ist \uplus \mathcal{R}epl)$

where $\mathcal{D}(K)$ is a polyhedron parameterized by K. The SPMD code is generated in the same way, polyhedrons $\mathcal{P}_1$ and $\mathcal{P}_2$ being now parameterized by K:

$$\begin{aligned}\mathcal{P}_1(X[\,h(K,I)\,], X'[\,h'(K,I)\,]) \;=\; \{\quad & ((j_i^X)_{i\in\mathcal{DD}(X)},\ (j_i^{X'})_{i\in\mathcal{DD}(X')},\ I) \quad / \\ & \forall i \in \mathcal{DD}(X) \quad 0 \le j_i^X \le \lceil H_i^X / s_i^X \rceil - 1, \\ & \forall i \in \mathcal{DD}(X') \quad 0 \le j_i^{X'} \le \lceil H_i^{X'} / s_i^{X'} \rceil - 1, \\ & I \in \mathcal{D}(K), \\ & h(K,I) \in Block(X, (j_i^X)_{i\in\mathcal{DD}(X)}), \\ & h'(K,I) \in Block(X', (j_i^{X'})_{i\in\mathcal{DD}(X')}) \\ \}\end{aligned}$$

$$\mathcal{P}_2(X[\,h(K,I)\,]) \;=\; \{\;\; ((j_i^X)_{i\in\mathcal{DD}(X)},\; I) \;\;/$$

$$\forall i \in \mathcal{DD}(X) \quad 0 \le j_i^X \le \lceil H_i^X / s_i^X \rceil - 1,$$

$$I \in \mathcal{D}(K),$$

$$h(K,I) \in \mathit{Block}(X, (j_i^X)_{i\in\mathcal{DD}(X)})$$

$$\}$$

and the methods discussed in 4.1 and 4.2 can be straightforward extended to compute the enumeration code of these parameterized polyhedrons in the context $K \in \mathcal{C}$.

5 Conclusion

The compilation scheme described in this paper has been implemented in the Pandore II compiler. Preliminary experiments show the efficiency of the generated code but also the need of an efficient management of temporary storage for distant array elements. A memory management based on a paging mechanism which tries to balance the speed of accesses and the memory requirements is studied. An improvement of the compilation method, that takes into account at compile-time the mapping of the blocks, is also investigated.

References

[BCJT93] C. Bareau, B. Caillaud, C. Jard, and R. Thoraval. Correctness of automated distribution of sequential programs. In A. Bode, M. Reeve, and G. Wolf, editors, *PARLE'93, Parallel Architectures and Languages Europe*, pages 517–528, Springer Verlag, June 1993.

[BCZ92] P. Brezany, B.M. Chapman, and H.P. Zima. *Automatic Parallelization for GENESIS.* Technical Report ACPC/TR 92-16, Austrian Center for Parallel Computation, November 1992.

[CFR93] J.F. Collard, P. Feautrier, and T. Risset. *Construction of DO Loops from Systems of Affine Constraints.* Technical Report 93-15, LIP, Lyon, France, 1993.

[Che87] M.C. Cheng. General criteria for redundant and nonredundant linear inequalities. *Journal of Optimization Theory and Applications*, 53(1):37–42, April 1987.

[Che93] O. Chéron. *Pandore II : un compilateur dirigé par la distribution des données.* PhD thesis, IFSIC/Université de Rennes I, July 1993.

[Duf74] R.J. Duffin. On fourier's analysis of linear inequality systems. *Mathematical Programming Study*, 1:71–95, 1974.

[Fea89] P. Feautrier. Semantical analysis and mathematical programming. application to parallelization and vectorization. In M. Cosnard and al., editors, *Parallel and Distributed Algorithms*, pages 309–320, Elsevier Science Publishers B.V. (North Holland), 1989.

[FM90] P. Feautrier and Werth M.R. A systematic approach to program transformations. In *International Workshop on Compilers for Parallel Computers*, pages 117–129, Paris, December 1990.

[For93] High Performance Fortran Forum. *High Performance Fortran Language Specification.* Technical Report Version 1.0, Rice University, May 1993.

[HKT92] S. Hiranandani, K. Kennedy, and C.W. Tseng. Compiling Fortran D for MIMD Distributed-Memory Machines. *Communications of the ACM*, 35(8), August 1992.

[IA91] F. Irigoin and C. Ancourt. Scanning polyhedra with do loops. In *Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 39–50, April 1991.

[Sch86] A. Schrijver. *Theory of Linear and Integer Programming. Wiley-Interscience Series in Discrete Mathematics and Optimization*, John Wiley and Sons, 1986.

Unité de recherche INRIA Lorraine, Technôpole de Nancy-Brabois, Campus scientifique,
615 rue de Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 46 avenue Félix Viallet, 38031 GRENOBLE Cedex 1
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105,
78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS
Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
(France)
ISSN 0249-6399