Toward Symbolic Performance Prediction of Parallel Programs

Thomas Fahringer
Institute for Software Technology and Parallel Systems
University of Vienna
Liechtensteinstrasse 22, A-1090 Vienna, Austria
e-mail: tf@par.univie.ac.at

Abstract

Critical analyses in performance estimators for parallel programs require an algorithm that count the number of integer solutions to a set of inequalities. Most current performance estimators are restricted to linear inequalities for this analysis. In this paper we describe a symbolic algorithm which can estimate the number of integer solutions to a set of both linear and non-linear inequalities. The result is either an integer value or a symbolic expression depending on whether the inequalities contain non-loop variables. We have implemented this algorithm and use it as part of P³T, a performance estimator for data parallel programs. We demonstrate the usefulness of this algorithm by predicting the work load of all processors for a parallel program and compare it to measurements taken on an iPSC/860 hypercube system.

1. Introduction

Counting the number of integer solutions to a set of inequalities has been shown to be a key issue in performance analysis of parallel programs. Numerous applications [2, 7] include: estimating the number of statement execution counts, branching probabilities, number of data transfers and cache misses. Even conventional compiler analysis can be supported, for instance, by detecting loops that never iterate (zero-trip loops) and consequently can be eliminated.

In what follows, we present a non-linear loop nest for which we compute the iteration count in order to motivate our research.

\[
\begin{align*}
\text{DO } & i_1 = 1, n_1 \\
\text{DO } & i_2 = 1, n_2 \\
& \text{if } i_1 \leq n_1 \text{ statement } S \\
\text{ENDDO} \\
\text{ENDDO}
\end{align*}
\]

\(i_1\) and \(i_2\) are (loop) variables and \(n_1, n_2\) are parameters (loop invariants). Computing how many times statement \(S\) is executed in the loop nest is equivalent to counting the number of integer solutions of \(I\), a set of inequalities:

\(\{1 \leq i_1 \leq n_1, 1 \leq i_2 \leq n_2 \cdot i_1, i_1 \leq n_2\}\). Note that we consider \(i_2 \leq n_2 \cdot i_1\) to be non-linear, although \(n_2\) is loop invariant. The statement execution count for \(S\) is given by:

\[
\sum_{i_2=1}^{n_2} \sum_{i_2=1}^{n_1} 1 = \left\{ \begin{array}{ll}
\frac{n_1 n_2}{2} + \frac{n_1 + n_2}{2} : & \text{if } 1 \leq n_1 \leq n_2 \\
\frac{n_2}{2} : & \text{if } 1 \leq n_2 < n_1
\end{array} \right.
\]

In general, every loop implies at least two constraints on its loop variable, one for its upper and one for its lower bound. Additional constraints on both parameters and variables can be implied, for instance, by conditional statements, or min and max functions.

Algorithms [2] that count the number of integer solutions of arbitrary large sets of linear inequalities over loop variables have already been implemented and successfully used in the context of performance analysis and parallelizing compilers. The drawback with this analysis, however, is that program unknowns (parameters) must be replaced by constant integers. In order to overcome this disadvantage M. Haghighe and C. Polychronopoulos [5] described an algebra of conditional values and a set of rules for transforming symbolic expressions. However, they did not present an algorithm that decides which rule to apply when. N. Tawbi [7] developed a symbolic sum algorithm which handles loop nests with parameters. Her approach is restricted to sums based on linear inequalities implied by loop header statements only. W. Pugh [6] improved Tawbi’s algorithm by extending it to techniques that count the number of integer solutions to an arbitrary set of linear inequalities.

So far non-linear inequalities have not been considered for counting integer solutions. This paper describes a symbolic sum algorithm that extends previous work by estimating the number of integer solutions to a class of non-linear inequalities. We implemented the algorithm and use it as part of a performance estimator. We present an experiment which demonstrates how to apply the symbolic sum algorithm to estimate the amount of work to be processed by the processors of a parallel program which is a key figure to evaluate the corresponding work distribution.

This paper is organized as follows: In Section 2, we describe the symbolic sum algorithm in detail and apply it to a
performance prediction application. Related work is presented throughout the paper. Conclusions and final remarks are given in Section 3.

