Issues in Parallel Execution of Non-monotononic Reasoning Systems

Marcello Balduccini

Department of Computer Science
Texas Tech University
Box 43104
Lubbock, TX 79409, USA

Enrico Pontelli, Omar Elkhatib, Hung Le

Department of Computer Science
New Mexico State University
Box 30001/CS
Las Cruces, NM 88003, USA

Abstract

We explore the major issues involved in the automatic exploitation of parallelism from the execution models of logic-based non-monotonic reasoning systems. We describe orthogonal techniques to parallelize the computation of models of non-monotonic logic theories, and demonstrate the effectiveness of the proposed techniques in prototypical implementation.

Key words: Logic Programming, Answer Set Programming, Non-monotonic Reasoning, Beowulf Clusters
1991 MSC: 68N17, 68T30, 68T25, 68T27, 68Q22

1 Introduction

It is becoming increasingly clear that, to fully realize the potential of computer revolution, computer scientists need to develop a *systematic methodology*

Email addresses: marcello.balduccini@ttu.edu (Marcello Balduccini), epontell@cs.nmsu.edu (Enrico Pontelli), okhatib@cs.nmsu.edu (Omar Elkhatib), hle@cs.nmsu.edu (Hung Le).

for the design and construction of advanced software systems. These systems should be capable of basing their behavior on knowledge about their environment and on the exchange of complex information and services with other systems as well as with humans. This realization led to recent advances in the development of programming paradigms centered around the notion of *knowledge*. These paradigms strive to reduce a substantial part of the programming process to the description of objects comprising the domain of interest and relations between these objects. The resulting description can be queried to establish truth or falsity of statements about the domain, or to find objects satisfying various properties. Knowledge can also be updated, modified, and used to make decisions on actions which need to be taken in a given situation to achieve certain goals.

Various languages, e.g., database query languages (e.g., SQL), Prolog, Frame-based, Rule-based, and Graph-based systems, support to a different extent this approach to programming. Typically, each of these languages is provided with a well-defined syntax and is equipped with an *inference engine*, capable of manipulating the knowledge encoded in the programs. Though different in style, all these languages are based on *classical logic*, developed to model mathematical reasoning. This severely limits the ability of these programming paradigms to represent common-sense knowledge and reasoning, which is crucial for development of sophisticated knowledge based systems [9] and intelligent agents [45].

In the last ten years we witnessed a rapid development of alternative logical systems, called *non-monotonic logics* [9,4,46]—which allow new axioms to retract existing theorems, and are more adequate for common-sense reasoning and for the modeling of dynamic and incomplete knowledge bases [8]. One of the outcomes of research in the field of non-monotonic logics is represented by the development of a number of languages for knowledge modeling and manipulation. In particular, in the last couple of years, a novel *programming paradigm* has arisen, called *Answer Set Programming (ASP)* [42,48,8], which builds on the mathematical foundations of *logic programming*, *constraint programming* and *non-monotonic reasoning*. ASP is a computation paradigm in which logical theories (composed of Horn clauses with negation as failure/default negation) serve as problem specifications, and solutions are represented by *collections of models* of such theories—also known as *stable models* or *answer sets* [42]. ASP originates from the extensive research on semantics of logic programming with negation [30,29] and it has evolved over the years in a complete programming paradigm, thanks to the availability of efficient inference engines—e.g., *Smodels* [49], DLV [23], DeRes [14], Cmodels [5], and ASSAT [40]—and expressive declarative languages—e.g., disjunctive logic programming [23], datalog with constraints [20], ordered logic programming [12], and probabilistic logic programming [17]. Domain-specific languages have also been proposed, along with sound and complete translation algorithms to map them to ASP (e.g., the languages $\mathcal{B}$, $\mathcal{C}$, and $\mathcal{K}$ for the specification of action theories for intelligent agents [31]).

In comparison to other non-monotonic logics, ASP is syntactically simpler and, at the same time, very expressive. The mathematical foundations of ASP have been extensively studied; in addition, there exist a large number of *building block* results about specifying and programming using ASP—e.g., results about dealing with incomplete information, reasoning with user-defined preferences, and abductive assimilation of new knowledge. ASP has offered novel and highly declarative solutions in a number of *well-defined application areas*, including intelligent agents [8], planning [38], software modeling & verification [35], complex systems diagnosis [6,28], semantic web services composition and monitoring [44], phylogenetic inference [26].

In spite of the continuous effort in developing fast execution models for ASP [23,20,49], execution of large programs remains a challenging task, limiting the scope of applicability of ASP in certain domains (e.g., planning). [1] In this work we propose the use of *parallelism* to improve performance of ASP engines and improve the scope of applicability of this paradigm. The core of our work is the identification of a number of potential sources for *implicit* exploitation of parallelism from a basic execution model for ASP programs—specifically the execution model proposed in the *Smodels* system [49]. We show that ASP has the potential to provide considerable amounts of loosely related tasks, which can be concurrently explored by different ASP engines. Exploitation of parallelism can be accomplished in a fashion similar to the models proposed to parallelize Prolog [34] and constraint propagation [47].

In this paper we overview the main issues in the exploitation of parallelism from the basic execution model of ASP. We identify two major forms of parallelism, *Horizontal* parallelism and *Vertical* parallelism, that respectively correspond to the two instances of non-determinism present in the propagation-based operational semantics commonly used for ASP. Building on recent theoretical results regarding the efficiency of parallel exploration of search trees [55,52], we investigate the development of techniques to handle the different forms of parallelism in an ASP engine and we present preliminary experimental results accomplished on shared memory and Beowulf platforms.

The work proposed in this paper—along with the work concurrently conducted by Finkel et al. [27]—represents the first exploration in the use of scalable architectures for the manipulation of non-monotonic logics ever proposed. The type of parallel engine designed in this work provides improved performance over systems like Smodels within the same application domains. In particular, we employ alternative parallelization strategies—*horizontal* and

[1] It is important to remember that computing stable models is in general a NP-hard problem [42].

vertical parallelism—to ensure a broader applicability to problems with different structure. We expect the proposed techniques to be of interest both to the parallel computing community—we explore and compare different dynamic work distribution and scheduling strategies, which could find applications to other domains (e.g., parallelization of other search-based computations)—and the answer set programming community.

The rest of this paper is organized as follows. Section 2 provides a brief introduction to ASP and to its theoretical foundations. Section 3 describes the execution model adopted for ASP in this work. Section 4 introduces the main forms of parallelism tackled in this work; Sections 5 and 6 discuss the two specific forms of parallelism (Vertical and Horizontal Parallelism) explored in this project. Section 7 analyzes the problem of speeding up the grounding phase and other optimizations. Finally, Section 8 presents conclusions and future directions of research.

2 Non-monotonic Reasoning and Answer Set Programming

2.1 Answer Set Semantics

Answer Set Semantics (AS) [30,29] (a.k.a. *Stable Models* semantics) was designed in the mid 80s as a tool to provide semantics for Logic Programming (LP) with *negation* [4]. A (*definite*) logic program [41] is composed of a collection of *rules* (a.k.a. *clauses*) of the form

$$Head : -A_1, \dots, A_n$$

where $Head, A_1, \dots, A_n$ are atomic logic statements (*atoms*). Each rule represents a logic implication—if $A_1, \dots, A_n$ are all true, then $Head$ will be inferred to be true as well. The semantics of a logic program is typically described using the notion of *minimal model*—which is the set of all the atoms which are *logical consequences* of the program. A known property is that, for a definite logic program, the set of all logical consequences is unique.

