TITLE

The Polyhedron Model

BYLINE

Paul Feautrier
LIP, ENS Lyon
Lyon
France
Paul.Feautrier@ens-lyon.fr

Christian Lengauer
Department of Informatics and Mathematics
University of Passau
Passau
Germany
Christian.Lengauer@uni-passau.de

SYNONYMS

Polytope model

DEFINITION

The polyhedron model (earlier known as the polytope model [21, 37]), is an abstract representation of a loop program as a computation graph in which questions such as program equivalence or the possibility and nature of parallel execution can be answered. The nodes of the computation graph, each of which represents an iteration of a statement, are associated with points of $\mathbb{Z}^n$. These points belong to polyhedra which are inferred from the bounds of the surrounding loops. In turn, these polyhedra can be analyzed and transformed with the help of linear programming tools. This enables the automatic exploration of the space of equivalent programs; one may even formulate an objective function (such as the minimum number of synchronization points) and ask the linear programming tool for an optimal solution. The polyhedron model has stringent applicability constraints (mainly to FOR loop programs acting on arrays), but extending its limits has been an active field of research. Beyond autoparallelization, the polyhedron model can be useful in many situations which call for a program transformation, such as in memory or performance optimization.

DISCUSSION

The basic model

Every compiler must have representations of the source program in various stages of elaboration, as for instance by character strings, abstract syntax trees, control graphs, three addresses code and many others. The basic component of all these representation is the statement, be it a high level language statement of a machine instruction. Unfortunately, these representations do not meet the needs of an autoparallelizer, simply because parallelism does not occur between statements, but between statement executions or *instances*. Consider:

for $i = 0$ to $n-1$ do
S : $\quad a[i] = 0.0$
od

It makes no sense to ask whether S can be executed in parallel with itself; in this case, parallelism depends both on the way S accesses memory and on the way the loop counter i is updated at each iteration.
A loop program must therefore be represented as a set of instances, its *iteration domain*, here named E. Each instance has a distinct name, and consists in the execution of the related statement or instruction, depending on the granularity of the analysis. This set is finite, in the case of a terminating program, or infinite in the case of a reactive or streaming system.
However, this is not sufficient to specify the object program. One needs to know in which order the instances are executed; E must be ordered by some relation $\prec$. If $u, v \in E$, $u \prec v$ means that u is executed before v. Since an operation cannot be executed before itself, $\prec$ is a strict order. It is easy to see that the usual control constructs (sequence, loops, conditionals, jumps) are compact ways of defining $\prec$. It is also easy to see that, in a sequential program, two arbitrary instances are always ordered: one says that, in this case, $\prec$ is a total order. Consideration of an elementary parallel program (in OpenMP notation):

#pragma omp parallel sections
$\quad S_1$
#pragma omp section
$\quad S_2$
#pragma omp end parallel sections

shows that S_1 may be executed before or after or simultaneously with S_2, depending on the available resources (processors) and the overall state of the target system. In that case, neither $S_1 \prec S_2$ nor $S_2 \prec S_1$ are true: one says that $\prec$ is a partial order. As an extreme case, an embarassingly parallel program, in which instances can be executed in any order, has the empty execution order. Therefore, one may say that parallelization results in replacing the total execution order of a sequential program by a partial one, under the constraint that the outcome of the program is not modified. This in turn raises the following question: *Under which conditions are two programs with the same iteration domain but different execution orders equivalent?*

Since program equivalence is undecidable in general, one must be content with conservative answers, i.e. with sufficient but not necessary equivalence conditions. The usual approach is based on the concept of *dependences* (see also the [Dependence] entry in this encyclopedia). Assuming that, given the name of an instance u, one can characterize the sets (or supersets) of read and written memory cells, $\mathcal{R}(u)$ and $\mathcal{W}(u)$, u and v are in dependence, written $u\,\delta\,v$, if both access some memory cell and one of them at least modifies it. In symbols: $u\delta v$ iff at least one of the sets $\mathcal{R}(u) \cap \mathcal{W}(v)$, $\mathcal{W}(u) \cap \mathcal{R}(v)$ or $\mathcal{W}(u) \cap \mathcal{W}(v)$ is not empty. The concept of a dependence was first formulated by Bernstein [8]. One can prove that two programs are equivalent if dependent instances are executed in the same order in both.

> **Aside.** Proving equivalence starts by showing that under Bernstein's conditions, two independent consecutive instances can be interchanged without modifying the final state of memory. In the case of a terminating program, the result follows by specifying a succession of interchanges that convert one order into the other without changing the final result. The proof is more complex for non terminating programs and depends on a fairness hypothesis, namely that every instance is to be executed eventually. One can then prove that the succession of values assigned to each variable – its history – is the same for both programs. One first shows that the succession of assignments to a given variable is the same for both programs, since they are in dependence, and, as a consequence, that the assigned values are the same, provided all instances are deterministic, i.e. return the same value when executed with the same arguments.

To construct a parallel program, one wants to remove all orderings between independent instances, i.e. construct the relation $\delta \cap \prec$, and take its transitive closure. This execution order may be too complex to be represented by the available parallel constructs, like the parallel sections or the parallel loops of OpenMP. In this case, one has to trade some parallelism for a more compact program.

It remains to explain how to name instances, how to specify the index domain of a program and its execution order, and how to compute dependences. There are many possibilities, but most of them ask for the resolution of undecidable problems, which is unsuitable for a compiler. In the polyhedron model, sets are represented as *polyhedra* in $\mathbb{Z}^n$, i.e. sets of (integer) solutions of systems of affine inequalities (inequalities of the form $A\,x \leq b$, where A is a constant matrix, x a variable vector and b a constant vector). It so happens that these sets are the subject of a well developed theory, (integer) linear programming [43], and that all the necessary tools have efficient implementations.

