Generation of Parallel Code

Cédric Bastoul

ALCHEMY Group
Laboratoire de Recherche en Informatique
University Paris-Sud 11 – INRIA Saclay Île-de-France
Cedric.Bastoul@lri.fr

BYLINE

Cédric Bastoul
ALCHEMY Group
Laboratoire de Recherche en Informatique
University Paris-Sud 11 – INRIA Saclay Île-de-France
Cedric.Bastoul@lri.fr

SYNONYMS

- Code Generation
- Polyhedra Scanning

DEFINITION

Parallel code generation is the action of building a parallel code from an input sequential code according to some scheduling and placement information. Scheduling specifies the desired order of the statement instances with respect to each other in the target code. Placement specifies the desired target processor for each statement instance.

DISCUSSION

1 Introduction

Exhibiting parallelism in a sequential code may require complex sequences of transformations. When they come from an expert in program optimization, they are usually expressed by way of directives like *tile* or *fuse* or *skew*. When they come from a compiler, they are typically formulated as functions that map every execution of every statement of the program in "time" (*scheduling*), to order them in a convenient way, and in "space" (*placement*), to distribute them amongst various processors.

Parallel code generation is the step in any program restructuring tool or parallelizing compiler that actually generates a target code which implements the user directives or the compiler mapping functions.

This task is challenging in many ways. *Feasibility* is the first challenge. Reconstructing a program with respect to scheduling and placement information at the statement execution level may seem an impossible problem at first sight.

It has been addressed by working on a convenient representation of the problem itself, as it will be described momentarily. *Scalability* is another important issue. As compiler vendors try to ensure that compile time is (almost) linear in the code length, the code generation step must be fast enough to be integrated into production tools. *Quality* of the generated code must be paramount. The generated code should not be too long and should not include heavy control overhead that may offset the optimization it is enabling. Finally, *flexibility* must be provided to allow a large span of transformations on a large set of programs.

2 Representation of the Problem

To solve the code generation problem, it needs to be formalized in some way. A mathematical representation, known as the *polyhedral model* (also referred in the literature as the *polytope model*) is a convenient abstraction. It allows the description of the problem in a compact and expressive way. This representation is also the key to solving it efficiently thanks to powerful and scalable mathematical libraries as described in Section 4. This model, with slight variations, has been used in most successful work on parallel code generation.

Two mathematical objects need to be defined for the parallel code generation problem. First, *iteration domains* provide the relevant information for code generation from the input code, they are detailed in Section 2.1. Second, *space-time mapping functions* provide the ordering and placement information to be implemented by the target code. They are described in Section 2.2.

2.1 Representing Statement Instances: Iteration Domains

The key aspect of the polyhedral model is to consider *statement instances*. A statement instance is *one* particular execution of a statement. Each instance of a statement that is enclosed inside a loop can be associated with the value of the outer loop counters (also called *iterators*). For instance, let us consider the polynomial multiply code in Figure 1: the instance of statement `S1` for $i = 2$ is `z[2] = 0`.

```
do i = 1, n
   z[i] = 0                          ! S1
do i = 1, n
   do j = 1, n
      z[i+j] = z[i+j] + x[i] * y[j] ! S2
```

Fig. 1. Polynomial Multiply Kernel

In the polyhedral model, statements are considered as functions of the outer loop counters that may produce statement instances: instead of simply "`S1`", the notation `S1(i)` is preferred. For instance, statement `S1` for $i = 2$ is written `S1(2)` and statement `S2` for $i = 4$ and $j = 2$ is written $\texttt{S1}\binom{4}{2}$. The vector of the iterator values is called the *iteration vector*.

