Semantical Analysis and Mathematical Programming Application to Parallelization and Vectorization

Paul Feautrier*
Laboratoire PRiSM,
Université de Versailles St-Quentin
45 Avenue des Etats-Unis
78035 VERSAILLES CEDEX FRANCE

October 1988

Abstract

This paper investigates a new algorithm for solving systems of linear inequalities in the presence of integer parameters. The applications are to various problems in the analysis of scientific programs. We give methods for computing dependences, for data-flow analysis and for several code generation questions. These techniques all are relevant to the automatic and semi-automatic construction of programs for parallel and vector super-computers.

1 Introduction

It is a well known fact that scientific programs spend most of their running time in executing loops operating on arrays. Hence if a restructuring compiler

*e-mail : Paul.Feautrier@prism.uvsq.fr

is to be a success, it must be able to do a very thorough analysis of the adressing patterns in such loops. If taken in full generality, the problem is intractable. In this paper, we delimit a class of programs for which this analysis is possible: programs with so-called static control and linear indices. There are reasons to believe that a large proportion of all numerical programs belongs to this class, and that many more may be converted to this form by appropriate preprocessing. The analysis of adressing patterns in this class may be reduced to the solution of parametric systems of linear inequalities in integers, for which we have devised an efficient algorithm in [Fea88b].

The first problem we will explore is the calculation of the data dependence relation. Beyond the construction of the dependence graph as used for instance by [Kuc78] or [AK82], our technique gives a precise criterium for loop interchange and allows the exact computation of direct dependences as defined in [Bra88]. In favourable cases, such dependences are regular and may be described by a set of vectors which spans the dependence cone [Iri87].

We will then study data-flow analysis; here, our aim is to find the source of the values which are used at each stage of the computation. Such information is very useful for array expansion [Fea88a], for verifying program correctness, etc.

Lastly, we will show how to solve the problem of enumerating the integer vectors which lies inside a polyhedron. This technique may be applied to code generation problems (loop interchange, supernode construction, etc.) and to memory management problems.

1.1 Notations

Notwithstanding the fact that most parallelizers use Fortran as a source language, all examples here will be given in Pascal.

Bold letters will denote vectors or vector valued functions; $|\mathbf{a}|$ is the dimension of vector $\mathbf{a}$. $\mathbf{a}[i..j]$ is the subvector of $\mathbf{a}$ built from components i to j. $\mathbf{a}[i]$ is a shorthand for $\mathbf{a}[i..i]$. Familiar operators and predicates like $+$ and $\geq$ will be tacitly extended to vectors. The sign $\ll$ will denote lexical ordering of integer vectors. The remainder operator will be written as % in the C fashion. Large letters will usually denote sets; N will be the set of non-negative integers. If A is a matrix, A_{ij} will be its generic element, $A_{i\bullet}$ its generic row and $A_{\bullet j}$ its generic column.

2 The program model

In this chapter, we delimit the set of programs to which our methods are applicable. We will distinguish in the sequel between *instructions*, which are syntactical parts of the program text, and *operations* which are actions inducing modifications of the computer store. Most often, an instruction will be executed several times, giving rise to as many distinct operations.

2.1 Static control programs

To recognize a static control program, one must first identifies its *structure parameters*: a set of integer variables which are defined only once in the program, the defining value depending only on the outside world (throught an input instruction) or on other already defined structure parameters.

Secondly, when the structure parameters are known, one must be able to enumerate the operations which will be executed when the program is run. If the program uses only `if-then-else`, `for` loops and procedure calls as control instructions, this imply that the predicate of an `if` or the bounds of a `for` loop may depend only on structure parameters and enclosing loop counters, and that the same restrictions should apply to all procedures. In the sequel, we will consider the special case where `for` is the only control instruction. Many programs may be brought to this form by preprocessing; the case of the `while` loop is an important exception and will be the subject of future studies. In a program in which the `for` loop is the only control instruction, naming an operation is easy; one only has to give the name of the instruction of which the operation is an instance, and the values of the surrounding loops counters. Such a tuple forms what we have called in [Fea88a] an (operation) coordinate.

2.2 Linearity conditions

The techniques we have devised are applicable only to linear problems. To be able to use them, we will suppose that our object programs obey the following restrictions:

- The loop bounds are integral affine functions of the structure parameters and of enclosing loop counters. This insure that the loop body

is executed for all integer vectors inside a polyhedron whose equations are easily retrieved from the program text; if $\{r, \mathbf{a}\}$ is the coordinate of an operation which belong to a given loop nest, the constraints on $\mathbf{a}$ will be written:

$$\mathbf{e}_r(\mathbf{a}) \geq 0 \tag{1}$$

where $\mathbf{e}_r$ is an affine function of $\mathbf{a}$. (1) is the existence predicate of operation $\{r, \mathbf{a}\}$.

- The indices of arrays are affine functions of the surrounding loop counters and the structure parameters.

As is customary, we will suppose that all arrays are non overlapping and that all indices are within the array bounds.