2. Symbolic Sum Computation

In this section we describe a symbolic algorithm which computes the number of integer solutions of a set of linear and non-linear inequalities. The problem we want to solve is the following: Let $\mathcal{V}$ be a set of variables and $\mathcal{P}$ a set of parameters. All variables and parameters are integers. $\mathcal{I}$ is a set of linear and non-linear inequalities over $\mathcal{V} \cup \mathcal{P}$. $I \in \mathcal{I}$ is restricted to be of the following form:

\[ p_1(\vec{P}) \ast v_1 + \ldots + p_k(\vec{P}) \ast v_k \ R E L \ 0 \]  

(1)

where $REL \in \{\leq, \geq, <, >, =, \neq\}$ represents an equality or inequality relationship. $\vec{P}$ is a vector over variables of $\mathcal{P}$. $p_i(\vec{P})$ are linear or non-linear expressions over $\mathcal{P}$, whose operations can be addition, subtraction, multiplication, division, floor, ceiling, and exponentiation. Min and max functions are substituted where possible by constraints free of min and max functions which is shown in Section 2.3. $v_i \in \mathcal{V} \ (1 \leq i \leq k)$ and not all variables in $\{v_1, \ldots, v_k\}$ are necessarily pairwise different. Then

\[ \mathcal{W} = \{ \vec{V} | \vec{V} \text{satisfies } \mathcal{I} \} \]  

(2)

is the set of integer solutions over all variables in $\mathcal{V}$ which satisfy the conjunction of inequalities in $\mathcal{I}$. Every $\vec{V}$ is a vector over variables in $\mathcal{V}$, which represents a specific solution to $\mathcal{I}$. Figure 1 shows the algorithm for counting the number of elements in $\mathcal{W}$, given $\mathcal{I}, \mathcal{P}, \mathcal{V}, \mathcal{E}$, and $\mathcal{R}$. $E$ is an intermediate result for a specific solution $S_i$ as described below. The result $\mathcal{R}$ is a set of $k$ tuples $(C_i, S_i)$ where $1 \leq i \leq k$. Each tuple corresponds to a conditional solution of the sum algorithm. Note that the conditions $C$ (satisfying (1)) among all solution tuples are not necessarily disjoint. The result has to be interpreted as the sum over all $S_i$ under the condition of $C_i$ as follows:

\[ \sum_{1 \leq i \leq k} \gamma(\neg C_i) \ast S_i \]  

(3)

where $\gamma$ is defined as

\[ \gamma(C) = \begin{cases} 1 & \text{if } C = \text{TRUE} \\ 0 & \text{otherwise} \end{cases} \]  

(4)

$E$ and $\mathcal{R}$ must be respectively set to 1 and $\phi$ (empty set) at the initial call of the algorithm.

2.1. The Algorithm

In each recursion the algorithm is eliminating one variable $v \in \mathcal{V}$. First, all lower and upper bounds of $v$ in $\mathcal{I}$ are determined. Then the maximum lower and minimum upper bound of $v$ is searched by splitting $\mathcal{I}$ into disjoint subsets of inequalities. For each such subset $I_i$ the algebraic sum (see Section 2.2) of the current $E$ over $v$ is computed. Then the sum algorithm is recursively called for $I_i$, the newly computed $E$, $\mathcal{V} = \{v\}$, $\mathcal{P}$, and $\mathcal{R}$. Eventually at the deepest recursion level, $\mathcal{V}$ is empty, then $E$ and its associated $I$ represent one solution tuple defined solely over parameters and constants.

In statement S1 we simplify $\mathcal{I}$, which includes detection and elimination of tautologies, contradictions, equalities and redundant inequalities. Our simplification techniques and algorithms are described in detail in [3]. Statement S2 returns the empty solution in case that $\mathcal{I}$ contains a contradiction. In statement S3 the algorithm checks for an empty set of variables. If so, then there are no variables remaining for being eliminated, the algorithm adds the current solution tuple $(I, E)$ into $\mathcal{R}$ and returns.

