Around Cooper's Algorithm with Maple

, Ontario Research Center for Computer Algebra
The University of Western Ontario, Canada

Recommended Reference Format:
. 2024. Around Cooper's Algorithm with Maple. 1, 1 (July 2024), 8 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Given a system of linear inequalities in x, $P(x)$, $\exists x P(x) \leftrightarrow true \vee false$ can be solved by the following: Integer hull computation; Integer Linear Programming (ILP) [4]; integer point counting using Barvinok's Algorithm [1] for counting integer points in polytopes as applied by M. Köppe and S. Verdoolaege [9]; the Omega Test [13]; and Integer Point Decomposition [6].

Now, given systems of linear inequalities $P(x, y)$ and $Q(y)$, $\exists x P(x, y) \leftrightarrow Q(y)$, where the values of y that make $Q(y)$ true are exactly those that make $\exists x P(x, y)$ true, can be solved by: Parametric Integer Linear Programming [4]; parametric integer point counting using Barvinok's Algorithm (forthcoming); the Omega Test [13]; and Integer Point Decomposition [6]. More linear algebra methods, like Hermite Normal Form (HNF), should be used to handle explicit or implicit equalities in P(x) and P(x, t).

The system of inequalities (and sometimes equalities also), can be expressed in the language of Presburger Arithmetic. However, computing with such an arithmetic is costly as there is a doubly exponential algebraic complexity lower bound in terms of the length of the input formula for *any* non-deterministic algorithm used to determine the truth of the formula; however, this is for the worst-case where the input formula has an alternation of existential and universal quantifiers [5]. Furthermore, a decision procedure like Cooper's Algorithm [2] has a triply exponential upper bound in terms of the the length of the input formula [11]. As a result, [3, 8, 12] look at the possibilities of using multicore parallelism.

First, we will look at Cooper's algorithm, followed by general considerations regarding its parallelization. Next, comes parallelization using the MAPLE language in both the Task and Grid Programming Models. We follow this up with experimental results, to end finally with improvements and optimizations.

2 Cooper's Algorithm

The steps in Cooper's algorithm by [12] are

- Steps 1 - 3: Normalize formula.
- Step 4: Get lcm of coefficients (calculate δ).
- Step 5A : Substitute comparison constraints with True and False.
- Step 5B: Substitute using lower-bound literals: for-loop for $|L|$ substitutions, where L is the set of lower bound constraints (of the form $b_i < x$ where b_i is x-free).
- Step 6: Gather results

Author's address:, Ontario Research Center for Computer Algebra
The University of Western Ontario, 1151 Richmond St, London, Canada, cmaligec@uwo.ca.

Step 5 is divided into two parts as it is the result of a case distinction from Theorem 1.

Tasks 1 - 4 above can each be estimated as being comparable to l, the number of literals (algebraic expressions) in the formula [12] (e.g., getting the lcm of coefficients requires reading each literal). Task 5 requires a for-loop with upper limit $|L|$, so we have an estimate of $|L| * l$, where L is the set of constraints of type $b_i < x$ (where b_i is x-free for all i), and task 6 can be ignored [12] as it simply requires bounding the results of task 5 with a big disjunction with upper limit δ.

The algorithm removes quantifiers using the following logical expression [12]

$$\exists x F(x) \equiv \bigvee_{k=1}^{\delta} F_{-\infty}(k) \vee \bigvee_{k=1}^{\delta} \bigvee_{b_i} F(b_i + k) \tag{1}$$

where b_i is x-free, and δ is the least common multiple of the coefficients of x and of the divisors of the divisibility constraints in F, and $F_{-\infty}(k)$ is false if there is at least one inequality of the form $b_i < x$; otherwise, it is the conjunction of the divisibility constraints [2]. The first half of the formula corresponds to Step 4, while the second half corresponds to Step 5. What follows is the algorithm for the serial implementation of Cooper's Algorithm.

