Automatic Complexity Analysis of Explicitly Parallel Programs

Torsten Hoefler
ETH Zurich
Zurich, Switzerland
htor@inf.ethz.ch

Grzegorz Kwasniewski
ETH Zurich
Zurich, Switzerland
grzegorz.kwasniewski@inf.ethz.ch

ABSTRACT

The doubling of cores every two years requires programmers to expose maximum parallelism. Applications that are developed on today's machines will often be required to run on many more cores. Thus, it is necessary to understand how much parallelism codes can expose. The work and depth model provides a convenient mental framework to assess the required work and the maximum parallelism of algorithms and their parallel efficiency. We propose an automatic analysis to extract work and depth from a source-code. We do this by statically counting the number of loop iterations depending on the set of input parameters. The resulting expression can be used to assess work and depth with regards to the program inputs. Our method supports the large class of practically relevant loops with affine update functions and generates additional parameters for other expressions. We demonstrate how this method can be used to determine work and depth of several real-world applications. Our technique enables us to prove if the theoretically maximum parallelism is exposed in a practical implementation of a problem. This will be most important for future-proof software development.

Categories and Subject Descriptors

F.2 [**Analysis of Algorithms and Problem Complexity**]: general

General Terms

Theory

Keywords

Loop iterations; Polyhedral model; Work and depth analysis

1. INTRODUCTION

Parallelism in today's computers is still growing exponentially, currently doubling approximately every two years.

SPAA'14, June 23–25, 2014, Prague, Czech Republic.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2821-0/14/06 ...$15.00.
http://dx.doi.org/10.1145/2612669.2612685.

This implies that programmers need to expose exponentially growing parallelism to exploit the full potential of the architecture. Parallel programming is generally hard and practical implementations may not always expose enough parallelism to be considered future-proof. This is exaggerated by continuous application development and the fact that applications are developed on systems with significantly lower core counts than their production environment. Thus, it is increasingly important that programmers understand bounds on the scalability of their implementation.

Parallel codes are manifold and numerous programming frameworks exist to implement parallel versions of sequential codes. We define the class of *explicitly parallel* codes as applications that statically divide their workload into several pieces which are processed in parallel. Explicitly parallel codes are the the most prevalent programming style in large-scale parallelism using the Pthreads, OpenMP, the Message Passing Interface (MPI), Partitioned Global Address Space (PGAS), or Compute Unified Device Architecture (CUDA) APIs. Many high-level parallel frameworks (e.g., [18]) and domain-specific languages (e.g., [16]) compile to such explicitly parallel languages.

The work and depth model is a simple and effective model for parallel computation. It models computations as vertices and data dependencies as edges of a directed acyclic graph (DAG). The total number of vertices in the graph is the *total work* W and the length of a longest path is called the *depth* D (sometimes also called span). We will now describe more properties of the model and possible analyses.

1.1 Work and depth and parallel efficiency

In practice, analyses are often not used to predict exact running times of an implementation on a particular architecture. Instead, they often determine how the running time behaves with regards to the input size. The work and depth model links sequential running time and parallelism elegantly. The work W is proportional to the time T_1 required to compute the problem on a single core. The depth D is the longest sequential chain and thus proportional to a lower bound to the time T_∞ required to compute the problem with an infinite number of cores.

Work and depth models are often used to develop parallel algorithms (e.g., [33]) or to describe their properties (e.g., in textbooks [22, 23]). Those algorithms are then often adapted in practical settings. We propose to use the same model, somewhat in the inverse direction, to analyze existing applications for bounds on their scalability and available parallelism. Our results can also be used to prove an implementation asymptotically optimal with regards to its parallel efficiency if bounds on work and depth of the problem are known. In our analysis we use the assumption from [13] that all operations are performed in unit time and the time required for accessing data, storing results, etc., is ignored.

Brent's lemma [13] bounds running times on p cores with $\frac{W}{p} \leq T_p \leq \frac{W}{p} + D$. D measures the sequential parts of the calculation and is equivalent to time t needed to perform an operation with sufficient number of processes, W is equivalent to the number of operations q in Brent's notation and $B = \frac{D}{W}$ is a lower bound of the sequential fraction that limits the returns from adding more cores. Applying Amdahl's law [2] shows that the speedup is limited to $S_p = \frac{T_1}{T_p} \leq \frac{1}{B+\frac{1-B}{p}}$. If we consider the parallel efficiency $E_p = \frac{S_p}{p}$, then we can bound the maximally achievable efficiency using the work and depth model as $E_p = \frac{T_1}{pT_p} \leq \frac{1}{1+B(p-1)}$. We observe that for fixed B, $\lim_{P\to\infty} E_p = 0$ such that every fixed-size computation can only utilize a limited number of cores *efficiently*, i.e., $E_p \geq 1 - \epsilon$.

This observation allows us to define *available parallelism* and good scaling in terms of ϵ as the maximum number of processes p for which T_p may decrease. Bounds on work and depth for certain problems also allow us to differentiate between a problem that is hard to parallelize (e.g., depth first search (DFS)) and a suboptimal parallelization; we can also define the *distance* of a given parallel code to a *parallelism-optimal* solution.

Work and depth are typically functions of the input size. In structured programming [17], loops and recursion are the only techniques to increase the work depending on program input parameters. Here, we focus on loops only and we assume that each program can be abstracted as a set of loops that determine the number of executions for each statement. We model each statement as a work item that takes unit time. To simplify the explanation further, we also assume that there is only one statement in each loop (since all statements will have identical iteration counts). Now, the problem of determining the work is equivalent to determine the loop iteration counts for each statement. The depth is relative to a special parameter p that represents the number of processes. We now discuss a simple motivating example:

Example I: Parallel sorting skeleton.

Assume the following loop is executed by $\mathsf{p} > 0$ processes[1] (p equally divides n, n>0 and n is a power of 2):

```
for(x=0; x<n/p; x++)
  for(y=1; y<n; y*=2) S1;
```

All variables that are not changed in the loop but influence the iteration counts are called *parameters*. The parameter `n` represents size of the input problem. `S1` is an arbitrary computation statement that models one work item. We now analyze work and depth for this explicitly parallel loop.

For any loop, the elements that determine the number of iterations can be split into three classes:

1. Initial assignment: `x=0, y=1`
2. Loop guards: `x<n/p, y<n`
3. Loop updates: `x++, y*=2`

[1]We use typewriter font to denote source code variables

The number of iterations of statement `S1` in this loop (depending on the parameters `n` and `p`) can be counted as

$$N(\mathsf{n},\mathsf{p}) = T_p = \mathsf{n}/\mathsf{p} \cdot \log_2(\mathsf{n}).$$

From $N(\mathsf{n},\mathsf{p})$, we can determine that the total work and depth is

$$W(\mathsf{n}) = N(\mathsf{n},1) = T_1 = \mathsf{n} \cdot \log_2(\mathsf{n})$$

$$D(\mathsf{n}) = N(\mathsf{n},\infty) = T_\infty = \log_2(\mathsf{n}).$$

The parallelization is work-conserving and the parallel efficiency $E_p = 1$. If this loop implements parallel sorting, then our analysis shows that it is asymptotically optimal in work and depth [25], and thus exposes maximum parallelism. In this paper, we will show how to perform this analysis automatically.

Example II: Parallel reductions.

