Automatic Parallelization in the Polytope Model

Paul Feautrier *

Laboratoire PRiSM,
Université de Versailles St-Quentin
45 Avenue des Etats-Unis
78035 VERSAILLES CEDEX FRANCEŒ

Abstract. The aim of this paper is to explain the importance of polytope and polyhedra in automatic parallelization. We show that the semantics of parallel programs is best described geometrically, as properties of sets of integral points in n-dimensional spaces, where n is related to the maximum nesting depth of `DO` loops. The needed properties translate nicely to properties of polyhedra, for which many algorithms have been designed for the needs of optimization and operation research. We show how these ideas apply to scheduling, placement and parallel code generation.

Résumé

Le but de cet article est d'expliquer le rôle joué par les polyèdres et les polytopes en parallélisation automatique. On montre que la sémantique d'un programme parallèle se décrit naturellement sous forme géométrique, les propriétés du programme étant formalisées comme des propriétés d'ensemble de points dans un espace à n dimensions. n est lié à la profondeur maximale d'imbrication des boucles `DO`. Les propriétés interessantes se traduisent en propriétés de polyèdres, qui peuvent être vérifiées à l'aide de l'un des nombreux algorithmes développés pour les besoins de l'optimisation et de la recherche opérationnelle. On applique ces idées à des problèmes d'ordonnancement, de placement et de génération de code.

1 The geometry of programs

Since the 1990, great progress has been made toward automatic or semi-automatic programming of supercomputers through the use of the *polytope model*. One may wonder what polytopes have to do with programming. The aim of this paper is to answer this question, at least in relation to the field of parallel programs. The polytope model may be used in many different situations, such as for program checking, but these applications still await further developments.

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

Generating a program for a parallel computer is a problem in translation, and, as is the case for all such problems, the better the understanding of the source text and the better the result. Assemblers do word for word translation while early compilers did only "phrase for phrase" translation. Each "part of speech" in the input text was looked up in a dictionary, and the associated translation was substituted with minor modifications. A limited amount of knowledge was then added to improve the final result, for instance in the form of a type system.

> In the case of a fully developed type system, like the one in ML [MTH90], knowledge about operators in the language is given to the compiler in the form of typing rules, which are essentially Horn clauses. A program is correct if, for each of its expressions, one can prove a theorem in the first-order theory generated by the typing rules. This is done quite easily with the help of unification and resolution.

This is the basic scheme for all sophisticated program handling systems. The program, or at least the interesting features of the program, is translated first in an easily manipulable language, like an algebra or a logical theory. Each time one needs non obvious information, one does a calculation or proves a theorem in the underlying system.

Optimizing compilers need much information to decide whether a transformation is allowed or not. The relevant information is related to the flow of control – may a given point in the program be reached from another one – and also to the flow of data – may a given value which has been defined at some point in the program still be used at some other point? Sophisticated techniques have been designed to abstract that kind of information from the program text. Let us note the following characteristics:

- They can be applied to arbitrary programs. In fact, most of them were designed to work on program *flowcharts*.
- They give static information only, i.e. time has become a universally quantified variable in their results. For instance, methods which generate properties of variables gives results of the form "Each time the control reaches a given point, such and such variables stand in the following relation to each other ....".

In the case of a parallel program, the course of events in the calculation is of crucial importance. One has to decide, e.g. if two calculations have to be executed in sequence or can be safely overlapped. One may say, in fact, that constructing a parallel program is equivalent to specifying its *execution order*, i.e. the "is executed before" relation between its operations.

The operations of a program form a set, and its execution order is a binary, transitive and asymmetric relation on the set of operations. For terminating programs, these sets are finite, but in real-life cases, they are too large to be handled in extension, by listing all their members. They must be handled in intension, as the set of solutions of a given system of constraints. For a large proportion of computation intensive programs, the relevant sets are (unions of) Z-polytopes, i.e. sets of integral solutions to systems of affine inequalities. Hence the importance of polytopes for modeling these programs. Fortunately, the theory of polyhedra, polytopes and Z-modules is well developed, as being the basis of linear programming. Practitioners of automatic parallelization have found a ready-made toolchest in Operation Research literature.

In the following, we will first describe how geometrical objects like polytopes may be used to specify the semantics of a certain class of programs. As all powerful techniques, the polytope model has a limited domain of application. We outline in Section 2 the needed constraints on the control statements and the data structures: basically, `DO` loops and linearly indexed arrays. We will then review the needed tools. Most of the optimizations we are interested in may be presented as transformations of the original program. These transformations often are linear or affine or piecewise affine, and hence they transform polytopes into polytopes. Their effect on Z-polytopes is more complicated. Hence, sophisticated techniques have to be used, especially for code generation of the transformed program.

In the conclusion, we assess what has been achieved, and what still need to be improved in the polytope model. Beside that, we try to indicate in which directions the model can be extended and which are the ultimate obstacles to these extensions.

2 Geometrical semantics

Our aim here is to delineate the kind of information a parallelizing compiler needs to do its job. We will show that in cases which cover a large subset of high performance computations, this information can be neatly packaged into Z-polytopes. Most questions the compiler needs answers to relate simply to questions about Z-polytopes, the most important one being the emptiness question.

2.1 Programs as orders

The usual method for defining the semantics of a sequential program is to associate with each elementary construct a function which specifies the transformation of the store which occurs when the construct is executed. For instance, when an assignment is executed, the right hand side is computed in the context of the old store. A new store is then constructed, which is identical to the old one at all locations except the one associated to the left hand side. Transformations associated with more complex constructs are obtained by combining simpler transformations. For instance, the sequence is associated with function composition, and the `while` loop is associated with a fixpoint calculation. The whole technique is called functional or denotational semantics.

Applying this method to parallel programs is not possible, since functional semantics considers as equivalent programs whose behaviour is quite different in a multiprocessing context. `x := x+1; x:= x+2` is equivalent to `x := x+3` in denotational semantics. As processes in a parallel program, the second one may be atomic while the first one is not.

We see that the program representation has to be in term of atomic events or operations, an operation being the execution of one instruction. For most situations, working with high-level statements (e.g., Fortran assignments) is sufficient as a first approach of the problem. We will thus suppose that we are given a set E of operations. As we are interested only in programs which terminate, this set will be supposed finite, albeit much too large to be handled explicitly in practical cases.

As any experienced programmer knows, knowledge of E is not enough to decide what the final result of the program will be. One needs to know the order in which the operations are to be executed. In a sequential program, this order is specified totally by the control statements. In that case, E is a totally ordered set. On a parallel computer, the order of execution of some operations may not be specified, for instance because they are executed independently on asynchronous processors. The execution order is then partial. An actual run of the parallel program is associated to some total extension of this order. There may be several such extensions, with different final results. Parallel programs may not be *determinate*. The problem of automatic parallelization thus boils down to the following scheme:

- We are given a set of operations E and a strict total order on it, $\prec$.
- Find a partial order $\prec_{//}$ on E such that execution of E under it is determinate and gives the same results as the original program.

The simplest case of the parallelization problem is the two-operation program $u; v$. There is only one way of converting it to a parallel program, $u \parallel v$. This program has the empty order, which can be extended to a total order in two ways: $u \prec v$ and $v \prec u$. The first order gives the sequential execution $u; v$ (i.e., the original program), and the second one gives $v; u$. If the parallel program is to be determinate, one must have $u; v = v; u$, in words operations u and v must commute. If they do not, they are said to be dependent, which is written: $u \,\delta\, v$.

