A General Framework for Static Cost Analysis of Parallel Logic Programs *

Maximiliano Klemen[1,2][0000−0002−8503−8379], Pedro López-García[1,3][0000−0002−1092−2071], John P. Gallagher[1,4][0000−0001−6984−7419], José F. Morales[1][0000−0001−9782−8135], and Manuel V. Hermenegildo[1,2][0000−0002−7583−323X]

[1] IMDEA Software Institute, Spain
[2] ETSI Informáticos, Universidad Politécnica de Madrid (UPM), Spain
[3] Spanish Council for Scientific Research (CSIC), Spain
[4] Roskilde University, Denmark
{maximiliano.klemen,pedro.lopez,john.gallagher,josef.morales,manuel.hermenegildo}@imdea.org

Abstract. The estimation and control of *resource usage* is now an important challenge in an increasing number of computing systems. In particular, requirements on timing and energy arise in a wide variety of applications such as internet of things, cloud computing, health, transportation, and robots. At the same time, parallel computing, with (heterogeneous) multicore platforms in particular, has become the dominant paradigm in computer architecture. Predicting resource usage on such platforms poses a difficult challenge. Most work on static resource analysis has focused on sequential programs, and relatively little progress has been made on the analysis of parallel programs, or more specifically on parallel logic programs. We propose a novel, general, and flexible framework for setting up cost equations/relations which can be instantiated for performing resource usage analysis of parallel logic programs for a wide range of resources, platforms, and execution models. The analysis estimates both lower and upper bounds on the resource usage of a parallel program (without executing it) as functions on input data sizes. In addition, it also infers other meaningful information to better exploit and assess the potential and actual parallelism of a system. We develop a method for solving cost relations involving the *max* function that arise in the analysis of parallel programs. Finally, we instantiate our general framework for the analysis of logic programs with Independent And-Parallelism, report on an implementation within the CiaoPP system, and provide some experimental results. To our knowledge, this is the first approach to the cost analysis of *parallel logic programs*.

Keywords: Resource Usage Analysis, Parallelism, Static Analysis, Complexity Analysis, (Constraint) Logic Programming, Prolog.

1 Introduction

Estimating in advance the resource usage of computations is useful for a number of applications; examples include granularity control in parallel/distributed systems,

* Research partially funded by Spanish MINECO TIN2015-67522-C3-1-R *TRACES* project, and the Madrid P2018/TCS-4339 *BLOQUES-CM* program. We are also grateful to the anonymous reviewers for their useful comments.

automatic program optimization, verification of resource-related specifications and detection of performance bugs, as well as helping developers make resource-related design decisions. Besides *time* and *energy*, we assume a broad concept of resources as numerical properties of the execution of a program, including the number of *execution steps*, the number of *calls* to a procedure, the number of *network accesses*, number of *transactions* in a database, and other user-definable resources. The goal of automatic static analysis is to estimate such properties without running the program with concrete data, as a function of input data sizes and possibly other (environmental) parameters.

Due to the heat generation barrier in traditional sequential architectures, parallel computing, with (heterogeneous) multi-core processors in particular, has become the dominant paradigm in current computer architecture. Predicting resource usage on such platforms poses important challenges. Most work on static resource analysis has focused on sequential programs, and relatively little progress has been made on the analysis of parallel programs, or on parallel logic programs in particular. The significant body of work on static analysis of sequential logic programs has already been applied to the analysis of other programming paradigms, including imperative programs. This is achieved via a transformation into *Horn clauses* [22]. In this paper we concentrate on the analysis of parallel Horn clause programs, which could be the result of such a translation from a parallel imperative program or be themselves the source program. Our starting point is the well-developed technique of setting up recurrence relations representing resource usage functions parameterized by input data sizes [29,25,8,7,9,24,2,26], which are then solved to obtain (exact or safely approximated) closed forms of such functions (i.e., functions that provide upper or lower bounds on resource usage). We build on this and propose a novel, general, and flexible framework for setting up cost equations/relations which can be instantiated for performing static resource usage analysis of parallel logic programs for a wide range of resources, platforms, and execution models. Such an analysis estimates both lower and upper bounds on the resource usage of a parallel program as functions on input data sizes. We have instantiated the framework for dealing with Independent And-Parallelism (IAP) [14,10], which refers to the parallel execution of conjuncts in a goal. However, the results can be applied to other languages and types of parallelism, by performing suitable transformations into Horn clauses.

The main contributions of this paper can be summarized as follows:

- We have extended a general static analysis framework for the analysis of sequential Horn clause programs [24,26], to deal with parallel programs.
- Our extensions and further generalizations support a wide range of resources, platforms, and parallel/distributed execution models, and allow the inference of both lower and upper bounds on resource usage. This is the first approach, to our knowledge, to the cost analysis of *parallel logic programs* that can deal with features such as backtracking, multiple solutions (i.e., non-determinism), and failure.
- We have instantiated the developed framework to infer useful information for assessing and exploiting the potential and actual parallelism of a system.
- We have developed a method for finding closed-form functions of cost relations involving the *max* function that arise in the analysis of parallel programs.

– We have developed a prototype implementation that instantiates the framework for the analysis of logic programs with Independent And-Parallelism within the CiaoPP system [13,24,26], and provided some experimental results.

2 Overview of the Approach

Prior to explaining our approach, we provide some preliminary concepts. Independent And-Parallelism arises between two goals when their corresponding executions do not affect each other. For pure goals (i.e., without side effects) a sufficient condition for the correctness of IAP is the absence of variable sharing at run-time among such goals. IAP has traditionally been expressed using the `&/2` meta-predicate as the constructor to represent the parallel execution of goals. In this way, the conjunction of goals (i.e., literals) `p & q` in the body of a clause will trigger the execution of goals `p` and `q` in parallel, finishing when both executions finish.

