A task-based approach to parallel parametric linear programming solving, and application to polyhedral computations

Camille Coti, David Monniaux, Hang Yu

A task-based approach to parallel parametric linear programming solving, and application to polyhedral computations

Camille Coti[1], David Monniaux[2], and Hang Yu[2]

[1]LIPN, CNRS UMR 7030, Université Sorbonne Paris Nord, 99, avenue Jean-Baptiste Clément, F-93430 Villetaneuse, France
[2]VERIMAG, Univ. Grenoble Alpes, CNRS, Grenoble INP*, F-38000 Grenoble, France

September 28, 2020

Abstract

Parametric linear programming is a central operation for polyhedral computations, as well as in certain control applications. Here we propose a task-based scheme for parallelizing it, with quasi-linear speedup over large problems. This type of parallel applications is challenging, because several tasks might be computing the same region. In this paper, we are presenting the algorithm itself with a parallel redundancy elimination algorithm, and conducting a thorough performance analysis.

1 Introduction

A *convex polyhedron* in dimension n is the solution set over $\mathbb{Q}^n$ (or, equivalently, [1]$\mathbb{R}^n$) of a system of inequalities (with integer or rational coefficients). Since all the polyhedra we consider here are convex, we shall talk of polyhedra for short. There also exist integer polyhedra, defined over $\mathbb{Z}^n$, but they are very different in many respects; we shall not consider them here.[2]

Computations over polyhedra arise in low dimension ($n = 2$ and $n = 3$) for modeling physical objects, but here we are interested in higher dimensions.

*Institute of Engineering Univ. Grenoble Alpes

[1]Whether emptiness tests, inclusion tests, projection etc. are specified with real or rational variables, the results are the same. It is impossible to distinguish the reals and the rationals using first-order formulas in linear arithmetic.

[2]Among differences, it is possible to check in polynomial time if a given list of inequalities defines an empty polyhedron over $\mathbb{Q}$ but the same problem is NP-complete over $\mathbb{Z}$.

Polyhedra in higher dimension are typically used to enclose the set of reachable states of systems whose state can be expressed, at least partially, as a vector of reals or rationals. For instance, one can study the flow of ordinary differential equations, or more generally the trajectories of hybrid systems, by enclosing the trajectories into a succession of convex polyhedra. If the internal state of a control system is expressed by a vector of n numeric variables, one may prove that this system never encounters a bad condition by exhibiting a polyhedron P such that the initial state belongs to P, no bad condition belongs to P, and all possible time steps of the control system leave P stable (that is, it is not possible to execute a step starting in P and ending outside of P).

One generally proves the correctness of programs by exhibiting *inductive invariants* — an inductive invariant for a loop is a set containing the precondition of the loop, stable by moving to the next loop iteration, and implying the desired postcondition. Since providing inductive invariants by hand is tedious, it is desirable to automate that process. One approach for doing so is *abstract interpretation*, where an ascending sequence of sets of states is computed until reaching a fixed point. One may for instance search such sets as products of intervals, an approach known as *interval analysis*, but the lack of relationships between the dimensions tends to severely limit the kind of properties that can be proved (e.g., one cannot have an invariant $i < n$ where n is a parameter: this is neither an interval on i nor on n nor a combination thereof). Cousot and Halbwachs proposed searching for polyhedral inductive invariants [10, 16]. The operations needed there are projection, more generally image by an affine transformation, convex hull, and inclusion (or equality) test, together with an extrapolation operator known as *widening*.

Such usages of convex polyhedra suffered from the *curse of dimensionality*: complexity grows quickly with the number of dimensions—the number of numeric variables in the program or hybrid system under analysis. This problem is exacerbated in approaches that double the number of variables to represent pairs of states, or add extra variables for representing nonlinear terms (e.g., a variable v_{xy} is added to represent xy [27]). This led to either restricting polyhedra to subclasses with lower computational costs (e.g., restricting inequalities to $\pm x_i \pm x_j \leq C$ as in the *octagons* [28]), or severely restricting the number of dimensions of the system under consideration (e.g., by focusing on a subset of interest of the program variables, disregarding relationships with variables outside of that subset).

Why this curse of dimensionality? There exist multiple libraries for computing over polyhedra (NewPolka,[3] Parma Polyhedra Library,[4] PolyLib,[5] CDD...[6]). They all use the *double description* [29] of convex polyhedra: both as *constraints* (inequalities or equalities) and *generators* (vertices, and, for unbounded polyhedra, lines and rays). Some operations are indeed simpler on one description than the other, and Chernikova's algorithm [24] converts between the two. Having

[3] Jeannet's NewPolka is now part of Apron `http://apron.cri.ensmp.fr/library/` [18]
[4] `https://www.bugseng.com/ppl` [4]
[5] `https://icps.u-strasbg.fr/PolyLib/`
[6] `https://www.inf.ethz.ch/personal/fukudak/cdd_home/`

both descriptions is also handy for removing redundant constraints and generators, which are produced by many of the algorithms.

The double description has many advantages, but one major weakness. A very common case of invariant is when one knows the interval of variation of each variable: $[l_1, h_1] \times \cdots \times [l_n, h_n]$. Such a polyhedron has $2n$ constraints ($l_i \leq x_i$ and $x_i \leq h_i$ for $1 \leq i \leq n$) and 2^n vertices (each x_i is independently chosen in $\{l_i, h_i\}$). The double description then blows up. One workaround for the above case is to detect that the polyhedron is exactly a Cartesian product of simpler polyhedra, and to compute as locally as possible in terms of that product [17], but this fails if the polyhedron is *almost* a Cartesian product; We thus chose to completely do away with the generator representation and use constraints only.[7]

The question was how to compute over convex polyhedra described by constraints only. It is possible to reduce most operations (image, convex hull etc.) to projection. An algorithm for projecting polyhedra described by inequalities only, Fourier-Motzkin elimination [22], has long been known. Unfortunately it tends to generate a very large number of redundant constraints, which must be eliminated using an expensive procedure.

We instead turned to *parametric linear programming*. Image, projection, convex hull can all be formulated as solutions to linear programs where parameters occur linearly within the objective [25, 5]: given a system of equalities and inequalities, defining a convex polyhedron P in higher dimension, and a parametric bilinear objective function $f(x, \lambda)$ where x is the point and λ a vector of parameters, give for each λ a vertex x^* of P such that $f(x^*, \lambda)$ is minimal. A solution to such a program is a quasi-partition of the space of parameters λ into convex polyhedra, with one optimum associated to each polyhedron. The issue is how to compute this solution efficiently. In this article, we describe how we parallelized our algorithm for doing so.

In addition to computations on convex polyhedra, parametric linear programming is also used for control applications [20]: instead of using a solver, whose computation times are high and hard to predict, inside the control loop, the solution of the linear program is tabulated as a piecewise linear function over the values of the parameters. Parametric linear programming is also used for affine linear approximations of nonlinear expressions [27].

Parallelizing this type of applications seems straightforward at first sight, since each polyhedron can be computed independently from the other ones. However, it is actually challenging, because several computation units (threads or processes) might be computing the same region at the same time. In this paper, we present a parallel redundancy elimination algorithm that improves the performance by eliminating redundant computations between concurrent threads, and conduct a thorough experimental study of the task-based parallel scheme presented shortly in [8].