This result can be generalized to the *commutation lemma*. A program with a partial order $\prec_{//}$ is deterministic if all operations pairs which are non-comparable by $\prec_{//}$ commute. Suppose we are given a sequential program with order $\prec$. We want to know if the partial order $\prec_{//}$ is valid for this program. As a corollary of the commutation lemma we find that a sufficient condition is that all pairs of dependent operations are ordered in the same way by $\prec$ and $\prec_{//}$:

$$u \prec v \wedge u \,\delta\, v \Rightarrow u \prec_{//} v. \qquad (1)$$

As a consequence, the coarsest valid order is:

$$\prec_{//} = (\prec \cap\, \delta)^{+},$$

where $+$ is the strict transitive closure. The relation $\prec \cap\, \delta$ is known as the detailed dependence graph of the source program.

Notice that this is only a sufficient condition for determinism: we have already sacrificed some parallelism for simplicity. Computing δ may be of arbitrary complexity. However, a sufficient condition for commutation is easily constructed

[Ber66]: let $R(u)$ (resp. $M(u)$) be the set of memory cells which are read (resp modified) by u. u and v commute if:

$$M(u) \cap R(v) = \emptyset, R(u) \cap M(v) = \emptyset, M(u) \cap M(v) = \emptyset.$$

The three terms in that formula appear to be symetrical, but the symetry is broken as soon as we suppose that $u \prec v$. Violation of the first condition is called a *flow dependence*. The other terms correspond to anti- and output dependences.

In the general case, when arbitrary address calculations are allowed, testing Bernstein's conditions may still be quite complicated. Suppose for instance that a, b, c, n are positive integers with $n > 2$. Then the two operations:

$$\mathtt{X}(a^n + b^n) = 0,$$

$$\mathtt{X}(c^n) = 1,$$

are independent if Fermat's Last Theorem holds. We cannot hope a compiler to be able to decide such cases. This has lead to the definition of static control programs, in which the complexity of address calculations is severely restricted.

2.2 Static control programs

The first condition is that the set of operations must be known at compile time. Programs with finite operations sets are quite uninteresting, while potentially nonterminating programs are very difficult to analyse. The middle ground seems to be `DO` loop programs, where the loop bounds depend on symbolic constants (called here *structure parameters*). In that case, the iteration domains are finite, but may have arbitrary size according to the value of the structure parameters. In such a program, an operation is an iteration of a statement, which can be specified by giving the values of the surrounding loop counters. These will be ordered from outside inward and called the *iteration vector*. Iteration $\mathbf{x}$ of statement `S` is $\langle \mathtt{S}, \mathbf{x} \rangle$. To be consistent, loops are also numbered from outside inward, which means that component p of $\mathbf{x}$ is the counter of loop p.

The iteration vector is constrained by the loop bounds. If we suppose that these bounds are affine functions of the surrounding loop counters and structure parameters, then the *iteration domain* of each statement is given by a set of linear inequalities, which have a special form. For a nest of N loops, there are $2N$ inequalities. Inequalities $2k-1$ and $2k$ depend only on the first k components of the iteration vector. As we will see later, it is interesting to generalize to iteration domains which are defined by any number of linear inequalities of whatever form. However, iteration domains should stay finite[2]. Since loop counters are integers, iteration domains are sets of integer vectors inside polytopes, or Z-polytopes. The iteration domain of statement `S` will be written as:

$$\mathrm{Dom}(S) = \{\mathbf{x} \mid D_{\mathtt{S}}\mathbf{x} + d_{\mathtt{S}} \geq 0\},$$

[2] Infinite iteration domains are interesting for online applications and can be handled in special cases.

where D_{S} and d_{S} are the matrix and constant vector which define the iteration polytope. d_{S} may depend linearly on the structure parameters.

The execution order of the operations in a static control program can be deduced from two facts:

- The iterations of a loop nest are executed according to the lexicographic order (noted $\ll$ here) of their iteration vectors.
- All other things being equal, two operations are executed according to their order in the program text, noted as $\lhd$ here.

Let us introduce the following notations:

- $\mathbf{x}[n..m]$ is the subvector of $\mathbf{x}$ built from components n to m. $\mathbf{x}[n]$ is an abbreviation for $\mathbf{x}[n..n]$.
-

  $$\mathbf{x} \ll_p \mathbf{y} \equiv \mathbf{x}[1..p] = \mathbf{y}[1..p] \wedge \mathbf{x}[p+1] < \mathbf{y}[p+1]$$

  and $\ll$ is given by:

  $$\mathbf{x} \ll \mathbf{y} \equiv \bigvee_{p=0}^{N-1} \mathbf{x} \ll_p \mathbf{y}, \tag{2}$$

  where N is the common dimension of $\mathbf{x}$ and $\mathbf{y}$.
- N_{RS} is the number of loops surrounding both `R` and `S`. Accordingly, the number of loop surrounding `S` should be written N_{SS}. It will be abbreviated to N_{S} here.

We have shown in [Fea91] that:

$$\langle \mathtt{R}, \mathbf{x}\rangle \prec \langle \mathtt{S}, \mathbf{y}\rangle \equiv \mathbf{x}[1..N_{\mathsf{RS}}] \ll \mathbf{b}[1..N_{\mathsf{RS}}] \vee (\mathbf{x}[1..N_{\mathsf{RS}}] = \mathbf{y}[1..N_{\mathsf{RS}}] \wedge R \lhd S). \tag{3}$$

The predicate $\prec$ is not convex, hence it cannot be represented as a polyhedron. However, $\prec$ can be split into $N_{\mathsf{RS}}+1$ linear predicates $\prec_p$ as in (2). Each term in the disjunction is then a polyhedron.

We will now restrict data structures to arrays with subscripts which are linear in the structure parameters and the outer loop counters. Furthermore, we will suppose that there is no aliasing – two arrays with differing names refer to non-overlapping regions in memory – and that subscripts stay within the array dimensions. This implies that array accesses stay within the allocated zone, and that the accessed address is in one-to-one correspondence with the subscripts.

Under these hypotheses, to be dependent, two operations must access the same array, and one of them at least must modify it. Let $\mathtt{X}[\mathbf{f}(\mathbf{x})]$ and $\mathtt{X}[\mathbf{g}(\mathbf{x})]$ be the conflicting array accesses. $\mathbf{f}$ and $\mathbf{g}$ are the subscripting functions; they have the same number of components, namely the rank of array `X`. The operations in dependence at depth p are the members of the following *dependence relation* [PW93]:

$$\{\langle \mathtt{R}, \mathbf{x}\rangle, \langle \mathtt{S}, \mathbf{y}\rangle \mid \mathbf{Q}^p_{\mathsf{RS}}(\mathbf{x}, \mathbf{y})\}$$

where $\mathbf{Q}^p_{\mathtt{RS}}$ is the following polytope:

$$\mathbf{Q}^p_{\mathtt{RS}}(\mathbf{x},\mathbf{y}) \equiv \mathbf{f}(\mathbf{x}) = \mathbf{g}(\mathbf{y}) \wedge \langle \mathtt{R}, \mathbf{x}\rangle \prec_p \langle \mathtt{S}, \mathbf{y}\rangle \wedge \mathbf{x} \in \mathcal{D}_{\mathtt{R}} \wedge \mathbf{y} \in \mathcal{D}_{\mathtt{S}} \tag{4}$$

The union of all dependence relations is a symbolic description of the detailed dependence graph. With these notations, (1) translates to:

$$\mathbf{Q}^p_{\mathtt{RS}}(\mathbf{x},\mathbf{y}) \Rightarrow \langle \mathtt{R}, \mathbf{x}\rangle \prec_{//} \langle \mathtt{S}, \mathbf{y}\rangle. \tag{5}$$

As we will see in the next section, these dependence relations can be handled directly in the polytope model. Various researchers have sought to find approximation to them, i.e. simpler polytopes which still enclose $\mathbf{Q}^p_{\mathtt{RS}}$. One possibility is to ignore the dependence on both $\mathbf{x}$ and $\mathbf{y}$ and to consider only dependence distances, i.e to project on the difference $\mathbf{y} - \mathbf{x}$ [DV94]. This has meaning only when statements `R` and `S` have the same iteration space, i.e. belong to the same loop nest. The set of dependence distances can be enclosed in a cone which can be represented by its extremal rays [IT87]. Another possibility is to note only the signs of the components of the extremal rays of the dependence cone, giving the dependence directions. The usual solution is to test each $\mathbf{Q}^p_{\mathtt{RS}}$ for emptiness. If this set is not empty, one supposes that all operations such that $\langle \mathtt{R}, \mathbf{x}\rangle \prec_p \langle \mathtt{S}, \mathbf{y}\rangle$ are in dependence, and one says that there is a depth p dependence from `R` to `S`. An important result is that in a perfect loop nest, if there is no dependence at depth p, then the loop numbered $p+1$ is parallel.