Given a program $\mathcal{P}$ and a predicate $\mathtt{p} \in \mathcal{P}$ of arity k and a set Π of k-tuples of calling data to $\mathtt{p}$, we refer to the *(standard) cost* of a call $\mathtt{p}(\bar{\mathtt{e}})$ (i.e., a call to $\mathtt{p}$ with actual data $\bar{\mathtt{e}} \in \Pi$), as the resource usage (under a given cost metric) of the complete execution of $\mathtt{p}(\bar{\mathtt{e}})$. The *standard cost* is formalized as a function $\mathcal{C}_\mathtt{p} : \Pi \to \mathcal{R}_\infty$, where $\mathcal{R}_\infty$ is the set of real numbers augmented with the special symbol ∞ (which is used to represent non-termination). We extend the function $\mathcal{C}_\mathtt{p}$ to the powerset of Π, i.e., $\hat{\mathcal{C}}_\mathtt{p} : 2^\Pi \to 2^{\mathcal{R}_\infty}$, where $\hat{\mathcal{C}}_\mathtt{p}(E) = \{\mathcal{C}_\mathtt{p}(\bar{\mathtt{e}}) \mid \bar{\mathtt{e}} \in E\}$. Our goal is to abstract (safely approximate, as accurately as possible) $\hat{\mathcal{C}}_\mathtt{p}$ (note that $\mathcal{C}_\mathtt{p}(\bar{\mathtt{e}}) = \hat{\mathcal{C}}_\mathtt{p}(\{\bar{\mathtt{e}}\})$). Intuitively, this abstraction is the composition of two abstractions: a size abstraction and a cost abstraction. The goal of the analysis is to infer two functions $\hat{\mathcal{C}}^{\downarrow}_\mathtt{p}$ and $\hat{\mathcal{C}}^{\uparrow}_\mathtt{p} : \mathcal{N}^m_\top \to \mathcal{R}_\infty$ that give lower and upper bounds respectively on the cost function $\hat{\mathcal{C}}_\mathtt{p}$, where $\mathcal{N}^m_\top$ is the set of m-tuples whose elements are natural numbers or the special symbol $\top$, meaning that the size of a given term under a given size metric is *undefined*. Such bounds are given as a function of tuples of data sizes (representing the concrete tuples of data of the concrete function $\hat{\mathcal{C}}_\mathtt{p}$). Typical size metrics are the actual value of a number, the length of a list, the size (number of constant and function symbols) of a term, etc. [24,26].

We now enumerate different metrics used to evaluate the performance of parallel logic programs, compared against its corresponding sequential version [27]. Here, these metrics are parameterized with respect to the resource in which the cost is expressed (e.g., number of *resolution steps*, *execution time*, or *energy consumption*):

– **Sequential cost (*Work*)**: It is the standard cost of executing a program, assuming no parallelism.
– **Parallel cost (*Depth*)**: It is the cost of executing a program in parallel, considering an unbounded number of processors.
– **Maximum number of processes running in parallel**: It is the maximum number of processes that may run simultaneously in a program. This is useful to determine what is the minimum number of processors that are required to guarantee that all the processes run in parallel.

The following example illustrates our approach.

Example 1. Consider the predicate `scalar/3` below, and a calling mode to it with the first argument bound to an integer n and the second one bound to a list of integers $[x_1, x_2, \ldots, x_k]$. Upon success, the third argument is bound to the list of products $[n \cdot x_1, n \cdot x_2, \ldots, n \cdot x_k]$. Each product is recursively computed by predicate `mult/3`. The calling modes are automatically inferred by CiaoPP (see [13] and its references): the first two arguments of both predicates are input, and their last arguments are output.

```
scalar(_,[],[]).
scalar(N,[X|Xs],[Y|Ys]):-
   mult(N,X,Y) & scalar(N,Xs,Ys).
```

```
mult(0,_,0).
mult(N,X,Y):-
   N>1,
   N1 is N - 1,
   mult(N1,X,Y0),
   Y is Y0 + X.
```

The call to the parallel `&/2` operator in the body of the second clause of `scalar/3` causes the calls to `mult/3` and `scalar/3` to be executed in parallel. We want to infer the cost of such a call to `scalar/3`, in terms of the number of resolution steps, as a function of its input data sizes. We use the CiaoPP system to infer size relations for the different arguments in the clauses, as well as dealing with a rich set of size metrics (see [24,26] for details). Assume that the size metrics used in this example are the *actual value* of `N` (denoted `int(N)`), for the first argument, and the *list-length* for the second and third arguments (denoted `length(X)` and `length(Y)` respectively). Since size relations are obvious in this example, we focus only on the setting up of cost relations for the sake of brevity. Regarding the number of solutions, in this example all the predicates generate at most one solution. For simplicity we assume that all builtin predicates, such as `is/2` and the comparison operators have zero cost (in practice they have a "trust"assertion that specifies their cost as if it had been inferred by the system). As the program contains parallel calls, we are interested in inferring both total resolution steps, i.e., considering a sequential execution (represented by the `seq` identifier), and the number of parallel steps, considering a parallel execution, with an unbounded number of processors (represented by `par`). In the latter case, the definition of this resource establishes that the aggregator of the costs of the parallel calls that are arguments of the `&/2` meta-predicate is the `max/2` function. Thus, the number of resolution steps performed in parallel for `p & q` is the maximum between the parallel steps performed by `p` and the ones performed by `q`. However, for computing the *total resolution steps*, the aggregation operator we use is the addition, both for parallel and sequential calls. For brevity, in this example we only infer upper bounds on resource usages.

We now set up the cost relations for `scalar/3` and `mult/3`. Note that the cost functions have two arguments, corresponding to the sizes of the input arguments.[5] In the equations, we underline the operation applied as cost aggregator for `&/2`.

For the sequential execution (`seq`), we obtain the following cost relations:

$$\begin{aligned}
\mathtt{C}_{\mathtt{scalar}}(n,l) &= 1 && \text{if } l = 0\\
\mathtt{C}_{\mathtt{scalar}}(n,l) &= 1 + \mathtt{C}_{\mathtt{mult}}(n) \underline{+} \mathtt{C}_{\mathtt{scalar}}(n,l-1) && \text{if } l > 0\\
\mathtt{C}_{\mathtt{mult}}(n) &= 1 && \text{if } n = 0\\
\mathtt{C}_{\mathtt{mult}}(n) &= 1 + \mathtt{C}_{\mathtt{mult}}(n-1) && \text{if } n > 0
\end{aligned}$$

[5] For the sake of clarity, we abuse notation in the examples when representing the cost functions that depend on data sizes.

After solving these equations and composing the closed-form solutions, we obtain the following closed-form functions:

$$\begin{array}{rl} \mathtt{C}_{\mathtt{scalar}}(n,l) = (n+2)\times l+1 & \text{if } n \geq 0 \wedge l \geq 0 \\ \mathtt{C}_{\mathtt{mult}}(n) = n+1 & \text{if } n \geq 0 \end{array}$$

For the parallel execution (`par`), we obtain the following cost relations:

$$\begin{array}{rl} \mathtt{C}_{\mathtt{scalar}}(n,l) = 1 & \text{if } l = 0 \\ \mathtt{C}_{\mathtt{scalar}}(n,l) = 1 + \underline{\underline{max}}(\mathtt{C}_{\mathtt{mult}}(n), \mathtt{C}_{\mathtt{scalar}}(n,l-1)) & \text{if } l > 0 \\ \mathtt{C}_{\mathtt{mult}}(n) = 1 & \text{if } n = 0 \\ \mathtt{C}_{\mathtt{mult}}(n) = 1 + \mathtt{C}_{\mathtt{mult}}(n-1) & \text{if } n > 0 \end{array}$$

Similarly, we obtain the following closed-form functions:

$$\begin{array}{rl} \mathtt{C}_{\mathtt{scalar}}(n,l) = n+l+1 & \text{if } n \geq 0 \wedge l \geq 0 \\ \mathtt{C}_{\mathtt{mult}}(n) = n+1 & \text{if } n \geq 0 \end{array}$$

By comparing the complexity order (in terms of resolution steps) of the sequential execution of `scalar/3`, $O(n \cdot l)$, with the complexity order of its parallel execution (assuming an ideal parallel model with an unbounded number of processors) $O(n+l)$, we can get a hint about the maximum achievable parallelization of the program.