2.3 The sequencing predicate

When given two coordinates $\{r, \mathbf{a}\}$ and $\{s, \mathbf{b}\}$, knowing which one will be executed first is an important information. We have shown in [Fea88a] that if N_{rs} is the number of loops which enclose both r and s, and if T_{rs} is true iff r precedes s in the text of the program, then the execution order of $\{r, \mathbf{a}\}$ and $\{s, \mathbf{b}\}$ is given by the sequencing predicate:

$$\{r, \mathbf{a}\} \prec \{s, \mathbf{b}\} = \mathbf{a}[1..N_{rs}] \ll \mathbf{b}[1..N_{rs}] \vee (\mathbf{a}[1..N_{rs}] = \mathbf{b}[1..N_{rs}] \wedge T_{rs}). \tag{2}$$

This is not linear, but may be split into $N_{rs} + 1$ linear predicates by replacing the lexicographical order by its familiar definition. The sequencing predicate at depth p, $(p = 0, N_{rs} - 1)$ is:

$$\{r, \mathbf{a}\} \prec_p \{s, \mathbf{b}\} = \mathbf{a}[1..p] = \mathbf{b}[1..p] \wedge \mathbf{a}[p+1] < \mathbf{b}[p+1], \tag{3}$$

while the version for $p = N_{rs}$ is:

$$\{r, \mathbf{a}\} \prec_p \{s, \mathbf{b}\} = \mathbf{a}[1..N_{rs}] = \mathbf{b}[1..N_{rs}] \wedge T_{rs}. \tag{4}$$

When dealing with program transformations, we will select one particular coordinate system which will remain fixed. The transformations will be

seen as modifications of the sequencing predicate. The distinguished coordinate system will usually be the one associated with the original sequential program.

We will use as example, in the sequel, the following version of the Gauss-Jordan elimination algorithm (declarations omitted):

```
read(n);
for i := 0 to n do
begin
    for j := 0 to i-1 do
        for k := i+1 to i+n do
{1}         a1[j,k] = a2[j,k] - a3[i,k] * a4[j,i] / a5[i,i];
    for j := i+1 to n do
        for k := i+1 to i+n do
{2}         a6[j,k] = a7[j,k] - a8[i,k] * a9[j,i] / a10[i,i];
end
```

We have indexed all references to `a` in the interest of clarity. `n` is the only structure parameter. The existence predicate for instances of instruction 1 is:

$$\mathbf{e}_1(i,j,k) = (0 \leq i \leq n) \wedge (0 \leq j \leq i-1) \wedge (i+1 \leq k \leq n).$$

The sequencing between an instance of 1 and an instance of 2 is given by:

$$\{1,i,j,k\} \prec \{2,i',j',k'\} = i < i' \vee i = i',$$

since $N_{12} = 1$ and $T_{12} =$ **true** .

3 The dependence computation

The above version of the sequencing predicate defines a total order, and, as such, reflects only the execution of a sequential program. Now, there are many parallel architectures and many styles of parallel programming. To each such style corresponds a particular structure for the sequencing predicate. In macrotasking style, the sequencing predicate is total for operations which belong to the same task, and is explicitly given by a finite dag for

different tasks. In a synchronous machine (a SIMD or VLIW machine or a systolic array), there is a timing function T (which may be extracted from the program or microprogram text) giving the date of execution of any operation; the squencing predicate is simply:

$$x \prec y = T(x) < T(y).$$

In general, the result of a computation depends on both the set of operations and the sequencing predicate. In most programs, however, there is a certain degree of freedom: the result is the same as the sequential one as long as certain operations are properly ordered. Two operations are dependent if their original sequencing must not be modified for the program to give the proper result. The dependence relation is a preorder and its transitive closure is the coarsest sequencing predicate such that the parallel program gives the same result as the sequential one.

Parallelization, may be seen as a two step process:

1. Determine the dependence relation;

2. For a given architecture (and hence for a given class of sequencing predicate) find the coarsest sequencing which is finer than the dependance relation.

Here again, the calculation of the true dependence relation is a very difficult task, which may involve complicated mathematical problems. Most often, one computes only the syntactical dependence relation, which is given by [Ber66] conditions.

3.1 The basic techniques and some extensions

By Bernstein's conditions, two operations are dependent if they access at least one common memory location, and one of these accesses is a write. Dependences may be classified according to the position of the write operation. If the write is executed first, one has a Producer-Consummer (PC) dependence. If executed last, one has a Consummer-Producer (CP) dependence, and finally a Producer-Producer (PP) dependence if both operations are writes. In general, the set of locations which are accessed by an operation depends both on the corresponding instruction and the current memory

state, throught the value of indices and pointers. In our program model, the situation is much simpler because the influence of the memory state is summarized by the values of the loop counters as embodied in the operation coordinates. Let $\mathtt{A}[\mathbf{f}(\mathbf{a})]$ and $\mathtt{A}[\mathbf{g}(\mathbf{b})]$ be two references to array `A` by operations $\{r, \mathbf{a}\}$ and $\{s, \mathbf{b}\}$. These array elements refer to the same memory location iff:

$$\mathbf{f}(\mathbf{a}) = \mathbf{g}(\mathbf{b}). \tag{5}$$

The dependence is from r to s iff:

$$\{r, \mathbf{a}\} \prec \{s, \mathbf{b}\}. \tag{6}$$

Finally, $\{r, \mathbf{a}\}$ and $\{s, \mathbf{b}\}$ are valid coordinates iff:

$$\mathbf{e}_r(\mathbf{a}) \geq 0 \wedge \mathbf{e}_s(\mathbf{b}) \geq 0. \tag{7}$$

Formulae (5) to (7) give a complete specification of the dependence relation of the object program. (5) and (7) are systems of linear constraints. If one goes back to (3), one sees that (6) splits into $N_{rs} + 1$ different linear systems. Each such system starts with p equalities ($p = 0, \ldots, N_{rs}$) and gives rise to the depth p dependences according to the definition of [AK82].