The crucial observation is that the iterations of a regular loop (a Fortran DO loop, or a Pascal FOR loop, or restricted forms of C, C++ and Java FOR loops) are represented by a segment (which is a one-dimensional polyhedron), and that the iterations of a regular loop nest are represented by a polyhedron with as many dimensions as the nest has loops. Consider, for instance, the first statement of the loop program in Fig. 1(a). It is enclosed in two loops. The instances it generates can be named by stating the values of i and j, and the iteration domain is defined by the constraints:

$$1 \leq i \leq n,\ \ 1 \leq j \leq i+m$$

for $i = 1$ to n do

  for $j = 1$ to $i+m$ do

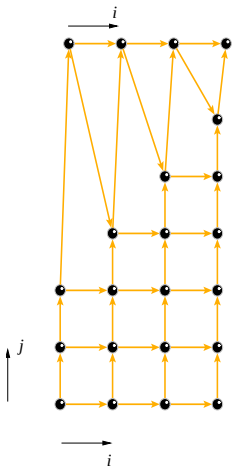
S_1 : $A(i,j) = A(i-1,j) + A(i,j-1)$

  od

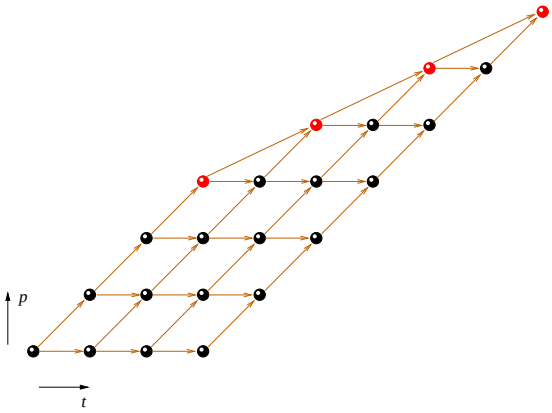
S_2 : $A(i,i+m+1) = A(i-1,i+m)+A(i,i+m)$

od

(a) source loop nest



(b) source iteration domain



(c) target iteration domain

for $t = 0$ to $m+2*n-1$ do

  parfor $p = \max(0, t-n+1)$ to $min(t, (\lceil t+m)/2 \rceil)$ do

    if $2*p = t+m+1$ then

S_2 : $A(p-m,p+1) = A(p-m-1,p) + A(p-m,p)$

    else

S_1 : $A(t-p+1,p+1) = A(t-p,p+1) + A(t-p+1,p)$

    fi

  od

od

(d) target loop nest

Figure 1: Loop nest transformation in the basic polyhedron model

which are affine and therefore define a polyhedron. In the same way, the iteration domain of the second statement is $1 \leq i \leq n$. For better readability, the loop counters are usually arranged from outside inward in a vector, the *iteration vector* of the instance. Observe that, in this representation, n and m are parameters, and that the size of the representation is independent of their values. Also, the upper bound of the loop on j is not constant: iteration domains are not limited to parallelepipeds.
The iteration domain of the program is the disjoint union of these two polyhedra, as depicted in Fig. 1(b) with dependences. (The dependences and the right side of the figure are discussed below.) To distiguish the several components of the union, one can use statement labels, as in:

$$E = \{\langle S_1, i, j\rangle \mid 1 \leq i \leq n, 1 \leq j \leq i+m\} \cup \{\langle S_2, i\rangle \mid 1 \leq i \leq n\}$$

The execution order can be deduced from two observations:

- In a program without control constructs, the execution order is *textual order*. Let $u <_{\rm txt} v$ be true iff u occurs before v in the program text.
- Loop iterations are executed according to the *lexicographic order* of the iteration vectors. Let $x <_{\rm lex} y$ be true iff the vector x is lexicographically less than y.

In more complex cases, these two observations may be combined to give:

$$\langle R, x\rangle \prec \langle S, y\rangle \equiv x[1:N] <_{\rm lex} y[1:N] \vee (x[1:N] = y[1:N] \wedge R <_{\rm txt} S),$$

where R and S are two statements, x and y their iteration vectors, N is the number of loops which encloses both R and S, and $x[1:N]$ is the vector x restricted to its N first components. Returning to Fig. 1, one has:

$$\langle S_1, i, j\rangle \prec \langle S_2, i'\rangle \equiv i < i' \vee (i = i' \wedge \mathbf{true}),$$

which simplifies into $\langle S_1, i, j\rangle \prec \langle S_2, i'\rangle \equiv i \leq i'$.
The assumption behind dependence analysis is that the sets $\mathcal{R}(u)$ and $\mathcal{W}(u)$ above depend only on the name of the instance u. This is obviously not true in general. In the polyhedron model, one assumes that all accesses are to scalars and arrays, and that, in the latter case, subscripts are known functions of the surrounding loop counters. One usually also assumes that there is no *aliasing* – two arrays with different names do not overlap – and that subscripts are always within the array bounds. Techniques for detecting and correcting violations of these assumptions are beyond the scope of this entry. With these assumptions, two instances $\langle R, x\rangle$ and $\langle S, y\rangle$ are in dependence if they both access the same array A of dimension d_A, and if the *subscript equations*

$$f_R(x) = f_S(y)$$

have solutions within the iteration domains of R and S. Here, f_R and f_S are the respective *subscript functions* of A in R and S. Solving such equations is easy only if each subscript is an affine function of the iteration vector:

$$f_R(x) = F_R\, x + g_R,$$

where F_R is a matrix of dimension $d_A \times d_R$ with d_R being the number of loops surrounding R, and g_R is a vector of dimension d_A. One may associate with each candidate dependence a system of constraints by gathering the subscripts equations, the constraints which define the iteration domains of R and S, and the sequencing predicate above. All of these constraints are affine, with the exception of the sequencing predicate which is a disjunction of affine constraints. Each disjunct can be tested for solutions, either by *ad hoc* conservative methods – see the [Banerjee test] entry in this encyclopedia – or by linear programming algorithms – see the [Dependence] entry.
In summary, a program can be handled in the polyhedron model – and is then called a *regular* or *static control program* – if its only control constructs are (also called regular) loops with affine bounds and its data structures are either scalars or arrays with affine subscripts in the surrounding loop counters. It should be noted that these restrictions must not be taken syntactically but semantically. For instance, in the program:

