*Parallel Logic Programming: A Sequel**

AGOSTINO DOVIER ANDREA FORMISANO
Università di Udine and GNCS-INdAM, Italy
(*e-mail:* agostino.dovier|andrea.formisano@uniud.it)

GOPAL GUPTA
University of Texas at Dallas, USA
(*e-mail:* gupta@utdallas.edu)

MANUEL V. HERMENEGILDO
IMDEA Software Institute and Universidad Politécnica de Madrid, Spain
(*e-mail:* manuel.hermenegildo@{imdea.org,upm.es})

ENRICO PONTELLI
New Mexico State University, USA
(*e-mail:* epontell@cs.nmsu.edu)

RICARDO ROCHA
CRACS/INESC TEC and Faculty of Sciences, University of Porto, Portugal
(*e-mail:* ricroc@dcc.fc.up.pt)

submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003

Abstract

Multi-core and highly-connected architectures have become ubiquitous, and this has brought renewed interest in language-based approaches to the exploitation of parallelism. Since its inception, logic programming has been recognized as a programming paradigm with great potential for automated exploitation of parallelism. The comprehensive survey of the first twenty years of research in parallel logic programming, published in 2001, has served since as a fundamental reference to researchers and developers. The contents are quite valid today, but at the same time the field has continued evolving at a fast pace in the years that have followed. Many of these achievements and ongoing research have been driven by the rapid pace of technological innovation, that has led to advances such as very large clusters, the wide diffusion of multi-core processors, the game-changing role of general-purpose graphic processing units, and the ubiquitous adoption of cloud computing. This has been paralleled by significant advances within logic programming, such as tabling, more powerful static analysis and verification, the rapid growth of Answer Set Programming, and in general, more mature implementations and systems. This survey provides a review of the research in parallel logic programming covering the period since 2001, thus providing a natural continuation of the previous survey. In order to keep the survey self-contained, it restricts its attention to parallelization of the major logic programming languages (Prolog, Datalog, Answer Set Programming) and with an emphasis on automated parallelization

* The authors would like to thank the anonymous reviewers for their careful reading and very valuable feedback. We would especially like to thank the editor, Mirek Truszczynski, for his very useful comments and encouragement. This research was partially supported by UNIUD PRID Encase, GNCS/INDAM grants, by NSF grants 1914635 and 1833630, by the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia, within project UIDB/50014/2020, by the Spanish MICINN project PID2019-108528RB-C21 *ProCode*, by the Madrid P2018/TCS-4339 *BLOQUES-CM* program, and by the Tezos foundation.

and preservation of the sequential observable semantics of such languages. The goal of the survey is to serve not only as a reference for researchers and developers of logic programming systems, but also as engaging reading for anyone interested in logic and as a useful source for researchers in parallel systems outside logic programming.
Under consideration in Theory and Practice of Logic Programming (TPLP).

KEYWORDS: Parallelism, high-performance computing, logic programming.

In loving memory of our friends Francisco Bueno, Ricardo Lopes, and German Puebla, whose contributions to the field of logic programming have paved the way of many innovations and discoveries.

1 Introduction

The universal presence of multi-core and highly-connected architectures has renewed interest in language-based approaches to the exploitation of parallelism. Since its original inception, *logic programming* has been recognized as one of the programming paradigms with the greatest potential for automated exploitation of parallelism. Kowalski, in his seminal book (Kowalski 1979) identifies parallelization as a strength of logic programming and Pollard explored the potential of the parallel execution of Prolog (Pollard 1981). This was the beginning of an intense branch of research, with a number of highlights over the years—e.g., during the fifth generation computing systems project, 1982–1990. A comprehensive survey of the first twenty years of parallel logic programming has served for many years as a reference to researchers and developers (Gupta et al. 2001). The content of that survey is still valid today. At the same time, the field of parallel and distributed logic programming has continued to evolve at a fast pace, touching on many exciting achievements in more recent years.

Many of these achievements and ongoing research have been driven by the rapid pace of technological innovation, including the advent of Beowulf clusters, the wide diffusion of multicore processors, the game-changing role of general-purpose Graphic Processing Units (GPUs), the wide adoption of cloud computing, and the advent of big data with related computing infrastructures.

In the past 20 years, there has been a wealth of additional research that has explored the role of new parallel architectures in speeding up and scaling up the execution of different logic programming paradigms. Notable achievements have been accomplished in the use of new generations of parallel architectures for both Prolog and Answer Set Programming (ASP), in the exploration of parallelism in constraint programming, and in the extensive use of general purpose GPUs to speedup the execution of various flavors of logic programming.

This survey provides a review of the research in parallel logic programming covering the period since 2000. The survey has been designed to be self-contained and accessible, however, being a natural continuation of the popular survey by Gupta et al., readers are strongly encouraged to read both surveys for a more comprehensive perspective of the field of parallel logic programming.

The goal of the survey is to serve not only as a reference for researchers and developers

of logic programming systems, but also as an engaging reading for anyone interested in logic programming. Furthermore, we believe that this survey can provide useful insights and ideas to researchers interested in parallel systems outside of the domain of logic programming, as already happened in the past. We will describe the key challenges encountered in realizing different styles of parallelism in logic programming and review the most effective solutions proposed in the literature. It is beyond the scope of this survey to review the actual performance results produced—as they have been derived using a diversity of benchmarks, coding techniques, and hardware architectures. The interested reader will be able to find such detailed results in the original papers cited in the bibliography of the survey. We would also like to stress that our focus is primarily on systems where parallelism does not modify the semantics of the programs being executed. The only exception is represented by the discussion on explicit parallelism, which introduces a different semantics with respect to the original Prolog systems.

A survey on the approaches to parallelism for constraint programming has been recently published (Gent et al. 2018). *Constraint Logic Programming (CLP)* languages, if used for pure constraint modeling, can immediately benefit from the results presented in the constraint programming survey. Considering that no recent work has appeared in the domain of parallelization of CLP, we will not discuss this area in this survey.

Many of the systems that appeared in the literature and are mentioned in this survey are publicly available. Interested readers are referred to the web page `www.logicprogramming.org` (Systems and Links tab), hosted by the Association for Logic Programming (ALP).

The survey is organized as follows. Section 2 provides some background on logic programming and parallelism. A quick review of the first 20 years of parallel logic programming is presented in Section 3. Section 4 explores the more recent advances in parallel execution of Prolog, reviewing progress in execution models for Or-parallelism (Section 4.1) and And-parallelism (Section 4.2), static analysis for exploitation of parallelism (Section 4.3), and finally exploitation of parallelism in Prolog systems with tabling (Section 4.4). Section 5 reviews the techniques proposed to exploit parallelism in ASP, including parallelism in Datalog (Section 5.1), traditional parallelism in ASP (Section 5.2), parallel grounding (Section 5.3), and other forms of parallelism used in ASP (Sections 5.4-5.5). The following sections explore the execution of logic programming in the context of big data frameworks ("going large," Section 6) and in the context of GPUs ("going small," Section 7). Section 6 also includes a discussion of execution models for logic programming on distributed computing platforms and frameworks designed for handling massive quantities of data (e.g., MapReduce). Section 8 provides some final remarks.

2 Background

We start this section with a brief introduction to the foundations of logic programming. Basically, a logic program consists of facts and logical rules, where deduction is carried out by automatizing some variants of the *modus ponens*. The idea is to automatically derive the set of logical consequences of a logic program. If this set is finite, as is the case with the common restriction used in ASP, a bottom-up computation approach can be

used. If, instead, this set is infinite, top-down computation methods allow us to derive selected inferences (e.g., for a given predicate). The interested reader is referred to the book by Lloyd (1987) for a review of the fundamental notions of logic programming. Some basic notions about parallelism and its limits are reported in Section 2.2.

2.1 Logic Programming

A logic programming language is built on a signature composed of a set of function symbols $\mathcal{F}$, a set of variables $\mathcal{X}$, and a set of predicate symbols $\mathcal{P}$. An arity function $ar(p)$ is associated to each function symbol and predicate symbol p. A *term* is either a variable $x \in \mathcal{X}$ or a formula of the form $f(t_1, \ldots, t_n)$, where $t_1, \ldots, t_n$ are themselves terms, $f \in \mathcal{F}$, and $ar(f) = n \geq 0$. A constant is a symbol $f \in \mathcal{F}$ such that $ar(f) = 0$.

An atomic formula (or simply *atom*) is a formula of the form $p(t_1, \ldots, t_n)$, where $p \in \mathcal{P}$, $t_1, \ldots, t_n$ are terms, and $ar(p) = n$. A *literal* is either an atom or an entity of the form *not* A, where A is an atom. A *clause* is a formula of the form

$$H \leftarrow B_1, \ldots, B_n, not\ C_1, \ldots, not\ C_m$$

where $H, B_1, \ldots, B_n, C_1, \ldots, C_m$ are atoms. If $m = 0$ (i.e., there are no negated literals), the clause is said to be *definite*, if $m = n = 0$, then the clause is said to be a *fact*.

Given a clause r, we define $head(r) = H$, $pos(r) = \{B_1, \ldots, B_n\}$ and $neg(r) = \{C_1, \ldots, C_m\}$, while $B_1, \ldots, B_n, not\ C_1, \ldots, not\ C_m$ is called the body of r. The symbol "$\leftarrow$" can be read as "if"; in the rest of this paper we will use $\leftarrow$ and its programming language syntactic notation `:-` interchangeably. Intuitively, if the body of the clause is true, then the head will be true as a consequence. Variables are universally quantified. For instance, the clause

$$grandparent(X, Y) \leftarrow parent(X, Z), parent(Z, Y)$$

can be read as: "*for every X, Y, Z, if X is parent of Z and Z is parent of Y then X is a grandparent of Y*". A program is a collection of clauses. A program is *definite* if all of its clauses are definite. Negated literals are here written at the end of the body, for simplicity, but they can appear interleaved with positive literals in actual programs.

Important information concerning the semantics of a program P can be obtained by the analysis of its *dependency graph* $\mathcal{G}(P)$ (Baral 2003; Gelfond and Kahl 2014). $\mathcal{G}(P)$ is a graph where nodes correspond to atoms in P; an edge $p \leftarrow q$ is added in $\mathcal{G}(P)$ if there is a clause in P with p as head and q in the body. The edge is labeled differently if q occurs as an atom or a negated atom in the body of the clause.

A term (atom, clause, program) is *ground* if it contains no variables. A substitution θ is a mapping from a set of variables to terms. If r is a term (literal, clause) and θ a substitution, $r\theta$ denotes the term (literal, clause) obtained by replacing each variable X in r with the term $\theta(X)$. Given two terms/atoms s, t, we say that s is subsumed by t (or s is an instance of t) if there is a substitution θ such that $s = t\theta$. Two terms/atoms s and t are said to be *variants* if s is subsumed by t and t is subsumed by s. In this case, s and t are identical modulo a variable renaming. An instance $t\theta$ of t is a *ground instance* if $t\theta$ contains no variables. Given a clause r, $ground(r)$ denotes the set of all ground instances of r; analogously, given a program P, $ground(P) = \bigcup_{r \in P} ground(r)$.

Given two substitutions σ and θ, σ is more general than θ if there is a substitution

γ such that for every term t we have that $(t\sigma)\gamma = t\theta$. θ is a *unifier* of two terms/atoms s and t if $s\theta = t\theta$, namely the two terms/atoms become syntactically equal after the substitution is applied. If two terms/atoms admit a unifier, they will also admit a *most general unifier (mgu)*—i.e., the most general substitution which is also a unifier; the mgu is unique modulo variable renaming.

Let us denote with B_P the set of all possible ground atoms that can be built with the function and predicate symbols occurring in a program P, also known as the Herbrand Base of P. An *interpretation* $I \subseteq B_P$ is a set of ground atoms—intuitively representing which atoms are true; all atoms in $B_P \setminus I$ are assumed to be false.

An interpretation I is a *model* of a ground clause r if either $head(r) \in I$, or $pos(r) \not\subseteq I$, or $neg(r) \cap I \neq \emptyset$ (namely, if the head is true or if the body is false). An interpretation I is a model of a clause if it is a model of all its ground instances. An interpretation I is a model of a program P if it is a model of each clause in P.

The semantics of definite programs, which is used as the foundation of Datalog and Prolog, is defined in terms of the set of logical consequences, namely by the set of atoms that are true in every model. For a definite logic program P the existence of a unique minimum model, denoted by M_P, is guaranteed. M_P can be defined as the intersection of all I such that I is a model of P; it can be shown that M_P corresponds to the set of all ground logical consequences of P.

The minimal model M_P of a definite logic program has a constructive characterization through the use of the *immediate consequence operator* T_P. The operator maps interpretations to interpretations, and it is defined as:

$$T_P(I) = \{head(r) \mid r \text{ ground instance of a clause in } P,\ pos(r) \subseteq I\}$$

T_P is a monotone and continuous operator; in addition, M_P is the least fixpoint of T_P, i.e., $M_P = T_P(M_P)$, and $M_P = T_P \uparrow \omega$, where $T_P \uparrow 0 = \emptyset$, $T_P \uparrow n = T_P(T_P \uparrow (n-1))$ for a successor ordinal n, and $T_P \uparrow \alpha = lub\{T_P \uparrow n \mid n < \alpha\}$ for a limit ordinal α. Computing M_P as $T_P \uparrow \omega = T_P(T_P(\cdots(T_P(\emptyset))\cdots))$ is called the *bottom-up approach* to the semantics of P.

In the context of Prolog, computation is expressed in terms of reasoning about the minimal model M_P. Given a program P and a conjunction of possibly non-ground atoms $\bigwedge_{i=1}^{n} p_i$, the objective is to determine substitutions θ for the variables in $p_1, \ldots, p_n$ such that $M_P \models \bigwedge_{i=1}^{n} p_i\theta$; these are referred to as *correct answers*. The conjunction $\bigwedge_{i=1}^{n} p_i$ is referred to as a *goal* and subsets of $\{p_1, \ldots, p_n\}$ are called *subgoals* (let us observe that a subgoal can contain more than one atom).

SLD resolution is a strategy used to derive correct answers. It can be described as the process of constructing a tree, the SLD-resolution tree, whose nodes are pairs composed of a goal and a substitution. Given a goal $\vec{p} = \bigwedge_{i=1}^{n} p_i$ and a program P, the root of the resolution tree is $\langle \vec{p}, \epsilon \rangle$, where ϵ is the identity substitution. If $\langle \bigwedge_{i=1}^{m} q_i, \theta \rangle$ is a node in the resolution tree, then such node will have as many children as there are rules $r \in P$ such that there is a substitution γ such that $head(r')\gamma = q_1\theta\gamma$, where r' is a renaming of the rule r with fresh new variables. In this case a most general unifier of $head(r')$ and $q_1\theta$ is chosen. Each child is of the form $\langle pos(r') \circ [q_2, \ldots, q_n], \theta\gamma \rangle$, where $\theta\gamma$ is the composition of the two substitutions and $\circ$ is the list concatenation operator. If $m = 0$, then the node will be labeled as a *success node* and θ is a correct answer. If $m > 0$ but no child can be constructed, then the node will be labeled as a *failed node*. Prolog typically implements

SLD resolution by building the resolution tree in a depth-first manner, where the children of each node are sorted according to the ordering of the clauses in the program P. The SLD resolution provides a *top-down* approach to reason about the semantics of P.

The top-down approach is extended in the case of negated literals in goals (general programs). The goal *not p* succeeds or fails as a consequence of the result of the goal p. If there is an answer substitution θ for p then the goal *not p* fails, otherwise it succeeds. This is referred to as *negation as failure*.

However, for programs with general clauses, the existence and uniqueness of a minimum model is no longer guaranteed. The *well-founded* model of a logic program is a unique 3-valued model (Van Gelder et al. 1991) or 4-valued model (Truszczynski 2018). We briefly report the procedure presented by Brass et al. (2001), based on the following variant of the T_P operator, for computing the 3-valued well-founded semantics. Given a ground program P and two sets of ground atoms I, J, we define the extended immediate consequence operator

$$T_{P,J}(I) = \{head(r) \mid r \in P, pos(r) \subseteq I, neg(r) \cap J = \emptyset\}$$

Let us denote by P^+ the set of definite clauses in P, the *alternating* fixpoint procedure is defined as follows

$$\begin{array}{ll} K(0) = lfp(T_{P^+}) & U(0) = lfp(T_{P,K(0)}) \\ K(i+1) = lfp(T_{P,U(i)}) & U(i+1) = lfp(T_{P,K(i+1)}) \end{array}$$

Let (K^*, U^*) be the minimum fixpoint of the computation; that is, if i is the smallest value such that $K(i) = K(i-1)$ and $U(i) = U(i-1)$, then $K^* = K(i)$ and $U^* = U(i)$.

The well-founded model of the program P is a 3-valued model W^* defined as $W^* = (K^*, U^*)$—where all atoms in K^* are true, all atoms in U^* are false, and all atoms in $H \setminus (K^* \cup U^*)$ are unknown.

The most successful approach for the two-valued semantics is the one based on the notion of *answer sets*. Given a program P (for simplicity let us assume P to be ground) and given an interpretation I, we define the reduct of P with respect to I, denoted by P^I as the set of definite clauses $P^I = \{head(r) \leftarrow pos(r) \mid r \in P, neg(r) \cap I = \emptyset\}$. A model M of P is an *answer set* of P if M is the minimum model of the definite program P^M. It should be noted that a program may have no answer sets, one answer set (for example, each definite program has exactly one answer set, corresponding to its minimal model), or multiple answer sets.

2.2 Parallelism and Speedups

When talking about parallelism, it is a naive belief that if a program runs in time T on a computing platform, then it should be possible to execute the program N times faster using N processors.[1] Namely, that the running time could decrease to $\frac{T}{N}$.

A definitive theoretical limit to this kind of reasoning was set by Amdahl (1967). The crucial point is that the program we are considering is composed of parts that are intrinsically sequential and other parts that can be parallelized. Let S (sequential) and P

[1] In the rest of the paper, we will use the terms *processor* and *process* in a general sense, as representing an entity capable of computation (e.g., a CPU, a core).

(parallel) be the running times of the two fractions of the program, scaled to guarantee that $S + P = 1$. With N processors one can expect an ideal running time of $T\left(S + \frac{P}{N}\right)$, with a speedup of $\frac{1}{S+\frac{P}{N}}$. Observe that with a "very large" N this leads to a maximum speedup of $\frac{1}{S}$. Even if S is "small" with respect to the program, e.g., $S = 0.1$, this means that the maximum speedup is 10, no matter how many processors are used. The naive reasoning mentioned at the beginning of the paragraph assumes implicitly that $S = 0$, which is often not a realistic assumption.

Gustafson (1988) approached the problem from another perspective, giving new hopes of high impact for parallel systems. The key observation of Gustafson is that Amdahl's law assumes that the problem size (the data size) is fixed. However, Gustafson argues that, in practice, it makes more sense to scale the size of the problem (e.g., the amount of data) in parallel with the addition of processors. Given the running time T as discussed earlier, let us assume that the parallel part of the program P has "to do with data," and that the architecture allows parallel processors to concurrently access independent portions of the data. Then in the same time T, in principle, the program could process N times more data, leading to a speedup of $\frac{PN+S}{P+S}$. If $S = 0.1$, with one thousand of processors we can, in principle, aim at reaching a speedup of ≈ 900.

3 The First 20 Years of Parallel Logic Programming: A Quick Review

The majority of the original research conducted on parallel execution of logic programming focused on the exploitation of parallelism from the execution of Prolog programs. In this section, we provide a brief summary of some of the core research directions explored in the original literature. Readers are encouraged to review the survey by Gupta et al. (2001) for further details.

3.1 Explicit Parallelism

Parallel execution of the different operations in a logic program results in *implicit* exploitation of parallelism, i.e., no input is required from the user to identify and exploit parallelism; rather, parallelism is automatically exploited by the inference system. Nevertheless, the literature has presented several approaches that explore extensions of a logic programming language with *explicit* constructs for the description of concurrency and parallelism. While these approaches are not the focus of this survey, we will briefly mention some of them in this section.

The approaches for the explicit description of parallelism in logic programming can be largely classified into three categories: **(1)** *message passing*; **(2)** *shared memory*; and **(3)** *data-flow*.

Methods based on *message passing* extend a logic programming language, typically rooted in Prolog, to enable the creation of concurrent computations and the communication among them through the explicit exchange of messages. Several systems have been described over the years with similar capabilities, such as Delta Prolog (Pereira et al. 1986) and CS-Prolog (Futó 1993). A more recent example is the April system (Fonseca et al. 2006), where the high-level modeling capabilities of logic programming are used to explicitly parallelize Machine Learning procedures into independent tasks, in the various stages of data analysis, search, and evaluation (Fonseca et al. 2009).

Message passing features have become common in many implementations of Prolog—for example, Ciao Prolog (Hermenegildo et al. 2012), SICStus Prolog (Carlsson and Mildner 2012), SWI-Prolog (Wielemaker et al. 2012), tuProlog (Calegari et al. 2018), XSB (Swift and Warren 2012), and YAP Prolog (Santos Costa et al. 2012).

Most Prolog implementations provide comprehensive multi-threading libraries, allowing the creation of separate threads and enabling message passing communication between them so (see also the paper by Körner et al. (2022)). The design of these libraries has also been guided by a 2007 ISO technical document which provides recommendations on how multi-threading predicates may be introduced in Prolog.
E.g., in YAP Prolog threads are created as follows:

```
thread_create(:Goal, -Id, +Options)
```

and the communication between them can be achieved through message queues, e.g.,

```
thread_send_message(+ThreadId, +Term)
thread_get_message(?Term)
```

The roots of message-passing Prolog systems date back to the early 80s. The Delta Prolog system (Pereira and Nasr 1984) draws inspiration from communicating sequential processes to provide the ability of creating processes capable of synchronizing and exchanging messages; for example, the following code snippet describes a simple producer-consumer structure:

```
main :- producer // consumer.
producer :- generate(X), X ! channel, producer.
consumer :- X ? channel, consume(X), consumer.
```

A more extensive design is proposed in the CS-Prolog system (Futó and Kacsuk 1989), which provides communicating process designed to execute on Transputer architectures.

The second class of methods is based on the use of *shared memory* to enable communication between concurrent processes. The most notable approaches rely on the use of blackboards, as exemplified in the Shared Prolog system (Ciancarini 1990) and in the various Linda libraries available in several Prolog implementations (e.g., SICStus). An interesting alternative is provided by Ciao Prolog, where the communication is done using *concurrent facts* in the Prolog database (Carro and Hermenegildo 1999). This approach is designed to provide a clear transactional behavior, elegantly interacting with backtracking and preserving, under well-defined conditions, the declarative semantics of logic programming.

Finally, a more extensive literature has been developed around the concept of *concurrent logic programming* (see for example the works by Shapiro (1987; 1989),Clark and Gregory (1986), and Tick (1995)) and implemented in systems like Parlog and GHC. These languages offer a syntax similar to traditional logic programming, but they view all goals in the program clauses as concurrent processes, capable of interacting through shared variables and synchronizing through dataflow constraints. For example, a goal of the type `?- generate(Z), proc(Z)` creates two concurrent processes, the first generating values in a list (used as a stream in concurrent logic programming) and the second process consuming values from the stream, e.g.,

```
proc([X|Y]) :- X > 0 | consume(X), proc(Y).
proc([X|Y]) :- X <= 0 | proc(Y).
```

The second process suspends until a value is available in the stream; the clause whose guard (i.e., the goal preceding the "|") is satisfied will be activated and executed. One important aspect to observe is that, due to the complications of combining search with this type of concurrency, these systems do not support the classical non-determinism of logic programs. They instead implement only the deterministic part—a form of computation referred to as "committed choice." The fact that all goals are concurrent processes may also lead to complex and unintended interactions, making programs harder to understand.

The idea of synchronization through shared variables is very appealing. This concept has generated several research efforts focused on supporting variable binding-based communication in conjunction with explicit concurrency in Prolog systems, e.g., in several proposals for dependent And-parallelism (Gupta and Pontelli 1997; Hermenegildo et al. 1995; Shen 1996). More information about these directions of work can be found later in this section and in Section 4.2.

3.2 Implicit Parallelism

The fundamental idea underlying the implicit exploitation of parallelism is to make the exploitation of parallelism transparent to the programmer, by automatically performing in parallel some of the steps belonging to the operational semantics of the language.

As mentioned earlier, most of the pre-2000 literature on implicit exploitation of parallelism focused on the parallelization of the execution model of Prolog. In very general terms, the traditional SLD-resolution process adopted by Prolog can be visualized as a search procedure (see Algorithm 1). There are three major steps in Algorithm 1: *Atom Selection, Clause Selection,* and *Unification.* Atom Selection (also known as literal selection in literature) is used to identify the next atom to resolve (line 3 of Algorithm 1). Clause Selection is used to identify which clause to use to conduct resolution (line 5). Unification determines the most general unifier to be used to unify the selected literal with the head of the selected clause (line 6).

Extensive research has been performed to improve efficiency of these steps, especially, for atom selection and clause selection. Techniques such as constraint programming (Gent et al. 2018) and the Andorra principle (Santos Costa et al. 1991b) attempt to select subgoals in an order that will lead to early pruning of the search space. The parallelization of the atom selection step leads to *independent* and *dependent And-parallelism* (Hermenegildo 1986b; Shen 1996; Gupta and Pontelli 1997), which allow multiple subgoals to be selected and solved in parallel. The distinction between the two forms of And-parallelism derives from whether shared unbound variables are allowed (dependent And-parallelism) or not (independent And-parallelism) between concurrently resolved subgoals. The parallelization of the Clause Selection step, which allows resolution of the selected subgoal using multiple clauses in parallel, leads to *Or-parallelism* (Zhang 1993; Lusk et al. 1990; Ali and Karlsson 1990b).

Unification parallelism arises when arguments of a goal are concurrently unified with those of a clause head with the same name and arity. This can be visualized in Algorithm 1

Algorithm 1: High-level view of SLD-Resolution

```
   Input: Input Goal
   Input: Program Clauses
   Output: Computed Substitution
1  i = 0
2  while (Goal not Empty) do
3      select_atom B from Goal                          /* And-Parallelism */
4      repeat
5          select_clause (H :- Body) from Program        /* Or-Parallelism */
6      until (unify(H,B) or no clause is left)      /* Unification Parallelism */
7      if (no clause is left) then
8          return Fail
9      else
10         θ_i = most_general_unifier(H,B)
11         Goal = (Goal \ {B} ∪ Body)θ_i
12         i++
13 return θ_0 θ_1 ··· θ_{i-1}
```

as the parallel execution of the steps present in the *unify* operation. The different argument terms can be unified in parallel as can the different subterms in a term (Barklund 1990). Unification parallelism is very fine-grained, it requires dealing with dependencies caused by multiple occurrences of the same variables, and is best exploited by building specialized processors with multiple unification units (Singhal and Patt 1989). Unification parallelism has not been the major focus of research in parallel logic programming, so we will not consider it any further.

