Fast and efficient Generation of Loop Bounds

Zbigniew Chamski

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Fast and Efficient Generation of Loop Bounds

Zbigniew Chamski

N° 2095

Octobre 1993

PROGRAMME 1

Architectures parallèles,
bases de données,
réseaux et systèmes distribués

ISSN 0249-6399

Fast and Efficient Generation of Loop Bounds

Zbigniew Chamski

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

Rapport de recherche n ° 2095 — Octobre 1993 — 10 pages

Abstract: Current loop generation techniques are based on Fourier-Motzkin pairwaise elimination, which is known to be very memory- and computation-intensive. In this paper we explore an alternative way: the use of parametric linear programming, which allows to separate the computation of distinct loop bounds and leads to a parallel algorithm for loop generation.

Key-words: loop generation, compilation, program transformation, linear programming

(Résumé : tsvp)

À paraître dans les actes de *Parco '93*, Grenoble, France.

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

Une méthode rapide et efficace pour la génération de bornes de boucles

Résumé : Les méthodes de génération de bornes de boucles sont pour la plupart basées sur l'élimination de Fourier-Motzkin. Cependant, cette technique est connue pour sa grande complexité temporelle et spatiale. Dans ce document nous nous intéressons à une approche alternative qui consiste à utiliser la *programmation linéaire paramétrique.* Cette approche permet de séparer les calculs de bornes distinctes et conduit à un algorithme parallèle pour la génération de boucles.

Mots-clé : génération de boucles, compilation, transformation de programmes, programmation linéaire

1 Introduction

The need for loop generation techniques has dramatically grown with the development of parallelizing compilers. The loop generation problem can be stated as follows: "given an iteration domain (a convex polyhedron) D, compute the bounds of a loop nest whose execution will scan all integer points of D in a given order." This problem appears often in parallelization, when the iteration space of the input program was modified using some geometric transformation, and a new loop structure must be generated in order to produce the parallelized program ([1, 8].) It is also faced when generating imperative programs (either parallel or sequential) from formal specifications ([2].)

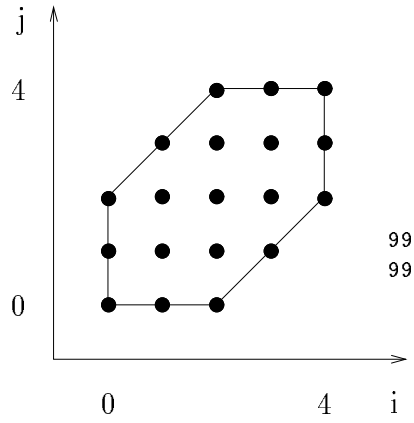
Current solutions ([1, 8]) are based on a well-known linear programming algorithm: the Fourier-Motzkin pairwise elimination. Due to its principle, the computation of loop bounds requires all the bounds to be computed at once. This is a strong limitation for domains with complex definitions, since the memory requirements of this algorithm are bounded by an exponential function of the number m of constraints defining the domain and the dimension d of the iteration space.

Therefore, it becomes important to find a way of dividing the loop generation problem into smaller and possibly independent subproblems. A solution to this was outlined by Feautrier in [5], where the author proposed to use a *parametric* linear programming algorithm to compute loop bounds. However, the exploitation of the results produced by the algorithms he proposed is unpractical due to the size of expressions to be evaluated at run-time. This may be improved to obtain an optimal expression size, as suggested in [2]. Moreover, it is possible to obtain a parallel algorithm for loop generation.

The remainder of the paper is organized as follows: we first illustrate the problem on an example and recall several basic concepts. Section 4 then presents the parametric linear programming algorithm PIP and the structure of its results. In Section 5 we describe the applications of this algorithm to loop generation, and give an efficient parallel algorithm for generating loop bounds. We then discuss the performances and the robustness of this method, and we conclude with directions for future work.

2 An example

To illustrate the problems one faces in loop generation, let us consider a hexagonal tile obtained when partitioning a program (fig. 1*a*.) The inequalities defining this tile are $i - j + 2 \geq 0$, $-i + j + 2 \geq 0$, $i \geq 0$, $j \geq 0$, $i \leq 4$, and $j \leq 4$. If one wants to scan this domain by making j vary before i, then the bounds of i must be constant and those of j may depend on i. The corresponding loop nest is the one shown in figure 1*b*.



a) the input domain

```
      DO 999 I = 0,4
        DO 998 J = MAX(0, I-2), MIN(4, I+2)
      C elementary computation
998     CONTINUE
999   CONTINUE
```

b) the corresponding loop nest

Figure 1: Sample domain: a hexagonal tile

3 Basic concepts

Consider a d-dimensional affine space $\mathcal{S}$. A *convex polyhedron* D is a set of points x of $\mathcal{S}$ satisfying a finite set of constraints $a_i x \geq b_i, i = 1, \ldots, m$. Written in matrix form, these constraints form the lines of an inequality system $Ax \geq b$, hence the notation $D = \{x \mid Ax \geq b\}$.