Another useful piece of information about `scalar/3` that we want to infer is the maximum number of processes that may run in parallel, considering all possible executions. For this purpose, we define a resource named `sthreads`. The operation `count_process/3` aggregates the cost of both arguments of the meta-predicate `&/2` for the `sthreads` resource, by adding the maximum number of processes for each argument plus one additional process, corresponding to the one created by the call to `&/2`. The sequential cost aggregator is now the *maximum* operator, in order to keep track of the maximum number of processes created along the different instructions of the program executed sequentially. Note that if the instruction `p` executes at most Pr_p processes in parallel, and the instruction `q` executes at most Pr_q processes, then the program `p, q` will execute at most $max(Pr_p, Pr_q)$ processes in parallel, because all the parallel processes created by `p` will finish before the execution of `q`. Note also that for the sequential execution of both `p` and `q`, the cost in terms of the `sthreads` resource is always zero, because no additional process is created. The analysis sets up the following recurrences for the `sthreads` resource and the predicates `scalar/3` and `mult/3` of our example:

$$\begin{array}{rl} \mathtt{C}_{\mathtt{scalar}}(n,l) = 0 & \text{if } l = 0 \\ \mathtt{C}_{\mathtt{scalar}}(n,l) = \mathtt{C}_{\mathtt{mult}}(n) + \mathtt{C}_{\mathtt{scalar}}(n,l-1) + 1 & \text{if } l > 0 \\ \mathtt{C}_{\mathtt{mult}}(n) = 0 & \text{if } n \geq 0 \end{array}$$

For which we obtain the following closed-form functions:

$$\begin{array}{rl} \mathtt{C}_{\mathtt{scalar}}(n,l) = l & \text{if } n \geq 0 \wedge l \geq 0 \\ \mathtt{C}_{\mathtt{mult}}(n) = 0 & \text{if } n \geq 0 \end{array}$$

As we can see, this predicate will execute, in the worst case, as many processes as there are elements in the input list.

3 The Parametric Cost Relations Framework for Sequential Programs

The starting point of our work is the standard general framework described in [24] for setting up parametric relations representing the resource usage (and size relations) of programs and predicates.[6] The analysis infers size relations for each predicate in a program: arithmetic expressions that provide the size of output arguments of the predicate as a function of its input data sizes. It also infers size relations for each clause, which give the input data sizes of the body literals as functions of the input data sizes to the clause head. Such size relations are instrumental for setting up cost relations.

The framework is doubly parametric: first, the costs inferred are functions of input data sizes, and second, the framework itself is parametric with respect to the type of approximation made (upper or lower bounds), and to the resource analyzed. Each concrete resource r to be tracked is defined by two sets of (user-provided) functions, which can be constants, or general expressions of input data sizes:

1. *Head cost* $\varphi_{[ap,r]}(H)$: a function that returns an approximation of type ap of the amount of resource r used by the unification of the calling literal (subgoal) $\mathtt{p}$ and the head H of a clause matching $\mathtt{p}$, plus any preparation for entering a clause (i.e., call and parameter passing cost).
2. *Predicate cost* $\Psi_{[ap,r]}(\mathtt{p}, \bar{\mathtt{x}})$: it is also possible to define the *full cost* for a particular predicate $\mathtt{p}$ for resource r and approximation ap, i.e., the function $\Psi_{[ap,r]}(\mathtt{p}) : \mathcal{N}_\top^m \rightarrow \mathcal{R}_\infty$ (with the sizes of $\mathtt{p}$'s input data as parameters, $\bar{\mathtt{x}}$) that returns the usage of resource r made by a call to this predicate. This is especially useful for built-in or external predicates, i.e., predicates for which the source code is not available and thus cannot be analyzed, or for providing a more accurate function than analysis can infer. In the implementation, this information can be provided by the user to the analyzer through *trust assertions*.

For simplicity we only show the equations related to our standard definition of cost. However, our framework has also been extended to allow the inference of a more general definition of cost, called accumulated cost, which is useful for performing static profiling, obtaining more detailed information regarding how the cost is distributed among a set of user-defined *cost centers*. See [11,21] for more details. In order to infer the resource usage functions, all predicates in the program are processed in a single traversal of the call graph in reverse topological order. Consider a predicate $\mathtt{p}$ defined by clauses $C_1, \ldots, C_m$. Assume $\bar{\mathtt{x}}$ are the sizes of $\mathtt{p}$'s input parameters. Then, the resource usage (expressed in units of resource r with approximation ap) of a call to $\mathtt{p}$, for an input of size $\bar{\mathtt{x}}$, denoted as $C_{pred[ap,r]}(\mathtt{p}, \bar{\mathtt{x}})$, can be expressed as:

$$C_{pred[ap,r]}(\mathtt{p}, \bar{\mathtt{x}}) = \bigodot_{1 \leq i \leq m} (C_{cl[ap,r]}(C_i, \bar{\mathtt{x}})) \tag{1}$$

[6] We give equivalent but simpler descriptions than in [24], which are allowed by assuming that programs are the result of a normalization process that makes all unifications explicit in the clause body, so that the arguments of the clause head and the body literals are all unique variables. We also change some notation for readability and illustrative purposes.

where $\bigodot = ClauseAggregator(ap, r)$ is a function that takes an approximation identifier ap and returns a function that applies over the cost of all the clauses, $C_{cl[ap,r]}(C_i, \bar{\mathtt{x}})$, for $1 \leq i \leq m$, in order to obtain the cost of a call to the predicate p. For example, if ap is the identifier for approximation "upper bound" (ub), then a possible conservative definition for $ClauseAggregator(ub, r)$ is the $\sum$ function. In this case, and since the number of solutions generated by a predicate that will be demanded is generally not known in advance, a conservative upper bound on the computational cost of a predicate is obtained by assuming that all solutions are needed, and that all clauses are executed (thus the cost of the predicate is assumed to be the sum of the costs of all of its clauses). However, it is straightforward to take mutual exclusion into account to obtain a more precise estimate of the cost of a predicate, using the maximum of the costs of mutually exclusive groups of clauses, as done in [26].