Algorithm 1: Serial Cooper's Algorithm

```
Input: An existentially quantified conjunction of inequalities (and maybe equalities), F.
Output: A boolean indicating whether there are integer solutions.
1  Eliminate negation;
2  Normailize comparison operators;
3  Normalize coefficients;
4  Calculate $\delta$;
5  FTruth := true;
6  for f in F do
7      if f has the form $x < a$ then
8          FTruth := FTruth and true;
9      else if f has the form $a < x$ then
10         FTruth := FTruth and false;
11         lowerbounds := [op(lowerbounds), a];
12     else
13         for i from 1 to $\delta$ do
14             FTruth := FTruth or evalb(f(i))          ▷ Where f is a divisibility constraint.
15 if Ftruth = true then
16     return true;
17 for i from 1 to $\delta$ do
18     for j from 1 to num(lowerbounds) do
19         b := lowerbounds[j];
20         arg := b + i;
21         cTruth := true;
22         for f in F do
23             cTruth := cTruth and evalb(f(arg)) ;
24             if cTruth = true then
25                 return true;
26 return false;
```

Bottlenecks: (i) There is expression swell as the number of disjuncts becomes huge after a number of quantifier eliminations (each elimination generates at least $|L|+1$ disjuncts). One way to manage this is to expand big disjuncts only when necessary [12]. (ii) The number of lower-bound constraints grows quite large with eliminations; however, there is a dual version of Cooper's Algorithm using upper-bound constraints instead of lower-bound ones, and we can always pick the version that has the fewer number of constraints [2, 12].

3 Parallelizing Cooper's Algorithm

The main source of parallelization is obtained from Formula (1) itself [12]. If L is the set of $ax > t$ constraints in ϕ, then there are $|L| + 1$ substitutions of this constraint required, and as these substitutions are independent, they can be done in parallel [12, 3]. This parallelization is what is attempted in this paper. An optimization would be to push existential quantifiers within disjunctions, so that we can have disjunctions of quantified formulas, each one individually more likely to fit into cache before moving on to the next round of eliminations subject to Formula (1) [12]. Note that $|L|$ increases greatly with each elimination of a quantifier, so memory usage can increase dramatically [3]. This optimization is not attempted as we only implement a univariate version of the algorithm.

Parallelization Factor: The following results were obtained in [12]:

- The Work W = $l + l + l + l + |L| * l = (|L| + 4) * l$.
- As the two "getting" tasks (2 and 3) can be performed in parallel as well as the two "substituting" tasks (4 and 5), the Span S = $l + l + l = 3l$.

Therefore, for one round of quantifier elimination, the Parallelization Factor [12] is PF = $(|L| + 4)/3$.

After eliminating the ith quantifier, the cumulative parallelism factor [12] is

$$PF = \frac{\sum_{i=1}^{n}(|L_i| + 4) \cdot \prod_{j=1}^{i-1}(|L_j| + 1) \cdot l}{\sum_{i=1}^{n} 3 \cdot \prod_{j=1}^{i-1}(|L_j| + 1) \cdot l} \approx \frac{|L_n| + 4}{3} \tag{2}$$

As after each quantifier elimination at least $|L| + 1$ disjuncts are produced, the formula $\prod_{j=1}^{i-1}(|L_j|+1) \cdot l$ is an estimation of the number of literals after the ith quantifier elimination [12]. As $|L_i| >> |L_{i-1}|$ due to the rapid growth in formulas after each elimination, $\frac{|L_n|+4}{3}$ is a good approximation of the LHS in formula (2) [12].

4 Parallelization in the MAPLE Programming Language

In the case of the MAPLE programming language, we can use two types of parallel programming: the Task Programming Model (which executes multiple tasks in a single process), and the Grid Programming Model (which uses multiple processes) [10]. In Task Programming, tasks work together using shared memory, but one must be careful that code running in one task must not interfere with that running in other tasks [10]. As this model is new, much of MAPLE 's core functionality cannot yet be used in a task-based code, unlike Grid Programming, where each process has its own independent memory [10]. However, as the processes are independent, they need to communicate via sending and receiving messages, the cost of which can be high, and balancing work over multiple processors can be difficult [10]. In this paper we use both models for comparison with serial execution.

4.1 The Grid Programming Model

The material covered in this subsection can be found in [10]. To start a grid computation one uses the Grid : −Launch() command, which starts multiple copies of MAPLE 's computation engine, called *nodes*. Each node is independent and does not share memory with the others, so communication between nodes requires the use of messages. Grid : −NumNodes() gives the number of nodes launched, and Grid : −MyNode() indicates the executing node (which can be from 0 to Grid : −NumNodes() - 1). When the function that executes in node 0 completes, node 0 returns to Launch, which returns the result, and all other remaining node executions terminate. Thus, Grid : −Barrier() can be used to assure that no node passes until all others complete. As metioned earlier, nodes can send messages to each other by using Grid : −Send(x, y) where x is the node that will receive the message, and y is the expression that is the message. Grid : −Receive(x) receives messages only from node x, while Grid : −Receive() receives messages from any node. Receive returns the message received.