Example 1 *The following is a simple definite logic program*

```
p :- q, r.
q .
r :- s.
r :- t.
s.
```

Note that if a rule has an empty body, then we write only its head (e.g., the rule `q.`*). The minimal model of this program is the set* $\{q, s, r, p\}$*, which contains all and only the logical consequences of the rules in the program.*

Example 2 *Consider the program*

```
p(0).
p(1).
q(X) :- p(X).
```

The capital letters denote variables. *The program is a short-hand for the set of all rules that can be obtained by replacing the variables with any legal values in the logic language at hand. If we assume that the only two constants in the language are* 0 *and* 1*, then the above program represents the set of rules:*

```
p(0).
p(1).
q(0) :- p(0).
q(1) :- p(1).
```

Definite logic programs allow only atomic statements in the rules; as a result, each rule can only draw describe relationships between *positive* statements—i.e., we can derive the truth of an element based on the truth of other elements. This type of language does not allow one to derive negative consequences (i.e., we cannot express statements of the type *"if x is true then y must be false"*) and it does not allow one to build knowledge on the false status of certain statements (i.e., we cannot write statements of the type *"if x is false then y must be true"*).

However, in a large number of cases it is useful to reason also about negative consequences, by allowing negative knowledge to be inferred and allowing negative assumptions in the rules of the program (i.e., allow elements of the form *not A* in the body of the rules). The ability to use negated formulas in the rules leads to a natural support for non-monotonic reasoning, and the availability of efficient computational mechanisms provides a natural setting for the study of proof systems for non-monotonic reasoning [48].

The introduction of negation in logic programming leads to a number of complications, starting from the fact that negation may lead to the loss of one of the key properties of definite logic programming, the existence of a unique minimal model for each program. In fact, in definite logic programming (where negation is not used), there is no ambiguity in what is true and what is false w.r.t. the program. On the other hand, if negation is present, it is possible to have programs which have distinct sets of logical consequences. Two classes of proposals have been developed to tackle the problem of providing semantics to logic programs in presence of negation. The first class [3,63] attempts to reconstruct a single minimal model, either by narrowing the class of admissible programs (e.g., *stratified programs* [59] are programs meeting certain syntactic conditions, which guarantee the existence of a single minimal model) or by switching to 3-valued semantics—i.e., admitting the fact that formulae can be not only true or false, but also undefined. The second direction of research instead admits the existence of a *collection* of minimal models for a program [15,30,29]. Answer sets is the most representative approach in this second class. AS has been recognized to provide the right semantics for LP with negation—e.g., it naturally extends the the minimal model semantics of definite logic programs, it subsumes the intended model in the approach based on stratification [4], etc.

AS relies on a very simple definition. Given a program [2] P and given a tentative model M, we can define a new program P^M (the *reduct* of P w.r.t. M) which is obtained by

- removing all rules containing negative elements which are contradicted by the model M;
- removing all negative elements from the remaining rules.

Thus, P^M contains only those rules of P that are applicable given the model M. Furthermore, P^M is a definite logic program, i.e., it does not contain negations, which admits a unique intended model M' [41]. M is an *answer set* (a.k.a. *stable model*) if M and M' coincide. Intuitively, a stable model contains all and only those atoms which have a justification in terms of the applicable rules in the program. These models can be proved to be *minimal*, and in general a program with negation may admit more than one answer set.

Example 3 *If we have a database indicating people working in different departments*

```
dept(hartley,cs).        dept(pfeiffer,cs).
dept(gerke,math).        dept(prasad,ee).
```

and we would like to select the existing departments and one representative employee for each of them:

```
depts_employee(Name,Dep)  :-  dept(Name,Dep), not other_emps(Name,Dep).
other_emps(Name,Dep)      :-  dept(Name,Dep), dept(Name1,Dep),
                              depts_employees(Name1,Dep),
                              Name ≠ Name1.
```

[2] Let us assume for the sake of simplicity that it does not contain variables.

The rules assert that `Name/Dep` *should be added to the solution only if no other member of the same department has been selected. AS produces for this program 2 answer sets, corresponding to the solutions:*

$$\{\langle \texttt{hartley}, \texttt{cs}\rangle, \langle \texttt{gerke}, \texttt{math}\rangle, \langle \texttt{prasad}, \texttt{ee}\rangle\}$$
$$\{\langle \texttt{pfeiffer}, \texttt{cs}\rangle, \langle \texttt{gerke}, \texttt{math}\rangle, \langle \texttt{prasad}, \texttt{ee}\rangle\}$$

2.2 Answer Set Programming: a Novel Paradigm

As recognized by a number of authors [37,42,48], the adoption of AS requires a *paradigm shift* to reconcile the peculiar features of AS—i.e., the existence of multiple admissible models—with the traditional program view of logic programming. This need arises for a number of reasons. In the first place, under AS, each program potentially admits more than one intended model. This ends up creating an additional level of non-determinism—specifically a form of *don't know* non-determinism. The presence of multiple answer sets complicates the framework in two ways. First of all, we need to provide programmers with a way of handling the multiple answer sets. On one hand, one could attempt to restore a more "traditional" view, where a single "model" exists. This has been attempted, for example, using *skeptical semantics* [42], where an atom is considered entailed from the program only if it is true in each answer set. For certain classes of programs skeptical semantics coincides with other semantics proposed for LP with negation. Nevertheless, skeptical semantics is often inadequate—e.g., in many situations it does not provide the desired result (see example 3), and in its general form provides excessive expressive power [42,43]. The additional level of non-determinism—removed by skeptical semantics—is indeed a real need for a number of applications; it is also possible to see some similarities between this and some of the proposals put forward in other communities—such as the *choice* and *witness* constructs used in the database community [36,1,56].

The presence of multiple answer sets, in turn, leads to a new set of requirements on the *computational mechanisms* used. Given a program, now the main goal of the computation is not to provide a goal-directed tuple-at-a-time answer (i.e., a true/false answer or an answer substitution), as in traditional LP, but the goal is to return *whole answer sets*. The traditional resolution-based control used in LP is largely inadequate, and should give place to a different form of control and different execution mechanisms.

In this project we embrace a different view of LP under AS, interpreted as a *novel programming paradigm*—that we will refer to as *Answer Set Programming (ASP)*. This term was originally created by V. Lifschitz, and nicely blends the notion of programming with the idea that the entities produced by the computation are answer sets. The notion of ASP is not completely new and has been advocated by others during the last two years: Niemela has recently proposed answer set semantics as a *constraint programming paradigm* [48], while Marek and Truszczynski have coined the term *Stable Logic Programming* [42] to capture the notion we are describing. A comprehensive overview of ASP can be found in [8].

In simple terms, the goal of an ASP program is to identify a *collection of answer sets*—i.e., each program is interpreted as a specification of a collection of *sets of atoms*. Each rule in the program plays the role of a *constraint* [48] on the collection of sets specified by the program: a generic rule

$$\mathit{Head}\ :-\ B_1, \ldots, B_n, \mathit{not}\ G_1, \ldots, \mathit{not}\ G_m$$

requires that whenever $B_1, \ldots, B_n$ are part of the answer set and $G_1, \ldots, G_m$ are not, then *Head* has to be in the answer set as well. Thus, the collection of rules in a program constrain what sets of literals can be considered admissible models.

The shift of perspective from LP to ASP is very important. The programmer is led to think about writing programs as manipulating sets of elements, and the outcome of the computation is going to be a collection of sets—instead of an answer substitution, as in traditional logic programming. This perspective comes very natural in a large number of application domains (graph problems deal with set of nodes/edges, planning problems deal with sets of actions or states, etc.).

