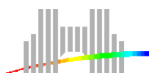
The Polytope Model: Past, Present, Future

Paul Feautrier

ENS de Lyon
Paul.Feautrier@ens-lyon.fr

8 octobre 2009




What is a Model ?

- A model is a mathematical (or computational) object that emulate a real world object (syntax).
- The "natural operations" of the model must emulate the behaviour of the real world object (semantics).
- An example : Newton's laws of motion :

$$\begin{aligned} F &= m\gamma, \\ F &= G\frac{m.m'}{r^2} \end{aligned}$$

A system of ordinary differential equations. Semantics :

- Proof of the existence of solutions
- Computing the solution, either in closed form or numerically
- If properly initialized the solution matches the trajectory of a rocket in the solar system.

What is a Polytope ?

There are two equivalent definitions :

- I – A polyhedron is the locus of the solutions of a system of affine inequalities :

$$P = \{x \mid Ax + b \geq 0\}, x : n, b : m, A : m \times n.$$

  The elements of A and b can be taken as integers.

- A $\mathbb{Z}$-polyhedron is the locus of the **integer** solutions of a system of affine inequalities :

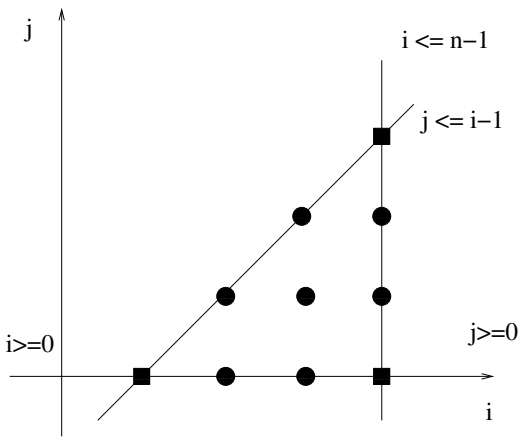
$$P' = \mathbb{Z}^n \cap P.$$

- A polytope is a bounded polyhedron.
- II – A polyhedron is the convex hull of a finite set of points, some of which may be at infinity :

$$P = \{\sum_{i=1}^{N} w_i x_i \mid w_i \geq 0, \sum_{i=1}^{N} w_i = 1\}$$

Principle of the Polytope Model

```
for(i=0; i<n; i++)
  for(j=0; j<i; j++)
    S;
```



$\{0 \leq i \leq n-1, 0 \leq j \leq i-1\}$

- The iterations of a *regular* loop nest can be represented as one (or several) $\mathbb{Z}$-polytope.
- The inequalities are directly extracted from the program text.
- The size of the representation is bounded, and may contain parameters.
- One may even consider infinite loops, which are represented by unbounded polyhedra.
- The model can be used to answer questions like : "is $i = 0, j = 0$ a legal iteration" (simple), or "how many iterations are executed" (difficult).

Regular Programs

The polytope model can only be applied to programs which satisfy the following constraints :

- One can identify a set of *parameters* – integer variables which are not modified in the program
- The data structure are (multidimensional) arrays and scalars of any type
- The control structures are arbitrarily nested DO loops and tests
- Each loop bounds must be affine functions in the surrounding loop counters and parameters
- Each test predicate must be affine in the surrounding loop counters and parameters
- Array subscripts must be affine in the surrounding loop counters and parameters

An example : Gaussian Elimination

```
    for(i=0; i<n; i++){
      pivot = 1.0/a[i][i];

      for(j=0; j<n; j++){
//regular test
        if(j==i)continue;
//regular loop
        for(k=i+1; k<n; k++)
//regular array accesses
          a[j][k] -=
             a[j][i]*a[i][k]*pivot;
      }
    }
```

- Observe that syntax does not matter
- n is a parameter
- i, j, k are loop counters

Fundamental Algorithms

- Conversion from systems of inequalities to vertices and back : Chernikova, 1967, parametric extension Loechner and Wilde, 1997.
- Feasibility tests :
  - Fourier-Motzkin elimination method, 1827, integer extension, Pugh, 1991, naturally parametric.
  - Simplex algorithm, Dantzig 1945, integer extension Gomory, 1950, parametric extension, PF, 1988.
- Counting integer points, E. Ehrhart, 1945, M. Brion, 1988, Ph. Clauss, A. Barvinok, S. Verdolaege, R. Seghir