3.1.1 Computing the dependence graph

For most parallelization or vectorization algorithms, one is interested only in the existence of at least one dependence between r and s at a given depth p. This is equivalent to deciding whether (5)–(7) has solutions in integers. This may be done by several integer programming algorithms, which were pioneereed by Gomory ([Gom63]). This technique is used in the parallelizer PAF ([TDF87]); a variation has been proposed in [Wal88].

In the computation of dependences, wrong decision are harmless (at least with respect to program correctness) provided they are always taken conservatively: deciding there is a dependence when in fact there is not. This remark may be applied in several ways:

- One may solve (5)–(7) in rationals by the Fourier-Motzkin elimination algorithm or by the simplex method;

- If the indexing function and/or loop bounds contains non-linear terms or variables beyond the loop counters, one may consider them as supplementary unknowns in the resolution process in the manner of [LT88].
- Lastly, one may use approximate decision methods like [Ban79] tests.

The Gauss-Jordan example contains some eighty potential dependences. Most of these are proved spurious by our technique. The remaining 18 real dependences are all at depth 0.

3.1.2 Loop Interchange

When restructuring a perfect loop nest for parallel or vector execution, one is often interested in deciding whether two loops (say at level i and $i+1$) may be interchanged. In accordance with a remark at the end of 2.3, this transformation will be seen as a modification of the sequencing predicate. In this case, let s be the loop body; two executions of s are sequenced according to:

$$\{s, \mathbf{a}\} \prec \{s, \mathbf{b}\} = \mathbf{a} \ll \mathbf{b}, \tag{8}$$

After interchange, the new sequencing predicate $\prec_c$, is given by:

$$\{s, \mathbf{a}\} \prec_c \{s, \mathbf{b}\} = P_{i,i+1}\mathbf{a} \ll P_{i,i+1}\mathbf{b}, \tag{9}$$

where $P_{i,i+1}$ is a permutation matrix. Now, this transformation will be correct if all operations whose execution order is reversed:

$$\{s, \mathbf{a}\} \prec \{s, \mathbf{b}\} \wedge \{s, \mathbf{a}\} \prec_c \{s, \mathbf{b}\},$$

or

$$\mathbf{a} \ll \mathbf{b} \wedge P_{i,i+1}\mathbf{b} \ll P_{i,i+1}\mathbf{a}, \tag{10}$$

are independent. Let k be the leftmost place at which $\mathbf{a}$ differs from $\mathbf{b}$. It is easy to see that for all value of k with $k = i$ excepted, $(\mathbf{a} \ll \mathbf{b})$ and $(P_{i,i+1}\mathbf{b} \ll P_{i,i+1}\mathbf{a})$ have the same value. Hence $k = i$ and the critical operations are given by:

$$\mathbf{a}[1..i-1] = \mathbf{b}[1..i-1], \mathbf{a}[i] < \mathbf{b}[i], \mathbf{b}[i+1] < \mathbf{a}[i+1].$$

We conclude that loops i and $i+1$ may be interchanged if there is no dependence at depth $i-1$ which satisfies the added constraint $\mathbf{b}[i+1] < \mathbf{a}[i+1]$; this is again a linear inequality problem. This criterium for loop interchange is more comprehensive than the one of [AK84], since it does not depends on the possibility of defining direction vectors. Note that if there are no dependences at depth i, the criterium above is trivially satisfied, and loop i is parallel. This result gives the following corollary ([PK87]): a perfect loop nest may always be rewritten with the sequential loops outermost. The reasoning may be extended to non-consecutive loops and to other kinds of loop restructuring (e.g. the choice of a timing function for a systolic array).

3.2 Direct dependences

The dependence relation often contains redundant edges, i.e. edges which could be regenerated by transitivity. It would be interesting to eliminate all such edges, but this is a quite difficult problem. A first step in this direction is the determination of direct dependences, a notion which was introduced in [Bra88]. Remember that the dependence relation is the union of all sets:

$$\begin{aligned} \mathsf{Q}(r,s,\mathtt{A},p) = \{ < \{r,\mathbf{a}\},\{s,\mathbf{b}\} > \quad | \quad & \mathbf{f}(\mathbf{a}) = \mathbf{g}(\mathbf{b}), \\ & \{r,\mathbf{a}\} \prec_p \{s,\mathbf{b}\}, \\ & \mathbf{e}_r(\mathbf{a}) \geq 0, \mathbf{e}_s(\mathbf{b}) \geq 0\} \end{aligned} \tag{11}$$

for all instructions r, s and all depths p not greater than N_{rs} and all array references $\mathtt{A}$ which are modified in at least one of r and s. The direct dependences are obtained simply by removing all redundant edges from $\mathsf{Q}(r,s,\mathtt{A},p)$.