[7] By polyhedral duality, which exchanges constraints and generators, the worst-case for generators translates into a worst-case for constraints. The crucial point is that the worst-case for generators occurs very naturally in the analysis of programs or hybrid systems.

2 Related works

Most libraries for computing over convex polyhedra are based on the double description approach: a polyhedron is described both as the convex hull of its vertices (and, in the case of unbounded polyhedra, rays and/or lines), and as the solution set of a system of equalities and inequalities. They convert from one description to the other using Chernikova's algorithm [24], which computes a set of generators (vertices, rays, lines) from a set of (in)equalities (and, dually, the converse) by considering each (in)equality in a sequence and computing the intersection of the polyhedron defined by the previous (in)equalities in the sequence and the current one. To our best knowledge, there is no parallel version of Chernikova's algorithm, and we fail to see how to parallelize its main loop. It may be possible to parallelize the inner loops that compute the generators of the intersection of a polyhedron P and an (in)equality C given the generators of P. An alternative to Chernikova is the reverse search vertex enumeration algorithm [2].

We also opted out of the double description because it is difficult to independently verify that a polyhedron described by generators includes the polyhedron that should have been computed.[8]

Such verifiability is desirable for certain applications: when one computes a polyhedron meant to include all reachable states of a system, to prove that no undesirable state can be reached, then it would be catastrophic that this polyhedron excludes some reachable state due to a bug in a library. Our Verified Polyhedron Library (VPL)[9] [1, 25, 26, 27, 11, 12, 13] provides, in addition to core computations, an optional layer, formally proved correct within the Coq proof assistant, that performs this verification.

VPL implements a constraint-only description (equalities and inequalities) for polyhedra, using two generations of algorithms. The first generation maps all operations, including convex hull, to projection, performed by Fourier-Motzkin elimination [22, 14]. The second generation maps all operations to parametric linear programming, performed as in Algorithm 1 except that no floating-point solver is used, just an exact-precision implementation of the simplex algorithm. Furthermore, VPL is implemented in OCaml, which does not currently allow running computations in multiple threads, and its data structures were designed for compatibility with Coq. For all these reasons, VPL has lower performance than the C++ implementation described in this paper, even with one thread.

[8] It is co-NP-hard to check that, given the description of a polyhedron A by constraints and a polyhedron B described by generators, A is included in B [15]. Furthermore, the *vertex enumeration* problem, that is, checking whether, given a polyhedron P described by a list of constraints and a set of vertices V of that polyhedron, P has another vertex not in V, is NP-hard for unbounded polyhedra [23]. As of 2008, it was unknown if the same problem for bounded polyhedra, or, equivalently, that of generator enumeration (V may contain rays and lines) for unbounded polyhedra, was also NP-complete (it is known to be in NP). No progress seems to have been made on this front since then.

Enumeration can be done in polynomial time for *simple* polyhedra, those for which a vertex corresponds to exactly one basis [2] (no degeneracy); note that degeneracy is also a major source of complication in our algorithms.

[9] `https://github.com/VERIMAG-Polyhedra/VPL`

Parametric linear programming with the parameters in the objective function is a generalization of vertex enumeration, for which there exist parallel implementations based on *reverse search* [3] (vertex enumeration is the case where there are as many independent parameters as there are variables, so that the optimization direction can point to any direction). A difference of our approach with reverse search is that we store the nodes already traversed in a central location, which they do not have to do. Jones and Maciejowski [21] applied reverse search to parametric linear programming; they however warn that while they have better asymptotic complexity than other approaches, the constant hidden in the big-O notation is huge and they warn that their approach is likely to be interesting only on larger examples. In contrast, we base ourselves on an approach already used in a sequential library that is competitive with double description approaches even on problems in moderate dimension [5].

3 Sequential algorithms

We shall leave out here how polyhedral computations such as projection and convex hull can be reduced to parametric linear programming — this is covered in the literature [19, 1] — and focus on solving the parametric linear programs.

3.1 Non-parametric linear programming

A linear program with n unknowns is defined by a system of equations $AX = B$, where A is an $m \times n$ matrix; a solution is a vector X such that $X \geq 0$ on all coordinates and $AX = B$.[10] The program is said to be *feasible* if it has at least one solution, *infeasible* otherwise. In a non-parametric linear program one considers an objective C: one wants the solution that maximizes $C^T X$. The program is deemed *unbounded* if it is feasible yet it has no such optimal solution.

Example 1. Consider the polygon P defined by $x_1 \geq 0$, $x_2 \geq 0$, $3x_1 - x_2 \leq 6$, $-x_1 + 3x_2 \leq 6$. Define $x_3 = 6 - 3x_1 + x_2$ and $x_4 = 6 + x_1 - 3x_2$. Let $X = (x_1, x_2, x_3, x_4)$, then P is the projection onto the first two coordinates of the solution set of $AX = B \wedge X \geq 0$ where $A = \begin{bmatrix} 1 & -3 & 0 & -1 \\ -3 & 1 & -1 & 0 \end{bmatrix}$ and $B = \begin{bmatrix} 6 \\ 6 \end{bmatrix}$.

A linear programming solver takes as input (A, B, C) and outputs "infeasible", "unbounded" or an optimal solution X^*. Some linear programming solvers take (A, B, C) and output X^* as exact rational numbers and ensure that the answer is correct. Most, however, operate fully on floating-point numbers and their final answer may be incorrect: they may answer "infeasible" whereas the problem is feasible, or propose "optimal solutions" that are not solutions, or are not optimal.

[10] There exist more general descriptions of linear programming with upper and/or lower bound constraints on each coordinate of X; our approach generalizes to them.

In addition to an exact X^* or floating-point $\tilde{X}^*$ output, the solvers also provide the discrete information of *optimal basis*: a solution is obtained by the simplex algorithm setting $n-m$ coordinates of X^* to 0 (known as *nonbasic variables*) and solving for the other coordinates (known as *basic variables*) using $AX^* = B$, and the solver provides the partition into basic and nonbasic variables it used. If a floating-point solver is used, it is possible to reconstruct an exact rational point X^* using that information and a library for solving linear systems in rational arithmetic. One then checks whether it is truly a solution by checking $X^* \geq 0$ on the reconstructed coordinates.

The optimal basis contains even more information: it contains a proof of optimality of the solution! The system computes the objective function $C^T X$ as $\sum_{i \in N} \alpha_i X_i$ where N is the set of indices of the nonbasic variables, and concludes that the solution obtained by setting these nonbasic variables to 0 is maximal because all the α_i are nonpositive.

If X^* is not a solution of the problem (the condition $X^* \geq 0$ fails) or is not optimal, then one can fall back to an exact implementation of the simplex algorithm, possibly starting it from the last basis considered by the floating-point implementation.

Example 1 (continued). Assume the objective is $C = \begin{bmatrix}1 & 1 & 0 & 0\end{bmatrix}$, that is, $C^T X = x_1 + x_2$. From $AX = B$ we deduce $x_1 = 3 - \frac{3}{8}x_3 - \frac{1}{8}x_4$ and $x_2 = 3 - \frac{1}{8}x_3 - \frac{3}{8}x_4$. Thus $x_1 + x_2 = 6 - \frac{1}{2}x_3 - \frac{1}{2}x_4$.