An important special case is that of uniform dependences, in which the set of dependence distances is a singleton:

$$\mathbf{Q}^p_{\mathtt{RS}}(\mathbf{x},\mathbf{y}) \equiv \mathbf{y} = \mathbf{x} + \mathbf{d}. \tag{6}$$

Obviously, since two statements may have many array references and since there are several depths to be considered, there may be many dependence vectors such as $\mathbf{d}$. It is easy to see that all such vectors are lexicopositive.

2.3 The dataflow calculation

Another possibility for simplifying the dependence relation is to remove redundant pairs. The basic technique is best explained on a scalar example:

```
W1 : x = ...
     ...
W2 : x = ...
     ...
R  : u = ... x ...
```

In this program skeleton, there are two flow dependences from `W1` to `R` and from `W2` to `R`, and an output dependence from `W1` to `W2`. It is clear that the first flow dependence is redundant, both in the sense that it can be reconstructed from the other two by transitivity, and also in the sense that the value written into `x` by `W1` never reach `R` since it is killed by `W2`. The set of flow dependences which give rise to a real flow of data constitutes the direct dependences [Bra88] or the value based dependences [PW93]. There are several methods for computing this set: I will describe here the original solution of [Fea88a, Fea91].

Suppose that in the dependence polytope (4), statement $\mathtt{R}$ writes into $\mathtt{X}$ and statement $\mathtt{S}$ reads it. Consider $\mathtt{Q}^p_{\mathtt{RS}}$ as a Z-polytope in $\mathbf{x}$ with parameters $\mathbf{y}$. It is clear that the value written by $\mathtt{R}$ which reaches $\langle \mathtt{S}, \mathbf{y}\rangle$ is the one which is written last according to $\prec$, i.e. has as its iteration vector the lexicographic maximum:

$$K^p_{\mathtt{RS}}(\mathbf{x}) = \max_{\ll}\{\mathbf{x} \mid \mathbf{Q}^p_{\mathtt{RS}}(\mathbf{x}, \mathbf{y})\}. \tag{7}$$

Each statement which writes into $\mathtt{X}$ and each possible depth p give such a potential *source*. The real source is the latest, i.e. their maximum according to $\prec$. We will describe in the next section the tools which are needed for such calculations.

3 Basic tools for handling polyhedra and Z-polyhedra

The basic reference on linear inequalities in rationals or integers is the treatise [Sch86].

3.1 Polyhedra and polytopes

There are two ways of defining a polyhedron. The simplest one is to give a set of linear inequalities:

$$A\mathbf{x} + \mathbf{a} \geq 0.$$

The polyhedron is the set of all $\mathbf{x}$ which satisfies these inequalities. A polyhedron can be empty – the set of defining inequalities is said to be *infeasible* – or unbounded. A bounded polyhedron is called a polytope.

The basic property of a polyhedron is *convexity*: if two points $\mathbf{a}$ and $\mathbf{b}$ belong to a polyhedron, then so do all convex combinations $\lambda\mathbf{a} + (1-\lambda)\mathbf{b}, 0 \leq \lambda \leq 1$. Conversely, it can be shown that any polyhedron can be generated by convex combinations of a finite set of points, some of which – rays – may be at infinity. Any polyhedron is generated by a minimal set of vertices and rays.

There exist non-polynomial algorithms for going from a representation by inequalities to a representation by vertices and rays and vice-versa. Each representation has its merits: for instance, inequalities are better for constructing intersections, while vertices are better for convex unions[3].

The basic algorithms for handling polyhedra are feasibility tests: the Fourier-Motzkin cross-elimination method [Fou90] and the Simplex [Dan63]. The interested reader is referred to the above quoted treatise of Schrijver for details. Both algorithms prove that the object polynomial is empty, or exhibit a point which belongs to it. For definiteness, this point is generally the lexicographic minimum of the polyhedron. In the case of the Fourier-Motzkin algorithm, the

[3] Notice that while the intersection of two polyhedra is a polyhedron, their union is not.

construction of the exhibit point is a well separated phase which is omitted in most cases.

In the Fourier-Motzkin algorithm, one selects a variable and scans all inequalities. If the variable has a positive coefficient, the inequality gives a lower bound for it. Conversely if the coefficient is negative, one gets an upper bound, while if the coefficient is zero, the inequality gives no information on the variable. The variable is eliminated by writing that each of its lower bounds is not greater than each of its upper bounds. At the end of the elimination process, one gets numerical inequalities. If one of them is false, the original system was infeasible. Conversely, if all final inequalities are true, then by going backward into the elimination sequence one can construct a feasible solution.

The asymptotic complexity of the Fourier-Motzkin method is super-exponential. However, it is very easy to program, and experiments have shown that it is very fast for small problems, say of the order of 10 inequalities at most.

Our implementation of the Simplex, PIP [Fea88b] is a geometrical method which can be explained in the following way. Let n be the number of unknowns and m be the number of inequalities in the problem to be solved. One obtains a vertex of a polyhedron by selecting n inequalities, transforming them into equations and solving. The solution point is a real vertex if it satisfies all other inequalities. Otherwise, it is a virtual or external vertex. In the Simplex, one goes from virtual vertices to virtual vertices in the direction of lexicographic increase, until a real one is obtained, or until evidence of unfeasibility has been found. The first real vertex to be found in this way is the lexicographic minimum of the polyhedron. Going from one vertex to the next one is akin to one step of Gaussian elimination, with special rules for the selection of the pivot. The complexity of each step is $O(nm)$, but there can be an exponential number of steps. However, it has been shown that this number has a high probability of being $O(n)$. All in all, the Simplex is faster than Fourier-Motzkin for large problems.

3.2 Z-modules

Let $v_1, \ldots, v_n$ be a set of linearly independent vectors of $\mathbb{Z}^n$ with integral components. The set:

$$\mathcal{L}(v_1, \ldots, v_n) = \{\mu_1 v_1 + \ldots + \mu_n v_n \mid \mu_i \in \mathbb{Z}\}$$

is the Z-module generated by $v_1, \ldots, v_n$. The set of all integral points in $\mathbb{Z}^n$ is the Z-module generated by the canonical basis vectors (the canonical Z-module).

Any Z-module can be characterized by the square matrix V of which $v_1, \ldots, v_n$ are the column vectors. We will use the notation $\mathcal{L}(V)$ for $\mathcal{L}(v_1, \ldots, v_n)$. However, many different matrices may represent the same Z-module. A square matrix is said to be unimodular if it has integral coefficients and if its determinant is ± 1. Let U be a unimodular matrix. It is easy to prove that V and VU generate the same lattice.

Conversely, it can be shown that any non-singular matrix V can be written in the form $V = HU$ where U is unimodular and H has the following properties:

- H is lower triangular,
- All coefficients of H are positive,
- The coefficients in the diagonal of H dominate coefficients in the same row.