Theorem D The set of direct dependences in $\mathsf{Q}(r,s,\mathtt{A},p)$ is given by:

$$\mathsf{D}(r,s,\mathtt{A},p) = \{ < \{r,\mathbf{K}(\mathbf{b})\},\{s,\mathbf{b}\} > | \mathbf{e}_s(\mathbf{b}) \geq 0 \}$$

where

$$\begin{aligned} \mathbf{K}(\mathbf{b}) &= \ll \max \mathsf{F}(\mathbf{b}), \\ \mathsf{F}(\mathbf{b}) &= \{\mathbf{a} | \mathbf{f}(\mathbf{a}) = \mathbf{g}(\mathbf{b}), \{r,\mathbf{a}\} \prec_p \{s,\mathbf{b}\}, \mathbf{e}_r(\mathbf{a}) \geq 0\}, \end{aligned} \tag{12}$$

if the dependence is PC, and:

$$\mathsf{D}(r,s,\mathtt{A},p) = \{ < \{r,\mathbf{a}\},\{s,\mathbf{K}(\mathbf{a})\} > | \mathbf{e}_r(\mathbf{a}) \geq 0 \}$$

where

$$\begin{aligned} \mathbf{K}(\mathbf{a}) &= \ll \min \mathsf{F}(\mathbf{a}), \\ \mathsf{F}(\mathbf{a}) &= \{\mathbf{b} | \mathbf{f}(\mathbf{a}) = \mathbf{g}(\mathbf{b}), \{r, \mathbf{a}\} \prec_p \{s, \mathbf{b}\}, \mathbf{e}_s(\mathbf{b}) \geq 0\}, \end{aligned} \tag{13}$$

if the dependence is PP or CP.

The proof is easy if one notice that, e.g., if $\{r, \mathbf{a}\}$ and $\{r, \mathbf{a}'\}$ both are in PC dependence with $\{s, \mathbf{b}\}$ and $\mathbf{a} \ll \mathbf{a}'$ then $\{r, \mathbf{a}\}$ and $\{r, \mathbf{a}'\}$ are in PP dependence and the result follows by transitivity. The computation of $\mathbf{K}(\mathbf{a})$ is no longer a simple integer programming probleme. The reasons are twofold:

- The feasible set $\mathsf{F}(\mathbf{a})$ is not constant: it depends on integer parameters $\mathbf{a}$ and may even be empty for some values of these parameters. We are interested in the solution as a fonction of $\mathbf{a}$. One may remark also that the parameters are not entirely arbitrary: the solution is interesting only if $\mathbf{a}$ satisfies the existence predicate $\mathbf{e}_r$.
- The elements of the feasible set are not ranked according to a linear cost function, as is customary, but according to the lexicographical order, $\ll$.

The problem may be solved by an extension of Gomory's algorithm, the Parametric Integer Programming (PIP) algorithm of [Fea88b]. A short description may be found in the appendix; the algorithm may be adapted to cope with the computation of the lexical maximum, and for the elimination of one or more variables from a system of constraints.

3.3 Dependence vectors

In the general case, the solution of a PIP is a multilevel conditional with quasi-affine predicates and values. The dependence is said to be regular if $\mathbf{K}(\mathbf{b})$ is a conditional whith one leaf of the form $\mathbf{b} + \mathbf{d}$ ($\mathbf{d}$ a constant vector) all other leaves being $\emptyset$. The vectors $\mathbf{d}$ which are constructed in this way are dependence vectors in the sense of [Lam74]. In fact, from the way they are constructed, they are the generators of the dependence cone as defined in [Iri87]. Dependence vectors are important for the application of algorithms such as the wavefront method [Lam74], for loop partitioning ([Iri87]) and for the automatic construction of systolic arrays ([Qui88]).

In the Gauss-Jordan example, reference 6 generates by itself a PP dependence for which the **K** function is:

$$\textbf{if } (k-i-2 \geq 0)(\textbf{if } (j-i-2 \geq 0)\langle i+1, j, k\rangle \textbf{ else } \emptyset) \textbf{ else } \emptyset,$$

which is regular with the dependence vector $\langle 1, 0, 0\rangle$. On the other hand, references 6 and 5 give rise to a PC dependence for which **K** is $\langle i-1, i, i\rangle$, which is not regular.

4 Data flow analysis

The aim of data flow analysis is to keep track of the values which are generated in the course of the calculation, in contrast to the program itself which is written in term of memory locations. The first step in the analysis is to give a name to each value. We will suppose that all instructions in our language produce one and only one value: a value may be named by the coordinate of the generating operation. The main problem is how to find the name of the values which are used in the right hand side (rhs) of instructions.

4.1 The source computation

A partial answer is given by the results of 3.2. If the source is reference i in the left hand side (lhs) of instruction r, then r and s are in PC dependence, and the source is the latest such operation, i.e. the one which produce the direct dependence from r to s. To each depth p and each possible source i in instruction s_i is associated a function $\mathbf{K}_{ip}$ which is defined by 12. This function is easily converted to a coordinate by inserting instruction names at appropriate places. What we need is an algorithm to compute the maximum of these sources according to the sequencing order, $\prec$. The solution is given by the following set of rewrite rules:

$$\prec \max(x, \emptyset) => x,$$

$$\prec \max(\textbf{if } (p)x \textbf{ else } y, z) => \textbf{if } (p) \prec \max(x, z) \textbf{ else } \prec \max(y, z),$$

$$\prec \max(\{r, \mathbf{a}\}, \{s, \mathbf{b}\}) = \textbf{if } (\{r, \mathbf{a}\} \prec \{s, \mathbf{b}\})\{s, \mathbf{b}\} \textbf{ else } \{r, \mathbf{a}\},$$

and their symmetric counterparts. Note that a direct dependence is in the form of a quasi-affine selection tree. One may see that when starting from such initial data, the rewriting process always terminate, and that the result is again in the correct format. Most often, the result may be further simplified by checking the compatibility of the predicates along each branch of the tree, again by using linear integer programming, and by applying the rule:

$$\textbf{if } (x)y \textbf{ else } y => y.$$

Let us consider reference 2 in the example. There are two possible sources, references 1 and 6. A straightforward application of the above rewriting rules gives a tree with five leaves. Two of these are eliminated by the compatibility test or otherwise simplified, and the final result is:

$$\textbf{if } (i-j-2 \geq 0)\{1, i-1, j, k\} \textbf{ else if } (j-1 \geq 0)\{2, j-1, j, k\} \textbf{ else } \emptyset.$$

4.2 Total expansion

A knowledge of the source of each value in a computation allows one to solve various problems connected to scalar and array expansion and renaming. All such transformation consist in modifying some or all lhs references with the aim of removing PP and CP dependences. "Evaluating" a source is the process of replacing in its leaves all operation coordinates by the corresponding lhs. After one or more references have been expanded or renamed, all rhs references which are in PC dependence with at least one modified lhs must be replaced by the evaluation of their sources in the modified context.