Obviously, dealing with statement instances does not mean that unrolling all loops is necessary. First because there would probably be too many instances to deal with, and second because the number of instances may not be known. For instance, when the loops are bounded with constants that are unknown at compile time (called "parameters"), e.g., `n` in the example code in Figure 1. A compact way to represent all the instances of a given statement is to consider the set of all possible values of its iteration vector. This set is called the statement's *iteration domain*. It can be conveniently described by all the constraints on the various iterators that the statement depends on. When those constraints are affine and depend only on the outer loop counters and some parameters, the set of constraints defines a *polyhedron* (more precisely this is a $\mathbb{Z}$-*polyhedron*, but *polyhedron* is used for short). Hence the name "polyhedral model". A matrix representation with the following general form for any statement S is used to facilitate the manipulation of the affine constraints:

$$\mathcal{D}_S = \{\boldsymbol{x}_S \in \mathbb{Z}^{n_S} \mid A_S\boldsymbol{x}_S + \boldsymbol{a}_S \geq \mathbf{0}\}$$

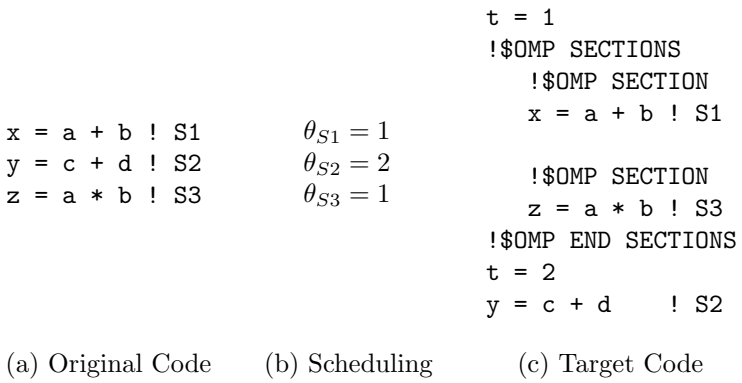
where $\boldsymbol{x}_S$ is the n_S-dimensional iteration vector, A_S is a constant matrix and $\boldsymbol{a}_S$ is a constant vector, possibly parametric. For instance, here are the iteration domains for the polynomial multiply example in Figure 1:

- $\mathcal{D}_{S1} = \left\{ (i) \in \mathbb{Z} \middle| \begin{bmatrix} 1 \\ -1 \end{bmatrix} (i) + \begin{pmatrix} -1 \\ n \end{pmatrix} \geq \mathbf{0} \right\}$
- $\mathcal{D}_{S2} = \left\{ \begin{pmatrix} i \\ j \end{pmatrix} \in \mathbb{Z}^2 \middle| \begin{bmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \end{bmatrix} \begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} -1 \\ n \\ -1 \\ n \end{pmatrix} \geq \mathbf{0} \right\}$

2.2 Representing Order and Placement: Mapping Functions

Iteration domains do not provide any information about the order in which statement instances have to be executed, nor do they inform about the processor that has to execute them. Such information is provided by other mathematical objects called *space-time mapping functions*. They associate each statement instance with a logical date *when* it has to be executed and a processor coordinate *where* it has to be executed. In the literature, the part of those functions dedicated to time is called *scheduling* while the part dedicated to space is called *placement* (or *allocation*, or *distribution*).

In the case of *scheduling*, the logical dates express at which time a statement instance has to be executed, with respect to the other statement instances. It is typically denoted θ_S for a given statement S. For instance, let us consider the three statements in Figure 2(a) and their scheduling functions in Figure 2(b). The first and third statements have to be executed both at logical date 1. This means they can be executed in parallel at date 1 but they have to be executed before the second statement since its logical date is 2. The target code implementing this scheduling using OpenMP pragmas is shown in Figure 2(c), where a fictitious variable t stands for the time. It can be seen that at time $t = 1$, both $S1$ and $S3$ are run in parallel, while $S2$ is executed afterward at time $t = 2$.