3.2.1 Or-Parallelism

Or-parallelism arises when multiple clauses have the same predicate in the head and the selected subgoal unifies with more than one clause head—the corresponding clause bodies can be explored in parallel, giving rise to Or-parallelism. This can be illustrated in Algorithm 1 as the parallelization of the choices present in the $select_{clause}$ operation. Or-parallelism is thus a way of efficiently searching for solutions to the goal, by exploring alternative solutions in parallel: it corresponds to the parallel exploration of the search tree originating from the choices performed when selecting different clauses for solving the selected subgoals (often referred to as the *or-tree*).

Let us consider a very simple example, composed of the Prolog clauses that define the concatenation of two lists:

```
append([], X, X).
append([H|X], Y, [H|Z]) :- append(X, Y, Z).
```

and the goal:

```
?- append(X, Y, [1,2,3]).
```

The append literal in the goal (and all append subgoals that arise recursively during resolution) will match both clauses, and therefore the bodies of these clauses can be executed in parallel to find the four solutions to this goal.[2]

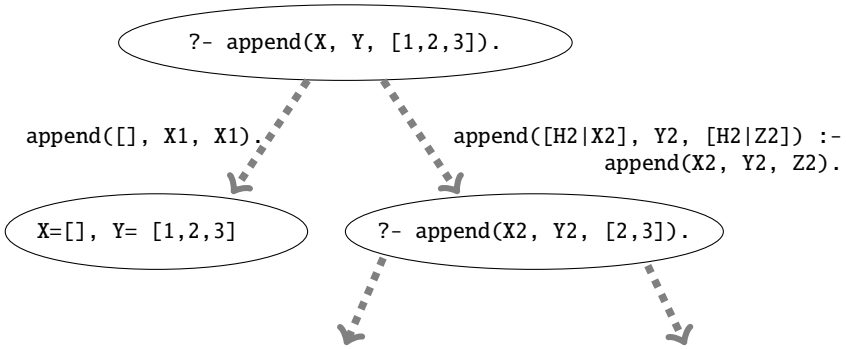

Fig. 1: Example of Or-parallelism

Or-parallelism should be, in principle, easy to achieve, since the various branches of the or-tree are independent of each other, as they each explore an alternative sequence of resolution steps. As such, their construction should require little communication between parallel computations.[3] However, in practice, implementation of or-parallelism is difficult because of the sharing of nodes in the or-tree. Given two nodes in two different branches of the or-tree, all nodes above (and including) their least common ancestor node are shared between the two branches. A variable created in one of these ancestor nodes might be bound differently in the two branches. The environments of the two branches have to be organized in such a way that, in spite of the ancestor nodes being shared, the correct bindings applicable to each of the two branches are easily discernible.

If a binding for a variable, created in one of the common ancestor nodes, is generated above (or at) the least common ancestor node, then this binding will be the same for both branches, and hence can be used as such. Such binding is known as an *unconditional binding* and such a variable is referred to as an unconditional variable. However, if a binding to such a variable is generated by a node below the least common ancestor node, then that binding shall be visible only in the branch to which the binding node belongs. Such a binding is known as a *conditional binding* and such a variable is referred to as a conditional variable. The main problem in implementing Or-parallelism is the efficient representation of multiple environments that co-exist simultaneously in the or-tree—commonly known as the *environment representation problem*. Note that the main problem in the management of multiple environments is how to efficiently represent and access the conditional bindings; the unconditional bindings can be treated as in normal sequential execution of logic programs. The environment representation problem has to be solved by devising a mechanism where each branch has some private areas where it

[2] Note however that in this example exploiting parallelism may not be too profitable since the *granularity* of the tasks is small. We will return to this important topic.

[3] Minor control synchronization is still needed. Moreover, we are not considering the challenge of dealing with side-effects, which require communication between branches of the or-tree.

stores the conditional bindings applicable to such branch. Several approaches have been explored to address this problem. For example:

- By storing the conditional bindings created by a branch in an array or a hash table private to that branch, from where the bindings are accessed whenever they are needed. This approach has been adopted, for example, in the binding array model, successfully used in the Aurora system (Warren 1984; Lusk et al. 1990).
- Keeping a separate copy of the environment for each branch of the tree, so that every time branching occurs at a node the environment of the old branch is copied to each new branch. This approach has been adopted, for example, in the stack-copying model, successfully used in the Muse and the ACE systems (Ali and Karlsson 1990a; Pontelli and Gupta 1997).
- Recording all the conditional bindings in a global data-structure and attaching a unique identifier with each binding which identifies the branch a binding belongs to. This approach has been explored, for example, in the version vectors model (Hausman et al. 1987).

3.2.2 And-Parallelism

And-parallelism arises when more than one subgoal is present in the goal or in the body of a clause, and multiple subgoals are selected and resolved concurrently. This can be visualized in Algorithm 1 as generating parallelism from the operation $select_{atom}$. The literature has traditionally distinguished between *independent And-parallelism* and *dependent And-parallelism*.

In the case of *independent And-parallelism*, subgoals selected for parallel execution are guaranteed to be independent of each other with respect to bindings of variables. In other words, two subgoals solved in parallel are guaranteed to not compete in the binding of unbound variables. A sufficient condition for this (called *strict independence*) is that any two subgoals solved in parallel do not have any unbound variables in common. Independent And-parallelism is, for example, a way of speeding up a divide-and-conquer algorithm by executing the independent subproblems in parallel. The advantage of independence is that the parallel execution can be carried out without interaction through shared variables. Communication is still needed for returning values and synchronization (including during backtracking).

The main systems proposed to support independent And-parallelism, e.g., &-Prolog (Hermenegildo 1986b) and &ACE (Pontelli et al. 1995), adopted initially a fork-join organization of the computation, inspired by the original proposal on *restricted* And-parallelism by DeGroot (1984). In this fork-join model independent goals selected to be run in parallel (identified, e.g., using a different conjunction operator &) are made available simultaneously for parallel execution, and the continuation of the computation waits for completion of these parallel goals before proceeding (e.g., see Figure 2). These And-parallel structures can be arbitrarily nested.

The major challenges in the development of independent And-parallel systems are:

- The implementation of distributed backtracking—the goal of maintaining the same visible behavior as a sequential Prolog system implies the need of allowing a

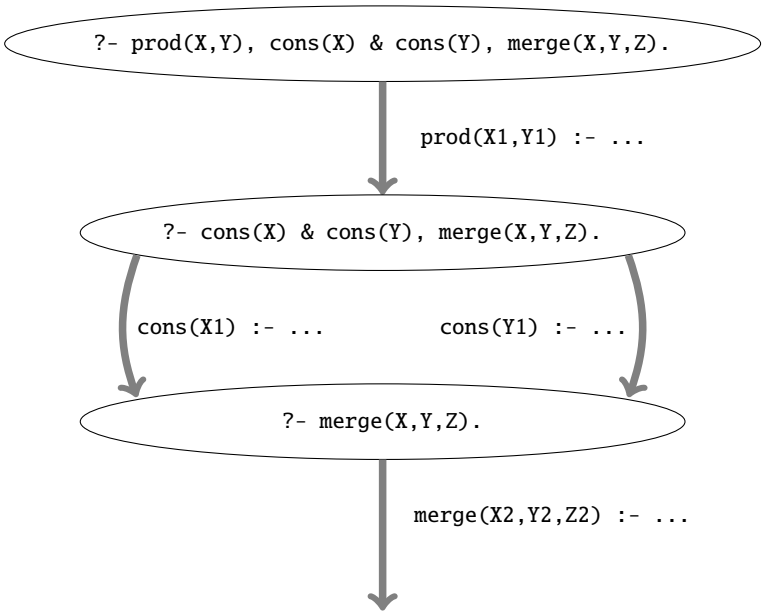

Fig. 2: Intuition behind independent And-parallelism

processor to backtrack over computations performed by other processors, with requirements of communication and synchronization; these issues have been extensively explored by Hermenegildo (1986a)Hermenegildo and Nasr (1986)Shen and Hermenegildo (1996)Pontelli and Gupta (2001); Chico de Guzmán et al.(2011; 2012).

- The identification of subgoals that will be independent at run-time—this problem has been addressed by Hermenegildo (1986a; 1986b) through manual program annotations and by Muthukumar et al. (1999) using static analysis techniques.
- The need to allow more relaxed notions of independence, i.e., the conditions that guarantee that subgoals can be solved in parallel without communication. These include:
  - — the classical notion of independence (called *strict independence* by Hermenegildo and Rossi (1995), i.e., goals should not share variables at run time;
  - — *non-strict independence*, characterized by the fact that only one goal can bind each shared variable (Hermenegildo and Rossi 1995);
  - — *search independence*, i.e., bindings are compatible (García de la Banda 1994);
  - — *constraint independence* (García de la Banda et al. 2000).

  Deterministic behavior is also a form of independence (Hermenegildo and Rossi 1995).

Dependent And-parallelism arises when two or more subgoals that are executed in parallel have variables in common and the bindings made by one subgoal affect the execution of other subgoals. Dependent And-parallelism can be exploited in two ways (let us consider the simple case of two subgoals):

1. The two subgoals can be executed independently until one of them accesses/binds the common variable. It is possible to continue executing the two subgoals independently in parallel, separately maintaining the bindings generated by each subgoal. In such a case, at the end of the execution of the two subgoals, the bindings produced by each will have to be checked for compatibility (this compatibility check at the end is referred to as *back unification*).
2. Once the common variable is accessed by one of the subgoals, it is bound to a structure, or *stream* (the goal generating this binding is called the *producer*), and the structure is read as an input argument by the other goal (called the *consumer*).

Case **(1)** is very similar to independent And-parallelism and can be seen as exploiting independence at a different granularity level. Case **(2)** is sometimes also referred to as *stream-parallelism* and is useful for speeding up producer-consumer interactions, by allowing the consumer goal to compute with one element of the stream while the producer goal is computing the next element. Note that stream-parallelism introduces a form of coroutining. Stream-parallelism forms the basis of the Committed Choice Languages mentioned earlier (e.g., Parlog (Clark and Gregory 1986), GHC (Ueda 1986), and Concurrent Prolog (Shapiro 1987; 1989)).

The main challenge in implementing dependent And-parallelism is controlling the parallel execution of the consumer subgoal. Given two subgoals that share a variable `X`, one a producer of `X` and another a consumer, the execution of both can be initiated in parallel. However, one must make sure that the consumer subgoal computes only as far as it does not instantiate the dependent variable `X` to a non-variable binding. If it attempts to do so, it must suspend. It will be woken up only after a non-variable binding has been produced for `X` by the producer subgoal—or, alternatively, after the producer has completed its execution without binding `X`. Thus, in addition to the concerns mentioned in the context of independent And-parallelism, the two additional main concerns in implementing dependent And-parallelism are:

1. Determining which instance of a given dependent variable is a producer instance and which instances are consumer instances;
2. Developing efficient mechanisms for waking up suspended subgoals when the variables on which they were suspended are instantiated (or turned themselves into producer instances).

In Prolog execution, subgoals in the current goal are resolved in a left-to-right order. Thus, when a subgoal is resolved, it will never have an unexecuted subgoal to its left in the goal. This rule, however, can be relaxed, and considerable advantage can be gained by processing goals in a different order. In particular, this can be applied to subgoals that have at most one matching clause, known as *determinate* subgoals. This has been realized in the *Andorra principle* (Santos Costa et al. 1991a). The Andorra principle states that all determinate subgoals in the current goal should be executed first, irrespective of their position in the goal. Once all determinate subgoals have finished execution, the leftmost non-determinate subgoal is selected and its various alternatives tried in the standard Prolog order. Along each alternative, the same principle is applied. Under the Andorra principle, all deterministic decisions are taken as soon as possible and this facilitates coroutining and leads to a significant narrowing of the search space.

The Andorra principle has been realized in the Andorra-I system (Santos Costa et al. 1991b). The Andorra-I system also has a determinacy analyzer which generates conditions for each subgoal at compile-time (Santos Costa et al. 1991c). These simple conditions are checked at runtime and their success indicates that the corresponding subgoal is determinate. The Andorra-I system is a *goal stacking* implementation of Prolog rather than the traditional *environment stacking*, due to the need to reorder subgoals during execution. The Andorra principle is in fact useful even beyond parallelism, as a control rule for Prolog, and can also be implemented using delay primitives (Bueno et al. 1994). This is supported for example by the Ciao Prolog system.

The Andorra principle has been generalized to the Extended Andorra Model (EAM) (Haridi and Brand 1988) to exploit both And- and Or-parallelism. In EAM, arbitrary subgoals can execute in And-parallel and clauses can be tried in Or-parallel. Computations that do not impact the environment external to a subgoal are freely executed, giving rise to parallelism. However, computations that may bind a variable occurring in argument terms of a subgoal are suspended, unless the binding is deterministic (in which case the binding is said to be *promoted*). If the binding is non-deterministic, then the subgoal is replicated for each binding (*non-determinate promotion*). These replicated subgoals are executed in Or-parallel. At any moment, constraints and bindings generated outside of a subgoal are immediately percolated down to that subgoal (propagation).

The Extended Andorra Model seeks to optimally exploit And-/Or-parallelism. It provides a generic model for the exploitation of coroutining and parallelism in logic programming and has motivated two main lines of research. The first path resulted in the Andorra Kernel Language (Haridi and Janson 1990) that can be thought of as a new paradigm that subsumes both Prolog and concurrent logic languages (Haridi and Brand 1988). The second focused on the EAM with Implicit Control (Warren 1990; Lopes et al. 2012), where the goal is to achieve efficient (parallel) execution of logic programs with minimal programmer control. An interpreter (in Prolog) was developed to better understand EAM with implicit control (Gupta and Warren 1992) and new concepts, such as lazy copying and eager producers, that give finer control over search and improve parallelism have been investigated. Gupta and Pontelli later experimented with an extension of dependent And-parallelism that provides some of the functionality of the EAM through parallelism (Gupta and Pontelli 1997). The first prototype parallel implementation of the EAM (called BEAM), based on the WAM, has been presented by Lopes et al.(2003; 2004). The BEAM system is promising, though the fine grain And-/Or-parallelism that EAM supports results in significant overhead and thus extracting good performance requires taking into account additional factors such as granularity control.

The state-of-the-art in the exploitation of dependent And-parallelism at the beginning of 2000 is represented by systems like ACE (Gupta and Pontelli 1997), BEAM (Lopes et al. 2012) and DDAS (Shen 1996), which are effective but complex. There are also some early attempts at simpler approaches to parallel implementations (Hermenegildo et al. 1995).

4 Parallel Execution of Prolog

Following the brief review made in the previous section of some of the core issues underlying parallel execution of Prolog, let us now turn our attention to the advances

made in the area of parallel execution of logic programming since 2000. We start, in this section, with reviewing the most recent progress in the context of parallel execution of Prolog. We survey new theoretical and practical results in Or- and And-parallelism, discuss advances in Static Analysis to aid with the exploitation of parallelism, and, finally, discuss the combination of parallelism with the notion of Tabling.

4.1 Or-Parallelism

4.1.1 Theoretical Results

The literature on Or-parallel execution of Prolog is extensive and it is primarily focused on the development of solutions for the *environment representation problem*. More than twenty different approaches have been presented in the literature to address this problem.

From the theoretical point of view, the environment representation problem has been formalized as a data structure problem over labeled trees (Ranjan et al. 1999). The Or-parallel execution of a program can be abstracted as building a labeled tree,[4] using the following operations: **(1)** *create_tree*(γ), which creates a tree containing only a root with label γ; **(2)** *expand*(u, γ_1, γ_2), which, given a leaf u and two labels γ_1 and γ_2, creates two new nodes (one per label) and adds them as children of u; **(3)** *remove*(u) which, given a leaf u of the tree, removes it from the tree.

The environment representation problem is associated to the management of variables and their bindings. This can be modeled as attributes of the nodes in the tree, assuming a set of attributes Γ. At each node u, three operations are possible: **(1)** *assign*(α, u) which associates the label α to node u; **(2)** *dereference*(α, u) which identifies the nearest ancestor v of u in which the operation *assign*(α, v) has been performed; **(3)** *alias*(α, β, u) which requires that any *dereference* operation for α in any descendant of u in the tree produces the same result as the *dereference* of β. This abstraction formalizes the informal considerations presented in the existing literature and can be used to classify the methods to address the environment representation problem (Gupta and Jayaraman 1990).

It has been demonstrated that it is impossible to derive a solution to the environment representation problem which achieves a constant-time solution to all the operations described above. In particular, Ranjan et al. (1999) demonstrated that the overall problem has a lower-bound complexity of $\Omega(\log n)$ on pure pointer machines, where n is the maximum number of nodes appearing in the tree. The investigation by Pontelli et al. (2002) provides also an optimal theoretical solution, achieving the complexity of $O(\log n)$; the solution makes use of a combination of generalized linked lists, which allow us to insert, delete, and compare positions of nodes in the list, and AVL trees. Even though the solution is theoretical, it suggests the possibility of improvement over existing methodologies, which have a complexity of $O(n)$.

4.1.2 Recent Solutions to the Environment Representation Problem

As introduced in Section 3, before 2000, the state of the art in Or-parallel Prolog systems was achieved using one of two approaches: *stack copying* (as used in the Muse system

[4] Without loss of generality, we assume trees to be binary.

and in ACE (Ali and Karlsson 1990b; Pontelli and Gupta 1997)), and *binding arrays* (as used in the Aurora system (Lusk et al. 1990)). These techniques provided comparable benefits and fostered a wealth of additional research (e.g., in the area of scheduling (de Castro Dutra 1994; Beaumont and Warren 1993; Ali and Karlsson 1992)). The advent of distributed computing architectures, especially Beowulf clusters, led researchers to investigate the development of Or-parallel Prolog systems in absence of shared memory. The two approaches above are not immediately suitable for distributed memory architectures, as they require some level of data sharing between workers[5] during the execution. For example, in the case of stack copying, *shared frames* need to be created to coordinate workers' access to unexplored choice point alternatives.

The PALS system (Villaverde et al. 2001b; Pontelli et al. 2007; Gupta and Pontelli 1999) introduced the concept of *stack splitting* as a new environment representation methodology suitable for distributed memory systems. Stack splitting replaces the dynamic synchronization among workers, e.g., as in the access to shared choice points during backtracking in the binding-arrays method, with a "*static*" partitioning of the unexplored alternatives between workers. Such partitioning is performed each time two workers collaborate to share unexplored branches of the resolution tree. Thus, stack splitting allows us to remove synchronization requirements by preemptively partitioning the remaining unexplored alternatives at the moment of sharing. The splitting allows both workers to proceed, each executing its branch of the computation, without any need for further synchronization when accessing parts of the resolution tree which are common among the workers.

The original definition of stack splitting (Villaverde et al. 2001a; Pontelli et al. 2006) explored two strategies of splitting. *Vertical stack splitting* (Figure 3–top) is suitable for computations where branches of the resolution tree are highly unbalanced and where choice points tend to have a small number of alternatives (e.g., binary trees). In this case, the available choice points are alternated between the two sharing workers—e.g., the first worker keeps the first, third, fifth, etc. choice point with unexplored alternatives, while the second worker keeps the second, fourth, sixth, etc. In the end, each worker ends up with approximately half of the choice points with unexplored alternatives. The alternation between choice points kept and choice points given away is meant to improve the chance of a balanced distribution of work between the two workers. *Horizontal stack splitting* has been, instead, designed to support sharing in the case of computations with few choice points, each with a large number of unexplored alternatives. In this case, the two workers partition the unexplored alternatives within each available choice point (Figure 3–middle).

Follow-up research has explored additional variations of stack splitting. *Diagonal stack splitting* (Rocha et al. 2003) provides a combination of horizontal and vertical stack splitting. *Middle stack splitting*, also known as *Half stack splitting* (Villaverde et al. 2003), is similar to vertical stack splitting, but with the difference that one worker keeps the top half of the choice points while the second worker keeps the bottom half. The advantage of this methodology is its efficient implementation and the ability to know the respective position of the two workers in the resolution tree immediately after the

[5] We use the generic term *worker* to denote a computing entity in a parallel Prolog system.

sharing operation—a useful property in order to manage side-effects and other order-sensitive predicates (Figure 3–bottom).

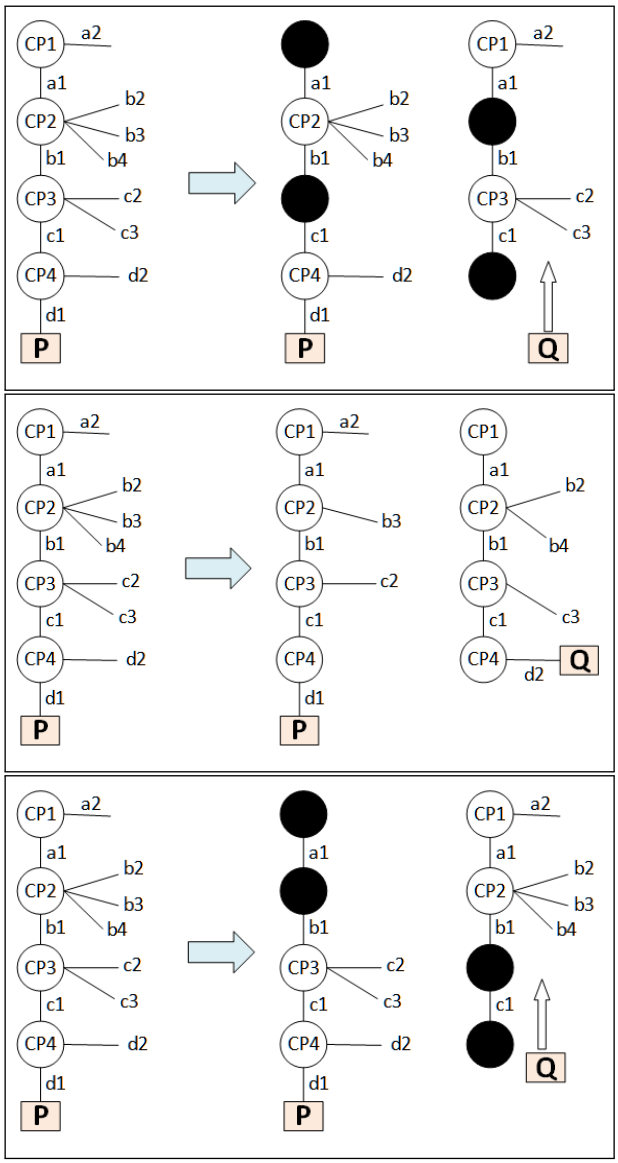

Fig. 3: From top to bottom: Vertical Splitting, Horizontal Splitting, Middle Splitting. Workers visit the resolution tree in depth first order; CP stands for choice point; CP1 is the "root" of the tree. P and Q denote different workers. A black node means that the choice point has no alternatives available. A connection between P/Q and a node indicates that the worker is executing the given alternative of the choice point. The arrow denotes immediate backtracking for the worker stealing work.

Even though stack splitting has been originally designed to sustain Or-parallelism on distributed memory architectures, this methodology provides advantages also on shared

memory platforms. The ACE system (Gupta and Pontelli 1999) demonstrated that stack-splitting outperforms stack-copying in a variety of benchmarks. A microprocessor-level simulation, using the Simics simulator, identified an improved cache behavior as the reason for the superior performance in shared memory platforms (Pontelli and Gupta 1999).

4.1.3 Systems and Implementations

The original stack splitting methodology has been developed in the context of the PALS project—an investigation of execution of Prolog on distributed memory machines. The PALS system introduces stack splitting by modifying the ALS Prolog system (Applied Logic Systems, Inc 2021), resulting in a highly efficient and scalable implementation. PALS explores not only novel techniques for the environment representation problem, but also the challenges associated to management of side-effects (Villaverde et al. 2003) and different scheduling strategies (Villaverde and Pontelli 2004). PALS operates on Beowulf clusters, mapping Prolog workers to processes on different nodes of a cluster and using MPI for communication between them. PALS implements vertical, horizontal, and middle stack splitting.

Santos Costa et al. (2010) implemented Or-parallelism on a multi-threaded implementation of YAP Prolog (ThOr). Their implementation relies on YapOr (Rocha et al. 1999b) which is based on the stack copying model. A multi-threaded implementation can exploit low-cost parallel architectures (such as common multi-core processors).

Santos and Rocha (2013; 2016) extended this approach to systems composed of clusters of multi-core processors—addressing the challenge of dealing with the combination of shared and distributed memory architectures. They propose a layered model based on *two levels* of workers, single workers and teams of workers, and the ability to exploit different scheduling strategies, for distributing work among teams and among the workers inside a team. A team of workers is formed by workers which share the same memory address space (workers executing in different computer nodes cannot belong to the same team). A computer node can contain more than one team. Several combinations of scheduling strategies are allowed. For distributed memory clusters of multi-cores, only static stack splitting is allowed for distributing work among teams. Inside a team, static and dynamic scheduling can be selected. The use of static stack splitting techniques (horizontal, vertical, diagonal, and so on) in shared memory architectures is described by Vieira et al. (2012). As far as dynamic stack splitting techniques, the authors rely on the *or-frame* data structures originally introduced in the Muse system.

4.2 And-Parallelism

4.2.1 Backtracking in Independent And-Parallel Prolog Systems

As explained in Section 3, Independent And-parallel (IAP) Prolog systems allow the parallel execution of subgoals which do not interfere with each other at run time, even if they produce multiple answers. Section 3 also pointed out that one of the most important areas of research in the context of IAP systems is the implementation of backtracking among parallel subgoals. The basic issues involved in this process have been addressed by Hermenegildo (1986a) and Hermenegildo and Nasr (1986). These works identify the

main challenges of backtracking in IAP, such as trapped nondeterministic subgoals and garbage slots. These concepts have been further developed and improved by Shen and Hermenegildo (1996), Pontelli et al. (1996), and Carro (2001).

An alternative and more efficient model for backtracking in IAP has been proposed by Pontelli and Gupta (2001). The implementation model is an extension of the original backtracking scheme developed by Hermenegildo and Nasr (1986) and includes a memory organization scheme and various optimizations to reduce communication and overhead. It also makes use of an adaptation of the ACE abstract interpretation-based analyzer (Pontelli et al. 1997), derived from that of the Ciao/&-Prolog system (Bueno et al. 1999; Muthukumar et al. 1999). In particular, detection and special treatment of *external variables* (i.e., variables appearing in the parallel call but created before the call itself) enable increased independence during backtracking. The results obtained show that speedups achieved during forward execution are not lost in heavy backtracking, and the "super-linear" speedups that can be obtained thanks to the semi-intelligent nature of backtracking in independent And-parallelism are also preserved. [6]

A common aspect of most traditional IAP implementations that support backtracking over parallel subgoals is the use of recomputation of answers and sequential ordering of subgoals during backtracking. While this can, in principle, simplify the implementation, recomputation can be inefficient if the granularity of the parallel subgoals is large and they produce several answers, while sequentially ordered backtracking limits parallelism. An alternative parallel backtracking model has been proposed by Chico de Guzmán et al. (2011) which features parallel out-of-order backtracking and relies on answer memoization to reuse and combine answers. Whenever a parallel subgoal backtracks, its siblings also perform backtracking after storing the bindings generated by previous answers, which are reinstalled when combining answers. In order to not penalize forward execution, non-speculative And-parallel subgoals which have not been executed yet take precedence over sibling subgoals which could be backtracked over. This approach brings performance advantages and simplifies the implementation of parallel backtracking.