Consider two vectors $x = (x_1, x_2, \ldots, x_n)$ and $y = (y_1, y_2, \ldots, y_m)$. Vector x is said *lexicographically smaller than* y (noted $x \preceq y$) if there exists a prefix $(x_1, x_2, \ldots, x_p)$ (with $p \leq \min(m, n)$) common to x and y and either $p = \min(m, n)$ or $x_{p+1} < y_{p+1}$. E.g., between the vertices of the hexagonal tile from fig. 1*a* holds the relation $(0,0) \preceq (0,2) \preceq (2,0) \preceq (2,4) \preceq (4,2) \preceq (4,4)$. The lexicographical ordering of any two vectors implies the "$\leq$" ordering of their first coordinates.

A *loop* is described by giving its loop index, lower and upper bounds, and its body. The loop body is a sequence of statements to be executed for every successive value of the loop index, ranging from the lower bound to the upper bound (inclusive). Loops can be nested, i.e., contained in the body of another loop, forming a *loop nest.* A loop nest is said to be *perfect* if for every loop in the nest, its loop body consists of either exactly one loop, or a loop-free sequence of instructions (as in figure 1*a*.)

For a particular execution (or iteration) of a loop, the values of the index of this loop and outer ones, taken from the outermost to the innermost, form a vector called *iteration vector.* The set of the iteration vectors of the innermost loop in a loop nest forms the *iteration space* of the loop nest. The execution of

a loop nest scans the points of its iteration space in ascending lexicographical order.

We are concerned here only with building perfect loop nests. To build such a nest, we need to compute the lower and upper bound for each loop, their bodies being implicitly defined by the perfect loop nest structure (intermediate levels) and the initial program (innermost loop.) Therefore, the loop generation problem can be restated as follows:

> *given an iteration domain D, for every coordinate x_i of points $x \in D$*
> *find the bounds of x_i as a function of the coordinates $x_1, \ldots, x_{i-1}$.*

In this definition, the computations of any two distinct bounds are independent problems and only the choice of the bound generation algorithm may affect their independence.

4 The parametric linear programming

Given a domain $D = \{x \mid Ax \geq 0\}$, we search for a parametric expression of the bounds of x_i as a function of $(x_1, x_2, \ldots, x_{i-1})$. In this sense, our goal is to find the parametric expression of extreme values of x_i such that there exists an $x = (x_1, \ldots, x_i, \ldots, x_d) \in D$. The x_i's satisfying the bounds form a projection on the i-th coordinate of the intersection of D with the hyperplane $H = \{y \mid y_j = x_j,\ 1 \leq j \leq i-1\}$, parameterized by $(x_1, x_2, \ldots, x_{i-1})$.

Since the characterisation of this projection is not directly available, we can search for a lexicographically extreme point of $D \cap H$. This allows us then to find the extrema of x_i, since the lexicographical ordering of vectors implies the standard "$\leq$" ordering of their first coordinates. The corresponding linear programming problem is then to find a lexicographically minimal point of a parametric polyhedron.

The search of an extreme point under a minimization constraint is efficiently performed by the simplex algorithm. However, the parametric aspect and the lexicographical ordering of extreme points must be embedded in the algorithm. This was done by Feautrier, who presented in [4] the PIP (*Parametric Integer Programming*) algorithm which offers, besides other functionalities, a rational parametric simplex algorithm.

In rational mode, PIP computes the lexicographical minimum of a parametric rational polyhedron

$$\mathcal{F}(z) = \{x \mid A_z z + A_y y \geq b, x \geq 0, y \geq 0, z \geq 0\}, \tag{1}$$

possibly under additional constraints on the parameter z, given by a system $Bz \geq c$ called the *context* of the problem.

The constraints on the sign of components of y and z are not restrictive: in a loop nest, only the outermost loop may have infinitely many iterations. By

making its index grow towards infinity we ensure that its lower bound is finite, and by an appropriate translation we can make all loop indices positive.