(a) Original Code (b) Scheduling (c) Target Code

Fig. 2. One-Dimensional Scheduling Example

Logical dates may be multidimensional, like clocks: the first dimension corresponds to days (most significant), next one is hours (less significant), the third to minutes and so on. The order of multidimensional dates with a decreasing significance for each dimension is called the *lexicographic* order. Again, it is not possible to assign one logical date to each statement instance for two reasons: this would probably lead to an intractable number of logical dates and the number of instances may not be known at compile time. Hence, a more compact representation called the *scheduling function* is used. A scheduling function associates a logical date with each statement instance of a statement. They have the following form for a statement S:

$$\theta_S(\boldsymbol{x}_S) = T_S \boldsymbol{x}_S + \boldsymbol{t}_S,$$

where $\boldsymbol{x}_S$ is the iteration vector, T_S is a constant matrix, and $\boldsymbol{t_S}$ is a constant vector, possibly parametric. Scheduling functions can easily encode a wide range of usual transformations such as skewing, interchange, reversal, shifting tiling etc. Many program transformation frameworks have been proposed on top of such functions, the first significant one being UTF (Unified Transformation Framework) by Kelly and Pugh in 1993 [8].

Placement is similar to scheduling, only the semantics is different: instead of logical dates, a placement function π_S associates each instance of statement S with a processor coordinate corresponding to the processor that has to execute the instance.

A space-time mapping function σ_S is a multidimensional function embedding both space and time information for statement S: some dimensions are devoted to scheduling while some others are dedicated to placement. For instance, a compiler may suggest the following space-time mapping for the polynomial multiply code shown in Section 2.1. Its first dimension is a placement that corresponds to a wavefront parallelism for `S2` and improves locality by executing the initialization of an array element by $S1$ on the same processor where it is used by $S2$. The second dimension is a very simple constant scheduling that ensures the

initialization of the array element is done before its use (it is usual to add the identity schedule at the last dimensions, however this will not be necessary for the continuation of this example):

- $\sigma_{S1}\left(i\right) = \begin{bmatrix}1\\0\end{bmatrix}\left(i\right) + \begin{pmatrix}0\\0\end{pmatrix}$
- $\sigma_{S2}\begin{pmatrix}i\\j\end{pmatrix} = \begin{bmatrix}1&1\\0&0\end{bmatrix}\begin{pmatrix}i\\j\end{pmatrix} + \begin{pmatrix}0\\1\end{pmatrix}$

While working in the polyhedral representation, the semantics of each dimension is not relevant: after code generation, each dimension will be translated to some loops that can be post-processed to become parallel or sequential according to their semantics (obviously semantics information can be used to generate a better code, but this is out of the scope of this introduction).

3 Putting Everything Together

Iteration domains can be extracted directly by analysing of the input code. They represent for each statement the set of their instances. In particular they do not encode any ordering information: iteration domains are nothing but "bags" of unordered statement instances. On the opposite, the space/time mapping functions, typically computed by an optimizing or parallelizing algorithm, provide the ordering information for each statement instance. It is necessary to collect all this information into a polyhedral representation before the actual code generation. There exist two ways to achieve this task. Let us consider an iteration domain defined by the system of affine constraints $A\boldsymbol{x} + \boldsymbol{a} \geq \boldsymbol{0}$ and the transformation function leading to a target index $\boldsymbol{y} = T\boldsymbol{x}$. Any of the following formulas can be chosen to build the target polyhedron $\mathcal{T}$ that embeds both instance and ordering information:

Inverse Transformation By noticing that $\boldsymbol{x} = T^{-1}\boldsymbol{y}$ it follows that the transformed polyhedron in the new coordinate system can be defined by:

$$\mathcal{T} : \left\{\boldsymbol{y} \mid [AT^{-1}]\boldsymbol{y} + \boldsymbol{a} \geq \boldsymbol{0}\right\}.$$

Generalized Change of Basis Alternatively, new dimensions corresponding to the ordering in leading positions can be introduced (note that in the following formula, constraints "above" the line are equalities while constraints "under" the line are inequalities):

$$\mathcal{T} : \left\{ \begin{pmatrix}\boldsymbol{y}\\\boldsymbol{x}\end{pmatrix} \;\middle|\; \left[\begin{array}{c|c}I & -T\\\hline 0 & A\end{array}\right]\begin{pmatrix}\boldsymbol{y}\\\boldsymbol{x}\end{pmatrix} + \begin{pmatrix}-\boldsymbol{t}\\\boldsymbol{a}\end{pmatrix} \begin{array}{c}=\\\hline\geq\end{array} \boldsymbol{0} \right\}.$$