The remaining challenges associated to trapped non-deterministic subgoals and garbage slots have been addressed by Chico de Guzmán et al.(2012); the proposed approach builds on the idea of a single stack reordering operation, that offers several advantages and tradeoffs over previous proposals. While the implementation of the stack reordering operation itself is not simple, in return it frees the scheduler from the constraints imposed by other schemes. As a result, the scheduler and the rest of the run-time machinery can safely ignore the trapped subgoal and garbage slot problems and their implementation is greatly simplified. Also, standard sequential execution remains unaffected.

4.2.2 High-level implementation of Unrestricted And-Parallelism

Original implementations of independent And-parallelism, as demonstrated in the &-Prolog and the &ACE systems, relied on low-level manipulations of the underlying

[6] This is really due to the change in the backtracking algorithm that independent And-parallel systems implement, which is a simple form of intelligent backtracking that takes advantage of the independence information.

abstract machine—typically some variant of the Warren Abstract Machine. This low-level approach is necessary in order to reduce execution overheads related to the management of parallelism. While this approach has been extensively used and produced highly efficient implementations, it leads to highly complex implementations, often hard to maintain and not able to keep up with improvements in sequential implementation technology.

High-level implementation. The improved performance of processors and the introduction of advanced compilation techniques have opened the doors to an alternative approach to the implementation of parallelism—by describing parallel execution models at a high level, in terms of Prolog predicates. The concept builds on the identification of a small collection of built-in predicates and their use to encode, as meta-programs, high-level models of parallelism (Casas 2008; Casas et al. 2008a; Casas et al. 2008b). The concept of implementing a form of parallelism via meta-programming encodings is not new (see for example the papers by Codish and Shapiro (1986), Pontelli and Gupta (1995), and Hermenegildo et al. (1995)), but it has been brought to full fruition only recently, in terms of a full implementation using a minimal set of core predicates. This is done by implementing in the engine a comparatively small number of concurrency-related primitives which take care of lower-level tasks, such as locking, stack set management, thread creation and management, etc. The implementation of parallel models (e.g., independent And-parallelism) is then realized through meta-programs that make use of such primitives. The approach does not eliminate altogether modifications to the abstract machine, but it does greatly simplify and encapsulate them, and it also facilitates experimenting with different alternative models of parallelism.

This approach also supports the implementation of flexible solutions for some of the main problems found in And-parallel implementations. In fact, the solutions presented by Chico de Guzmán et al. (2011; 2012) have all been implemented taking advantage of the flexibility afforded by this new approach. The experiments also show that, although such source-level implementation of parallelism introduces overheads, the performance penalties are reasonable, especially if paired with some form of granularity control. This is the (unrestricted) IAP implementation supported currently by the Ciao Prolog system.

Unrestricted And-Parallelism. Another interesting aspect of the approach by Casas et al. is that it facilitates the implementation of alternative models of And-parallelism. In particular, this approach has facilitated the exploration of *unrestricted And-Parallelism*, i.e., a form of IAP which does not follow the traditional fork-join structure used in the previous IAP systems. This flexibility has served as target for new parallelizers for unrestricted IAP (see Section 4.3). This work also shows that the availability of unrestricted parallelism contributes to improved parallel performance.

The high-level level primitives used in this system to express unrestricted And-parallelism are those described in the papers by Hermenegildo et al. (1995), Cabeza and Hermenegildo (1996), and Cabeza (2004):

- `G &> H` schedules subgoal `G` for parallel execution and continues with the code after `G`. `H` is a *handle* that provides access to the state of subgoal `G`.
- `H <&` waits for the subgoal associated with `H` (`G`, in the previous item) to finish. At

that point, all bindings G could possibly generate are ready, since G has reached a solution. Assuming subgoal independence between G and the calls performed while G was being executed, no binding conflicts will arise. This point is also the "anchor" for any backtracking performed by G.

Note that, with the previous definitions, the &/2 operator can be expressed as:

```
A & B :- A &> H, call(B), H <&.
```

The approach shares some similarities with the concept of futures in parallel functional languages (Flanagan and Felleisen 1995; Halstead Jr. 1985). A future is meant to hold the return value of a function so that a consumer can wait for its complete evaluation. However, the notions of "return value" and "complete evaluation" do not make sense when logic variables are present. Instead, H <& waits for the moment when the producer subgoal has completed execution, and the "received values" (typically a tuple) will be whatever (possibly partial) instantiations have been produced by such subgoal.

The new operators are very flexible, allowing the encoding of And-parallel executions which are not tied to the fork-join model implicit in the & operator. In particular, one can interleave execution of subgoals to ensure that variables are bound and thus enable an increasing level of parallelism. For example, in a computation where it has been determined (through, e.g., global analysis) that p(A,B) produces the values for A,B, q(C) produces the value for C, while r(A) and s(C,B) consume such values, the following encoding provides a higher degree of parallelism than a traditional fork-join structure:

```
q(C), p(A,B) &> H1,
        r(A) &> H2,
        H1 <&,
       s(C,B), H2 <&.
```

The &-Prolog engine implemented these constructs natively offering high parallel performance. The Ciao Prolog system includes the higher-level (Prolog) management of threads (Casas et al. 2008a; Casas et al. 2008b); for example, the &> operator can be encoded as:

```
Goal &> Handle :- add_goal(Goal, nondet, Handle),
                  undo(cancellation(Handle)),
                  release_suspended_thread.
```

which adds the Goal to a goal queue and enables one thread if available.

In closing, let us observe that these constructs are very general and can be used to explore a wide variety of parallelization schemes, including schemes that do not respect the original observable semantics of Prolog.

4.2.3 Dependent And-Parallelism

The concept of dependent And-parallelism has not been extensively investigated beyond the ideas already presented in the previous survey by Gupta et al. (2001) and briefly reviewed in Section 3. The state-of-the-art at the beginning of 2000 is represented by systems like ACE with the introduction of the filtered binding model (Gupta and Pontelli

1997), which supports the dynamic management of producer and consumer subgoals for variables shared among And-parallel subgoals. Models like this and previously proposed ones (e.g., the DDAS model (Shen 1996)) suffer from high complexity of implementation, which complicates their maintenance and evolution. While not further explored, we envision the extension of models like those proposed in Section 4.2.2 to represent a more viable approach for implementation of dependent And-parallelism, as hinted at already, e.g., by Hermenegildo et al. (1995).

On the other hand, the techniques for dependent And-parallelism developed for Prolog have found use and extension in logic programming systems which have evolved from Prolog, such as Mercury and the EAM, as described in the following sections.

Mercury. A simplified architecture for dependent And-parallelism can be found in the parallel implementation of Mercury (Conway 2002). The proposed architecture uses a multi-threaded implementation to create a number of workers, which concurrently execute different subgoals. The implementation takes advantage of the fact that the Mercury language requires each variable to have a single producer designated at compile time. Thus, in a legal Mercury program, it is guaranteed that only a single point in the code will try to bind a given free variable. As a result, producers and consumers are known at time of execution and do not require a complex dynamic management for shared variables. This architecture introduces sophisticated mechanisms to support context switch of suspended subgoals. At the language level, Mercury is extended with an & operator analogous to the one used in And-parallel Prolog systems, to identify subgoals meant for parallel execution.

The initial parallel implementation of Mercury restricted the parallel execution to non-communicating subgoals (Conway 2002). This was later extended to an implementation that supports communication between subgoals (Bone 2012). The work by Bone provides a number of architectural extensions and optimizations. The system proposed by Bone et al. (2011; 2012) replaces the use of a single centralized task queue with a collection of distributed local work queues and work-stealing models, similarly to other And-parallel implementations of Prolog. This was dictated by the need to reduce the bottleneck caused by a single-access centralized task queue. Another component of this system is the use of a Mercury-specific cost analysis to detect, at compile-time, promising subgoals for parallel execution.

Extended Andorra Model. Basic and Extended Andorra Model (EAM) are presented at the end of Section 3. In simple terms, the EAM allows subgoals to proceed concurrently as long as they are deterministic or as long as they do not request to bind non-deterministically an external variable (this has can also be seen as a fine-grain form of independence). When non-determinism in the latter case is present, the computation can split (in a form of Or-parallelism).

The computation in the EAM can be described as a series of rewriting operations applied to an and-or tree. The tree includes two types of boxes: *and-boxes*, representing conjunctions of subgoals, and *or-boxes*, representing alternative clauses for a selected literal. The BEAM (Lopes et al. 2003; Lopes et al. 2004) is a parallel implementation of such a rewriting process, enhanced with a collection of control rules to enhance efficiency (e.g., by delaying splitting, the operation which realizes Or-parallelism, until no deterministic

steps are possible). The BEAM provides a shared-memory realization of the EAM, using an approach called RAINBOW (Rapid And-parallelism with INdependence Built upon Or-parallel Work). The BEAM supports independent And-parallelism enhanced with determinacy, along with different implementation models of Or-parallelism (e.g., version vectors and binding arrays).

4.3 Static Analysis for Parallelism

Static analysis, generally based on Cousots' theory of abstract interpretation (Cousot and Cousot 1977), is a supporting technology in a number of areas related to parallelism in logic programming, and especially in the process of automatic parallelization of logic programs to exploit And-parallelism. In fact, logic programming pioneered the development of abstract interpretation-based analyzers —such as MA3 and Ms (Warren et al. 1988), PLAI (Muthukumar and Hermenegildo 1990;1992;García de la Banda et al. 1996), or GAIA (Le Charlier and Van Hentenryck 1994), and the extension of this style of analysis to CLP/CHCs (García de la Banda and Hermenegildo 1993; García de la Banda et al. 1996; Kelly et al. 1998). Arguably, the MA3 and PLAI parallelizers for independent And-parallelism were the first complete, practical applications of abstract interpretation in a working compiler (see, e.g., Van Roy (1994)).

Static analysis is an area that has seen significant progress in this period. In this section we briefly focus on the advances made in static analysis for parallelism in logic programs since the survey paper by Gupta et al. (2001), with a quick view on main older results.

Sharing analyses. One of the main instrumental properties for automatic (And-)parallelization is the safe approximation of the sharing (aliasing) patterns between variables. Apart from being necessary for obtaining analyses that are at the same time precise and correct, variable sharing is a basic component of essentially all notions of independence (Hermenegildo and Rossi 1995; Bueno et al. 1999; Muthukumar et al. 1999; Marriott et al. 1994; García de la Banda et al. 1995;2000). The main abstract domains developed for detecting independence typically seek to capture either set-sharing (Muthukumar and Hermenegildo 1989;1991;1992;Jacobs and Langen 1989; Codish et al. 2000; Hill et al. 2002) or pair-sharing (Søndergaard 1986; Lagoon and Stuckey 2002; Bueno and García de la Banda 2004).

Results in this area implied the development of *widenings* (Cousot and Cousot 1977) and/or alternative more efficient representations and abstract domain operations for sharing domains. Widenings are used in abstract interpretation-based analyses to support infinite abstract domains and also to reduce the cost of complex domains, such as sharing. In both cases the essence of the technique is to lose precision in return for reduced analysis time (at the limit, termination) by generalizing at some point in the analysis to a larger abstract value. This implies arriving at fixpoints that are less precise than the minimal fixpoint, but still safe approximations of the concrete values.

In this line, widenings of sharing have been proposed for example by Zaffanella et al. (1999), by performing a widening switch to a modification of Fecht's domain (Fecht 1996) and incorporating other techniques such as combining with the `Pos` domain. Navas et al. (2006) proposed an encoding of sharing that represents sets of variables that have total sharing (which gives rise to large sharing sets) in a compact way and also widens

abstract values to this representation when they are close to total sharing. Li et al. (2006) developed a lazy technique that postpones computations of sharing sets until they are really needed. Trias et al. (2008) proposed an encoding that uses negative information representation techniques to switch to a dual representation for certain variables when there is a high degree of sharing among them. Also, Méndez-Lojo et al. (2008) proposes and studies a representation using ZBDDs.

Determinism and non-failure analyses. Inference of determinacy (Lopez-Garcia et al. 2010) and non-failure (Bueno et al. 2004), including multivariant (i.e., context/path-sensitive) analyses, based on the PLAI framework have been investigated. These analyses are instrumental in And-parallelism, since if goals can be determined to not fail and/or be deterministic, significant simplifications can be performed in the handling of their parallel execution. This is due to the inherent complexity in handling backtracking across parallel goals (Section 4.2), significant parts of which can be avoided if such information is available.

Cost analysis and granularity control. The advances in determinism and non-failure analyses are also useful for improving the precision of cost analysis (Debray et al. 1990;1994;1997), which is instrumental in parallelism for performing task granularity control (Debray et al. 1990; Lopez-Garcia et al. 1996; Lopez-Garcia 2000). Such analyses have been extended to estimate actual execution time (Mera et al. 2008), rather than steps, as in previous work, which is arguably more relevant to automatic parallelization and granularity control. Techniques have also been developed for fuzzy granularity control (Trigo de la Vega et al. 2010). Another important related line of work has been the static inference of the cost of *parallelized programs* (Klemen et al. 2020). The base cost analyses have also been extended to be parametric (Navas et al. 2007) (where the analysis can track user-defined resources and compound resources) and multivariant, formulated as an abstract domain (Serrano et al. 2014).

Static analysis for parallelization algorithms. There has also been progress in the development of improved, static analysis-based parallelization algorithms that allow automatic parallelization for non-restricted And-parallelism (Casas et al. 2007; Casas 2008) (i.e., not limited to fork-join structures, as supported by Casas et al. (2008a), see Section 4.2.2). Also, improved parallelization algorithms for non-strict independence (Hermenegildo and Rossi 1995) using sharing and freeness information have been proposed (Cabeza and Hermenegildo 2009).

In another line of work, by Vidal (2012), a novel technique has been presented for generating annotations for independent And-parallelism based on partial evaluation. A partial evaluation procedure is augmented with (run-time) groundness and variable sharing information so that parallel conjunctions are added to the residual clauses when the conditions for independence are met. The results are parallel annotated programs which are shown to be capable of achieving interesting speedups.

Improvements in static analysis related to scalability are directly relevant to improving the practicality of automatic parallelization techniques. In this context, much work has been done in improving these aspects for LP and CLP analyses, including combined

modular and incremental analysis, assertion-guided multivariant analysis, and analysis of programs with assertions and open predicates (Garcia-Contreras et al. 2018;20192020).

Run-time checking overhead. The reduction of the overhead implied by run-time tests (through caching techniques (Stulova et al. 2015; Stulova 2018) and also static analysis (Stulova et al. 2018), similarly to what done in the works by Bueno et al. (1999) and Puebla and Hermenegildo (1999)), as well as in generating static performance guarantees for programs with run-time checks (Klemen et al. 2018), have been subject of active research. This work was done in the context of run-time tests for assertions but it is directly relevant to run-time checking for conditional And-parallelism.

Applications to other paradigms and areas. The techniques developed for LP/CLP parallelization have been used to support the parallelization of other paradigms (Hermenegildo 2000). This is a large topic that goes beyond the scope of this survey, but examples include the application of pair-sharing (Secci and Spoto 2005) and set-sharing (Méndez-Lojo and Hermenegildo 2008) to object-oriented programs, sharing analysis of arrays, collections, and recursive structures (Marron et al. 2008), identifying logically related heap regions (Marron et al. 2009), identification of heap-carried dependencies (Marron et al. 2008), or context-sensitive shape analysis (Marron et al. 2006;2008).

As stated before, the demand for precise global analysis and transformation stemming from automatic parallelization spurred the development of the first abstract interpretation-based "production" analyzers. Having these systems readily available led to the early realization that analysis and transformation are very useful also in program verification and static debugging, as illustrated by the pioneering Ciao Prolog system (Bueno et al. 1997;Hermenegildo et al. 199920052012). During the past two decades, the analysis and verification of a large variety of other programming paradigms —including imperative, functional, object-oriented, and concurrent ones— using LP/CLP-related analysis techniques, has received significant interest, and many verification approaches and tools have recently been implemented which are based in one way or another in a translation into LP/CLP (referred to in this context as Constrained Horn Clauses –CHCs) (Peralta et al. 1998;Henriksen and Gallagher 2006;Méndez-Lojo et al. 2007;Navas et al. 2008;2009;Albert et al. 2007;Gómez-Zamalloa et al. 2009;Grebenshchikov et al. 2012;Gurfinkel et al. 2015;De Angelis et al. 2015;Kahsai et al. 2016;Lopez-Garcia et al. 2018;Liqat et al. 2014;2016;Gallagher et al. 2020). The main reason is that LP and CLP are effective as languages for specifying program semantics and program properties. De Angelis et al. 2022 offers an up-to-date, comprehensive survey of this approach.

4.4 Parallelism and Tabling

Tabling, introduced by Chen and Warren (1996), is a powerful implementation technique that overcomes some limitations of traditional Prolog systems in dealing with recursion and redundant sub-computations. Tabling has become a popular and successful technique thanks to the ground-breaking work in the XSB Prolog system and in the SLG-WAM engine (Sagonas and Swift 1998). The success of SLG-WAM led to several alternative implementations that differ in the execution rule, in the data-structures used

to implement tabling, and in the changes to the underlying Prolog engine (Swift and Warren 2012; Santos Costa et al. 2012; Zhou 2012; Guo and Gupta 2001; Somogyi and Sagonas 2006; Chico de Guzmán et al. 2008; Desouter et al. 2015; Zhou et al. 2015).

Tabling is a refinement of SLD resolution that stems from one simple idea: programs are evaluated by saving intermediate answers for tabled subgoals so that they can be reused when a *similar call* appears during the resolution process. First calls to tabled subgoals are considered *generators* and are evaluated as usual, using SLD resolution, but their answers are stored in a global data space, called the *table space*. Similar calls are called *consumers* and are resolved by consuming the answers already stored for the corresponding generator, instead of re-evaluating them against the program clauses. During this process, as further new answers are found, they are stored in their table entries and later returned to all similar calls. Call similarity thus determines if a subgoal will produce their own answers or if it will consume answers from a generator call.

A key procedure in tabled evaluation is the *completion* procedure, which determines whether a subgoal is *completely evaluated*. A subgoal is said to be completely evaluated when all its possible resolutions have been performed, i.e., when no more answers can be generated and all consumers have consumed all the available answers. A number of subgoals may be mutually dependent, forming a *strongly connected component* (*SCC*), and therefore can only be completed together. In this case, completion is performed by the *leader* of the SCC, which is the oldest generator subgoal in the SCC, when all possible resolutions have been made for all subgoals in the SCC (Sagonas and Swift 1998).

4.4.1 Parallel Tabling

The first proposal on how to exploit implicit parallelism in tabling is due to Freire et al. (1995) that has been not implemented later in available systems. More recent approaches are based on reordering the execution of alternatives corresponding to multiple clauses that match a goal in the search tree (Zhou et al. 2008; Guo and Gupta 2001). Guo and Gupta's approach is a tabling scheme based on *dynamic reordering of alternatives with variant calls*: both the answers to tabled subgoals and the (looping) alternatives leading to variant calls are tabled. After exploiting all matching clauses, the subgoal enters a looping state, where the looping alternatives start being tried repeatedly until a fix-point is reached. The process of retrying alternatives may cause redundant recomputations of the non-tabled subgoals that appear in the body of a looping alternative. It may also cause redundant consumption of answers if the body of a looping alternative contains several variant subgoal call. Within this model, the traditional forms of parallelism can still be exploited and the looping alternatives can be seen as extra unexplored choice point alternatives. However, parallelism may not come so naturally as for SLD evaluations as, by nature, tabling implies sequentiality – an answer can not be consumed before being found and consuming an answer is a way to find new ones – which can lead to more recomputations of the looping alternatives until reaching a fix-point.

The first system to implement support for the combination of tabling with some form of parallelism was the YAP Prolog system (Santos Costa et al. 2012). A first design, named the OPTYap design (Rocha et al. 1999a), combines the tabling-based SLG-WAM execution model with implicit Or-parallelism using shared memory processes. A second design supports explicit concurrent tabled evaluation using threads (Areias and Rocha

2012), where from the threads point of view the tables are private, but at the engine level the tables are shared among threads using a *common table space*. To the best of our knowledge, YAP's designs are the only effective implementations with experimental results showing good performance on shared-memory parallel architectures.

The XSB system also implements some sort of explicit concurrent tabled evaluation using threads that extends the default SLG-WAM execution model with a *shared tables design* (Marques and Swift 2008). It uses a semi-naive approach that, when a set of subgoals computed by different threads is mutually dependent, then a *usurpation operation* synchronizes threads and a single thread assumes the computation of all subgoals, turning the remaining threads into consumer threads. The design ensures the correct execution of concurrent sub-computations but the experimental results showed some limitations (Marques et al. 2010).

Other proposals for concurrent tabling relies on a distributed memory model. Hu (1997) was the first to formulate a method for distributed tabled evaluation termed *Multi-Processor SLG (SLGMP)*. As in the approach of Freire et al. (1995), each worker gets a single subgoal and it is responsible for fully exploiting its search tree and obtain the complete set of answers. One of the main contributions of SLGMP is its controlled scheme of propagation of subgoal dependencies in order to safely perform distributed completion.

A different approach for distributed tabling was proposed by Damásio (2000). The architecture for this proposal relies on four types of components: a *goal manager* that interfaces with the outside world; a *table manager* that selects the clients for storing tables; *table storage clients* that keep the consumers and answers of tables; and *prover clients* that perform the evaluation. An interesting aspect of this proposal is the completion detection algorithm. It is based on a classical credit recovery algorithm (Mattern 1989) for distributed termination detection. Dependencies among subgoals are not propagated and, instead, a controller client, associated with each SCC, controls the credits for its SCC and detects completion if the credits reach the zero value. An implementation prototype has also been developed, but further analysis is required.

4.4.2 YAP Prolog

We focus here on the already cited YAP Prolog system that provides the ground technology for both implicit and explicit concurrent tabled evaluation, but separately. From the user's point of view, tabling can be enabled through the use of single directives of the form "*:- table p/n*", meaning that common sub-computations for *p/n* will be synchronized and shared between workers at the engine level, i.e., at the level of the tables where the results for such sub-computations are stored. Implicit concurrent tabled evaluation can be triggered if using the OPTYap design (Rocha et al. 2005), which exploits implicit Or-parallelism using shared memory processes. Explicit concurrent tabled evaluation can be triggered if using the thread-based implementation (Areias and Rocha 2012); in this case, the user still needs to implement the thread management and scheduler policy for task distribution.

Implicit Or-Parallel Tabled Evaluation. The OPTYap system builds on the YapOr (Rocha et al. 1999b) and YapTab (Rocha et al. 2000) engines. YapOr extends YAP's sequential

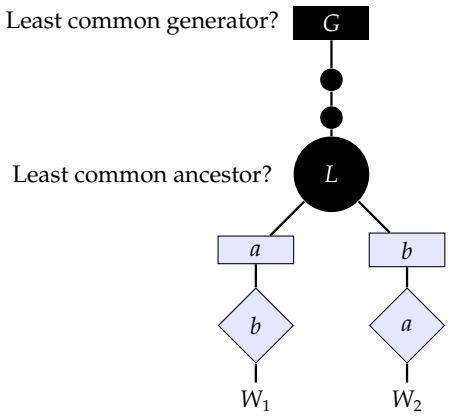

Fig. 4: Public completion scheme. Black nodes are public nodes, rectangle nodes are generators, rhombus nodes are consumers. W_1 and W_2 are the workers.

engine to support implicit Or-parallelism based on the environment copying model. YapTab extends YAP's execution model to support (sequential) tabled evaluation. In the OPTYap design, tabling is the base component of the system as, most of the time, each worker behaves as a full sequential tabling engine. The Or-parallel component of the system is triggered to allow synchronized access to the shared region of the search space or to schedule work. Work sharing is implemented through stack copying with *incremental copying* of the stacks (Ali and Karlsson 1990a).

Workers exploiting the public (shared) region of the search space must synchronize to ensure the correctness of the tabling operations. Synchronization is required: (i) when backtracking to public generator or interior (non-tabled) nodes to take the next available alternative; (ii) when backtracking to public consumer nodes to take the next unconsumed answer; or, (iii) when inserting new answers into the table space. Moreover, since Or-parallel systems can execute alternatives early, the relative positions of generator and consumer nodes are not as clear as for sequential tabling. As a result, it is possible that generators will execute earlier, and in a different branch than in sequential execution. Or that different workers may execute the generator and the consumer calls. Or, in the worst case, workers may have consumer nodes while not having the corresponding generators in their branches. This induces complex dependencies between workers, hence requiring a more elaborated and specific *public completion* operation (Rocha et al. 2001).

Figure 4 illustrates this kind of dependencies. Starting from a common public node, worker `W1` takes the leftmost alternative while worker `W2` takes the rightmost. While exploiting their alternatives, `W1` calls a tabled subgoal `a` and `W2` calls a tabled subgoal `b`. As this is the first call to both subgoals, a generator node is stored for each one. Next, each worker calls the tabled subgoal firstly called by the other, and consumer nodes are therefore allocated. At that point, we may question at which (leader) node should we check for completion? In OPTYap, public completion is performed at the least common ancestor node (node `L` in Figure 4). But that leader node can be any type of node and not necessarily a generator node as in the case of sequential tabling.

Consider now the case where `W1` has explored all its private work and backtracks to the public leader node `L` common to `W2`. Since work is going on below `L`, `W1` cannot complete

its current SCC (which includes L, the generator node for a and the consumer node for b). The reason for this is that W2 can still influence W1's branch, for instance, by finding new answers for subgoal b. On the other hand, we would like to move W1 in the tree, say to node N, where there is available work and, for that, we may need to reset the stacks to the values in N. As a result, in order to allow W1 to continue execution, it becomes necessary to *suspend the SCC* at hand. This is the only case where Or-parallelism and tabling conflict in OPTYap (a worker needs to move in the tree above an uncompleted leader node). OPTYap's solution is to save the SCC's stacks to a proper space, leaving in L a reference to where the stacks were saved. These suspended computations are considered again when the remaining workers check for completion at L. To resume a suspended SCC a worker needs to copy the saved stacks to the correct position in its own stacks, and thus, it has to suspend its current SCC first. To minimize that, OPTYap adopts the strategy of resuming suspended SCCs *only when the worker finds itself at a leader node*, since this is a decision point where the worker either completes or suspends its current SCC. OPTYap's public completion algorithm and associated data structures is one of the major contributions of OPTYap's design.