Example 4 *[38] The simple ASP program in Figure 1 computes the hamiltonian cycles of a graph. For the graph in the example, the program admits a single answer set:*

$$\{in(0,1), in(1,2), in(2,3), in(3,0)\}$$

In spite of these differences, ASP maintains many of the positive properties of traditional logic programming, including its declarative nature and the separation between logic and control—where the logic is given by the content of the program and the control is determined the mechanisms used to compute the answer sets.

```
%% Graph:
vertex(0). vertex(1). vertex(2). vertex(3).
edge(0,1). edge(0,2). edge(1,3). edge(1,2). edge(2,3). edge(3,0).
%% choice rules to select one edge
in(U,V) :- vertex(U), vertex(V), edge(U,V), not nin(U,V).
nin(U,V) :- vertex(U), vertex(V), edge(U,V), not in(U,V).
%% each node is traversed once
false :- vertex(U), vertex(V), vertex(W), V ≠ W, in(U,V), in(U,W).
false :- vertex(U), vertex(V), vertex(W), U ≠ V, in(U,W), in(V,W).

reachable(U) :- vertex(U), in(0,U).
reachable(V) :- vertex(U),vertex(V),reachable(U), in(U,V).
%% Guarantees that each vertex is reachable from 0
false :- vertex(U), U ≠ 0, not reachable(U).
```

Fig. 1. Program to Computer Hamiltonian Cycles of a Graph

2.3 Why ASP?

ASP has received great deal of attention within the knowledge representation and deductive database communities, as it allows for the representation of default assumptions, constraints, uncertainty and nondeterminism *in a direct way* [9]. The automation of non-monotonic reasoning may well rely upon automatic ASP, through the well-studied equivalences with, e.g., autoepistemic logic. ASP is related, both ideally and through formal equivalences, to the algorithmic study of satisfaction of boolean formulas (SAT). It is believed that ASP encoding of traditionally hard problems is more compact than SAT encoding. For instance, [42] argues that ASP encodings of the Hamiltonian cycle problem are *asymptotically* more concise than SAT ones. This implies that, other things being equal, ASP interpretations can be as efficient as satisfiability and even as constraint satisfiability systems. For example, [18] reports ASP solutions of planning problems in time comparable to ad-hoc planning algorithms. Finally, ASP syntax corresponds to $DATALOG^{\neg}$ of deductive databases, and should make database access transparent and straightforward.

3 Execution of ASP Programs

Computing with ASP is fairly different from computing in definite logic programs (e.g., Prolog)—in the latter we are interested in a single answer substitution computed w.r.t. a unique intended model, while in the former we are interested in computing sets of atoms representing different models of the logical theory. Designing an architecture for the computation of answer sets is not a straightforward task—indeed, the original definition of AS [30,29] is inherently non-constructive, as it requires guessing models and successively verifying whether they are answer sets or not. Nevertheless, in recent years a number of proposals have been made which provide approaches for computation of answer sets [11,13,14,49,23]. Chen and Warren [13] propose a method for computing answer sets which builds on their work on tabled evaluation of LP [54]. The method has the advantages of allowing a more relaxed program syntax and of being integrated in the context of an efficient Prolog system. On the other hand, the goal directed nature of this approach does not make it directly applicable as an engine for ASP [9]. Work is in progress by the XSB team to overcome this limitation. The three most efficient systems which support computation in the ASP paradigm are `dlv` [23], DeRes [14], and Smodels [49]. [3] These systems, which have been proposed very recently and are continuously developing, provide comparable efficiency and relatively similar features. Smodels relies on efficient and robust algorithms, and accepts as input language a (very powerful) superset of A-Prolog that includes *choice rules* [62]. DeRes is a system originally developed to deal with a larger class of programs than ASP—*default theories*—but capable of efficiently handling ASP programs. The DeRes group currently provide a version of DeRes (called *stable*) which is specialized for the computation of answer sets, highly suitable as computational engine for ASP. `dlv` supports a very general language (which includes *disjunctions* in the head of the logical rules) and it provides different application specific front-ends—e.g., a front-end for abductive diagnosis [22]. The development of different implementations capable of handling ASP programs is very important—as it indicates the existence of a community which needs the power of ASP as well as it demonstrates that efficient execution of ASP is not beyond our reach.

3.1 Sequential Execution Models

The sequential architecture we propose in this project is a new generation engine obtained from the original design of the *Smodels* system [49]. Smodels relies on efficient and robust algorithms for the execution of ASP programs. The basic structure is sketched in Fig. 2—the algorithm alternates choices and propagations, in the style of typical constraint programming solutions.

Figure 3 provides a schematic illustration of the overall proposed architecture. The components are described in this section.

[3] Recently new models have also been proposed based on the use of off-the-shelves SAT solvers [5,40].

```
function compute (Π : Program, A : LiteralsSet)
    B := expand(Π, A) ;
    while ( (B is consistent) and
              (B is not complete) )
         l := choose_literal(Π, B);
         B := expand(Π , A ∪ { l }) ;
    endwhile
    if (B stable model of Π) then
         return B;
```

Fig. 2. Basic Execution Model for ASP

Fig. 3. Sequential Architecture

3.1.1 Preprocessor

Many of the systems proposed so far rely on the use of a preprocessor to transform the input program to a format suitable for processing. The preprocessor we propose includes the following components:

- *Program Transformation:* this is a collection of different modules used to perform source-to-source code transformation. Source transformations are used, first of all, to support alternative input languages (e.g., the language $\mathcal{B}$ [9] used to describe action theories for software agents) which are mapped to pure logic programs with negation [9,10,50]. Another objective of program transformations is to determine code transformations which can improve efficiency of execution.
- *Grounder:* answer set semantics [30,29] relies on the manipulation of *ground programs*—i.e., programs which have been completely instantiated on a domain of interest and do not contain variables. During preprocessing it is necessary to *ground* the input program—identifying finite domains for each variable. Both Smodels and DeRes rely on the same software, called *lparse* [62], which provides sophisticated grounding procedures.

3.1.2 Engine:

The analogy between ASP and constraint programming, advocated by various researchers [42,48] has been used to a certain extent in the design of existing ASP engines. Computation of answer sets relies on propagation techniques—selecting an atom as being true or false constrains, via the program rules, a number of other atoms to a specific logical value. Most systems take advantage of this feature (e.g., the `expand` procedure in Fig. 4). The proposal in [11] even translates the problem of computing answer sets into a problem of solving a linear programming problem—which can be tackled directly via constraint programming techniques. The procedure `choose_literal` selects one literal (i.e., an atom or its negation) to add to the answer set, while the procedure `expand` determines which atoms have a determined value in the partial answer set A—i.e., `expand` identifies those atoms whose truth value is uniquely defined by the choices made up to that point. The actual algorithms used in many systems are refinements of this execution cycle.

The meaning of the partial answer set B is that, if atom a belongs to B, then a will belong to the final model. If *not* a belongs to B, a will *not* belong to the final model. Inconsistent interpretations are those containing contradictory atoms.

As ensues from Fig. 2, computation of answer sets is a highly non-deterministic and time-consuming activity. Non-determinism arises in different phases of this computation. The `expand` phase involves applying program rules in various ways (e.g., forward and backward chaining) to infer truth values of other literals. This process is seen as a fixpoint computation where, at each step, one rule is selected and used. Being the result of this phase deterministic, `expand` can be seen as an instance of *don't care non-determinism*. The fact that ASP programs may admit different answer sets implies that the `choose_literal` procedure is also non-deterministic; different choices will potentially lead to distinct answer sets. Thus, the process of selecting literals to add to the answer set represents a form of *don't know non-determinism*. This form of non-determinism has some resemblance to the non-determinism present in traditional LP (rule selection during one resolution step).

```
function expand (Π : Program, A : LiteralsSet)
    B := A ;
    while ( B ≠ B' ) do
        B' := B;
        B := apply_rule(Π, B);
    endwhile
    return B ;
```

Fig. 4. Expand procedure

Each non-deterministic computation can terminate either successfully—i.e., B assigns a truth value to all the atoms and it represents an answer set of Π—or unsuccessfully—if either the process tries to assigns two distinct truth values to the same atom or if B does not represent an answer set of the program (e.g., truth of certain selected atoms is not "supported" by the rules in the program). As in traditional logic programming, non-determinism is handled via backtracking to the choice points generated by `choose_literal`. Observe that each choice point produced by `choose_literal` has only two alternatives: one assigns the value true to the chosen literal, and one assigns the value false to it.