$i = 0; k = 0;$
while $i < n$ do
  $a[k] = 0.0;$
  $i = i+1;$
  $k = k+3$
od

the loop is in fact regular with counter i, and the subscript of a is really $3\,i$, which is affine. There are many classical techniques – here, induction variable detection – for transforming such constructs into a more "polyhedron-friendly" form.
Regular programs are mainly found in scientific computing, linear algebra and signal processing, where unbounded iteration domains are frequent. Perhaps more surprisingly, many variants of the Smith and Waterman algorithm [44], which is the basic tool for genetic sequence analysis, are regular and can be optimized with polyhedral tools [30]. Also, while large programs rarely fit in the model, it is often possible to extract regular kernels and to process them in isolation.

Transformations

The main devices for program optimization in the polyhedron model are coordinate transformations of the iteration domain.

An example

Consider Fig. 1 as an illustration of the use of transformations. Fig. 1(a) presents a sequential source program with two nested loops. The loop nest is *imperfect*: not all statements belong to the innermost loop body.
Fig. 1(b) depicts the iteration domain of the source program, as explained in the previous section. The arrows represent dependences and impose a partial order on the loop steps. Apart from these ordering constraints, steps can be executed in any order or in parallel.

In the source iteration domain, parallelism is in some sense hidden. The loop on j is sequential, since the value stored in $A(i,j)$ at iteration j is used as $A(i,j-1)$ in the next iteration. The same is true for the loop on i. However, parallelism can be made visible by applying a skewing transformation as in Fig. 1(c). For a given value of t, there are no dependences between iterations of the p loop, which is therefore parallel. The required transformation can be viewed as a change of coordinates or a renaming:

$$S_1 : \begin{pmatrix} t \\ p \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} -2 \\ -1 \end{pmatrix} \quad S_1 : \begin{pmatrix} t \\ p \end{pmatrix} = \begin{pmatrix} 2 \\ 1 \end{pmatrix} (\ i \ ) + \begin{pmatrix} m-1 \\ m \end{pmatrix}$$

Observe that the transformation for S_1 has the non-singular matrix $\begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}$ and, hence, is bijective. Furthermore, the determinant of this matrix is 1 (the matrix is *unimodular*), which means that the transformation is bijective in the integers.
A target loop nest which corresponds to the target iteration domain is depicted in Fig. 1(d). The issue of target code generation is addressed later. For now, just note that the target loop nest is much more complex than the source loop nest, and that it would be cumbersome and error-prone to derive it manually. On the other hand, the fact that both transformation matrices are unimodular simplifies the target code: both loops have unit stride.

The search for a transformation

The fundamental constraint on a transformation in the polyhedron model is *affinity*. As explained before, each row of the transformation matrix corresponds to one axis of the target coordinate system. Each axis represents either a sequential loop or a parallel loop. Iterations of a sequential loop are executed successively; hence, the loop counter can be interpreted as (logical) time. Iterations of a parallel loop are executed simultaneously (available resources permitting) by different processors; the values of their loop counters correspond to processor names. Finding the coefficients for the sequential axes constitutes a problem of *scheduling*, finding the coefficients for the parallel axes one of *placement* or *allocation*. Different methods exist for solving these two problems.
The order in which sequential and parallel loops are nested is important. One can show that it is always possible to move the parallel loops deeper inside the loop nest, which generates lock-step parallelism, suitable for vector or VLIW processors. For less tightly coupled parallelism, suitable for multicores or message-passing architectures, one would like to move the parallel loops farther out, but this is not always possible.

Scheduling

A schedule maps each instance in the iteration domain to a logical date. In contrast to what happens in task graph scheduling (see the corresponding entry), the number of instances is large, or unknown at compile time, or even infinite, so that it is impossible to tabulate this mapping. The schedule must be a closed-form function of the iteration

vector; we will see presently that its determination is easy only if restricted to affine functions.
Let $\theta_R(i)$ be the schedule of instance $\langle R, i\rangle$. Since the source of a dependence must be executed before its destination, the schedule must satisfy the following *causality constraint*:

$$\forall i, j : \langle R, i\rangle \, \delta \, \langle S, j\rangle \Rightarrow \theta_R(i) < \theta_S(j).$$

There are as many such constraints as there are dependences in the program. The unknowns are the coefficients of θ_R and θ_S. The first step in the solution is the elimination of the quantifiers on i and j. There are general methods of quantifier elimination [38] but, due to the affinity of the constraints in the polyhedron model, more efficient methods can be applied. In fact, the form of the causality constraint above asserts that the affine *delay* $\theta_S(j) - \theta_R(i)$ must be positive inside the *dependence polyhedron* $\{i, j \mid \langle R, i\rangle \, \delta \, \langle S, j\rangle\}$. To this end, it is necessary and sufficient that the delay be positive at the vertices of the dependence polyhedron, or that it be an affine positive combination of the dependence constraints (Farkas lemma). The result of quantifier elimination is a linear system of inequalities which can be solved by any linear programming tool. This system of constraints may not be *feasible*, i.e. it may have no solution. This means simply that no linear-time parallel execution exists for the source program. The solution is to construct a multidimensional schedule. In the target loop nest, there will be as many sequential loops as the schedule has dimensions.
More information on scheduling can be found in the [Scheduling Algorithms] entry of this encyclopedia.