A. Schrijver : Theory of linear and integer programming, Wiley, NewYork,1986
A. Barvinok : A Course in Complexity, AMS, 2002

Conversion

- Given : a system of affine inequalities $Ax + b \geq 0$;
- Find : a minimum system of vertices $v_1, \ldots, v_N$.
- Method : add the inequalities in the system one by one.
- Since a polyhedron may have an exponential number of vertices, (example : the cube), the complexity is exponential.
- Many efficient implementations (Polylib, PPL, etc) all based on H. LeVerge work.
- The same algorithm is used for the inverse transformation ! ! !
- Since the system of vertices is unique, applying this algorithm twice is a way of normalizing a system of inequalities.
- Can be used as a projection algorithm.
- Can be used as a feasibility test or as a LP algorithm. Not recommended.

The Fourier-Motzkin Algorithm

- Given : a system of affine inequalities $Ax + b \geq 0$;
- Decide if it has solutions or not.
- Method : eliminate each unknown in turn
- Each inequality involving x can be transformed in one of the two forms :

$$\begin{aligned} l_i &\leq x, \\ x &\leq u_j, \end{aligned}$$

  from which follows $l_i \leq u_j$.
- After elimination, decide numerical inequalities.
- Super exponential, redundant. The redundancy can be controlled.
- Integer extention : the Omega test.

B. Pugh :The Omega Test : A Fast and Practical Integer Programming Algorithm for Dependence Analysis, Supercomputing, 1991

The Simplex

- Given : a system of affine inequalities $Ax + b \geq 0$;
- Find its lexicographic minimum or prove it has no solution
- Method : apply a succession of change of variables (Gaussian pivots) until the solution is obvious : Dantzig, 1945.
- Complexity : almost always $O(n^3)$.
- Integer extension : Gomory, 1950. NP-complete.
- Parametric extension : find the lexicographic minimum of x such that :

$$Ax + By + c \geq 0$$

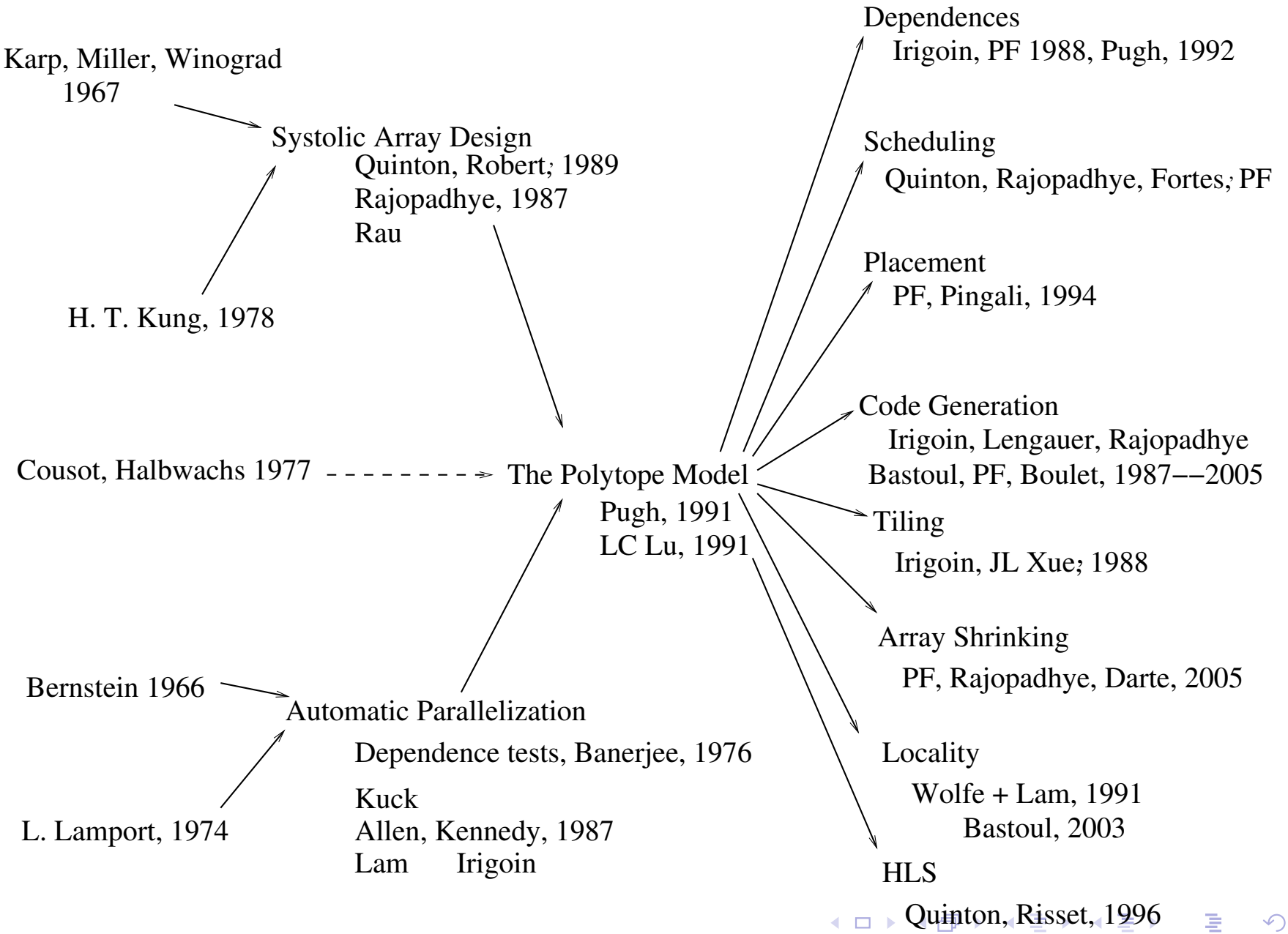
as a function of y. The result is a multilevel conditional or *quast*.