The `expand` procedure mentioned in the algorithm in Figure 2 is intuitively described in Figure 4. This procedure repeatedly applies expansion rules to the given set of literals until no more changes are possible. The expansion rules are derived from the program Π and allow to determine which literals have a definite truth value w.r.t. the existing partial answer set. This is accomplished by applying the rules of the program Π in different ways [49]. Efficient implementation of this procedure requires considerable care to avoid unnecessary steps, e.g., by dynamically removing invalid rules and by using smart heuristics in the `choose_literal` procedure [49], e.g.,

1. *forward rule:* if the rule $h \leftarrow a_1, \ldots, a_n, not\, b_1, \ldots, not\, b_m$ is in Π and all the elements in the body of the rule are true (i.e., they are in B), then also h can be assumed to be true in B
2. *nullary rule:* if there are no rules having the atom a as a head, then a can assumed to be false in B
3. *single positive rule:* if the atom h is true (i.e., it is in B) and there is a single rule

$$h \leftarrow a_1, \ldots, a_n, not\, b_1, \ldots, not\, b_m$$

   in Π having h as head, then all the elements of the body of the rule can be added to B (i.e., they have to be true as well)
4. *negative rules:* if the literals $not\ h, l_1, \ldots, l_{i-1}, l_{i+1}, \ldots, l_m$ are in B and the rule

$$h \leftarrow l_1, \ldots, l_m$$

   is in Π, then $\bar{l}_i$ can be added to B, where $\bar{l}$ indicates the complement of the literal l—i.e., if l is the positive atom a (negative literal $not\ a$) then $\bar{l}$ is the literal $not\ a$ (a).

Efficient implementation of this procedure requires considerable care to avoid unnecessary steps, e.g., by dynamically removing invalid rules and by using smart heuristics in the `choose_literal` procedure [60].

The execution model sketched in the previous section has been originally proposed in the context of the *Smodels* system, and adopted in a variety of other systems computing answer sets (e.g., some versions of the DLV system [27]). This execution model is at the core of the prototypes used for the investigation in parallel execution of ASP programs presented in this paper. The ideas presented in this work have been integrated in prototypes developed at New Mexico State University and Texas Tech University. These systems provide a number of additional features, including:

- support for various language extensions, such as *choice rules* [62] and *weak constraints* [21];
- support for *consistency restoring rules* [7];
- an object-oriented interface with Prolog, which allows Prolog programs to include ASP modules;
- support for various forms of aggregates [53].

4 Sources of Parallelism

As described in Section 3.1, the execution model for ASP is inherently non-deterministic, due to the different choices that have to be performed during the computation of each model. This inherent non-determinism suggests a natural direction for *automatic parallelization* of the computation of answer sets.

The structure of the computation of answer sets previously illustrated can be easily interpreted as an instance of a constraint-based computation [61], where the application of the expansion rules (`expand` procedure) represents the propagation step of the constraint computation, and the selection of a literal in `choose_literal` represents a labeling step. From this perspective, it is possible to identify two sources of non-determinism:

- *horizontal non-determinism:* which arises from the choice of a method for the expansion of the partial answer set (e.g., in `expand`);
- *vertical non-determinism:* which arises from the choice of the literal to add to the partial answer set (in `choose_literal`).

These two forms of non-determinism bear similarities respectively to the *don't care* and *don't know* non-determinism traditionally recognized in constraint and logic programming [34]. The goal of this project is to explore avenues for the exploitation of parallelism from these two sources of non-determinism—by exploring the different alternatives available in each point of non-determinism in parallel.

In particular, we will use the terms

- *Horizontal Parallelism* to indicate the use of separate threads of computation to concurrently apply different expansion techniques to a partial answer set.
- *Vertical Parallelism* to indicate a situation where separate threads of computation are employed to explore alternatives arising from vertical non-determinism;

Horizontal parallelism is aimed at the use of different computation agents to construct one of the models of the program—thus, the different agents *cooperate* in the construction of one model of the original program. Vertical Parallelism, on the other hand, makes use of separate computing agents for the computation of *different models* of the program—each execution thread is working on a different answer set of the program. In the rest of this paper we focus on the exploitation of Vertical Parallelism, and on a particular (and promising) form of Horizontal Parallelism, that we call *Parallel Lookahead*.

5 Vertical Parallelism

The alternative choices of literals during the derivation of answer sets (`choose_literal` in Fig. 2) are *independent* and can be concurrently explored, generating separate threads of computation, each potentially leading to a distinct answer set. We will refer to this form of parallelism as *Vertical Parallelism*. Thus, *vertical parallelism* parallelizes the computation of *different* answer sets.

5.1 Issues in Managing Vertical Parallelism

As ensues from research on parallelization of search tree applications and non-deterministic programming languages [55,2,16,34], the issue of designing the appropriate data structures to maintain the correct state in the different concurrent branches, is essential to achieve efficient parallel behavior. Observe that straightforward solutions to related problems have been formally proved to be ineffective, leading to unacceptable overheads [55].

The architecture for vertical parallel ASP that we envision is based on the use of a number of ASP engines (*agents*), which are concurrently exploring the search tree generated by the search for answer sets—specifically the search tree whose nodes are generated by the execution of the `choose_literal` procedure. Each agent explores a distinct branch of the tree; idle agents are allowed to acquire unexplored alternatives generated by other agents.

The major issue in the design of such architecture is to provide efficient mechanisms to support this sharing of unexplored alternatives between agents. Each node P of the tree is associated to a partial answer set $B(P)$—the partial answer set computed in the part of the branch preceding P. An agent acquiring an unexplored alternative from P needs to continue the execution by expanding $B(P)$ together with the literal selected by `choose_literal` in node P. Efficient computation of $B(P)$ for the different nodes in the tree is a known complex problem [55].

Since ASP computations can be very ill-balanced and irregular, we opt to adopt a dynamic scheduling scheme, where idle agents navigate through the system in search of available tasks. Thus, the partitioning of the available tasks between agents is performed dynamically and is initiated by the idle agents. This justifies the choice of a design where different agents are capable of traversing a shared representation of the search tree to detect and acquire unexplored alternatives. In addition, this view allows one to reuse the *optimization schemes* developed for other parallel execution models to improve efficiency of these mechanisms, via run-time transformations of the search tree [33]—e.g., flattening the tree to facilitate work sharing.

5.1.1 Basic Structure of the Parallel Engine

As mentioned earlier, the system is organized as a collection of agents, which are cooperating in computing the answer sets of a program. Each agent is a separate ASP engine, which owns a set of *private* data structures employed for the computation of answer sets. Additionally, a number of *global* data structures, i.e., accessible by all the agents, are introduced to support cooperation between agents. This structuring of the system implies that we rely on a shared-memory architecture.

The different agents share a common representation of the ASP program to be executed. This representation is stored in one of the global data structures. Program representation has been implemented following the general data structure originally proposed in [19]—proved to guarantee very efficient computation of models for definite logic programs. This representation is summarized in Figure 5. Each rule is represented by a descriptor; all rule descriptors are collected in a single array, which allows for fast scan of the set of rules. Each rule descriptor contains, between the other things, pointers to the descriptors for all atoms which appear in the rule—the head atom, the atoms which appear positive in the body of the rule, and the atoms which appear negated in the body of the rule.

Each atom descriptor contains information such as

- an array containing pointers to the rules in which the atom appears as head
- an array containing pointers to the rules in which the atom appears as positive body element
- an array containing pointers to the rules in which the atom appears as negative body element
- an *atom array* index

Differently from the schemes adopted in sequential ASP engines [19,49], our atom descriptors *do not* contain the truth value of the atom. Truth values of atoms are instead stored in a separate data structure, called *atom array*. Each agent maintains a separate atom array, as shown in Figure 5; this allows each agent to have an independent view of the current (partial) answer set constructed, allowing atoms to have different truth values in different agents. E.g., in Figure 5, the atom of index i is true in the answer set of one agent, and false in the answer set computed by another agent.