Explicit Concurrent Tabled Evaluation. In YAP, threads run independently within their own execution stacks. For tabling, this means that each thread evaluation depends only on the computations being performed by itself, i.e., from the thread point of view, each thread has its own private tables but, at the engine level, YAP uses a *common table space* shared among all threads.

Figure 5 shows the general table space organization for a tabled predicate in YAP. At the entry level is the *table entry* data structure where the common information for the predicate is stored. This structure is allocated when a tabled predicate is being compiled, so that a pointer to the table entry can be included in the compiled code. This guarantees that further calls to the predicate will access the table space starting from the same point. Below the table entry, is the *subgoal trie structure*. Each different tabled subgoal call to the predicate at hand corresponds to a unique path through the subgoal trie structure, always starting from the table entry, passing by several subgoal trie data units, the *subgoal trie nodes*, and reaching a leaf data structure, the *subgoal frame*. The subgoal frame stores additional information about the subgoal and acts like an entry point to the *answer trie structure*. Each unique path through the answer trie data units, the *answer trie nodes*, corresponds to a different answer to the entry subgoal. To deal with multithreading tabling, YAP implements several designs with different degrees of sharing of the table space data structures.

We report here the main features of three sharing designs proposed in (Areias and Rocha 2012): *No-Sharing (NS)*, *Subgoal-Sharing (SS)*, and *Full-Sharing (FS)*.

NS was the starting design for multithreading tabling support in YAP: each thread allocates fully private tables and the table entry is extended with a *thread array*, where each thread has its own entry, which then points to the private subgoal tries, subgoal frames and answer tries for the thread.

In the SS design, the threads share part of the table space, namely, the subgoal trie structures are now shared among the threads and the leaf data structure in each subgoal trie path, instead of referring a subgoal frame, it now points to a thread array. Each entry in this array then points to private subgoal frames and answer trie structures. In

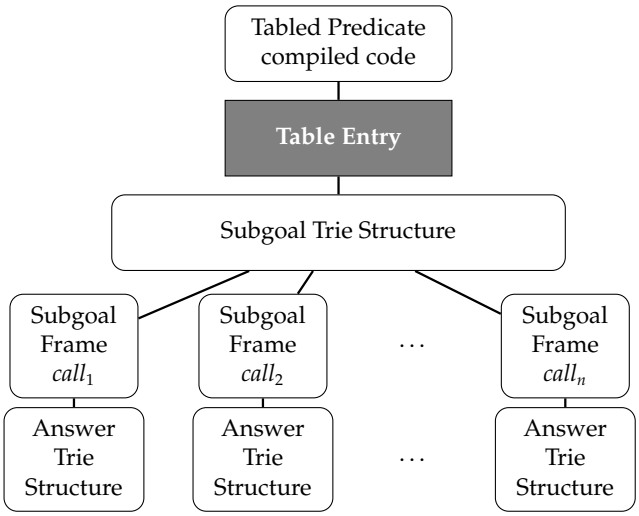

Fig. 5: YAP's table space organization

this design, concurrency among threads is restricted to the allocation of trie nodes on the subgoal trie structures. Tabled answers are still stored in the private answer trie structures of each thread. The *Partial Answer Sharing (PAS)* design (Areias and Rocha 2017) extends the SS design to allow threads to share answers. The idea is as follows: whenever a thread calls a new tabled subgoal, first it searches the table space to lookup if any other thread has already computed the full set of answers for that subgoal. If so, then the thread reuses the available answers, thus avoiding recomputing the subgoal call from scratch. Otherwise, it computes the subgoal itself. The first thread completing a subgoal call, shares the results by making them available (public) to the other threads. The PAS design avoids the usage of the thread array and instead it uses a list of private subgoal frames corresponding to the threads evaluating the subgoal call. In order to find the subgoal frame corresponding to a thread, we may have to pay an extra cost for navigating in the list but, once a subgoal frame is completed and made public, its access is immediate since it is moved to the beginning of the list.

Finally, the FS design tries to maximize the amount of data structures being shared. In this design, the answer tries and part of the subgoal frame information are also shared among threads. A new *subgoal entry* data structure stores the shared information for the subgoal, which includes access to the shared answer trie and to the thread array that keeps pointing to the subgoal frames, where the remaining private information is kept. In this design, concurrency among threads includes the access to the new subgoal entries and the allocation of tries nodes on the answer trie structures. Memory usage is reduced to a minimum and, since threads share the answer tries, answers inserted by a thread for a particular subgoal call are automatically made available to all other threads when they call the same subgoal. However, since different threads can be inserting answers in the same answer trie, when an answer already exists, it is not possible to determine if the answer is new or repeated for a particular thread without further support. This can be a problem if the tabling engine implements a *batched scheduling* strategy (Freire et al. 1996). To mitigate this problem, the *Private Answer Chaining (PAC)* design (Areias and

Rocha 2015) extends the FS design to keep track of the answers that were already found and propagated per thread and subgoal call.

4.4.3 Perspective on the Future

We believe that a challenging goal for the combination of tabling with parallelism is the design of a framework that integrates both implicit and explicit concurrent tabled evaluation, as described earlier, in a single tabling engine. This is a very complex task since we need to combine the explicit control required to launch, assign and schedule tasks to workers, with the built-in mechanisms for handling tabling and implicit concurrency, which cannot be controlled by the user.

In such a framework, a program begins as a single worker that executes sequentially until reaching a *parallel construct*. A parallel construct can then be used to trigger implicit or explicit concurrent tabled evaluation. If the parallel construct identifies a request for *explicit concurrent evaluation*, the execution model launches a set of additional workers to exploit concurrently a set of independent sub-computations (which may include tabled and non-tabled predicates). From the workers' point of view, each concurrent sub-computation computes its tables but, at the implementation level, the tables can be shared following YAP's design presented above. Otherwise, if the construct requires *implicit concurrent evaluation*, the execution model launches a set of additional workers to exploit in parallel a common sub-computation. Parallel execution is then handled implicitly by the execution model taking into account possible directive restrictions. For example, we may have directives to define the number of workers, the scheduling strategy to be used, load balancing policies, etc. By taking advantage of these parallel constructs, a user can write parallel logic programs from scratch or parallelize existing sequential programs, by incrementally pinpointing the sub-computations that can benefit from parallelism, using the available directives to test and fine tune the program in order to achieve the best performance. Combining the inherent implicit parallelism of Prolog with high-level parallel constructs will clearly enhance the expressiveness and declarative style of tabling and simplify concurrent programming.

5 Parallelism and Answer Set Programming

Answer Set Programming is a programming paradigm for knowledge representation and reasoning based on some key points: the use of negation as failure, the semantics based on stable models (also known as answer sets), and a bottom-up model computation on a ground version of the program. Many solvers for Answer Set Programming became available in the last decades. The work by Gebser et al. (2018) is a recent survey on ASP systems.

The presentation in this section starts with a quick review of parallelism in Datalog, the language for deductive Databases. Even though pure Datalog lacks negation (which is a crucial starting point for ASP) and uses implementation techniques which are different from ASP, we opt to start our conversation from Datalog as the original logic programming paradigm based on a bottom-up model computation—i.e., a paradigm where the result of the computation is the desired (i.e., minimal) model of a logic program. We then focus on extraction of parallelism from the ASP computation; in particular, we consider

parallelization of the search process used by ASP, parallelization of the grounding phase, the exploitation of portfolio parallelism, and conclude with some final considerations on other opportunities for parallelism in ASP. For other details on this topic the reader is referred to the survey by Dovier et al. (2018). Various forms of parallelism have been implemented in modern ASP solvers and experimented with in ASP Competitions (Gebser et al. 2020).

5.1 Parallelism and Datalog

Research on parallelization of Datalog inherited results and techniques developed in the field of relational DBMS, such as parallelization of relational operations and of SQL query evaluation. Particularly relevant are the approaches to parallelization of natural join operations, as they are at the core of the naive and semi-naive bottom-up computation in Datalog, and query optimization; the literature has explored such issues in the context of a variety of computing architectures (Wang et al. 2018; Diamos et al. 2013; Zeuch 2018; Zinn et al. 2016; Shehab et al. 2017).

Initial attempts towards the parallelization of Datalog appeared early in literature. Among many, we mention the works by Wolfson and Silberschatz (1988), Ganguly et al. (1992), and Zhang et al. (1995) which explore the execution of Datalog on distributed memory architectures. These approaches are mainly restricted to definite programs or, in the case of programs with negation, to stratified programs. The core idea consists in parallelizing the computation of the minimum model of a Datalog program, computed using the semi-naive bottom-up technique. Program rules are partitioned and assigned to the distributed workers. Communication between workers is implemented through explicit message passing. The different early proposals differ on the techniques used to distribute sets of rules different workers and management of the communication, used to exchange components of the minimal model as it is computed.

Similar approaches have been developed to operate on multi-core shared-memory machines—by exploring hash functions to partition computation of relations that guarantee the avoidance of locks (Yang et al. 2015). Significant speedups can be obtained by using as little synchronization as possible during the program evaluation. An example is the work by Martinez-Angeles et al. (2014), where the load of computing the model is distributed among the various GPU threads that can access and modify the data in the GPU shared memory.

In more recent years, various tools and systems have been developed to evaluate Datalog programs in parallel or distributed settings. Some of such parallel/distributed engines have been mainly designed to support declarative reasoning in application domains like declarative networking, program analysis, distributed social networking, security, and graph/data analytic. These approaches often extend plain Datalog with some form of aggregation and negation to meet the needs of the specific application domain.

Moustafa et al. (2016) propose *Datalography*, a bottom-up evaluation engine for Datalog tailored to graph analytics. The system enables the distributed evaluation of Datalog queries, also involving aggregates; the target architecture for this implementation is a BSP-style graph processing engine. Queries are compiled into sets of rules that can be

executed locally by the distributed workers. Optimizations are applied to partition the work and to minimize communication between workers.

A different approach to large-scale Datalog applications is adopted in the *Soufflé* system (Jordan et al. 2016; Nappa et al. 2019; Zhao et al. 2020). Efficient Datalog evaluation is obtained by compiling declarative specifications into C++ programs involving openMP directives that enable parallel execution on shared-memory multi-core architectures.

The *BigDatalog* system (Condie et al. 2018; Das and Zaniolo 2019; Shkapsky et al. 2016) supports scalable analytics and can be executed on distributed and multi-core architectures. The notion of *"pre-mappability"* (Zaniolo et al. 2017) is exploited to extend Datalog's fixpoint semantics in order to enable aggregates in recursion.

Seo et al. (2013) describe *Distributed SociaLite,* an extension of the SociaLite system for parallel and distributed large-scale graph analysis. In this case, users can specify how data should be partitioned and shared across workers in a cluster of multi-cores. The system optimizes the communication needed between workers.

In the *Myria* system (Wang et al. 2015) for large-scale data analytics, Datalog queries, possibly involving aggregates and recursion, are translated into parallel query plans to be executed in a shared-nothing multi-core cluster. Both synchronous and asynchronous evaluation schemes are possible.

Concerning the exploitation of parallelism in Datalog-based approaches to data analysis, we also mention *RecStep* (Fan et al. 2019), an implementation of a general-purpose Datalog engine built on top of a parallel RDBMS. The target architecture is a single-node multi-core system and the language extension of plain Datalog offers aggregates and stratified negation. *Yedalog* (Chin et al. 2015) extends Datalog to enable computations and analysis on large collections of semi-structured data. The *DeALS* system (see the paper by Shkapsky (2016) and the references therein) which supports standard SQL-like aggregates as well as user-defined aggregates, combined with (stratified) negation, has been ported to both multicore platforms and distributed memory systems (using the Spark library).

In Sections 6 and 7 we will also show other approaches to Datalog parallelism.

5.2 Search (Or-) Parallelism in ASP

The most popular ASP solvers proposed in the literature implement search processes that explore the search space of possible truth value assignments to the atoms of the ground program—directly or indirectly (e.g., through activation of program rules). This has prompted the study of parallelization of the search process, by supporting the concurrent exploration of different parts of the search space. This form of parallelization resembles the exploitation of Or-Parallelism in Prolog, as discussed earlier.

5.2.1 General Design

The concept of Or-parallelism (or Search parallelism) in ASP emerged early on in the history of the paradigm—the first popular ASP solvers appeared around 1997 (e.g., Smodels (Niemela and Simons 1997), DLV (Citrigno et al. 1997)) and the first reports of parallel ASP solvers were presented in 2001 (El-Khatib and Pontelli 2000; Finkel et al. 2001; Pontelli 2001). These first Or-parallel ASP implementations originated from modifications

of the Smodels inference engine. The intuitive structure of the algorithm underlying Smodels is illustrated in Algorithm 2. The procedure incrementally constructs a partial interpretation by identifying atoms as being true (added to S^+) or false (added to S^-). If all atoms of the Herbrand base of the program P (B_P) are assigned, the solution is returned. The *Expand* procedure is used to *deterministically* expand the partial interpretation constructed so far, using the clauses of the program to infer the truth value of other atoms. Intuitively, if $\langle S^+, S^- \rangle$ is the current partial interpretation, the *Expand* procedure determines a subset of

$$\langle \{A \mid P \cup S^+ \cup \{\neg B \mid B \in S^-\} \models A\}, \{A \mid P \cup S^+ \cup \{\neg B \mid B \in S^-\} \models \neg A\} \rangle$$

The implementation of *Expand* in Smodels uses inference rules which are equivalent to the computation of the well-founded model of the program $P \cup S^+ \cup \{\leftarrow p | p \in S^-\}$ (similar to the program transformations proposed by Brass et al. (2001)). The *Select_Atom* function heuristically selects an unassigned atom to assign next, while the *Choose* function creates a non-deterministic choice between the two following alternatives. Or-parallelism arises from the concurrent exploration of the alternatives generated by *Choose*—see also Figure 6.

Algorithm 2: Intuition of Smodels.

Input: A ground program P
Output: Answer Set

```
1  ⟨S+, S−⟩ = ⟨∅, ∅⟩
2  loop forever
3      ⟨S+, S−⟩ = Expand(P, ⟨S+, S−⟩)
4      if (S+ ∩ S− ≠ ∅) then
5          return Fail
6      if (S+ ∪ S− = BP) then
7          return ⟨S+, S−⟩
8      p = Select_Atom(P, ⟨S+, S−⟩)
9      Choose:
10             1: S+ = S+ ∪ {p}
11             2: S− = S− ∪ {p}
```

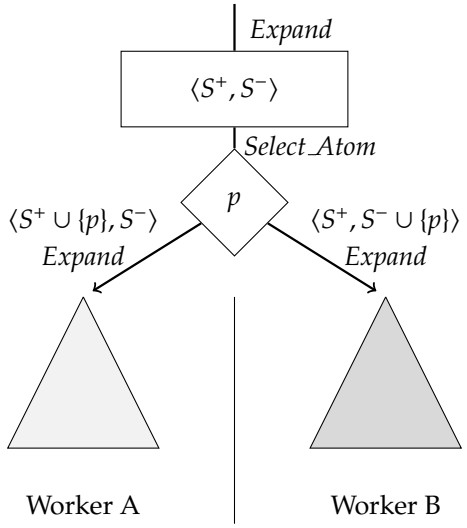

Fig. 6: Intuition of Or-Parallelism in ASP

The model proposed by Finkel et al. (2001) relies on a centralized scheduling structure, with a single central master and a collection of slave workers. The master is responsible for coordinating the distribution of unexplored parts of the search tree among workers. Paths in the search tree are described as binary strings (i.e., a 0 represent a left child while a 1 represents a right child). Initially, each worker receives the complete ASP program and a binary string which is used to deterministically choose its position in the search tree (Figure 7). A similar model has been adapted for execution on distributed memory architectures in the claspar system (Ellguth et al. 2009; Schneidenbach et al. 2009). The claspar system adds the ability of organizing workers in a deeper hierarchical structure and the ability to exchange learned nogoods across parallel computations.

The binary search tree created by the execution of Smodels is irregular—thus leading to an unbalanced distribution of work among workers. When a worker has exhausted its assigned search tree, it requests a new branch to the master; in turn, the master requests work from a randomly chosen worker. The selected worker will transfer the highest (i.e., closest to the root) choice point with open alternatives to the master for redistribution, marking it as fully explored. This approach avoids the risk of different workers exploring the same branch of the search tree. Selecting unexplored choices closer to the root is a known heuristic aimed at increasing the chance of assigning to a worker a potentially large task.

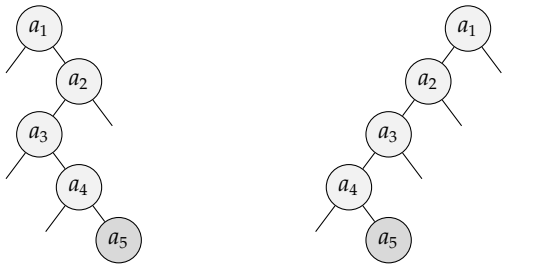

Fig. 7: Examples of initial distribution in ParStab: 0 left child, 1 right child

The system initially proposed by El-Khatib and Pontelli (2000) and Pontelli (2001), and later fully developed by Balduccini et al. (2005) and Pontelli et al. (2010), represents a fully Or-parallel implementation of Smodels with symmetric workers—without the presence of a master serving as broker for distribution of unexplored tasks. Each worker explores parts of the search tree as well as participates in the distribution of unexplored parts of the search tree to other, idle, workers. Thus, each worker alternates **(1)** *computation* steps (corresponding to the execution of Smodels), and **(2)** *load balancing* steps to relocate workers to branches of the search tree with unexplored alternatives. The design has been developed on both shared memory systems (Pontelli 2001; Balduccini et al. 2005) as well as on Beowulf clusters (Pontelli et al. 2010). The two implementations have a similar design, where branches of the search tree are represented locally within the data structures of each worker (i.e., a complete Smodels solver).

5.2.2 Scheduling and Heuristics

The process of load-balancing is essential to allow workers to remain busy and increase the degree of parallelism exploited. Load balancing is composed of two activities: **(1)** *scheduling*, which is used to identify the location in the search tree where an idle worker should be moved, and **(2)** *task sharing*, which is the actual process of moving the worker to the new location.

Several dimensions have been explored for both scheduling and task sharing, and experimental comparisons have been presented by Le et al. (2005; 2007; 2010). While significant performance differences can be observed, thus making it hard to identify a clear winning strategy, on average the most effective methodology for scheduling and

task sharing is the *Recomputation with Reset* strategy (Figure 8). Intuitively, scheduling is based on selecting the highest node in the tree with unexplored alternatives. The process of task sharing is realized by restoring the state of the worker to the root of the tree (Figure 8 top left) and repeating the computation from the root to the selected node of the tree (Figure 8 top right). The latter operation can be performed efficiently using a single *Expand* operation. The idle worker is then able to restart the computation with an unexplored alternative from the selected node (Figure 8 bottom). The idea of recomputation in Or-parallelism is not novel—it has been explored in the context of Or-parallelism in Prolog by several systems, such as the Delphi model (Clocksin and Alshawi 1988) and the Randomized Parallel Backtracking model (Janakiram et al. 1988; Lin 1989).

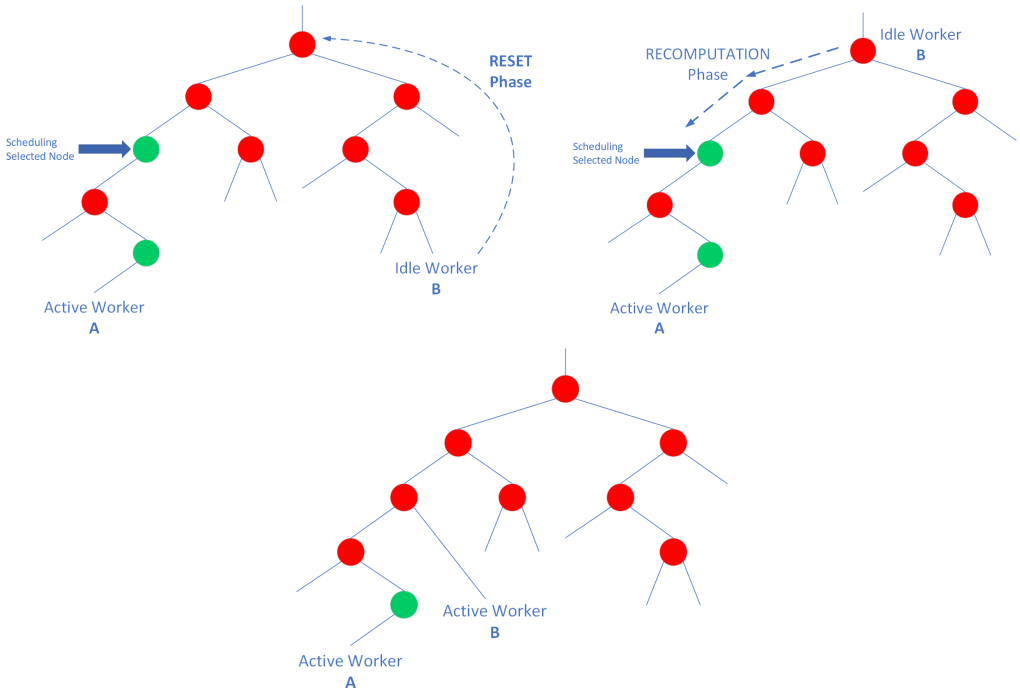

Fig. 8: Recomputation with Reset

5.3 Parallel Grounding

In this section we focus on the parallelization of the grounding stage that transforms the first order logic program P into an equivalent propositional program $ground(P)$. P uses a finite set of constant symbols C. A rule using n different variables should, in principle, be replaced by $|C|^n$ ground clauses where variables are replaced by elements of C in all possible ways. This simple idea can be easily parallelized; however it might lead to a ground program of unacceptable size. Since the first grounder Lparse (Simons et al. 2002) this problem has been addressed by splitting program-defined predicates into two classes: domain and non-domain predicates. The precise definition of domain

predicates has been changed in the evolution of grounders, but the idea is that they are those predicates that can be extensionally computed deterministically using a bottom-up procedure. In particular, all predicates extensionally defined by ground facts are domain predicates. Every variable occurring in a rule should occur in the argument of a positive atom in the body as well to help grounding (domain/range restriction). This introduces a partial ordering between parts of the program. This ordering has been exploited by parallel grounders, as well. Precisely, parallelism for grounding can be implemented at three different levels:

1. *Components Level* parallelism is based on the analysis of the strongly connected components (SCC) of the dependency graph $\mathcal{G}(P)$. The program is split in modules and the grounding follows the topological ordering of the SCC. Independent modules can be managed in parallel and synchronization points are used to control the overall process.
2. *Rules Level* parallelism. Each rule can be, in principle, grounded in parallel. Non-recursive rules are grounded first. Grounding of rules involved in recursion are delayed. Their grounding follows the bottom-up fixpoint procedure (precisely, the semi-naive evaluation procedure developed for Datalog) that ends when no new ground clauses are generated.
3. *Single Rule Level* parallelism takes care of parallelizing into different threads the grounding of a single rule. Assume a rule contains n different variables $X_1, \ldots, X_n$. Each variable X_i, $i \in \{1, \ldots, n\}$, occurs in an atom based on a domain predicate and we know that X_i ranges in the set of constant symbols $C_i \subseteq C$. The grounding of the rule produces $\Pi_{i=1}^{n} |C_i|$ ground instances. It is expected that $|C_i| \ll |C|$ but, in any case, the number of instances grows exponentially with n. Thus, implementing this form of parallelism is crucial for rules containing several variables.

The research in parallel grounding can be traced back to the work of Balduccini et al. (2005), where authors exploited the property of range restrictedness of Lparse programs and implemented single rule parallelization following a static partition of rules. Let us observe that each parallel processor is assumed to be aware of the domain predicates used in the rule. The group of University of Calabria has deeply investigated the multilevel parallelism hierarchy described above (see, for instance, the work by Calimeri et al. (2008) and by Perri et al. (2013)) with outstanding performance improvements. An interesting observation of these works is the evidence that, in the majority of the explored benchmarks, single rule level parallelism represents the dominating component for performance improvement.

5.4 Other Forms of Parallelism

Alternative forms of parallelism have also been explored in the context of ASP.

Lookahead parallelism has been considered as a technique to improve performance of the deterministic steps of the ASP computation. Lookahead is an optimization technique introduced in the Smodels system. Lookahead is part of the *Select_Atom* operation. Before selecting a chosen atom, the lookahead operation performs a quick set of *Expand* operations (which are typically very efficient) on undefined atoms: given a partial answer set $\langle S^+, S^- \rangle$ and a set of atoms A such that $A \cap (S^+ \cup S^-) = \emptyset$, for each $x \in A$ the

system executes both $Expand(P, \langle S^+ \cup \{x\}, S^- \rangle)$ and $Expand(P, \langle S^+, S^- \cup \{x\} \rangle)$. If both of them leads to failure, then backtracking should be initiated; if only one of them leads to failure, then x can be added to the partial answer set and the process continue; if both operations succeed without failure, then the element can be considered as an option for the non-deterministic choice. The intuition of parallel lookahead is to perform such tests concurrently to eliminate unsuitable options for choice. Balduccini et al. (2005) demonstrate speedups in the range from 5 to 24 using 50 processors on a variety of benchmarks.

5.5 Portfolio Parallelism

ASP solvers, constraint solvers (used in CLP), and SAT solvers (used as auxiliary solvers in logic programming languages such as Picat (Zhou and Kjellerstrand 2016)) are composed of many distinguished parts, but they share a common scheme: **(1)** choice of a not yet instantiated variable/literal, **(2)** choice of the value (e.g., true/false for ASP and SAT solvers, a domain element in many constraint solvers), **(3)** deterministic propagation of the value chosen that allows us to reduce the remaining part of the search space. When the assignments made so far cause a *conflict* (e.g., a rule/clause/constraint unsatisfied), a **(4)** backtracking/backjumping activity starts, possibly after a (deterministic) analysis of the conflict that might lead to **(5)** learning new clauses/constraints. These clauses are implied by the problem and therefore from a logic point of view they are redundant, but their explicit addition can speed up the remaining part of the search, thanks to the new inference power they support. Moreover, the search can be sometimes *restarted* **(6)** from the beginning but new parts of the search tree are visited thanks to the new constraints.

This general scheme supports many variants, especially for the choices **(1)**, **(2)**, and **(6)**. Thus, one could use different solvers (a *portfolio of algorithms* (Gomes and Selman 2001)) or a solver with many parameters that can, in principle, be tuned to the particular instance of the problem to provide the best possible performance (*algorithm configuration*). A solver has a set of parameters with values in discrete or continuous domains. Even if continuous domains are discretized, the number of possible tuples of parameter values would make a manual optimization infeasible. The algorithm configuration approach applies statistical techniques to automatically find an "optimal" configuration for a family of problem instances that follow a certain distribution. This is usually implemented by iterating a local search routine starting from an initial solution and verifying it in a set of training instances.