To apply this algorithm to our problem we can notice that by setting

$$A_z z + A_y y = A \begin{bmatrix} z \\ y \end{bmatrix} \tag{2}$$

$\mathcal{F}(z)$ is the projection on y of the polyhedron $D = \{(z, y) \mid A(z, y) \geq b\}$. Hence, by setting $z = (x_1, x_2, \ldots, x_{i-1})$ and $y = (x_i, \ldots, x_d)$ in equation 1 we can compute the lexicographical minimum (or maximum, see [6]) of $(x_i, \ldots, x_d)$ as a function of $(x_1, x_2, \ldots, x_{i-1})$.

Given our objectives, the PIP algorithm allows us to compute one loop bound per call, leading to the resolution of smaller problems than in the case of the Fourier-Motzkin elimination. Also, since PIP is based on the simplex algorithm, we have a much better behaviour wrt. in terms of space complexity: the simplex is a constant-size algorithm, and the parametric computation leads to a binary tree of recursive calls to the PIP algorithm, whose dimension strictly decreases when the depth of the call grows (see [4].) This tree is explored in a depth-first order and the result is constructed on the fly. Hence, only the current branch of the tree must be held in memory. One can show that its average depth is $\mathcal{O}(\lfloor d/2 \rfloor \log m)$, where m is the number of constraints defining the domain and d is the dimension of the index space. This leads to an average space complexity of $\mathcal{O}(md\lfloor d/2 \rfloor \log m)$.

In comparison, the classical bound on memory requirements of the Fourier-Motzkin elimination is given by $\mathcal{O}(dm^{2^d})$. This bound may be improved by adding redundant constraint removal or history checks. However, the former case implies one linear program to be solved for each redundancy check (e.g., using the nonparametric simplex algorithm) and slows considerably the execution, whereas history checks require an order of m additional storage space for every constraint generated by the algorithm. Therefore, the maximum size of tractable problems will be usually much larger for the PIP algorithm than for the Fourier-Motzkin elimination.

The results produced by PIP are binary selection trees on the parameter vector z, i.e., conditional expressions $E(z)$ of the form

$$\mathit{if}\ cz \geq 0\ \mathit{then}\ E_1(z)\ \mathit{else}\ E_2(z)$$

where E_1 and E_2 are either selection trees or vectors of affine functions on z. The vacuity of $E(z)$ is denoted by the expression "()". This case may arise when no context is specified in the initial problem, as for some values of z there will be no points in D with prefix z.

The leaves of the tree are vectors of affine functions on z, and describe the coordinates of the lexicographical minimum of $\mathcal{F}(z)$ as functions of z. For example, the lower bound of j in the hexagonal tile in figure 1 will be represented as "if $-i + 4 \geq 0$ then if $-i + 2 \geq 0$ then 0 else $i - 2$ endif else () endif". This

means that for i strictly greater than 4 the polyhedron is empty, and otherwise, if i is smaller than 2, the lower bound of j is 0. Finally, for i satisfying $2 < i \leq 4$ the lower bound of j is $i - 2$.

5 Loop generation algorithms based on PIP

In [5], Feautrier suggests to use directly the conditional terms produced by PIP as actual bound expressions. The main idea of the loop generation algorithm is then to compute the conditional terms for all bounds (L_i and U_i, for lower and upper bound of the i-th index, respectively.) He proposes two algorithms, both operating iteratively from the outermost to the innermost index.

The first algorithm does not use contexts, and produces larger bound expressions. A second one is then proposed to reduce the size of bound expressions, and uses the bounds of outer indices as the context when computing the bounds of the next index. The difference in the size of result trees comes from the fact that when no adequate context is used, the algorithm explores values of parameter for which the solution is empty (we measured up to 30% of "()" nodes in the trees computed without context.)

However, we must add here several new arguments to the discussion presented in [5]. First, the mutual independence of distinct bound computations allows to perform the whole loop generation in parallel. Also, the "()" nodes may be ignored, as they correspond to iteration vector prefixes which will never be reached during the execution of the loop nest (see [3]:) the semantics of loop nest execution ensures the enforcement of the context, which consists of the bounds of outer loop indices. Finally, experiments show that avoiding the use of context leads to a more time-efficient execution (see section 6.)

Lets us note "PIP(*extr, parm, var, sys, ct*)" a call to PIP which solves an appropriate extremum problem (according to the value of *extr*) with parameter vector *parm,* variable vector *var*, constraint system *sys* and context *ct,* and let *list-of-functions* be the function which computes the *set* of constraints defining the bounds of x_i. With the above, the "context-free" algorithm of [5] can be written as a parallel one (fig. 2.)