Our second example illustrates a common problem in parallel shared memory codes: reductions. Programmers often employ inefficient algorithms because efficient tree-based schemes are significantly harder to implement. A sequential reduction would be implemented as follows (addition operations on the variable `sum` are performed atomically):

```
sum=0;  for(i=0; i<n; i++) sum=sum+a[i];
```

A simple parallelization (assuming $\mathsf{n} > \mathsf{p}$) would be

```
for(i=id*n/p; i<min((id+1)*n/p,n); i++)
    s[id]+=a[i];
for(i=0; i<p; i++) sum=sum+s[i];
```

where `id` is the thread number and `s` is an array of size `p` for keeping the partial sums of each thread. The total number of iterations of the most loaded process is $N(\mathsf{n},\mathsf{p}) = T_p = \lceil \mathsf{n}/\mathsf{p} \rceil + \mathsf{p}$ and the efficiency $E_p = (\mathsf{n}+1)/(\mathsf{p}\lceil \mathsf{n}/\mathsf{p} \rceil + \mathsf{p}^2)$. This implementation is not work-efficient because the lower bound is $T_p = \Omega(\mathsf{n}/\mathsf{p} + \log_2(\mathsf{p}))$ and the efficiency decreases with p^2. The lower bound can be achieved if we combine partial results of the sum in a tree structure

```
for(i=id*n/p; i<min((id+1)*n/p,n); i++)
    s[id]+=a[i];
for(i=1; i<p; i*=2) combine_partial_sums(s);
```

with the iteration count of the most loaded process $N(\mathsf{n},\mathsf{p}) = T_p = \lceil \mathsf{n}/\mathsf{p} \rceil + \lceil \log_2(\mathsf{p}) \rceil$. The work of this solution is $W(\mathsf{n}) = T_1 = \mathsf{n}$ and the depth $D(\mathsf{n}) = T_\infty = \infty$ because the parallelization is not work-conserving (more work is created as threads are added). The parallel efficiency is $E_p = \mathsf{n}/(\mathsf{p}\lceil \mathsf{n}/\mathsf{p} \rceil + \mathsf{p}\lceil \log_2(\mathsf{p}) \rceil)$ which decreases slowly because $\log_2(\mathsf{p})$ work is added per process. From E_p, we can derive that the available parallelism is n.

This example shows that it is crucial to catch loop behavior in the analysis of parallel programs. Different implementations solving the same problem may have different work and depth, some of which resulting in limited scalability. Experiments at small scale may not expose those limitations as the constants are often rather small. However, our analysis enables us to find those issues early during the development.

The main contributions of this work are:

- We develop a mechanism to symbolically bound the number of iterations in program loops depending on the input parameters and the number of processes.

- We show how to interpret the iteration counts in terms of work and depth. This allows the user to determine the parallel efficiency of a given code.
- We briefly outline how our method can be implemented in a compiler or code analysis tool.
- We demonstrate the applicability of our method and analyze a set of real-world applications for their parallel work and depth and efficiencies.

2. PROBLEM DESCRIPTION

Counting numbers of loop iterations of arbitrary codes is impossible because even termination of arbitrary loop nests cannot be decided [34]. In our work, we focus on the class of loops where all loop update functions and loop guards are affine functions of *iteration variables*, i.e., variables that change during loop execution. It was shown in previous works that a subset of this class covers many important codes in parallel computing [9].

Our method is strictly more powerful than other iteration counting approaches (e.g., [6]) that require that loop update functions are valid expressions in Presburger arithmetic (which supports only addition and subtraction of symbolic values and constants). We refer the reader to Section 8 for a more detailed differentiation. In this paper we focus on the extraction of work and depth for affine loop nests. To do so, we need to find the number of iterations of the program as a function of the number of processes.

Affine loop. Let $x \in \mathbb{Z}^m$ be an integer-valued *iteration variable vector* and x_0 its *initial assignment* right before entering the loop. We call a loop *affine* if we can present it in the form[2] :

```
$x \leftarrow x_0$             // Initial assignment
while($c^T x < g$)       // Loop guard
    $x \leftarrow Ax + b$       // Loop update
```

Listing 1: Affine Loop

The *loop guard* $c^T x < g$ is determined by the constant vector $c \in \mathbb{R}^m$ and bounded by a scalar constant g. The loop update function $Ax + b$, consisting of a real matrix $A \in \mathbb{R}^{m\times m}$ and a constant vector $b \in \mathbb{R}^m$, determines how the iteration variables are updated during each iteration. Each constant may represent a symbolic loop parameter.

Perfectly Nested Loops. We extend our definition to a program consisting of r nested affine loops:

1. Each *loop guard* $c_k^T x < g_k$ at level k is an affine predicate of the iteration variables from levels $1 \dots k$.
2. Each loop body at levels $1 \dots r-1$ consists of three elements:
   (a) initial assignment - $A_k x + b_k$
   (b) nested loop(s)
   (c) loop update - $U_k x + v_k$

We require well-structured programs [17]: For a loop at level k, the initial assignment, loop guard and loop update may only use variables defined at the same or higher levels $1 \dots k$. Iteration variables of any parent loop at level $1 \dots k-1$ may not be changed in nested loops at levels $k \dots r$. Such loops can thus be expressed in the general form

[2]We use an arrow ($\leftarrow$) symbol to denote an assignment in math notation

```
while($c_1^T x < g_1$) {
  $x \leftarrow A_1 x + b_1$
  while($c_2^T x < g_2$) {
    ...
     $x \leftarrow A_{k-1} x + b_{k-1}$
     while($c_k^T x < g_k$) {
        $x \leftarrow A_k x + b_k$
        while($c_{k+1}^T x < g_{k+1}$) {...}
        $x \leftarrow U_k x + v_k$  }
     $x \leftarrow U_{k-1} x + v_{k-1}$}
    ...
  $x \leftarrow U_1 x + v_1$}
```

where $A_k, U_k \in \mathbb{R}^{m\times m}$, $b_k, v_k, c_k \in \mathbb{R}^m$, $g_k \in \mathbb{R}$ and $k = 1 \dots r$. Furthermore, $\forall i < k, i \neq j : A_{k,i,j} = U_{k,i,j} = 0$, $\forall i < k, i = j : A_{k,i,j} = U_{k,i,j} = 1$ and $\forall i > k : g_{k,i} = 0$.

Note that even though all the assignments and loop guards are affine, the number of iterations of such a nested loop may not be affine. For example the following affine loop will iterate $\lceil \log_2(n) \rceil$ times:

```
x=1;
while(x<n) x=2*x;
```

Perfectly nested loops are rare and loops often contain multiple loops at the same level. We now outline how our scheme also supports multipath loops.

Multipath Loops are loops that may contain multiple nested loops in one parent loop body. The example in Listing 2 shows such a loop: Inside the outer loop body we have two inner loops. How multiple loops are combined to fit the model description is covered in Section 4.5.

```
x=1;
while(x<n/p+1) {
  y=x;
  while(y<m) {S1; y=2*y;}
  z=x;
  while(z<m) {S2; z=z+x;}
  x=2*x;}
```

Listing 2: Complex Multipath Loop Nest

It is time-consuming and error-prone for humans to derive work and depth of complex loops like the one shown in Listing 2. Our algorithm computes work and depth for each statement automatically. For example, the number of executions N of the statement `S2` is bounded by

$$2\mathtt{m}\left(1 - \left\lceil \frac{\mathtt{n}}{\mathtt{p}} + 1 \right\rceil^{-1}\right) - \log_2\left(\left\lceil \frac{\mathtt{n}}{\mathtt{p}} + 1 \right\rceil\right) \leq N \leq \mathtt{m}\left(2 - \left\lceil \frac{\mathtt{n}}{\mathtt{p}} + 1 \right\rceil^{-1}\right).$$

This bounds the work W on a single process

$$2\mathtt{m}\left(1 - (\mathtt{n}+1)^{-1}\right) - \log_2(\mathtt{n}+1) \leq W \leq \mathtt{m}\left(2 - (\mathtt{n}+1)^{-1}\right)$$

and the depth D

$$0 \leq D \leq \mathtt{m}.$$

3. SKETCH OF THE ALGORITHM

We first introduce the concept of a closed-form *affine representation*. The affine representation of a single affine loop consists of two elements:

1. A single affine statement, which represents the value of the vector x after i iterations of the loop

$$x(i) = L(i) \cdot x_0 + p(i), \text{ and}$$

2. the *counting function* $n(x_0)$ that states how many times the loop will iterate before the loop guard $c^T x(i) < g$ is violated.