Placement

A placement maps each instance to a (virtual) processor number. Again, this mapping must be in the form of a closed affine function. In contrast to scheduling, there is no legality constraint for placements: any placement is valid, but may be inefficient.
For each dependence between instances that are assigned to distinct processors, one must generate a communication or a synchronization, depending on whether the target architecture has distributed or shared memory. These are costly operations, which must be kept at a minimum. Hence, the aim of a placement algorithm is to find a function:

$$\pi : E \rightarrow [0, P[$$

where P is the number of processors, such that the size of the set:

$$\mathcal{C} = \{u, v \in E \mid u \, \delta \, v, \pi(u) \neq \pi(v)\}$$

is minimal. Since counting integer points inside a polyhedron is difficult, one usually uses the following heuristics: try to "cut" as many dependences as possible. A dependence from statement R to S can be *cut* if the following constraint holds:

$$\langle R, i\rangle \, \delta \, \langle S, j\rangle \Rightarrow \pi_R(i) = \pi_S(j).$$

This condition can be transformed into a system of homogeneous linear equations for the coefficients of π. The problem is that, in most cases, if one tries to satisfy all the

cutting constraints, the only solution is $\pi(u) = 0$, which corresponds to execution on only one processor: this, indeed, results in the minimal number of synchronizations (namely zero)! A possible way out is to solve the cutting constraints one at time, in order of decreasing size of the dependence polyhedron, and to stop just before generating the trivial solution. The uncut dependences induce synchronization operations. If all dependences can be cut, the program has *communication-free parallelism* and can be rewritten with one or more outermost parallel loops.
In the special case of a perfect loop nest with uniform dependences, one may approximate the dependence graph by the translations of the lattice generated by the dependence vectors. If the determinant of this lattice is larger than 1, the program can be split into as many independent parts [17].
Lastly, instead of assigning a processor number to each instance, one may assign all iterations of one statement to the same processor [35, 47]. This results in the construction of a Kahn process network [33].

Code generation

In the polyhedron model, a transformation of the source iteration domain can be found automatically, which optimizes some objective function. The highest execution speed (i.e. the minimum number of steps to be executed in sequence) may be the first thing that comes to mind, but many other functions are possible.
Unfortunately, it is not trivial to generate efficient target code from the optimal solution in the model. There are several factors that can degrade performance seriously. The enumeration of the points in the target iteration domain involves tests for the lower and upper border. If the code is not chosen wisely, these tests will often degrade scalability. For example, in Fig. 1, a maximum and a minimum is involved. The example of Fig. 1 also shows that additional control (the IF statement) may be introduced, which degrades performance. Of course, synchronizations and communications can also degrade performance seriously.
For details on code generation in the polyhedron model, see the [Parallel Code Generation] entry.

Extensions

The following extensions have successively been made to the basic polyhedron model.

WHILE loops

The presence of a WHILE loop in the loop nest turns the iteration domain from a finite set (a polytope) into an infinite set (a polyhedron). If the control dependence that the termination test of the loop imposes is being respected, the iteration must necessarily be sequential. However, the steps of a WHILE loop in a nest with further (FOR or WHILE) loops may be distributed in space. There have been two approaches to the parallelization of WHILE loops.

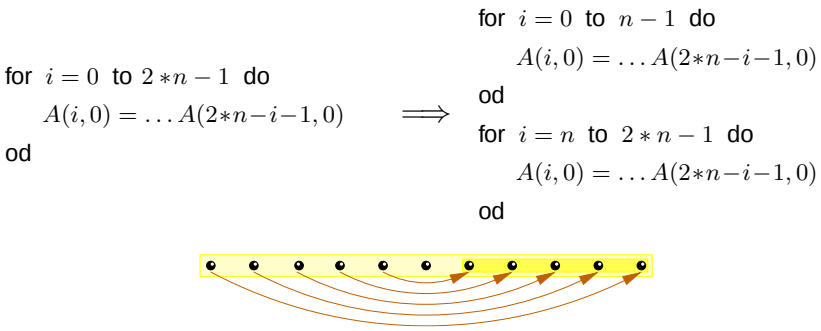


Figure 2: Iteration domain splitting

The *conservative approach* [22, 25] respects the control dependence. One challenge here is the discovery of global termination. The *speculative approach* [14] does not respect the control dependence. Thus, several loop steps may be executed in parallel if there is no other dependence between them. The price paid is the need for storage of intermediate results, in case a rollback needs to be done when the point of termination has been discovered but further steps have already been executed. In some cases, overshooting the termination point does not jeopardize the correctness of the program and no rollback is needed. Discovering this property is beyond the capability of present compilers.

Conditional statements

The basic model permits only assignment statements in the loop body. The challenge of conditionals is that a dependence may hold only for certain executions, i.e., not for all branches. A static analysis can only reveal the union of these dependences [13].