What follows is the author's version of Cooper's algorithm using the Grid model where lcm is the lowest common multiple of all of the lhs of the divisibility constraints (i.e., lcm = δ).

Algorithm 2: Cooper's Algorithm Grid Model

```
Input: An existentially quantified conjunction of inequalities (and maybe equalities), F.
Output: A boolean indicating whether there are integer solutions.
1  n := Grid:-NumNodes();
2  me := Grid:-MyNode();
3  step := floor (lcm / n);
4  if me = 0 then
5      start := 1;
6      endp := step;
7  else if me = n-1 then
8      start := step*(n-1)+1;
9      endp := lcm;
10 else
11     start := step*me+1;
12     endp := step*(me+1);
13 PC := Parallel_Cooper(F, start, endp);
14 Grid:-Barrier();
15 if me = 1 then
16     local z := Grid:-Receive(0, PC);
17     Grid:-Send(0,PC or z);
18 else if me = 0 then
19     Grid:-Send(1,PC);
20     Grid:-Receive(1) or . . . or Grid:-Receive(n-1);
21 else
22     Grid:-Send(0,PC);
```

4.2 The Task Programming Model

Everything we will talk about in this subsection regarding the Task Model can be found in [10]. A *task* consists of a procedure (function) combined with a set of arguments. The MAPLE kernel can automatically schedule tasks and distribute them to available processors. As an example of the Task Model in action, take the following MAPLE procedure:

$$f := proc() f_c(f_1(args_1), ..., f_n(args_n)); end\ proc : \tag{3}$$

Each $f_i(args_i)$ can be executed by a task t_i, with the procedure of the task being f_i and the associated arguments, the $args_i$. However, just as f_c needs to wait for all the $f_i(args_i)$ to return, so t_c needs to wait for the t_i to complete before it can execute. The procedure associated with t_c is f_c, while the associated arguments are the values returned by the t_i. Here, the *parent task* is t_c, while the *child tasks* are the t_i. The task t_c is called the *continuation task*. It is so called, because when a task t creates a continuation task, it can finish executing without waiting for the child tasks to complete as the continuation task will handle the return. Therefore, any task can replace itself with child tasks and a continuation task. Furthermore, as child tasks can also create more tasks, this forms a *task tree*. Leaf tasks can run at any time as they do not need to wait for any child tasks, and their parents may in turn become leafs as a result. To start a task computation, one uses the Threads : −Task : −Start() command.

What follows is the author's version of Cooper's algorithm using the Tasks model, where lcm is the lowest common multiple of all of the lhs of the divisibility constraints (i.e., lcm = δ), where the sub-routine, cont, gives the combined results of the returns from all the tasks, and where n is the number of tasks.

Algorithm 3: Cooper's Algorithm Task Model

```
Input: An existentially quantified conjunction of inequalities (and maybe equalities), F; an
       int n.
Output: A boolean indicating whether there are integer solutions.
1 if lcm < threshold then
2     Serial_Cooper(F);
3 step := floor(lcm/n);
4 Threads:- Task:-Continue(cont, Tasks=[Parallel_Cooper, seq([i*step+1, (i+1)*step)], i =
    0..n-2), [(n-1)*step+1, lcm)]]);
```

5 Experimentation

These three versions of Cooper's Algorithm have been implemented without negation, for & connectives only, and for one variable. The experiments were run on an HP Laptop 15-dw3xxx, Intel Core i3-1115G4, 1 Processor, 2 Cores, 4 Threads, 2.99 GHz. The cache specifications are as follows: L1 Instruction 32.0 KB x 2, L1 Data 48.0 KB x 2, L2 1.25 MB x 2, L3 6.00 MB x 1. The results are based on the following systems of inequalities (and equalities in some cases, where the equality is transformed into two inequalities): the first eleven test cases are to test correctness, for which all passed, and the cases following those are for performance. Note: L stands for $\leq$, while G stands for $\geq$.