The variable i represents the current iteration step. We will refer to i as the *iteration counter*; x_0 is the value of the vector x before entering the loop.

We now provide an intuitive sketch of our algorithm: Given r perfectly nested affine loops, starting from the inner loop, we replace each loop with its affine representation. For r nested loops the result is

$$x(i_1, \ldots, i_r) = A_{final}(i_1, \ldots, i_r)x_0 + b_{final}(i_1, \ldots, i_r) \quad (1)$$

where $i_k = 0 \ldots n_k(x_{0,k})$, matrix A_{final} and vector b_{final} are the compositions of all L_k and p_k, $k = 1 \ldots r$.

The function $n_k(x_{0,k})$ represents the number of iterations in the kth loop with the starting conditions $x_{0,k}$. The starting conditions depend on iteration counters of all the loops at higher levels, i.e., $x_{0,k} = \phi_k(i_1, \ldots, i_{k-1})$.

Number of iterations. We can compute the total number of iterations of the innermost loop using the counting function of each loop:

$$N = \sum_{i_1=0}^{n_1(x_{0,1})} \sum_{i_2=0}^{n_2(x_{0,2})} \cdots \sum_{i_{r-1}=0}^{n_{r-1}(x_{0,r-1})} n_r(x_{0,r}). \quad (2)$$

To solve Equation (2), we need to compute:

1. the affine representations for all the loops together with their counting functions $n_k(x_{0,k})$,

2. the starting conditions for all loops as functions of iteration counters $x_{0,k} = \phi_k(i_1, \ldots, i_{k-1})$, and

3. all the sums in Equation (2).

Work and depth analysis. The number of processes in explicitly parallel programs is always available as a special variable which we call p. In parallelized codes, p is used in loop guards or loop update functions to divide the work into p pieces. Our algorithm determines the number of iterations as a function of all program parameters. We can then define the work of a program as $W = N|_{p=1}$ and depth $D = N|_{p\to\infty}$. Parallel efficiency and exposed parallelism can be computed as described in Section 1.1.

4. ALGORITHM DESCRIPTION

We now describe all the steps and approximations needed to solve Equation (2) which determines the final loop count.

4.1 Affine representation of nested loops

We now explain how we transform a perfectly nested loop into a single affine statement. This statement can then be combined with the initial assignments and the original loop update function of the parent loop into a new loop update function that represents the whole loop nest.

Each loop update statement $x \leftarrow Ax + b$ is a recursive formula for the value of the vector x in the current step, given the value in the previous step. The closed form of that formula for vector x after i iterations and with the starting value x_0 can be written as

$$\hat{x}(i, x_0) = A^i \cdot x_0 + \sum_{j=0}^{i-1} A^j b. \quad (3)$$

Using $x(i, x_0)$, we compute the number of iterations d after which the loop guard is not satisfied

$$n(x_0) = \left\lceil \underset{d}{\operatorname{argmin}}(c^T \cdot x(d, x_0) \geq g) \right\rceil. \quad (4)$$

Equation (4) defines the *counting function* $n(x_0)$. Let $L = A^i$ and $p = \sum_{j=0}^{i-1} A^j b$ from Equation (3). After we have obtained the closed affine form of a loop at level $k+1$, we can transform the loop nest at level k to

```
while($c_k^T x < g_k$) {
  $x \leftarrow A_k x + b_k$     // Initial assignment ($x_{0,k+1}$)
  $x \leftarrow L_k x + p_k$     // Nested loop (aff. rep.)
  $x \leftarrow U_k x + v_k$}    // Loop update
```

where $x = L_k x + p_k$ is the closed-form representation of the $(k+1)$st loop. Furthermore, the three affine statements can be combined to one

$$x \leftarrow U_k(L_k(A_k x + b_k) + p_k) + v_k. \quad (5)$$

We can then use it to form the affine representation of the parent loop. Applying this procedure recursively for all k loop nests will produce the final affine representation $x = A_{final}x_0 + b_{final}$ that expresses the whole loop nest (cf. Equation (1)).

Example of an affine representation.

The following example illustrates how to transform a loop into its affine representation. Consider the following loop

```
y=y0; z=z0;
while(y<z) {y+=2; z--;}
```

that we can write in matrix form as

$$x_0 = \begin{pmatrix} \mathtt{y}_0 \\ \mathtt{z}_0 \end{pmatrix}, \; x(i+1) = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} x(i) + \begin{pmatrix} 2 \\ -1 \end{pmatrix}, \; c = \begin{pmatrix} 1 \\ -1 \end{pmatrix}$$

and $g = 0$. Using Equation (3), we get the *affine representation* of that loop

$$x(d, x_0) = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} x_0 + \begin{pmatrix} 2d \\ -d \end{pmatrix}.$$

Equation (4) results in

$$\left\lceil \underset{d}{\operatorname{argmin}} \left(\begin{pmatrix} 1 & -1 \end{pmatrix} \cdot \left(\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} x_0 + \begin{pmatrix} 2d \\ -d \end{pmatrix} \right) \geq 0 \right) \right\rceil,$$

that can be simplified to $\lceil \operatorname{argmin}_d(\mathtt{z}_0 - \mathtt{y}_0 \leq 3d) \rceil$. A symbolic solver (e.g., MuPAD[10]) will determine the solution for $\lceil d = (\mathtt{z}_0 - \mathtt{y}_0)/3 \rceil$, which leads to the counting function $n(x_0) = \lceil (\mathtt{y}_0 - \mathtt{y}_0)/3 \rceil$.

4.2 Starting conditions

The starting conditions $x_{0,k+1}$ for a loop at level $k+1$ are determined by the value of the vector x before entering the loop. For each loop, at depths $k = 1, \ldots, r$, let $\hat{x}_k$ denote the corresponding function defined in equation (3), giving the i_kth *initial assignment* at level k, for $i_k = 1, \ldots, n_k(x_{0,k})$,

$$x_{0,k+1} = A_k \cdot \hat{x}_k(i_k, x_{0,k}) + b_k. \quad (6)$$

We can now count the starting conditions recursively until we reach the top level, where $x_{0,1} = x_0$. In general, the starting condition at level k are compositions of affine representations and initial assignments of all the loops from level $1 \ldots k-1$, treating all iteration variables $i_1, i_2, \ldots, i_{k-1}$ as parameters.

Example for the starting condition.