Iteration domain splitting

In some cases, the schedule can be improved by orders of magnitude if one splits the iteration domain in appropriate places [24]. One example is depicted in Fig. 2. With the best affine schedule of $\lfloor i/2 \rfloor$ each parallel step contains two loop iterations, i.e. the execution is sped up by a factor of 2. (The reason is that the shortest dependence has length 2.). The domain split on the right yields two partitions, each without dependences between its iterations. Thus, all iterations of the upper loop (enumerating the left partition) can be executed in a first parallel step, and the iterations of the lower loop (enumerating the right partition) in a second one, for a speedup of $n/2$

Tiling

The technique of domain splitting has a further, larger significance. The polyhedron model is prone to yielding very fine-grained parallelism. To coarsen the grain when not enough processors are available, one partitions (parts of) the iteration domain in

equally sized and shaped *tiles*. Each tile covers a set of iterations and the points in a tile are enumerated in time rather than in space, i.e., the iteration over a tile is resequentialized.
One can tile the source iteration domain or the target iteration domain. In the latter case, one can tile space and also time. Tiling time corresponds to adding hands to a clock and has the effect of coarsening the grain of processor communications. The habilitation thesis of Martin Griebl [23] offers a comprehensive treatment of this topic and an extensive bibliography. See also the [Tiling] entry of this encyclopedia.

Treatment of expressions

In the basic model, expressions are considered atomic. There is an extension of the polyhedron model to the parallelization of the evaluation of expressions [18]. It also permits the identification of common subexpressions and provides a means to choose automatically the suitable point in time and the suitable place at which to evaluate it just once. Its value is then communicated to other places.

Relaxations of affinity

The requirement of affinity enters everywhere in the polyhedron model: in the loop bounds, in the array index expressions, in the transformations. Quickly, after the polyhedron model had been developed, the desire arose to transcend affinity in places. Iteration domain splitting is one example.
Lately, a more encompassing effort has been made to leave affinity behind. One circumstance that breaks the affinity of index expressions is that the so-called *structure parameters* (e.g. variables n and m in the loops of Fig. 1 and 2) enter multiplicatively as unevaluated variables, not as constants. For example, when a two-dimensional array is linearized, array subscripts are of the form $n\,i+j$ with i, j being the loop iterators. As a consequence, subscript equations are non-linear in the structure parameters, too. An algorithm for computing the solutions of equation systems with exactly one such structure parameter exists [29].
In transformations and code generation, non-linear structure parameters, as in expressions $n\,i$, $n^2\,i$ or $n\,m\,i$, can be handled by generalizing existing algorithms (for the case without non-linear parameters) using quantifier elimination [28]. Code generation can even be generalized to handle non-linear loop indices, as in $n\,i^2$, $n^2\,i^2$ or $i\,j$. To this end, cylindrical algebraic decomposition (CAD) [27], which corresponds to Fourier-Motzkin elimination in the basic model, is used for computing loops nests which enumerate the points in the transformed domains efficiently. This extends the frontier of code generation to arbitrary polynomial loop bounds.

Applications other than loop parallelization

Array expansion

It is easy to see that, if a loop modifies a scalar, there is a dependence between any two iterations, and the loop must remain sequential. When the modification occurs early in the loop body, before any use, the dependence can be removed by expanding the scalar to a new array, with the loop counter as its subscript. This idea can be extended to all cases in which a memory cell – be it a scalar or part of an array – is modified more than once. The transformation proceeds in two steps:

- Replace the left side of each assignment by a fresh array, subscripted by the counters of all enclosing loops.
- Inspect all the right sides and replace each reference by its *source* [20].

The source of a use is the latest modification that precedes the use in the sequential execution order. It can be computed by *parametric integer programming*. The result of this transformation is a program in *dynamic single-assignment form*. Each memory cell is written to just once in the course of a program execution. As a consequence, the sets $\mathcal{W}(u) \cap \mathcal{W}(v)$ are always empty: the transformed program has far fewer dependences and, occasionally, much more parallelism than the original.

Array shrinking

A consequence of the previous transformation is a large increase in the memory footprint of the program. In many cases, the same degree of parallelism can be achieved with less expansion, or the target architecture cannot exploit all parallelism there is, and some of the parallel loops have to be sequentialized. Another situation, in a purely sequential context, is when a careless programmer has used more memory than strictly necessary to implement an algorithm.
The aim of *array shrinking* is to detect these situations, and to reduce the memory needs by inserting modulo operators in subscripts. Suppose, for instance, that in the following code:

for $i = 0$ to $n-1$ do
  $a[i] = ...$;
od

one replaces $a[i]$ by $a[i \mod 16]$. The dimension of a, which is n in the first version, is reduced to 16 in the second version. Of course, this means that the value stored in $a[i]$ is destroyed after sixteen iterations of the loop. This transformation may change the outcome of the program, unless one can prove that the *lifetime* of $a[i]$ does not exceed sixteen iterations.
Finding an automatic solution to this problem has been the subject of much work since 1990 (Darte [16] offers a good discussion). The proposed solution is to construct an interference polyhedron for the elements of a fixed array, and to cover it by a maximally tight lattice such that only the lattice origin falls inside the polyhedron. The basis

vectors of the lattice are taken as coordinate axes of the reduced array, and their lengths are related to the modulus of the new subscripts.

Communication Generation

When constructing programs for distributed memory architectures, be it with data distribution directives in languages like High-Performance Fortran (HPF) or under the direction of a placement function, one has to generate communication code. It so happens that this is also a problem in polyhedron scanning. It can be solved with the same techniques and the same tools that are used for code generation.