The termination of PIP ([4]) ensures the termination of this algorithm. However, to be used in practice it must be modified in order to give simpler bound expressions. The goal is here to attain the form of expressions produced by Fourier-Motzkin elimination: maxima and minima of several affine functions, defining a lower and an upper bound, respectively. An example of such bounds are the bounds of `J` in figure 1*b*.

The results produced by PIP can be replaced by an appropriate extremum (either minimum or maximum) of a set of functions defining the first coordinate of the corresponding lexicographical extremum ([2].) A formal proof of the validity of this transformation was given by Collard, Feautrier and Risset in [3].

algorithm parallel-PIP

inputs $Ax \geq b$;
outputs $(L_i)_{1 \leq i \leq d}$, $(U_i)_{1 \leq i \leq d}$;
  for i := 1 **to** d **in parallel do**
    L_i := PIP(min, $[x_1, x_2, \ldots, x_{i-1}], [x_i, \ldots, x_d], Ax \geq b, \emptyset$)
  done
    ||
  for i := 1 **to** d **in parallel do**
    U_i := PIP(max, $[x_1, x_2, \ldots, x_{i-1}], [x_i, \ldots, x_d], Ax \geq b, \emptyset$) ;
  done

Figure 2: Parallel bound generation algorithm based on PIP.

As an example, consider the expression of the lower bound of j in the hexagonal tile from figure 1. The conditional term "if $-i+4 \geq 0$ then if $-i+2 \geq 0$ then 0 else $i-2$ endif else () endif" produced by PIP reduces to $\max(0, i-2)$ which is the expected form.

The set of affine functions appearing in a bound expression can be computed using the function `list-of-functions` described above. The improvements obtained in practice are very good: even large trees (several hundreds of nodes) can be reduced to an extremum of a few functions, whose number is of the order of the number of constraints m. Moreover, this set of functions is irredundant in rational numbers, i.e., there is no function implied by other functions in the set. This can be easily shown by noting that each function extracted from the result tree is an actual extremum for some values of the parameters.

By collapsing the conditional bound expressions into extrema of affine functions, we obtain a new form of algorithm `parallel-PIP`, shown in fig 3.

6 Performances

We outline here the results achieved with various PIP-based algorithms. To study practical domains, we rewrote the PIP algorithm using a multiple precision library ([7].) We compared the performance of this implementation with a 32-bit integer version of Fourier-Motzkin elimination used in the PIPS project at École des Mines de Paris.

Sample iteration domains were produced by a prototype of a parallelizing compiler. The corresponding performance measurements are shown in table 1. Memory requirements for PIP-based approaches are given for the worst-case call of each example. Time measurements of parallel PIP stand for the longest subprocess execution, and the cumulated execution time of all subprocesses.

algorithm efficient-parallel-PIP

inputs $Ax \geq b$; **outputs** $(L'_i)_{1 \leq i \leq d}$, $(U'_i)_{1 \leq i \leq d}$;
  for i := 1 **to** d **in parallel do**
    L'_i := max(list-of-functions(PIP(min, $[x_1, x_2, \ldots, x_{i-1}]$, $[x_i, \ldots, x_d], Ax \geq b, \emptyset)$))
  done
    ||
  for i := 1 **to** d **in parallel do**
    U'_i := min(list-of-functions(PIP(max, $[x_1, x_2, \ldots, x_{i-1}]$, $[x_i, \ldots, x_d], Ax \geq b, \emptyset)$)) ;
  done

Figure 3: Algorithm `parallel-PIP` with optimized bound expressions.

We were not able to generate the loop bounds in the last example using Fourier-Motzkin elimination, the computation requiring more than the available memory space.

Table 1: Memory usage comparison between Fourier-Motzkin, context-based PIP and parallel PIP

Domain		Memory (Mbytes)		
dim (d)	constrs (m)	F-M	PIP w/context	parallel PIP
3	8	0.27	0.22	0.25
4	12	0.33	0.35	0.37
6	20	1.32	1.42	1.55
8	28	>50	4.31	5.50

The results show two important points: the moderate requirements of PIP, especially for large computations, and the excellent performance of the algorithm `parallel-PIP`, which even sequentialized is from 5 up to 75% faster than the context-based version from [5].