For instance, if in the above Gauss-Jordan program, reference 1 is changed to `a1[i,j,k]` and 6 to `a2[i,j,k]`, then reference 2 becomes[1]:

```
if(i-j-2 >= 0) a1[i-1, j, k]
else if(j -1 >= 0) a2[j-1,j, k]
     else ∅.
```

Note that all renaming and expansion are not legitimate. When one needs a value, one must take care that it has not been overwritten sometime before. The following stategies are safe:

[1]For the meaning of the last term, see 4.3

- Gives a new name to an array or scalar;
- Replace the index list by the counters of all surrounding loops.

If this is done systematically, one gets a single assignment program or equivalently a system of recurrence equations ([Qui88]).

4.3 Program correctness and optimization

The results of the source computation may be used for program checking and improvement. If an instruction occurs in no source, it can be removed ; the process can be iterated until all such dead code is eliminated.

The presence of a $\emptyset$ in a source indicates access to an undefined memory cell. According to the context, this may be taken as an error (if the program is complete) or as the caracterization of an input datum. In the case of the source for reference 2 which was given above, the presence of $\emptyset$ simply means that array `a` must be initialized somewhere else before being used by iteration 0 of the `i` loop.

5 Enumeration problems

We have already said that scientific computations are repetitive: the same instruction or block of instructions is executed many times according to the variation of the surrounding loop counters. In so doing, indices range over subarrays whose caracterization is important for architectures with more than one level of memory. For programs which conform to the model of paragraph 2, all these problems may be expressed as the enumeration of the integers vectors which lie inside a polyhedron. We will first present the general technique then give several applications.

5.1 Enumerating the integer vectors of a polyhedron

Let:

$$\mathsf{B} = \{\mathbf{i} | A\mathbf{i} + \mathbf{b} \in \mathsf{N}, \mathbf{i} \in \mathsf{N}\},$$

be the set of integer vectors belonging to a polyhedron defined by a system of linear inequalities. Vector $\mathbf{b}$ may depend on auxilliary integer parameters. Let n be the dimension of $\mathbf{i}$. The problem is to construct a loop nest:

```
for i1 := a1 to b1 do...
  for in := an to bn do ...
```

such that $\langle i_1, ..., i_n \rangle$ visits all vectors inside B. Let $\mathsf{D}_k(\mathbf{x})$ be the following polyhedron :

$$\mathsf{D}_k(\mathbf{x}) = \{\mathbf{y} | \mathbf{x}\mathbf{y} \in \mathsf{B}\},$$

where $\mathbf{x}$ is of dimension $k-1$ and $\mathbf{y}$ is of dimension $n-k+1$. Compute :

$$\mathbf{u} = \ll \min \mathsf{D}_k(i_1, \ldots, i_{k-1}),$$

$$\mathbf{v} = \ll \max \mathsf{D}_k(i_1, \ldots, i_{k-1}),$$

by algorithms M and N. It easy to prove that the solution to the enumeration problem is given by :

$$a_k = \mathbf{u}[1], b_k = \mathbf{v}[1].$$

Here again the solutions are quasi-affine selection trees. The value $\emptyset$ indicates that the corresponding polyhedron is empty. This is no problem if we extend the semantics of for loops by the convention that a loop with undefined bounds is not executed at all.

The expressions for a_k and b_k may be somewhat simplified if one takes into account the bounds on $i_1, \ldots, i_{k-1}$ which where obtained at the previous steps of the algorithm. This is most easily done if these bounds are affine, by adding them as context when computing $\mathbf{u}$ and $\mathbf{v}$.

5.2 Loop interchange

As a first application of the above method, consider the problem of rewriting a loop nest after one or more loop interchanges. In the case of rectangular loops there is no difficulty. In the general case, each bound may depend on outer loop variables. One simply has to express the loop bounds as linear inequalities, reorder the variables to reflect the new nesting pattern, and apply the above method. Consider for instance the following nest:

```
for i:= 0 to m do
   for j := 0 to n do
      for k := 0 to i+j do
```

which is to be rewritten in the order k, j, i. The first problem to be solved is:

$$\mathsf{D}_1 = \{k, j, i | i \leq m, j \leq n, k \leq i + j\}$$

and the result is $0 \leq k \leq m + n$. The second problem is:

$$\mathsf{D}_2(k) = \{j, i | i \leq m, j \leq n, k \leq i + j\},$$

which is to be solved in the context $k \leq m + n$. The result is:

$$(\textbf{if } (m - k \geq 0)\, 0 \,\textbf{else } k - m) \leq j \leq n,$$

which is by no means obvious. The last problem is :

$$\mathsf{D}_3(k, j) = \{i | i \leq m, j \leq n, k \leq i + j\},$$

in the context $k \leq m + n, j \leq n$. We get the following bounds for i:

$$\left[\begin{array}{ll} \textbf{if} & (j - k \geq 0)\, 0 \\ \textbf{else} & \textbf{if } (m - k + j \geq 0)\, k - m \\ & \textbf{else } \emptyset \end{array}\right] \leq i \leq \left[\begin{array}{l} \textbf{if } (m - k + j \geq 0)\, m \\ \textbf{else } \emptyset \end{array}\right].$$

This result could have been simplified by taking into account the fact that $m - k + j \geq 0$ is a consequence of the bounds on j, but this is, at the present time, beyond the capabilities of our algorithm.