5.1 Correctness Tests

- s1 := "x=34": (equality)
- s2 := "5*x G x - 34": (greater than or equal to)
- s3 := "x L 2": (less than or equal to)
- s4 := "x > 34": (greater than)
- s5 := "3*x+1 = 34": (non-unitary coefficient)
- s6:= "5*x + 1 G 34 & 8*x + 3 > 35": (system of two inequalities)
- s7 := "3*x - 6 G 7 & 5*x + 9 > 8 & 2*x < x + 1": (system with an inequality with the variable on both sides)
- s8 := "x+1 < 3*x & x L 2": (system often used as a running example)
- s9 := "2*x = 7 & x < 5 & 14*x < 6": (system of an equality and inequalities)

- s10 := "2 < x & 3 < x & 4 < x & x G 5": (system of four inequalities)
- s11 := "2 < x & 3 < x & 4 < x & x G 5 & 6 < x & 7 L x & 8 < x & x G 9 & 10 < x & 11 < x & 12 < x & 13 L x & 14 < x & 15 L x & 16 < x": (large system to test step 5B - consists of 15 constraints)

5.2 Performance Tests

- s12 := "3*x + 5 = 60 & x G 24 & 25*x < 12*x + 18 & 35*x > 23 & 21*x > 14*x + 18 & 5*x = 24 & 34*x + 24 G 14*x + 45 & 6*x - 34 L 23 & 65*x - 98 L 17"; (large and complex system - consists of 9 large constraints)
- s13 := "2 < x & 3 < x & 4 < x & x G 5 & 2 < x & 3 < x & 4 < x & x G 5 & 6 < x & 7 L x & 8 < x & x G 9 & 10 < x & 11 < x & 12 < x & 13 L x & 14 < x & 15 L x & 16 < x & 3*x + 5 = 60 & x G 24 & 25*x < 12*x + 18 & 35*x > 23 & 21*x > 14*x + 18 & 5*x = 24 & 34*x + 24 G 14*x + 45 & 6*x - 34 L 23 & 65*x - 98 L 17": (very large and complex system with 28 constraints - it consists of systems s10-s12)
- s14 := "54*x L 98 - 54*x & 45*x > 78*x - 25 & 88*x + 78 < 26*x & 78*x = 67*x = 67":
- s15 := "34*x < 23*x - 879 & -16*x = 34 + 9*x & 45*x L 79 - 87*x & 56*x + 87 G 23 - 2*x":
- s16 := s16 := "46 + 78*x > 89 & 21*x = 67 - 98*x & 56*x + 87 = -76*x - 65":
- s17 := "2034 > 76*x - 56 & 78*x < 67*x - 97 & 35*x L 24*x +25 & -15*x G -90*x + 74 & 78*x + 87 > 15*x & 24*x -22 < 18":
- s18 := "67*x - 9 = 34*x & 65*x> 3089*x - 75 & 45*x L 67 & 34*x L 79*x - 55 & 85*x G 23*x - 65":

5.3 Large Numbers of Lowerbounds and Formulas

- s19 := " x > 50 & x > 1 & x > 2 & x > 3 & x > 4 & x > 5 & x > 6 & x > 7 & x > 8 & x > 9 & x > 10 & x > 11 & x > 12 & x > 13 & x > 14 & x > 15 & x > 16 & x > 17 & x > 18 & x > 19 & x > 20 & x > 21 & x > 22 & x > 23 & x > 24 & x > 25 & x > 26 & x > 27 & x > 28 & x > 29 & x > 30 & x > 31 & x > 32 & x > 33 & x > 34 & x > 35 & x > 36 & x > 37 & x > 38 & x > 39 & x > 40 & x > 41 & x > 42 & x > 43 & x > 44 & x > 45 & x > 46 & x > 47 & x > 48 & x > 49": (fifty lowerbounds)
- s20 := cat(s19, "&", s19): (100 lowerbounds)
- s21 := cat(s20, "&", s20, "&", s20, "&", s20, "&",s20, "&", s20, "&",s20, "&", s20, "&",s20, "&", s20): (1000 lowerbounds)
- s22 := cat(s21, "&", s21, "&", s21, "&", s21, "&",s21, "&", s21, "&",s21, "&", s21, "&",s21, "&", s21): (10,000 lowerbounds)