Locality enhancement

Most modern processors have caches: small but fast memories that retain a copy of recently accessed memory cells. A program has locality if memory accesses are clustered such that there is a high likelihood of finding a copy of the needed information in cache rather than in main memory. Improving the locality of a program is highly beneficial for performance, since caches are usually accessed in one cycle while memory latency may range from ten to a hundred cycles.
Since the cache controller returns old copies to memory in order to find room for new ones, locality is enhanced by changing the execution order such that the *reuse distance* between successive accesses to the same cell is minimal. This can be achieved, for instance, by moving all such accesses to the innermost loop of the program [49].
Another approach consists of dividing a program into *chunks* whose memory footprints are smaller than the cache size. Conceptually, the program is executed by filling the cache with the necessary data for one chunk, executing the chunk without any cache miss, and emptying the cache for the next chunk. One can show that the memory traffic will be minimal if each datum belongs to the footprint of only one chunk. The construction of chunks is somewhat similar to scheduling [7]. It is enough to have asymptotic estimates of the footprint sizes. One advantage of this method is that it can be adapted easily to the management of *scratchpad memories*, software-controlled caches as can be found in embedded processors.

Dynamic optimization

Dynamic optimization resulted from the observation that modern processors and compilers are so complex that building a realistic performance estimator is nearly impossible. The only way of evaluating the quality of a transformed program is to run it and take measurements.
In the polyhedron model, one can define the polyhedron of all legal schedules (see the previous section on scheduling). Usually, one selects one schedule in this polyhedron according to some simple objective function. Another possibility is to generate one program for each legal schedule, measure its performance, and retain the best one. Experience shows that, in many cases, the best program is unexpected, the proof of its

legality is not obvious, and the reasons for its efficiency are difficult to fathom. As soon as the source program has more than a few statements, the size of the polyhedron of legal schedules explodes: sophisticated techniques including genetic algorithms and machine learning are needed to restrict the exploration to “interesting” solutions [39, 40].

Tools

There is a variety of tools which support several phases in the polyhedral parallelization process.

Mathematical support

PIP [19] is an all integer implementation of the Simplex algorithm, augmented with Gomory cuts for integer programming [43]. The most interesting feature of PIP is that it can solve parametric problems, i.e. find the lexicographic minimal x such that

$$Ax \leq By + c$$

as a function of y.
Omega [41] is an extension of the Fourier-Motzkin elimination method to the case of integer variables. It has been extended into a fully fledged tool for the manipulation of Presburger formulas (logical formulas in which the atoms are affine constraints on integer variables).
There are many so-called polyhedral libraries; the oldest one is the PolyLib [11]. The core of these libraries is a tool for converting a system of affine constraints into the vertices of the polyhedron it defines, and back. The PolyLib also includes a tool for counting the number of integer points inside a parametric polyhedron, the result being an Ehrhart polynomial [10]. More recent implementations of these tools, occasionally using different algorithms, are the Parma Polyhedral Library [2], the Integer Set Library [46], the Barvinok Library [48], and the Polka Library [32]. This list is probably not exhaustive.

Code generation

CLooG [6] takes as input the description of an iteration domain, in the form of a disjoint union of polyhedra, and generates an efficient loop nest that scans all the points in the iteration domain in the order given by a set of scattering functions, which can be schedules, placements, tiling functions and more. For a detailed description of CLooG, see the [Parallel Code Generation] entry in this encyclopedia.

Fully-fledged loop restructurers

LooPo [26] was the first polyhedral loop restructurer. Work on it was started at the University of Passau in 1994 and it was developed in steps over the years and is still

being extended. LooPo is meant to be a research platform for trying out and comparing different methods and techniques based on the polyhedron model. It offers a number of schedulers and allocators and generates code for shared-memory and distributed memory architectures. All of the extensions mentioned above have been implemented and almost all are being maintained.
Pluto [9] was developed at Ohio-State University. Its main objective is to use placement functions to improve locality, and to integrate tiling into the polyhedron model. Its target architectures are multicores and graphical processing units (GPUs).
GRAPHITE [45] is an extension of the GCC compiler suite whose ultimate aim is to apply polyhedral optimization and parallelization techniques, where possible, to run-of-the-mill programs. Graphite looks for static control parts (SCoPs) in the GCC intermediate representation, generates their polyhedral representation, applies transformations, and generates target code using CLooG. At the time of writing, the set of available transformations is still rudimentary, but is supposed to grow.

RELATED ENTRIES

Banerjee test
Dependence abstractions
Dependence analysis
High-Performance Fortran
Loop-level speculation
Loop nest parallelization
Loop scheduling
OpenMP
Parallel Code Generation
Scheduling algorithms
Task Graph Scheduling
Tiling

BIBLIOGRAPHIC NOTES AND FURTHER READING

The development of the polytope model was driven by two nearly disjoint communities. Hardware architects wanted to take a set of recurrence equations, expressing, for instance, a signal transformation, and derive a parallel processor array from it. Compiler designers wanted to take a sequential loop nest and derive parallel loop code from it. One can view the seed of the model for architecture in the seminal paper by Karp, Miller and Winograd on analyzing recurrence equations [34] and the seed for software in the seminal paper by Lamport on Fortran DO loop parallelization [36]. Lamport used hyperplanes (the slices in the polyhedron that make up the parallel steps), instead of polyhedra. In the early Eighties, Quinton drafted the components of the polyhedron model [42], still in the hardware context (at that time: systolic arrays).

The two communities met around the end of the Eighties at various workshops and conferences, notably the International Conference on Supercomputing and CONPAR and PARLE, the predecessors of the Euro-Par series. Two developments made the polyhedron model ready for compilers: parametric integer programming, worked out by Feautrier [19], which is used for dependence analysis, scheduling and code generation, and seminal work on code generation by Irigoin et al. [1, 31]. Finally, Lengauer [37] gave the model its name.

The Nineties saw the further development of the theory underlying the model's methods, particularly for scheduling, placement and tiling. Extensions and applications other than loop parallelization came mainly in the latter part of the Nineties and in the following decade.