The inverse transformation solution has been introduced since the seminal work on parallel code generation by Ancourt and Irigoin [1]. It is simple and compact but has several issues: the transformation matrix must be invertible, and even when it is invertible, the target polyhedra may embed some integer points that have no corresponding elements in the iteration domain (this happens when

the transformation matrix is not unimodular, i.e., whose determinant is neither $+1$ or -1). This necessitates specific code generation processing, briefly discussed in Section 4.1. The second formula is attributed to Le Verge, who named it the *Generalized Change of Basis* [11]. It does not require any property on the transformation matrix. Nevertheless, the additional dimensions may increase the complexity of the code generation process. It has been rediscovered independently from Le Verge's work and used in production code generators only recently [2]. Both formulas are used, and possibly mixed, in current code generation tools, depending on the desired transformation properties. For instance, to apply the space-time mapping of the polynomial multiply proposed in Section 2.2, it is convenient to use the Generalized Change of Basis because the transformation matrices are not invertible:

$$- \ \mathcal{T}_{S1} = \left\{ \begin{pmatrix} p \\ t \\ \hline i \end{pmatrix} \in \mathbb{Z}^2 \left| \left[\begin{array}{cc|c} 1 & 0 & -1 \\ 0 & 1 & 0 \\ \hline 0 & 0 & 1 \\ 0 & 0 & -1 \end{array}\right] \begin{pmatrix} p \\ t \\ \hline i \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ \hline -1 \\ n \end{pmatrix} \begin{matrix} = \\ \hline \geq \end{matrix} \mathbf{0} \right. \right\}$$

$$- \ \mathcal{T}_{S2} = \left\{ \begin{pmatrix} p \\ t \\ i \\ j \end{pmatrix} \in \mathbb{Z}^3 \left| \left[\begin{array}{cc|cc} 1 & 0 & -1 & -1 \\ 0 & 1 & 0 & 0 \\ \hline 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & -1 \end{array}\right] \begin{pmatrix} p \\ t \\ i \\ j \end{pmatrix} + \begin{pmatrix} 0 \\ -1 \\ \hline -1 \\ n \\ -1 \\ n \end{pmatrix} \begin{matrix} = \\ \hline \geq \end{matrix} \mathbf{0} \right. \right\}$$

In the target polyhedra, whatever the chosen formula, the order of the dimensions is meaningful: the ordering is encoded as the lexicographic order of the integer points. Thus, the parallel code generation problem is reduced to generating a code that enumerates the integer points of several polyhedra, with respect to the lexicographic ordering.

4 Scanning Polyhedra

Once the target code information has been encoded into some polyhedra that embed the iteration spaces as well as the scheduling and placement constraints, the code generation problem translates to a *polyhedra scanning* problem. The problem here is to find a code (preferably efficient) visiting each integral point of each polyhedra, once and only once, with respect to the lexicographic order. Three main methods have been successful in doing this. Fourier-Motzkin elimination-based techniques have been the very first, introduced by the seminal work of Ancourt and Irigoin [1]. They are discussed briefly in Section 4.1. While Fourier-Motzkin-based techniques aim at generating loop nests, an alternative method based on Parametric Integer Programming has been suggested by Boulet and Feautrier to generate lower-level codes [3]. This method is discussed briefly in Section 4.2. Lastly, Quilleré, Rajopadhye and Wilde showed how to take advantage of high-level polyhedral operations to generate efficient codes directly [14]. As this later technique is now widely adopted in production environments, it is discussed in some depth in Section 4.3.