P. Feautrier : Parametric Integer Programming, RAIRO-RO,**22**, pp. 243–268, 1988

Counting Integer Points

- Given : a system of affine inequalities, with one (or more) integer parameter, n,
- Construct a function giving the number of integer solutions.
- Method I : a result of E. Ehrhart shows that the function is a polynomial of known degree. Count the solutions for enough values of n and interpolate. Implemented in the Polylib.
- Method II : a theorem of M. Brion explains how to compute a generating function. The required count is the limit of the generating function when the formal variable tends to 1. Implemented in the Barvinok library.
- Usage : counting the number of iterations of a loop nest, load balancing, locality evaluation...

http ://icps.u-strasbg.fr/polylib
http ://www.kotnet.org/ skimo/barvinok


Karp, Miller, Winograd
1967
Systolic Array Design
Quinton, Robert; 1989
Rajopadhye, 1987
Rau
H. T. Kung, 1978
Cousot, Halbwachs 1977
The Polytope Model
Pugh, 1991
LC Lu, 1991
Bernstein 1966
Automatic Parallelization
Dependence tests, Banerjee, 1976
Kuck
Allen, Kennedy, 1987
Lam Irigoin
L. Lamport, 1974
Dependences
Irigoin, PF 1988, Pugh, 1992
Scheduling
Quinton, Rajopadhye, Fortes; PF
Placement
PF, Pingali, 1994
Code Generation
Irigoin, Lengauer, Rajopadhye
Bastoul, PF, Boulet, 1987––2005
Tiling
Irigoin, JL Xue; 1988
Array Shrinking
PF, Rajopadhye, Darte, 2005
Locality
Wolfe + Lam, 1991
Bastoul, 2003
HLS
Quinton, Risset, 1996

The Early days

- Dependences
- Program Transformations, Why and How
- Parallelization Algorithms

Dependences

Two operation are in dependence if they both access the same memory cell, one of the access being a write.
Dependences are oriented in the direction of sequential execution. The execution order of two dependent operations must be the same in the sequential and parallel program.