We now show a small example to illustrate the computation of the starting conditions for inner loops. Given the two nested loops

```
y=y₀; z=z₀;
while(y<m) {
  z=y;
  while(z<m) {z++;}
  y*=2;}
```

Listing 3: Nested Loop

The affine representation for the inner loop with iterator i_2 and starting conditions $x_{0,2}(i_1)$ depending on the outer loop

$$x(i_2) = \begin{pmatrix}1 & 0\\0 & 1\end{pmatrix} x_{0,2}(i_1) + \begin{pmatrix}0\\i_2\end{pmatrix},\ n_2 = \mathtt{m} - \begin{pmatrix}0 & 1\end{pmatrix} \cdot x_{0,2}(i_1),$$

and for the outer loop with iterator i_1 and starting conditions x_0

$$x(i_1) = \begin{pmatrix}2^{i_1} & 0\\ \frac{2^{i_1}}{2} & 0\end{pmatrix} x_0 + \begin{pmatrix}0\\i_2\end{pmatrix},\ n_1 = \left\lceil \log_2\left(\frac{\mathtt{m}}{\begin{pmatrix}1 & 0\end{pmatrix}\cdot x_0}\right)\right\rceil.$$

The initial assignment for the outer loop is

$$A_1 = \begin{pmatrix}1 & 0\\1 & 0\end{pmatrix};\quad b_1 = \begin{pmatrix}0\\0\end{pmatrix};\quad x_0 = \begin{pmatrix}\mathtt{y_0}\\\mathtt{z_0}\end{pmatrix}.$$

Starting conditions for the inner loop using Equation (6) are

$$x_{0,2}(i_1) = A_1 x(i_1) + b_1 = \begin{pmatrix}2^{i_1} & 0\\2^{i_1} & 0\end{pmatrix} x_0 = \begin{pmatrix}2^{i_1}\mathtt{y_0}\\2^{i_1}\mathtt{y_0}\end{pmatrix}.$$

Using Equation (2) leads to the exact number of iterations.

4.3 Counting the number of iterations

All counting functions and starting conditions can be combined into a final symbolic iteration count. First, we compute all r starting conditions $x_{0,1}, x_{0,2}, \ldots, x_{0,r}$ in the form $x_{0,1} = x_0$, $x_{0,2} = f_2(i_1, x_0)$, …, $x_{0,r} = f_r(i_1, i_2, \ldots, i_{r-1}, x_0)$. Then, we compute all counting functions in the form $n_1(x_{0,1})$, $n_2(x_{0,2})$, …, $n_r(x_{0,r})$. This enables us to calculate the sums of Equation (2) and solve for the final loop iteration count and derive work and depth from this. We use a symbolic solver to simplify and solve the equations.

In some cases, it is not possible to symbolically determine the exact solution from Equation (2). We differentiate two cases:

1. The counting function contains a ceiling, e.g., $\sum_{i=1}^{\lceil \frac{n}{2} \rceil} i = \begin{cases}\frac{(n+2)n}{8} & \text{if } n \text{ is even}\\ \frac{(n+3)(n+1)}{8} & \text{if } n \text{ is odd}\end{cases}$

2. The symbolic solver cannot find a closed form, e.g., $\sum_{i=1}^{n} i \cdot \log_2(i)$.

In both cases, we derive lower and upper bounds of the respective sum.

Bounded Sum Approximation (BSA) Algorithm. We now show our BSA algorithm that tightly approximates lower and upper bounds of Equation (2) in the two cases where the exact solution cannot be determined.

Obtaining bounds in the first case is simple. For a function $\lceil f(n) \rceil$, we determine the upper bound as $f(n) + 1$ and the lower bound as $f(n)$.

For the second case, with no symbolic solution for a sum, we approximate the sum with an integral [28]. For a non-decreasing function $f_1(i)$

$$\int_0^{n+1} f_1(x-1)\,\mathrm{d}x \le \sum_{i=0}^{n} f_1(i) \le \int_0^{n+1} f_1(x)\,\mathrm{d}x,$$

and for a non-increasing function $f_2(i)$

$$\int_0^{n+1} f_2(x-1)\,\mathrm{d}x \ge \sum_{i=0}^{n} f_2(i) \ge \int_0^{n+1} f_2(x)\,\mathrm{d}x.$$

If $f(i)$ is not monotonic in the interval $[0, n]$, then we split it into smaller intervals in which $f(i)$ is monotonic. For this, we compute the first $\frac{df}{di}$ and second $\frac{d^2f}{di^2}$ derivatives symbolically. Then we apply the approximation in each segment and combine them to get the proper upper and lower bounds.

While solving Equation (2) we need to carry the lower and upper bounds forward recursively. In the branch of the lower bounds, we only consider lower bounds of parent loops and similarly in the upper bound branch. We may require case differentiations if some counting functions are not monotonic. However, we rarely observed non monotonic counting functions in practice.

Example for bounded sum approximation.

Assume the following nested loop:

```
k=1; l=2;
while(k>0) {
  m = k;
  while(m<s) m++;
  k = k+l;
  l--;}
```

We see that $\mathtt{k}_{0,1} = \mathtt{k}_0 = 1$ and $\mathtt{l}_{0,1} = \mathtt{l}_0 = 2$. The counting function for the inner loop is

$$n_2 = \mathtt{s} - \mathtt{k}_{0,2}$$

and for the outer loop

$$n_1 = \left\lceil \mathtt{l}_{0,1} + \frac{\sqrt{4\mathtt{l}_{0,1}{}^2 + 4\mathtt{l}_{0,1} + 8\mathtt{k}_{0,1} + 1} + 1}{2} \right\rceil = 6.$$

The starting conditions for the inner loop are

$$\mathtt{k}_{0,2} = \mathtt{k}_{0,1} + i_1 \cdot \mathtt{l}_{0,1} - \frac{i_1 \cdot (i_1 - 1)}{2} = -\frac{1}{2} i_1^2 + \frac{5}{2} i_1 + 1.$$

The number of iterations of the loop nest, according to Equation 2 is

$$N = \sum_{i_1=0}^{n_1} n_2 = \sum_{i_1=0}^{n_1} (\mathtt{s} + \frac{1}{2} i_1^2 - \frac{5}{2} i_1 - 1)$$

We now approximate this sum with an integral. Analyzing the first and second derivative of n_2 shows that within

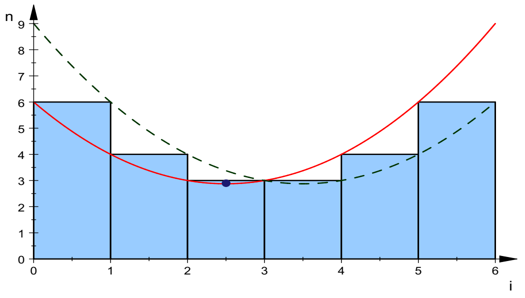

Figure 1: **Series approximation.** Bars represent the series $n_2(i)$, the red line shows the function $n_2(k)$ and the dashed green line shows the function $n_2(k-1)$. The function $n_2(k)$ over-approximates the series in the interval $[0,2]$ and under-approximates it in the interval $[3,6]$; the dot shows the saddle point.

the interval (0,n_1) the function is decreasing in (0,2.5) and increasing in (2.5,n_1). The sum can then be bounded from above by

$$U = \int_0^2 n_2(i_1-1)\,\mathrm{d}i_1 + n_2(2) + \int_3^{n_1} n_2(i_1)\,\mathrm{d}i_1 = 6s - \frac{47}{12}$$

and from below by

$$L = \int_0^2 n_2(i_1)\,\mathrm{d}i_1 + n_2(2) + \int_3^{n_1} n_2(i_1-1)\,\mathrm{d}i_1 = 6s - \frac{161}{12}.$$

Figure 1 illustrates the example.