4.1 Fourier-Motzkin Elimination-Based Scanning Method

Ancourt and Irigoin [1] proposed in 1991 the first solution to the polyhedron scanning problem. Their work is based on the Fourier-Motzkin pair-wise elimination technique [15]. The scope of their method was quite restrictive since it could be applied to only one polyhedron, with unimodular transformation matrices. The basic idea was, for each dimension from the first one (outermost) to the last one (innermost), to project the polyhedron onto the axis and to deduce the corresponding loop bounds. For a given dimension i_k, the Fourier-Motzkin algorithm can establish that $L(i_1, ..., i_{k-1}) + \boldsymbol{l} \leq c_k i_k$ and $c_k i_k \leq U(i_1, ..., i_{k-1}) + \boldsymbol{u}$, where L and U are constant matrices, $\boldsymbol{l}$ and $\boldsymbol{u}$ are constant vectors of size m_l and m_u respectively, and c_k is a constant. Thus, the corresponding scanning code for the dimension i_k can be derived:

```
...
do i_k  =  MAX_{j=1}^{m_l} ⌈(L_j(i_1, ..., i_{k-1}) + l_j)/c_k⌉,
   i_k  ≤  MIN_{j=1}^{m_u} ⌊(U_j(i_1, ..., i_{k-1}) + u_j)/c_k⌋
    ...
    Body
```

The main drawback of this method is the large amount of redundant control since eliminating a variable with the Fourier-Motzkin algorithm may generate up to $n^2/4$ constraints for the loop bounds where n is the initial number of constraints. Many of those constraints are redundant and it is necessary to remove them for efficiency.

Most further works tried to extend this first technique in order to reduce the redundant control and to deal with more general transformations. Le Fur presented a new redundant constraint elimination policy by using the simplex method [10]. Li and Pingali [13] as well as several other authors proposed to relax the unimodularity constraint of the transformation to an invertibility constraint by using the Hermite Normal Form [15] to avoid scanning "holes" in the polyhedron. Griebl, Lengauer and Wetzel [5] relaxed the constraints of code generation further to transformation matrices with non-full rank, and also presented preliminary techniques for scanning several polyhedra using a single loop nest. Finally, Kelly, Pugh and Rosser showed how to scan several polyhedra in the same code by generating a naive perfectly nested code and then (partly) eliminating redundant conditionals [9]. Their implementation relies on an extension of the Fourier-Motzkin technique called the Omega test. The implementation of their algorithm within the Omega Calculator is one of the most popular parallel code generators [7].

4.2 Parametric Integer Programming-Based Scanning Method

Boulet and Feautrier proposed in 1998 a parallel code generation technique which relies on Parametric Integer Programming (PIP for short) to build a code for scanning polyhedra [3]. The PIP algorithm computes the lexicographic minimal integer point of a polyhedron. Because the minimum point may not be the same depending on the parameter values, it is returned as a tree of conditions on the

parameters where each leaf is either the solution for the corresponding parameter constraints or $\perp$ (called *bottom*), i.e., no solution for those parameter constraints.

The basic idea of the Boulet and Feautrier algorithm (in the simplified case of scanning one polyhedron) is to find the first integer point of the polyhedron, called *first*, then to build a function *next* which for a given integer point returns the next integer point in the polyhedron according to the lexicographic ordering. Both *first* and *next* computations can be expressed as a problem of finding the lexicographic minimum in a polyhedron. Finally, the code can be built according to the following canvas, where x is an integer point of the polyhedron that represents the iteration domain:

```
   x = first
1  if x = ⊥ then goto 2
   Body
   x = next
   goto 1
2  ...
```

Generalizing this method to many polyhedra implies combining the different trees of conditions and subsequent additional control cost and code duplication. While this technique has no widely used implementation, it is quite different than the others since it does not aim at generating high-level loop statements. This property may be relevant for specific targets, e.g., when the generated code is not the input of a compiler but of a high-level synthesis tool.