Statement S4 describes the core of the algorithm, which addresses several issues:

- In statement S4.1 a variable is chosen which is being eliminated in this recursion. A heuristic is used which selects the variable appearing in as few inequalities of $\mathcal{I}$ as possible. Note that an inequality which contains more than one variable is a (lower or upper) bound for all of these variables.
- A variable $v \in \mathcal{V}$ may have several lower and upper bounds in $\mathcal{I}$. In order to compute the sum of all integer solutions for $v$ over all inequalities in $\mathcal{I}$, we need to know the maximum lower bound and minimum upper bound of $v$. In general we may not know them, therefore, we have to split $\mathcal{I}$ into disjoint subsets of inequalities. For each such subset $I_{i,j}$ it is assumed that $l_i$ and $u_j$ are the associated maximum lower and minimum upper bounds for $v$. This is shown in statement S4.3. Clearly, in this step we may create sets of inequalities ($I_{i,j}$) containing contradictions, which are tried to be eliminated in S1 by using our simplifier [3]. Choosing $v$ according to the previous heuristic implies a minimum number of different $I_{i,j}$ created at this step.
- For all of the previously created subsets $I_{i,j}$ we compute the algebraic sum (see Section 2.2) based on the current $E$ for $v$ bounded by $l_i$ and $u_j$.
- The algorithm is recursively called with $I_{i,j}$, $\mathcal{V} = \{v\}$, $\mathcal{P}$, the current $E$, and $\mathcal{R}$.

We further extended the sum algorithm of Figure 1 to a larger class of inequalities that goes beyond $\mathcal{I}$ as described in (1). In the following we assume that $\mathcal{I}$ represents a set of inequalities that satisfy the conditions mentioned in (1). Let $\mathcal{I}'$ be a set of inequalities over $\mathcal{V} \cup \mathcal{P}$ which is defined by

\[ p_1(\vec{P}) f_1(\vec{V}_1) + \ldots + p_k(\vec{P}) f_k(\vec{V}_k) \ R E L \ 0 \]  

(5)

$p_i(\vec{P})$ and $\mathcal{R}$ have the same meaning as described by (1). $\vec{V}_i$ describes a vector of variables over $\mathcal{V}_i$ (subset of variables in $\mathcal{V}$) with $1 \leq i, j \leq k$. $f_i(\vec{V}_i)$ are linear or non-linear expressions defined over $\mathcal{V}$ of the following form (simplified down to its simplest form):
where all variables in \( \{q_1, \ldots, q_l, h_1, \ldots, h_l \} \subseteq \mathcal{V} \) must be pairwise different after simplification. Furthermore, any variable \( v \in \mathcal{V} \) may appear in several \( f_i \) with the restriction that \( v \) must always appear either in the denominator or in the numerator of all \( f_i \) in which it appears. For instance, \( v \) may appear in the numerator of all \( f_i \). However, \( v \) cannot appear in the numerator of \( f_i \) and in the denominator of \( f_j \) with \( i \neq j \).

\[
\sum_{i=1}^{l} \frac{q_i \cdot \ldots \cdot q_l}{h_1 \cdot \ldots \cdot h_l}
\]

**SUM(\mathcal{I}, \mathcal{V}, \mathcal{P}, \mathcal{E}; \mathcal{R})**

- **INPUT:**
  - \( \mathcal{I} \): set of linear and non-linear inequalities over \( \mathcal{V} \cup \mathcal{P} \)
  - \( \mathcal{V} \): set of variables
  - \( \mathcal{P} \): set of parameters
  - \( \mathcal{E} \): linear and non-linear symbolic expression over \( \mathcal{V} \cup \mathcal{P} \)

- **INPUT-OUTPUT:**
  - \( \mathcal{R} \): set of solution tuples \((\mathcal{C}_i, \mathcal{S}_i)\) where \( 1 \leq i \leq k \). \( \mathcal{C}_i \) is a set of linear or non-linear inequalities over \( \mathcal{P} \). \( \mathcal{S}_i \) is a linear or non-linear polynomial over \( \mathcal{P} \) such that for all pairs of \( \mathcal{C}_i, \mathcal{C}_j (i \neq j \) and \( 1 \leq i, j \leq k \) the following holds: \( \mathcal{C}_i \cup \mathcal{C}_j \) not necessarily disjoint.