Assume x_3 and x_4 are nonbasic variables and thus set to 0, then $X^* = (x_1, x_2, x_3, x_4) = (3, 3, 0, 0)$. It is impossible to improve upon this solution: as $X \geq 0$, changing the values of x_3 and x_4 can only decrease the objective $o = 6 - \frac{1}{2}x_3 - \frac{1}{2}x_4$. This expression of o in terms of the nonbasic variables can be obtained by linear algebra once the partition into basic and nonbasic variables is known.

If the linear programming solver uses floating-point arithmetic and is not to be trusted, it is still possible to reconstruct, by pure linear arithmetic, the expression of the objective function as a function of the nonbasic variables and check the signs of the coefficients.

While the optimal value $C^T X^*$, if it exists, is unique for a given (A, B, C), there may exist several X^* for it, a situation known as *dual degeneracy*.

Example 1 (continued). Assume the objective is $C = \begin{bmatrix}-1 & 0 & 0 & 0\end{bmatrix}$, that is, $C^T X = -x_1$. X^* can be chosen to be $(0, 0, 6, 6)$, $(0, 2, 8, 0)$ or any point in between.

The same X^* may be described by different bases, a situation known as *primal degeneracy*, happening when more than $n - m$ coordinates of X^* are zero, and thus some basic variables could be used as nonbasic and the converse.

Example 2. Consider the regular polygon B with $k \geq 3$ vertices $\big(1+\cos(2i\pi/k), 1+\sin(2i\pi/k)\big) \mid 1 \leq i < k$. The convex hull of B and $T = (1, 1, 1)$ is a pyramid with $k+1$ faces (e.g., $k = 4$ is a square pyramid), defined using $3+k$ unknowns and k equations (e.g., for $k = 4$, $x_4 = 1 - x_1 - x_2 - x_3$, $x_5 = 1 + x_1 - x_2 - x_3$, $x_6 = 1 - x_1 + x_2 - x_3$, $x_5 = 1 + x_1 + x_2 - x_3$), plus $x_1, \ldots, x_{k+3} \geq 0$. The

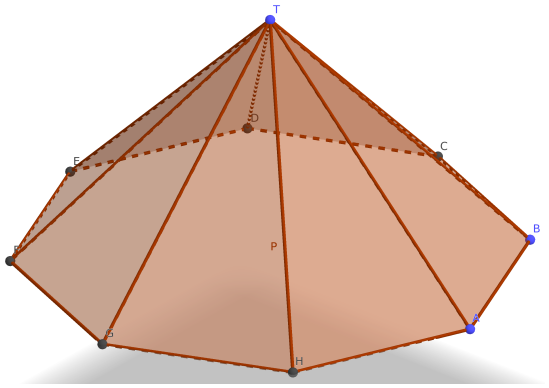


Figure 1: A pyramid based on an octagon, the apex T described by $\binom{8}{3}$ bases.

apex T of the pyramid corresponds to $(1, 1, 1, 0, \ldots, 0)$. It is obtained by picking any 3 variables out of $x_4, \ldots, x_{3+k}$ as nonbasic: there are $\binom{k}{3}$ bases defining it (Fig. 1).

3.2 Parametric linear programming

For a *parametric* linear program, we replace the constant vector C by $C_0 + \sum_{i=1}^{k} \mu_i C_i$ where the μ_i are parameters.[11] When the μ_i change, the optimum X^* changes. Assume temporarily that there is no degeneracy. Then, for given values of the μ_i, the problem is either unbounded, or there is one single optimal solution X^*. It can be shown that the region of the $(\mu_1, \ldots, \mu_k)$ associated to a given optimum X^* is a convex polyhedron (for C_0, a convex polyhedral cone), and that these regions form a quasi partition of the space of parameters (two regions may overlap at their boundary, but not in their interior) [19, 20, 1]. The output of the parametric linear programming solver is this quasi-partition, and the associated optima—in our applications, the problem is always bounded in the optimization directions, so we do not deal with the unbounded case.

Let us see in more detail how to compute these regions. We wish to attach to each basis (at least, each basis that is optimal for at least one vector of parameters) the region of parameters for which it is optimal.

Example 1 (continued). Instead of $C = \begin{bmatrix} 1 & 1 & 0 & 0 \end{bmatrix}$ we consider the parametric $C = \begin{bmatrix} \mu_1 & \mu_2 & 0 & 0 \end{bmatrix}$. Let us now express $o = C^T X$ as a function of the nonbasic variables x_3 and x_4:

$$o = (3\mu_1 + 3\mu_2) + \left(-\frac{3}{8}\mu_1 - \frac{1}{8}\mu_2\right) x_3 + \left(-\frac{1}{8}\mu_1 - \frac{3}{8}\mu_2\right) x_4 \qquad (1)$$

The coefficients of x_3 and x_4 are nonpositive if and only if $3\mu_1 + \mu_2 \geq 0$ and

[11] There exists another, dual, kind of parametric linear programming where the parameters are in the right-hand side B. We do not consider it here.

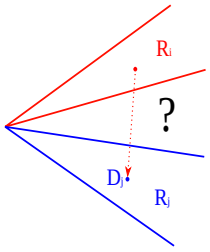


Figure 2: R_i and R_j are not adjacent. The intermediate region is missed.

$\mu_1 + 3\mu_2 \geq 0$, which define the cone of optimality associated to that basis and to the optimum $X^* = (3, 3, 0, 0)$.

Note that the description of the cone (or polyhedron) of optimality by the constraints obtained from the sign conditions in the objective function may contain redundant constraints, that is, constraints that can be removed without changing the cone. It is desirable to remove these. Some procedures for removing redundant constraints from the description of a region R_1 also provide a set of vectors outside of R_1: for each constraint in the description they provide a vector violating it [26], a feature that will be useful.

Assume now we have solved the optimization problem for a value $C(D)$ of the optimization direction, for a vector of parameters D_1, and obtained a region R_1 in the parameters (of course, $D_1 \in R_1$). We now pick $D_2 \notin R_1$ — if the redundancy elimination procedure provided us with a set of vectors outside of R_1, we can store them in a "working set" to be processed and choose D_2 in it. We compute the region R_2 associated to D_2. Assume that R_2 and R_1 are adjacent, meaning that they have a common boundary. We get vectors outside of R_2 and add them to the working set. We pick D_3 in the working set, check that it is not covered by R_1 or R_2, and, if it is not, compute a region R_3, etc. This amounts to a traversal of the adjacency graph of the optimality regions. The algorithm terminates when the working set becomes empty, meaning the $R_1, \ldots$ produced form the desired quasi-partition.

This simplistic algorithm might fail because it assumes that it is discovering the adjacency relation of the graph. The problem is that, if we move from a region R_i to a vector $D_j \notin R_i$, it is not certain that the region R_j generated from D_j is adjacent to it, so we could miss some intermediate region (Fig. 2). In order to cope with this issue, we modify our traversal algorithm as follows. The working set contains pairs (R, D') where R is a region and $D' \notin R$ a vector (there is also a special value none for R). The region R' corresponding to D' is computed. If R and R' are not adjacent, then a vector D'' in between R and R' is computed, and the pair (R, D'') is added to the working set. This ensures that at the end, we obtain a quasi-partition. An additional benefit is that we can obtain a spanning tree of the region graph, with edges from R to R', tracking which region led to which other one.