4.3 QRW-Based Scanning Method

Quilleré, Rajopadhye and Wilde proposed in 2000 the first code generation algorithm that builds a target code without redundant control *directly* [14]. While previous schemes started from a generated code with some redundant control and then tried to improve it, their technique (referred as the QRW algorithm) never fails at removing control, and the processing is easier. Eventually it generates a better code more efficiently.

The QRW algorithm is a generalization to several polyhedra of the work of Le Verge, Van Dongen and Wilde on loop nest synthesis using polyhedral operations [12]. It relies on high-level polyhedral operations (like polyhedral intersection, union, projection etc.) which are available in various existing polyhedral libraries. The basic mechanism is, starting from (1) the list of polyhedra to scan and (2) a polyhedron encoding the constraints on the parameters called the *context*, to recursively generate each level of the abstract syntax tree of the scanning code (AST).

The algorithm is sketched in Figure 3 and a simplified example is shown in Figures 4, 5 and 6. It corresponds to the generation of the code implementing the polynomial multiply space-time mapping introduced in Section 2.2. Its input is the list of polyhedra to scan, the context and the first dimension to scan. This corresponds to Figure 4 in our example, with the first dimension to scan being p. The first step of the algorithm intersects the polyhedra with the context to

QRW: build a polyhedron scanning code AST without redundant control.

Input: a polyhedron list, a context C, the current dimension d.
Output: the AST of the code scanning the input polyhedra.

1. Intersect each polyhedron in the list with the context C;
2. Project the polyhedra onto the outermost d dimensions;
3. Separate these projections into disjoint polyhedra (this generates loops for dimension d and new lists for dimension $d+1$);
4. Sort the loops to respect the lexicographic order;
5. Recursively generate loop nests that scan each new list with dimension $d+1$, under the context of the dimension d;
6. Return the AST for dimension d.

Fig. 3. Sketch of the QRW Code Generation Algorithm

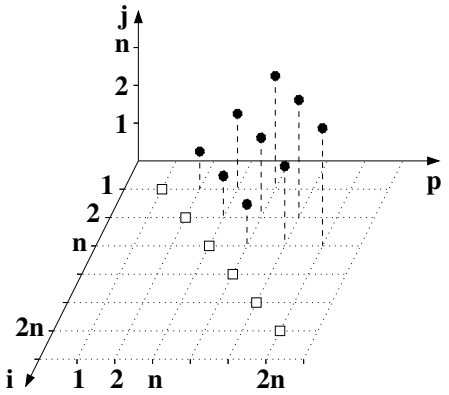
ensure no instance outside the context will be executed. Then it projects them onto the first dimension and separates the projections into disjoint polyhedra. For instance, for two polyhedra, this could correspond to one domain where the first polyhedron is "alone", one domain where the second polyhedron is "alone" and one domain where the two polyhedra coexist. This is depicted in the Figure 5 for our example: it depicts the projection onto the p axis and the separation (it can be seen here that the domain where S_2 is "alone" is empty). The constraints on dimension p for the resulting polyhedra give directly the loop bounds. As the semantics of the placement dimension is to distribute instances across different processors, this loop is parallel. Then the algorithm recursively generates the next dimension loops for each disjoint polyhedron separately. The final result is shown in Figure 6 for our example.

The QRW algorithm is simple and efficient in practice, despite the high theoretical complexity of most polyhedral operations. However, in its basic form, it tends to generate codes with costly modulo operations, and the separation process is likely to result in very long codes. Several extensions to this algorithm have been proposed to overcome those issues [2, 16]. CLooG, a popular implementation of the extended QRW technique demonstrated effectiveness of the algorithm [2]. It is now used in production environments such as in GCC or IBM XL.

5 Parallel Code Generation Today

For a long time, scheduling and placement techniques were many steps forward code generation capabilities. In 1992, Feautrier provided a general scheduling technique for multiple polyhedra and general affine functions [4]. At this time, the only code generation algorithm available had been designed in 1991 by Ancourt and Irigoin and supported only one polyhedron and unimodular scheduling functions [1]. Some scheduling functions had to wait for nearly one decade to be successfully applied by a code generator.