4.4 Correctness of the algorithm

All the steps of the algorithm except the sum approximation are proper algebraic transformations. If no approximation is needed then the algorithm produces the exact number of iterations. For example for the loops in Listing 3 the total number of iterations of the inner loop is exactly

$$N = \mathtt{y}_0 - 2^{\left\lceil \log_2\left(\frac{\mathtt{m}}{\mathtt{y}_0}\right)\right\rceil}\mathtt{y}_0 + \mathtt{m}\left\lceil \log_2\left(\frac{\mathtt{m}}{\mathtt{y}_0}\right)\right\rceil.$$

If approximation is required, then we need to prove that we obtain proper upper and lower bounds and that this property propagates further through the algorithm.

Lemma 4.1. *The Bounded Sum Approximation algorithm gives valid lower and upper bounds for Equation* (2)*.*

Proof. Ceiling upper and lower bound for type 1 approximations are correct by the definition of the ceiling function. Let us consider type 2 sum approximations.

First we need to prove that the upper and lower bounds for a series $f(n)$, where $n = 0, \ldots, k$, found by the algorithm are correct. Let's denote

$$U_{[a,b]}(x) = \begin{cases} f(x), & \text{if } \forall x \in [a,b] : \frac{df}{dx} \geq 0 \\ f(x-1), & \text{if } \forall x \in [a,b] : \frac{df}{dx} \leq 0 \end{cases}$$

$$L_{[a,b]}(x) = \begin{cases} f(x-1), & \text{if } \forall x \in [a,b] : \frac{df}{dx} \geq 0 \\ f(x), & \text{if } \forall x \in [a,b] : \frac{df}{dx} \leq 0 \end{cases}$$

as upper and lower bounds of the monotonic series f in the interval $[a,b]$, as stated in Section 4.3.

Let $c \in [0,k]$ be the only saddle point of function $f(x)$. Intervals with multiple saddle points can be split to smaller intervals where each contains a single saddle point. Then, U and L will change from $f(x-1)$ to $f(x)$ or from $f(x)$ to $f(x-1)$ at that point c. The upper bound $U_{[0,\lfloor c\rfloor]} = U_{[0,c]} \neq U_{[c,k]} = U_{[\lceil c\rceil,k]}$ does not change in the intervals $[0,\lfloor c\rfloor]$ and $[\lceil c\rceil, k]$. We can then bound the value of $f(\lfloor c\rfloor)$ with $U_{[0,c]}(\lfloor c\rfloor)$. Thus,

$$\int_0^{\lfloor c\rfloor} U_{[0,c]}(x)\,\mathrm{d}x \geq \sum_{i=0}^{\lfloor c\rfloor} f(i), \int_{\lceil c\rceil}^{k} U_{[c,k]}(x)\,\mathrm{d}x \geq \sum_{i=\lceil c\rceil}^{k} f(i)$$

$$U_{[a,c]}(\lfloor c\rfloor) \geq f(\lfloor c\rfloor).$$

From this follows that the upper bound of the non-monotonic sum of series $\sum_{i=0}^{k} f(i)$, with saddle point c, can be expressed as:

$$U = \int_0^{\lfloor c\rfloor} U_{[0,c]}(x)\,\mathrm{d}x + U_{[0,c]}(c) + \int_{\lceil c\rceil}^{k} U_{[c,n]}(x)\,\mathrm{d}x \geq \sum_{i=0}^{k} f(i).$$

The lower bounds discussion follows a similar reasoning.

We now prove propagation of this property through consecutive sums in Equation 2: Let f_k be the kth sum from Equation 2, and U_k and L_k upper and lower bounds of f_k. We then can present it as

$$\sum_{i_k=0}^{n_k} n_{k+1}(i_1, \ldots, i_{k-1}, x_0) = f_k(i_1 \ldots, i_k, x_0)$$

and bound it with

$$L_k(i_1 \ldots, i_{k-1}, x_0) \leq f_k(i_1, \ldots, i_{k-1}, x_0) \leq U_k(i_1, \ldots, i_{k-1}, x_0).$$

The next sum at level $k-1$ will be

$$\sum_{i_{k-1}=0}^{n_{k-1}} f_k(i_1, \ldots, i_{k-1}, x_0)$$

Upper and lower bounds are not changed by summing, such that

$$\sum_{i_{k-1}=0}^{n_{k-1}} f_k(i_1, \ldots, i_{k-1}, x_0) \geq \sum_{i_{k-1}=0}^{n_{k-1}} L_k(i_1, \ldots, i_{k-1}, x_0)$$

and

$$\sum_{i_{k-1}=0}^{n_{k-1}} f_k(i_1, \ldots, i_{k-1}, x_0) \leq \sum_{i_{k-1}=0}^{n_{k-1}} U_k(i_1, \ldots, i_{k-1}, x_0)$$

implies $L_1(x_0) \leq f_1(x_0) = N \leq U_1(x_0)$. □

4.5 Multipath loops

We formalize a loop containing multiple statements as

```
while($c_k^T x < g_k$) {
  $x \leftarrow A_{k,1}x + b_{k,1}$
  $x \leftarrow A_{k,2}x + b_{k,2}$
   ...
  $x \leftarrow A_{k,m}x + b_{k,m}$}
```

where each of the statements $x \leftarrow A_{k,i}x + b_{k,i}$ may be a simple assignment or an affine representation of a loop. We compose them in the same way as we did in Equation (5), forming a single affine statement.

Starting conditions. We compute the starting conditions for each loop by generalizing Equation (6). For multipath loops the starting condition for a loop represented by its affine representation $x \leftarrow A_{k,i}x + b_{k,i}$ is the composition of all the affine statements that precede it:

$$x_{0,k+1,i} = A_{k,i-1}(\ldots(A_{k,1} \cdot \hat{x}_k(i_k, x_{0,k-1}) + b_{k,1})\ldots) + b_{k,i-1}$$

For illustration consider the following example loop:

```
while($c_k^T x < g_k$) {
   $x \leftarrow A_{k,1}x + b_{k,1}$
   $x \leftarrow A_{k,2}x + b_{k,2}$
   $x \leftarrow A_{k,3}x + b_{k,3}$
   $x \leftarrow A_{k,4}x + b_{k,4}$
   $x \leftarrow A_{k,5}x + b_{k,5}$}
```

Assume that in the example above, $x \leftarrow A_{k,2}x + b_{k,2}$, $x \leftarrow A_{k,3}x + b_{k,3}$ and $x \leftarrow A_{k,4}x + b_{k,4}$ are affine representations of three nested loops. Then, the starting condition for the third loop $x \leftarrow A_{k,4}x + b_{k,4}$ is

$$x_{0,k+1,4} = A_{k,3}(A_{k,2}(A_{k,1} \cdot \hat{x}_k(i_k, x_{0,k-1}) + b_{k,1}) + b_{k,2}) + b_{k,3}.$$

Counting the number of iterations. We solve Equation (2) using the appropriate counting function $n_r(x_{0,r})$ for each loop. The series of sums is formed according to the hierarchy of loops starting at the innermost loop.

5. PRACTICAL CONSIDERATIONS

We now briefly outline how the developed method can be used to assess work and depth of real applications. This paper is focusing on the fundamental techniques, yet, we want to provide a coarse view of how we apply our method in practical settings.