5.4 Experimental Data with Time in ms

S	Serial	Grid	Task	LCM	LB	SL	B	S/G	S/T	log(LCM)
12	1002.2	2014.5	1062.6	5460	6	12	F	0.49749	0.94316	3.73719
13	7915.8	5082.1	5626.7	5460	25	31	F	1.55758	1.40683	3.73719
14	531.2	383.4	433	36828	1	6	F	1.3855	1.22679	4.56618
15	2799.4	1318.5	2211	95700	2	6	F	2.12317	1.26612	4.98091
16	8730.2	3646.2	7371.6	204204	3	6	F	2.39433	1.1843	5.31006
17	84782	38957.3	70235.8	2633400	2	7	F	2.17628	1.20711	6.42052
18	252203	109632	198688	5155920	3	7	F	2.30046	1.26934	6.71231

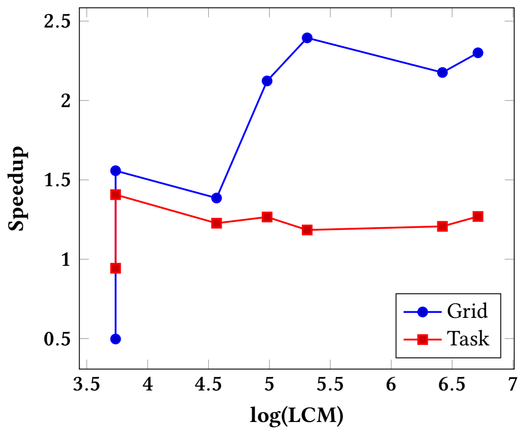

Fig. 1. Speedup with respect to the log of the lcm of the coefficients in a system.

5.5 Analysis

In the correctness tests (s1 - s11), the grid and task models have much longer running times than the serial one. This is due to low lcm's and overhead. Performance test s12 is an outlier; we see a poor grid time and a subpar task time due to unknown reasons. In the performance tests (s13 - s18), we see a change where the grid and task versions are much faster than the serial one with speedups from about 1.2 to 2.3 times. This is due to large lcm's which range from 5,460 to 5,155,920. With the large number of lowerbounds and formulas (s19 - s22), the grid and task models are quite slow due to a low lcm of 1 in each case. In Figure 1, only the performance cases (s12- s18) are graphed as s1 - s11 are just correctness tests, and s19 - s22 have low lcm's (lcm = 1) and high overhead, resulting in no possible advantage over the serial version.

6 Improvements and Optimizations

Cooper's Algorithm focuses on eliminating existential quantifiers, but universal quantifiers can also be handled by adding a few straightforward rules. As a result, eliminating universal quantifiers with linear constraints reduces to solving too:

(1) $(\forall x_1 \ldots x_n\ (a_1x_1 + \ldots + a_nx_n = b)) \Leftrightarrow (a_1 = \ldots = a_n = b = 0)$
(2) $(\forall x_1 \ldots x_n\ (a_1x_1 + \ldots + a_nx_n \neq b)) \Leftrightarrow (a_1 = \ldots = a_n = 0 \wedge b \neq 0)$
(3) $(\forall x_1 \ldots x_n\ (a_1x_1 + \ldots + a_nx_n < b)) \Leftrightarrow (a_1 = \ldots = a_n = 0 \wedge b > 0)$
(4) $(\forall x_1 \ldots x_n\ (a_1x_1 + \ldots + a_nx_n \nless b)) \Leftrightarrow (a_1 = \ldots = a_n = 0 \wedge b \leq 0)$

In the case that there are linear equations among the constraints, instead of converting all of them to inequalities, we could solve them over $\mathbb{Z}$. We recall a procedure for doing so, taken from [7].

Let $A \in \mathbb{Z}^{m\times n}$ be a matrix with rank m and $\mathbf{v} \in \mathbb{Z}^m$ be a vector. Consider the system of linear equations

$$A\mathbf{x} = \mathbf{v}. \tag{4}$$

Let $U_{n\times n}$ be a unimodular matrix so that we have

$$A_{m\times n}U_{n\times n} = [\mathbf{0}_{m\times(n-m)}, R_{m\times m}],$$

which is called the *Hermite Normal Form* of A and R is nonsingular. Then, it is known that $A\mathbf{x} = \mathbf{v}$ has integer solutions if and only if $R^{-1}\mathbf{v} \in \mathbb{Z}^m$ holds. Moreover, writing $U = [U_L, U_R]$, where

$U_L \in \mathbb{Z}^{n\times(n-m)}$, the integer solutions of Equation (4) are of the form

$$\mathbf{x} \;=\; U_R R^{-1}\mathbf{v} + U_L\mathbf{z},$$

where $\mathbf{z}$ is any element of $\mathbb{Z}^{n-m}$.