- Dependences are easily detected in the absence of address computation
- In the case of array accesses, one has to decide if the *subscript equations* have solution in integers
- This gave rise to the search for fast approximate dependence tests : gcd, Banerjee, I-test, Power test, etc.
- Superseded now by polyhedra emptiness tests : Omega or the Simplex
- No satisfactory solution for pointer accesses

Dependence classification

Dependences are classified according to the read/write order. Each community (hardware, software) has its own terminology :

- Read after Write, flow dependence
- Write after Read, anti dependence
- Write after Write, output dependence
- (Read after Read, input dependence, useful only for locality)

WAR and WAW dependences can be removed by renaming or expansion, RAW dependences are intrinsic to the algorithm.

Program Transformations

- Dependences can be used to parallelize basic blocs, and to detect parallel loops
- However, it was soon found that most programs had very low parallelism
- Causes : over-eager optimizations. Using scalars in place of arrays, fusing loops, strength reduction, ...
- Solution : apply enabling transformations. Loop splitting, scalar expansion, induction variable detection, skewing ...
- Selecting a transformation sequence is a difficult combinatorial problem

Parallelization Algorithms

Can one find the parallel program without doing the intermediate transformations ?

- the Kennedy and Allen Algorithm
  - Combine maximum loop splitting, loop interchange and the search for parallel loops
  - Split according to the *strongly connected components* of the dependence graph
  - Since parallel loops can always be innermosted, the result is (almost) a vector program.
- the Wolf and Lam algorithm : combine loop skewing and parallelization
- using the polytope model, one can generalize to all affine transformations

J. R. Allen and Ken Kennedy : Automatic Translation of Fortran Programs to Vector Form, ACM TOPLAS,(9)4 :491–542, 1987

M. Wolf and Monica S. Lam, A loop transformation theory and an algorithm to maximize parallelism, IEEE TPDS, 2(4),452–471, 1991

State of the Art

- Dependences in the Polytope Model
- Scheduling
- Placement
- Code Generation
- Memory Management : Array Shrinking
- Memory Management : Locality Optimization

Dependences

Definition of the dependence polytope

operation	$\langle S, \vec{i} \rangle : A[f(\vec{i})]$		$\langle T, \vec{j} \rangle : A[g(\vec{j})]$
domain	$\vec{i} \in D_S$		$\vec{j} \in D_T$
subscript		$f(\vec{i}) = g(\vec{j})$	
order	$i_1 < j_1$	$\langle S, \vec{i} \rangle \prec \langle T, \vec{j} \rangle$ $i_1 = j_1, i_2 < j_2$	$i_1 = j_1, i_2 = j_2$
depth	0	1	2

One can test for emptyness, approximate, or use the dependence polyhedron as is.

Approximating dependences

A dependence can be conservatively approximated (the dependence polyhedron is enlarged).

- Ignore difficult constraints (e.g. integrity constraints)
- Simplify the polyhedron :
  - dependence depth,
  - dependence distance,
  - dependence cone,
  - direction vectors,
  - octagons

Scheduling

Assign a logical date to each *operation* in the program.

- Since the number of operations is usually unknown, the date must be given by a formula, not by a table.
- An operation is denoted by the name of its statement and by its iteration vector. The schedule is assumed to be affine in the iteration vector.
- **Causality Constraints** If operation $\langle S, \vec{i} \rangle$ is dependent on $\langle T, \vec{j} \rangle$, then their schedules must satisfy :

$$\theta(S, \vec{i}) < \theta(T, \vec{j}).$$

Beside :

$$\theta(S, \vec{i}) \geq 0.$$

Solution

- If one replace $\vec{i}$ and $\vec{j}$ by numerical values, one gets a linear constraint on the coefficients of θ
- However, the number of such constraints may be very large or even infinite
- It is enough to write the constraints at the vertices of the dependence polyhedron, or to use Farkas lemma
- The solution is obtained by application of any LP solver.
- The system of constraints may not be feasible if the original program has more than linear parallel complexity. A multi-dimensional schedule is needed.