The whole mechanism can be implemented in a source-code analysis tool or a compiler. We use the Low Level Virtual Machine (LLVM [26]) and will outline the implementation. LLVM's internal program representation uses Single Static Assignment (SSA), which makes it simple to determine loops (identified by back-edges), loop guards (identified by conditional branches with back-edges), and all dependent variables.

From this information, we create initial assignments, loop guards, and loop updates for each loop and apply the procedure described in Section 4. While the vast majority of loops in practical programs are affine, some loops may depend on more complex conditions and thus do not fit our framework. However, one of the main strengths of our method is that we can still compute the number of iterations of loop nests containing non-affine functions as we will describe in the following section.

5.1 Extensions for non-affine loops

If a loop guard is not an affine function of iteration variables and constant parameters then we may not be able to determine the exact number of iterations statically. Examples are loops with iteration counts determined by unknown functions or complex sources like arrays that keep dynamic data. This is often the case in applications that iterate until a complex convergence criterion is reached, e.g., conjugate gradient methods. If the loop exit conditions cannot be represented as affine statements, then the whole block is treated as a symbolic value u (*undefined*).

A strength of our method is that it still solves the remaining affine loop nests symbolically as u is simply treated as a parameter that propagates while solving Equation 2. In addition to just treating non-affine loops as new symbolic parameters, our tool enables the user to annotate such loops with affine upper and lower bound functions.

We demonstrate the technique with a loop that we found during one of our case studies. The following Fortran code is extracted from the NAS CG benchmark.

```
do j=1,lastrow-firstrow+1
   sum = 0.d0
   do k=rowstr(j),rowstr(j+1)-1
      sum = sum + a(k)*p(colidx(k))
   enddo
   w(j) = sum
enddo
```

Our tool traces the expression `lastrow-firstrow+1` back to the program parameter $\texttt{row\_size} = \frac{\texttt{na}}{\texttt{nprows}}$ or $\texttt{row\_size} = \frac{\texttt{na}}{\texttt{nprows}} + 1$, depending on the process id. This reflects the fact that `nprows` may not divide `na`, where `na` is the problem size. However, the value of `rowstr(j)` cannot be determined statically because it represents the location of the first nonzero value in row j of one of the program arrays. Our algorithm then represents the previous loop nest as:

```
j=1;
while(j<=row_size) {$u$; j++;}
```

u is treated as a loop with the fixed number of iterations u =`irowstr(j+1)-rowstr(j)`. The total number of iterations of this code fragment for the most busy process is represented as

$$N = \left\lceil \frac{\texttt{na}}{\texttt{nprows}} \right\rceil u.$$

It is possible to provide the user with information about the exact code fragment that is the origin of u. Users can then determine upper and lower bounds for each unknown parameter.

6. CASE STUDIES

In this section we present our results from analyzing several benchmarks. We compute work and depth of several parallel programs to demonstrate the insight we gained into the bounds on parallel efficiency of those practical codes. Our analysis allows us to make statements about parallel efficiency without studying the problem or implementation.

We analyzed major loops of the NAS parallel benchmarks version 3.2 [3] and the Mantevo micro applications version 2.0 [5]. We only present an interesting subset in our case studies due to space constraints.

In all the cases presented, no approximation was needed, so presented results give exact number of iterations (with respect to the introduced constants). If not stated otherwise, in the following benchmarks m represents the problem scale, n is a program parameter to NAS specifying the number of iterations to perform, and p denotes number of processes. In some cases processes are arranged into multiple dimensions. In those cases, $p_1, p_2, \ldots, p_k$ represents number of processes in corresponding dimensions and $p = \prod_{i=1}^{k} p_i$. We also assume that the decomposition is square, i.e., $p_1 \approx p_2 \approx \cdots \approx p_k$. For easier work-depth analysis, presented results are simplified by replacing constant terms with auxiliary constants c_i. We use constants instead of asymptotic notation to retain lower-order terms.

6.1 NAS Parallel Benchmarks: EP

The NAS EP benchmark represents a typical Monte Carlo simulation and thus performs nearly no communication. Our analysis found that only one out of seven loops could not be resolved due to a conditional **goto** statement, resulting in a single u. The work of EP is

$$N = \left\lceil \frac{2^{m-16} \cdot (u + 2^{16})}{p} \right\rceil.$$

The following listing shows the non-affine loop that determines u after removing all statements that do not influence the iteration count:

```
u :    do i=1,100
         ik =kk/2
         if (ik .eq. 0) goto 130
         kk=ik
       continue
```

A programmer can easily determine that $u \leq 100$, which is negligible compared to 2^{16}. Thus, we can approximate the work using one thread $W = T_1 \approx 2^m$ and depth $D = T_\infty \approx 1$. This shows that the the parallelization is work-optimal and the efficiency

$$E_p \approx \frac{2^m}{p \left\lceil \frac{2^m}{p} \right\rceil}.$$

This means that $E_p \approx 1$ if $p \lesssim 2^m$ and $E_p \approx 2^m/p$ if $p \gtrsim 2^m$. We conclude that the maximum available parallelism in EP is 2^m, because N cannot further decrease for $p \gtrsim 2^m$. This does not mean that the code will efficiently execute with 2^m tasks, however, it specifies an upper bound to the speedup. This is an expected result since the code is considered “embarrassingly parallel”.

6.2 NAS Parallel Benchmarks: CG

The NAS CG program represents a typical conjugate gradient solver. Our tool found that only 2 out of 23 analyzed loops were not affine (cf. Section 5.1). The two undefined loops had identical guards, resulting in a single parameter u that can be bounded as $0 \leq u \leq m$. This allows us to bound the number of iterations

$$N \lesssim n \left(g \left\lceil \frac{m}{p} \right\rceil + (6 + 5g)\sqrt{\left\lceil \frac{m}{p} \right\rceil} + (3g + 4)\log_2(\sqrt{p}) \right)$$

where g is the program parameter `cgitmax`. We can approximate work on one thread $W = T_1 \lesssim (g\ m + \sqrt{m}(5g + 6))n$. However, CG is not work optimal as the parallel work is monotonically growing. This causes the depth to be $D = T_\infty = \infty$. If we treat the problem size m as constant, then CG’s parallel efficiency is

$$E_p = c_1 \left(c_2\ p \log_2(\sqrt{p}) + c_3 \left\lceil \frac{m}{p} \right\rceil + c_4 \sqrt{\left\lceil \frac{m}{p} \right\rceil} \right)^{-1}.$$

This means that the work per process increases with $\log_2(\sqrt{p})$ due to a parallel reduction among $\sqrt{p}$ processes. The available parallelism is m.

6.3 NAS Parallel Benchmarks: IS

The NAS IS program represents a parallel bucket sort algorithm. In each iteration, all processes perform their local sorting and exchange information. The communication overhead is expected to grow with the number of processes. A total of 5 out of 15 analyzed loops were not affine. Those five loops fall in two classes: the first class iterates over maximum and minimum key value represented as u_1=`max_key_val-min_key_val+1`; the second class iterates over the buckets after redistribution and is represented as u_2=`bucket_distrib_ptr2[k] - bucket_distrib_ptr1[k]`. Both loop iteration counts depend on the structure of the input and the distribution and can thus not easily be bounded tightly. The total number of iterations of IS is

$$N = n \left(3(b + t) + 2 \left\lceil \frac{m}{p} \right\rceil + p + 2 \cdot u_1 + 2 \cdot u_2 \right)$$

where b is the number of buckets, t is the size of the test array, and m is the number of keys to be sorted. The total work on one thread is

$$W = T_1 = n(3(b + t) + 2m + 2u_1 + 2u_2 + 1).$$

The depth $D = \infty$ because the parallelization is not work efficient which is due to the necessary inter-process communications. The parallel efficiency of IS is $E_p = c_1/\left(p^2 + c_2\ p + c_3\ p \left\lceil \frac{m}{p} \right\rceil\right)$, which drops quickly with the number of processes used. The reason is that each process may need to communicate with each other process. The available parallelism is also m in this case.