Since then, the challenge of feasibility has been tackled: state of the art parallel code generators can handle any affine transformation for many iteration domains.

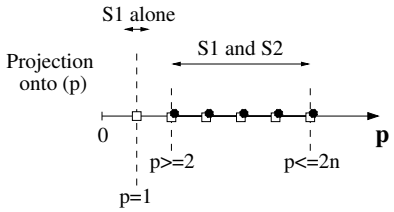


Context: $n \geq 3$

$$\mathcal{T}_{S_1} : \begin{cases} p = i \\ t = 0 \\ 1 \leq i \leq 2n \end{cases}$$

$$\mathcal{T}_{S_2} : \begin{cases} p = i + j \\ t = 1 \\ 1 \leq i \leq n \\ 1 \leq j \leq n \end{cases}$$

Fig. 4. QWR Code Generation Example (1/3): Polyhedra to Scan and Context Information. The graphical representation does not show the degenerated scheduling dimension t.



`doall p = 1, 1`

$$\mathcal{T}'_{S_1} : \begin{cases} t = 0 \\ i = 1 \end{cases}$$

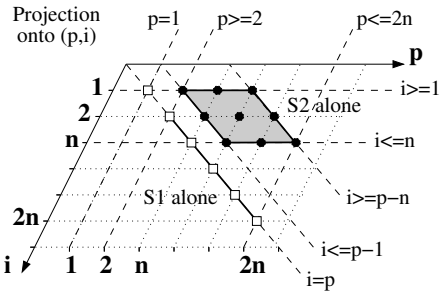
`doall p = 2, 2*n`

$$\mathcal{T}''_{S_1} : \begin{cases} t = 0 \\ i = p \end{cases}$$

$$\mathcal{T}'_{S_2} : \begin{cases} t = 1 \\ 1 \leq i \leq n \\ 1 \leq j \leq n \\ i + j = p \end{cases}$$

Fig. 5. QWR Code Generation Example (2/3): Intersection with the Context, Projection and Separation onto the First Dimension. Two disjoint polyhedra are created: one where S_1 is alone on p (it has only one integer point but a loop is generated to scan it, for consistency) and one where S_1 and S_2 are together on p. In the right side, the new polyhedra to scan have been intersected with the context (for the next step, p is a parameter as well as n).

Moreover, the scalability of code generators is good enough to enable parallel code generation as an option in production compilers. However, the quality of the generated code is still not guaranteed. Summarily, code generators are very good for simple (unimodular) transformations, reasonably good when the coefficients of the transformation functions are small and unpredictable in the general case. The flexibility challenge is also solved only partly because only regular codes that fit the polyhedral model can be processed and only affine transformations can be applied.




```
doall p = 1, 1
  do t = 0, 0
    do i = 1, 1
      z[i] = 0                    ! S1
doall p = 2, 2*n
  do t = 0, 0
    do i = p, p
      z[i] = 0                    ! S1
  do t = 1, 1
    do i = max(1,p-n), min(p-1,n)
      do j = p-i, p-i
        z[i+j] += x[i] * y[j] ! S2
```

Fig. 6. QWR Code Generation Example (3/3): Recursion on the Next Dimensions. First, the projection/separation on (p, t) is done. It is trivial because t is a constant in every polyhedron: it only enforces disjonction and ordering of the polyhedra inside the second `doall` loop. Next the same processing is applied for (p, t, i): the loop bounds of the remaining dimensions can be deduced from the graphical representation (the trivial dimension t is not shown).

6 Future Directions

Two challenges of parallel code generation are partly solved: quality and flexibility. To achieve the best results, autoparallelizers have to take into account some constraints related to code generation that may conflict with the extraction of parallelism, e.g., limiting the absolute value of the transformation coefficients or relying on unimodular transformations. However, there exists an infinity of transformations that implement the same mapping but have different properties with respect to code generation. Finding "code generation friendly" equivalent transformations is a promising solution to enhance the generated code quality.