5.3 Region extraction

When using computers with several levels of memory, one has to plan (or analyze) the movements of data between levels. In its most basic form, the problem may be cast in the following terms:

- Consider a loop nest and an array reference in the body of the loop:

```
for i1 := a1 to b1 do
   ...
      for in := an to bn do
{r} ...  A[f(i1,...,in)] ...
```

where $\mathbf{f}$ is an affine multi-dimensional function of $\mathbf{i} = \langle i_1, \ldots, i_n \rangle$ and perhaps other structure parameters.

- Write a copy loop to move the elements of `A` which are in the range of $\mathbf{f}$ from one level of memory to another one. Such a subset of `A` will be called a region.

- The problem is to be solved with varying precision. There is usually no harm done (except to the program run time) if one reads more than necessary. In contrast, writing more than necessary may induce coherence problems, unless special help is provided by the hardware; see [GJG88] for details. The hardware itself may impose restrictions on the shape of the regions which can be handled efficiently.

The first step is to decide if $\mathbf{f}$ is bijective or not. This is done simply by testing for autodependence on the distinguished reference (whether this reference is a read or a write). Similarly, testing for dependence between two distinct references indicates whether the corresponding regions are disjoint or not.

Let $\mathbf{u}$ be a vector of the same dimension as $\mathbf{f}$. The accessed region is the set:

$$\mathsf{R} = \{\mathbf{u} | \exists \mathbf{i} : \mathbf{u} = \mathbf{f}(\mathbf{i}), \mathbf{e}_r(\mathbf{i}) \geq 0\}$$

This set is not necessarily convex. If R is computed by algorithm E in the appendix, the result will be in the form of a boolean expression in disjunctive normal form with quasi-affine predicates. Each litteral will describe a subset of R, these subsets being mutually disjoint. If such a subset is a polyhedron (i.e. if its equations do not use the `div` operator), application of the method of 5.1 will give the required copy code. There are two reasons for the presence of `div` operators. First, all points of R lies on the lattice generated by the column vectors of $\mathbf{f}$. This lattice does not necessarily span all space. The problem may be corrected by computing the Hermitte normal form of $\mathbf{f}$ and using its column vectors as a new base. The second reason is that, due to the integrity condition, the edges of R are not necessarily straight lines. They may exhibit periodic patterns which are described by remainder operators.

Consider the following example, which is adapted from [GJG88]:

```
for i := 0 to l do
  for j := 0 to m do
```

```
for k := 0 to n do
  ... a[3*i+k, j+k] ...
```

In this case, the lattice generated by $\langle 3, 0\rangle$, $\langle 0, 1\rangle$ and $\langle 1, 1\rangle$ spans all space. Let u and v be the components of $\mathbf{u}$. The result of algorithm E in this case (in Lisp notation), is:

```
(or (and (plusp (+ v (* -1 u)))
         (plusp (+ m (* -1 v) u))
         (plusp (+ n (* -1 u))))
    (and (plusp (+ v (* -1 u)))
         (plusp (+ m (* -1 v) u))
         (plusp (+ (* -1 n) u -1))
         (plusp (+ n (* 3 l) (* -1 u)))
         (plusp (+ n m (* -1 v)))
         (plusp (+ (* 3 (div (+ n (* 2 u)) 3)) m (* -1 v) (* -2 )))))
    (and (plusp (+ (* -1 v) u -1))
         (plusp (+ (* 3 l) v (* -1 u)))
         (plusp (+ n (* -1 v)))
         (plusp (+ (* 3 (div (+ v (* 2 u)) 3)) (* -2 u))))
    (and (plusp (+ (* -1 v) u -1))
         (plusp (+ (* 3 l) v (* -1 u)))
         (plusp (+ (* -1 n) v -1))
         (plusp (+ n (* 3 l) (* -1 u))) (plusp (+ n m (* -1 v)))
         (plusp (+ (* 3 (div (+ n (* 2 u)) 3)) m (* -1 v) (* -2 u)))))
```

The first litteral describe the polyhedron

$$\{v \geq u, v \leq u + m, u \leq n\}.$$

The second litteral includes the following predicate:

$$3((n + 2u) \div 3) + m - v \geq 0,$$

which is equivalent to:

$$v \leq m + n - (n + 2u)\%3,$$

and is a description of the characteristic saw-tooth shape of the upper edge of the region. Let us note that this technique is not limited by the nature of loop bounds. It applies equally well to constant, symbolic or variable bounds.

6 Conclusions

We hope that these examples suffice to give an idea of the power and adaptability of the parametric integer programming algorithm. We have found it very robust. The largest example in this paper (paragraph 5.3) involve 3 unknowns, 6 parameters, and 9 constraints. Our C code solved it on a 16-bit microcomputer in about 4". Porting it to a 32-bits mini would allow the speedy solution of quite larger problems.