An easy way of exploiting parallelism in this context could be simply that of using a set of (fixed) algorithms and run them in parallel, and taking the first solution generated by one of the parallel threads. In the case of algorithm configuration, local search techniques can be run in parallel (with different random choices) and the best solution in a finite amount of time is retrieved. Thus, algorithm portfolio and configuration activities will benefit from a parallel architecture.

The research in the area has been rather active in the last years showing excellent performances. The heuristics that drive to the choices **(1)** and **(2)** can be static (computed only at the beginning of the search) or dynamic (updated during the search) and based on the analysis of some features. Typical features are of the following types (as described by Maratea et al. (2014)):

Problem size features number of rules r, number of atoms a, ratios r/a and ratios reciprocal a/r of the ground program.

Balance features ratio of positive and negative atoms in each rule body, ratio of positive and negative occurrences of each variable, fraction of unary, binary, and ternary rules, etc

Proximity to Horn features fraction of Horn rules and number of atoms occurrences in Horn rules.

ASP peculiar features number of true and disjunctive facts, fraction of normal rules and constraints, head sizes, occurrences of each atom in heads, bodies and rules, occurrences of true negated atoms in heads, bodies and rules, sizes of the SCCs of $\mathcal{G}(P)$, number of Head-Cycle Free (HCF) and non-HCF, etc

Moreover, dynamic features can consider the ratio of the variables currently assigned by propagation vs those assigned non-deterministically, the number of restarts, the number of rules learned since the last restart, and so on. Modern solvers offer a number of built-in sets of parameter configurations. For instance, the heuristics `frumpy` of the ASP solver clasp (Gebser et al. 2011) sets the following parameters

```
--eq=5 --heuristic=Berkmin --restarts=x,100,1.5
--deletion=basic,75 --del-init=3.0,200,40000
--del-max=400000 --contraction=250
--loops=common  --save-p=180 --del-grow=1.1
--strengthen=local --sign-def-disj=pos
```

In particular, it uses the variable selection rule developed for the SAT solver `Berkmin` (Goldberg and Novikov 2007). Furthermore, clasp allows the user to choose the `auto` option that select the most promising configuration based on the features of the current instance or to input a precise configuration from a file.

The selection of the algorithm is implemented using supervised and unsupervised machine learning techniques. In the case of algorithm configuration, instead, parameters are often tuned using local search technique. We give a brief summary here of the approaches in the area of SAT, ASP, and Constraint Programming (a complete and up-to-date survey can be found in the paper by Tarzariol (2019)).

SATzilla (Xu et al. 2008) is the first algorithm selection implementation in the area of SAT solving. Several versions follow its first prototype adding new statistical techniques and obtaining much more performances. CLASPfolio (Gebser et al. 2011) executes algorithm selection among a set of twelve configurations of the clasp solver with "complementary strengths" using support vector regression techniques. A static approach is made by ME–ASP (Maratea et al. 2014) analyzing a set of parameters as explained above.

A common experience in problem-solving is that (often) a solver either solves a problem in few seconds or does not solve it in days. Then, the idea is to use the algorithm portfolio with a limited time and analyze the output for selecting the best algorithm and repeat this several times during the search. This technique is called *algorithm scheduling*. In this case, it is crucial that the portfolio selection is made in parallel. This technique has been implemented and applied with success in Constraint Programming by SUNNY —*SUbset of portfolio solvers by using k-Nearest Neighbor technique to define a lazY learning*

model—(Amadini et al. 2014), in ASP by aspeed (Hoos et al. 2015) that is based on the clasp solver, and in SAT by Malitsky et al. (2012). All the three proposals exploit multi-core architectures and parallelism.

Portfolio parallelism has been considered in the second revision of claspar (Schneidenbach et al. 2009). Portfolio parallelism is realized by instructing a pool of workers to attempt to solve the same problem but with different configurations of the solver—e.g., different heuristics and different parameters to guide search. This allows the workers to "compete" in the resolution of a problem by creating different organizations of the search space and exploring them in parallel. Schneidenbach et al. (2009) provided speedups of the order of 2 using this technique.

Some systems combine algorithm portfolio and configuration. CLASPfolio 2 (Hoos et al. 2014) improves the selection technique of CLASPfolio by defining a static pre-solving scheduling that may intervene if the learned selection model performs poorly. AutoFolio (Lindauer et al. 2015) executes Algorithm Configuration over CLASPfolio 2, choosing the optimal learning techniques with its optimal parameters configuration.

As a final observation, the use of portfolio techniques may have an impact on the "observable semantics" of a system in some situations. In the context of solving optimization problems or in determining all models of a program, portfolio techniques will not modify the behavior of the system; on the other hand, if the system is used to determine one answer set, then portfolio techniques may lead to a different answer set than the one found by a sequential system.

6 Going Large: Logic Programming and Big Data Frameworks

The scalability of logic programming technologies has been a constant focus of attention for the research community. Scalability in terms of speed has been well understood and materialized in a number of highly efficient systems. Scalability in terms of memory consumption, instead, is still an open challenge. There are several examples offered in the literature that capture this challenge. For example,

- In the domain of planning using ASP, scalability in terms of solving large planning problem instances is negatively impacted by the large grounding produced by the combination of the number of actions and plan length. For example, in the encoding of the popular Biochemical Pathway planning benchmarks, from the International Planning Competitions, ASP can ground only instances with less than 70 actions (instances 1–4), running out of memory with Instance 5 (which contains 163 actions) (Son and Pontelli 2007).
- The use of logic programming techniques for processing knowledge bases (e.g., RDF stores) faces the need for smart preprocessing techniques or interfaces with external databases in order to cope with the sheer size of large repositories—e.g., as in the case of the CDAOStore, an ASP-based system for phylogenetic inference over a repository with 5 Billion triples, stored in 957 GB (Chisham et al. 2011; Pontelli et al. 2012).

These challenges have prompted a number of research directions, ranging from the use of interfaces between logic programming systems and external repositories (e.g., to avoid the need for fully in-memory reasoning), as in the DLV-HEX system (Eiter et al.

2018), to systems working with lazy-grounding or non-ground computations (Bonatti et al. 2008; Dal Palù et al. 2009; Leutgeb and Weinzierl 2017). An alternative approach relies on the use of distributed programming techniques to address scalability.

6.1 Introduction to Large Scale Data Paradigms

The literature has offered access to popular infrastructures and paradigms to facilitate the development of applications on distributed platforms, taking advantage of both the parallelism and the ability to distribute data over the memory hierarchies of multiple machines. In this section, we will briefly review some of the fundamentals of such distributed programming infrastructures (see, e.g., Hadoop and Spark documentation (White 2015; Karau et al. 2015) for further insights).

Distributed File Systems (DFS) are designed to provide a scalable and highly fault-tolerant file system, which can be deployed on a collection of machines using possibly low-cost hardware. The *Hadoop Distributed File System (HDFS)* represents a popular implementation of a DFS (Apache 2020a). It adopts a master-slave architecture. *NameNodes* (masters) regulate access to files, track data files, and store metadata. File content is partitioned into small *chunks* (blocks) of data distributed across the network. Chunks may range in size (typically, from 16 *MB* to 64 *MB*) and replicated to provide redundancy and seamless recovery in case of hardware failures. An HDFS instance may consist of thousands of server machines and provides hardware failure detection, a write-once-read-many access model, and streaming data access.

Map-Reduce is a distributed programming paradigm designed to analyze and process large data sets. The foundations of the concept of Map-Reduce can be traced back to fundamental list operations in functional programming. However, the concept gained popularity as a rigid paradigm for distributed programming, piloted by Google and popularized in implementations as in the Apache Hadoop framework (Ullman 2010; Apache 2020a). The Map-Reduce paradigm provides a basic interface consisting of two methods (see Figure 9–left):

- `map( )` that maps a function over a collection of objects. It outputs a collection of "key-value" tuples;
- `reduce( )` that takes as input a collection of key-value pairs and merges the values of all entries with the same key.

The map method is usually performed in order to transform and filter a collection, while the reduce method usually performs a summary (e.g., counting the elements of a collection, or the word frequencies in a text).

Extended implementations of Map-Reduce have been devised to allow the iterative execution (Figure 9–right) of Map-Reduce cycles optimizing communication (Bu et al. 2010; Ekanayake et al. 2010).

Graphs Primitives are made available for HDFS within the framework Apache Spark (Karau et al. 2015). Spark is an in-memory data processing engine that allows for streaming, data processing, machine learning and SQL functionalities; it relies on the concept of

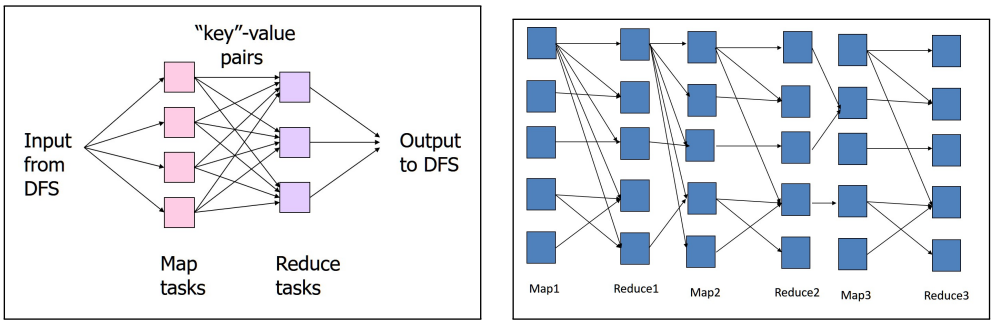

Fig. 9: Map-Reduce paradigm

Resilient Distributed Dataset (RDD). A RDD is an immutable, fault-tolerant distributed collection of objects, a read-only, partitioned collection of records, organized into logical partitions, that may be located and processed on different nodes of the HDFS. Among the libraries built on top of Spark Core, *GraphX* (Apache 2020b) has been developed for graphs modeling and graph-parallel computation. GraphX takes full advantage of the RDD data structure and extends it providing a distributed property multigraph abstraction. Property graphs are directed multigraphs with user-defined objects associated to vertices and edges. They are encoded by pairs of RDDs containing the properties for vertices and edges, and, therefore, inherit RDDs features, such as map, filter, and reduce.

GraphX also gives access to a complete interface for dealing with graphs, as well to an implementation of the Pregel API (Malewicz et al. 2010). Pregel is a programming model for large-scale graph problems and for fix-point computations over graphs. A typical Pregel computation consists of a sequence of *super-steps*. Within each super-step, vertices may interact with their neighbors by sending messages. Each vertex analyzes the set of messages received in the previous super-step (if any) and alters its content according to a user-defined function. In turn, each node can generate new messages to the neighboring nodes. A Pregel computation stops when a super-step does not generate any messages or when other halting conditions are encountered (e.g., a maximum number of iterations).

6.2 Large Scale Computing in Datalog

Computing Natural Joins Using Map-Reduce. The underlying core component of most implementations of logic programming using Map-Reduce is the ability to scale the computation of natural join operations over large datasets, as originally studied by Afrati et al. (2010; 2011).

Let us start considering the basic case of a 2-way join between two relations R and S, denoted by $R \bowtie S$. Let us assume, for the sake of simplicity, that R and S are binary and that the second attribute name of the former is also the first attribute name of the latter (briefly, it can be denoted as $R(A, B) \bowtie S(B, C)$). A Map-Reduce computation can be achieved as follows:

- Each Map task receives tuples drawn from the two relations R and S and produces key-values pairs where the *key* is the value of the common argument B and the values are the remaining component of the tuple. Namely, $(a, b) \in R$ is mapped to

the pair $\langle b, (a, R)\rangle$, and $(b, c) \in S$ is mapped to $\langle b, (c, S)\rangle$ (R and S are here the names of the two relations).
- Each Reduce task receives a pair with *one value* of the common argument and a list of all tuples containing such value, e.g., $\langle b, L\rangle$, with $L = [(a, R), (a', R), (c, S), (c', S), \dots]$ and outputs a resulting (ternary) relation RS with tuples $(x, b, y) \in RS$ for each $(x, R), (y, S)$ in the list L.

Thus, the Reduce task exploits data parallelism, using one worker for every singe element of the common domain.

This can be extended to the case of multi-way joins, e.g., $R(A, B) \bowtie S(B, C) \bowtie T(C, D)$, without the need of cascading two-way joins (which could lead to very large intermediate relations). The model explored relies on the use of hashing. Let us adopt a hash function h for the attribute B, partitioned in β buckets, and a hash function g for the common attribute C, partitioned in γ buckets.

- Each Map task performs the following hash operations: **(1)** each tuple $(x, y) \in R$ is mapped to $\langle (h(y), c), (R, x, y)\rangle$ for each $1 \le c \le \gamma$; **(2)** each tuple $(y, z) \in S$ is mapped to $\langle (h(y), g(z)), (S, y, z)\rangle$; **(3)** each tuple $(z, w) \in T$ is mapped to $\langle (b, g(z)), (T, z, w)\rangle$ for each $1 \le b \le \beta$.
- For each pair $(b, c) \in \{1, \dots, \beta\} \times \{1, \dots, \gamma\}$ there is a reduce task taking care of pairs of the form $\langle (b, c), L\rangle$. It outputs the relation RST with tuples (x, y, z, w) for every triple of triples $(R, x, y), (S, y, z), (T, z, w)$ in L.

In this case the distribution of the work is split in $\beta\gamma$ workers, and β and γ can be defined arbitrarily, thus determining the amount of parallelism we would like to exploit.

Distributed Computation in Datalog. Distributing the computing of the join is the basis for the general computation of bottom-up semantics. In particular, the WebPIE system (Urbani et al. 2012) has been designed to support reasoning over RDF stores. The proposed system is capable of determining inferences according to RDFS semantics and the OWL *ter Horst* fragment (ter Horst 2005). The approach adopted consists of encoding the inference rules capturing these semantics as Datalog rules. The computation of the least fixpoint of the set of rules given a collection of RDF triples is achieved as an iterated Map-Reduce computation, which captures the bottom-up application of the inference rules. The WebPIE system takes advantage of the specific format of the resulting Datalog rules. In the case of RDFS, the system takes advantage of the following factors:

- Each rule has at most two subgoals in the body—thus allowing the use of the Map-Reduce method to compute 2-way natural joins to capture the bottom-up application of each rule;
- In most rules, one of the two subgoals is one of the triples from the RDF store—allowing to optimize the computation by keeping the original triples in memory and matching them with generated ones;
- In most cases, it is possible to order the application of the rules to perform the computation using only three phases of Map-Reduce.

The generalization to arbitrary Datalog programs has been investigated in a variety of works (e.g., those by Afrati et al. (2011) and by Afrati and Ullman (2012)). While

different approaches have provided a variety of optimizations, based on special types of rules (e.g., transitive closures) and size of the definitions of different predicates in the extensional database, they all build on the principles of iterated execution of Map-Reduce workflows, using approaches like HaLoop (Bu et al. 2010).

This work direction has been expanded to consider Datalog with a stratified use of negation, as a natural iteration of the approaches described earlier (Tachmazidis and Antoniou 2013). Following the lexicographical sort of the $\mathcal{G}(P)$, the computation can proceed by iterating the standard Datalog computation (implementing the T_P using joins and exploiting Map-Reduce). The extra ingredient here is handling of negated literals where *anti-join* needs to be implemented.

For example, given a rule of the type

$$h(A, B) \texttt{:-} p(A, C), q(C, B), \texttt{not } r(B).$$

the inference is realized using two Map-Reduce processes:

1. The first implements a natural join to derive the consequences of $p(A, C), q(C, B)$, as discussed earlier (e.g., producing a temporary relation $pq(A, C, B)$);
2. The second Map-Reduce phase performs an *anti-join*—with analogous structure as the natural join, with the exception that the reduce step is used to filter out those tuples with matching values in the pq relation and the r relation.

Strong performance results have been presented, thanks also to a broad range of optimizations based on special cases present in the program clauses, such as lack of common arguments in the body of a rule (Tachmazidis and Antoniou 2013).

The approach can be extended to support the computation of the well-founded semantics of logic programs with negation (c.f. Section 2.1), as demonstrated by Tachmazidis et al. (2014).

Recent approaches for increasing the speedup of Datalog distributed computation introduce new data structures (Jordan et al. 2019) and exploit network topology (Blanas et al. 2020) for increasing the speedup of Datalog distributed computation.

A similar methodology was also used to support inferences in stratified Datalog with defeasible reasoning (Tachmazidis et al. 2012). In simplified terms, a theory is composed of a Datalog program with two types of rules:

$$\underset{\textit{non-defeasible}}{\textit{head} \leftarrow \textit{body}} \qquad \underset{\textit{defeasible}}{\textit{head} \Leftarrow \textit{body}}$$

along with an ordering among defeasible rules. The work by Tachmazidis et al. (2012) explores the use of Map-Reduce in the case of stratified defeasible theories—i.e., theories where the dependency graph is acyclic—and thus organized in strata (e.g., the top stratum includes predicates with no outgoing edges, the preceding one contains the predicates with links to the top stratum, etc.). The computation requires two Map-Reduce phases for each stratum, starting with the lowest one. The first Map-Reduce is used to identify applicable rules, using multi-way joins as discussed earlier, recording for each derivable element the producing rule and the defeasible nature of the derivation. The second phase, which is primarily composed of a Reduce step, compares derived rules based on the ordering of defeasible rules to identify undefeated conclusions.

6.3 Large Scale Computing and ASP

The use of Map-Reduce and related paradigms to support the execution of ASP has been only recently considered and with only very preliminary results. The extension of the previously mentioned approaches to the case of ASP is not trivial, due to the switch from a bottom-up computation leading to a single minimal model, as in Datalog, to the case of a non-deterministic computation leading to possibly multiple models.

The foundation of the methodology considered for the distributed computation of ASP lies in the ability of modeling the computation of the semantics of logic programming in terms of operations of graphs (Konczak et al. 2006), and taking advantage of existing models for large scale distributed computations over graphs. This idea was piloted by Maiterth (2012).

Basically, from a ground logic program P, a graph akin to the rule dependency graph $\mathcal{G}(P)$ is computed. The idea is of looking for a coloring of the nodes with two colors; colors encode the fact that the body of a rule is *activated* (by the tentative model) and therefore its head should be in the model, as well, or not. A correspondence between these colorings and answer sets has been established and therefore the computation is delegated to graph operations that can be parallelized exploiting the library GraphX of Apache.

These concepts have been used to develop a preliminary ASP solver, using the Pregel support of Apache Spark (Igne et al. 2018; De Bortoli et al. 2019). The approach allows dealing with programs with huge grounding that could not be stored in a single memory. However, answer set computation based on coloring can not exploit the many heuristics and the conflict driven clause learning techniques commonly implemented in ASP solvers and the running time is not comparable. Map-Reduce has been experimented with a general parallel approach for distributed computation of the fixpoint and of the well-founded semantics using always an approach based on distributed graph computation (De Bortoli et al. 2019). However, the algorithms proposed are general, architecture-independent and do not exploit any possible optimization. Thus, even if running time scales with the number of processors the performances are not comparable to those of traditional methods in a single processor.

7 Going Small: Logic Programming and GPUs

Graphical Processing Units (GPUs) are massively parallel devices, originally developed to efficiently implement graphics pipelines for the rendering of 2D and 3D scenes. The use of GPUs has become pervasive in general-purpose applications that are not directly related to computer graphics, but demand massive computational power to process large amounts of data, such as molecular dynamics, data mining, genome sequencing, computational finance, etc. Vendors such as AMD and NVIDIA provide dedicated APIs and promote frameworks such as *OpenCL* (Khronos Group Inc 2015) and *CUDA (Computing Unified Device Architecture)* (NVIDIA Corporation 2021) to support GPU-computing. In this section we focus on the efforts made to exploit this type of parallelism in logic programming.

7.1 GPU-Based Parallelism

GPUs are designed to execute a very large number of concurrent threads on multiple data. The underlying conceptual parallel model is defined as *Single-Instruction Multiple-Thread (SIMT)*. In the CUDA framework, threads are organized and executed in groups of 32 threads called *warps*. Cores are grouped in a collection of *Streaming MultiProcessors (SMs)* of the same size and warps are scheduled and executed on the SMs. Threads in the same warp are expected (but not forced) to follow the same program address. Whenever two (or more) groups of threads belonging to the same warp fetch/execute different instructions, *thread divergence* occurs. In this case the execution of the different groups is serialized and the overall performance decreases.

From the programmer's perspective, threads are logically grouped in 3D blocks and blocks are organized in 3D grids. Each thread in a grid executes an instance of the same *kernel* (namely, a C/C++ or Fortran procedure). A typical CUDA program includes parts meant for execution on the CPU (the *host*) and parts meant for parallel execution on the GPU (the *device*). The host program contains instructions for device data initialization, grids/blocks/threads configuration, kernel launch, and retrieval of results. GPUs also exhibit a hierarchical memory organization. The threads in the same block share data using high-throughput on-chip shared memory organized in *bank*s of equal dimension. Threads of different blocks can only share data through the off-chip global memory.

To take full advantage of GPU architecture, one has to:

- proficiently distribute the workload among the cores to maximize GPU *occupancy* (exploit all available device resources, such as SMs, registers, shared memory,...) and minimize *thread divergence*;
- achieve the highest possible throughput in memory accesses—namely, **(1)** adopt *strided* access pattern to shared memory to minimize *bank conflicts* (i.e., accesses to locations in the same bank by threads of the same block. In this case accesses are serialized); **(2)** employ *coalesced* accesses to global memory (see Figure 10), as this minimizes the number of memory transactions.

These requirements make the model of parallelization used on GPUs deeply different from those employed in more "conventional" parallel architectures. Existing serial or parallel solutions need substantial re-engineering to become profitably applicable in the context of GPUs.

The design of parallel engines for logic programming taking full advantage of computational power of modern massively parallel graphic accelerators, posed a number of challenges typically found in *irregular applications*. Briefly, applications are considered irregular when the exploitation of parallelism changes while the execution proceeds. Irregularity appears both in data accesses and in control flow. It is mainly due to the intrinsic nature of the data, often represented through pointer-based data structures such as lists and graphs, and to the concurrency patterns of the related algorithms. Because of data-dependencies these applications frequently produce concurrent activity per data element, require unpredictable fine-grain communications, and exhibit peculiar load imbalances (Lumsdaine et al. 2007). Needless to say, the presence of irregularity makes it hard to maximize GPU occupancy and memory throughput, while minimizing thread divergence and bank conflicts. This makes the development of solutions to

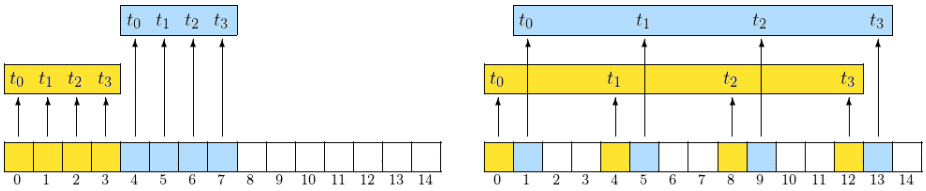

Fig. 10: Memory-access patterns on an array data-structure by a group of four threads $t_0 - t_3$. Coalesced access pattern (left) and strided access pattern (right).

irregular applications a difficult task, where high performance and scalability are not an obvious outcome. This is especially true when porting to GPUs serial algorithms or even solutions originally targeted at more traditional parallel/distributed architectures, such as cluster and multi-core systems. Parallel graph algorithms constitute significant examples, that, like SAT/ASP solving, are characterized by the need to process large, sparse, and unstructured data, and exhibit irregular and low-arithmetic intensity combined with data-dependent control flow and memory access patterns (Dal Palù et al. 2015; Formisano et al. 2017; Dovier et al. 2019; Formisano et al. 2021).

7.2 GPU-Based Datalog

Datalog engines can be obtained by exploiting (parallel) implementations of relational algebra operations *select*, *join*, and *projection*. This approach is, in principle, viable also for GPU-based parallelism or, more generally, for the case of parallel/distributed heterogeneous architectures (i.e., systems encompassing different devices, such as multi-cores, GPUs, FPGAs, etc). Several proposals appeared in literature enabling the mechanization on (multi-)GPUs systems and heterogeneous architectures, of relational operators and SQL, also including aggregate operators (Huang and Chen 2015; Wang et al. 2018; Rui and Tu 2017; Diamos et al. 2013; Saeed et al. 2015).

Concerning Datalog, Martinez-Angeles et al. (2014; 2016) design a GPU-based engine by exploiting GPU-accelerated relational algebra operators. The computation order is driven by the dependency graph of the Datalog program and fixpoint procedures are employed in case of recursive predicates. The host preprocesses the program, converting each rule into an internal numerical representation; the host decides which relational-algebra operators are needed for each rule, while their executions are delegated to the device. In particular, *select* is implemented using three different device function executions. The first one marks the rows of a matrix that satisfy the selection predicate, the second one performs a prefix sum on the marks to determine the size of the results buffer and the location where each GPU thread must write the results, and the last device function writes the results. The *projection* operator simply moves the elements of each required column to a different location. Concerning *join*, the authors adopted a standard *Indexed Nested Loop Join* algorithm. We refer the reader to the mentioned works by Martinez-Angeles et al. for the details on the CUDA implementation and a report on experimental evaluations showing significant speedups of the GPU-based engine with respect to engines running on single and multi-core CPUs. The experiments demonstrate scalability in presence of extensional databases with several million of tuples.

The Datalog-like language LogiQL is used as front-end in the Red Fox system (Wu et al. 2014). LogiQL is a variant of Datalog including aggregations, arithmetic, integrity constraints and active rules. Red Fox provides an environment enabling relational query processing on GPUs, through compilation of LogiQL queries into optimized GPU-based relational operations. Then, an optimized query plan is generated and executed on the device. The approach is demonstrated to be faster than the corresponding (commercial) CPU-oriented implementation of LogiQL.

Nguyen et al. (2018) propose a different approach to Datalog parallelization, not directly relying on parallelization of relational algebra. In this work, the computation of the least model of a definite program is obtained by first translating the program into a linear algebra problem. Then, the multi-linear problem is solved on GPU by using standard CUDA libraries for matrix computations. The solution of the multi-linear problem identifies the model of the program. A similar approach is also viable to compute stable models of disjunctive logic programs. The approach demonstrates encouraging performance results for randomly generated programs with over 20,000 rules.