Let us see now how to compute the resource usage of a clause. Consider a clause C of predicate p of the form H :- $L_1, \ldots, L_k$ where $L_j, 1 \leq j \leq k$, is a literal (either a predicate call, or an external or builtin predicate), and H is the clause head. Assume that $\psi_j(\bar{\mathtt{x}})$ is a tuple with the sizes of all the input arguments to literal L_j, given as functions of the sizes of the input arguments to the clause head. Note that these $\psi_j(\bar{\mathtt{x}})$ size relations have previously been computed during size analysis for all input arguments to literals in the bodies of all clauses. Then, the cost relation for clause C and a single call to p (obtaining all solutions), is:

$$C_{cl[ap,r]}(C, \bar{\mathtt{x}}) = \varphi_{[ap,r]}(H) + \sum_{j=1}^{lim(ap,C)} sols_j(\bar{\mathtt{x}}) \times C_{lit[ap,r]}(L_j, \psi_j(\bar{\mathtt{x}})) \tag{2}$$

where $lim(ap, C)$ gives the index of the last body literal that is called in the execution of clause C, and $sols_j$ represents the product of the number of solutions produced by the predecessor literals of L_j in the clause body:

$$sols_j(\bar{\mathtt{x}}) = \prod_{i=1}^{j-1} s_{pred}(L_i, \psi_i(\bar{\mathtt{x}})) \tag{3}$$

where $s_{pred}(L_i, \psi_i(\bar{\mathtt{x}}))$ gives the number of solutions produced by L_i, with arguments of size $\psi_i(\bar{\mathtt{x}})$. The number of solutions and size relations are both inferred automatically by the framework (we refer the reader to [8,7,9,26] for a description).

Finally, $C_{lit[ap,r]}(L_j, \psi_j(\bar{\mathtt{x}}))$ is replaced by one of the following expressions, depending on L_j:

- If L_j is a call to a predicate q which is in the same strongly connected component as p (the predicate under analysis), then $C_{lit[ap,r]}(L_j, \psi_j(\bar{\mathtt{x}}))$ is replaced by the symbolic call $C_{pred[ap,r]}(\mathtt{q}, \psi_{\mathtt{j}}(\bar{\mathtt{x}}))$, giving rise to a recurrence relation that needs to be bounded with a closed-form expression by the solver afterwards.
- If L_j is a call to a predicate q which is in a different strongly connected component than p, then $C_{lit[ap,r]}(L_j, \psi_j(\bar{\mathtt{x}}))$ is replaced by the closed-form expression that bounds $C_{pred[ap,r]}(\mathtt{q}, \psi_{\mathtt{j}}(\bar{\mathtt{x}}))$. The analysis guarantees that this expression has been inferred beforehand, due to the fact that the analysis is performed for each strongly connected component, in a reverse topological order.

– If L_j is a call to a predicate q, whose cost is specified (with a trust assertion) as $\Psi_{[ap,r]}(q, \bar{y})$, then $C_{lit[ap,r]}(L_j, \psi_j(\bar{x}))$ is replaced by the expression $\Psi_{[ap,r]}(q, \psi_j(\bar{x}))$.

4 Our Extended Resource Analysis Framework for Parallel Programs

In this section, we describe how we extend the resource analysis framework detailed above, in order to handle logic programs with Independent And-Parallelism, using the binary parallel `&/2` operator. First, we introduce a new general parameter that indicates the execution model the analysis has to consider. For our current prototype, we have defined two different execution models: standard *sequential* execution, represented by *seq*, and an abstract parallel execution model, represented by $par(n)$, where $n \in \mathcal{N} \cup \{\infty\}$. The abstract execution model $par(\infty)$ is similar to the *work* and *depth* model, presented in [6] and used extensively in previous work such as [16]. Basically, this model is based on considering an unbounded number of available processors to infer bounds on the depth of the computation tree. The *work* measure is the amount of work to be performed considering a sequential execution. These two measures together give an idea on the impact of the parallelization of a particular program. The abstract execution model $par(n)$, where $n \in \mathcal{N}$, assumes a finite number n of processors.

In order to obtain the cost of a predicate, equation (1) remains almost identical, the only difference being the addition of the new parameter to indicate the execution model.

Now we address how to set up the cost for clauses. In this case, equation (2) is extended with the execution model ex, and also the default sequential cost aggregation, $\sum$, is replaced by a parametric associative operator $\bigoplus$, that depends on the resource being defined, the approximation, and the execution model. For $ex \equiv par(\infty)$ or $ex \equiv seq$, the following equation is set up:

$$C_{cl[ap,r,ex]}(C, \bar{x}) = \varphi_{[ap,r]}(H) + \bigoplus_{j=1}^{lim(ap,ex,C)} (sols_j(\bar{x}) \times C_{lit[ap,r,ex]}(L_j, \psi_j(\bar{x}))) \quad (4)$$

Note that the cost aggregation operators must depend on the resource r (besides the other parameters). For example, if r is *execution time*, then the cost of executing two tasks in parallel must be aggregated by taking the maximun of the execution times of the two tasks. In contrast, if r is *energy consumption*, then the aggregation is the addition of the energy of the two tasks.

Finally, we extend how the cost of a literal L_i, expressed as $C_{lit[ap,r,ex]}(L_i, \psi_i(\bar{x}))$, is set up. The previous definition is extended considering the new case where the literal is a call to the *meta-predicate* `&/2`. In this case, we introduce a new parallel aggregation associative operator, denoted by $\bigotimes$. Concretely, if $L_i = B_1 \& B_2$, where B_1 and B_2 are two sequences of goals, then:

$$C_{lit[ap,r,ex]}(B_1 \& B_2, \bar{x}) = C_{body[ap,r,ex]}(B_1, \bar{x}) \bigotimes C_{body[ap,r,ex]}(B_2, \bar{x}) \quad (5)$$

$$C_{body[ap,r,ex]}(B,\bar{\mathtt{x}}) = \bigoplus_{j=1}^{lim(ap,ex,B)} (sols_j(\bar{\mathtt{x}}) \times C_{lit[ap,r,ex]}(L_j^B, \psi_j(\bar{\mathtt{x}}))) \quad (6)$$

where $B = L_1^B, \ldots, L_m^B$.

Consider now the execution model $ex \equiv par(n)$, where $n \in \mathcal{N}$ (i.e., assuming a finite number n of processors), and a recursive parallel predicate $\mathtt{p}$ that creates a parallel task $\mathtt{q}_i$ in each recursion i. Assume that we are interested in obtaining an upper bound on the cost of a call to $\mathtt{p}$, for an input of size $\bar{\mathtt{x}}$. We first infer the number k of parallel tasks created by $\mathtt{p}$ as a function of $\bar{\mathtt{x}}$. This can be easily done by using our cost analysis framework and providing the suitable assertions for inferring a resource named "*ptasks*." Intuitively, the "counter" associated to such resource must be incremented by the (symbolic) execution of the `&/2` parallel operator. More formally, $k = C_{pred[ub,ptasks]}(\mathtt{p}, \bar{\mathtt{x}})$. To this point, an upper bound m on the number of tasks executed by any of the n processors is given by $m = \lceil \frac{k}{n} \rceil$. Then, an upper bound on the cost (in terms of resolution steps, i.e., $r = steps$) of a call to $\mathtt{p}$, for an input of size $\bar{\mathtt{x}}$ can be given by:

$$C_{pred[ub,r,par(n)]}(\mathtt{p}, \bar{\mathtt{x}}) = C^u + Spaw^u \quad (7)$$

where C^u can be computed in two possible ways: $C^u = \sum_{i=1}^{m} C_i^u$; or $C^u = m\ C_1^u$, where C_i^u denotes an upper bound on the cost of parallel task $\mathtt{q}_i$, and $C_1^u, \ldots, C_k^u$ are ordered in descending order of cost. Each C_i^u can be considered as the sum of two components: $C_i^u = Sched_i^u + T_i^u$, where $Sched_i^u$ denotes the cost from the point in which the parallel subtask $\mathtt{q}_i$ is created until its execution is started by a processor (possibly the same processor that created the subtask), i.e. the cost of task preparation, scheduling, communication overheads, etc. T_i^u denotes the cost of the execution of $\mathtt{q}_i$ disregarding all the overheads mentioned before, i.e., $T_i^u = C_{pred[ub,r,seq]}(\mathtt{q}, \psi_q(\bar{\mathtt{x}}))$, where $\psi_q(\bar{\mathtt{x}})$ is a tuple with the sizes of all the input arguments to predicate $\mathtt{q}$ in the body of $\mathtt{p}$. $Spaw^u$ denotes an upper bound on the cost of creating the k parallel tasks $\mathtt{q}_i$. It will be dependent on the particular system in which $\mathtt{p}$ is going to be executed. It can be a constant, or a function of several parameters, (such as input data size, number of input arguments, or number of tasks) and can be experimentally determined.

4.1 Solving Cost Recurrence Relations Involving *max* Operation

We propose a method for finding closed-form functions for cost relations that use the parallel and sequential cost aggregation operators $\bigotimes$ and $\bigoplus$, which include the *max* function in their definitions.

Automatically finding closed-form upper and lower bounds for recurrence relations is an uncomputable problem. For some special classes of recurrences, exact solutions are known, for example for linear recurrences with one variable. For some other classes, it is possible to apply transformations to fit a class of recurrences with known solutions, even if this transformation obtains an appropriate approximation rather than an equivalent expression.

Particularly for the case of analyzing independent and-parallel logic programs, recurrences involving the *max* operator are quite common. For example, if we are analyzing elapsed time of a parallel logic program, a proper parallel aggregation operator is the maximum between the times elapsed for each literal running in

parallel. To the best of our knowledge, no general solution exists for recurrences of this particular type. However, in this paper we identify some common classes of this type of recurrences, for which we obtain closed forms that are proven to be correct. In this section, we present these different classes, together with the corresponding method to obtain a correct bound.

Consider the following function $f : \mathcal{N}^m \to \mathcal{N}$, defined as a general form of a first-order recurrence equation with a *max* operator:

$$f(\bar{\mathtt{x}}) = \begin{cases} max(C, f(\bar{\mathtt{x}}_{|i} - 1)) + D & x_i > a \\ B & x_i \leq a \end{cases} \tag{8}$$

where $a \in \mathcal{N}$, and C, D and B are arbitrary expressions possibly depending on $\bar{\mathtt{x}}$. Note that $\bar{\mathtt{x}} = x_1, x_2, \ldots, x_m$. We define $\bar{\mathtt{x}}_{|i} - 1 = x_1, \ldots, x_i - 1, \ldots, x_m$, for a given i, $1 \leq i \leq m$. If C and D do not depend on x_i, then C and D do not change through the different recursive instances of f. In this case, an equivalent closed form is defined by the following theorem:

Theorem 1. *Given $f : \mathcal{N}^m \to \mathcal{N}$ as defined in (8), where C and D are functions of $\bar{\mathtt{x}} \setminus x_i$. Then, $\forall \bar{\mathtt{x}}$:*

$$f(\bar{\mathtt{x}}) = f^{'}(\bar{\mathtt{x}}) = \begin{cases} max(C, B) + (x_i - a) \cdot D & x_i > a \\ B & x_i \leq a \end{cases}$$

For the case where $C = g(\bar{\mathtt{x}})$ and $D = h(\bar{\mathtt{x}})$ are functions non-decreasing on x_i, then the upper bound is given by the following closed form:

Theorem 2. *Given $f : \mathcal{N}^m \to \mathcal{N}$ as defined in (8), where g and h are functions of $\bar{\mathtt{x}}$, non-decreasing on x_i. Then, $\forall \bar{\mathtt{x}}$:*

$$f(\bar{\mathtt{x}}) \leq f'(\bar{\mathtt{x}}) = \begin{cases} max(g(\bar{\mathtt{x}}), B) + (x_i - a - 1) \times max(g(\bar{\mathtt{x}}), h(\bar{\mathtt{x}}_{|i} - 1)) + h(\bar{\mathtt{x}}) & x_i > a \\ B & x_i \leq a \end{cases}$$

The proofs of both theorems are available in [18]. If the recurrence is not included in the classes defined by Theorems (1) and (2), we try to eliminate the *max* operator by simplification. Consider an expression $max(e_1, e_2)$ appearing in a recurrence relation. First, we use the function comparison capabilities of CiaoPP, presented in [19,20]. If e_i contains non-closed recurrence function calls, we use an SMT solver [23] representing non-closed functions as uninterpreted functions, assuming that they are positive and non-decreasing. Concretely, for each non-closed function call $f(\bar{\mathtt{x}})$ appearing in e_i, we add the properties $\forall \bar{\mathtt{x}}. f(\bar{\mathtt{x}}) \geq 0$ and $\forall \bar{\mathtt{x}}, \bar{\mathtt{y}}. \bar{\mathtt{x}} \leq \bar{\mathtt{y}} \iff f(\bar{\mathtt{x}}) \leq f(\bar{\mathtt{y}})$ to a set M. Then, we check if either $M \models e_1 \leq e_2$ or $M \models e_2 \leq e_1$ hold.[7]

Finally, if no proof is found, we replace the *max* operator with an addition, losing precision but still finding safe upper bounds.

[7] As the algorithm used by SMT solvers in this case is not guaranteed to terminate, we set a timeout.

Table 1. Description of the benchmarks.

`map_add1/2`	Parallel increment by one of each element of a list.
`fib/2`	Parallel computation of the nth Fibonacci number.
`add_mat/3`,`mmatrix/3`	Parallel matrix multiplication and addition.
`blur/2`	Generic parallel image filter.
`intersect/3`, `union/3`, `diff/3`	Set operations.
`dyade/3`, `dyade_map/3`	Dyadic product of two vectors (and on a set of vectors).
`append_all/3`	Appends a prefix to each list of a list of lists.

