Algorithms for Weighted Counting over Parametric Polytopes: A Survey and a Practical Comparison

Sven Verdoolaege*,1,2, Maurice Bruynooghe*,1

* Katholieke Universiteit Leuven, Celestijnenlaan 200A, B-3001 Leuven, Belgium

ABSTRACT

The polytope model is widely used in compiler analysis for representing a certain class of programs. Many counting problems that occur in the analysis of such programs can be solved by counting the number of integer points in a parametric polytope. In other counting problems, polynomial weights are assigned to the integer points of a parametric polytope and the objective is to find the sum of these weights over all integer points. This paper briefly surveys a number of algorithms for solving such problems. The paper also serves to document some of the algorithms implemented in the freely available barvinok library.

KEYWORDS: Barvinok’s decomposition; Laurent expansion; local Euler-Maclaurin formula; nested sums; piecewise quasi-polynomial; polytope model

1 Introduction

Counting the number of integer points in a parametric polytope is the driving force behind many compiler optimization techniques. Counting examples include the number of memory locations touched by a loop, the number of operations performed by a loop or the amount of cache misses generated by a loop. Program transformations and optimizations exploiting these numbers include parallelization, memory size optimization and cache effectiveness optimization. For a more extensive overview, we refer to [Verd07].

1E-mail: {sven, maurice}@cs.kuleuven.be
2This author was supported by FWO-Vlaanderen.
\[ p = a; \]
\[ for \ (i = 0; i < N; ++i) \]
\[ \quad for \ (j = i; j < N; ++j) \{ \]
\[ \quad \quad p += j * ((j-i)/4); \]
\[ \quad \quad *p = 0; \]
\[ \} \]

Listing 1: Artificial pointer conversion example.

For some compiler techniques, we are not interested simply in the number of integer points satisfying some constraints, but in the sum of some (quasi)polynomial evaluated in each integer point satisfying the constraints. Examples include the amount of memory dynamically allocated by a piece of code \[Brab03\], where we need to sum the amount of memory allocated in each iteration of a loop; the estimation of the worst case execution time \[Lisp03, vE06\], where we need to sum the time required by an operation or a function call in each iteration of a loop; and pointer conversion or array recovery \[vE01, Fran03\], where pointer manipulations are transformed into array indexing.

As an example of array recovery, consider the slightly contrived code in Listing 1 and suppose we want to replace accesses to array \(a\) through pointer \(p\) by explicit accesses to \(a\). To obtain an explicit indexation in terms of the iterators, we need to accumulate the increments to pointer \(p\) over all previous iterations. That is, we need to sum the increment over all \(0 \leq i' < N\) and \(i' \leq j' < N\) such that \((i', j') \preceq (i, j)\), where \(\preceq\) denotes the lexicographical order. Note that \((j-i)/4\) is an integer division, i.e., the increment is

\[
c(i', j') = j' \left\lfloor \frac{j' - i'}{4} \right\rfloor.
\]

The accumulated increment is therefore

\[
d(i, j) = \sum_{(i', j') \in S, (i', j') \preceq (i, j)} c(i', j'),
\]

with \(S = \{ (i', j') \in \mathbb{Z}^2 \mid 0 \leq i' < N \land i' \leq j' < N \}\).

## 2 Weighted Counting

We consider the computation of

\[
s(s) = \sum_{t \in P(s) \cap \mathbb{Z}^n} q(s, t) = \sum_i \sum_{t \in P_i(s) \cap \mathbb{Z}^n} q_i(s, t),
\]

where \(q(s, t)\) is a piecewise quasi-polynomial function, i.e., a function that, within each of a finite set of polyhedra \(P_i\) that form a subdivision of \(P\) and for each residue class modulo a lattice in \(P_i\), behaves as a polynomial. The period of \(q_i(s, t)\) is the index of this lattice. We may assume, w.l.o.g. that \(q(s, t)\) is a quasi-polynomial. A quasi-polynomial can be represented by a “step-polynomial”, i.e., a polynomial expression in greatest integer parts (floors), or by a table of polynomials. We will focus on the step-polynomial representation.
Since only one of the techniques discussed below has any (very limited) support for handling quasi-polynomials directly, we will need various reductions, depending on the applied counting algorithm. A quasi-polynomial can be written as polynomial by introducing a new variable for each distinct floor expressions. This reduction is best performed on each monomial individually, to reduce the number of variables introduced in each counting problem.

For methods that only directly support unweighted counting, we need a further reduction. If \( t_1 \) attains only non-negative values throughout the domain \( P(s) \), we can remove \( t_1 \) from any given monomial by introducing as many new variables as the exponent of \( t_1 \) in the monomial, expressing this power of \( t_1 \) as the number of integer points in a cube. If negative values are possible, we have to split \( P(s) \) and apply transformations on the parts such that only non-negative values remain.

We briefly describe six methods here. For more information, we refer to [Verd08].

- **Nested sums [Sake96]**: an incremental weighted method based on
  \[
  \sum_{t=0}^{m-1} t^k = \frac{1}{k+1} \sum_{n=0}^{k} \binom{k+1}{n} B_n m^{k+1-n}
  \]
  with \( B_n \) the Bernoulli numbers; requires splintering to obtain polynomials from quasi-polynomials and is therefore exponential in the input size.

- **Interpolation [Clau98]**: based on the structure of the resulting formula; exponential in the input size.

- **Barvinok’s algorithm [Barv99]**: an unweighted method based on a decomposition into unimodular cones and a generating function representation
  \[
  f_P(x) = \sum_{t \in \mathbb{Z}^d} t^x, \quad \text{whence } \#P = f_P(1); \quad \text{polynomial in input size, exponential in dimension and degree (due to conversion).}
  \]

- **Local Euler-Maclaurin formula [Berl06]**: a weighted method based on Barvinok’s decomposition and the local Euler-Maclaurin formula
  \[
  \sum_{t \in P(s) \cap \Lambda} p(s)(t) = \sum_{F(s) \in F(P(s))} \int_{F(s)} D_{P(s),F(s)} \cdot p(s);
  \]
  exponential in the dimension.

- **Iterated Laurent Series [Bald08]**: a weighted method based on Barvinok’s decomposition and the observation
  \[
  f(e^y) = \sum_{t \in P \cap \mathbb{Z}^d} e^{ty} = \sum_{t \in P \cap \mathbb{Z}^d} \sum_{n \geq 0} t^n y^n = \sum_{n \geq 0} \left( \sum_{t \in P \cap \mathbb{Z}^d} t^n \right) \frac{y^n}{n!};
  \]
  i.e., \( \sum_{t \in P \cap \mathbb{Z}^d} t^n \) is \( n! \) times the coefficient of \( y^n \) in \( f(e^y) \); exponential in the dimension.

- **Derivation [DL06]**: a weighted method based on Barvinok’s decomposition and the observation
  \[
  \left( x_1 \frac{\partial}{\partial x_1} \right)^{n_1} \cdots \left( x_d \frac{\partial}{\partial x_d} \right)^{n_d} f_P(z) = \sum_{t \in \mathbb{Z}^d} [t \in P] t^n x^t;
  \]
  exponential in the dimension, increasing the “dimension” by the degree.

All but the interpolation and derivation methods have been (partially) implemented in barvinok.
References