6.4 Mantevo Benchmarks: CoMD

The Mantevo CoMD benchmark represents a classical molecular dynamics simulation. Eight out of 18 analyzed loops contain non-affine statements. The code distributes atoms to processes. The first class updates atoms in the partitions and u_1 represents the number of iterations of those loops.

```
u1: while(i<boxes->nAtoms[iBox]) {
      int jBox=getBox(atoms->r[iOff+i]);
      if (jBox!=iBox) moveAtom(i,iBox,jBox);
      else ++i;}
```

The second class u_2=`qsort(nAtoms[iBox])` is limited by the data sizes to be sorted. The number of iterations of the CoMD Benchmark is

$$N = n \left(g(B + 3) \left\lceil \frac{m}{p} \right\rceil + g\ T \left(\left\lceil \frac{m}{p} \right\rceil u_1 + u_2 \right) + \left\lceil \frac{m}{p\ B} \right\rceil + 2 \right)$$

where B is the fixed amount of atoms in each box, g is the `print rate` program parameter, and T is the total number of boxes:

$$T = 2 \left(\frac{\sqrt[3]{m}}{p_1} + 2 \right) \left(\frac{\sqrt[3]{m}}{p_2} + \frac{\sqrt[3]{m}}{p_3} + 2 \right) + \frac{\sqrt[3]{m^2}}{p_2\ p_3} + \frac{m}{p_1\ p_2\ p_3}.$$

If we bound $u_1 < B^2$ and $u_2 < B \log_2(B)$, then we can approximate work and depth: $W \lesssim c_1\ m + c_2\ m^{2/3} + c_3\ m^{1/3} + c_4$, and $D \lesssim n(c_5 + c_6\ B(\log_2(B)))$. The implementation is work-optimal and the efficiency $E_p \lesssim c_7/(p + c_8)$ is decreasing with number of processors, which is the result of sequential parts of the program that cannot be parallelized. The available parallelism is m.

Scalability Analysis.

We were able to determine bounds for work, depth, parallel efficiency, and available parallelism for several real-world applications. We see that the available parallelism in all investigated applications scales linearly with the input problem size. While this suggests good scaling, we show that for CG and IS communication overheads increase the work with the number of processes. For example, for IS, this overhead grows linearly with the number of used processes.

We were able to perform those analyses by pure code introspection which was guided by our tool without requiring knowledge of the implemented methods or algorithms. If the solved problem is known, then one could even proof optimality in terms of parallel efficiency or available parallelism. However, this is outside the scope of this paper.

7. DISCUSSION

We now discuss the limitations of our approach and briefly outline potential additional use-cases.

Limitations.

Since our analysis only counts loop iterations and does not account for the exact costs of each loop, we can only provide bounds on the expected execution time on a parallel system. However, those bounds are always asymptotically correct. Since we limit ourselves to the work and depth model, we cannot account for communication or synchronization overheads in real codes. Yet, the bounds we provide are useful to determine the relative behavior of work and depth and allow us to reason about exposed parallelism and parallel efficiency just like the work and depth model.

Extending the Models.

While outside the scope of this paper, it is simple to extend our work and depth models to account for system parameters such as memory or network latency and bandwidth. Blelloch [12] discusses further options.

Model-based Mapping to Heterogeneous Systems.

Having a model for the work and depth of each loop in a program can be useful when the program is to be mapped to future heterogeneous architectures. Those systems will most likely contain Latency Compute Units (LCU, cf. today's CPU cores) and Throughput Compute Unites (TCU, cf. today's accelerators such as GPUs or Xeon Phi). A compiler would need to determine the target architecture for each loop statically. It could use the generated work/depth models to assign code pieces with low parallelism (small W/D) to LCUs and code pieces with larger parallelism (large W/D) to TCUs.

8. RELATED WORK

Counting loop iterations and assessing scalability of parallel codes are important research problems. Rodriguez-Carbonell and Kapur [31] find polynomial loop invariants using an algebraic approach. Sharma et al. [32] use a data driven approach to iteratively guess the correct polynomial loop invariant and then check its correctness, and Matringe et al. [30] generate loop invariants also for non-linear differential systems. Loop invariants can be used to bound loop iteration counts but the resulting bounds are often not tight.

Multiple research groups use the polyhedral model (PM) to determine the exact number of loop iterations [1, 8, 37]. In this case, the number of iterations can be approximated by counting integer points in that polyhedron using a polynomial-time algorithm [6]. The PM is widely used, not only in loop analysis [9]. However, it has a serious limitation - it requires that the loop update function can be expressed in Presburger arithmetic and thus cannot deal with non-constant updates such as $x = 2 * x$. To the best of our knowledge, no previous work handled such cases properly. Methods like the one proposed by Blanc et al. [11] require explicitly that loops cannot include such statements.

Other works utilize dynamic approaches to extrapolate program performance and assess scalability in practical settings. Barnes et al. [4] use regression to linear and logarithmic functions to predict scalability of nearly linear-scaling HPC applications. Calotoiu et al. [14] select a scaling model from a set of predefined candidate functions and fit the parameters with regression. Other works, such as [20, 27] use multi-layer neural networks or statistical techniques to predict scalability.

More complex performance prediction frameworks consider the effect of communication [15, 29] and extrapolate single-node runs [36]. Partial execution [35] can improve those techniques. Other studies provide advice for modeling the general performance [21] and scalability [19] of parallel applications. In addition, many application-specific studies exist but cannot be generalized [7, 24].

We extend previous work significantly in two directions: first, we show a technique that can tightly bound the numbers of iterations of arbitrary affine loop nests and second, we show how this method can be used to assess work and depth of parallel applications.

9. CONCLUSIONS

We show a method to symbolically count loop iterations of practical codes in terms of their input sizes. Our method provides either an exact solution or tight upper and lower bounds using bounded sum approximation. It is applicable to affine and non-affine loops. While it can bound all affine loops accurately, it handles non-affine loop counts as a symbolic constant and allows the user to provide lower and upper bounds.

We show how to derive parallel work and depth from the loop count models. Using the work and depth model we approximate bounds on the parallel efficiency of those codes. This technique allows us to specify upper bounds to scalability of practical parallel codes. In general, our method allows a developer to quickly check how an explicitly parallel code scales with the numbers of processes and input sizes.

We are applying a standard algorithmic analysis technique (measuring work and depth) to real source codes. Our developed techniques pave the way to quickly and automatically assess program scalability and will thus quickly become an important tool for future parallel application development and analysis.

Acknowledgments

We thank the anonymous reviewers for outstandingly helpful comments to improve the quality of the paper.

References

[1] C. Alias, A. Darte, P. Feautrier, and L. Gonnord. Multi-dimensional rankings, program termination, and complexity bounds of flowchart programs. In *Static Analysis*, volume 6337 of *Lecture Notes in Computer Science*, pages 117–133. Springer, 2011.

[2] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proc. of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.

[3] D. H. Bailey et al. The NAS Parallel Benchmarks: Summary and Preliminary Results. In *Proc. of the 1991 ACM/IEEE Conference on Supercomputing*, pages 158–165, New York, NY, USA, 1991. ACM.