Some of our results are preliminary in nature and must be further developped, especially as regard total expansion and the simplification of quasi-affine selection trees. There are also a host of new problems which suggest themselves: let us note the construction of independence condition, the construction of array predicates, the resolution of array recurrences, etc.

A last question: is there a way of extending our results to more general programs, in the presence of conditionals, while loops, procedure and function calls?

A The parametric integer algorithm

A.1 The basic algorithm

A parametric integer program (PIP) may be formulated in the following way. Let $\mathsf{F}(\mathbf{z})$ be the set of integer points inside a convex polyhedron:

$$\mathsf{F}(\mathbf{z}) = \{\mathbf{x} | S\mathbf{x} + \mathbf{t}(\mathbf{z}) \in \mathsf{N}\} / K\mathbf{z} + \mathbf{h} \in \mathsf{N}, \tag{14}$$

where S and K are matrices and $\mathbf{t}(\mathbf{z})$ is an integer vector whose components are affine functions of the integer vector $\mathbf{z}$. $\mathbf{z}$ is not arbitrary, but is constrained by the set of inequalities

$$K\mathbf{z} + \mathbf{h} \in N,$$

the *context* of the problem. As a matter of convenience, we will suppose that both S and K are such that they restrict $\mathbf{x}$ and $\mathbf{z}$ to non-negative integer values.

The problem is to decide for which values of $\mathbf{z}$ is $\mathsf{F}(\mathbf{z})$ empty, and if not, to compute its lexical minimum, as a function of $\mathbf{z}$. The solution is given by the following algorithm:

Algorithm N

1. Determine the signs of the components of $\mathbf{t}(\mathbf{z})$ in the context $(K\mathbf{z} + \mathbf{h} \geq 0)$, by solving non-parametric auxilliary integer programs[2];

2. If there is a negative $\mathbf{t}_i(\mathbf{z})$, then either:

   (a) All elements of $S_{i\bullet}$. are negative. In this case, $\mathsf{F}(\mathbf{z})$ is empty, and the solution is written as $\emptyset$;

   (b) There is at least a positive S_{ij}; a pivoting step is executed, giving a new problem $\langle S', \mathbf{t}'(\mathbf{z})\rangle$. The solution of the initial problem is the same as that of the problem $\langle S', \mathbf{t}'(\mathbf{z})\rangle$ in the context $(K\mathbf{z}+\mathbf{h} \geq 0)$; in so doing, keep track of D, the product of the pivots;

3. If all $\mathbf{t}_i(\mathbf{z})$ are positive, select the earliest row i such that $(DS_{ij})\%D$ and $(D\mathbf{t}_i(\mathbf{z}))\%D$ are not identically 0. If no such row exists (in particular if $D = 1$), the solution has been found; it is given by the first $|\mathbf{x}|$ components of $\mathbf{t}(\mathbf{z})$. If such a row exists, let q be a new parameter. Add:

$$0 \leq ((-D\mathbf{t}_i(\mathbf{z}))\%D) - qD \leq D - 1$$

   to the context. Let m be the number of rows in S. Add to the S the new row $m + 1$ with the following coefficients:

$$S_{(m+1)j} = ((DS_{ij})\%D)/D,$$

$$\mathbf{t}_{m+1}(\mathbf{z}) = (-((-D\mathbf{t}_i(\mathbf{z}))\%D)/D) + q,$$

   and start again at step (1).

4. In the remaining case, select a $\mathbf{t}_i(\mathbf{z})$ whose sign is unknown; let $\mathbf{x}_+$ and $\mathbf{x}_-$ be respectively the solutions of $\langle S, \mathbf{t}(\mathbf{z})\rangle$ in the contexts $\{K\mathbf{z} + \mathbf{h} \geq 0, \mathbf{t}_i(\mathbf{z}) \geq 0\}$ and $\{K\mathbf{z} + \mathbf{h} \geq 0, \mathbf{t}_i(\mathbf{z}) < 0\}$. The solution of the initial problem is:

$$\textbf{if } (\mathbf{t}_i(\mathbf{z}) \geq 0)\, \mathbf{x}_+ \textbf{ else } \mathbf{x}_-.$$

[2]For instance, $\mathbf{t}(\mathbf{z}) \geq 0$ if the program $\{K\mathbf{z} + \mathbf{h} \geq 0, \mathbf{t}(\mathbf{z}) < 0\}$ has no solution.

This algorithm is guaranteed to terminate (see [Fea88b]). The result is a multilevel conditional expression whose predicates and leaves are affine functions of the parameters. The new parameters like q above may be replaced by their expressions as integer quotients of affine forms. In this paper, this kind of expression will be called a quasi-affine selection tree.

The algorithm above is not entirely deterministic; there are many equivalent solutions to the same PIP. Experience has shown that a few simple heuristics suffice for selecting a well behaved solution; avoid splitting at all cost (e.g. by grouping the case $\mathbf{t}_i(\mathbf{z}) = 0$ with the positive or negative case if the other does not exist); if forced to split, select a row with all coefficients negative, which implies that $\mathbf{x}_- = \emptyset$. This algorithm has been implemented both in Le_Lisp and C; these codes have been used to run all examples in this paper.

A.2 Some extensions

A.2.1 The lexical maximum

In many cases of interest, one has to compute the lexical maximum rather than a minimum. Sometimes, a transformation from one problem to the other is in evidence. We favour, however, the following systematic procedure.