5 Implementation and Experimental Results

We have implemented a prototype of our approach, leveraging the existing resource usage analysis framework of CiaoPP. The implementation basically consists of the parameterization of the operators used for sequential and parallel cost aggregation, i.e., for the aggregation of the costs corresponding to the arguments of `,/2` and `&/2`, respectively. This allows the user to define resources in a general way, taking into account the underlying execution model. We use off-the-shelf Computer Algebra Systems, as well as a builtin recurrence solver extended with the techniques presented in this paper, in order to solve recurrence relations that arise during analysis. We also use an external SMT Solver (Z3 [23]), for the simplification of some recurrences with a `max` operator.

We selected a set of benchmarks that exhibit different common parallel patterns, briefly described in Table 1, together with the definition of a set of resources that help understand the overall behavior of the parallelization. Table 2 shows some results of the experiments that we have performed with our prototype implementation. Column **Bench** shows the main predicates analyzed for each benchmark. Set operations (`intersect`, `union` and `diff`), as well as the programs `append_all`, `dyade` and `add_mat`, are Prolog versions of the benchmarks analyzed in [16], which is the closest related work we are aware of. Column **Res** indicates the name of each of the resources inferred for each benchmark: *sequential resolution steps* (**SCost**), *parallel resolution steps* assuming an unbounded number of processors (**PCost**), and *maximum number of processes executing in parallel* (**SThreads**). The latter gives an indication of the maximum parallelism that can potentially be exploited. We are considering a resolution step as the overhead of spawning a new thread. Column **Bound Inferred** shows the upper bounds obtained for each of the resources indicated in Column **Res**. While in the experiments both upper and lower bounds were inferred, for the sake of brevity, we only show upper-bound functions. Column **BigO** shows the complexity order, in big O notation, corresponding to each resource. For all the benchmarks in Table 2 we obtain the exact complexity orders. We also obtain the same complexity order as in [16] for the Prolog versions of the benchmarks taken from that work. Finally, Column $\mathbf{T_A}$(**ms**) shows the analysis times in milliseconds. The results show that most of the benchmarks have different asymptotic behavior in the sequential and parallel execution models. In particular, for `fib(x)`, the analysis infers an exponential upper bound for sequential execution steps, and a linear upper bound for parallel execution steps. As mentioned before, this is an upper bound for an ideal case, assuming an unbounded number of processors. Nevertheless, such upper-bound information is useful for understanding

Table 2. Resource usage inferred for Independent And-Parallel Programs.

Bench	Res	Bound Inferred	BigO	$\mathbf{T_A}$(ms)
map_add1(x)	SCost	$2 \cdot l_x + 1$	$\mathcal{O}(l_x)$	31.17
	PCost	$2 \cdot l_x + 1$	$\mathcal{O}(l_x)$	
	SThreads	l_x	$\mathcal{O}(l_x)$	
fib(x)	SCost	$F(i_x) + L(i_x) - 1$	$\mathcal{O}(2^{i_x})$	127.81
	PCost	$2 \cdot i_x + 1$	$\mathcal{O}(i_x)$	
	SThreads	$F(i_x) + L(i_x) - 1$	$\mathcal{O}(2^{i_x})$	
mmatrix(m_1, n_1, m_2, n_2)	SCost	$i_{n_2} \cdot i_{m_2} \cdot i_{m_1} + 2 \cdot i_{m_2} \cdot i_{m_1} + 2 \cdot i_{m_1} + 1$	$\mathcal{O}(i_{n_2} \cdot i_{m_2} \cdot i_{m_1})$	194.45
	PCost	$i_{n_2} + 2 \cdot i_{m_2} + 2 \cdot i_{m_1} + 1$	$\mathcal{O}(i_{n_1} + i_{m_1} + i_{m_2})$	
	SThreads	$i_{m_2} \cdot i_{m_1} + i_{m_1}$	$\mathcal{O}(i_{m_2} \cdot i_{m_1})$	
blur(m,n)	SCost	$2 \cdot i_m \cdot i_n + 2 \cdot i_n + 1$	$\mathcal{O}(i_m \cdot i_n)$	126.63
	PCost	$2 \cdot i_m + 2 \cdot i_n + 1$	$\mathcal{O}(i_m + i_n)$	
	SThreads	i_n	$\mathcal{O}(i_n)$	
add_mat(m,n)	SCost	$i_m \cdot i_n + 2 \cdot i_n + 1$	$\mathcal{O}(i_m \cdot i_n)$	128.93
	PCost	$i_m + 2 \cdot i_n + 1$	$\mathcal{O}(i_m + i_n)$	
	SThreads	i_n	$\mathcal{O}(i_n)$	
intersect(a, b)	SCost	$l_a \cdot l_b + 3 \cdot l_a + 3$	$\mathcal{O}(l_a \cdot l_b)$	233.14
	PCost	$l_b + 3 \cdot l_a + 3$	$\mathcal{O}(l_a + l_b)$	
	SThreads	l_a	$\mathcal{O}(l_a)$	
union(a, b)	SCost	$l_a \cdot l_b + 3 \cdot l_a + 3$	$\mathcal{O}(l_a \cdot l_b)$	218.31
	PCost	$2 \cdot l_b + 3 \cdot l_a + 3$	$\mathcal{O}(l_a + l_b)$	
	SThreads	l_a	$\mathcal{O}(l_a)$	
diff(a, b)	SCost	$l_a \cdot l_b + 3 \cdot l_a + 3$	$\mathcal{O}(l_a \cdot l_b)$	232.55
	PCost	$l_b + 3 \cdot l_a + 3$	$\mathcal{O}(l_a + l_b)$	
	SThreads	l_a	$\mathcal{O}(l_a)$	
dyade(a, b)	SCost	$l_a \cdot l_b + 2 \cdot l_a + 1$	$\mathcal{O}(l_a \cdot l_b)$	82.71
	PCost	$l_b + 2 \cdot l_a + 1$	$\mathcal{O}(l_a + l_b)$	
	SThreads	l_a	$\mathcal{O}(l_a)$	
dyade_map(l, m)	SCost	$i_{max(m)} \cdot l_m \cdot l_l + 2 \cdot l_m \cdot l_l + 2 \cdot l_m + 1$	$\mathcal{O}(i_{max(m)} \cdot l_m \cdot l_l)$	177.91
	PCost	$i_{max(m)} + 2 \cdot l_m + 2 \cdot l_l + 1$	$\mathcal{O}(i_{max(m)} + l_m + l_l)$	
	SThreads	$l_l \cdot l_m + l_l$	$\mathcal{O}(l_m \cdot l_l)$	
append_all(l, m)	SCost	$l_l \cdot l_m + 2 \cdot l_m + 1$	$\mathcal{O}(l_l \cdot l_m)$	81.97
	PCost	$l_l + 2 \cdot l_m + 1$	$\mathcal{O}(l_l + l_m)$	
	SThreads	l_m	$\mathcal{O}(l_m)$	

$F(n)$, $L(n)$ represent the nth. element of the Fibonacci sequence and the nth. Lucas number, respectively. l_n, i_n represent the size of n in terms of the metrics *length* and *int*, respectively.

Table 3. Resource usage inferred for a bounded number of processors.