Each agent essentially acts as a separate ASP engine. Each agent maintains a local stack structure (the *trail*), which keeps track of the atoms whose truth value has already been determined. Each time the truth value of an atom is determined (i.e., the appropriate entry in the atom array is set to store the atom's truth value), a pointer to the atom's descriptor is pushed in the trail stack. The trail stack is used for two purposes:

- (during `expand`) the agent uses the elements newly placed on the trail to determine which program rules may be triggered for execution;
- a simple test on the current size of the trail stack allows each agent to determine whether all atoms have been assigned a truth value or not.

The use of a trail structure provides also convenient support for exploitation of horizontal parallelism [24].

To support the exploitation of vertical parallelism, we have also introduced an additional simple data structure in each agent: a *choice point* stack (or *core stack*). The elements of the choice point stack are pointers to the trail stack. These pointers are used to identify those atoms whose truth value has been "guessed" by the `choose_literal` function. The choice points are used during backtracking: they are used to determine which atoms should be removed from the answer set during backtracking, as well as which alternatives can be explored to compute other answer sets. This is akin to the mechanisms used to support backtracking in trail-based constraint systems [57,58].

Fig. 5. Representation of Rules and Atoms

The open issue which remains to be discussed is *how* agents *interact* in order to exchange unexplored alternatives—i.e., how agents *share* work. Each idle agent attempts to obtain unexplored alternatives from other active agents. In our context, an *unexplored alternative* is represented by a partial answer set together with a new literal to be added to it.

In this project we have explored two alternative approaches to tackle this problem:

- *Recomputation-based Work Sharing:* agents share work just by exchanging the list of *chosen* literals which had been used in the construction of an answer set; the receiving agent will use these to *reconstruct* the answer set and then perform local backtracking to explore a new alternative.
- *Copy-based Work Sharing:* agents share work by exchanging a complete copy of the current answer set (both *chosen* as well as *determined* literals) and then performing local backtracking.

In both schemes, the sharing of work is a local transaction between the two agents involved. Sharing of work is initiated by the idle agent. The interaction is realized by *copying* data from the (private) area belonging to one agent to the area of another agent. Note that this is feasible since even the data structures that belong to a specific agent (e.g., its choice point stack) are stored in shared memory.

The two schemes provide a different balance between amount of data copied from one agent to the other and amount of time needed to restart the computation with a new alternative in a different agent. These two methods are discussed in detail in the next sections. Although many alternative methods have been discussed in the literature to handle parallel execution of search-based applications (e.g., see [32] for a survey), we have focused on these two models for the following reasons:

- these two methodologies have been theoretically demonstrated to be *optimal* with respect to a reasonable abstraction of the problem of supporting concurrent search;
- our intention is to target exploitation of vertical parallelism across a wide range of parallel platforms, including distributed memory platforms. It has been proven that these two methodologies are the most effective in absence of shared memory.

Another important aspect that has to be considered in dealing with this sort of systems is termination detection. The overall computation needs to determine when a global fixpoint has been reached—i.e., all the answer sets have been produced and no agent is performing active computation any longer. In the system proposed we have adopted a centralized termination detection algorithm. One of the agents plays the role of controller and at given intervals polls the other agents to verify global termination. Details of this algorithm are omitted for lack of space.

5.1.2 Model Recomputation

The idea of recomputation-based sharing of work is derived by similar schemas adopted in the context of *or-parallel* execution of Prolog [34]. In the recomputation-based scheme, an idle agent obtains a partial answer set from another agent in an *implicit* fashion. Let us assume that agent $\mathcal{P}$ wants to send its partial answer set B to agent $\mathcal{Q}$. To avoid copying the whole partial answer set B, the agents exchange only a list containing the literals which have been chosen by $\mathcal{P}$ during the construction of B. These literals represent the "core" of the partial answer set. In particular, we are guaranteed that an `expand` operation applied to this list of literals will correctly produce the whole partial answer set B. This communication process is illustrated in Fig. 6. The core of the current answer set is represented by the set of literals which are pointed to by the choice points in the core stack (see Fig. 6). In particular, to make the process of sharing work more efficient, we have modified the core stack so that each choice point not only points to the trail, but also contains the corresponding chosen literal (the literal it is pointing to in the trail stack). As a result, when sharing of work takes place between agent $\mathcal{P}$ and agent $\mathcal{Q}$, the only required activity is to transfer the content of the core stack from $\mathcal{P}$ to $\mathcal{Q}$. Once $\mathcal{Q}$ receives the chosen literals, it will proceed to install their truth values (by recording the literals' truth values in the Atom Array) and perform an `expand` operation to reconstruct (on the trail stack) the partial answer set. The last chosen literal will be automatically complemented to obtain the effect of backtracking and constructing the "next" answer set. This copying process can be also made more efficient by making it *incremental*: agents exchange only the *difference* between the content of their core stacks. This reduces the amount of data exchanged and allows to reuse part of the partial answer set already existing in the idle agent.

Fig. 6. Recomputation Sharing of Work

5.1.3 Model Copying

The copying-based approach to work sharing adopts a simpler approach than recomputation. Upon work sharing from agent $\mathcal{P}$ to $\mathcal{Q}$, the entire partial answer set existing in $\mathcal{P}$ is directly copied to agent $\mathcal{Q}$. The use of copying has been frequently adopted to support computation in constraint programming systems [58] as well as to support or-parallel execution of logic and constraint programs [34]. The partial answer set owned by $\mathcal{P}$ has an explicit representation within the agent $\mathcal{P}$: it is completely described by the content of the trail stack. Thus, copying the partial answer set from $\mathcal{P}$ to $\mathcal{Q}$ can be simply reduced to the copying of the trail stack of $\mathcal{P}$ to $\mathcal{Q}$. This is illustrated in Figure 7. Once this copying has been completed, $\mathcal{Q}$ needs to install the truth value of the atoms in the partial answer set—i.e., store the correct truth values in the atom array. Computation of the "next" answer set is obtained by identifying the most recently added literal whose value has been "guessed" and performing local backtracking to it. The identification of the backtracking literal is immediate as this literal lies always at the top of copied trail stack. As in the recomputation case, we can improve performance by performing incremental copying, i.e., by copying not the complete answer set but only the difference between the answer set in $\mathcal{P}$ and the one in $\mathcal{Q}$.

5.1.4 Hybrid Sharing Schemes

The experiments performed on shared memory architectures (described in the next Section) have indicated that Model Copying behaves better than Model Recomputation in most of the cases. This is due to the high cost of recomputing parts of the answer set w.r.t. the cost of simply performing a memory copying operation.

Fig. 7. Copy-based Sharing of Work

To capture the best of both worlds, we have switched in our prototype to an hybrid work sharing scheme, where both Model Recomputation and Model Copying are employed. The choice of which method to use is performed dynamically (*each time* a sharing operation is required). Various heuristics have been considered for this selection, which take into account the size of the core and the size of the partial answer set. Some typical observations that have been made from our experiments include: *(i)* if the size of the core is sufficiently close to the size of the answer set, then recomputation would lead to a loss w.r.t. copying. *(ii)* if the size of the answer set is very large compared to the size of the core, then copying appears still to be more advantageous than recomputation. This last property is strongly related to the speed of the underlying interconnection network—the slower the interconnection network, the larger is the partial answer set that one can effectively recompute. We have concretized these observations by experimentally identifying two thresholds (*low* and *high*) and a function f which relates the size of the core and the size of the answer set; Recomputation is employed whenever $low \leq f(\ \mathit{sizeof}(Core), \mathit{sizeof}(Partial\ Answer\ Set)\ ) \leq high$. The same considerations are even more significant in the context of execution of ASP on distributed memory platforms: in this context the cost of copying is higher (due to the higher cost of moving data across the interconnection network) and the threshold in favor of recomputation is wider.