A comparison of performances of the subprocesses of algorithm `parallel-PIP` (fig. 4) shows that their complexity is an exponential function of the number of parameters (i.e., nesting level,) except for the computation of the innermost index where the actual complexity *decreases.* This allows to partition the computations for small numbers of processors (from two up to eight) with a good load balance. Given these performance results, we are currently working on a quantitative model of the running time of `parallel-PIP` in order to predict

Table 2: Execution time comparison between Fourier-Motzkin, context-based PIP and parallel PIP

Domain		User time (s)			
dim (d)	constrs (m)	F-M	PIP w/ context	parallel PIP	
				max per process	total
3	8	0.23	1.08	0.21	0.74
4	12	4.29	3.80	1.04	3.63
6	20	3457.00	71.64	13.00	58.10
8	28	n/a	911.43	145.63	521.81

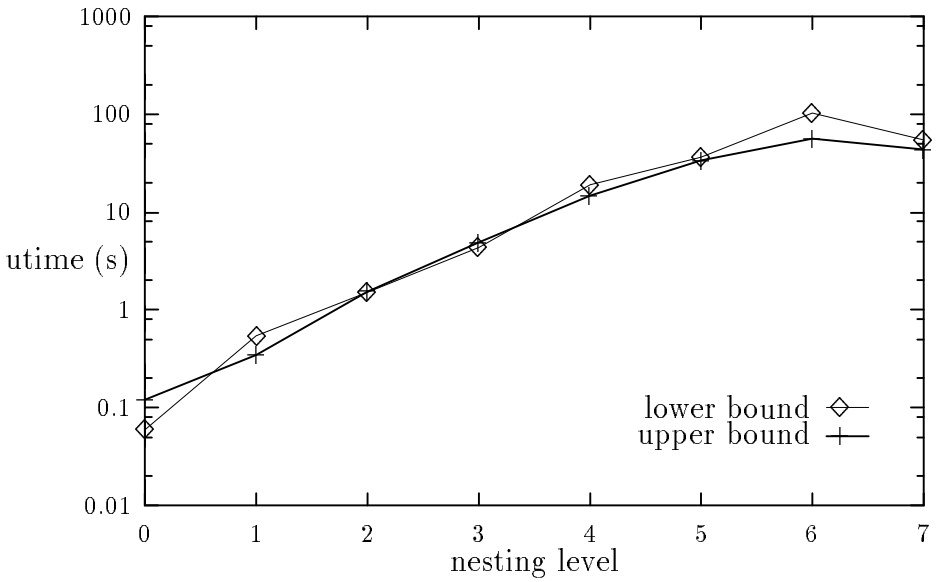


Figure 4: Execution time of subprocesses of `parallel-PIP` ($d = 8,\ m = 28$)

the workload for a given domain and to be able to handle parallel processing resources offered by multiprocessor workstations.

7 Conclusion

The loop generation problem is one of the keys to the generation of programs in parallelization and parallel program transformation. However, existing methods of loop generation, based on Fourier-Motzkin pairwise elimination, suffer from two limitations inherent to this algorithm: an important space complexity and an iterative execution requiring all loop bounds to be computed at once.

In this paper we investigated the practical use of a loop generation method formerly suggested by Feautrier in [5]. His basic idea was to use parametric linear programming tools to compute loop bounds one by one and express them as conditional terms, to be evaluated at run-time. This method subdivides the initial problem into smaller ones leading to an efficient computation, but produces large and inefficient source code.

We explored the performance of a modified approach which produces optimal bound expressions. We also show the possibility of parallelizing the loop generation process, thus reducing the compilation time. From our experiences, PIP-based algorithms appear to be faster and more robust than Fourier-Motzkin elimination. In particular, problems untractable with the latter, due to its space requirements, have been solved with very good execution times.

The simplicity of the modified parallel algorithm makes it very easy to use and implement, and allows to exploit the multiprocessing capabilities of today's parallel workstations. We are currently working on the quantitative modeling of the performances of the algorithm, which should help in fully exploiting such functionalities.

References

[1] C. Ancourt and F. Irigoin. Scanning polyhedra with DO loops. In *ACM/SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pages 39–50, ACM, June 1991.

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

[3] J.-F. Collard, P. Feautrier, and T. Risset. *Construction of DO Loops from Systems of Affine Constraints.* Research Report 93-15, École Normale Supérieure de Lyon, Lyon, France, May 1993.

[4] P. Feautrier. Parametric integer programming. *Recherche Opérationnelle/Operations Research*, 22(3):243–268, 1988.

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

[6] P. Feautrier and N. Tawbi. *Résolution de systèmes d'inéquations linéaires ; mode d'emploi du logiciel PIP.* Technical Report, Laboratoire MASI, Univ. Pierre et Marie Curie, Paris, France, December 1989.

[7] T. Granlund. *GNU MP: The GNU Multiple Precision Arithmetic Library.* Reference manual, Free Software Foundation, Inc., December 1991.

[8] M. J. Wolf and M. S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):452–471, October 1991.

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