Bench	Bound Inferred	BigO	$\mathbf{T_A}$(ms)
map_add1(x)	$2 \cdot \lceil \frac{l_x}{p} \rceil + 1$	$\mathcal{O}(\lceil \frac{l_x}{p} \rceil)$	54.36
blur(m,n)	$2 \cdot \lceil \frac{i_n}{p} \rceil \cdot i_m + 2 \cdot \lceil \frac{i_n}{p} \rceil + 1$	$\mathcal{O}(\lceil \frac{i_n}{p} \rceil \cdot i_m)$	205.97
add_mat(m,n)	$\lceil \frac{i_n}{p} \rceil \cdot i_m + 2 \cdot \lceil \frac{i_n}{p} \rceil + 1$	$\mathcal{O}(\lceil \frac{i_n}{p} \rceil \cdot i_m)$	185.89
intersect(a, b)	$\lceil \frac{l_a}{p} \rceil \cdot l_b + 2 \cdot \lceil \frac{l_a}{p} \rceil + l_a + 2$	$\mathcal{O}(\lceil \frac{l_a}{p} \rceil \cdot l_b)$	330.47
union(a, b)	$\lceil \frac{l_a}{p} \rceil \cdot l_b + 2 \cdot \lceil \frac{l_a}{p} \rceil + l_a + l_b + 2$	$\mathcal{O}(\lceil \frac{l_a}{p} \rceil \cdot l_b)$	311.3
diff(a, b)	$\lceil \frac{l_a}{p} \rceil \cdot l_b + 2 \cdot \lceil \frac{l_a}{p} \rceil + l_a + 2$	$\mathcal{O}(\lceil \frac{l_a}{p} \rceil \cdot l_b)$	339.01
dyade(a, b)	$\lceil \frac{l_a}{p} \rceil \cdot l_b + 2 \cdot \lceil \frac{l_a}{p} \rceil + 1$	$\mathcal{O}(\lceil \frac{l_a}{p} \rceil \cdot l_b)$	120.93
append_all(l, m)	$\lceil \frac{l_m}{p} \rceil \cdot l_l + 2 \cdot \lceil \frac{l_m}{p} \rceil + 1$	$\mathcal{O}(\lceil \frac{l_m}{p} \rceil \cdot l_l)$	117.8

p is defined as the minimum between the number of processors and **SThreads**.

how the cost behavior evolves in architectures with different levels of parallelism. In addition, this *dual* cost measure can be combined together with a bound on the number of processors in order to obtain a general asymptotic upper bound (see for example Brent's Theorem [12], which is also mentioned in [16]). The program `map_add1(l)` exhibits a different behavior: both sequential and parallel upper bounds are linear. This happens because we are considering *resolution steps*, i.e., we are counting each head unification produced from an initial call `map_add1(l)`. Even under the parallel execution model, we have a chain of head unifications whose

length depends linearly on the length of the input list. It follows from the results of this particular case that this simple, non-associative parallelization will not be useful for improving the number of resolution steps performed in parallel.

Another useful information inferred in our experiments is the maximum number of processes that can be executed in parallel, represented by the resource named **SThreads**. We can see that for most of our examples the analysis obtains a linear upper bound for this resource, in terms of the size of some of the inputs. For example, the execution of `intersect(a,b)` (parallel set intersection) will create *at most* l_a processes, where l_a represents the length of the list a. For other examples, the analysis shows a quadratic upper bound (as in `mmatrix`), or even exponential bounds (as in `fib`). The information about upper bounds on the maximum level of parallelism required by a program is useful for understanding its scalability in different parallel architectures, or for optimizing the number of processors that a particular call will use, depending on the size of the input data.

Finally, the results of our experiments considering a bounded number of processors are shown in Table 3.

6 Related Work

Our approach is an extension of an existing cost analysis framework for sequential logic programs [9,11,20], which extends the classical cost analysis techniques based on setting up and solving recurrence relations, pioneered by [29], with solutions for relations involving `max` and `min` functions. The framework handles characteristics such as backtracking, multiple solutions (i.e., non-determinism), failure, and inference of both upper and lower bounds including non-polynomial bounds. These features are inherited by our approach, and are absent from other approaches to parallel cost analysis in the literature.

The most closely-related work to our approach is [16], which describes an automatic analysis for deriving bounds on the worst-case evaluation cost of first order functional programs. The analysis derives bounds under an abstract *dual* cost model based on two measures: *work* and *depth*, which over-approximate the sequential and parallel evaluation cost of programs, respectively, considering an unlimited number of processors. Such an abstract cost model was introduced by [6] to formally analyze parallel programs. The work is based on type judgments annotated with a cost metric, which generate a set of inequalities which are then solved by linear programming techniques. Their analysis is only able to infer multivariate resource polynomial bounds, while non-polynomial bounds are left as future work. In [15] the authors propose an automatic analysis based on the *work* and *depth* model, for a simple imperative language with explicit parallel loops.

There are other approaches to cost analysis of parallel and distributed systems, based on different models of computation than the independent and-parallel model in our work. In [3] the authors present a static analysis which is able to infer upper bounds on the maximum number of *active* (i.e., not finished nor suspended) processes running in parallel, and the total number of processes created for imperative *async-finish* parallel programs. The approach described in [1] uses recurrence (cost) relations to derive upper bounds on the cost of concurrent object-oriented programs, with shared-memory communication and future variables. They address concurrent execution for loops with semi-controlled scheduling, i.e., with no arbitrary interleavings. In [4] the authors address the cost of parallel execution of object-

oriented distributed programs. The approach is to identify the synchronization points in the program, use serial cost analysis of the blocks between these points, and then, exploiting the techniques mentioned, construct a graph structure to capture the possible parallel execution of the program. The path of maximal cost is then computed. The allocation of tasks to processors (called "locations") is part of the program in these works, and thus, although independent and-parallel programs could be modeled in this computation style, it is not directly comparable to our more abstract model of parallelism.

Solving, or safely bounding recurrence relations with `max` and `min` functions has been addressed mainly for recurrences derived from divide-and-conquer algorithms [5,28,17]. In [2] the authors present solutions for Cost Relation Systems by obtaining upper bounds for both the number of nodes and the cost added in each node in the derived evaluation tree. These bounds are then combined in order to obtain a closed-form upper-bound expression. This closed form possibly contains maximization operations to express upper bounds for a set of subexpressions. However, each cost relation is defined as a summatory of costs, while in our approach, in addition to summations, we also consider other operations for aggregating the costs, including `max` operators. The presence of these operators often produces recurrence relations where the recursive calls are under the scope of such a `max` operator, for which we present a method to obtain a closed-form bound. This class of recurrences are not handled by most of the current computer algebra systems, as the authors in [2] mention.