The Shape of the Parallel Program

Let :

$$L = \max \theta(S, \vec{i}),$$

the latency of the parallel program, and :

$$\mathcal{F}(t) = \{S, \vec{i} \mid \theta(S, \vec{i}) = t\},$$

the *front* at time t.

```
for 0 <= t <= L
  do in parallel F(t)
```

- Since all operations in a front are independent, this is nearly a vector program (*node splitting*)
- The locality is poor
- Allen and Kennedy, Wolf and Lam are scheduling algorithms where the exact dependences are replaced by approximations

Placement, how and why

On distributed memory architectures, one needs to know :

- in which memory each piece of data is stored : $\pi(x)$
- on which processor each operation is executed : $\pi(u)$

To avoid communications, if operation $\langle S, \vec{i} \rangle$ uses datum $A[f(\vec{i})]$, one must impose :

$$\pi(S, \vec{i}) = \pi(A[f(\vec{i})]).$$

Another presentation : if two operation u and v access the same x, then they are in dependence, cannot be executed in parallel, and may as well be assigned to the same processor :

$$\pi(u) = \pi(x) = \pi(v)$$

Since all operations that (re)use the same datum are on the same processor, the resulting program has good locality.

Placement algorithm

In most cases, the placement equation cannot be satisfied everywhere : the program has no *communication free* parallelism. The solution :

- Each instance of the placement equation represents a (potential) communication
- If the equation is satisfied, the communication disappear
- The communication volume can be estimated (e.g. by counting points)
- Ignore the communication with the smallest volume and try to solve
- Continue ignoring communications until a solution is found

Code Generation

In contrast to parallelization algorithm (e.g. Kennedy/Allen), the outcome of scheduling or allocation is not a program but a *change of variables* to be applied to the iteration space.

- Basic method : rewrite the iteration space constraints as functions of the new variables
- Construct a system of loop nests which scans the new iteration space in lexicographic order
- Overhead : integer divisions, modulos, guards ...

CLooG

The same basic framework, but :

- the transformation is seen as a *renaming*
- code duplication is used to avoid guards
- loop steps > 1 are used to avoid modulos
- any degree of freedom in the schedule is used to simplify the target code

A long history : Z. Chamsky, F. Quilleré and S. Rajopadhye, C. Bastoul

http ://www.CLooG.org

Another Code Generation Method

Given :

- a set of operations $E = \cup_S \langle S, D_S \rangle$,
- an execution order $\ll$ (parallel or sequential) on E.

compute :

$$\mathrm{first}() = \min_{\ll} E,$$

$$\mathrm{next}(S, \vec{i}) = \min_{\ll} \cup_T \{ T, \vec{j} \mid \vec{j} \in D_T, \langle S, \vec{i} \rangle \ll \langle T, \vec{j} \rangle \}$$

using parametric integer programing.
The result can be interpreted as a finite state machine, where the next function enumerates the transitions from a given state.
Especially suitable for high level synthesis.

Pierre Boulet and Paul Feautrier, Scanning Polyhedra without DO loops, PACT'98, 1998

Array Contraction

Motivation

- For embedded systems, minimize the size of memory
- For multicore, minimize memory bandwidth
- But beware : more parallelism needs more memory

Method

- For a given schedule, compute the *life span* of variable x ;

$$L = [\min\{\theta(u) \mid u \text{ writes } x\}, \max\{\theta(v) \mid v \text{ reads } x\}]$$

- If the life spans of two variables do not overlap, reuse the same memory cell.

Lefebvre, Quilleré and Rajopadhye, Darte et. al.

Alain Darte, Robert Schreiber and Gilles Villard, Lattice-Based Memory Allocation, IEEE Transactions on Computers, 54(10),1242–1257, 2005

Cache Optimization

- The length of the life span :

$$d = \max\{\theta(v) \mid v \text{ reads } x\} - \min\{\theta(u) \mid u \text{ writes } x\}$$

  can be interpreted as the *reuse distance* of x.
- Intuitively, the probability of finding x in the cache increases when d decreases
- transform loops so that all accesses to x are at the deepest level of a loop nest (Wolf and Lam, 1991)
- or, use an upper bound for d as an objective function when solving for the schedule by linear programming

U. Bondhugula et.al., Automatic Transformations for Communication-Minimized Parallelization and Locality Optimization in the Polyhedral Model, Compiler Construction, 2008, 132–146

Self-assessment

The polytope model is a nice framework, but :

- it has scalability problems
- some questions are still unsolved or needs better solutions
- it can only be applied to severely restricted programs

Scalability

- The size of the scheduling problem is proportional to the number of dependences, which increases as the square of the program size
- Other factors (array dimension and loop nesting depth) are "small integers"
- Linear programming is almost always cubic (like Gaussian elimination)
- Hence, the scheduling problem scales as the *six* power of the program size
- Scheduling is not scalable ! ! ! !