- **ALGORITHM:**

  **S1:** Simplify \( \mathcal{I} \)

  **S2:** if \( \mathcal{I} \) is inconsistent (no solution) then return

  **endif**

  **S3:** if \( \mathcal{V} \equiv \emptyset \) then

  \( \mathcal{R} := \mathcal{R} \cup \{ \mathcal{I}, \mathcal{E} \} \)

  return

  **endif**

  **S4:** Split loop

  **S4.1:** Choose variable \( v \in \mathcal{V} \) for being eliminated

  **S4.2:** \( \mathcal{L} := \) subset of \( \mathcal{I} \) not involving \( v \)

  \( \mathcal{U} := \) set of upper bounds of \( v \) in \( \mathcal{I} \)

  \( \mathcal{L} \cup \{ i_t \leq i_{t-1} \leq i_t \leq i_{t+1} \leq i_{t-1} \} \)

  \( \mathcal{L} \cup \{ i_t \leq i_{t-1} \leq i_t \leq i_{t+1} \leq i_{t-1} \} \)

  **for each** \( (i_t, i_{t-1}) \in \mathcal{L} \times \mathcal{U} \) do

  \( E_{i_{t-1}, i_t} := \sum_{i=1}^{l} E \)

  **SUM(\mathcal{L}, \mathcal{V} \setminus \{v\}, \mathcal{P}, E_{i_{t-1}, i_t}, \mathcal{R}) \)

  **endfor**

  **S5:** return

**Figure 1. Symbolic Sum Algorithm**

Clearly, \( \mathcal{I}' \supseteq \mathcal{I} \). Unfortunately, in general the algorithm cannot handle inequalities of the form described by (5), as for certain \( \mathcal{I}' \) the algorithm may create a new set of inequalities in statement S4.3 that does not honor the constraints of (5). We have not yet found a way to narrow down the constraints on \( \mathcal{I}' \), such that the algorithm never creates an inequality out of it, which does not satisfy (5). Nevertheless, we extended the symbolic sum algorithm to handle a larger class of inequalities as described in (1) by simply checking at the beginning of the algorithm whether \( \mathcal{I} \) satisfies (5). If yes, then the algorithm proceeds, otherwise, the algorithm returns and indicates to the user that the original \( \mathcal{I} \) cannot be processed for the reasons mentioned above. For instance, this will enable the algorithm to count the integer solutions to the following set of inequalities which is not covered by (1):

\[
1 \leq i_1 \leq n, \quad 1 \leq i_1 * i_2 \leq m * i_1, \quad n * m \leq i_1 * i_2
\]

### 2.2. Algebraic Sum

In the following we describe the computation of the algebraic sum \( \sum_{v=1}^{u} \mathcal{E} \) over a variable \( v \in \mathcal{V} \), where \( i \) and \( u \) have been extracted (S4.2 in Figure 1) from inequalities of \( \mathcal{I} \). \( \mathcal{E} \) is a symbolic expression over \( \mathcal{V} \cup \mathcal{P} \), which has been simplified down to its simplest form using our own symbolic manipulation package [3]. Symbolic expressions denote both linear and non-linear symbolic expressions, if not mentioned otherwise. Computing the algebraic sum is then reduced to solving:

\[
\sum_{v=1}^{u} \mathcal{E} = \mathcal{E}_0 + \mathcal{E}_1 \cdot v^{\pm 1} + \mathcal{E}_2 \cdot v^{\pm 2} + \ldots + \mathcal{E}_q \cdot v^{\pm q}
\]

\( \mathcal{E}_k \) is a linear or non-linear expression over \( \mathcal{P} \cup \mathcal{V} \setminus \{v\} \). \( \pm q \) (q is an integer constant) means that the sign of \( q \) can either be positive or negative. The problem, therefore, amounts to computing the following: \( \sum_{v=1}^{u} v^r \). For the case where \( r \geq 0 \) we can use standard formulas for sums of powers of integers. They are described in [4] and reviewed in [7]. For \( r < 0 \) there are no closed forms known. However, for \( r = -1 \) and \( 2 \leq n \) it has been shown [4] that \( \sum_{v=1}^{u} v^{-1} \) (\( \ln(n) \) is the natural logarithm)

\[
\ln(n) < \sum_{v=1}^{u} \frac{1}{v} < \ln(n) + 1
\]

