Symbolic Evaluation of Sums for Parallelising Compilers

Rizos Sakellariou
Department of Computer Science, University of Manchester
Oxford Road, Manchester M13 9PL, United Kingdom
e-mail: rizos@cs.man.ac.uk

Keywords: Symbolic Sums, Parallelising Compilers, Loop Parallelisation.

ABSTRACT

The evaluation of sums over polynomials when symbolic, i.e., unknown, variables are involved in the bounds of the sums is considered. Such sums typically occur when analysing, in computer programs, the properties of loops which can be executed in parallel. Existing packages for symbolic mathematical computations are not capable of handling these sums properly. The problems which may arise are identified and an algorithm to overcome them is presented.

INTRODUCTION

The term *parallelising compilers* refers to those software tools which, in a broad sense, aim to transform automatically a program written in a sequential language to a semantically equivalent parallel program [11, 12]. In this process, these tools need to perform an extensive analysis of the sequential code; a large part of this analysis focuses on the investigation of the properties of loop nests, which are generally considered as the richest source of parallelism in scientific programs.
Usually, loop nests are represented by means of a polytope [1]. Then, a number of important questions concerning the behaviour of the program can be answered by counting the number of the integer points (that is, those points having integer coordinates) contained in the polytope. In the context of loop nests, this corresponds to evaluating multiple sums, such as

$$\sum_{i_1=l_1}^{u_1} \sum_{i_2=l_2}^{u_2} \cdots \sum_{i_m=l_m}^{u_m} 1, \tag{1}$$

where each sum applies over the iteration space of a single loop, say j, with bounds l_j, u_j. The result of a sum such as the above can be used, for instance, to predict the execution time of a loop nest by considering how many times each statement of the loop body is executed [8, 9].
It is not known whether a polynomial algorithm for counting the number of integer points in a polytope exists; it appears that the problem is more complicated than the widely studied problem of computing the volume of a polytope [2]. However, in the case of program analysis, the resulting sums are rather simple instances of the general problem, thus permitting a quick evaluation. Their most challenging aspect is that the result must be computed symbolically, in terms of the parameters involved (recall that some of the variables involved in the loop bounds may have a value which is unknown, at compile-time).
Existing packages for symbolic mathematical computations, such as Mathematica, have some capability for evaluating symbolic sums. However, the `SymbolicSum` function used by Mathematica (version 2.2) makes assumptions which may result in an erroneous answer; for instance, it reports that

$$\sum_{i=1}^{b} \sum_{j=a}^{i} 1 = \frac{b(b-2a+3)}{2}.$$

This is correct only if $b \geq 1 \geq a$; if $b \geq a \geq 1$ the correct result is $(b-a+2)(b-a+1)/2$. Mathematica computes the sum correctly whenever a, b have numerical values; in this case, it proceeds by resorting to the run-time execution of a loop nest corresponding to the sum. Clearly, this is a time-consuming approach and closed formulas would be helpful.

In this paper, we formulate the necessary rules for evaluating symbolically sums over polynomials in terms of the sum indices. We incorporate these into an algorithm based on the recursive evaluation of a series of symbolic sums which must satisfy a set of constraints; this approach makes it suitable for use by a symbolic package. The evaluation of sums over integer functions is also considered, since the latter arise often when transforming loops for parallelism. Our algorithm is applied to test examples taken from the literature, which correspond to problems usually tackled by parallelising compilers; the mistakes that can be made when handling such examples are discussed.

BACKGROUND

Multiple sums of the form shown in (1) can be manipulated by evaluating the Σ's from right to left (inside-out, or from the innermost sum to the outermost). The innermost sum can be evaluated by applying the rule

$$\sum_{i=l_m}^{u_m} 1 = \begin{cases} u_m - l_m + 1, & \text{if } u_m \geq l_m, \\ 0, & \text{otherwise.} \end{cases} \tag{2}$$

This rule emanates directly from the semantics of loop structures in programming languages, such as the `DO` ... `ENDDO` in FORTRAN. The evaluation of the remaining sums in (1) can also be based on (2) by making use of some fundamental properties of sums, such as the distributive, the associative, and the commutative law; for an excellent discussion, the reader is referred to [5, Ch. 2].