7.3 GPU-Based ASP

The first attempt in exploiting GPU parallelism for ASP solving has been described by Dovier et al. (2015; 2016; 2019), proposing the solver YASMIN. The authors design a conflict-driven ASP-solver reminiscent of the conventional structure of sequential conflict-driven ASP solvers (Gebser et al. 2012). However, substantial differences lay in both the implemented algorithms and in the specific solutions adopted to optimize GPU occupancy, minimize thread divergence, and maximizing memory throughput. To this aim, difficult to parallelize and intrinsically sequential components of serial solvers have been replaced by parallel counterparts. More specifically, **(1)** the notion of *ASP computations* (Liu et al. 2010) is exploited to avoid the introduction of loop formulas and the need of performing *unfounded set checks* (Gebser et al. 2012); **(2)** a parallel conflict analysis procedure is used as an alternative to the sequential resolution-based technique used in CLASP. Memory accesses have been regularized by suitably sorting input data with respect to size of rules, by storing them using Compressed Sparse Row (CSR) format, and by designing specific device functions, each one tailored to process groups of rules of homogeneous size. All this enables efficient balanced mapping of data to threads. Moreover, to maximize device performance, the authors exploited specific features supported by CUDA framework, such as *shuffling* (a high efficient intra-warp communication mechanism) and *stream-based parallelism* (a support to concurrent kernels asynchronous execution available in the CUDA programming framework).

GPU-based parallelism can be combined with host parallelism by exploiting host POSIX *pthreads* and CUDA streams. In particular, Algorithm 3 (simplified form of the ASP-solver designed by Dovier et al. (2019)) shows the main part of the host code of the multi-pthread procedure PTHREADED_YASMIN which first splits (line 1) the given problem into a number *Npb* of subproblems by applying some heuristics/criteria. The simplest possibility would be to apply Or-parallelism and split the search space by assigning in different ways the truth values of a subset of the input atoms. Then, the procedure (lines 2–3) spawns a pool of *N* host POSIX threads. Note that these pthreads share host

Algorithm 3: Host code of the main procedure PTHREADED_YASMIN of the pthreaded ASP-solver YASMIN (simplified)

```
   Input: Ground ASP program P
   Input: Number of subproblems Npb and number of pthreads N
   Output: Stable models
1  Parts ← partition_problem(P, Npb, Opts)
   /* spawns N concurrent instances of yasmin_launcher():          */
2  for (pth ← 0; pth < N; pth++) do
3      pths[pth] ← pthread_create(yasmin_launcher(pth))
4  for (pth ← 0; pth < N; pth++) do
5      pthread_join(pths[pth])          /* wait for pthread's completion */
6  cudaDeviceSynchronize () /* wait for termination of all device code */
7  outputStableModels ()
```

Algorithm 4: Host code of YASMIN_LAUNCHER, called in Algorithm 3 (simplified)

```
   Input: Pthread ID pth
   Output: Stable models (stored in global device memory)
1  while exists S ∈ Parts do       /* repeat while there are subproblems */
2      Parts ← Parts \ {S}
3      yasmin(S, pth)
4      cudaStreamSynchronize ()    /* wait for operations in the stream */
5  pthread_exit()          /* pthread ends and joins the main pthread */
```

variables (such as *P*, *Parts*,...), but each pthread issues device commands in a different CUDA stream. Each pthread created in line 3 runs an instance of the host procedure `yasmin_launcher` (described in Algorithm 4). After the termination of all issued device commands (lines 4-6) results are collected and output (line 7).

Algorithm 4 describes the procedure `yasmin_launcher` executed by each concurrent pthread. Such procedure iterates by extracting (in mutual exclusion) one of the unsolved subproblems (line 2) and by running an instance of the CUDA solver in a dedicated CUDA stream (line 3). Notice that, since each pthread runs an instance of the solver in a private CUDA stream, each of them proceeds by issuing commands (memory transfers, kernel launches, etc) in such stream, independently and concurrently with the other solver instances. This helps in maximizing GPU occupancy, because it permits overlapping between computation and memory transfers and allows scheduling of warps on all available SMs.

The combination of host-parallelism and device-parallelism opens up further refinements, such as the introduction of techniques like *parallel lookahead* (Dovier et al. 2018), the development of more powerful *multiple learning* schemes (Formisano and Vella 2014), and paves the way to the exploitation of multi-GPU and heterogeneous architectures in ASP-solving.

8 Conclusion

We have presented a review of the "second twenty years" of research in parallelism and logic programming. The choice of the period is motivated by the availability of a comprehensive survey of the first twenty years, published in 2001, that has served as a fundamental reference to researchers and developers since. While the contents of this classic survey are quite valid today, we have attempted to gather herein the later evolution in the field, which has continued at a fast pace, driven by the high speed of technological evolution, that has led to innovations including very large clusters, the wide diffusion of multi-core processors, the game-changing role of general-purpose graphic processing units, or the ubiquitous adoption of cloud computing. In particular, after a quick review of the major milestones of the first 20 years, we have reviewed the recent progress in parallel execution of Prolog, including Or-parallelism, And-parallelism, static analysis, and the combination with tabling. We have covered the significant amount of work done in the context of parallelism and Answer Set Programming and Datalog, including search parallelism, other forms of parallelism, parallel grounding, or portfolio parallelism. Finally, we have addressed the connections with big data frameworks and graphical processing units.

This new survey highlights once more the wide diversity of techniques explored by the logic programming community to promote the efficient exploitation of parallelism within the various variants of the paradigm that have been emerging. Over these years, we have seen the emergence of more declarative styles of logic programming, such as Answer Set Programming, as well as an evolution in this same direction within the other languages that follow the Prolog-style line. Nevertheless, the experiences reported in the survey show that many of the techniques developed in the early days of parallel logic programming are still applicable and have paved the way to the efficient parallelization of these more modern logic programming variants. These lessons have also not been limited to the domain of logic programming but have also benefited exploitation of parallelism in other domains. This includes, e.g., the static analysis examples in Section 4.3 or the conceptual models used in or-parallelism, which have supported work on parallel planning (e.g., (Tu et al. 2009)). We expect the general principles of parallel logic programming to remain valid and benefit broader efforts to parallelism in even more domains.

Some of the works summarized in this survey are in their infancy, but they are expected to become dominant trends in the years to come. Just as the use of GPUs has provided the backbone to success of paradigms like machine learning, we expect GPUs to gain an even more prominent role in parallel logic programming—e.g., supporting more complex heuristics and novel extensions (such as the integration of Answer Set Programming with constraint satisfaction and optimization). The role of multi-platforms is going to become prominent, especially with the growing emphasis on edge-to-cloud computing.

We hope to have put together a worthy continuation of the classic survey, covering these last twenty years, and hope that it will serve not only as a reference for researchers and developers of logic programming systems, but also as engaging reading for anyone interested in logic and as a useful source for researchers in parallel systems outside logic programming. The interested reader will find details of the performance results

obtained using a diversity of coding techniques, architectures and benchmarks, in the original contributions cited in this paper.

References

Afrati, F. N., Borkar, V. R., Carey, M. J., Polyzotis, N., and Ullman, J. D. 2011. Map-Reduce extensions and recursive queries. In *14th International Conference on Extending Database Technology*. ACM, New York, 1–8.

Afrati, F. N. and Ullman, J. D. 2010. Optimizing joins in a Map-Reduce environment. In *13th International Conference on Extending Database Technology*. ACM, New York, 99–110.

Afrati, F. N. and Ullman, J. D. 2012. Transitive closure and recursive Datalog implemented on clusters. In *15th International Conference on Extending Database Technology*. ACM, New York, 132–143.

Albert, E., Arenas, P., Genaim, S., Puebla, G., and Zanardini, D. 2007. Cost analysis of Java bytecode. In *16th European Symposium on Programming, ESOP'07*, R. D. Nicola, Ed. Lecture Notes in Computer Science, vol. 4421. Springer, Heidelberg, Germany, 157–172.

Ali, K. A. M. and Karlsson, R. 1990a. The Muse approach to Or-parallel Prolog. *International Journal of Parallel Programming 19,* 2, 129–162.

Ali, K. A. M. and Karlsson, R. 1990b. The Muse Or-parallel Prolog model and its performance. In *1990 N. American Conf. on Logic Prog.* MIT Press, Cambridge, MA, USA, 757–776.

Ali, K. A. M. and Karlsson, R. 1992. Scheduling speculative work in Muse and performance results. *International Journal of Parallel Programming 21,* 6, 449–476.

Amadini, R., Gabbrielli, M., and Mauro, J. 2014. SUNNY: a lazy portfolio approach for constraint solving. *Theory and Practice of Logic Programming 14,* 4-5, 509–524.

Amdahl, G. M. 1967. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the AFIPS '67 Spring Joint Computer Conference*. AFIPS Conference Proceedings, vol. 30. AFIPS / ACM / Thomson Book Company, Washington DC, 483–485.

Apache. 2020a. The Apache Software Foundation: Apache Hadoop. Tech. rep., `https://hadoop.apache.org/`.

Apache. 2020b. The Apache Software Foundation: GraphX programming guide. Tech. rep., `http://spark.apache.org/docs/latest/graphx-programming-guide.html`.

Applied Logic Systems, Inc. 2021. ALS Prolog. Tech. rep., `https://alsprolog.com/`.

Areias, M. and Rocha, R. 2012. Towards multi-threaded local tabling using a common table space. *Theory and Practice of Logic Programming, International Conference on Logic Programming, Special Issue 12,* 4 & 5, 427–443.

Areias, M. and Rocha, R. 2015. Batched evaluation of full-sharing multithreaded tabling. In *Post-Proceedings of the 4th Symposium on Languages, Applications and Technologies*. Number 563 in CCIS. Springer, Heidelberg, Germany, 113–124.

Areias, M. and Rocha, R. 2017. On scaling dynamic programming problems with a multithreaded tabling system. *Journal of Systems and Software 125*, 417–426.

Balduccini, M., Pontelli, E., El-Khatib, O., and Le, H. 2005. Issues in parallel execution of non-monotonic reasoning systems. *Parallel Computing 31,* 6, 608–647.

Baral, C. 2003. *Knowledge representation, reasoning and declarative problem solving*. Cambridge University Press, UK.

Barklund, J. 1990. Parallel unification. Ph.D. thesis, Uppsala University. Uppsala Theses in Computing Science 9.

Beaumont, A. J. and Warren, D. H. D. 1993. Scheduling speculative work in Or-parallel Prolog systems. In *Proceedings of the International Conference on Logic Programming*, D. S. Warren, Ed. MIT Press, Cambridge, MA, USA, 135–149.

Blanas, S., Koutris, P., and Sidiropoulos, A. 2020. Topology-aware parallel data processing: Models, algorithms and systems at scale. In *CIDR 2020, 10th Conference on Innovative Data Systems Research*. www.cidrdb.org, 1–8.

Bonatti, P., Pontelli, E., and Son, T. C. 2008. Credulous resolution for answer set programming. In *National Conference on Artificial Intelligence (AAAI)*. AAAI Press, 418–423.

Bone, P. 2011. Automatic parallelism in Mercury. In *Technical Communications of the 27th International Conference on Logic Programming*. Vol. 11. LIPICS, 251–254.

Bone, P. 2012. Automatic parallelization for Mercury. Ph.D. thesis, University of Melbourne.

Bone, P., Somogyi, Z., and Schachte, P. 2012. Controlling loops in parallel Mercury code. In *Proceedings of the POPL 2012 Workshop on Declarative Aspects of Multicore Programming*. ACM, New York, 11–20.

Brass, S., Dix, J., Freitag, B., and Zukowski, U. 2001. Transformation-based bottom-up computation of the well-founded model. *Theory and Practice of Logic Programming 1,* 5, 497–538.

Bu, Y., Howe, B., Balazinska, M., and Ernst, M. D. 2010. HaLoop: Efficient iterative data processing on large clusters. *Proceedings of the VLDB Endowment 3,* 1, 285–296.

Bueno, F., Debray, S. K., García de la Banda, M., and Hermenegildo, M. V. 1994. QE-Andorra: A Quiche–Eating implementation of the basic Andorra model. Technical Report CLIP13/94.0, TU of Madrid (UPM).

Bueno, F., Deransart, P., Drabent, W., Ferrand, G., Hermenegildo, M. V., Maluszynski, J., and Puebla, G. 1997. On the role of semantic approximations in validation and diagnosis of constraint logic programs. In *Proc. of the 3rd Int'l. WS on Automated Debugging–AADEBUG*. U. Linköping Press, 155–170.

Bueno, F. and García de la Banda, M. 2004. Set-sharing is not always redundant for pair-sharing. In *7th International Symposium on Functional and Logic Programming (FLOPS 2004)*. Number 2998 in Lecture Notes in Computer Science. Springer-Verlag, Heidelberg, Germany, 117–131.

Bueno, F., García de la Banda, M., and Hermenegildo, M. V. 1999. Effectiveness of abstract interpretation in automatic parallelization: A case study in logic programming. *ACM Transactions on Programming Languages and Systems 21,* 2, 189–238.

Bueno, F., Lopez-Garcia, P., and Hermenegildo, M. V. 2004. Multivariant non-failure analysis via standard abstract interpretation. In *7th International Symposium on Functional and Logic Programming (FLOPS 2004)*. Number 2998 in Lecture Notes in Computer Science. Springer-Verlag, Heidelberg, Germany, 100–116.

Cabeza, D. 2004. An extensible, global analysis friendly logic programming system. Ph.D. thesis, Universidad Politécnica de Madrid (UPM), Facultad Informatica UPM, 28660-Boadilla del Monte, Madrid-Spain.

Cabeza, D. and Hermenegildo, M. V. 1996. Implementing distributed concurrent constraint execution in the CIAO system. In *Proc. of the AGP'96 Joint Conference on Declarative Programming*. 67–78.

Cabeza, D. and Hermenegildo, M. V. 2009. Non-strict independence-based program parallelization using sharing and freeness information. *Theoretical Computer Science 410,* 46, 4704–4723.

Calegari, R., Denti, E., Mariani, S., and Omicini, A. 2018. Logic programming as a service. *Theory and Practice of Logic Programming 18,* 5-6, 846–873. Special Issue "Past and Present (and Future) of Parallel and Distributed Computation in (Constraint) Logic Programming".

Calimeri, F., Perri, S., and Ricca, F. 2008. Experimenting with parallelism for the instantiation of ASP programs. *J. Algorithms 63,* 1-3, 34–54.

Carlsson, M. and Mildner, P. 2012. SICStus Prolog – the first 25 years. *Theory and Practice of Logic Programming 12,* 1-2, 35–66.

Carro, M. 2001. Some contributions to the study of parallelism and concurrency in logic programming. Ph.D. thesis, Universidad Politécnica de Madrid (UPM), Facultad Informatica UPM, 28660-Boadilla del Monte, Madrid-Spain.

Carro, M. and Hermenegildo, M. V. 1999. Concurrency in Prolog using threads and a shared database. In *1999 International Conference on Logic Programming*. MIT Press, Cambridge, MA, USA, 320–334.

Casas, A. 2008. Automatic unrestricted independent And-parallelism in declarative multi-paradigm languages. Ph.D. thesis, University of New Mexico (UNM), Electrical and Computer Engineering Department, University of New Mexico, Albuquerque, NM 87131-0001 (USA).

Casas, A., Carro, M., and Hermenegildo, M. V. 2007. Annotation algorithms for unrestricted independent And-parallelism in logic programs. In *17th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'07)*. Number 4915 in Lecture Notes in Computer Science. Springer-Verlag, The Technical University of Denmark, 138–153.

Casas, A., Carro, M., and Hermenegildo, M. V. 2008a. A high-level implementation of non-deterministic, unrestricted, independent And-parallelism. In *24th International Conference on Logic Programming (ICLP'08)*, M. García de la Banda and E. Pontelli, Eds. Lecture Notes in Computer Science, vol. 5366. Springer-Verlag, Heidelberg, Germany, 651–666.

Casas, A., Carro, M., and Hermenegildo, M. V. 2008b. Towards a high-level implementation of execution primitives for non-restricted, independent And-parallelism. In *10th International Symposium on Practical Aspects of Declarative Languages (PADL'08)*, D. S. Warren and P. Hudak, Eds. Lecture Notes in Computer Science, vol. 4902. Springer-Verlag, Heidelberg, Germany, 230–247.

Chen, W. and Warren, D. S. 1996. Tabled evaluation with delaying for general logic programs. *Journal of the ACM 43,* 1, 20–74.

Chico de Guzmán, P., Carro, M., Hermenegildo, M. V., Silva, C., and Rocha, R. 2008. An improved continuation call-based implementation of tabling. In *International Symposium on Practical Aspects of Declarative Languages*. Number 4902 in Lecture Notes in Computer Science. Springer, Heidelberg, Germany, 197–213.

Chico de Guzmán, P., Casas, A., Carro, M., and Hermenegildo, M. V. 2011. Parallel backtracking with answer memoing for independent And-parallelism. *Theory and Practice of Logic Programming, 27th Int'l. Conference on Logic Programming (ICLP'11) Special Issue 11,* 4–5, 555–574.

Chico de Guzmán, P., Casas, A., Carro, M., and Hermenegildo, M. V. 2012. A segment-swapping approach for executing trapped computations. In *PADL'12*, N.-F. Zhou and C. Russo, Eds. Lecture Notes in Computer Science, vol. 7149. Springer Verlag, Heidelberg, Germany, 138–152.

Chin, B., von Dincklage, D., Ercegovac, V., Hawkins, P., Miller, M. S., Och, F. J., Olston, C., and Pereira, F. 2015. Yedalog: Exploring knowledge at scale. In *1st Summit on Advances in Programming Languages, SNAPL 2015, May 3-6, 2015, Asilomar, California, USA*, T. Ball, R. Bodík, S. Krishnamurthi, B. S. Lerner, and G. Morrisett, Eds. LIPICS, vol. 32. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 63–78.

Chisham, B., Wright, B., Le, T., Son, T. C., and Pontelli, E. 2011. CDAO-Store: Ontology-driven data integration for phylogenetic analysis. *BMC Bioinformatics 12*, 98.

Ciancarini, P. 1990. Blackboard programming in shared Prolog. In *Languages and Compilers for Parallel Computing*, D. Gelernter, A. Nicolau, and D. Padua, Eds. MIT Press, Cambridge, MA, USA, 170–185.

Citrigno, S., Eiter, T., Faber, W., Gottlob, G., Koch, C., Leone, N., Mateis, C., Pfeifer, G., and Scarcello, F. 1997. The DLV system: Model generator and application frontends. In *Twelfth Workshop Logic Programming, WLP 1997, 17-19 September 1997, München, Germany, Technical Report PMS-FB-1997-10*. Ludwig Maximilians Universität München, 128–137.

Clark, K. and Gregory, S. 1986. Parlog: Parallel programming in logic. *Transactions on Programming Languages and Systems 8,* 1, 1–49.

Clocksin, W. and Alshawi, H. 1988. A method for efficiently executing Horn clause programs using multiple processors. *New Generation Computing 6,* 5, 361–36.

Codish, M., Lagoon, V., and Bueno, F. 2000. An algebraic approach to sharing analysis of logic programs. *Journal of Logic Programming 42,* 2, 111–149.

Codish, M. and Shapiro, E. Y. 1986. Compiling Or-parallelism into And-parallelism. In *Third International Conference on Logic Programming*. Number 225 in Lecture Notes in Computer Science. Imperial College, Springer-Verlag, Heidelberg, Germany, 283–298.

Condie, T., Das, A., Interlandi, M., Shkapsky, A., Yang, M., and Zaniolo, C. 2018. Scaling-up reasoning and advanced analytics on BigData. *Theory and Practice of Logic Programming 18,* 5–6, 806–845.

Conway, T. 2002. Towards parallel Mercury. Ph.D. thesis, University of Melbourne.

Cousot, P. and Cousot, R. 1977. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Records of the ACM Symposium on Principles of Programming Languages*. ACM Press, New York, 238–252.

Dal Palù, A., Dovier, A., Formisano, A., and Pontelli, E. 2015. CUD@SAT: SAT solving on gpus. *J. of Experimental & Theoretical Artificial Intelligence (JETAI) 27,* 3, 293–316.

Dal Palù, A., Dovier, A., Pontelli, E., and Rossi, G. 2009. GASP: Answer set programming with lazy grounding. *Fundamenta Informaticae 96,* 3, 297–322.

Damásio, C. V. 2000. A distributed tabling system. In *Conference on Tabulation in Parsing and Deduction (TAPD2000), Proceedings*. University of Vigo, 65–75.

Das, A. and Zaniolo, C. 2019. A case for stale synchronous distributed model for declarative recursive computation. *Theory and Practice of Logic Programming 19,* 5–6, 1056–1072.

De Angelis, E., Fioravanti, F., Gallagher, J. P., Hermenegildo, M. V., Pettorossi, A., , and Proietti, M. 2022. Analysis and transformation of constrained horn clauses for program verification. *Theory and Practice of Logic Programming (to appear)*.

De Angelis, E., Fioravanti, F., Pettorossi, A., and Proietti, M. 2015. Semantics-based generation of verification conditions by program specialization. In *17th International Symposium on Principles and Practice of Declarative Programming*. ACM, New York, 91–102.

De Bortoli, M., Igne, F., Tardivo, F., Totis, P., Dovier, A., and Pontelli, E. 2019. Towards distributed computation of answer sets. In *Proceedings of the 34th Italian Conference on Computational Logic*. CEUR Workshop Proceedings, vol. 2396. CEUR-WS.org, Aachen, 316–326.

de Castro Dutra, I. 1994. Strategies for scheduling and- and or-parallel work in parallel logic programming systems. In *Logic Programming, Proceedings of the 1994 International Symposium, Ithaca, New York, USA, November 13-17, 1994*, M. Bruynooghe, Ed. MIT Press, Cambridge, MA, USA, 289–304.

Debray, S. K., Lin, N.-W., and Hermenegildo, M. V. 1990. Task granularity analysis in logic programs. In *Proc. 1990 ACM Conf. on Programming Language Design and Implementation (PLDI)*. ACM Press, New York, 174–188.

Debray, S. K., Lopez-Garcia, P., Hermenegildo, M. V., and Lin, N.-W. 1994. Estimating the computational cost of logic programs. In *SAS'94*. Number 864 in Lecture Notes in Computer Science. Springer-Verlag, Heidelberg, Germany, 255–265.

Debray, S. K., Lopez-Garcia, P., Hermenegildo, M. V., and Lin, N.-W. 1997. Lower bound cost estimation for logic programs. In *1997 International Logic Programming Symposium*. MIT Press, Cambridge, MA, USA, 291–305.

DeGroot, D. 1984. Restricted and-parallelism. In *Proceedings of the International Conference on Fifth Generation Computer Systems, FGCS 1984, Tokyo, Japan, November 6-9, 1984*. OHMSHA Ltd. Tokyo and North-Holland, 471–478.

Desouter, B., van Dooren, M., and Schrijvers, T. 2015. Tabling as a library with delimited control. *Theory and Practice of Logic Programming 15,* 4 & 5, 419–433.

Diamos, G. F., Wu, H., Wang, J., Lele, A., and Yalamanchili, S. 2013. Relational algorithms for multi-bulk-synchronous processors. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '13, Shenzhen, China, February 23-27, 2013*, A. Nicolau, X. Shen, S. P. Amarasinghe, and R. W. Vuduc, Eds. ACM, New York, 301–302.

Dovier, A., Formisano, A., and Pontelli, E. 2018. Parallel answer set programming. In *Handbook of Parallel Constraint Reasoning*, Y. Hamadi and L. Sais, Eds. Springer, Heidelberg, Germany, 237–282.

Dovier, A., Formisano, A., Pontelli, E., and Vella, F. 2015. Parallel execution of the ASP computation. In *Tech.Comm. of ICLP 2015*, M. De Vos, T. Eiter, Y. Lierler, and F. Toni, Eds. Vol. 1433. CEUR-WS.org, Aachen.

Dovier, A., Formisano, A., Pontelli, E., and Vella, F. 2016. A GPU implementation of the ASP computation. In *PADL 2016*, M. Gavanelli and J. H. Reppy, Eds. Lecture Notes in Computer Science, vol. 9585. Springer, Heidelberg, Germany, 30–47.

Dovier, A., Formisano, A., and Vella, F. 2019. GPU-based parallelism for ASP-solving. In *Declarative Programming and Knowledge Management - Conference on Declarative Programming, DECLARE 2019, Unifying INAP, WLP, and WFLP, Cottbus, Germany, September 9-12, 2019, Revised Selected Papers*, P. Hofstedt, S. Abreu, U. John, H. Kuchen, and D. Seipel, Eds. Lecture Notes in Computer Science, vol. 12057. Springer, Heidelberg, Germany, 3–23.

Eiter, T., Germano, S., Ianni, G., Kaminski, T., Redl, C., Schüller, P., and Weinzierl, A. 2018. The DLVHEX system. *Künstliche Intell. 32,* 2-3, 187–189.

Ekanayake, J., Li, H., Zhang, B., Gunarathne, T., Bae, S.-H., Qiu, J., and Fox, G. C. 2010. Twister: A runtime for iterative Map-Reduce. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing,*. ACM Press, New York, 810–818.

El-Khatib, O. and Pontelli, E. 2000. Parallel evaluation of answer sets programs preliminary results. In *Workshop on Parallelism and Implementation Technologies for (Constraint) Logic Programming Languages*. London, UK.

Ellguth, E., Gebser, M., Gusowski, M., Kaufmann, B., Kaminski, R., Liske, S., Schaub, T., Schneidenbach, L., and Schnor, B. 2009. A simple distributed conflict-driven answer set solver. In *Logic Programming and Non-Monotonic Reasoning*. Springer Verlag, Heidelberg, Germany, 490–495.

Fan, Z., Zhu, J., Zhang, Z., Albarghouthi, A., Koutris, P., and Patel, J. M. 2019. Scaling-up in-memory Datalog processing: Observations and techniques. *Proceedings of the VLDB Endowment 12,* 6, 695–708.

Fecht, C. 1996. An efficient and precise sharing domain for logic programs. In *PLILP*, H. Kuchen and S. D. Swierstra, Eds. Lecture Notes in Computer Science, vol. 1140. Springer, Heidelberg, Germany, 469–470.

Finkel, R., Marek, V., Moore, N., and Truszczyński, M. 2001. Computing stable models in parallel. In *Proceedings of the AAAI Spring Symposium on Answer Set Programming*, A. Provetti and S. C. Tran, Eds. AAAI/MIT Press, Cambridge, MA, USA, 72–75.

Flanagan, C. and Felleisen, M. 1995. The semantics of Future and its use in program optimization. In *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23-25, 1995*, R. K. Cytron and P. Lee, Eds. ACM Press, 209–220.

Fonseca, N. A., Silva, F. M. A., and Camacho, R. 2006. April - an inductive logic programming system. In *European Conference on Logics in Artificial Intelligence*. Number 4160 in Lecture Notes in Artificial Intelligence. Springer, Heidelberg, Germany, 481–484.

Fonseca, N. A., Srinivasan, A., Silva, F. M. A., and Camacho, R. 2009. Parallel ILP for distributed-memory architectures. *Mach. Learn. 74,* 3, 257–279.