and also [1] that \( \sum_{v=1}^{\infty} \frac{1}{v} = \ln(\infty) = 1.65 \). Based on this we can prove that for all \( i \geq 2 \) the following holds: \( \sum_{v=1}^{u} v^{-i} \geq \sum_{v=1}^{u} \frac{1}{v^2} \), and finally

\[
1 < \sum_{v=1}^{\infty} \frac{1}{v^2} < \frac{\pi^2}{6} \quad \text{if } i \geq 2 \text{ and } i \text{ is even}
\]

\[
-\frac{\pi^2}{6} < \sum_{v=1}^{\infty} \frac{1}{v^2} < -1 \quad \text{if } i > 2 \text{ and } i \text{ is odd}
\]

For our algebraic sum computation we can approximate all cases in (6) by the specified upper or lower bound or an approximate expression (average between upper and lower bound).

The algorithm as shown in Figure 1 is an extension to what has been sketched by W. Pugh as the convex sum for linear inequalities in [6]. Pugh only superficially outlined
his techniques and has not implemented his algorithm according to [6], while we have presented a precise definition for the symbolic sum algorithm, fully implemented it, and apply it to performance prediction applications under P3T [2], a state-of-the-art performance estimator for parallel programs. Pugh did not mention the detection and elimination of contradictions, which is critical for the sum algorithm in order to reduce its complexity. It is not clear from his version of choosing a variable for being eliminated (see Figure 1 S4.1), whether an inequality containing more than 1 variable is a bound for all variables or only for one of them. This is important for the heuristic described in statement S4.1. He also does not mention in general how to handle min/max functions as loop bounds and “not equal” inequalities. The significant differences between our work and his are the following:

- Pugh described his algorithm only for a set of linear inequalities, while we extended this algorithm to cover also non-linear inequalities.
- Pugh’s techniques are based on simplifications for linear inequalities only, whereas we built a powerful simplifier handling also non-linear inequalities. Due to space constraints we could not describe our simplifier in this paper. A detailed description is given in [3].
- Pugh’s algebraic sum algorithm is restricted to variables handles negative exponents.

2.3. Miscellaneous

In this section we briefly describe how we handle equalities, $\neq$ inequalities, min/max functions, and fractional expressions.

- Equalities can be implied by conditional statements in a program. Our simplification algorithm may also detect two inequalities to represent an equality relationship (e.g. $i \neq j \leq n$ and $i \neq j \geq n \iff \exists i, j = n$). The algorithm is solving an equality for a variable with a simple coefficient and substitutes it for all occurrences in $I$ and $E$. Therefore, the equality can be deleted.

- If $I$ contains a “$\neq$” inequality $l$ such as e.g. $i \neq j \neq n$, then we split $I$ into two disjoint subsets $I_1$ and $I_2$ such that $I_1 := I \cup \{i \neq j \gt n\}$ and $I_2 := I \cup \{i \neq j \lt n\}$. The algorithm is then called for both subsets of $I$ and the combined result is equivalent to the original problem including the “$\neq$” inequality.

- In many cases we can replace min and max functions appearing in inequalities. E.g let $l$ be $\leq \min(j, k)$, then we replace $l$ by the conjunction of two inequalities $i \leq j$ and $i \leq k$ in the underlying $I$. If, however, $l$ is $\leq \max(j, k)$, then we have to split $I$ into two disjoint subsets $I_1$ and $I_2$ such that $I_1 := I \cup \{i \leq j, j \geq k\}$ and $I_2 := I \cup \{i \leq k, k \geq j\}$. The algorithm is then called for both subsets of $I$ and the combined result is equivalent to the original problem including the min. max functions are handled similarly.

- We have implemented several methods [3] to handle fractional expressions (floor and ceiling functions) in inequalities and general symbolic expressions. For instance, fractional expressions can be replaced by their corresponding lower or upper bounds based on the following observation: $\frac{n}{2} - 1 < \frac{n}{2} \leq \frac{n}{2} \leq \frac{n}{2} < \frac{n}{2} + 1$