Sums where the index of the sum is present in the summand can be evaluated using the so-called Bernoulli formula,

$$\sum_{i=1}^{n} i^p = \frac{1}{p+1} \sum_{k=0}^{p} \binom{p+1}{k} B_k (n+1)^{p-k+1}, \quad \text{for all integers } p, n \geq 1, \tag{3}$$

where B_k are the Bernoulli numbers defined by the recurrence relation

$$\sum_{i=0}^{n} \binom{n+1}{i} B_i = 0, \quad \text{for all integers } n \geq 1, \text{ and } B_0 = 1. \tag{4}$$

Note also that, by normalising the loop bounds [11], it is possible to have a lower bound of 1 for all sums. Alternatively, sums not having a lower bound of 1 can be transformed using the commutative law. Thus, the sum

$$\sum_{i=l}^{u} i^p, \quad \text{where } l \leq u, p \geq 1,$$

can be transformed as follows:

- If $l > 1$, then:

$$\sum_{i=l}^{u} i^p = \sum_{i=1}^{u} i^p - \sum_{i=1}^{l-1} i^p. \tag{5}$$

- If $l < 0 < u$, then:

$$\sum_{i=l}^{u} i^p = (-1)^p \sum_{i=1}^{-l} i^p + \sum_{i=0}^{u} i^p. \tag{6}$$

- If $l < u \leq 0$, then

$$\sum_{i=l}^{u} i^p = (-1)^p \sum_{i=-u}^{-l} i^p = (-1)^p \sum_{i=1}^{-l} i^p - (-1)^p \sum_{i=1}^{-u-1} i^p. \tag{7}$$

Similarly, loops having a non-unit stride can be converted to loops with a unit stride.
In order to illustrate the above, we consider the evaluation of the multiple sum

$$I = \sum_{i=5}^{n} \sum_{j=1}^{i} \sum_{k=1}^{j} 1,$$

which corresponds to the number of times, I, the statements in the body of a triple nested loop are executed; it is assumed that $n \geq 5$. Starting from the innermost sum, we have:

$$I = \sum_{i=5}^{n} \sum_{j=1}^{i} j.$$

Applying (3), i.e., Bernoulli's formula, for $p = 1$, we get:

$$I = \sum_{i=5}^{n} \frac{i(i+1)}{2} = \frac{1}{2} \left(\sum_{i=5}^{n} i^2 + \sum_{i=5}^{n} i \right).$$

Applying again Bernoulli's formula and (5), we end up with:

$$I = \frac{1}{2} \left(\frac{n(n+1)(2n+1)}{6} - 30 + \frac{n(n+1)}{2} - 10 \right) = \frac{n^3 + 3n^2 + 2n - 120}{6}.$$

METHODOLOGY

Our approach for handling symbolic variables in sums consists of following strictly the constraints that must hold for the evaluation to be correct, as for instance in (2). Thus, when evaluating multiple sums where unknown variables are involved in the bounds, a number of different outcomes are expected, each of which holds under a given set of constraints. Splitting the final result into a number of different, disjoint outcomes should be regarded as an additive process. For instance, the result of the example of the previous section can also be written as

$$(\frac{n^3 + 3n^2 + 2n - 120}{6})[n \geq 5] + 0[n < 5],$$

where the result of $[exp]$ is 1 if exp is true, or 0 if exp is false, and exp is a true-or-false statement.footnote This notation is adopted in [5, p. 24]; the idea is attributed to Kenneth Iverson, who introduced it in his programming language APL.
The entire approach is summarised, in the form of an algorithm, in the following lines. The input of the algorithm is a multiple sum (initially, the first and only outcome) and the expected output is a set of symbolic expressions (outcomes) each of which holds under a given set of constraints.