Algorithm M

Refering back to (14), introduce a new "very large" parameter m and solve:

$$\mathbf{u} = \ll \min \mathsf{G}(\mathbf{z}, m)/K\mathbf{z} + \mathbf{h} \in \mathsf{N},$$

where[3]:

$$\mathsf{G}(\mathbf{z}, m) = \{\mathbf{y} | 0 \leq \mathbf{y} \leq m, -S\mathbf{y} + S\mathbf{1}m + \mathbf{t}(\mathbf{z}) \in \mathsf{N}\}.$$

Compute $\mathbf{v} = m\mathbf{1} - \mathbf{u}$ and prune the solution by replacing all predicates in which m has a positive coefficient by **true** and conversely. A leaf in which m occurs with a positive coefficient is associated to a range of the parameters where $\mathsf{F}(\mathbf{z})$ is unbounded. This case will never occur in the problems we are interested in.

It is easy to prove that $\mathbf{v}$ is the required maximum; it is also easy to devise methods to do the pruning "in line", so as to keep the extra computation

[3] $\mathbf{1}$ is the vector all of whose components are 1.

to a minimum. For instance, in step (1) of algorithm N, if m occurs with a positive sign in $\mathbf{t}_i(\mathbf{z})$, the i-th line may be taken as positive. We have found in practice that in cases where we need to compute both the maximum and the minimum of the same set (see 5.1), both algorithms have operation counts of the same order of magnitude, and neither of them is systematically longer than the other.

A.2.2 Elimination

Let $S\mathbf{x} + \mathbf{t} \in \mathsf{N}$ be a system of linear constraints. Suppose vector $\mathbf{x}$ is split in two parts $\mathbf{x} = \mathbf{yz}$; to eliminate z is simply to compute the predicate:

$$P(\mathbf{y}) = \exists \mathbf{z} : (S\mathbf{yz} + \mathbf{t} \in \mathbf{N}).$$

Now, if $P(\mathbf{y})$ is true, the set:

$$\mathsf{F}(\mathbf{y}) = \{\mathbf{z} | S\mathbf{yz} + \mathbf{t} \in \mathsf{N}\}$$

is not empty and hence, has a lexical minimum. This suggests the following algorithm:

Algorithm E

- Compute $\ll \min \mathsf{F}(\mathbf{y})$ by algorithm N;
- In the resulting conditional, replace $\emptyset$ by **false** and any other value by **true**, and simplify the result.

Elimination may be seen as the projection of the set F on the plane of the un-eliminated variables. If F is taken as a set of rationals , the projection is again a polyhedron, and the Fourier-Motzkin method directly gives the result (see [Tri84]). When working with integers, the projection is no longer convex; as a consequence, the result of algorithm E is not necessarily a conjunction of linear predicates. We will use a simplification method which yields a result in disjunctive normal form.

References

[AK82] J. R. Allen and Ken Kennedy. *PFC : a program to convert FORTRAN to parallel form.* Technical Report MASC-TR82-6, Rice Univ., 1982.

[AK84] J.R. Allen and Ken Kennedy. Automatic loop interchange. In *Proc. of the 1984 ACM SIGPLAN Compiler Conference*, pages 233–246, June 1984.

[Ban79] Utpal Banerjee. *Data dependence in ordinary programs.* Technical Report 79-989, Dept of Comp. Science., University of Illinois at Urbana-Champlain, 1979.

[Ber66] A.J. Bernstein. Analysis of programs for parallel processing. *IEEE Trans. on El. Computers*, EC-15, 1966.

[Bra88] Thomas Brandes. The importance of direct dependences for automatic parallelization. In *ACM Int. Conf. on Supercomputing*, St Malo, France, July 1988.

[Fea88a] Paul Feautrier. Array expansion. In *ACM Int. Conf. on Supercomputing, St Malo*, 1988.

[Fea88b] Paul Feautrier. Parametric integer programming. *RAIRO Recherche Opérationnelle*, 22:243–268, September 1988.

[GJG88] K. Gallivan, William Jalby, and D. Gannon. On the problem of optimizing parallel programs for hierarchical memory systems. In *ACM Int. Conf. on Supercomputing*, St Malo, 1988.

[Gom63] R. E. Gomory. *An algorithm for integer solutions to linear programs.* Mac-Graw Hill, New York, 1963.

[Iri87] François Irigoin. *Partitionnement de boucles imbriquées, une technique d'optimisation pour les programmes scientifiques.* PhD thesis, Université P. et M. Curie, Paris, June 1987.

[Kuc78] David J. Kuck. *The Structure of Computers and Computations.* J. Wiley and sons, New York, 1978.

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

[LT88] Alain Lichnewsky and François Thomasset. Introducing symbolic problem solving techniques in the dependence testing phases of a

vectorizer. In *ACM Int. Conf. on Supercomputing*, St Malo, France, July 1988.

[PK87] Constantine Polychronopoulos and David J. Kuck. Guided self-scheduling. *IEEE Transactions on Computers*, C-36:1425–1439, December 1987.

[Qui88] Patrice Quinton. Mapping recurrences on parallel architectures. In *3rd Int. Conf. on Supercomputing*, Boston, May 1988.

[TDF87] Nadia Tawbi, Alain Dumay, and Paul Feautrier. *PAF : un parallèliseur automatique pour FORTRAN.* Technical Report 185, MASI, 1987.

[Tri84] Rémi Triolet. *Contribution à la parallélisation automatique de programmes Fortran comportant des appels de procédures.* PhD thesis, UPMC, Paris, 1984.

[Wal88] D. R. Wallace. Dependence of multidimensional array references. In *ACM Int. Conf. on Supercomputing*, St Malo, France, 1988.