H is the Hermite normal form of V. Two matrices generate the same Z-module if they have the same Hermite normal form. The Hermite normal form of a unimodular matrix is the identity matrix, which generates the canonical Z-module.

Computing the Hermite normal form of an $n \times n$ matrix is of complexity $O(n^3)$, provided that the integers generated in the process are of such size that arithmetic operations can still be done in time $O(1)$.

3.3 Z-polyhedra

A Z-polyhedron is the intersection of a Z-module and a polyhedron:

$$F = \{\mathbf{z} \mid \mathbf{z} \in \mathcal{L}(V), A\mathbf{z} + \mathbf{a} \geq 0\}.$$

If the context is clear, and if $\mathcal{L}(V)$ is the canonical Z-module ($V = I$), it may be omitted in the definition.

The basic problem about Z-polyhedra is the question of their emptiness or not. For canonical Z-polyhedra, this is the linear integer programming question [Sch86, Min83]. I will briefly sketch two integer programming algorithm: the Omega test [Pug91a] which is an extension of Fourier-Motzkin, and the Gomory cut method, which is an extension of the Simplex [Gom63].

Recall that in the Fourier-Motzkin method, we start by extracting lower and upper bounds for the selected variable, and then write that each lower bound is not greater than each upper bound. This condition is enough to ensure the existence of a rational value, but not of an integer value for the selected variable. In fact, if one of the bounds is an integer, the existence of an integer solution is guaranteed. This happens in two cases: the bound is an affine form with integer coefficients, or the bound is a number, which can be replaced by its floor or ceiling. In the remaining case, one can prove that the possible values of the selected variable are of the form $x = dy + r, 0 \leq r < d$ for some number d. The original problem splits into d problems, one for each value of r, in which x is eliminated in favor of y. It is possible to prove that in this way one can proceed to eliminate all variables using only exact integer elimination. In the original Omega test software, various devices are used to eliminate redundant inequalities, to make the most out of equations, and to order the eliminations in the most favorable way. The resulting test is very fast.

In the case of the Simplex, one proceeds as in the rational case, until the optimum is found. If the solution is integral, there is nothing more to do. If not, one constructs a *Gomory cut*, i.e. a new constraint which excludes the optimum but no integer point in the Z-polyhedron. The Simplex is then restarted with one more constraint. It can be proved – see [Sch86] or [Fea88b] – that either the algorithm fails because one of the extended set of constraints proves infeasible, or an integral optimum is found after a finite number of cuts.

Both the Omega test and the Gomory cut method are inherently non polynomial algorithms, since the integer programming problem is known to be NP-complete.

3.4 Parametric Problems

A linear programming problem is parametric if some of its elements – e.g. the coefficients of the constraint matrix or those of the economic function – depend on parameters. In problems associated to parallelization, it so happens that constraints are often linear with respect to parameters.. In fact, most of the time we are given a polyhedron $\mathcal{P}$:

$$A \begin{pmatrix} \mathbf{x} \\ \mathbf{y} \end{pmatrix} + \mathbf{a} \geq 0$$

in which the variables have been partitioned in two sets, the unknowns: $\mathbf{x}$, and the parameters: $\mathbf{y}$. Setting the values of the parameters to $\mathbf{p}$ is equivalent to considering the intersection of $\mathcal{P}$ with the hyperplane $\mathbf{y} = \mathbf{p}$, which is also a polyhedron. In a parametric problem, we have to find the lexicographic minimum of this intersection as a function of $\mathbf{p}$.

The Fourier-Motzkin method is "naturally" parametric in this sense. One only has to eliminate the unknowns from the last component of $\mathbf{x}$ to the first. When this is done, the remaining inequalities give the conditions that the parameters must satisfy for the intersection to be non empty. If this condition is verified, each unknown is set to its minimum possible value, i.e. to the maximum of all its lower bounds. Let $C\mathbf{y} + \mathbf{c} \geq 0$ be the resulting inequalities after elimination of all unknowns. The parametric solution may be written:

$$\min_{\ll}(\mathcal{P} \cap \{\mathbf{y} = \mathbf{p}\}) = \textbf{if } C\mathbf{p} + \mathbf{c} \geq 0 \textbf{ then } \begin{pmatrix} \max(f(\mathbf{p}), \ldots, g(\mathbf{p})) \\ \cdots \\ \max(h(\mathbf{p}), \ldots, k(\mathbf{p})) \end{pmatrix} \textbf{ else } \perp$$

where $\perp$ is the undefined value and the functions $f, \ldots, k$ are affine.

In the case of the Simplex, the situation is more complicated. One may notice that since the coefficients of the constraint matrix A are constant, once the pivot is known, a step of Gaussian elimination can be done without difficulty. Similarly, when the pivot line is known, the choice of the pivot column depends only on the constraint matrix, hence does not depend on parameters. The only difficulty lies in the choice of the pivot line, which is such that its constant coefficient must be negative. Since this coefficient depends in general on the parameters, its sign cannot be ascertained; the problem must be split in two, with opposite hypotheses on this sign. These hypotheses are not independent; each one restricts the possible values of the parameters, until inconsistent hypotheses are encountered. At this point, the splitting process stops. By climbing back the problem tree, one may reconstruct the solution in the form of a multistage conditional. The advantage of the parametric Simplex over the Fourier-Motzkin algorithm is that it can be extended to the all-integer case. Parametric Gomory cuts can be constructed by introducing new parameters which represent in fact integer quotients. The reader is referred to [Fea88b] for details of the Parametric Integer Programming (PIP) algorithm which implements these ideas.

In this way, calculation of $K^p_{\mathtt{RS}}$ in (7) is a straightforward application of PIP (with a little fiddling for computing a maximum). The combination of the various sources for constructing the final solution is an exercise in formal simplification.

4 Program transformations

Since the 1980's, many reaserchers have designed dozens of program transformations with the aim of finding more and more parallelism in static control programs. These transformations can be classified along the following lines:

- Transformations which bring the source code nearer to the static control model, like `GOTO`s elimination, inductive variable detection and `DO` loop reconstruction. For more information, the reader is referred to any book on sequential compilation, as for instance [ASU86].
- Transformations which change the execution order of a program, the set of operations being left untouched. The polytope model offers an integrated way of choosing and applying these transformations, to which we will return later.
- Transformations which change the data structures of the source program, for instance by expanding scalars to arrays. The key to this transformation is dataflow analysis.
- Transformations which rely on the mathematical properties of the source algorithm. One may for instance use the associativity of some arithmetic operators like + and * to find parallelism in reductions. Another case is the replacement of iteration by chaotic iteration for some convergent algorithms. The problem with these transformations is that they modify – for better or for worse – the sensitivity of the algorithm to rounding errors, and have to be used with caution. The study of that kind of transformation is just beginning; the interested reader is referred to [RF93].

4.1 Reordering Transformations

Introduction One of the earliest discovery in the field was that most "old style" reordering transformations were in fact linear or affine transformations of iteration spaces. As a very simple exemple, consider the well known *loop inversion* transformation:

```
do i = ...                            do j = ...
  do j = ...                            do i = ...
    S                ===>                 S
  end do                                end do
end do                                end do
```

If we rename the target loop counters j' and i' to avoid confusion, this is associated to the linear transformation:

$$\begin{pmatrix} j' \\ i' \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix}.$$

Many other transformations, like loop splitting or loop skewing, can be represented in this way [Pug91b, Lu91], but we can also represent more transformations which have never been named in the litterature.