2.4. Example

In what follows we present an experiment that shows how the symbolic sum algorithm can be used to estimate the amount of work to be processed by every specific processor of a data parallel program. The following code shows a High Performance Fortran - HPF code excerpt with a processor array $PR$ of size $P$.

```fortran
INTEGER A(n1)
$\text{!HPF$ PROCESSORS :: PR(P)$!}$
$\text{!HPF$ DISTRIBUTE (BLOCK) ONTO PR :: A$!}$
DO 10 i1=1,n1
DO 10 i2=1,i1 + n1
if ( i2 \leq n2 ) A(i2) = ... 
10 CONTINUE
```

The loop contains a write operation to a one-dimensional array $A$ which is block-distributed onto $P$ processors. Let $k (1 \leq k \leq P)$ denote a specific processor of the processor array. Computations that define the data elements owned by a processor $k$ are performed exclusively by $k$. For the sake of simplicity we assume that $P$ evenly divides $n_2$. Therefore, a processor $k$ is executing the assignment to $A$ based on the underlying block distribution if $\frac{n_2 + k - 1}{P} + 1 \leq i_2 \leq \frac{n_2 + k}{P}$. The precise work to be processed by a processor $k$ is the number of times $k$ is writing $A$, which is defined by $work(k)$.

<table>
<thead>
<tr>
<th>$n_1$</th>
<th>$n_2$</th>
<th>$\sum_{k=1}^{P} w \times k \times k$</th>
<th>$\sum_{k=1}^{P} \frac{w \times k \times k}{P}$</th>
<th>error in %</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>100</td>
<td>1000000</td>
<td>10025</td>
<td>0.25</td>
</tr>
<tr>
<td>200</td>
<td>100</td>
<td>200000</td>
<td>20238</td>
<td>1.19</td>
</tr>
<tr>
<td>200</td>
<td>200</td>
<td>400000</td>
<td>40450</td>
<td>1.12</td>
</tr>
<tr>
<td>400</td>
<td>200</td>
<td>800000</td>
<td>80474</td>
<td>0.59</td>
</tr>
<tr>
<td>400</td>
<td>400</td>
<td>1600000</td>
<td>160900</td>
<td>0.56</td>
</tr>
<tr>
<td>800</td>
<td>400</td>
<td>3200000</td>
<td>320950</td>
<td>0.29</td>
</tr>
<tr>
<td>800</td>
<td>800</td>
<td>6400000</td>
<td>641800</td>
<td>0.28</td>
</tr>
</tbody>
</table>

| $P=4$ |

Table 1. Measured versus estimated values for the amount of work to be done by all processors for $P = 4$.

The problem to estimate the amount of work to be done by processor $k$ can now be formulated as counting the number of integer solutions to $I$ which is given by:
1 \leq i_1 \leq n_1 \\
1 \leq i_2 \leq i_1 \cdot n_1 \\
i_2 \leq n_2 \\
\frac{n_2 \cdot (k-1)}{p} + 1 \leq i_2 \leq \frac{n_2 \cdot k}{p}

In the following we substitute \frac{n_2 \cdot (k-1)}{p} + 1 by \( i_b \) and \frac{n_2 \cdot k}{p} by \( i_k \). By using our simplification algorithm [3] we can determine that \( 1 \leq i_2 \) is made redundant by \( i_b \leq i_2 \) and \( i_2 \leq n_2 \) by \( i_k \leq i_2 \). Therefore, the simplified \( I \) with all redundant inequalities removed is given by

\begin{align*}
1 & \leq i_1 \leq n_1 \\
i_b & \leq i_2 \leq i_k \\
i_2 & \leq i_1 \cdot n_1
\end{align*}

In the first recursion we can either choose variable \( i_1 \) or \( i_2 \) for being eliminated according to the heuristic of S4.1. Both variables appear in 3 inequalities of the simplified \( I \). By selecting \( i_2 \) we obtain two upper bounds for \( i_2 \) which are given by \([n_1 \cdot n_1, i_k] \). Based on S4.3, we split the inequalities of \( I \) into \( i_1 \) and \( i_2 \).

- \( I_1 := \{ 1 \leq i_1 \leq n_1, i_1 \cdot n_1 \leq i_b, i_b \leq i_1 \cdot n_1 \} \) with \( \sum_{i_2 = i_b}^{i_1 \cdot n_1} 1 = i_1 \cdot n_1 - i_b \).