[4] B. J. Barnes, B. Rountree, D. K. Lowenthal, J. Reeves, B. de Supinski, and M. Schulz. A regression-based approach to scalability prediction. In *Proc. of the 22nd Annual Intl. Conference on Supercomputing*, ICS '08, pages 368–377, New York, NY, USA, 2008. ACM.

[5] R. F. Barrett, M. A. Heroux, P. T. Lin, C. T. Vaughan, and A. B. Williams. Poster: Mini-applications: Vehicles for co-design. In *Proc. of the 2011 High Performance Computing Networking, Storage and Analysis Companion*, SC '11, pages 1–2. ACM, 2011.

[6] A. I. Barvinok. A polynomial time algorithm for counting integral points in polyhedra when the dimension is fixed. *Math. Oper. Res.*, 19(4):769–779, Nov. 1994.

[7] G. Bauer, S. Gottlieb, and T. Hoefler. Performance Modeling and Comparative Analysis of the MILC Lattice QCD Application su3 rmd. In *Proc. of the 2012 12th IEEE/ACM Intl. Symposium on Cluster, Cloud and Grid Computing*, pages 652–659, May 2012.

[8] A. M. Ben-Amram and S. Genaim. On the linear ranking problem for integer linear-constraint loops. *SIGPLAN Not.*, 48(1):51–62, Jan. 2013.

[9] M.-W. Benabderrahmane, L.-N. Pouchet, A. Cohen, and C. Bastoul. The Polyhedral Model Is More Widely Applicable Than You Think. In R. Gupta, editor, *Compiler Construction*, volume 6011 of *LNCS*, pages 283–303. Springer, 2010.

[10] L. Bernardin. A review of symbolic solvers. *SIGSAM Bull.*, 30(1):9–20, Mar. 1996.

[11] R. Blanc, T. Henzinger, T. Hottelier, and L. Kovacs. ABC: Algebraic Bound Computation for Loops. In E. Clarke and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, volume 6355 of *LNCS*, pages 103–118. Springer, 2010.

[12] G. E. Blelloch. Programming parallel algorithms. *Commun. ACM*, 39(3):85–97, Mar. 1996.

[13] R. P. Brent. The parallel evaluation of general arithmetic expressions. *J. ACM*, 21(2):201–206, Apr. 1974.

[14] A. Calotoiu, T. Hoefler, M. Poke, and F. Wolf. Using automated performance modeling to find scalability bugs in complex codes. In *Proc, of SC13: Int'l Conf. for High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 45:1–45:12. ACM, 2013.

[15] L. Carrington, A. Snavely, and N. Wolter. A performance prediction framework for scientific applications. *Future Gener. Comput. Syst.*, 22(3):336–346, Feb. 2006.

[16] Z. DeVito et al. Liszt: A Domain Specific Language for Building Portable Mesh-based PDE Solvers. In *Proc. of 2011 Intl. Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 9:1–9:12, New York, NY, USA, 2011. ACM.

[17] E. W. Dijkstra. Structured programming. chapter Notes on Structured Programming, pages 1–82. Academic Press Ltd., London, UK, UK, 1972.

[18] T. Goodale, G. Allen, G. Lanfermann, J. Massó, T. Radke, E. Seidel, and J. Shalf. The cactus framework and toolkit: Design and applications. In *High Performance Computing for Computational Science*, pages 197–227. Springer, 2003.

[19] T. Hoefler, W. Gropp, M. Snir, and W. Kramer. Performance Modeling for Systematic Performance Tuning. In *Intl. Conference for High Performance Computing, Networking, Storage and Analysis (SC'11)*, Nov. 2011.

[20] E. Ipek, B. R. de Supinski, M. Schulz, and S. A. McKee. An approach to performance prediction for parallel applications. In *Proc. of the 11th Intl. Euro-Par Conference on Parallel Processing*, Euro-Par'05, pages 196–205. Springer, 2005.

[21] R. Jain. *The Art Of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling.* Wiley, 2008.

[22] J. JáJá. *An Introduction to Parallel Algorithms.* Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1992.

[23] R. M. Karp and V. Ramachandran. Handbook of Theoretical Computer Science (Vol. A). chapter Parallel Algorithms for Shared-memory Machines, pages 869–941. MIT Press, Cambridge, MA, USA, 1990.

[24] D. J. Kerbyson et al. Predictive performance and scalability modeling of a large-scale application. In *Proc. of the 2001 ACM/IEEE conference on Supercomputing*, pages 37–37. ACM, 2001.

[25] D. E. Knuth. *Sorting and Searching, The Art of Computer Programming, Volume 3: (2Nd Ed.) Sorting and Searching.* Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.

[26] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proc. of the Intl. Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, Washington, DC, USA, 2004. IEEE Computer Society.

[27] B. C. Lee, D. M. Brooks, B. R. de Supinski, M. Schulz, K. Singh, and S. A. McKee. Methods of inference and learning for performance modeling of parallel applications. In *Proc. of the 12th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, PPoPP '07, pages 249–258, 2007.

[28] C. E. Leiserson, R. L. Rivest, C. Stein, and T. H. Cormen. *Introduction to Algorithms.* The MIT press, 2001.

[29] G. Marin and J. Mellor-Crummey. Cross-architecture performance predictions for scientific applications using parameterized models. In *Proc. of the Joint Int'l Conf. on Measurement and Modeling of Computer Systems*, SIGMETRICS '04, pages 2–13. ACM, 2004.

[30] N. Matringe, A. Moura, and R. Rebiha. Generating invariants for non-linear hybrid systems by linear algebraic methods. In *Static Analysis*, volume 6337 of *LNCS*, pages 373–389. Springer, 2011.

[31] E. Rodríguez-Carbonell and D. Kapur. Automatic generation of polynomial loop invariants: Algebraic foundations. In *Proc. of the 2004 Intl. Symposium on Symbolic and Algebraic Computation*, ISSAC '04, pages 266–273, New York, NY, USA, 2004. ACM.

[32] R. Sharma, S. Gupta, B. Hariharan, A. Aiken, P. Liang, and A. Nori. A data driven approach for algebraic loop invariants. In M. Felleisen and P. Gardner, editors, *Progr. Languages and Systems*, volume 7792 of *LNCS*, pages 574–592. Springer, 2013.

[33] Y. Shiloach and U. Vishkin. An $O(n^2 \log n)$ Parallel Max-flow Algorithm. *J. Alg.*, 3(2):128–146, Feb. 1982.

[34] A. M. Turing. On Computable Numbers, with an Application to the Entscheidungsproblem. *Proc. London Math. Soc.*, 2(42):230–265, 1936.

[35] L. T. Yang, X. Ma, and F. Mueller. Cross-platform performance prediction of parallel applications using partial execution. In *Proc. of the 2005 ACM/IEEE Conference on Supercomputing*, SC '05, Washington, DC, USA, 2005. IEEE Computer Society.

[36] J. Zhai, W. Chen, and W. Zheng. Phantom: Predicting performance of parallel applications on large-scale parallel machines using a single node. In *Proc. of the 15th ACM Symp. on Principles and Practice of Parallel Progr.*, PPoPP '10, pages 305–314. ACM, 2010.

[37] F. Zuleger, S. Gulwani, M. Sinn, and H. Veith. Bound analysis of imperative programs with the size-change abstraction. In E. Yahav, editor, *Static Analysis*, volume 6887 of *Lecture Notes in Computer Science*, pages 280–297. Springer Berlin Heidelberg, 2011.