There are many possible styles of transformations, according to the complexity one tolerates in the source program and the number of degrees of freedom one handles in the transformation. On the one hand, one may consider only perfect loop nests, where all statements in the loop body have the same iteration domain. In that case, it is customary to use essentially the same transformation for all statements. On the other hand, one may have an arbitrary static control program, and use a different affine transformation for each statement S:

$$\mathbf{x}' = T_{\mathtt{S}}\mathbf{x} + t_{\mathtt{S}}.$$

In order to simplify the notations, we will use $\mathcal{T}(\mathtt{S}, x)$ for $T_{\mathtt{S}}\mathbf{x} + t_{\mathtt{S}}$.

Since the number of operations in the transformed space is to be the same as in the original space, all T's have to be one to one. If we suppose for simplicity that each iteration domain is full dimensional – avoiding such oddities as:

```
do i = 1,n
  do j = i,i
```

then the dimension of the transformed space must be at least equal to the dimension of the original space. There is no objection, however, for it to be greater. If needed, we may, for instance, pad $\mathbf{x}'$ with constant values. We may thus suppose that all images of iteration domains belong to the same target space, and that operations are to be executed – provisionally at least – in lexicographic order of their transformed coordinates. In the following, we will use N as the dimension of the target iteration space, with $N \geq N_{\mathtt{S}}$ for all statements S.

Legality of a Transformation We thus see that a transformation defines an execution order:

$$\langle \mathtt{R}, \mathbf{x}\rangle \prec_T \langle \mathtt{S}, \mathbf{y}\rangle \equiv \mathcal{T}(\mathtt{R}, x) \ll \mathcal{T}(\mathtt{S}, y).$$

Such a transformation is legal if all operations in dependence are correctly ordered – see section 2. Suppose that there is a dependence at depth p from R to S. The legality condition is:

$$\mathbf{Q}^p_{\mathtt{RS}}(\mathbf{x}, \mathbf{y}) \Rightarrow \mathcal{T}(\mathtt{R}, x) \ll \mathcal{T}(\mathtt{S}, y). \tag{8}$$

This can easily be transformed into a legality test by reductio at absurdum: a transformation is illegal if the following system is feasible:

$$\mathbf{Q}^p_{\mathtt{RS}}(\mathbf{x}, \mathbf{y}) \wedge \mathcal{T}(\mathtt{S}, y) \underline{\ll} \mathcal{T}(\mathtt{R}, x). \tag{9}$$

Here $\preccurlyeq$ is the "lexicographically less than or equal to" predicate. It can be split into the disjuction of $N+1$ linear predicates – one more term than in (2). Hence, testing the legality of a transformation entails testing the emptiness of $N+1$ polyhedra from each dependence, and this can be done by the methods of the preceeding section.

This test can be simplified somewhat in the case of a perfect loop nest with constant dependence vectors $\mathbf{d}_1, \ldots, \mathbf{d}_m$. In that case, there is only one transformation matrix T. The legality condition is obtained by combining (6) and (8), giving:

$$T\mathbf{d}_k \gg 0, k = 1, m.$$

To be legal, T must transform all dependence vectors into lexicopositive vectors.

After being transformed, the program may be seen as a single loop nest where the counters are the components of the transformed iteration vectors. When some of these components are constant, the corresponding loop may be unrolled, giving the equivalent of the familiar loop splitting transformation. Beside being legal, a program transformation must be useful, i.e. some of the loops in the target program must be parallel. In general, one knows beforehand which loops are parallel and which loops are sequential: this is a byproduct of the selection of $\mathcal{T}$. If necessary, the following test can be used. Let R and S be two statements in dependence at depth p. In the transformed program, the dependence is at depth q iff:

$$\mathbf{Q}^p_{\mathtt{RS}}(\mathbf{x}, \mathbf{y}) \Rightarrow \mathcal{T}(\mathtt{R}, x)[1..q] = \mathcal{T}(\mathtt{S}, y)[1..q],$$

As above, this can be tested by reductio at absurdum.

Here again, the test simplifies if the dependences are uniform. The transformed dependence vector $T\mathbf{d}_k$ is at depth q if its first q components are zero.

Selection of a transformation The only case in which it has been possible to devise an algorithm for finding $\mathcal{T}$ in one step is the one of uniform perfect loop nests, see [WL91]. Another possibility is to search for a good transformation among a finite – albeit very large – set of possible candidates, see [KP94]. Other researchers use methods which find only parts of $\mathcal{T}$. The problem is then to extend $\mathcal{T}$ to a one-to-one transformation, or to fit the parts together.

Scheduling Since the pioneering papers of [KMW67] and [Lam74], there have been a large number of papers on scheduling, mainly from the "systolic" community. The basic observation is that for any function θ from the set of operations to any totally ordered set, the following relation:

$$u \prec_\theta v \equiv \theta(u) < \theta(v)$$

is a partial order whose non-comparable pairs are such that $\theta(u) = \theta(v)$. It should be clear that not all partial orders can be represented in this way, since $\prec_\theta$ has a transitive non-comparability relation, which is not the case in general.

If $\prec_\theta$ is to be a correct parallel order for a given program, then it must satisfy the following adaptation of (5):

$$\mathbf{Q}^p_{\mathtt{RS}}(\mathbf{x},\mathbf{y}) \Rightarrow \theta(\langle \mathtt{R},\mathbf{x}\rangle) < \theta(\langle \mathtt{S},\mathbf{y}\rangle). \tag{10}$$

This set of functional inequalities can be interpreted in several ways. We have seen one, in which a special representation for $\prec_{//}$ has been selected. Notice that since this is not the most general one, we have sacrificed some parallelism in the interest of simplicity.

Since the set of operations in a DO loop program is finite, it is always possible to suppose that the range of θ is $\mathbb{N}$ or $\mathbb{N}^d$, the corresponding total order being either the familiar integer ordering or lexicographic ordering. Let us consider the first case for simplicity. Since θ has integer values, (10) can be rewritten as:

$$\mathbf{Q}^p_{\mathtt{RS}}(\mathbf{x},\mathbf{y}) \Rightarrow \theta(\langle \mathtt{R},\mathbf{x}\rangle) + 1 \leq \theta(\langle \mathtt{S},\mathbf{y}\rangle). \tag{11}$$

In this form, one may consider that $\theta(\langle \mathtt{R},\mathbf{x}\rangle)$ gives the execution time of operation $\langle \mathtt{R},\mathbf{x}\rangle$ on a computer with an unbounded number of processors which execute all operations in unit time. θ is a *schedule* for the source program.

> One may replace the 1 in (11) by the actual execution time of the corresponding operation. This refinement is of no great importance in the case of massively parallel programming. It has, however, great impact in the case of *Instruction Level Parallelism.*

The last interpretation of (10) is that we are constructing a transformation, θ giving its first d components. To apply the above theory, θ must be affine. We may then complete it by adding $N - d$ lines in such a way that the resulting $\mathcal{T}$ transformation is one-to-one. It is easy to see that (11) will then entail (8). What is the shape of the resulting parallel program? We may suppose that if a d dimensional θ has been used, it is because a smaller dimensional schedule would not have met condition (11), or, equivalently, that in the transformed program there are dependences from depth 0 to d. This means that the d outer loops of the transformed program are sequential, as befit loop counters which represent time. We may always suppose that the constant term in θ has been adjusted in such a way that its minimum value is 0. Supposing for simplicity that $d = 1$, and that the transformed iteration vector is $(t, \mathbf{z})^T$, then the parallel program shape is:

```
program S
do t = 0, L
    doall z ∈ F(t)
        T^{-1}(t, z)^T
```

$\mathcal{T}$ being invertible, $\mathcal{T}^{-1}(t,\mathbf{z})$ is the unique operation u such that $\mathcal{T}(u) = (t,\mathbf{z})$. $\mathcal{F}(t)$ is the set:

$$\mathcal{F}(t) = \{u \mid \theta(u) = t\},$$

and is known in the litterature as the *front* at time t. Finally, L is the maximum value of θ, i.e. the latency of the parallel program. One should notice that this program sketch, which consists of one or more sequential loops enclosing parallel loops, is in the best possible shape for vector computers.