The last difficulty is degeneracy. So far we have assumed that each optimization direction corresponded to exactly one optimal vector X^*, and that this optimal vector is described by exactly one basis. This is not the case in general; in this case, the interiors of the optimality regions may overlap. This situation results in a performance hit; also the final result is no longer a quasi-partition, but instead just a covering of the parameter space (for each possible vector D of parameters, there is at least one region that covers it). Of course, the value of the objective function $C(D)^T X^*$ must be the same for all optimal vectors X^* associated with the regions covering D. A covering suffices however for the correctness of our projection, convex hull etc. algorithms.

We are currently investigating approaches for getting rid of degeneracy — enforcing one optimal vector X^* and only one optimal basis per vector D, except at the boundaries. The methods for doing so rely on lexicographic orderings or perturbations on the objective and/or constant term, or pivoting rules [20]. We have recently proposed a working solution to degeneracy [31] but there is still room for improvement.

4 Parallel algorithms

4.1 Parallel redundancy elimination

A polyhedron may be specified by a redundant system of inequalities, meaning that some inequalities can be discarded without changing the polyhedron. The first step is to eliminate syntactic redundancies — constraints simplified into true or false, or subsumed by another (e.g., $2x + 2y \leq 2$ subsumes $x + y \leq 2$); a constraint of the form $0 \leq -1$ after simplification makes the polyhedron empty; a constraint of the form $0 \leq 1$ after simplification is to be discarded; if two constraints are of the form $C^T X \leq B_1$ and $C^T X \leq B_2$ where $B_1 \leq B_2$, then the latter is to be discarded (note that this involves putting the vector of rationals C in canonical form: flushing denominators and removing common factors, so that e.g., $2x + 2y \leq 3$ can be discarded as subsumed by $x + y \leq 1$).

The general case of redundancy elimination is harder. Checking that an inequality C is redundant with respect to other inequalities $C_1 \wedge \cdots \wedge C_n$ boils down to finding a vector D such that D satisfies $C_1 \wedge \cdots \wedge C_n$ but not C: such a vector exists if and only if the inequality is irredundant. This is a pure satisfiability problem in linear programming (it uses a strict inequality $\neg C$, but this can be dealt with). Note that if an inequality is irredundant, a vector D not satisfying that inequality is provided: this is handy since *eliminate_redundancy*(S) is to, in addition to removing constraints, provide a set of vectors each violating one constraint but not the others.

A sequential algorithm for removing redundant constraints thus considers each constraint in sequence, and tests its irredundancy with respect to all the other remaining constraints that have not been shown to be redundant yet. This can also be done in parallel, as in Algorithm 2. The only requirement is that the table marking which constraints have already been found to be redundant

Algorithm 1 Sequential parametric linear programming solver.

float_lp(A, B, C) is an external procedure returning the optimal basis for maximizing $C^T X$, $AX = B$, $X \geq 0$; it may provide incorrect results.
exact_lp returns the exact optimum and optimal basis.
midpoint(R, R', D'), where $D' \in R'$, computes a vector in between regions R and R'.
exact_point computes the exact rational X^* point corresponding to the basis.
exact_objective computes the objective function as a bilinear function of the parameters and the nonbasic variables (the output is a matrix).
sign_conditions translates it into sign conditions on the parameters, defining a cone.
eliminate_redundancy(S) returns (R, D_{next}), where R is an irredundant set of inequalities defining the same cone as S and D_{next} are vectors outside of that cone, each violating one different inequality in R but not the others.

procedure PLP(A, B, C)
  pick any nonzero vector of parameters D_0
  $W \leftarrow \{(\mathsf{none}, D_0)\}$
  regions $\leftarrow \emptyset$
  while $W \neq \emptyset$ **do**
    Pick (R_{from}, D) in W and remove it from W
    $R_{\text{cov}} \leftarrow$ *is_covered*$(D,$ *regions*$)$
    if $R_{\text{cov}} ==$ none **then**
      basis $\leftarrow$ *float_lp*$(A, B, C(D))$
      $X^* \leftarrow$ *exact_point*(*basis*)
      $o \leftarrow$ *exact_objective*(*basis*)
      if $\neg(X^* \geq 0 \wedge o \leq 0)$ **then**
        $(basis, X^*) \leftarrow$ *exact_lp*$(A, B, C(D))$
      end if
      $S \leftarrow$ *sign_conditions*(*basis*)
      $R \leftarrow$ *eliminate_redundancy*(S)
      for each constraint i in R **do**
        $D_{\text{next}} \leftarrow$ *compute_next*(R, i)
        $W \leftarrow W \cup \{(R, D_{\text{next}})\}$
      end for
      regions $\leftarrow$ *regions* $\cup \{(R, X^*)\}$
      $R_{\text{cov}} \leftarrow R$
    end if
    if $\neg$*are_adjacent*$(R_{\text{from}}, R_{\text{cov}})$ **then**
      $D' \leftarrow$ *midpoint*$(R_{\text{from}}, R_{\text{cov}}, D)$
      $W \leftarrow W \cup \{(R_{\text{from}}, D')\}$
    end if
  end while
  return(*regions*)
end procedure

procedure *is_covered*$(D,$ *regions*$))$
  for $(R, X^*) \in$ *regions* **do**
    if D covered by R **then**
      return(R)
    end if
  end for
  return(none)
end procedure

Algorithm 2 Parallel redundancy elimination.

```
is_redundant ← make_array(n, false)
for i ∈ 0 ... n − 1 do
    F ← {¬C_i}
    for j ∈ 0 ... n − 1 do
        atomic r ← is_redundant[j]
        if j ≠ i ∧ r then
            F ← F ∪ {C_j}
        end if
    end for
    if ¬check_sat(F) then
        atomic is_redundant[i] ← true
    end if
end for
```

should support atomic accesses. We applied the same parallelization scheme to our more refined "ray-tracing" redundancy elimination algorithm [26].

We can use parallel redundancy elimination within parallel parametric linear programming. It helps in some cases where the coarse-grained parallelism of the parametric linear programming solver is limited due to the lack of tasks that can be executed in parallel, e.g., projections of polyhedra in smaller dimensions and few faces in the result; otherwise it does not help and hampers tuning.

4.2 Parallel parametric linear programming

We implemented two variants of task-based coarse-grained parallelism: one using Intel's Thread Building Blocks (TBB),[12] the other using OpenMP tasks [30].

Algorithm 1 boils down to executing tasks taken from a working set, which can themselves spawn new tasks. In addition to the working set, it uses a shared data structure: the set *regions* of regions already seen, used: i) for checking whether a vector D belongs to a region already covered (*is_covered*); ii) for checking adjacency of regions; iii) for appending new regions found.

We implemented it as a concurrent extensible array, with a "push at end" operation, random read accesses, and iteration: either `tbb::concurrent_vector`, or our simple lock-free implementation based on an array with a large, statically defined maximal capacity, using atomic operations for increasing the current number n_{fill} of items; the latter has the advantage of not needing TBB. There is a little subtlety involved here, in both implementations: n_{fill} is incremented atomically before the item is pushed, so that if two items are pushed concurrently, they are pushed to different slots. However, not all items in slots $0 \dots n_{\text{fill}} - 1$ are ready for reading: there may be items currently being written in the last slots. We therefore need a second index, $n_{\text{ready}} \leq n_{\text{fill}}$, such that all slots $0 \dots n_{\text{ready}} - 1$ contain data ready for reading, which is updated as in

[12] `https://www.threadingbuildingblocks.org/`

Algorithm 3.