7 Conclusions

We have presented a novel, general, and flexible analysis framework that can be instantiated for estimating the resource usage of parallel logic programs, for a wide range of resources, platforms, and execution models. To the best of our knowledge, this is the first approach to the cost analysis of *parallel logic programs*. Such estimations include both lower and upper bounds, given as functions on input data sizes. In addition, our analysis also infers other information which is useful for improving the exploitation and assessing the potential and actual parallelism of a program. We have also developed a method for solving the cost relations that arise in this particular type of analysis, which involve the *max* function. Finally, we have developed a prototype implementation of our general framework, instantiated it for the analysis of logic programs with Independent And-Parallelism, and performed an experimental evaluation, obtaining encouraging results w.r.t. accuracy and efficiency.

References

1. Albert, E., Arenas, P., Genaim, S., Gómez-Zamalloa, M., Puebla, G.: Cost Analysis of Concurrent OO programs. In: Proc. of APLAS'11. LNCS, vol. 7078, pp. 238–254. Springer (2011)
2. Albert, E., Arenas, P., Genaim, S., Puebla, G.: Closed-Form Upper Bounds in Static Cost Analysis. Journal of Automated Reasoning **46**(2), 161–203 (2011)
3. Albert, E., Arenas, P., Genaim, S., Zanardini, D.: Task-Level Analysis for a Language with Async-Finish parallelism. In: Proc. of LCTES'11. pp. 21–30. ACM Press (2011)
4. Albert, E., Correas, J., Johnsen, E., Pu, K., Román-Díez, G.: Parallel cost analysis. ACM Trans. Comput. Logic **19**(4) (Nov 2018)

5. Alonso, L., Reingold, E., Schott, R.: Multidimensional divide-and-conquer maximin recurrences. SIAM J. Discret. Math. **8**(3), 428–447 (Aug 1995)
6. Blelloch, G.E., Greiner, J.: A provable time and space efficient implementation of NESL. In: ACM Int'l. Conf. on Functional Programming. pp. 213–225 (May 1996)
7. Debray, S.K., Lin, N.W.: Cost analysis of logic programs. ACM TOPLAS **15**(5), 826–875 (November 1993)
8. Debray, S.K., Lin, N.W., Hermenegildo, M.V.: Task Granularity Analysis in Logic Programs. In: Proc. PLDI'90. pp. 174–188. ACM (June 1990)
9. Debray, S.K., Lopez-Garcia, P., Hermenegildo, M.V., Lin, N.W.: Lower Bound Cost Estimation for Logic Programs. In: ILPS'97. pp. 291–305. MIT Press (1997)
10. Gupta, G., Pontelli, E., Ali, K., Carlsson, M., Hermenegildo, M.V.: Parallel Execution of Prolog Programs: a Survey. ACM TOPLAS **23**(4), 472–602 (July 2001)
11. Haemmerlé, R., Lopez-Garcia, P., Liqat, U., Klemen, M., Gallagher, J.P., Hermenegildo, M.V.: A Transformational Approach to Parametric Accumulated-cost Static Profiling. In: FLOPS'16. LNCS, vol. 9613, pp. 163–180. Springer (2016)
12. Harper, R.: Practical Foundations for Programming Languages. Cambridge University Press, 2 edn. (2016). https://doi.org/10.1017/CBO9781316576892
13. Hermenegildo, M., Puebla, G., Bueno, F., Garcia, P.L.: Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). Science of Computer Programming **58**(1–2), 115–140 (2005)
14. Hermenegildo, M., Rossi, F.: Strict and Non-Strict Independent And-Parallelism in Logic Programs: Correctness, Efficiency, and Compile-Time Conditions. Journal of Logic Programming **22**(1), 1–45 (1995)
15. Hoefler, T., Kwasniewski, G.: Automatic complexity analysis of explicitly parallel programs. In: 26th ACM Symp. on Parallelism in Algorithms and Architectures. pp. 226–235. SPAA '14 (2014)
16. Hoffmann, J., Shao, Z.: Automatic static cost analysis for parallel programs. In: Vitek, J. (ed.) Programming Languages and Systems. pp. 132–157. Springer Berlin Heidelberg, Berlin, Heidelberg (2015)
17. Hwang, H., Tsai, T.H.: An asymptotic theory for recurrence relations based on minimization and maximization. Theoretical Computer Science **290**(3), 1475 – 1501 (2003)
18. Klemen, M., Lopez-Garcia, P., Gallagher, J., Morales, J., Hermenegildo, M.V.: Towards a General Framework for Static Cost Analysis of Parallel Logic Programs. Tech. Rep. CLIP-1/2019.0, The CLIP Lab, IMDEA Software Institute and T.U. Madrid (July 2019), `http://arxiv.org/abs/1907.13272`
19. Lopez-Garcia, P., Darmawan, L., Bueno, F.: A Framework for Verification and Debugging of Resource Usage Properties. In: Technical Communications of ICLP. LIPIcs, vol. 7, pp. 104–113. Schloss Dagstuhl (July 2010)
20. Lopez-Garcia, P., Darmawan, L., Klemen, M., Liqat, U., Bueno, F., Hermenegildo, M.V.: Interval-based Resource Usage Verification by Translation into Horn Clauses and an Application to Energy Consumption. TPLP **18**, 167–223 (March 2018)
21. Lopez-Garcia, P., Klemen, M., Liqat, U., Hermenegildo, M.V.: A General Framework for Static Profiling of Parametric Resource Usage. TPLP **16**(5-6), 849–865 (2016)
22. Méndez-Lojo, M., Navas, J., Hermenegildo, M.: A Flexible (C)LP-Based Approach to the Analysis of Object-Oriented Programs. In: LOPSTR. LNCS, vol. 4915, pp. 154–168. Springer-Verlag (August 2007)
23. de Moura, L.M., Bjørner, N.: Z3: An Efficient SMT Solver. In: TACAS. LNCS, vol. 4963, pp. 337–340. Springer (2008)
24. Navas, J., Mera, E., Lopez-Garcia, P., Hermenegildo, M.: User-Definable Resource Bounds Analysis for Logic Programs. In: Proc. of ICLP'07. LNCS, vol. 4670, pp. 348–363. Springer (2007)

25. Rosendahl, M.: Automatic Complexity Analysis. In: Proc. of FPCA'89. pp. 144–156. ACM Press (1989)
26. Serrano, A., Lopez-Garcia, P., Hermenegildo, M.V.: Resource Usage Analysis of Logic Programs via Abstract Interpretation Using Sized Types. TPLP, ICLP'14 Special Issue **14**(4-5), 739–754 (2014)
27. Shen, K., Hermenegildo, M.: High-level Characteristics of Or- and Independent And-parallelism in Prolog. Int'l. Journal of Parallel Programming **24**(5), 433–478 (1996)
28. Wang, B.F.: Tight bounds on the solutions of multidimensional divide-and-conquer maximin recurrences. Theoretical Computer Science **242**(1), 377 – 401 (2000)
29. Wegbreit, B.: Mechanical Program Analysis. Comm. of the ACM **18**(9), 528–539 (1975)