5.2 Experimental Results

5.2.1 Model Recomputation on Shared Memory Platforms

In this section we present performance results for a preliminary prototype which implements an ASP engine with Recomputation-based vertical parallelism. The prototype is capable of computing the answer sets of standard ASP programs, pre-processed by the lparse grounding program [62].

All performance figures presented are in milliseconds and have been achieved as average execution times over 10 consecutive runs on a lightly loaded machine. The benchmarks adopted are programs obtained from various sources (all written by other researchers); they include some large scheduling applications (sjss, rcps), planners (logistics 1,2, strategic), graph problems (color), as well various synthetic benchmarks (T4, T5, T15, T8, P7). These benchmarks range in size from few tens of rules (e.g., T4, T5) to hundreds of rules (e.g., rcps).

Name	Smodels	1 Agent	2 Agents	3 Agents	4 Agents	8 Agents	10 Agents
Scheduling (sjss)	130997.22	141710.67	71157.78	47891.41	38362.39	21182.46	17870.2
Scheduling (rcps)	71553.27	77021.983	38549.54	30686.05	30918.86	14236.97	11632.76
Color (Random)	1100323.37	1232359.24	616389.19	400608.21	309117.38	183933.32	163225.41
Color (Ladder)	973.21	993.77	624.31	497.01	332.2	305.11	314.67
Logistics (1)	8679.98	8918.5	9001.78	8634	4032.78	3995.67	3235.82
Logistics (2)	5681.26	6020.33	3241.01	2723.14	2278.3	1387.12	1270
Strategic	12488.88	12573.32	6672.39	4677.679	3654.12	2080.89	1817.64
T5	111.60	127.31	62.57	70.97	71.11	66.59	76.62
T4	104.02	103.01	71.24	68.33	74.18	88.79	108.21
T8	3070.51	3205.79	1669.11	1034.26	899.21	701.03	705.52
P7	2881.08	3260.01	1668.37	1252.9	871.04	415.13	425.14
T15	398.88	446.73	245.4	198.03	132.1	145	117.11
T23	3722.15	3741	1790.76	1685.38	1523.86	1411.79	1410.7

Table 1
Recomputation-based Sharing: Execution Times (msec.)

As can be seen from the figures in Table 1, the system is capable of producing good speedups for most of the selected benchmarks. On the scheduling (sjss, rcps), graph coloring, and planning (strategic, logistics) benchmarks the speedups are very high (mostly between 6 and 8 using 10 agents). This is quite a remarkable result, considering that these benchmarks are very large and some produce highly unbalanced computation trees, with tasks having very different sizes. The apparently low speedup observed on the logistics with the first plan (logistics 1), is actually still a positive result, since the number of choices performed across the computation is just 4 (thus we cannot expect a speedup higher than 4). On the very fine-grained benchmarks T4 and T5 the system does not behave as well; in particular we can observe a degradation of speedup for a large number of agents—in this case the increased number of interactions between agents overcome the advantages of parallelization, as the different agents attempt to exchange very small tasks. In T4 we even observe a slow-down when using more than 8 agents. Two slightly disappointing results are in T8 and P7. T8 is a benchmark which produces a very large number of average-to-small size tasks; the top speedup is below 5 and denotes some difficulty in maintaining good efficiency in presence of frequent task switching. P7 on the other hand has a very low number of task switching, but generates extremely large answer sets. The speedup tends to decrease with large number of agents because some agents end up obtaining choice points created very late in the computation, and thus waste considerable time in rebuilding large answer sets during the recomputation phase. The speedups obtained for all these benchmarks are plotted in Figure 8.

Note that the sequential overhead observed in all cases (the ratio between the sequential engine and the parallel engine running on a single processor) is extremely low, i.e., within 5% to 15% for most of the benchmarks.

Fig. 8. Speedups using Recomputation

5.2.2 Model Copying on Shared Memory Platforms

We modified our implementation to support Copy-based work sharing, and tested its performance on the same pool of benchmarks. Also in this case, the sequential overhead is very low (on average it is between 5% and 15%).

Name	**Smodels**	**1 Agent**	**2 Agents**	**3 Agents**	**4 Agents**	**8 Agents**	**10 Agents**
Scheduling (sjss)	130997.22	141710.67	67607.98	46374.28	33888.71	18650.48	15577.87
Scheduling (rcps)	71553.27	77021.99	38500.06	23692.36	19250.04	10349.47	7243.66
Color (Random)	1100323.37	1232362.54	606857.17	360152.65	286253.38	166034.76	136679.54
Color (Ladder)	973.21	987.71	539.51	366.95	308.05	262.65	261.27
Logistics (1)	8679.98	8920.51	3907.65	3727.05	3711.46	2416.99	2359.14
Logistics (2)	5681.26	6018.13	3051.41	2587.08	2017.03	1135.55	1066.35
Strategic	12488.88	12576.32	6693.23	4640.96	3252.46	1854.47	1686.73
T5	111.60	82.12	42.22	43.41	43.6	47.99	50.57
T4	104.02	91.56	58.6	51.53	62.47	62.08	62.94
T8	3070.51	3200.79	1587	1076.12	890.46	433.43	358.89
P7	2881.08	3260.01	1905.23	1435.22	1111.92	573.58	513.06
T15	398.88	440.97	223.08	148.72	126.73	128.94	131.21
T23	3722.15	2622	1357.59	990.23	748.61	391.72	451.69

Table 2
Copy-based Sharing: Execution Times (msec.)

The results reported in Table 2 (and the corresponding speedups plotted in Figure 9) are remarkable. The large benchmarks (e.g., the two scheduling applications) report speedups in the range $8.5-10$ for 10 agents, maintaining linear speedups for small number of agents (from 2 to 5 agents).

The fine grained benchmarks (such as `T4` and `T5`) provide speedups similar (usually slightly better) to those observed earlier. In both cases we note a slight degradation of speedup for large number of agents. As in the case of recomputation, this indicates that if the tasks are too fine grained, additional steps are needed in order to achieve performance improvements. We have experimented with a simple optimization, which semi-automatically unfolds selected predicates a constant number of times, in order to create larger grain tasks (by effectively combining together consecutive tasks). The simple optimization has produced improvements, as shown in Table 3.

Name	1 Agent	2 Agents	3 Agents	4 Agents	8 Agents	10 Agents
T5	1.0	1.94/1.99	1.89/1.99	1.88/1.97	1.71/1.95	1.62/1.93
T4	1.0	1.56/1.92	1.77/1.93	1.46/1.95	1.47/1.93	1.45/1.91

Table 3
Speedup Improvement using Task-collapsing Optimization (before/after)

The Copy-based scheme behaves quite well in presence of a large number of average-to-small tasks, as seen in the T8 benchmark. The speedups reported in this case are excellent. This is partly due to the lower cost, in this particular case, of copying w.r.t. recomputation, as well as the adoption of a smarter scheduling strategy, made possible by the use of copying, as discussed in the next section.

For what concerns the benchmark P7, the situation is sub-optimal. In this case the need of copying large answer sets during sharing operations penalizes the overall performance. We expect this case to become less of a problem with the introduction of *incremental copying* techniques—i.e., instead of copying the whole answer set, the agents compute the actual difference between the answer sets currently present in their stacks, and transfer only such difference. Our current prototype does not include this optimization.