A number of textbooks focus on polyhedral methods. There is the three-part series of Banerjee [3, 4, 5], a book on tiling by Xue [50] and a comprehensive book on scheduling by Darte, Robert and Vivien [15]. Collard [12] applies the model to the optimization of loop nests for sequential as well as parallel execution and studies a similar model for recursive programs.

In the past several years, the polyhedron model has become more mainstream. The seed of this development was an advance in code generation methods [6]. With the GCC community taking an interest, it is to be expected that polyhedral methods will increasingly find their way into production compilers.

BIBLIOGRAPHY

[1] Corinne Ancourt and François Irigoin. Scanning polyhedra with DO loops. In *Proc. Third ACM SIGPLAN Symp. on Principles & Practice of Parallel Programming (PPoPP)*, pages 39–50. ACM Press, 1991.

[2] Roberto Bagnara, Patricia M. Hill, and Enea Zaffanella. The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Science of Computer Programming*, 72(1–2):3–21, 2008. Web site: http://www.cs.unipr.it/ppl.

[3] Utpal Banerjee. *Loop Transformations for Restructuring Compilers: The Foundations*. Series on Loop Transformations for Restructuring Compilers. Kluwer, 1993.

[4] Utpal Banerjee. *Loop Parallelization*. Series on Loop Transformations for Restructuring Compilers. Kluwer, 1994.

[5] Utpal Banerjee. *Dependence Analysis*. Series on Loop Transformations for Restructuring Compilers. Kluwer, 1997.

[6] Cédric Bastoul. Code generation in the polyhedral model is easier than you think. In *Proc. 13th Int. Conf. on Parallel Architecture and Compilation Techniques (PACT)*, pages 7–16. IEEE Computer Society Press, September 2004. Web site: http://www.cloog.org/.

[7] Cédric Bastoul and Paul Feautrier. Improving data locality by chunking. In *Compiler Construction (CC)*, Lecture Notes in Computer Science 2622, pages 320–335. Springer-Verlag, 2003.

[8] A. J. Bernstein. Analysis of programs for parallel processing. *IEEE Trans. on Electronic Computers*, EC-15:757–762, October 1966.

[9] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. *SIGPLAN Notices*, 43(6):101–113, 2008. Web site: http://pluto-compiler.sourceforge.net/.

[10] Philippe Clauss. Counting solutions to linear and non-linear constraints through ehrhart polynomials. In *Proc. ACM/IEEE Conf. on Supercomputing*, pages 278–285. ACM Press, 1996.

[11] Philippe Clauss and Vincent Loechner. Parametric analysis of polyhedral iteration spaces, extended version. *J. VLSI Signal Processing*, 19(2):179–194, 1998. Web site: http://icps.u-strasbg.fr/polylib/.

[12] Jean-François Collard. *Reasoning About Program Transformations – Imperative Programming and Flow of Data*. Springer-Verlag, 2003.

[13] Jean-François Collard and Martin Griebl. A precise fixpoint reaching definition analysis for arrays. In Larry Carter and Jean Ferrante, editors, *Languages and Compilers for Parallel Computing (LCPC)*, Lecture Notes in Computer Science 1863, pages 286–302. Springer-Verlag, 1999.

[14] Jean-François Collard. Automatic parallelization of `while`-loops using speculative execution. *Int. J. Parallel Programming*, 23(2):191–219, 1995.

[15] Alain Darte, Yves Robert, and Frédéric Vivien. *Scheduling and Automatic Parallelization*. Birkhäuser, 2000.

[16] Alain Darte, Robert Schreiber, and Gilles Villard. Lattice-based memory allocation. *IEEE Trans. on Computers*, TC-54(10):1242–1257, October 2005.

[17] Eric H. D'Hollander. Partitioning and labeling of loops by unimodular transformations. *IEEE Trans. on Parallel and Distributed Systems*, 3(4):465–476, 1992.

[18] Peter Faber. *Code Optimization in the Polyhedron Model - Improving the Efficiency of Parallel Loop Nests*. PhD thesis, Department of Informatics and Mathematics, University of Passau, 2007. http://www.fim.uni-passau.de/cl/publications/docs/Faber07.pdf.

[19] Paul Feautrier. Parametric integer programming. *Operations Research*, 22(3):243–268, 1988. Web site: http://www.piplib.org.

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

[21] Paul Feautrier. Automatic parallelization in the polytope model. In Guy-René Perrin and Alain Darte, editors, *The Data Parallel Programming Model*, Lecture Notes in Computer Science 1132, pages 79–103. Springer-Verlag, 1996.

[22] Martin Griebl. *The Mechanical Parallelization of Loop Nests Containing* while *Loops*. PhD thesis, Department of Mathematics and Informatics, University of Passau, January 1997. http://www.fim.uni-passau.de/cl/publications/docs/Gri96.pdf.

[23] Martin Griebl. *Automatic Parallelization of Loop Programs for Distributed Memory Architectures*. Habilitation thesis, Department of Informatics and Mathematics, University of Passau, June 2004. http://www.fim.uni-passau.de/cl/publications/docs/Gri04.pdf.

[24] Martin Griebl, Paul Feautrier, and Christian Lengauer. Index set splitting. *Int. J. Parallel Processing*, 28(6):607–631, 2000. Special Issue on the Int. Conf. on Parallel Architectures and Compilation Techniques (PACT'99).

[25] Martin Griebl and Christian Lengauer. On the space-time mapping of WHILE-loops. *Parallel Processing Letters*, 4(3):221–232, September 1994.

[26] Martin Griebl and Christian Lengauer. The loop parallelizer loopo – announcement. In David Sehr, editor, *Languages and Compilers for Parallel Computing (LCPC)*, Lecture Notes in Computer Science 1239, pages 603–604. Springer-Verlag, 1997. Web site: http://www.infosun.fim.uni-passau.de/cl/loopo/.