Algorithm 3 Concurrent push on the shared region structure.

procedure PUSH_REGION(R)
  atomic ($i \leftarrow n_{\text{fill}}$; $n_{\text{fill}} \leftarrow n_{\text{fill}} + 1$)
  $regions[i] \leftarrow R$
  while $n_{\text{ready}} < i$ **do**
    possibly use a condition variable instead of spinning
  end while ▷ $n_{\text{ready}} = i$
  atomic $n_{\text{ready}} \leftarrow i + 1$
end procedure

The working set is managed differently depending on whether we use TBB or OpenMP. For TBB, we use `tbb::parallel_do`, which dynamically schedules tasks from the working set on a number of threads and allows dynamically adding tasks to the working set. For OpenMP, when we run a task, we collect all the tasks (R, D) that it generates for spawning, and we spawn them at the end.

The resulting implementation however had disappointing performance. In fact, we obtained better performance using a naive OpenMP implementation that collected all tasks to spawn, spawned them and waited for them to complete, using a barrier, before the next round of spawning!

We identified the reason. It was frequent that the working set contained two tasks (R_1, D_1) and (R_2, D_2) such that the regions generated from D_1 and D_2 were the same. In that case, there were two tasks that each solved a linear program, found the same basis, reconstructed exactly the solution and the parametric objective function in that basis, etc. The workaround was to add a hash table (either a `tbb::concurrent_unordered_set` or a normal hash table, protected by a mutex) that stores the set of bases (each basis being identified by the ordered set of its nonbasic variables) that have been processed or are currently under processing. A task aborts after solving the floating-point linear program if it finds a basis identical to one already in the table.

The overall algorithm is simple: create an initial task (none, D_0) and then run the tasks over available threads as long as there are uncompleted tasks (Algorithm 4).

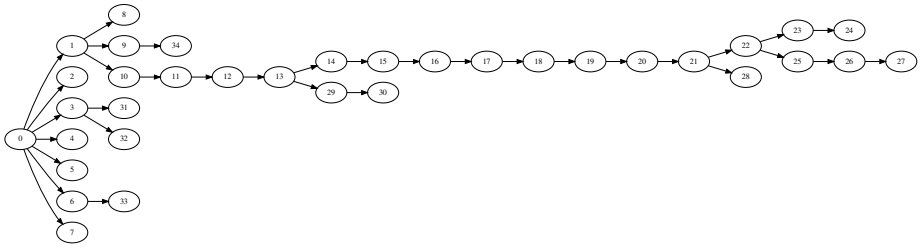
The number of tasks running to completion (not aborted early due to a test) is the same as the number of generated regions. Thus, if, geometrically, the problem does not have many enough regions in comparison to the number of available threads of execution, its parallelism is intrinsically limited.

The *is_covered*$(D, regions)$ loop can be easily parallelized as well. We opted against it as it would introduce a difficult-to-tune second level of parallelism.

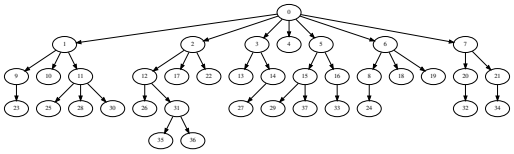
Figure 3 presents the tasks generated by the sequential and the parallel algorithms on a real example, and their dependencies. Figure 3a show how the sequential algorithm handle tasks: as long as the tasks generate subtasks, these subtasks are computed. For instance, task 10 has a long series of descendants

that are computed sequentially. Figure 3b show how the tasks are handled in parallel: after an initial task 0 that generates a set of subtasks (tasks 1 to 9), these subtasks are computed in parallel, as well as the subtasks they generate.

This figure also illustrates the parallelism extracted from the computation, on a polyhedron involving 29 constraints and 16 variables. In particular, we can see on Fig. 3b the number of parallel tasks and their dependencies. For instance, task 5 is generated by task 0 and generates tasks 15 and 16. We can see that, at the beginning of the computation, task 0 generates 7 tasks (i.e., regions to compute). Hence, 7 cores will be used by the parallel algorithm. Since the computation time of each task varies a lot between tasks, the second level of the tree is not necessarily executed at the same time.



(a) Task graph with 1 thread



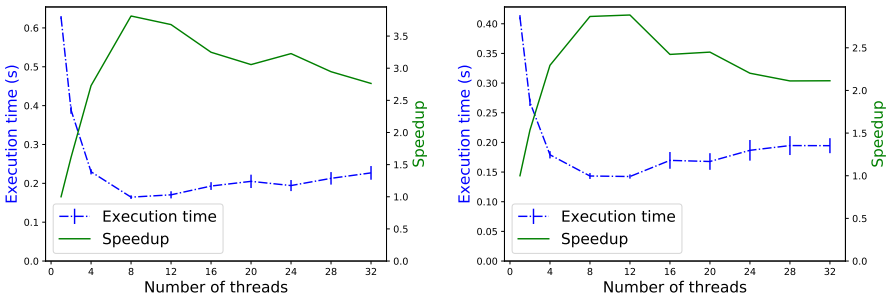
(b) Task graph with multiple threads

Figure 3: Generation graph of the regions from one typical polyhedron, computed with 1 thread and 30 threads. The region graphs, depending on overlaps etc., are different; the numbers in both trees have no relationship.

5 Performance evaluation

We implemented our parallel algorithms in C++, with three alternate schemes selectable at compile-time: no parallelism, OpenMP parallelism, TBB. We use:

Eigen for floating-point matrix computations outside of linear programming. Eigen supports internal OpenMP parallelism; we disabled it since these computations take very little time in our overall execution and using it would have entailed tuning for two levels of parallelism. We used Eigen 3.3.4.



(a) 2 dimensions projected

(b) 5 dimensions projected

Figure 4: Computation time and speedup for 20 projections of polyhedra with 9 constraints with no redundant constraints and 16 variables, on Paranoia (20 hyperthreaded cores, OpenMP). Each parametric linear program has 2–36 regions

GLPK for floating-point linear programming. This library was not thread-safe, meaning it was impossible to solve linear programs in multiple threads at the same time. We suggested to the maintainer making one global variable thread-local, which solved the problem and was made the default as of GLPK version 4.61; we used 4.63. GLPK has no internal parallelism.

Flint for computations on rationals and rational matrices. This library is designed to be thread-safe, but does not use parallelism by itself, at least for the operations that we use. We used Flint 2.5.2.

All the benchmarks were run on the Paranoia cluster of Grid'5000 [7] and on a server called Pressembois. Paranoia features 8 nodes, each equipped with 2 Intel Xeon E5-2660v2 CPUs (10 cores or 20 threads/CPU, 40 threads per node) and 128 GB of RAM. Although the network was not used for these experiments, each node as two 10 Gbps and one 1 Gbps Ethernet NICs. The code was compiled using GCC 6.3.1 and OpenMP 4.5 (201511). The nodes run a Linux Debian Stretch environment with a 4.9.0 kernel. Pressembois fetures 2 Intel(R) Xeon(R) Gold 6138 CPU (20 cores or 40 threads/CPU, 80 threads per node) and 192 GB of RAM. It runs a 4.9 Linux kernel, and we used GCC 6.3 to compile the code. Every experiment was run 10 times, and the plots presented in this section provide the average and standard deviation. Paranoia was used for the OpenMP experiments, whereas Pressembois was used for the TBB experiments.