Fig. 9. Speedups using Copying

5.2.3 Performance on Distributed Memory Platforms

The engine used in the previous experiment has been converted to support execution on distributed platforms. The main modifications performed deal with the fact that, on a Beowulf, agents cannot exchange data by copying data structures (e.g., from the choice point stack of one agent to the choice point stack of another agent). These operations have been converted to explicit message passing (based on MPI). The results have been obtained on a Pentium-based Beowulf cluster (purely distributed memory architectures). The experiments have been performed by executing a number of ASP programs (mostly obtained from other researchers).

All timings presented have been obtained as average over 10 runs. For the distributed memory prototype we have decided to consider the Hybrid Method of task exchange (see Sect. 5.1.4), rather than implementing Model Copying and Model Recomputation separately (like we did for shared memory platforms). The reason for this choice is the higher cost of communication between processors in distributed memory architectures. This factor, in fact, leads to a higher number of situations where Model Copying provides sub-optimal performances. On the other hand, the shift of balance is not large enough to justify a static application of Model Recomputation.

Table 4 reports the execution times observed on a set of benchmarks, while Fig. 10 illustrates the speedups observed using the hybrid scheme on a set of ASP benchmarks. Some of the benchmarks, e.g., `T8` and `P7`, are synthetic benchmarks developed to study specific properties of the inference engine, while others are ASP programs obtained from other researchers. `Color` is a graph coloring problem, `Logistics` and `Strategic` are scheduling problems, while `sjss` is a planner. Note also that `sjss` is executed searching for a single model while all others are executed requiring all models to be produced. The tests marked `[*]` in Fig. 10 indicate those cases where Recomputation instead of Copying has been triggered most of the times. The results presented have been obtained by using an experimentally determined threshold to discriminate between copying and recomputation. The rule adopted in the implementation can be summarized as: if

$$min \leq \frac{size(Partial\ Answer\ Set)}{size(Core)} \leq max$$

then model recomputation is applied, otherwise model copying is used. The intuition is that *(i)* if the ratio is too low, then, there is no advantage in copying just the core, while *(ii)* if the ratio is too high, then the cost of recomputing the answer set is likely to be excessive. The *min* and *max* used for these experiments where set to 1.75 and 12.5. Fig. 11 shows the impact of using recomputation in the benchmarks marked with `[*]` in Fig. 10. Some benchmarks have shown rather low speedups—e.g., `Color` on a ladder graph and `Logistics`. The first generates very fine grained tasks and suffers the penalty of the cost of communication between processors—the same benchmarks on a shared-memory platform produces speedups close to 4. For what concerns `Logistics`, the results are, after all, quite positive, as the maximum speedup possible is actually 5 and there seem to be no degradation of performance when the number of agents is increased beyond 5.

It is interesting to compare the behavior of the distributed memory implementation with that of the shared memory engine presented in the previous subsection. Fig. 12 presents a comparison between the speedups observed on selected benchmarks in the shared memory and the distributed memory engines. In the majority of the cases we observed relatively small degradation in the speedup. Only benchmarks where frequent scheduling of small size tasks is

Name	1 Agent	2 Agents	3 Agents	4 Agents	8 Agents
Color (Ladder)	311434	225465	212393	264278	266523
Color (Random2)	2126786	1195970	879276	621776	319453
Logistics 2	3934631	2170681	1841471	1651771	1040842
Strategic	69348	36553	25777	19714	11447
sjss	99869303	50028294	37178170	24567899	14226396
T8	1766751	863535	588916	443887	226499
P7	1781537	946618	712330	553272	222733

Table 4
Execution Times (in μs.) on Beowulf

Fig. 10. Speedups from Vertical Parallelism

Fig. 11. Impact of using Recomputation

required lead to a more relevant difference (e.g., `Color` for the ladder graph).

Fig. 12. Comparison of Shared and Distributed Memory Engines

In the context of our system, two scheduling decisions have to be taken by each idle processor in search of work:

1. select from which agent work will be taken;
2. select which unexplored alternative will be taken from the selected agent.

In the current prototype, we have tackled the first issue simply by maintaining a *work-load* count (i.e., number of local unexplored alternatives) for each agent and attempting to take work from the agent with the highest work-load. This simple scheme has proved to work well in practice.

The second decision turned out to be more complicated and has a deeper impact on the performance of the system. Our experimental results have indicated that the choice of which unexplored alternative to take work from (i.e., which choice point to steal from another agent) may lead to substantial variations in parallel performance.

In our experiments we have considered two approaches to this problem. In the first approach, agents are forced to steal the first choice point (i.e., the oldest choice point) from another agent (we call this approach *Top scheduling*). This technique was expected to perform well since:

- detecting the first choice point is a fast operation;
- selecting the first choice point reduces the size of the partial answer set transfered between agents;
- if the computation tree is balanced, then by taking the first choice point we should minimize the frequency of sharing operations.

The alternative technique considered is the dual of the one described above: at each sharing operation the last choice point created is taken (we call this approach *Bottom scheduling*). This approach is expected to have the following advantage: with simple modifications to the backtracking scheme, it allows to share at once not just a single choice point but a collection of them—e.g., all the choice points owned by an agent. On the other hand, the cost of sharing work under this scheme is considerable higher, since larger answer sets have to be exchanged.

The implementation of the first method is relatively simple; the first choice point is easily detected (by keeping an additional register in each agent for this purpose). The choice point indicates the segment of trail that has to be transferred to the other agent.

The second method has been realized as follows:

- the last choice point is easily detected as it lies on the top of the choice point stack; this allows to determine immediately what is the part of the trail that has to be copied;
- to allow sharing of multiple choice points at once, we push on the choice point stack a special choice point, which simply represents a link to a choice point lying in another processor's stack. This allows the backtracking activity to seamlessly flow between choice points belonging to different agents. (This technique resembles a similar methodology used for *public backtracking* in and-parallel logic programming systems [51]).

We have implemented both schemes and compared them on the selected pool of benchmarks. Figure 13 compares the speedups achieved using the two scheduling schemes in the Copy-based sharing scheme. The results clearly indicate that Bottom scheduling is superior in the large majority of the cases. Particularly significant are the differences in the sjss and the graph coloring problems. These are all programs where a large number of choice points are created; the bottom scheduling scheme allows to share in a single sharing operation a large number of alternatives, thus reducing the number of scheduling interactions between agents. The Top scheduling scheme provides better performance in those benchmarks where either there are few choices (e.g., T15) or the choices tend to be located always towards the beginning of the trail stack (T8).

Also in this case we can clearly identify a preferable scheme (the Bottom scheduling scheme); nevertheless a mixed approach which selects alternative scheduling policies depending on the structure of the program or the structure of the current answer set is likely to provide superior performance.

Fig. 13. Scheduling: Top vs. Bottom Scheduling

5.4 Optimizing Vertical Parallelism

We have also explored the performance of the distributed engine on a number of other benchmarks. The preliminary results obtained on this second batch of benchmarks were rather disappointing; indeed, on a number of sufficiently large grain computations, we observed severe slow-downs when increasing the number of agents employed. The problem was pinpointed to derive from the large size of the models generated by these benchmarks. During the sharing operations, each idle agent has to undo the existing computation (via backtracking), receive a complete copy of the trail, and install the new entries. As pointed out earlier, the current prototype does not include the notion of incremental copying, which may improve this sort of situations. Instead of building the complete infrastructure for incremental copying—which is potentially quite complex—we have tried to simply optimize the task of abandoning the current computation. Instead of blindly proceeding in a complete backtracking phase, the idle processor performs a test on the current size of the partial model located in its trail stack. If the size is above an experimentally determined threshold, then complete backtracking is replaced by a brute-force memory zeroing operation (using Unix's `memset`) to wipe out the content of the atom array. Experimental results have shown that if the trail's content is very large, this operation is considerably faster.