The main question, however, is how to solve (11). The starting point is the assumption that schedules are affine functions of the loop counters. There is no justification for this assumption beside expediency. Even in very simple cases, it can be shown that schedules can be very complicated functions. The assumption acts rather as a filtering device. Experience shows that most static control programs have a large number of schedules, from which we select those which are affine. Furthermore, it can be shown that for programs with uniform dependences, affine schedules are *asymptotically optimal*, i.e. they give latency of the same order as the best possible or free schedule [KMW67, DKR91, d'A95].

Let us set:

$$\theta(S, \mathbf{x}) = \tau_{\mathtt{S}}.\mathbf{x} + c_{\mathtt{S}},$$

where $\tau_{\mathtt{S}}$ is an unknown timing vector and $c_{\mathtt{S}}$ is an unknown offset. If such a prototype is inserted into (11), we get:

$$\mathbf{Q}^p_{\mathtt{RS}}(\mathbf{x}, \mathbf{y}) \Rightarrow \tau_{\mathtt{R}}.\mathbf{x} + c_{\mathtt{R}} + 1 \leq \tau_{\mathtt{S}}.\mathbf{x} + c_{\mathtt{S}}. \tag{12}$$

One way of using this formula is to select arbitrarily numerical values for $\mathbf{x} \in \mathcal{D}_{\mathtt{R}}$ and $\mathbf{y} \in \mathcal{D}_{\mathtt{S}}$. Either these values are such that $\mathbf{Q}^p_{\mathtt{RS}}(\mathbf{x}, \mathbf{y})$ is true, in which case we get a linear inequality involving the unknowns $\tau_{\mathtt{R}}$, $c_{\mathtt{R}}$, $\tau_{\mathtt{S}}$ and $c_{\mathtt{S}}$, or else $\mathbf{Q}^p_{\mathtt{RS}}(\mathbf{x}, \mathbf{y})$ is false, in which case we get nothing. In this way, we can get a very large (if the iteration domains are finite) or even an infinite set of linear constraints, to be solved for the unknowns. This is evidently not a practical procedure. A solution is possible, however, because the fact that all domains and constraints are linear allows one to construct a finite summary for this potentially infinite problem.

One way of obtaining a summary is to notice that (12) is true everywhere iff it is true at the vertices of $\mathbf{Q}^p_{\mathtt{RS}}$ [Qui87]. We obtain in this way as many linear constraints as the $\mathbf{Q}^p_{\mathtt{RS}}$'s have vertices. Another solution is to use the affine version of Farkas lemma [Sch86, Fea92a]: the general solution of

$$(A\mathbf{x} + \mathbf{b} \geq 0) \Rightarrow (\boldsymbol{\alpha}.\mathbf{x} + \beta \geq 0)$$

is:

$$\boldsymbol{\alpha}.\mathbf{x} + \beta = \lambda_0 + \boldsymbol{\lambda}.(A\mathbf{x} + \mathbf{b}),$$

where $\lambda_0 \geq 0$ and $\boldsymbol{\lambda} \geq 0$.

This last equality is to be considered as an identity, in which coefficients of like components of $\mathbf{x}$ can be equated, to give linear relations between the $\boldsymbol{\alpha}$'s and β and the new positive unknowns $\lambda_0, \boldsymbol{\lambda}$.

Whatever the method, one gets a system of linear inequalities to be solved for the coefficients in the schedule. This system may be infeasible; in which case one must resort to multidimensional scheduling – the reader is referred to [Fea92b] for details. If feasible, the system generally has many solutions. There are several ways of choosing the "best" one. For instance, since the latency can be expressed as a linear form in the coefficients of the schedule, one can set up a linear program for finding minimum latency schedules. Other possibilities are leftmost linear schedules and bounded delays schedules.

One should be aware that one has a wide range of possibilities for the choice of a schedule. Program S above has to be executed on a limited number of processors, say P, by a run-time scheduler. Let T_P be the execution time on P processors. T_1 is the sequential time, and T_∞ is the latency on an unlimited number of processors, i.e. the L in program S. If the run-time scheduler is greedy – i.e. if no processor stays idle if there is work to do – and if we neglect problems of interference and communication between processors, then, by Brent's lemma:

$$T_p \leq T_\infty + T_1/P,$$

which implies that the efficiency $\frac{T_1}{PT_P}$ is near one when $\pi = T_1/T_\infty$ is large. π is the mean parallelism in the program. For most algorithms in numerical analysis, π grows without limit for any reasonable schedule when the size of the problem grows. For instance, for Gaussian elimination on a system of n equations with n unknowns, $T_1 = O(n^3)$ and $T_\infty = O(n)$ hence $\pi = O(n^2)$. In such cases, the choice of a schedule is not too critical.

Placement While constructing a transformation from a schedule gives good results on a synchronous computer, where, conceptually, each front can be executed at each tick of the global clock, a finer analysis is needed in the case of a distributed memory computer. The main problem here is to avoid communications between processors through the interconnection network, which is always orders of magnitude slower than local memory accesses. In the case of systolic arrays, no attempt is made at minimizing the amount of communication, the rationale probably being that since the array is custom designed, one can always provide the necessary channels. In a distributed memory processor, one cannot enlarge the network at will, hence the numerous attempts at transforming the program in such a way that most communications are made local.

This result can be obtained only if the concept of a transformation is extended to include the data space of a program. In the same way that $\mathcal{T}$ maps an operation to an abstract space, some coordinates of which were later interpreted as time, we will suppose that the same transformation maps an array cell $\mathtt{A}[i]$ – where the components of $\mathbf{i}$ are subscripts – to an abstract space some coordinate of which are interpreted as – virtual – processor names.

In placement, we are only interested in the part of $\mathcal{T}$ which relates to processors. $\mathcal{T}(\mathtt{S}, x)$ is now supposed to be the name of the processor which executes $\langle \mathtt{S}, \mathbf{x} \rangle$; similarly, $\mathtt{A}$ being an array, $\mathcal{T}(\mathtt{A}, i)$ is the name of the processor whose memory holds $\mathtt{A}[i]$. If operation $\langle \mathtt{S}, \mathbf{x} \rangle$ contains a reference to $\mathtt{A}[\mathbf{f}(\mathbf{x})]$, there will be a remote data reference unless these two entities are in the same processor:

$$\mathcal{T}(\mathtt{S}, x) = \mathcal{T}(\mathtt{A}, \mathbf{f}(\mathbf{x})). \tag{13}$$

These equations are solved in the same fashion as the scheduling equations (11). We assume that $\mathcal{T}$ is affine, replace it by a prototype:

$$\mathcal{T}(\mathtt{U}, x) = \boldsymbol{\pi}_{\mathtt{U}}.\mathbf{x} + \varpi_{\mathtt{U}}, \tag{14}$$

where `U` is either a statement or an array. The problem is then to find relations between the unknwns $\boldsymbol{\pi}$ and ϖ which are equivalent to (13). This is similar to the method we used to solve the scheduling inequalities (12). There are, however, some important differences:

- Since (13) is an equation, it suffices that it holds for $\mathbf{x} = 0$ and for $N_{\mathtt{S}\mathtt{S}}$ other linearly independent values of $\mathbf{x}$ to hold everywhere. Hence, as soon as the iteration domain of `S` is large enough, (13) is true everywhere. By replacing $\mathbf{x}$ successively by 0 and by the unit vectors, we get the required relations between the unkowns $\boldsymbol{\pi}$ and ϖ. identity.
- The result is a system of homogeneous linear equations. It always has at least the trivial solution, whose meaning is that we can suppress all communications by using only processor 0. Let $\boldsymbol{\pi}$ be a vector in which all coefficients of $\mathcal{T}$ are concatenated. The resulting system may be written:

$$C\boldsymbol{\pi} = 0;$$

  C is the *communication matrix* of the source program.
- Equation (13) has a different meaning than, for instance, (11). This last equation is a constraint: if not satisfied, the resulting parallel program is invalid. (13) is more in the nature of an economic function: for each value of $\mathbf{x}$ for which it is false, one has to program a remote data access. The real constraint here is that all operations and all data of the program do not collapse on the same processor.