We evaluated our parallel parametric linear programming implementation by using it to project polyhedra. This is a very fundamental operation on polyhedra, since it is used for computing forward images (image of a polyhedron by an affine linear transformation), and may also be used for computing convex hulls, although there exists a more direct approach for that.

We used a set of typical polyhedra, with different characteristics: numbers of dimensions and of constraints, sparsity, number of dimensions to be projected. Here we present a subset of these benchmarks. Each benchmark comprises 50 to 100 polyhedra.

Our polyhedra were randomly generated. The reason is that it is difficult to obtain polyhedra typically used for the target application (static analysis): because libraries based on the dual description approach behave exponentially with respect to the dimension, static analysers are typically designed to keep low the dimension of the polyhedra, at the expense of analysis precision.[13]

On problems that have only few regions, not enough parallelism can be extracted to exploit all the cores of the machine. For instance, Figure 4 presents two experiments on 2 to 36 regions using the OpenMP version. It gives an acceptable speedup on a few cores (up to 10), then the computation does not generate enough tasks to keep the additional cores busy.

As expected, when the solution has a large number of regions, computation scales better. Figure 5 presents the performance obtained on polyhedra made of 24 constraints, involving 8 to 764 regions and using the OpenMP version. The speedup is sublinear. This is likely due to the synchronizations when tasks are created (lookup in the hash table), some contention on shared data structures and task management.

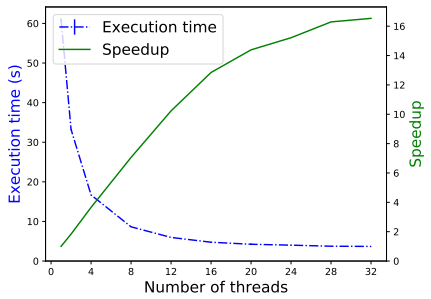
On larger polyhedra, with 120 constraints and 50 variables, we can see however that the speedup is close to a linear one with OpenMP as well as with TBB (Figure 8).

The speedup is good for larger problems with many regions (which can be computed independently from each other): the number of tasks is larger than the number of available cores, hence allowing an efficient parallel computation (Fig. 6). When the problem does not have enough regions to allow the algorithm to extract enough parallelism to use all the available cores, the speedup is bounded by how many independent tasks are generated. We have seen an example presented Figure 3, where at the beginning of the computation, 7 cores at most can be used in parallel. Hence, a plateau occurs when the problems do not have enough regions (Fig. 9); the limited width of the region graph then limits parallelism.

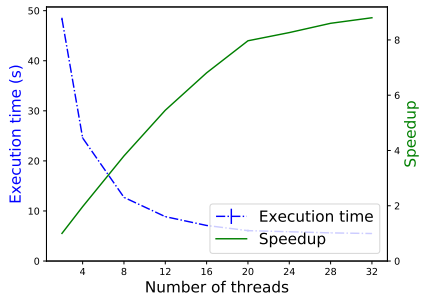
6 Conclusion and future work

We have successfully parallelized computations over convex polyhedra represented as constraints, most importantly parametric linear programming. Speedups are particularly satisfactory for the most complex cases: polyhedra with many constraints in higher dimension.

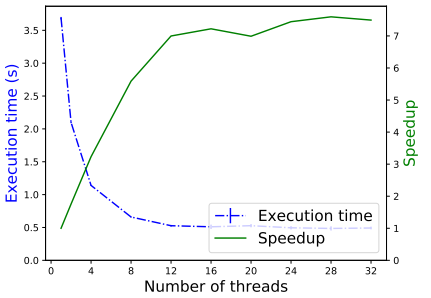
[13] There is a chicken-and-egg problem there: static analysis tools do not use general convex polyhedra or only in low dimension due to the cost of the dual description in higher dimension, thus there is little incentive to develop libraries more efficient in higher dimension. Designers of such libraries then lack higher dimension examples from static analysis.



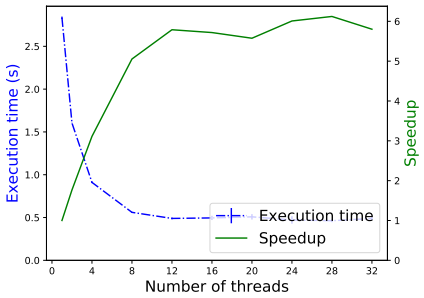
(a) 2 dimensions projected, 4 redundant constraints



(b) 5 dimensions projected, 4 redundant constraints



(c) 2 dimensions projected, 12 redundant constraints



(d) 5 dimensions projected, 12 redundant constraints

Figure 5: Computation time and speedup for different numbers of projections of polyhedra with 10 variables and variable numbers of redundant constraints, on Paranoia (20 hyperthreaded cores, OpenMP). Each parametric linear program has 8–764 regions.

The main cause of inefficiency in our current implementation is geometrical degeneracy, which causes overlapping regions. Several approaches are being studied in this respect, all based on enforcing that, for given parameters, there should be only one optimal basis.[14]

The floating-point simplex algorithm is restarted from scratch in each task. It could be more efficient to store in the task structure the last basis used, or even the full simplex tableau, to start solving from that basis instead of the default initial. The intuition is that neighboring regions are likely to have similar bases.

[14]For some of our applications, such as polyhedral projection, there cannot be several optimal vertices for the same parameters, except at region boundaries, so that source of degeneracy is not present. There however remains the other source, that is, several bases for the same vertex.

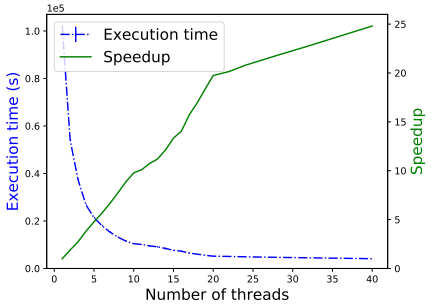


Figure 6: Computation time and speedup for 50 projections of polyhedra in dimension 120, on Paranoia (20 hyperthreaded cores, OpenMP). Each parametric linear program has 3460–3715 regions.

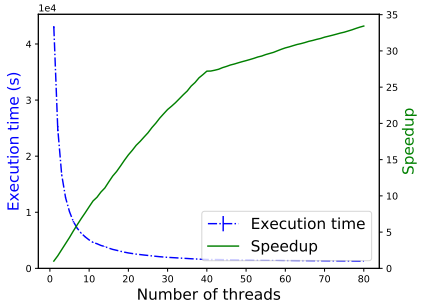
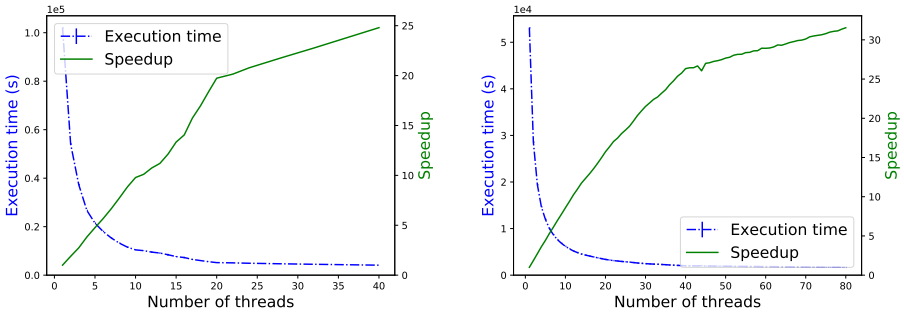


Figure 7: Computation time and speedup for 50 projections of polyhedra in dimension 100, on Pressembois (40 hyperthreaded cores, TBB).