In S1 of the algorithm (see Figure 1) we detect that \( 1 \leq i_1 \) is made redundant by \( \frac{i_b}{n_1} \leq i_1 \). We now eliminate \( i_1 \) based on two upper bounds \([\frac{n_2}{n_1}, i_k] \) for this variable. This yields two solutions \( C_1 \) and \( C_2 \):

\[
C_1 := \{ \frac{n_2}{n_1} \leq n_1, i_b \leq i_k \} = \{ i_b \leq n_1, P \leq n_2 \} \quad \text{with} \quad S_1(k) := \sum_{i_1 = 1}^{\frac{n_1}{n_1}} i_1 \cdot n_1 - i_b = \left( \frac{n_1 + i_k - i_b}{n_1} \right) \left( \frac{k \cdot 2 \cdot n_1 + 2 \cdot i_b + i_k \cdot n_1 + i_b}{2 \cdot n_1} \right).
\]

\[
C_2 := \{ \frac{n_2}{n_1} > n_1, i_b \leq \frac{n_1}{n_1} \} \quad \text{with} \quad S_2(k) := \sum_{i_1 = 1}^{\frac{n_1}{n_1}} i_1 \cdot n_1 - i_b = \left( n_1 - \frac{i_b}{n_1} + 1 \right) \cdot \left( \frac{n_2}{n_1} - \frac{i_b}{n_1} + 1 \right).
\]

- \( I_2 := \{ 1 \leq i_1 \leq n_1, i_1 \cdot n_1 > i_k, i_b \leq i_k \} = \{ 1 \leq i_1 \leq n_1, i_1 \geq \frac{n_2}{n_1} + 1, i_b \leq i_k \} \) with \( \sum_{i_2 = i_b}^{i_1 \cdot n_1} 1 = \frac{n_1}{n_2} \).

The algorithm detects in \( S \) that \( 1 \leq i_1 \) is made redundant by \( \frac{u_k + 1}{n_1} \leq i_1 \). Therefore, we can eliminate \( i_1 \) for \( C_3 := \{ n_2 \geq P, n_1^2 \geq i_k + 1 \} \) which yields

\[
S_3(k) := \sum_{i_1 = \frac{n_2}{n_1}}^{\frac{n_1}{n_2}} i_1 \cdot n_1 - \frac{u_k + 1}{n_1} = \frac{n_2}{n_2} \left( n_1 - \frac{u_k + 1}{n_1} + 1 \right).
\]

Therefore, statement \( S \) in the example code is approximately executed \( \text{work}(k) := \sum_{1 \leq i_1 \leq n_1} \gamma(C_1) \cdot S_i(k) \) times by a specific processor \( k \) \((1 \leq k \leq P)\) for the parameters \( n_1, n_2 \) and \( P \). \( \gamma(C_i) \) is defined by (4). Table 1 shows an experiment were we compare measured against estimated values for a 4 (= \( P \)) processor version of the example code by varying the values for \( n_1 \) and \( n_2 \). The measurements have been done by executing the example code on an iPSC/860 hypercube system and enumerating for each processor the number of times it executes statement \( S \). It can be seen that the estimates (\( \text{work}(k) \)) are very close to the measurements (\( \text{work}(k) \)). In the worst case the estimates are off by 1.19 % for a relative small problem size (\( n_1 = 200, n_2 = 100 \)). Moreover, we also observe that for larger problem sizes the estimation accuracy consistently improves.

3 Conclusions

Symbolic analysis is important for performance prediction of parallel programs as well as for compiler analysis. We have described and implemented a symbolic sum algorithm that estimates the number of integer solutions to a set of linear and non-linear inequalities defined over program unknowns and loop variables. Preliminary experiments under a performance estimator indicate very reasonable estimation accuracy.

A key component for the symbolic sum algorithm is to reduce and simplify the set of inequalities of the given problem by eliminating tautologies, contradictions, equalities and redundant inequalities. Due to space constraints we could not include our research on simplifying constraints in this paper. A detailed description of both the sum algorithm as well as the simplifier can be found in an extended version of this paper at: http://www.par.univie.ac.at.

4 Acknowledgments

I’d like to thank Dr. Hoon Hong from RISC Linz for very helpful discussions on the symbolic sum algorithm at the beginning of this research, and Alex Celic for helping me to implement the algorithms described in this paper.

References