Algorithm Evaluate_Symbolic_Sums:

begin

1. Get the next outcome (i.e., a multiple sum) and the corresponding constraint (if any).

2. If the index of the innermost sum occurs in a condition of the constraint, then incorporate this condition in the bounds of the sum. For instance, the expression

$$n = \sum_{i=1}^{n} \sum_{j=l}^{u} 1, \quad \text{if } (j \leq n) \wedge (n \leq 100),$$

can be rewritten as

$$n = \sum_{i=1}^{n} \sum_{j=l}^{\min(u,n)} 1, \quad \text{if } (n \leq 100).$$

3. Eliminate all $\min$s and $\max$s from the bounds of the current sum by adding more outcomes (and constraints). For instance, the sum

$$\sum_{i=\min(a,b)}^{\max(c,d)} 1$$

would be replaced by the following four outcomes (and constraints):

$$\begin{aligned} &\sum_{i=a}^{c} 1, \quad \text{if } (a \leq b) \wedge (c \geq d), \\ &\sum_{i=a}^{d} 1, \quad \text{if } (a \leq b) \wedge (d > c), \\ &\sum_{i=b}^{c} 1, \quad \text{if } (b < a) \wedge (c \geq d), \\ &\sum_{i=b}^{d} 1, \quad \text{if } (b < a) \wedge (d > c). \end{aligned} \tag{8}$$

4. Evaluate the innermost sum. Ensure the correctness of the evaluation with respect to the upper and lower bounds (i.e., consider (2), or, if Bernoulli's formula is applied, make the appropriate transformations).

5. If there are more outcomes then start again from step 1.

6. Simplify the constraints. Outcomes depending on conditions which are never true can be eliminated.

7. If there are any Σ's — i.e., sums — left in the resulting expressions (outcomes), start again from step 1.

end

In order to illustrate the algorithm, we consider the evaluation of the sum

$$S = \sum_{i=l_1}^{u_1} \sum_{j=l_2+l_1-i}^{u_2} 1, \tag{9}$$

which corresponds to the number of times, S, the `(statements)` in the body of the following double loop in FORTRAN are executed:

```
      DO 10 I=L1,U1
         DO 10 J=L2+L1-I,U2
10          (statements)
```

Starting from the innermost sum, and, since, initially, there are no constraints, we proceed to step 4 of the algorithm; thus, evaluating the sum, we have:

$$S = \begin{cases} \sum_{i=l_1}^{u_1} (u_2 - (l_2 + l_1 - i) + 1), & \text{if } l_2 + l_1 - i \leq u_2 \Longleftrightarrow l_2 + l_1 - u_2 \leq i, \\ 0, & \text{otherwise.} \end{cases}$$

Before evaluating the remaining sum, the condition which has already been stated must be considered. Since it defines a lower bound for i, it must be compared with the lower bound of the sum having i as its index. Thus:

$$S = \begin{cases} \sum_{i=l_1}^{u_1} (i + u_2 - l_2 - l_1 + 1), & \text{if } l_1 \geq l_2 + l_1 - u_2 \Longleftrightarrow u_2 \geq l_2, \\ \sum_{i=l_2+l_1-u_2}^{u_1} (i + u_2 - l_2 - l_1 + 1), & \text{if } l_1 < l_2 + l_1 - u_2 \Longleftrightarrow u_2 < l_2, \\ 0, & \text{otherwise.} \end{cases}$$

The next step, before evaluating each sum, is to ensure that the upper bound is greater than the lower bound. We have:

$$S = \begin{cases} \sum_{i=l_1}^{u_1} (i + u_2 - l_2 - l_1 + 1), & \text{if } (u_2 \geq l_2) \wedge (l_1 \leq u_1), \\ \sum_{i=l_2+l_1-u_2}^{u_1} (i + u_2 - l_2 - l_1 + 1), & \text{if } (u_2 < l_2) \wedge (l_2 + l_1 - u_2 \leq u_1), \\ 0, & \text{otherwise.} \end{cases}$$

Since the polynomials in both summations involve unknown lower and upper bounds, the rules described in Equations (5) through (7) must be applied. After evaluation, the following five outcomes are obtained:

- If $((l_1 \geq 0) \vee (l_1 < 0 \wedge u_1 > 0)) \wedge (u_2 \geq l_2) \wedge (l_1 \leq u_1)$, then
$$S = (1 - l_1 + u_1)(2 - l_1 - 2l_2 + u_1 + 2u_2)/2.$$
- If $(l_1 < 0) \wedge (u_1 \leq 0) \wedge (u_2 \geq l_2) \wedge (l_1 \leq u_1)$, then
$$S = (l_1 + u_1 - l_1^2 - u_1^2)/2 + (u_1 - l_1 + 1)(u_2 - l_2 - l_1 + 1).$$
- If $((l_2 + l_1 - u_2 \geq 0) \vee (l_2 + l_1 - u_2 < 0 \wedge u_1 > 0)) \wedge (u_2 < l_2) \wedge (l_2 + l_1 - u_2 \leq u_1)$, then
$$S = (2 - 3l_1 + l_1^2 - 3l_2 + 2l_1l_2 + l_2^2 + 3u_1 - 2l_1u_1 - 2l_2u_1 + u_1^2 + 3u_2 - 2l_1u_2 - 2l_2u_2 + 2u_1u_2 + u_2^2)/2.$$
- If $(l_2 + l_1 - u_2 < 0) \wedge (u_1 \leq 0) \wedge (u_2 < l_2) \wedge (l_2 + l_1 - u_2 \leq u_1)$, then
$$S = (2 - 3l_1 + l_1^2 - 3l_2 + 2l_1l_2 + l_2^2 + 3u_1 - 2l_1u_1 - 2l_2u_1 - u_1^2 + 3u_2 - 2l_1u_2 - 2l_2u_2 + 2u_1u_2 + u_2^2)/2.$$
- If none of the above conditions is satisfied, then $S = 0$.

For the above sum, the `SymbolicSum` package in Mathematica returns only the first expression. The same also expression is derived by the techniques used in [4], who consider the sum in (9) for $l_1 = 2$, $u_1 = 5, u_2 = 5$; the error which may result is observed and discussed.

CONCLUSION

An algorithm for evaluating symbolically sums where some of the bounds are unknown variables has been presented. The need for such an algorithm has been motivated by problems arising in the analysis of parallel loops by compilers. In this context, previous work attempted to provide solutions using rather expensive frameworks based on Presburger formulas [7] or Ehrhart polynomials [3]. Other related work has concentrated on an algorithm for polytope splitting [10], while, in [6], an algebra of conditional values is defined, which, however, returns complex expressions.

REFERENCES

[1] U. Banerjee, *Loop Transformations for Restructuring Compilers: The Foundations*, Kluwer Academic Publishers (1993).

[2] A. I. Barvinok, *Computing the Volume, Counting Integral Points, and Exponential Sums*, Discrete & Computational Geometry, 10(2), pp. 123–141 (1993).

[3] P. Clauss, *Counting Solutions to Linear and Nonlinear Constraints through Ehrhart polynomials: Applications to Analyze and Transform Scientific Programs*, Proceedings of the 1996 International Conference on Supercomputing, ACM Press, pp. 278–285 (1996).

[4] M. Cosnard, M. Loi, *Automatic Task Graph Generation Techniques*, Proceedings of the 28th Annual Hawaii International Conference on System Sciences (Vol. II, Software Technology), IEEE Computer Society Press, pp. 113–122 (1995).

[5] R. L. Graham, D. E. Knuth, O. Patashnik, *Concrete Mathematics*, Addison-Wesley (1991).

[6] M. R. Haghighat, C. D. Polychronopoulos, *Symbolic Analysis: A Basis for Parallelization, Optimization, and Scheduling of Programs*, in U. Banerjee, D. Gelernter, A. Nicolau, D. Padua (Eds.), Languages and Compilers for Parallel Computing (6th International Workshop, Aug. 1993), Lecture Notes in Computer Science 768, Springer-Verlag, pp. 567–585 (1994).

[7] W. Pugh, *Counting Solutions to Presburger Formulas: How and Why*, Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation, ACM SIGPLAN Notices, 29(6), pp. 121–134 (1994).

[8] R. Sakellariou, *On the Quest for Perfect Load Balance in Loop-Based Parallel Computations*, PhD Thesis, Department of Computer Science, University of Manchester (1996).

[9] R. Sakellariou, J. R. Gurd, *Compile-Time Minimisation of Load Imbalance in Loop Nests*, Proceedings of the 1997 International Conference on Supercomputing, ACM Press (1997).

[10] N. Tawbi, *Estimation of Nested Loops execution time by Integer Arithmetic in Convex Polyhedra*, Proceedings of the 8th International Parallel Processing Symposium, IEEE Computer Society Press, pp. 217–221 (1994).

[11] M. Wolfe, *High Performance Compilers for Parallel Computing*, Addison-Wesley (1996).

[12] H. Zima, B. Chapman, *Supercompilers for Parallel and Vector Computers*, ACM Press & Addison-Wesley (1990).