Formisano, A., Gentilini, R., and Vella, F. 2017. Accelerating energy games solvers on modern architectures. In *Proc. of the 7th Workshop on Irregular Applications: Architectures and Algorithms, IA3@SC*. ACM, New York, 12:1–12:4.

Formisano, A., Gentilini, R., and Vella, F. 2021. Scalable energy games solvers on GPUs. *IEEE Trans. Parallel Distributed Syst. 32,* 12, 2970–2982.

Formisano, A. and Vella, F. 2014. On multiple learning schemata in conflict driven solvers. In *Proc. of ICTCS.*, S. Bistarelli and A. Formisano, Eds. CEUR Workshop Proceedings, vol. 1231. CEUR-WS.org, Aachen, 133–146.

Freire, J., Hu, R., Swift, T., and Warren, D. S. 1995. Exploiting parallelism in tabled evaluations. In *International Symposium on Programming Languages: Implementations, Logics and Programs*. Number 982 in Lecture Notes in Computer Science. Springer, Heidelberg, Germany, 115–132.

Freire, J., Swift, T., and Warren, D. S. 1996. Beyond depth-first: Improving tabled logic programs through alternative scheduling strategies. In *International Symposium on Programming Language Implementation and Logic Programming*. Number 1140 in Lecture Notes in Computer Science. Springer, Heidelberg, Germany, 243–258.

Futó, I. 1993. Prolog with communicating processes: From T-Prolog to CSR-Prolog. In *International Conference on Logic Programming*. The MIT Press, Cambridge, MA, USA, 3–17.

Futó, I. and Kacsuk, P. 1989. CS-PROLOG on multi-transputer systems. *Microprocessors and Microsystems 13,* 2, 103–112.

Gallagher, J. P., Hermenegildo, M. V., Kafle, B., Klemen, M., Lopez-Garcia, P., and Morales, J. F. 2020. From big-step to small-step semantics and back with interpreter specialization (invited paper). In *International WS on Verification and Program Transformation (VPT 2020)*. EPTCS. Open Publishing Association, 50–65.

Ganguly, S., Silberschatz, A., and Tsur, S. 1992. Parallel bottom-up processing of Datalog queries. *Journal of Logic Programming 14,* 1-2, 101–126.

Garcia-Contreras, I., Morales, J. F., and Hermenegildo, M. V. 2018. Towards incremental and modular context-sensitive analysis. In *Technical Communications of the 34th International Conference on Logic Programming (ICLP 2018)*. OpenAccess Series in Informatics (OASIcs). Dagstuhl Press. (Extended Abstract).

Garcia-Contreras, I., Morales, J. F., and Hermenegildo, M. V. 2019. Multivariant assertion-based guidance in abstract interpretation. In *Proceedings of the 28th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'18)*. Number 11408 in Lecture Notes in Computer Science. Springer-Verlag, Heidelberg, Germany, 184–201.

Garcia-Contreras, I., Morales, J. F., and Hermenegildo, M. V. 2020. Incremental analysis of logic programs with assertions and open predicates. In *Proceedings of the 29th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'19)*. Lecture Notes in Computer Science. Springer-Verlag, Heidelberg, Germany, 36–56.

García de la Banda, M. 1994. Independence, global analysis, and parallelism in dynamically scheduled constraint logic programming. Ph.D. thesis, Universidad Politecnica de Madrid.

García de la Banda, M. and Hermenegildo, M. V. 1993. A practical approach to the global analysis of constraint logic programs. In *1993 International Logic Programming Symposium*. MIT Press, Cambridge, MA, USA, 437–455.

García de la Banda, M., Hermenegildo, M. V., Bruynooghe, M., Dumortier, V., Janssens, G., and Simoens, W. 1996. Global analysis of constraint logic programs. *ACM Transactions on Programming Languages and Systems 18,* 5, 564–615.

García de la Banda, M., Hermenegildo, M. V., and Marriott, K. 2000. Independence in CLP languages. *ACM Transactions on Programming Languages and Systems 22,* 2, 269–339.

García de la Banda, M., Marriott, K., and Stuckey, P. J. 1995. Efficient analysis of constraint logic programs with dynamic scheduling. In *1995 International Logic Programming Symposium*. MIT Press, Cambridge, MA, USA, 417–431.

Gebser, M., Kaminski, R., Kaufmann, B., and Schaub, T. 2012. *Answer Set Solving in Practice*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers.

Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T., Schneider, M. T., and Ziller, S. 2011. A portfolio solver for answer set programming: Preliminary report. In *Logic Programming and Nonmonotonic Reasoning - 11th International Conference, LPNMR 2011, Vancouver, Canada, May 16-19, 2011. Proceedings*, J. P. Delgrande and W. Faber, Eds. Lecture Notes in Computer Science, vol. 6645. Springer, Heidelberg, Germany, 352–357.

Gebser, M., Kaufmann, B., Kaminski, R., Ostrowski, M., Schaub, T., and Schneider, M. 2011. Potassco: The Potsdam answer set solving collection. *AI Commun. 24,* 2, 107–124.

Gebser, M., Leone, N., Maratea, M., Perri, S., Ricca, F., and Schaub, T. 2018. Evaluation techniques and systems for answer set programming: a survey. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden*, J. Lang, Ed. ijcai.org, 5450–5456.

Gebser, M., Maratea, M., and Ricca, F. 2020. The seventh answer set programming competition: Design and results. *Theory Pract. Log. Program. 20,* 2, 176–204.

Gelfond, M. and Kahl, Y. 2014. *Knowledge Representation, Reasoning, and the Design of Intelligent Agents: The Answer-Set Programming Approach*. Cambridge University Press, UK.

Gent, I. P., Miguel, I., Nightingale, P., McCreesh, C., Prosser, P., Moore, N. C. A., and Unsworth, C. 2018. A review of literature on parallel constraint solving. *Theory and Practice of Logic Programming 18,* 5-6, 725–758.

Goldberg, E. and Novikov, Y. 2007. Berkmin: A fast and robust sat-solver. *Discret. Appl. Math. 155,* 12, 1549–1561.

Gomes, C. P. and Selman, B. 2001. Algorithm portfolios. *Artif. Intell. 126,* 1-2, 43–62.

Gómez-Zamalloa, M., Albert, E., and Puebla, G. 2009. Decompilation of Java bytecode to Prolog by partial evaluation. *JIST 51,* 1409–1427.

Grebenshchikov, S., Lopes, N. P., Popeea, C., and Rybalchenko, A. 2012. Synthesizing software verifiers from proof rules. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12*, J. Vitek, H. Lin, and F. Tip, Eds. ACM, New York, 405–416.

Guo, H.-F. and Gupta, G. 2001. A simple scheme for implementing tabled logic programming systems based on dynamic reordering of alternatives. In *International Conference on Logic Programming*. Number 2237 in Lecture Notes in Computer Science. Springer, Heidelberg, Germany, 181–196.

Gupta, G. and Jayaraman, B. 1990. On criteria for Or-parallel execution models of logic programs. In *1990 N. American Conf. on Logic Prog.* MIT Press, Cambridge, MA, USA, 604–623.

Gupta, G. and Pontelli, E. 1997. Extended dynamic dependent and-parallelism in ACE. In *Proceedings of the 2nd International Workshop on Parallel Symbolic Computation, PASCO 1997, July 20-22, 1997, Kihei, Hawaii, USA*, H. Hong, E. Kaltofen, and M. A. Hitz, Eds. ACM, New York, 68–79.

Gupta, G. and Pontelli, E. 1999. Stack-splitting: A simple technique for implementing Or-parallelism and And-parallelism on distributed machines. In *International Conference on Logic Programming*, D. De Schreye, Ed. MIT Press, Cambridge, MA, USA, 290–304.

Gupta, G., Pontelli, E., Ali, K. A. M., Carlsson, M., and Hermenegildo, M. V. 2001. Parallel execution of Prolog programs: a survey. *ACM Transactions on Programming Languages and Systems 23,* 4, 472–602.

Gupta, G. and Warren, D. H. D. 1992. An interpreter for the extended Andorra model (preliminary report). Technical report, Univ. of Bristol, UK.

Gurfinkel, A., Kahsai, T., Komuravelli, A., and Navas, J. A. 2015. The SeaHorn verification framework. In *International Conference on Computer Aided Verification, CAV 2015*. Number 9206 in Lecture Notes in Computer Science. Springer, Heidelberg, Germany, 343–361.

Gustafson, J. L. 1988. Reevaluating Amdahl's Law. *Commun. ACM 31,* 5, 532–533.

Halstead Jr., R. H. 1985. Multilisp: A language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst. 7,* 4, 501–538.

Haridi, S. and Brand, P. 1988. Andorra Prolog: An integration of Prolog and committed choice languages. In *Proceedings of the International Conference on Fifth Generation Computer Systems, FGCS 1988, Tokyo, Japan, November 28-December 2, 1988*. OHMSHA Ltd. Tokyo and Springer-Verlag, 745–754.

Haridi, S. and Janson, S. 1990. Kernel Andorra Prolog and its computation model. In *Proceedings of the International Conference on Logic Programming*, D. H. D. Warren and P. Szeredi, Eds. MIT Press, Cambridge, MA, USA, 31–46.

Hausman, B., Ciepielewski, A., and Haridi, S. 1987. Or-parallel Prolog made efficient on shared memory multiprocessors. In *Symposium on Logic Programming*. IEEE Computer Society, USA, 69–79.

Henriksen, K. S. and Gallagher, J. P. 2006. Abstract interpretation of PIC programs through logic programming. In *SCAM '06*. IEEE Computer Society, USA, 184–196.

Hermenegildo, M. V. 1986a. An abstract machine based execution model for computer architecture design and efficient implementation of logic programs in parallel. Ph.D. thesis, U. of Texas at Austin.

Hermenegildo, M. V. 1986b. An abstract machine for restricted And-parallel execution of logic programs. In *Third International Conference on Logic Programming*. Number 225 in Lecture Notes in Computer Science. Imperial College, Springer-Verlag, Heidelberg, Germany, 25–40.

Hermenegildo, M. V. 2000. Parallelizing irregular and pointer-based computations automatically: Perspectives from logic and constraint programming. *Parallel Computing 26,* 13–14, 1685–1708.

Hermenegildo, M. V., Bueno, F., Carro, M., Lopez-Garcia, P., Mera, E., Morales, J. F., and Puebla, G. 2012. An overview of CIAO and its design philosophy. *Theory and Practice of Logic Programming 12,* 1–2, 219–252.

Hermenegildo, M. V., Bueno, F., García de la Banda, M., and Puebla, G. 1995. The CIAO multi-dialect compiler and system: An experimentation workbench for future (C)LP systems. In *Proceedings of the ILPS'95 Workshop on Visions for the Future of Logic Programming*. Portland, Oregon, USA. Available from `http://www.cliplab.org/`.

Hermenegildo, M. V., Cabeza, D., and Carro, M. 1995. Using attributed variables in the implementation of concurrent and parallel logic programming systems. In *ICLP'95*. MIT Press, Cambridge, MA, USA, 631–645.

Hermenegildo, M. V. and Nasr, R. I. 1986. Efficient management of backtracking in And-parallelism. In *Third International Conference on Logic Programming*, E. Y. Shapiro, Ed. Number 225 in Lecture Notes in Computer Science. Springer-Verlag, Heidelberg, Germany, 40–54.

Hermenegildo, M. V., Puebla, G., and Bueno, F. 1999. Using global analysis, partial specifications, and an extensible assertion language for program validation and debugging. In *The Logic Programming Paradigm: a 25–Year Perspective*. Springer-Verlag, Heidelberg, Germany, 161–192.

Hermenegildo, M. V., Puebla, G., Bueno, F., and Garcia, P. L. 2005. Integrated program debugging, verification, and optimization using abstract interpretation (and the CIAO system preprocessor). *Science of Computer Programming 58,* 1–2, 115–140.

Hermenegildo, M. V. and Rossi, F. 1995. Strict and non-strict independent And-parallelism in logic programs: Correctness, efficiency, and compile-time conditions. *Journal of Logic Programming 22,* 1, 1–45.

Hill, P. M., Bagnara, R., and Zaffanella, E. 2002. Soundness, idempotence and commutativity of set-sharing. *Theory and Practice of Logic Programming 2,* 2, 155–201.

Hoos, H. H., Kaminski, R., Lindauer, M., and Schaub, T. 2015. aspeed: solver scheduling via answer set programming. *Theory and Practice of Logic Programming 15,* 1, 117–142.

Hoos, H. H., Lindauer, M., and Schaub, T. 2014. claspfolio 2: Advances in algorithm selection for answer set programming. *Theory and Practice of Logic Programming 14,* 4-5, 569–585.

Hu, R. 1997. Efficient tabled evaluation of normal logic programs in a distributed environment. Ph.D. thesis, Department of Computer Science, State University of New York.

Huang, Y. and Chen, W. 2015. Parallel query on the in-memory database in a CUDA platform. In *10th International Conference on P2P, Parallel, Grid, Cloud and Internet Computing, 3PGCIC 2015, Krakow, Poland, November 4-6, 2015*, F. Xhafa, L. Barolli, F. Messina, and M. R. Ogiela, Eds. IEEE Computer Society, USA, 236–243.

Igne, F., Dovier, A., and Pontelli, E. 2018. MASP-Reduce: A proposal for distributed computation of stable models. In *Technical Communications of the 34th International Conference on Logic Programming*. OASICS 64, Schloss Dagstuhl, 8:1–8:4.

Jacobs, D. and Langen, A. 1989. Accurate and efficient approximation of variable aliasing in logic programs. In *1989 North American Conference on Logic Programming*. MIT Press, Cambridge, MA, USA.

Janakiram, V., Agrawal, D., and Mehrotra, R. 1988. A Randomized Parallel Backtracking Algorithm. *IEEE Transactions on Computers 37,* 12, 1665–1676.

Jordan, H., Scholz, B., and Subotic, P. 2016. Soufflé: On synthesis of program analyzers. In *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*, S. Chaudhuri and A. Farzan, Eds. Lecture Notes in Computer Science, vol. 9780. Springer, Heidelberg, Germany, 422–430.

Jordan, H., Subotic, P., Zhao, D., and Scholz, B. 2019. A specialized B-tree for concurrent Datalog evaluation. In *Proceedings of the 24th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2019, Washington, DC, USA, February 16-20, 2019*, J. K. Hollingsworth and I. Keidar, Eds. ACM, New York, 327–339.

Kahsai, T., Rümmer, P., Sanchez, H., and Schäf, M. 2016. JayHorn: A framework for verifying Java programs. In *Computer Aided Verification - 28th International Conference, CAV 2016*, S. Chaudhuri and A. Farzan, Eds. Lecture Notes in Computer Science, vol. 9779. Springer, Heidelberg, Germany, 352–358.

Karau, H., Konwinski, A., Wendell, P., and Zaharia, M. 2015. *Learning Spark*. O'Reilly, USA.

Kelly, A., Marriott, K., Søndergaard, H., and Stuckey, P. J. 1998. A practical object-oriented analysis engine for CLP. *Software: Practice and Experience 28*, 2, 188–224.

Khronos Group Inc. 2015. *OpenCL: The open standard for parallel programming of heterogeneous systems*. http://www.khronos.org.

Klemen, M., Lopez-Garcia, P., Gallagher, J. P., Morales, J. F., and Hermenegildo, M. V. 2020. A general framework for static cost analysis of parallel logic programs. In *Proceedings of the 29th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'19)*, M. Gabbrielli, Ed. Lecture Notes in Computer Science, vol. 12042. Springer-Verlag, Heidelberg, Germany, 19–35.

Klemen, M., Stulova, N., Lopez-Garcia, P., Morales, J. F., and Hermenegildo, M. V. 2018. Static performance guarantees for programs with run-time checks. In *20th Int'l. ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP'18)*. ACM Press, New York.

Konczak, K., Linke, T., and Schaub, T. 2006. Graphs and colorings for answer set programming. *Theory and Practice of Logic Programming 6*, 1-2, 61–106.

Körner, P., Leuschel, M., Barbosa, J., Santos Costa, V., Dahl, V., Hermenegildo, M., Morales, J., Wielemaker, J., Diaz, D., Abreu, S., and Ciatto, G. 2022. A Multi-Walk Through the Past, Present and Future of Prolog. *Theory and Practice of Logic Programming (to appear)*.

Kowalski, R. A. 1979. *Logic for Problem Solving*. Elsevier North-Holland Inc.

Lagoon, V. and Stuckey, P. J. 2002. Precise pair-sharing analysis of logic programs. In *Principles and Practice of Declarative Programming*. ACM Press, New York, 99–108.

Le, H. and Pontelli, E. 2005. An investigation of sharing strategies for answer set solvers and SAT solvers. In *Euro-Par*. Springer Verlag, Heidelberg, Germany, 750–760.

Le, H. and Pontelli, E. 2007. Dynamic scheduling in parallel answer set programming solvers. In *Proceedings of the 2007 Spring Simulation Multiconference, SpringSim 2007*. ACM Press, New York, 367–374.

Le Charlier, B. and Van Hentenryck, P. 1994. Experimental evaluation of a generic abstract interpretation algorithm for Prolog. *ACM Transactions on Programming Languages and Systems 16*, 1, 35–101.

Leutgeb, L. and Weinzierl, A. 2017. Techniques for efficient lazy-grounding ASP solving. In *Declarative Programming and Knowledge Management - Conference on Declarative Programming*. Lecture Notes in Computer Science, vol. 10997. Springer, Heidelberg, Germany, 132–148.

Li, X., King, A., and Lu, L. 2006. Lazy set-sharing analysis. In *8th. Int'l. Symp. on Functional and Logic Programming*, P. Wadler and M. Hagiya, Eds. Lecture Notes in Computer Science. Springer-Verlag, Heidelberg, Germany, 177–191.

Lin, Z. 1989. Expected performance of the randomized parallel backtracking method. In *Proceedings of the North American Conference on Logic Programming*. The MIT Press, Cambridge, MA, USA, 677–696.

Lindauer, M., Hoos, H. H., Hutter, F., and Schaub, T. 2015. AutoFolio: An automatically configured algorithm selector. *J. Artif. Intell. Res. 53*, 745–778.

Liqat, U., Georgiou, K., Kerrison, S., Lopez-Garcia, P., Hermenegildo, M. V., Gallagher, J. P., and Eder, K. 2016. Inferring parametric energy consumption functions at different software levels: ISA vs. LLVM IR. In *Foundational and Practical Aspects of Resource Analysis: 4th International Workshop, FOPARA 2015, London, UK, April 11, 2015. Revised Selected Papers*, M. V. Eekelen and U. D. Lago, Eds. Lecture Notes in Computer Science, vol. 9964. Springer, Heidelberg, Germany, 81–100.

Liqat, U., Kerrison, S., Serrano, A., Georgiou, K., Lopez-Garcia, P., Grech, N., Hermenegildo, M. V., and Eder, K. 2014. Energy consumption analysis of programs based on XMOS ISA-level models. In *Logic-Based Program Synthesis and Transformation, 23rd International Symposium, LOPSTR 2013, Revised Selected Papers*, G. Gupta and R. Peña, Eds. Lecture Notes in Computer Science, vol. 8901. Springer, Heidelberg, Germany, 72–90.

Liu, L., Pontelli, E., Son, T. C., and Truszczyński, M. 2010. Logic programs with abstract constraint atoms: The role of computations. *Artificial Intelligence 174,* 3-4, 295–315.

Lloyd, J. W. 1987. *Foundations of Logic Programming*. Springer-Verlag, Heidelberg, Germany.

Lopes, R., Santos Costa, V., and Silva, F. M. A. 2003. On the BEAM implementation. In *11th Portuguese Conference on Artificial Intelligence, EPIA 2003*. Springer Verlag, Heidelberg, Germany, 131–135.

Lopes, R., Santos Costa, V., and Silva, F. M. A. 2004. Exploiting parallelism in the extended Andorra model. In *Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Networks*. IASTED/ACTA, 483–489.

Lopes, R., Santos Costa, V., and Silva, F. M. A. 2012. A design and implementation of the extended Andorra model. *Theory and Practice of Logic Programming 12,* 3, 319–360.

Lopez-Garcia, P. 2000. Non-failure analysis and granularity control in parallel execution of logic programs. Ph.D. thesis, Universidad Politécnica de Madrid (UPM), Facultad Informatica UPM, 28660-Boadilla del Monte, Madrid-Spain.

Lopez-Garcia, P., Bueno, F., and Hermenegildo, M. V. 2010. Automatic inference of determinacy and mutual exclusion for logic programs using mode and type information. *New Generation Computing 28,* 2, 117–206.

Lopez-Garcia, P., Darmawan, L., Klemen, M., Liqat, U., Bueno, F., and Hermenegildo, M. V. 2018. Interval-based resource usage verification by translation into Horn clauses and an application to energy consumption. *Theory and Practice of Logic Programming, Special Issue on Computational Logic for Verification 18,* 2, 167–223. arXiv:1803.04451.

Lopez-Garcia, P., Hermenegildo, M. V., and Debray, S. K. 1996. A methodology for granularity based control of parallelism in logic programs. *Journal of Symbolic Computation, Special Issue on Parallel Symbolic Computation 21,* 4–6, 715–734.

Lumsdaine, A., Gregor, D., Hendrickson, B., and Berry, J. 2007. Challenges in parallel graph processing. *Parallel Processing Letters 17,* 01, 5–20.

Lusk, E., Butler, R., Disz, T., Olson, R., Stevens, R., Warren, D. H. D., Calderwood, A., Szeredi, P., Brand, P., Carlsson, M., Ciepielewski, A., Hausman, B., and Haridi, S. 1990. The Aurora Or-parallel Prolog system. *New Generation Computing 7,* 2/3, 243–271.

Maiterth, M. 2012. Parallel Datalog on Pregel. M.S. thesis, Ludwig-Maximilians Universitat Munchen.

Malewicz, G., Austern, M. H., Bik, A. J. C., Dehnert, J. C., Horn, I., Leiser, N., and Czajkowski, G. 2010. Pregel: a system for large-scale graph processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*, A. K. Elmagarmid and D. Agrawal, Eds. ACM, New York, 135–146.

Malitsky, Y., Sabharwal, A., Samulowitz, H., and Sellmann, M. 2012. Parallel SAT solver selection and scheduling. In *Principles and Practice of Constraint Programming - 18th International Conference, CP 2012, Québec City, QC, Canada. Proceedings*, M. Milano, Ed. Lecture Notes in Computer Science, vol. 7514. Springer, Heidelberg, Germany, 512–526.

Maratea, M., Pulina, L., and Ricca, F. 2014. A multi-engine approach to answer-set programming. *Theory and Practice of Logic Programming 14,* 6, 841–868.

Marques, R. and Swift, T. 2008. Concurrent and local evaluation of normal programs. In *International Conference on Logic Programming*. Number 5366 in Lecture Notes in Computer Science. Springer, Heidelberg, Germany, 206–222.

Marques, R., Swift, T., and Cunha, J. C. 2010. A simple and efficient implementation of concurrent local tabling. In *International Symposium on Practical Aspects of Declarative Languages*. Number 5937 in Lecture Notes in Computer Science. Springer, Heidelberg, Germany, 264–278.

Marriott, K., García de la Banda, M., and Hermenegildo, M. V. 1994. Analyzing logic programs with dynamic scheduling. In *20th. Annual ACM Conf. on Principles of Programming Languages*. ACM, New York, 240–254.

Marron, M., Hermenegildo, M. V., Kapur, D., and Stefanovic, D. 2008. Efficient context-sensitive shape analysis with graph-based heap models. In *International Conference on Compiler Construction (CC 2008)*, L. Hendren, Ed. Lecture Notes in Computer Science. Springer, Heidelberg, Germany, 245–259.

Marron, M., Kapur, D., and Hermenegildo, M. V. 2009. Identification of logically related heap regions. In *ISMM'09: Proceedings of the 8th international symposium on Memory management*. ACM Press, New York, 89–98.

Marron, M., Kapur, D., Stefanovic, D., and Hermenegildo, M. V. 2006. A static heap analysis for shape and connectivity. In *Languages and Compilers for Parallel Computing (LCPC'06)*, G. Almási, C. Cascaval, and P. Wu, Eds. Lecture Notes in Computer Science, vol. 4382. Springer, Heidelberg, Germany, 345–363.

Marron, M., Kapur, D., Stefanovic, D., and Hermenegildo, M. V. 2008. Identification of heap-carried data dependence via explicit store heap models. In *21st Int'l. WS on Languages and Compilers for Parallel Computing (LCPC'08)*. Lecture Notes in Computer Science. Springer-Verlag, Heidelberg, Germany, 94–108.

Marron, M., Méndez-Lojo, M., Hermenegildo, M. V., Stefanovic, D., and Kapur, D. 2008. Sharing analysis of arrays, collections, and recursive structures. In *ACM WS on Program Analysis for Software Tools and Engineering (PASTE'08)*. ACM, New York, 43–49.

Martinez-Angeles, C. A., de Castro Dutra, I., Santos Costa, V., and Buenabad-Chávez, J. 2014. A Datalog engine for GPUs. In *Declarative Programming and Knowledge Management - Declarative Programming Days, KDPD 2013, Unifying INAP, WFLP, and WLP, Kiel, Germany, September 11-13, 2013, Revised Selected Papers*, M. Hanus and R. Rocha, Eds. Lecture Notes in Computer Science, vol. 8439. Springer, Heidelberg, Germany, 152–168.

Martinez-Angeles, C. A., Wu, H., de Castro Dutra, I., Santos Costa, V., and Buenabad-Chávez, J. 2016. Relational learning with GPUs: Accelerating rule coverage. *Int. J. Parallel Program. 44,* 3, 663–685.

Mattern, F. 1989. Global quiescence detection based on credit distribution and recovery. *Information Processing Letters 30,* 4, 195–200.

Méndez-Lojo, M. and Hermenegildo, M. V. 2008. Precise set sharing analysis for Java-style programs. In *9th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'08)*. Number 4905 in Lecture Notes in Computer Science. Springer-Verlag, Heidelberg, Germany, 172–187.

Méndez-Lojo, M., Lhoták, O., and Hermenegildo, M. V. 2008. Efficient set sharing using ZBDDs. In *21st Int'l. Workshop on Languages and Compilers for Parallel Computing (LCPC'08)*. Lecture Notes in Computer Science, vol. 5335. Springer-Verlag, Heidelberg, Germany, 94–108.

Méndez-Lojo, M., Navas, J., and Hermenegildo, M. V. 2007. A flexible (C)LP-based approach to the analysis of object-oriented programs. In *LOPSTR*. Lecture Notes in Computer Science, vol. 4915. Springer-Verlag, Heidelberg, Germany, 154–168.

Mera, E., Lopez-Garcia, P., Carro, M., and Hermenegildo, M. V. 2008. Towards execution time estimation in abstract machine-based languages. In *10th Int'l. ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP'08)*. ACM Press, New York, 174–184.