Several directions are under investigation to provide parallel code generation with more flexibility. Irregular extensions have been successfully implemented to some polyhedral code generators and ambitious techniques based on polynomials instead of affine expressions may be the next step for parallel code generation [6].

RELATED ENTRIES

- Omega Test
- Automatic Parallelization
- Loop Nest Parallelization
- The Wolfe and Lam Algorithm
- Unimodular Transformations
- Scheduling Algorithms

BIBLIOGRAPHIC NOTES AND FURTHER READING

We detailed in Section 4 the three main techniques designed for parallel code generation. The reader will find a deeper level of details in the related papers.

Kelly, Pugh and Rosser's paper on code generation for multiple mappings provides the extensive description of the techniques behind the Omega Code Generator [9]. Boulet and Feautrier's paper on code generation without `do`-loops gives thorough depiction of the PIP-based code generation technique [3]. Finally, the details of the most powerful code generation technique known so far are provided in Quilleré, Rajopadhye and Wilde's paper on generation of efficient nested loops from polyhedra [14]. This reading is complemented by Bastoul's paper, which details several extensions to their algorithm and demonstrates robustness of the extended technique for production compilers [2].

References

[1] C. Ancourt and F. Irigoin. Scanning polyhedra with DO loops. In *3rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 39–50, June 1991.

[2] C. Bastoul. Code generation in the polyhedral model is easier than you think. In *IEEE International Conference on Parallel Architectures and Compilation Techniques (PACT'04)*, pages 7–16, Juan-les-Pins, September 2004.

[3] P. Boulet and P. Feautrier. Scanning polyhedra without do-loops. In *IEEE International Conference on Parallel Architectures and Compilation Techniques (PACT'98)*, pages 4–11, Paris, France, October 1998.

[4] P. Feautrier. Some efficient solutions to the affine scheduling problem, part II: multidimensional time. *International Journal of Parallel Programming*, 21(6):389–420, Dec. 1992.

[5] M. Griebl, C. Lengauer, and S. Wetzel. Code generation in the polytope model. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT'98)*, pages 106–111, 1998.

[6] A. Größlinger. *The Challenges of Non-linear Parameters and Variables in Automatic Loop Parallelisation.* doctoral thesis, Department of Informatics and Mathematics, University of Passau, Dec. 2009.

[7] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and D. Wonnacott. The Omega library. Technical report, University of Maryland, Nov. 1996.

[8] W. Kelly and W. Pugh. A framework for unifying reordering transformations. Technical Report UMIACS-TR-92-126.1, University of Maryland Institute for Advanced Computer Studies, 1993.

[9] W. Kelly, W. Pugh, and E. Rosser. Code generation for multiple mappings. In *Frontiers'95 Symposium on the frontiers of massively parallel computation*, McLean, 1995.

[10] M. Le Fur. Scanning parameterized polyhedron using Fourier-Motzkin elimination. *Concurrency - Practice and Experience*, 8(6):445–460, 1996.

[11] H. Le Verge. Recurrences on lattice polyhedra and their applications, April 1995. Unpublished work based on a manuscript written by H. Le Verge just before his untimely death in 1994.

[12] H. Le Verge, V. Van Dongen, and D. Wilde. Loop nest synthesis using the polyhedral library. Technical Report 830, IRISA, 1994.

[13] W. Li and K. Pingali. A singular loop transformation framework based on non-singular matrices. *International Journal of Parallel Programming*, 22(2):183–205, April 1994.

[14] F. Quilleré, S. Rajopadhye, and D. Wilde. Generation of efficient nested loops from polyhedra. *International Journal of Parallel Programming*, 28(5):469–498, Oct. 2000.

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

[16] N. Vasilache, C. Bastoul, and A. Cohen. Polyhedral code generation in the real world. In *Proceedings of the International Conference on Compiler Construction (ETAPS CC'06)*, LNCS 3923, pages 185–201, Vienna, Austria, Mar. 2006.