We see that $\boldsymbol{\pi}$ is a vector in the kernel of C. If we want the target system to be a grid of processors, then the range of $\mathcal{T}$ must have as many dimensions as the grid. This is obtained by selecting enough linearly independent vectors in $\ker(C)$, if possible.

Most of the time, however, C is of full rank and its kernel is trivial.

> In the case of uniform dependences, the dependence vectors directly give the column vectors of C. It is a well known fact that there is no placement – or no outermost parallelization – of a unform program if the dependence vectors span the whole iteration space.

The solution in that case is to satisfy only a subset of equations (13). Equations which do not belong to the chosen subset correspond to residual communications. There are various heuristics for choosing the residual communications, for which the reader is referred to [Fea94, DR95].

Discussion Computing a schedule, as in the preceding paragraph, or computing a placement, as above, are two independent ways of finding parallelism in a program. Each method aims at transforming the dependence graph in such a way that the resulting program has a simple shape. In the case of scheduling, one adds edges to the dependence graph. Two operations u and v may be such that $\theta(u) < \theta(v)$ and yet be independent. The target dependence graph is in the form of a one level series-parallel graph. Since the resulting program is more constrained than the original, it is ipso facto correct, but we may have lost some parallelism.

On the contrary, when computing a placement we ignore some edges, namely those which correspond to residual communications. The aim is to partition the dependence graph into independent subsets which are then executed sequentially. Since edges have been ignored, the program is invalid unless one reintroduces them as communications. The result is a system of cooperating sequential processes. There may be more processes than processors. In general, the task of multiplexing several processes on one processor is left to the underlying operating system. *Virtual processors* are processes whose programming interface has been modeled on the underlying physical processor.

In some cases, one needs both a placement, because the machine has distributed memory, and a schedule, either because the machine is synchronous, or just for convenience. In these cases, we have to build two transformations as above and fit them as best we can. At the time of writing, there is no integrated theory of *space-time transforms*.

4.2 Storage management

The question is how to integrate in the above framework transformations which act on the data structures of the program. One knows from previous research that some dependences are due to memory reuse while others, the direct flow dependences, are inherent to the algorithm. If memory is not reused, then the first type of dependence disappears, thus giving a less constrained parallel program.

The simplest situation is when there is enough space on a distributed memory computer to duplicate the whole data space. This expansion remove all dependences except direct flow dependences. Of these, some have been taken into account when constructing the placement, thus giving local data accesses. The residual dependences give rise to communications, which can be constructed with the help of dataflow analysis. The resulting code may then be optimized by deallocating unused memory.

When the parallel program is to be constructed via a schedule, the first idea that come to mind is to ignore non flow dependences. Ince non-flow dependences are generated by memory reuse, they can be eliminated by scalar and array expansion. One may observe that data expansion remove not only non flow dependences, but also spurious flow dependences, which are eliminated by array dataflow analysis. This justifies ignoring all but direct flow dependences when computing a schedule. I have proposed to restore the correctness of the parallel program by converting it to Single Assignment form [Fea91]. Recent developments show that it is possible to achieve the same result at a much lower cost in memory, by ignoring the dependences which are already satisfied by the selected schedule.

5 Loop rewriting and code generation

The essence of the polytope model is to apply affine transformations to the iteration spaces of a program. When this is done, the operation in the original program are to be executed according to the lexicographic order *in the transformed iteration space.* The problem of code generation is thus the problem of writing a loop nest which scans the image of a polytope by an affine transform. When a whole program has been reordered, one has to scan the union of the images of several polytopes.

5.1 The case of a perfect loop nest

In the case of a perfect loop nest, there is essentially one statement and one transformation, T. The points to be scanned are defined by:

$$T(\mathcal{D}) = \{\mathbf{y} \mid \exists \mathbf{x} : \mathbf{y} = T\mathbf{x}, D\mathbf{x} + \mathbf{d} \geq 0\},$$

where $D\mathbf{x} + \mathbf{d} \geq 0$ is the system of constraints which define the iteration domain in the source program. This shows that $\mathbf{y}$ belongs to the lattice generated by T. In $\mathcal{L}(T)$, T is invertible. Hence, the set $T(\mathcal{D})$ may be rewritten:

$$T(\mathcal{D}) = \{\mathbf{y} \mid \mathbf{y} \in \mathcal{L}(T), DT^{-1}\mathbf{y} + \mathbf{d} \geq 0\}$$

which define a Z-polyhedron.

Unimodular transformations If T is unimodular, $\mathcal{L}(T)$ is the canonical Z-module, i.e. the set of vectors with integer coordinates. There are several ways of computing the bounds of a loop nest which scans $\mathcal{L}(T)$. For instance, we may use the Fourier-Motzkin algorithm in the following way [Iri87, AI91]:

- Compute the lower and upper bounds l_N and u_N of the last component of $\mathbf{y}$. Since these bounds may be rational while y_N is an integer, we have to apply ceiling and floor functions to compute the actual bounds. The results looks like:
$$\texttt{do } y_N = \lceil l_N \rceil , \lfloor u_N \rfloor$$
- Eliminate y_N and start again for the next component of $\mathbf{y}$.

Since the Fourier-Motzkin algorithm has a tendency to generate redundant inequalities, this method may result in more complicated bounds than is necessary, unless one programs a redundancy eliminator. Another solution is to use PIP for computing maxima and minima, in which case redundancy is automatically eliminated [CBF95].

Non-unimodular transformations In case T is not unimodular, the solution is to build its Hermite normal form $T = HU$ [Dar93, Xue94]. One builds, according to the above method, a loop nest which scans $U(\mathcal{D})$. Since, due to the special form of H, the transformation $\mathbf{y} = H\mathbf{z}$ is monotonic with respect to lexicographic ordering, it is enough to apply H to the loop counters of the new loop nest in order to generate the correct code. Alternatively, one may apply H directly to the new loop nest. It is easy to see that the diagonal elements of H give the loop *steps*, while the off diagonal elements generate initial offsets.

5.2 The case of a complete program

Here, each source statement generates one image, and the target code has to scan the union of these images. The nave solution consists in constructing a convex polytope which includes all these images. One may use the convex hull or the rectangular hull of the union of all iteration domains. This polytope is then scanned as above. All statements are then inserted in the innermost loop body, with guards that ensure they are executed only at the proper time. The resulting code is inefficient, since overhead operations (the guards) are inserted at the innermost level. There are various devices for improving the results. One may move invariant calculations up through the loop hierarchy, split loops according to the value of a guard, peel loops, and so on [AALL93].

5.3 Communication code

If the parallel program is to run on a distributed memory machine, one has to insert code for the residual communications. This depends in a complicated way on the architecture of the target computer. In the case of an asynchronous computer, the simplest solution is to duplicate the sequential code and its data structures in each processor. One then adds guards to avoid duplicating the calculations and communications.