We have presented an approach to handle the heterogeneous, non-predictible computation time, and lack of predictibility of the domain decomposition. One major challenge of this problem is that each subtask covers an area of the domain, but the size and shape of this area is not known before the task is actually computed. In addition, precautions must be taken to avoid computing the same area multiple times on multiple tasks.

Checking whether a vector belongs to a region that has already been processed is currently implemented following a very simplistic approach: the vector is searched for in every region. A binary space partitioning region storage could be used instead: build a tree with nodes adorned by hyperplanes, all regions wholly on one side of the hyperplane under the first child, all regions wholly on the other side under the second child, all regions straddling the hyperplane in both branches, and so recursively in the branches. Then a region covering a vector is searched for by testing, at each node, on which side of the hyperplane the vector lies, and entering the branch. An appropriate locking scheme would however have to be designed, hence introducing synchronizations and requiring more collaboration from the operating system.

Our coarse-grained parallelism for parametric linear programming dependent highly on the geometry of the problem. If there are too few regions, too few tasks will keep the cores busy and little speedup will be achieved. One could use two levels of parallelism, with parallel linear programming, parallel exact reconstruction, etc... onn each task, using parallel matrix computations. One possibility would be to parallelize floating-point linear programming and exact matrix computations, both handled by external libraries. These phases are based on matrix operations (either floating-point, rational, or integer modular), so the usual parallelization schemes for matrix computations should apply. One should keep in mind, however, that our matrices are much smaller than the



(a) OpenMP on Paranoia

(b) TBB on Pressembois

Figure 8: 120 constraints, 50 variables, 1 dimension projected, 3459–3718 regions.

ones typically used in high performance computing applications. One option could be to replace GLPK by another library based on BLAS, such as Coin-OR LP, and use a parallel BLAS implementation such as GotoBLAS or Intel MKL, using an adaptative, hierarchical parallelism, as mentionned in [9]. With respect to Flint, we attempted to run certain internal matrix computation loops in parallel, but gave up due to crashes; perhaps this could be done with more careful examination of shared data structures or the collaboration of the library maintainer.

While our approach based on (in)equalities only does not blow up exponentially when the polyhedron is a Cartesian product of simple polyhedra, as opposed to the double description, it would likely benefit from being applied separately to terms of a Cartesian product as opposed to over the full product, perhaps in parallel. Some computations are performed over dense matrices, and, even if sparseness is exploited, it is easier to exploit it at a coarse level. We think of combining our approach with a Cartesian decomposition [17].

Acknowledgements

Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see `https://www.grid5000.fr`).

References

[1] David Monniaux Alexandre Maréchal and Michaël Périn. "Scalable minimizing-operators on polyhedra via parametric linear programming". In: *Static*

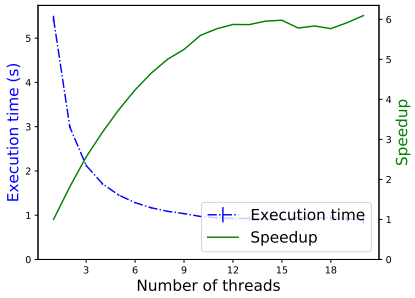


Figure 9: Computation time and speedup for smaller, simpler polyhedra: 16 projections in dimension 29, on Paranoia (20 cores, OpenMP).

analysis (SAS). Ed. by Francesco Ranzato. Springer, 2017. DOI: 978-3-319-66706-5_11. HAL: hal-01555998.

[2] David Avis. "A revised implementation of the reverse search vertex enumeration algorithm". In: *Polytopes — Combinatorics and Computation*. Ed. by Gil Kalai and Günter M. Ziegler. Vol. 29. DMV Seminar. Deutsche Mathematiker-Vereinigung. Birkhäuser, 2000. ISBN: 978-3-7643-6351-2. DOI: 10.1007/978-3-0348-8438-9_9.

[3] David Avis and Charles Jordan. "mplrs: A scalable parallel vertex/facet enumeration code". In: *Math. Program. Comput.* 10.2 (2018), pp. 267–302. DOI: 10.1007/s12532-017-0129-y.

[4] 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". In: *Science of Computer Programming* 72.1–2 (2008), pp. 3–21. DOI: 10.1016/j.scico.2007.08.001. arXiv: cs/0612085.

[5] Sylvain Boulmé et al. "The Verified Polyhedron Library: an overview". In: *20th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC 2018)*. Ed. by Erika Ábrahám et al. IEEE Computer Society, 2019, pp. 9–17. ISBN: 978-1-7281-0625-0.

[6] David Cantor, Basil Gordon, and Bruce L. Rothschild, eds. *Selected papers of Theodore S. Motzkin*. Birkhäuser, 1983. ISBN: 3-7643-3087-2.

[7] Franck Cappello et al. "Grid'5000: A large scale and highly reconfigurable grid experimental testbed". In: *Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing CD (SC—05)*. IEEE/ACM. Seattle, Washington, USA, Nov. 2005, pp. 99–106.

[8] Camille Coti, David Monniaux, and Hang Yu. "Parallel parametric linear programming solving, and application to polyhedral computations". In: *International conference on computational science (ICCS)*. Springer, 2019, pp. 566–572. DOI: 10.1007/978-3-030-22750-0_52. HAL: hal-02097321.

[9] Camille Coti et al. "Solving 0-1 quadratic problems with two-level parallelization of the BiqCrunch solver". In: *2017 Federated Conference on Computer Science and Information Systems (FedCSIS)*. IEEE. 2017, pp. 445–452.

[10] Patrick Cousot and Nicolas Halbwachs. "Automatic discovery of linear restraints among variables of a program". In: *SIGACT-SIGPLAN Symposium on Principles of programming languages (POPL)*. Ed. by Alfred V. Aho, Stephen N. Zilles, and Thomas G. Szymanski. ACM. ACM Press, 1978, pp. 84–96. DOI: 10.1145/512760.512770.

[11] Alexis Fouilhé. "Revisiting the abstract domain of polyhedra : constraints-only representation and formal proof". PhD thesis. Saint Martin d'Hères, France: Université Grenoble Alpes, 2015. TEL: tel-01286086.

[12] Alexis Fouilhé and Sylvain Boulmé. "A certifying frontend for (sub)polyhedral abstract domains". In: *Verified software: theories, tools and experiments (VSTTE)*. Vol. 8471. Lecture Notes in Computer Science. Springer, 2014, pp. 200–215. DOI: 10.1007/978-3-319-12154-3_13. HAL: hal-00991853.