The following example is from [7]:

$$A = \begin{pmatrix} 6 & 8 & 7 & 3 \\ 2 & 4 & 5 & 4 \\ 3 & 6 & 8 & 9 \end{pmatrix} \text{ and } \mathbf{v} = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}.$$

The Hermite Normal Form H of A and the unimodular transformation matrix U are given by:

$$H = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \text{ and } U = \begin{pmatrix} -26 & -19 & -25 & 3 \\ 39 & 29 & 36 & -4 \\ -24 & -18 & -21 & 2 \\ 4 & 3 & 3 & 0 \end{pmatrix}.$$

Here the matrix R and the general form of the solution are given by

$$R = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \text{ and } \mathbf{x} = U_R R^{-1}\mathbf{v} + U_L\mathbf{z} = \begin{pmatrix} -60 \\ 89 \\ -54 \\ 9 \end{pmatrix} + z\begin{pmatrix} -26 \\ 39 \\ -24 \\ 4 \end{pmatrix}, z \in \mathbb{Z}.$$

Another optimization would be to use Barvinok's algorithm, which calculates a generating function, which in turn, calculates the *number* of integer points in a polytope as opposed to Cooper's Algorithm, which just indicates whether such points exist or not. Then there is the MAPLE command IntegerPointDecomposition. Its algorithm, IntegerSolve(K), decomposes the polyhedron K's integer points ($K \cap \mathbb{Z}^d$) into the disjoint union $(K_1 \cap \mathbb{Z}^d)\dot{\cup}\ldots\dot{\cup}(K_e \cap \mathbb{Z}^d)$, where $K_1, \ldots, K_e$ are simpler polyhedra, such that $(K_i \cap \mathbb{Z}^d) \neq \emptyset$ for all $1 \leq i \leq e$ [6].

References

[1] Barvinok, A. I. A polynomial time algorithm for counting integral points in polyhedra when the dimension is fixed. *Mathematics of Operations Research 19*, 4 (1994), 769–779.

[2] Cooper, D. C. Theorem proving in arithmetic without multiplication. *Machine intelligence 7*, 91-99 (1972), 300.

[3] Dung, P. A., and Hansen, M. R. From functional programming to multicore parallelism: A case study based on presburger arithmetic. In *23rd Nordic Workshop on Programming Theory* (2011).

[4] Feautrier, P. Automatic parallelization in the polytope model. *The Data Parallel Programming Model: Foundations, HPF Realization, and Scientific Applications* (2005), 79–103.

[5] Fischer, M. J., and Rabin, M. O. Super-exponential complexity of presburger arithmetic. In *Quantifier Elimination and Cylindrical Algebraic Decomposition*. Springer, 1998, pp. 122–135.

[6] Jing, R.-J., and Moreno Maza, M. Computing the integer points of a polyhedron, i: algorithm. In *International Workshop on Computer Algebra in Scientific Computing* (2017), Springer, pp. 225–241.

[7] Jing, R.-J., and Moreno Maza, M. Personal communication. 2023.

[8] Karrenberg, R., Košta, M., and Sturm, T. Presburger arithmetic in memory access optimization for data-parallel languages. In *Frontiers of Combining Systems: 9th International Symposium, FroCoS 2013, Nancy, France, September 18-20, 2013. Proceedings 9* (2013), Springer, pp. 56–70.

[9] Köppe, M., and Verdoolaege, S. Computing parametric rational generating functions with a primal barvinok algorithm. *arXiv preprint arXiv:0705.3651* (2007).

[10] Maplesoft. 15 parallel programming, n.d.

[11] Oppen, D. C. A 222pn upper bound on the complexity of presburger arithmetic. *Journal of Computer and System Sciences 16*, 3 (1978), 323–332.

[12] Phan, A.-D., and Hansen, M. R. An approach to multicore parallelism using functional programming: A case study based on presburger arithmetic. *Journal of Logical and Algebraic Methods in Programming 84*, 1 (2015), 2–18.

[13] Pugh, W. The omega test: a fast and practical integer programming algorithm for dependence analysis. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing* (1991), pp. 4–13.