Let us suppose that distribution is specified by a placement function $\mathcal{T}$, and let $\mathbf{q}$ be the current processor number. Operation u is replaced by the following code [ZBG88]:

$$\begin{array}{l} \forall \mathtt{a} \in R(u) : \mathtt{if}\ \mathcal{T}(u) \neq \mathbf{q} \wedge \mathcal{T}(\mathtt{a}) = \mathbf{q}\ \mathtt{then\ Send(a)\ to}\ \mathcal{T}(u) \\ \qquad\qquad\quad \mathtt{if}\ \mathcal{T}(u) = \mathbf{q} \wedge \mathcal{T}(\mathtt{a}) \neq \mathbf{q} \\ \qquad\qquad\quad \mathtt{then\ Receive(a)\ from}\ \ \mathcal{T}(\mathtt{a}) \\ \mathtt{if}\ \mathcal{T}(u) = \mathbf{q}\ \mathtt{then\ c} = f(R(u)) \end{array}$$

This code is highly inefficient, due to the numerous guards. In the case of static control programs, most guards can be resolved at compile time and "pushed up" into the surrounding loop bounds. Similarly, each processor uses only a fraction of its data space. The remnants can be deallocated, at the price of more complicated subscripts.

6 Conclusion: the limits of the polytope model

There are still some fine points that are not completely solved in the polytope model. Among them are the construction of more general placement functions, the choice of the best style of transformation for a given architecture, minimum data expansion, code generation for arbitrary programs, and communication code construction. These problems are the subjects of active research, and there is hope they will be solved in the near future.

The main question is quite different: what is the range of the polytope model? Are real life programs in the model or not? The answer is more ambiguous than we would like. It seems that most real programs do not have static control, with the exception of toy examples and small library subroutines. However, it is possible to isolate static control kernels in large programs and have them parallelized by the above methods [Les96]. If it so happens that these kernels represent a large fraction of the total running time, our job is done.

Some programs have irregular control and/or irregular data accesses. It is still possible to extend dataflow analysis and scheduling to these situations, by the use of approximation methods. The only way of extracting parallelism from them, however, seems to be by the use of speculative execution. In some cases, what appear to be irregular are in fact regular accesses to other data structures (e.g. trees) which have been implemented as arrays. The equivalent of the polytope model for these situations is still to be built.

Lastly, in many cases, irregular programs are really regular programs which have been optimized for special situations. This is the case, e.g., for sparse codes, in which familiar algorithms like the matrix-vector product have been modified to avoid doing multiplications by zero. A solution in that case is to use run-time parallelization. Another one is to parallelize the original code, then do the optimization for sparsity on the parallel version.

Acknowledgements

I would like to thank Monica Lam and Peter Drakenberg for helping me improve the presentation of this paper. All remaining errors are mine.

References

[AALL93] Saman P. Amarasinghe, Jennifer M. Anderson, Monica S. Lam, and Amy W. Lim. An overview of a compiler for scalable parallel machines. In *Sixth Annual Workshop on Languages and Compilers for Parallel Computing*, pages 253–272. Springer Verlag, LNCS 768, August 1993.

[AI91] Corinne Ancourt and François Irigoin. Scanning polyhedra with DO loops. In *Proc. third SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pages 39–50. ACM Press, April 1991.

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, Reading, Mass, 1986.

[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.

[CBF95] Jean-François Collard, Denis Barthou, and Paul Feautrier. Fuzzy array dataflow analysis. In *ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*. ACM, July 1995.

[d'A95] Patrick Le Goueslier d'Argence. *Contribution à l'étude des probleèmes d'ordonnancement cycliques multidimensionnels*. PhD thesis, Université Paris VI, December 1995.

[Dan63] G. B. Dantzig. *Linear Programming and Extensions.* Princeton University Press, 1963.

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

[DKR91] Alain Darte, Leonid Khachiyan, and Yves Robert. Linear scheduling is nearly optimal. *Parallel Processing Letters*, 1(2):73–81, 1991.

[DR95] Michèle Dion and Yves Robert. Mapping affine loop nest: new results. In *HPCN Conf.* LNCS 919, 1995.

[DV94] Alain Darte and Frédéric Vivien. Automatic parallelization based on multi-dimensional scheduling. Technical Report RR 94-24, LIP, 1994.

[Fea88a] Paul Feautrier. Array expansion. In *ACM Int. Conf. on Supercomputing*, pages 429–441, 1988.

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

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

[Fea92a] Paul Feautrier. Some efficient solutions to the affine scheduling problem, I, one dimensional time. *Int. J. of Parallel Programming*, 21(5):313–348, October 1992.

[Fea92b] Paul Feautrier. Some efficient solutions to the affine scheduling problem, II, multidimensional time. *Int. J. of Parallel Programming*, 21(6):389–420, December 1992.

[Fea94] Paul Feautrier. Toward automatic distribution. *Parallel Processing Letters*, 4(3):233–244, 1994.

[Fou90] J. B. J. Fourier. *Oeuvres de Fourier, Tome II.* Gauthier-Villard, Paris, 1890.

[Gom63] R. E. Gomory. An algorithm for integer solutions to linear programs. In R. L. Graves and P. Wolfe, editors, *Recent Advances in Math. Programming*, chapter 34, pages 269–302. 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.

[IT87] François Irigoin and Rémi Triolet. Computing dependence direction vectors and dependence cones with linear systems. Technical Report CAI-87-E94, Ecole des Mines de Paris, 1987.

[KMW67] R. M. Karp, R. E. Miller, and S. Winograd. The organization of computations for uniform recurrence equations. *Journal of the ACM*, 14:563–590, 1967.

[KP94] Wayne Kelly and William Pugh. Selecting affine mappings based on performance estimations. *Parallel Processing Letters*, 4(3):205–220, September 1994.

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

[Les96] Arnauld Leservot. *Analyse Interprocédurale du flot des données.* PhD thesis, Université Paris VI, March 1996.

[Lu91] Lee-Chung Lu. A unified framework for systematic loop transformations. *SIGPLAN Notices*, 26:28–38, July 1991. 3rd ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming.

[Min83] Michel Minoux. *Programmation Mathématique, théorie et algorithmes.* Dunod, Paris, 1983.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The definition of Standard ML.* The MIT Press, 1990.

[Pug91a] William Pugh. The omega test: A fast and practical integer programming algorithm for dependence analysis. In *Supercomputing*, 1991.

[Pug91b] William Pugh. Uniform techniques for loop optimization. *ACM Conf. on Supercomputing*, pages 341–352, January 1991.

[PW93] William Pugh and David Wonnacott. An evaluation of exact methods for analysis of value-based array data dependences. In *Sixth Annual Workshop on Programming Languages and Compilers for Parallel Computing*, pages 546–566. Springer-Verlag LNCS 768, August 1993.

[Qui87] Patrice Quinton. The systematic design of systolic arrays. In F. Fogelman, Y. Robert, and M. Tschuente, editors, *Automata networks in Computer Science*, pages 229–260. Manchester University Press, December 1987.

[RF93] Xavier Redon and Paul Feautrier. Detection of reductions in sequential programs with loops. In Arndt Bode, Mike Reeve, and Gottfried Wolf, editors, *Procs. of the 5th Int. Parallel Architectures and Languages Europe*, pages 132–145. LNCS 694, June 1993.

[Sch86] A. Schrijver. *Theory of linear and integer programming.* Wiley, NewYork, 1986.

[WL91] M. Wolf and Monica S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Trans. on Parallel and Distributed Systems*, 2(4):452–471, October 1991.

[Xue94] J. Xue. Automating non-unimodular loop transformations for massive parallelism. *Parallel Computing*, 20(5):711–728, May 1994.

[ZBG88] H. P. Zima, H. J. Bast, and M. Gerndt. SUPERB : A tool for semi-automatic MIMD/SIMD parallelization. *Parallel Computing*, 6:1–18, 1988.

This article was processed using the LaTeX macro package with LLNCS style