Scalability : Outline of a Solution

Some programs (especially *streaming* programs) can be split in processes which communicate throught channels (write once / read many arrays) :

- For each process, compute a schedule, parametric in the "clocks" of its input and output channels
- Solve for the channel clocks
- Adjust the process schedules

Problem Most programs are not written that way :

- Can the method be extended to interprocedural scheduling ? (I guess not)
- Can ordinary programs be converted into process systems ? (Maybe)

Paul Feautrier, Scalable and Structured Scheduling, Int. J. of Parallel Programming, 34, 459–487, 2006

Memory, Resources, and Locality

Scheduling usually results in a lot of fine grain parallelism : e.g. Gaussian elimination has $O(n^2)$ parallelism. The run-time system is responsible for adjusting it to the degree of parallelism of the target computer (e.g. a dual-core processor)

- large overhead
- excessive use of memory
- bad locality
- what about architectures without a run-time system (hardware or embedded processors) ?

Constructing threads

Fronts are OK for SIMD processors (Multimedia extensions) and vector processors (GPU), but SMPs (multicores) need *threads*.

- Split the code into phases (probably along Strongly Connected Components)
- For each phase, select a placement function (which gives the name of the thread for each operation)
- Schedule under the additional condition that each thread execute at most one operation at a time (how ?)

Extending the Polytope Model

The polytope model deals only with highly constrained programs :

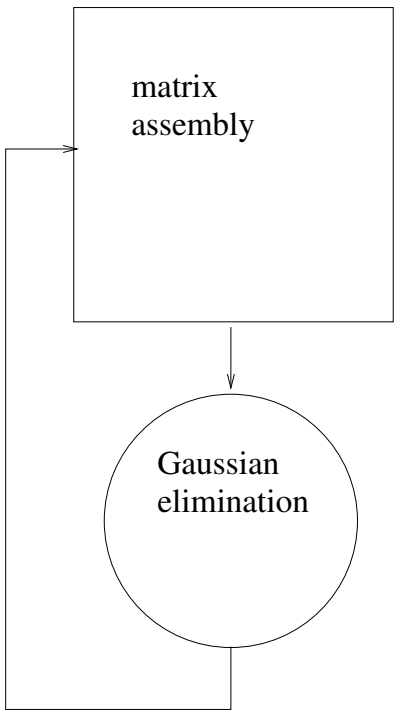
- DO loops with affine bounds
- Arrays with affine subscripts

One can enlarge the set of tractable programs by preprocessing and approximations :

- induction variable detection
- while loop analysis
- enlarging polyhedra by ignoring non-affine constraints

Problems with code generation

SCoPs and GRAPHITE



While most programs are not regular *in toto* one can usually find large pieces which are : Static Control Parts or SCoP.

- extract the SCoPs
- parallelize each SCoP using the polytope model
- plug the result back into the original program
- let the host compiler generate the target code
- partially implemented in the GRAPHITE branch of gcc
- beware of Amdahl's law !

Run-time Parallelization

The basic approach is speculative :

- Execute each loop as if it were parallel
- Test for dependences on the fly
- If a dependence if found, rollback and execute sequentially
- Overhead ?
- Can be improved by excluding obviously sequential loops at compile time
- May be generalized to *iterators* insted of loops

Another approach : the *inspector-executor* method : the inspector dynamically construct a parallel schedule that is used several time by the executor.

Kulkarni, Pingali et. al. : Optimistic Parallelism Requires Abstractions, CACM 52(9), 89–97, 2009

Domain Specific Parallelization

In many cases, the necessary information is not present in the program text :

- Dependences may be excluded or negligible for physical reasons (the two wingtips of an aircraft)
- The execution order of some operations may be indifferent (the exploration order in a branch-and-bound tree, the order of additions in a summation – up to rounding errors)
- It may be indifferent to add or remove a few operations (to iterate slightly farther than convergence)

These situations can only be handled by domain specific languages and parallelizers. Some exemples already exists : FFTW and Spiral for the Fast Fourier Transform, code for the solution of chemical and biological ODE.