[27] Armin Größlinger. *The Challenges of Non-linear Parameters and Variables in Automatic Loop Parallelisation*. PhD thesis, Department of Informatics and Mathematics, University of Passau, December 2009. http://nbn-resolving.de/urn:nbn:de:bvb:739-opus-17893.

[28] Armin Größlinger, Martin Griebl, and Christian Lengauer. Quantifier elimination in automatic loop parallelization. *J. Symbolic Computation*, 41(11):1206–1221, November 2006.

[29] Armin Größlinger and Stefan Schuster. On computing solutions of linear diophantine equations with one non-linear parameter. In *Proc. 10th Int. Symp. on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*, pages 69–76. IEEE Computer Society Press, September 2008.

[30] Pascale Guerdoux-Jamet and Dominique Lavenier. SAMBA: Hardware accelerator for biological sequence comparison. *Computer Applications in the Biosciences*, 13(6):609–615, 1997.

[31] François Irigoin and Rémy Triolet. Dependence approximation and global parallel code generation for nested loops. In Michel Cosnard, Yves Robert, Patrice Quinton, and Michel Raynal, editors, *Parallel & Distributed Algorithms*, pages 297–308. North-Holland, 1989.

[32] Bertrand Jeannet and Antoine Miné. APRON: A library of numerical abstract domains for static analysis. In *Computed Aided Verification (CAV)*, Lecture Notes in Computer Science 5643, pages 662–667. Springer-Verlag, 2009. Web site: http://apron.cri.ensmp.fr/library.

[33] Gilles Kahn. The semantics of simple language for parallel programming. In *Proc. IFIP Congress*, pages 471–475, 1974.

[34] Richard M. Karp, Raymond E. Miller, and Shmuel Winograd. The organization of computations for uniform recurrence equations. *J. ACM*, 14(3):563–590, July 1967.

[35] Bart Kienhuis, Edwin Rijpkema, and Ed F. Deprettere. Compaan: Deriving process networks from matlab for embedded signal processing architectures. In Frank Vahid and Jan Madsen, editors, *Proc. Eighth Int. Workshop on Hardware/Software Codesign (CODES 2000)*, pages 13–17. ACM Press, 2000.

[36] Leslie Lamport. The parallel execution of DO loops. *Comm. ACM*, 17(2):83–93, February 1974.

[37] Christian Lengauer. Loop parallelization in the polytope model. In Eike Best, editor, *CONCUR'93*, Lecture Notes in Computer Science 715, pages 398–416. Springer-Verlag, 1993.

[38] Rüdiger Loos and Volker Weispfenning. Applying linear quantifier elimination. *The Computer J.*, 36(5):450–462, 1993.

[39] Louis-Noël Pouchet, Cédric Bastoul, Albert Cohen, and John Cavazos. Iterative optimization in the polyhedral model: Part II, multidimensional time. *SIGPLAN Notices*, 43(6):90–100, June 2008.

[40] Louis-Noël Pouchet, Cédric Bastoul, Albert Cohen, and Nicolas Vasilache. Iterative optimization in the polyhedral model: Part I, one-dimensional time. In *IEEE/ACM Fifth Int. Symp. on Code Generation and Optimization (CGO'07)*, pages 144–156. IEEE Computer Society Press, 2007.

[41] William Pugh. The omega test: a fast and practical integer programming algorithm for dependence analysis. In *Proc. 5th Int. Conf. on Supercomputing*, pages 4–13. ACM Press, 1991. Web site: http://www.cs.umd.edu/projects/omega.

[42] Patrice Quinton. The systematic design of systolic arrays. In Françoise F. Soulié, Yves Robert, and Maurice Tchuenté, editors, *Automata Networks in Computer Science*, chapter 9, pages 229–260. Manchester University Press, 1987. Also: Technical Reports 193 and 216, IRISA (INRIA-Rennes), 1983.

[43] Alexander Schrijver. *Theory of linear and integer programming*. John Wiley & Sons, 1986.

[44] Temple F. Smith and Michael S. Waterman. Identification of common molecular subsequences. *J. Molecular Biology*, 147(1):195–197, 1981.

[45] Konrad Trifunovic, Albert Cohen, David Edelsohn, Li Feng, Tobias Grosser, Harsha Jagasia, Razya Ladelsky, Sebastian Pop, Jan Sjödin, and Ramakrishna Upadrasta. GRAPHITE two years after. In *Proc. 2nd Int. Workshop on GCC Research Opportunities (GROW)*, pages 4–19, January 2010. http://gcc.gnu.org/wiki/GROW-2010.

[46] Sven Verdoolaege. An integer set library for program analysis. In *Advances in the Theory of Integer Linear Optimization and its Extensions*. AMS 2009 Western Section, 2009. Web site: http://freshmeat.net/projects/isl/.

[47] Sven Verdoolaege, Hristo Nikolov, Nikolov Todor, and Plamenov Stefanov. Improved derivation of process networks. In *Proc. 4th Int. Workshop on Optimization for DSP and Embedded Systems (ODES)*, 2006. http://www.ece.vill.edu/~deepu/odes/odes-4_digest.pdf.

[48] Sven Verdoolaege, Rachid Seghir, Kristof Beyls, Vincent Loechner, and Maurice Bruynooghe. Counting integer points in parametric polytopes using Barvinok's rational functions. *Algorithmica*, 48(1):37–66, 2007. Web site: http://freshmeat.net/projects/barvinok.

[49] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pages 30–44. ACM Press, 1991.

[50] Jingling Xue. *Loop Tiling for Parallelism*. Kluwer, 2000.