[13] Alexis Fouilhé, David Monniaux, and Michaël Périn. "Efficient generation of correctness certificates for the abstract domain of polyhedra". In: *Static analysis (SAS)*. 2013, pp. 345–365. ISBN: 978-3-642-38855-2. DOI: 10.1007/978-3-642-38856-9_19. HAL: hal-00806990.

[14] Joseph Fourier. "Histoire de l'Académie, partie mathématique (1824)". In: *Mémoires de l'Académie des sciences de l'Institut de France*. Vol. 7. Gauthier-Villars, 1827, xlvij–lv. Gallica: ark:/12148/bpt6k32227/f53.

[15] Robert M. Freund and James B. Orlin. "On the complexity of four polyhedral set containment problems". In: *Math. Program.* 33.2 (1985), pp. 139–145. DOI: 10.1007/BF01582241.

[16] Nicolas Halbwachs. "Détermination automatique de relations linéaires vérifiées par les variables d'un programme". French. PhD thesis. Grenoble, France: Université Scientifique et Médicale de Grenoble & Institut National Polytechnique de Grenoble, Mar. 1979. HAL: tel-00288805. URL: https://tel.archives-ouvertes.fr/tel-00288805.

[17] Nicolas Halbwachs, David Merchat, and Laure Gonnord. "Some ways to reduce the space dimension in polyhedra computations". In: *Formal Methods in System Design* 29.1 (2006), pp. 79–95. DOI: 10.1007/s10703-006-0013-2. HAL: hal-00189633.

[18] Bertrand Jeannet and Antoine Miné. "Apron: a library of numerical abstract domains for static analysis". In: *Computer aided verification (CAV)*. Ed. by Ahmed Bouajjani and Oded Maler. Vol. 5643. Lecture Notes in Computer Science. Springer, 2009, pp. 661–667. DOI: 10.1007/978-3-642-02658-4_52. eprint: hal-00786354.

[19] Colin. Jones et al. "On polyhedral projections and parametric programming". In: *J. Optimization Theory and Applications* 138.2 (2008), pp. 207–220. DOI: 10.1007/s10957-008-9384-4.

[20] Colin N. Jones, Eric C. Kerrigan, and Jan M. Maciejowski. "Lexicographic perturbation for multiparametric linear programming with applications to control". In: *Automatica* (43 2007). DOI: 10.1016/j.automatica.2007.03.008.

[21] Colin N. Jones and Jan M. Maciejowski. "Reverse Search for Parametric Linear Programming". In: *Proceedings of the 45th IEEE Conference on Decision and Control*. IEEE, Dec. 2006, pp. 1504–1509. DOI: 10.1109/CDC.2006.377799.

[22] Leonid Khachiyan. "Fourier-Motzkin elimination method". In: *Encyclopedia of Optimization*. Ed. by Christodoulos A. Floudas and Panos M Pardalos. 2nd ed. Springer, 2009, pp. 1074–1076. ISBN: 978-0-387-74760-6.

[23] Leonid Khachiyan et al. "Generating all vertices of a polyhedron is hard". In: *Discrete & Computational Geometry* 39.1-3 (2008). Also as DIMACS TR 2005-21 http://archive.dimacs.rutgers.edu/pub/dimacs/TechnicalReports/TechReports/2005/2005-21.pdf, pp. 174–190. DOI: 10.1007/s00454-008-9050-5.

[24] Hervé Le Verge. *A note on Chernikova's Algorithm*. Tech. rep. 635. Rennes, France: IRISA, 1992. HAL: inria-00074895.

[25] Alexandre Maréchal. "New algorithmics for polyhedral calculus via parametric linear programming. (Nouvelle algorithmique pour le calcul polyédral via programmation linéaire paramétrique)". PhD thesis. Saint Martin d'Hères, France: Université Grenoble Alpes, 2017. TEL: tel-01695086.

[26] Alexandre Maréchal and Michaël Périn. "Efficient elimination of redundancies in polyhedra by raytracing". In: *Verification, model checking, and abstract interpretation (VMCAI)*. Ed. by Ahmed Bouajjani and David Monniaux. Vol. 10145. Lecture Notes in Computer Science. Springer, 2017, pp. 367–385. DOI: 10.1007/978-3-319-52234-0_20. HAL: hal-01385653.

[27] Alexandre Maréchal et al. "Polyhedral approximation of multivariate polynomials using Handelman's theorem". In: *Verification, model checking, and abstract interpretation (VMCAI)*. Ed. by Barbara Jobstmann and K. Rustan M. Leino. Vol. 9583. Lecture notes in computer science. Springer, Jan. 2016, pp. 166–184. DOI: 10.1007/978-3-662-49122-5_8. HAL: hal-01223362.

[28] Antoine Miné. "The octagon abstract domain". In: *Higher-Order and Symbolic Computation* 19.1 (2006), pp. 31–100. DOI: 10.1007/s10990-006-8609-1. HAL: hal-00136639.

[29] Theodore S. Motzkin et al. "The double description method". In: *Contributions to the theory of games, vol. II*. Ed. by Harold W. Kuhn and Albert W. Tucker. Vol. 28. Annals of Mathematics Studies. Reprinted as [6, Ch. 2]. Princeton University Press, 1953, pp. 51–74. ISBN: 0691079358.

[30] *OpenMP Application Programming Interface*. 4.5. OpenMP Architecture Review Board. Nov. 2015. URL: https://www.openmp.org/wp-content/uploads/openmp-4.5.pdf.

[31] Hang Yu and David Monniaux. "An efficient parametric linear programming solver and application to polyhedral projection". In: *Static analysis (SAS)*. Ed. by Bor-Yuh Evan Chang. Vol. 11822. Lecture notes in computer science. Springer, 2019, pp. 203–224. DOI: 10.1007/978-3-030-32304-2_11. URL: https://doi.org/10.1007/978-3-030-32304-2_11.

Algorithm 4 Task for parallel linear programming solver.

push_tasks adds new tasks to those to be processed; its implementation is different under TBB and OpenMP.
test_and_insert(T, x) checks whether x already belongs to the hash table T, in which case it returns true; otherwise it adds it and returns false. This operation is atomic.

```
procedure PROCESS_TASK((R_from, D))
    R_cov ← is_covered(D, regions)
    if R_cov == none then
        basis ← float_lp(A, B, C(D))
        if ¬test_and_insert(bases, basis) then
            X* ← exact_point(basis)
            o ← exact_objective(basis)
            if ¬(X* ≥ 0 ∧ o ≤ 0) then
                (basis, X*) ← exact_lp(A, B, C(D))
            end if
            S ← sign_conditions(basis)
            R ← eliminate_redundancy(S)
            for each constraint i in R do
                D_next ← compute_next(R, i)
                push_tasks(D_next)
            end for
            push_region(R, X*)
            R_cov ← R
        end if
    end if
    if ¬are_adjacent(R_from, R_cov) then
        D' ← midpoint(R_from, R_cov, D)
        W ← W ∪ {(R_from, D')}
    end if
end procedure

procedure is_covered(D, regions))
    for i ∈ 0 ... n_ready − 1 do        ▷ n_ready to be read at every loop iteration
        (R, X*) ← regions[i]
        if D covered by R then
            return(R)
        end if
    end for
    return(none)
end procedure
```