Moustafa, W. E., Papavasileiou, V., Yocum, K., and Deutsch, A. 2016. Datalography: Scaling Datalog graph analytics on graph processing systems. In *2016 IEEE International Conference on Big Data, BigData 2016, Washington DC, USA, December 5-8, 2016*, J. Joshi, G. Karypis, L. Liu, X. Hu, R. Ak, Y. Xia, W. Xu, A. Sato, S. Rachuri, L. H. Ungar, P. S. Yu, R. Govindaraju, and T. Suzumura, Eds. IEEE Computer Society, USA, 56–65.

Muthukumar, K., Bueno, F., García de la Banda, M., and Hermenegildo, M. V. 1999. Automatic compile-time parallelization of logic programs for restricted, goal-level, independent And-parallelism. *Journal of Logic Programming 38,* 2, 165–218.

Muthukumar, K. and Hermenegildo, M. V. 1989. Determination of variable dependence information through abstract interpretation. In *Logic Programming, Proceedings of the North American Conference 1989, Cleveland, Ohio*, E. L. Lusk and R. A. Overbeek, Eds. MIT Press, Cambridge, MA, USA, 166–185.

Muthukumar, K. and Hermenegildo, M. V. 1990. Deriving a fixpoint computation algorithm for top-down abstract interpretation of logic programs. Technical Report ACT-DC-153-90, Microelectronics and Computer Technology Corporation (MCC), Austin, TX 78759.

Muthukumar, K. and Hermenegildo, M. V. 1991. Combined determination of sharing and freeness of program variables through abstract interpretation. In *Logic Programming, Proceedings of the Eigth International Conference, Paris, France*, K. Furukawa, Ed. MIT Press, Cambridge, MA, USA, 49–63.

Muthukumar, K. and Hermenegildo, M. V. 1992. Compile-time derivation of variable dependency using abstract interpretation. *Journal of Logic Programming 13,* 2/3, 315–347.

Nappa, P., Zhao, D., Subotic, P., and Scholz, B. 2019. Fast parallel equivalence relations in a Datalog compiler. In *28th International Conference on Parallel Architectures and Compilation Techniques, PACT 2019, Seattle, WA, USA, September 23-26, 2019*. IEEE, USA, 82–96.

Navas, J., Bueno, F., and Hermenegildo, M. V. 2006. Efficient top-down set-sharing analysis using cliques. In *8th International Symposium on Practical Aspects of Declarative Languages (PADL'06)*. Number 2819 in Lecture Notes in Computer Science. Springer-Verlag, Heidelberg, Germany, 183–198.

Navas, J., Méndez-Lojo, M., and Hermenegildo, M. V. 2008. Safe upper-bounds inference of energy consumption for Java bytecode applications. In *The Sixth NASA Langley Formal Methods Workshop (LFM 08)*. NASA Langley Research Center, Hampton, Virginia, USA, 29–32. Extended Abstract.

Navas, J., Méndez-Lojo, M., and Hermenegildo, M. V. 2009. User-definable resource usage bounds analysis for Java bytecode. *Electronic Notes in Theoretical Computer Science 253,* 5, 65–82.

Navas, J., Mera, E., Lopez-Garcia, P., and Hermenegildo, M. V. 2007. User-definable resource bounds analysis for logic programs. In *Proc. of ICLP'07*. Lecture Notes in Computer Science, vol. 4670. Springer, Heidelberg, Germany, 348–363.

Nguyen, H. D., Sakama, C., Sato, T., and Inoue, K. 2018. Computing logic programming semantics in linear algebra. In *Multi-disciplinary Trends in Artificial Intelligence - 12th International Conference, MIWAI 2018, Hanoi, Vietnam, November 18-20, 2018, Proceedings*, M. Kaenampornpan, R. Malaka, D. D. Nguyen, and N. Schwind, Eds. Lecture Notes in Computer Science, vol. 11248. Springer, Heidelberg, Germany, 32–48.

Niemela, I. and Simons, P. 1997. Smodels - an implementation of the stable model and well-founded semantics for normal LP. In *Logic Programming and Non-monotonic Reasoning*. Springer Verlag, Heidelberg, Germany, 421–430.

NVIDIA Corporation. 2021. *NVIDIA CUDA Zone*. https://developer.nvidia.com/cuda-zone.

Peralta, J. C., Gallagher, J. P., and Sağlam, H. 1998. Analysis of imperative programs through analysis of constraint logic programs. In *Static Analysis. 5th International Symposium, SAS'98, Pisa*, G. Levi, Ed. Lecture Notes in Computer Science, vol. 1503. Springer, Heidelberg, Germany, 246–261.

Pereira, L. M., Monteiro, L., Cunha, J., and Aparcio, J. N. 1986. Delta Prolog: A distributed

backtracking extension with events. In *International Conference on Logic Programming*, E. Shapiro, Ed. Lecture Notes in Computer Science, vol. 225. Springer Verlag, Heidelberg, Germany, 69–83.

Pereira, L. M. and Nasr, R. I. 1984. Delta-Prolog: A distributed logic programming language. In *Proceedings of the International Conference on Fifth Generation Computer Systems, FGCS 1984, Tokyo, Japan*. OHMSHA Ltd. Tokyo and North-Holland, 283–291.

Perri, S., Ricca, F., and Sirianni, M. 2013. Parallel instantiation of ASP programs: techniques and experiments. *Theory and Practice of Logic Programming 13,* 2, 253–278.

Pollard, G. H. 1981. Parallel execution of Horn clause programs. Ph.D. thesis, Imperial College, London. Dept. of Computing.

Pontelli, E. 2001. Experiments in parallel execution of answer set programs. In *International Parallel and Distributed Processing Symposium*. IEEE Computer Society, USA, 20.

Pontelli, E. and Gupta, G. 1995. On the duality between And-parallelism and Or-parallelism. In *Proceedings of EuroPar*, S. Haridi and P. Magnusson, Eds. Springer Verlag, Heidelberg, Germany, 43–54.

Pontelli, E. and Gupta, G. 1997. Parallel symbolic computation in ACE. *Annals of Mathematics and Artificial Intelligence 21,* 2-4, 359–395.

Pontelli, E. and Gupta, G. 1999. A simulation study of distributed execution of constraint logic programs with stack splitting. Tech. rep., New Mexico State University.

Pontelli, E. and Gupta, G. 2001. Backtracking in independent And-parallel implementations of logic programming languages. *Transactions on Parallel and Distributed Systems 12,* 11, 1169–1189.

Pontelli, E., Gupta, G., and Hermenegildo, M. V. 1995. &ACE: A high-performance parallel Prolog system. In *Proceedings of the International Parallel Processing Symposium*. IEEE Computer Society, USA, 564–571.

Pontelli, E., Gupta, G., Pulvirenti, F., and Ferro, A. 1997. Automatic compile-time parallelization of Prolog programs for dependent And-parallelism. In *Proc. of the Fourteenth International Conference on Logic Programming*, L. Naish, Ed. MIT Press, Cambridge, MA, USA, 108–122.

Pontelli, E., Gupta, G., Tang, D., Carro, M., and Hermenegildo, M. V. 1996. Improving the efficiency of nondeterministic And–parallel systems. *The Computer Languages Journal 22,* 2/3, 115–142.

Pontelli, E., Le, H., and Son, T. C. 2010. An investigation in parallel execution of answer set programs on distributed memory platforms. *Computer Languages, Systems and Structures 36,* 2, 158–202.

Pontelli, E., Le, T., Nguyen, H., and Son, T. C. 2012. ASP at work: An ASP implementation of PhyloWS. In *Technical Communications of the 28th International Conference on Logic Programming, ICLP 2012, Budapest, Hungary*, A. Dovier and V. Santos Costa, Eds. LIPICS, vol. 17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 359–369.

Pontelli, E., Ranjan, D., and Dal Palù, A. 2002. An optimal data structure to handle dynamic environments in non-deterministic computations. *Computer Languages 28,* 2, 181–201.

Pontelli, E., Villaverde, K., Guo, H.-F., and Gupta, G. 2006. Stack splitting: A technique for efficient exploitation of search parallelism on share-nothing platforms. *Journal of Parallel and Distributed Computing 66,* 10, 1267–1293.

Pontelli, E., Villaverde, K., Guo, H.-F., and Gupta, G. 2007. PALS: Efficient Or-parallel execution of Prolog on Beowulf clusters. *Theory and Practice of Logic Programming 7,* 6, 633–695.

Puebla, G. and Hermenegildo, M. V. 1999. Abstract multiple specialization and its application to program parallelization. *Journal of Logic Programming 41,* 2&3, 279–316.

Ranjan, D., Pontelli, E., and Gupta, G. 1999. On the complexity of Or-parallelism. *New Generation Computing 17,* 3, 285–308.

Rocha, R., Silva, F. M. A., and Martins, R. 2003. YapDss: An Or-parallel prolog system for scalable Beowulf clusters. In *11th Portuguese Conference on Artificial Intelligence, EPIA 2003,*. Springer Verlag, Heidelberg, Germany, 136–150.

Rocha, R., Silva, F. M. A., and Santos Costa, V. 1999a. Or-parallelism within tabling. In *International Workshop on Practical Aspects of Declarative Languages*. Number 1551 in Lecture Notes in Computer Science. Springer, Heidelberg, Germany, 137–151.

Rocha, R., Silva, F. M. A., and Santos Costa, V. 1999b. YapOr: an Or-parallel prolog system based on environment copying. In *Portuguese Conference on Artificial Intelligence*. Number 1695 in Lecture Notes in Artificial Intelligence. Springer, Heidelberg, Germany, 178–192.

Rocha, R., Silva, F. M. A., and Santos Costa, V. 2000. A tabling engine for the Yap Prolog system. In *APPIA-GULP-PRODE Joint Conference on Declarative Programming, La Habana, Cuba, Dec 4–6*. Cuba.

Rocha, R., Silva, F. M. A., and Santos Costa, V. 2001. On a tabling engine that can exploit Or-parallelism. In *International Conference on Logic Programming*. Number 2237 in Lecture Notes in Computer Science. Springer, Heidelberg, Germany, 43–58.

Rocha, R., Silva, F. M. A., and Santos Costa, V. 2005. On applying Or-parallelism and tabling to logic programs. *Theory and Practice of Logic Programming 5,* 1 & 2, 161–205.

Rui, R. and Tu, Y. 2017. Fast equi-join algorithms on GPUs: Design and implementation. In *Proceedings of the 29th International Conference on Scientific and Statistical Database Management, Chicago, IL, USA, June 27-29, 2017*. ACM, New York, 17:1–17:12.

Saeed, I., Young, J., and Yalamanchili, S. 2015. A portable benchmark suite for highly parallel data intensive query processing. In *Proceedings of the 2nd Workshop on Parallel Programming for Analytics Applications*. ACM, New York, 31–38.

Sagonas, K. and Swift, T. 1998. An abstract machine for tabled execution of fixed-order stratified logic programs. *ACM Transactions on Programming Languages and Systems 20,* 3, 586–634.

Santos, J. and Rocha, R. 2013. Or-parallel Prolog execution on clusters of multicores. In *2nd Symposium on Languages, Applications and Technologies*, J. P. Leal, R. Rocha, and A. Simões, Eds. OpenAccess Series in Informatics (OASIcs), vol. 29. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 9–20.

Santos, J. and Rocha, R. 2016. On the implementation of an Or-parallel prolog system for clusters of multicores. *Theory and Practice of Logic Programming 16,* 5-6, 899–915.

Santos Costa, V., de Castro Dutra, I., and Rocha, R. 2010. Threads and Or-parallelism unified. *Theory and Practice of Logic Programming 10,* 4-6, 417–432.

Santos Costa, V., Rocha, R., and Damas, L. 2012. The YAP Prolog system. *Theory and Practice of Logic Programming 12,* 1-2, 5–34.

Santos Costa, V., Warren, D. H. D., and Yang, R. 1991a. Andorra-I: A parallel Prolog system that transparently exploits both And- and Or-parallelism. In *Proceedings of the ACM Symposium on Principles and Practice of Parallel Programming*. ACM Press, New York, 83–93.

Santos Costa, V., Warren, D. H. D., and Yang, R. 1991b. The Andorra-I engine: a parallel implementation of the basic Andorra model. In *Proceedings of the International Conference on Logic Programming*, K. Furukawa, Ed. MIT Press, Cambridge, MA, USA, 825–839.

Santos Costa, V., Warren, D. H. D., and Yang, R. 1991c. The Andorra-I preprocessor: Supporting full Prolog on the basic Andorra model. In *Proceedings of the International Conference on Logic Programming*, K. Furukawa, Ed. MIT Press, Cambridge, MA, USA, 443–456.

Schneidenbach, L., Schnor, B., Gebser, M., Kaminski, R., Kaufmann, B., and Schaub, T. 2009. Experiences running a parallel answer set solver on blue gene. In *16th European PVM/MPI Users' Group Meeting*. Springer Verlag, Heidelberg, Germany, 64–72.

Secci, S. and Spoto, F. 2005. Pair-sharing analysis of object-oriented programs. In *12th International Symposium Static Analysis Symposium (SAS'05)*. Lecture Notes in Computer Science, vol. 3672. Springer, Heidelberg, Germany, 320–335.

Seo, J., Park, J., Shin, J., and Lam, M. S. 2013. Distributed SociaLite: A Datalog-based language for large-scale graph analysis. *Proceedings of the VLDB Endowment 6,* 14, 1906–1917.

Serrano, A., Lopez-Garcia, P., and Hermenegildo, M. V. 2014. Resource usage analysis of logic programs via abstract interpretation using sized types. *Theory and Practice of Logic Programming, ICLP'14 Special Issue 14,* 4-5, 739–754.

Shapiro, E. Y. 1987. *Concurrent Prolog: Collected Papers*. MIT Press, Cambridge, MA, USA.

Shapiro, E. Y. 1989. The family of concurrent logic programming languages. *ACM Computing Suveys 21,* 3, 413–510.

Shehab, E., Algergawy, A., and Sarhan, A. M. 2017. Accelerating relational database operations using both CPU and GPU co-processor. *Comput. Electr. Eng. 57*, 69–80.

Shen, K. 1996. Overview of DASWAM: Exploitation of dependent And-parallelism. *Journal of Logic Programming 29,* 1/3, 245–293.

Shen, K. and Hermenegildo, M. V. 1996. Flexible scheduling for non-deterministic, And-parallel execution of logic programs. In *Proceedings of EuroPar'96*. Number 1124 in Lecture Notes in Computer Science. Springer-Verlag, Heidelberg, Germany, 635–640.

Shkapsky, A. 2016. A declarative language for advanced analytics and its scalable implementation. Ph.D. thesis, University of California, Los Angeles, USA.

Shkapsky, A., Yang, M., Interlandi, M., Chiu, H., Condie, T., and Zaniolo, C. 2016. Big data analytics with Datalog queries on Spark. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, F. Özcan, G. Koutrika, and S. Madden, Eds. ACM, New York, 1135–1149.

Simons, P., Niemelä, I., and Soininen, T. 2002. Extending and implementing the stable model semantics. *Artif. Intell. 138,* 1-2, 181–234.

Singhal, A. and Patt, Y. N. 1989. Unification parallelism: How much can we exploit? In *Proceedings of the North American Conference on Logic Programming*, E. Lusk and R. Overbeek, Eds. MIT Press, Cambridge, MA, USA, 1135–1147.

Somogyi, Z. and Sagonas, K. 2006. Tabling in Mercury: Design and implementation. In *International Symposium on Practical Aspects of Declarative Languages*. Number 3819 in Lecture Notes in Computer Science. Springer, Heidelberg, Germany, 150–167.

Son, T. C. and Pontelli, E. 2007. Planning for biochemical pathways: A case study of answer set planning in large planning problem instances. In *Proceedings of the First International SEA'07 Workshop, Tempe, Arizona, USA, 14th May 2007*, M. D. Vos and T. Schaub, Eds. CEUR Workshop Proceedings, vol. 281. CEUR-WS.org, Aachen.

Søndergaard, H. 1986. An application of abstract interpretation of logic programs: Occur check reduction. In *European Symposium on Programming*. Number 123 in Lecture Notes in Computer Science. Springer-Verlag, Heidelberg, Germany, 327–338.

Stulova, N. 2018. Improving run-time checking in dynamic programming languages. Ph.D. thesis, Escuela Técnica Superior de Ingenieros Informáticos, UPM.

Stulova, N., Morales, J. F., and Hermenegildo, M. V. 2015. Practical run-time checking via unobtrusive property caching. *Theory and Practice of Logic Programming, 31st Int'l. Conference on Logic Programming (ICLP'15) Special Issue 15,* 04-05, 726–741.

Stulova, N., Morales, J. F., and Hermenegildo, M. V. 2018. Some trade-offs in reducing the overhead of assertion run-time checks via static analysis. *Science of Computer Programming 155*, 3–26.

Swift, T. and Warren, D. S. 2012. XSB: Extending Prolog with tabled logic programming. *Theory and Practice of Logic Programming 12,* 1 & 2, 157–187.

Tachmazidis, I. and Antoniou, G. 2013. Computing the stratified semantics of logic programs over big data through mass parallelization. In *Theory, Practice, and Applications of Rules on the Web - 7th International Symposium, RuleML 2013*, L. Morgenstern, P. S. Stefaneas, F. Lévy, A. Z. Wyner, and A. Paschke, Eds. Lecture Notes in Computer Science, vol. 8035. Springer, Heidelberg, Germany, 188–202.

Tachmazidis, I., Antoniou, G., and Faber, W. 2014. Efficient computation of the well-founded semantics over big data. *Theory and Practice of Logic Programming 14,* 4-5, 445–459.

Tachmazidis, I., Antoniou, G., Flouris, G., Kotoulas, S., and McCluskey, L. 2012. Large-scale parallel stratified defeasible reasoning. In *European Conference on Artificial Intelligence (ECAI)*. IOS Press.

Tarzariol, A. 2019. Evolution of algorithm portfolio for solving strategies. In *Proceedings of the 34th Italian Conference on Computational Logic, Trieste, Italy, June 19-21, 2019*, A. Casagrande and E. G. Omodeo, Eds. CEUR Workshop Proceedings, vol. 2396. CEUR-WS.org, Aachen, 327–341.

ter Horst, H. J. 2005. Completeness, decidability and complexity of entailment for RDF schema and a semantic extension involving the OWL vocabulary. *Journal of Web Semantics 3,* 2-3, 79–115.

Tick, E. 1995. The deevolution of concurrent logic programming languages. *Journal of Logic Programming 23,* 2, 89–123.

Trias, E., Navas, J., Ackley, E. S., Forrest, S., and Hermenegildo, M. V. 2008. Negative ternary set-sharing. In *International Conference on Logic Programming, ICLP*. Number 5366 in Lecture Notes in Computer Science. Springer-Verlag, Heidelberg, Germany, 301–316.

Trigo de la Vega, T., Lopez-García, P., and Muñoz-Hernández, S. 2010. Towards fuzzy granularity control in parallel/distributed computing. In *International Conference on Fuzzy Computation (ICFC 2010)*. SciTePress, 43–55.

Truszczynski, M. 2018. An introduction to the stable and well-founded semantics of logic programs. In *Declarative Logic Programming: Theory, Systems, and Applications*, M. Kifer and Y. A. Liu, Eds. ACM / Morgan & Claypool, USA, 121–177.

Tu, P. H., Pontelli, E., Son, T. C., and To, S. T. 2009. Applications of parallel processing technologies in heuristic search planning: methodologies and experiments. *Concurrency and Computation: Practice and Experience 21,* 15, 1928–1960.

Ueda, K. 1986. Guarded Horn clauses. Ph.D. thesis, University of Tokyo.

Ullman, J. D. 2010. Cluster Computing and Datalog. In *Datalog 2.0: The Resurgence of Datalog in Academia and Industry*. `http://datalog20.org/`.

Urbani, J., Kotoulas, S., Maassen, J., van Harmelen, F., and Bal, H. 2012. WebPIE: A web-scale parallel inference engine using MapReduce. *Journal of Web Semantics 10*, 59–75.

Van Gelder, A., Ross, K. A., and Schlipf, J. S. 1991. The well-founded semantics for general logic programs. *Journal of the ACM 38,* 3, 620–650.

Van Roy, P. 1994. 1983-1993: The Wonder Years of Sequential Prolog Implementation. *Journal of Logic Programming 19/20,* 385–441.

Vidal, G. 2012. Annotation of logic programs for independent and-parallelism by partial evaluation. *Theory and Practice of Logic Programming 12,* 4-5, 583–600.

Vieira, R., Rocha, R., and Silva, F. M. A. 2012. On comparing alternative splitting strategies for Or-parallel Prolog execution on multicores. In *Colloquium on Implementation of Constraint and Logic Programming Systems*. 71–85.

Villaverde, K. and Pontelli, E. 2004. An investigation of scheduling in distributed constraint logic programming. In *Proceedings of the ISCA 17th International Conference on Parallel and Distributed Computing Systems*. ISCA, 98–103.

Villaverde, K., Pontelli, E., Guo, H.-F., and Gupta, G. 2001a. Incremental stack splitting mechanisms for efficient parallel implementation of search-based systems. In *International Conference on Parallel Processing*. IEEE Computer Society, USA, 287–294.

Villaverde, K., Pontelli, E., Guo, H.-F., and Gupta, G. 2001b. PALS: An Or-parallel implementation of Prolog on Bewoulf architectures. In *Procs. International Conference on Logic Programming*. Springer Verlag, Heidelberg, Germany, 27–42.

Villaverde, K., Pontelli, E., Guo, H.-F., and Gupta:, G. 2003. A methodology for order-sensitive execution of non-deterministic languages on Beowulf platforms. In *Euro-Par 2003. Parallel Processing, 9th International Euro-Par Conference*. Springer Verlag, Heidelberg, Germany, 694–703.

Wang, H., Xiong, F., Li, J., Shi, S., Li, J., and Gao, H. 2018. Data management on new processors: A survey. *Parallel Comput. 72*, 1–13.

Wang, J., Balazinska, M., and Halperin, D. 2015. Asynchronous and fault-tolerant recursive Datalog evaluation in shared-nothing engines. *Proceedings of the VLDB Endowment 8,* 12, 1542–1553.

Warren, D. H. D. 1990. The extended Andorra model with implicit control. In *Parallel Logic Programming Workshop*, Sverker Jansson, Ed. SICS, Box 1263, S-163 13 Spanga, SWEDEN.

Warren, D. S. 1984. Efficient Prolog memory management for flexible control strategies. In *International Symposium on Logic Programming*. IEEE Computer Society, USA, 198–203.

Warren, R., Hermenegildo, M. V., and Debray, S. K. 1988. On the practicality of global flow analysis of logic programs. In *Fifth International Conference and Symposium on Logic Programming*. MIT Press, Cambridge, MA, USA, 684–699.

White, T. 2015. *Hadoop - The Definitive Guide: Storage and Analysis at Internet Scale (4. ed., revised & updated)*. O'Reilly, USA.

Wielemaker, J., Schrijvers, T., Triska, M., and Lager, T. 2012. SWI-Prolog. *Theory and Practice of Logic Programming 12,* 1-2, 67–96.

Wolfson, O. and Silberschatz, A. 1988. Distributed processing of logic programs. In *Proceedings of the SIGMOD International Conference on Management of Data*, H. Boral and P. Larson, Eds. ACM Press, New York, 329–336.

Wu, H., Diamos, G. F., Sheard, T., Aref, M., Baxter, S., Garland, M., and Yalamanchili, S. 2014. Red Fox: An execution environment for relational query processing on GPUs. In *12th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2014, Orlando, FL, USA, February 15-19, 2014*, D. R. Kaeli and T. Moseley, Eds. ACM, New York, 44.

Xu, L., Hutter, F., Hoos, H. H., and Leyton-Brown, K. 2008. Satzilla: Portfolio-based algorithm selection for SAT. *J. Artif. Intell. Res. 32*, 565–606.

Yang, M., Shkapsky, A., and Zaniolo, C. 2015. Parallel bottom-up evaluation of logic programs: DeALS on shared-memory multicore machines. In *Proceedings of the Technical Communications of the 31st International Conference on Logic Programming (ICLP) 2015, Cork, Ireland*, M. D. Vos, T. Eiter, Y. Lierler, and F. Toni, Eds. CEUR Workshop Proceedings, vol. 1433. CEUR-WS.org, Aachen.

Zaffanella, E., Bagnara, R., and Hill, P. M. 1999. Widening sharing. In *Principles and Practice of Declarative Programming*, G. Nadathur, Ed. Lecture Notes in Computer Science, vol. 1702. Springer-Verlag, Heidelberg, Germany, 414–432.

Zaniolo, C., Yang, M., Das, A., Shkapsky, A., Condie, T., and Interlandi, M. 2017. Fixpoint semantics and optimization of recursive Datalog programs with aggregates. *Theory and Practice of Logic Programming 17,* 5–6, 1048–1065.

Zeuch, S. 2018. Query execution on modern CPUs. Ph.D. thesis, Humboldt University of Berlin, Germany.

Zhang, K. 1993. Exploiting Or-parallelism in logic programs: A review. *Future Generation Computing Systems 9,* 3, 259–280.

Zhang, W., Wang, K., and Chau, S.-C. 1995. Data partition and parallel evaluation of Datalog programs. *IEEE Transactions on Knowledge and Data Engineering 7*, 163–176.

Zhao, D., Subotic, P., and Scholz, B. 2020. Debugging large-scale Datalog: A scalable provenance evaluation strategy. *ACM Trans. Program. Lang. Syst. 42,* 2, 7:1–7:35.

Zhou, N., Sato, T., and Shen, Y. 2008. Linear tabling strategies and optimizations. *Theory and Practice of Logic Programming 8,* 1, 81–109.

Zhou, N.-F. 2012. The language features and architecture of B-Prolog. *Theory and Practice of Logic Programming 12,* 1 & 2, 189–218.

Zhou, N.-F. and Kjellerstrand, H. 2016. The Picat-SAT compiler. In *Practical Aspects of Declarative Languages - 18th International Symposium, PADL 2016, St. Petersburg, FL, USA, January 18-19, 2016. Proceedings*, M. Gavanelli and J. H. Reppy, Eds. Lecture Notes in Computer Science, vol. 9585. Springer, Heidelberg, Germany, 48–62.

Zhou, N.-F., Kjellerstrand, H., and Fruhman, J. 2015. *Constraint Solving and Planning with Picat*. Springer, Heidelberg, Germany.

Zinn, D., Wu, H., Wang, J., Aref, M., and Yalamanchili, S. 2016. General-purpose join algorithms for large graph triangle listing on heterogeneous systems. In *Proceedings of the 9th Annual*

Workshop on General Purpose Processing using Graphics Processing Unit, GPGPU@PPoPP 2016, Barcelona, Spain, March 12 - 16, 2016, D. R. Kaeli and J. Cavazos, Eds. ACM, New York, 12–21.