Figure 14 compares the speedup curves achieved with and without this optimization. While for the benchmarks in the left diagram the improvements are relatively small, the benchmarks on the right indicate that the impact of this optimization can be very high. Both benchmarks (`Color 6` is the coloring of another large ladder graph, while `rpcs 4` is a different version of the `rpcs` scheduling program) lead to a slowdown using a large number of agents. The use of the optimization allows the benchmarks to produce acceptable (for the `Color 6` program) or really good (for the `rpcs 4` benchmark) speedups. It is important to observe that the cost of the optimization is negligible (a simple test), compared to the cost of performing a full-blown incremental copying (e.g., cost of determining the part of the trail in common between two interacting agents).

Fig. 14. Speedup Curves with and without Memory Zeroing Optimization

6 Horizontal Parallelism

An orthogonal direction for parallel ASP can be achieved by parallelizing the steps occurring along *one* branch of the search tree. This implies, as in Fig. 2, the parallel evaluation of the individual executions of `expand`. The procedure `expand` determines the literals whose truth value is immediately determined by the partial answer set B. This is achieved by applying in various ways the program rules (e.g., forward and backward chaining), to expand the partial answer set, without performing any choice. Each rule can provide a different contribution, depending on the partial answer set. *Horizontal parallelism* can be achieved by allowing concurrent application of different rules to expand the partial answer set.

6.1 Static Horizontal Parallelism

The most direct approach in the exploitation of horizontal parallelism arises from the parallelization of the operations present in the `expand` procedure. As illustrated in Section 3.1, the `expand` operation consists of a fixpoint computation aimed at expanding the partial answer set by applying in different ways the rules present in the program.

Static parallelization of this process is obtained by partitioning the set of program rules between the different processors, so that each processor is in charge of applying a given set of rules (*program fragment*) to the partial answer set.

Two major issues have to be considered when developing an horizontal parallel ASP solution:

(1) *partitioning scheme:* the partitioning scheme is the policy used to distribute the program rules between the set of available processors.
(2) *interaction policy:* the interaction policy determines the frequency and pattern of interaction between the processors cooperating in the expansion of a partial answer set.

In our preliminary experiments we have adopted the following policies:

(1) *partitioning scheme:* the current policy is derived from the work on parallel constraint propagation [47] and assigns to each processor a collection of *procedures*—where a procedure is a collection of all the rules with the same predicate in the head. The partitioning is *static* and it is not modified during the execution of the program. The collection of procedures

assigned to the processors are determined according to two heuristics:

- processors should receive fragments of the same size—where the size is given by the total number of atoms in the fragment;
- procedures which are "close" in the dependency graph of the ASP program are assigned to the same processor. In particular, the current heuristics tries to keep elements belonging to a strongly connected component of the dependence graph within the same and-agent.

(2) *interaction policy:* each and-agent maintains a local stack where it maintains the part of the partial answer set that has been determined exclusively using the locally available program rules. Each time a local fixpoint is determined, the content of the local stack is transferred to a global representation (in shared memory) of the partial answer set.

We have initiated the development of an horizontal parallel ASP engine constructed according to the previously described policies. The prototype has been tested on a collection of automatically generated synthetic benchmarks (`Synth1` through `Synth4`). The benchmarks are composed of 50,000 program rules, each having a random number of body elements (between 0 and 4). The results are presented in Table 5.

Benchmark	**Number of Agents**			
	1	**2**	**3**	**4**
`Synth1`	393662	37945	17720	2048
`Synth2`	811335	89353	2337	1852
`Synth3`	2565765	132473	61480	3426
`Synth4`	64386763	260800	211890	45162

Table 5
Execution Times (in μsec)

The current prototype has the following properties:

- the indicated benchmarks show super-linear speedups; this arises from the fact that all the synthetic benchmarks considered do not have models—the parallel execution allows to detect inconsistencies faster.
- it provides sub-optimal sequential performance, due to excessive locking in the access of the shared representation of the partial answer set.

Work is in progress to attempt to reduce communication costs and sustain acceptable speedups on regular (non-contradictory) benchmarks.

6.2 Lookahead Parallelism

The (sequential) *Smodels* algorithm presented earlier builds the stable models of an answer set program incrementally. The algorithm presented in Fig. 2 can be refined to introduce the use of lookahead during the "guess" of a literal. The algorithm is modified as follows: *(1)* Before guessing a literal to continue expansion, unexplored literals are tested to verify whether there is a literal l such that $\mathsf{expand}(\Pi, B \cup \{l\})$ is consistent and $\mathsf{expand}(\Pi, B \cup \{\text{not } l\})$ is inconsistent. Such literals can be immediately added to B. *(2)* After such literals have been found, choose_literal can proceed by guessing an arbitrary unexplored literal. Step 1 is called the `lookahead` step. It is important to observe that any introduction of literals performed in this step is *deterministic* and does not require the creation of a choice point. In addition, the work performed while testing for the various unexplored literals can be used to choose the "best" literal to be used in step 2, according to some heuristic function.

During the lookahead step, every test performed on a pair $\langle l, not\ \ l\rangle$ is substantially independent from the tests run on any other pair $\langle l', not\ \ l'\rangle$. Each test involves up to two calls to expand (one for l, the other one for $not\ \ l$), which results in a comparatively expensive computation. These characteristics make the lookahead step a natural point where the algorithm could be parallelized. Notice that *Parallel Lookahead* is an instance of the general concept of Horizontal Parallelism, since the results of the parallel execution of lookahead are combined, rather than being considered alternative to each other, as in Vertical Parallelism. The appeal of exploiting Horizontal Parallelism at the level of `lookahead`, rather than at the level of `expand`, lies in the fact that the first involves a coarser-grained type of parallelism.

In this section, we describe our approach to Parallel Lookahead. In order to distinguish the prototype presented earlier with the one implementing Parallel Lookahead, we will call the latter `ParLook`.

6.2.1 Basic Design

The parallelization of the lookahead step is obtained in a quite straightforward way by splitting the set of unexplored literals, and assigning each subset to a different agent. Each agent then performs the test described in step 1 on the unexplored literals that it has been assigned. Finally, a new partial answer set, B' is built by merging the results generated by the agents.

Notice that, even in the parallel implementation, the lookahead step can be exploited in order to determine the best literal to be used in `choose_literal` (provided that the results returned by the agents are suitably combined). This greatly reduces the computation performed by `choose_literal`, and provides a simple way of combining Vertical and Horizontal Parallelism by applying a work-sharing method similar to the *Basic Andorra Model* [34], studied for parallelization of Prolog computation.

6.2.2 Scheduling

The key for the integration of Vertical and Horizontal Parallelism is in the way work is divided in work units and assigned to the agents. To keep the system simple, the prototype that we used to test Parallel Lookahead is based on a central scheduler, and a set of agents that are dedicated to the actual computation of the answer sets. Every work unit corresponds to a lookahead step performed on a partial answer set, B, using a set of unexplored literals, U. Work units related to different partial answer sets can be processed at the same time by the system. Every time all the work units associated with certain partial answer set have been completed, the scheduler gathers the results and executes `choose_literal` – which, as we stated before, requires a very small amount of computation, and can thus be executed directly on the scheduler. `choose_literal` returns two (possibly) partial answer sets [4], and the scheduler generates work units for both of them, thus completing a (parallel) iteration of the algorithm in Fig. 2, extended with `lookahead`. Under this perspective, Horizontal Parallelism corresponds to the parallel execution of work units related to the same partial answer set. Vertical Parallelism, instead, is the parallel execution of work units related to different partial answer sets. The way the search space is traversed, as well as the balance between Vertical and Horizontal Parallelism, are determined by: *(1)* the number agents among which the set of unexplored literals is split, and *(2)* the priority given to pending work units. In our implementation we assign priorities to pending work units according to a "simulated depth first" strategy, i.e., the priority of a work unit depends on: (1) the depth, d, in the search space, of the corresponding node, n, and (2) the number of nodes of depth d present to the left of n. This choice guarantees that, if a computation based only on Horizontal Parallelism is selected, the order in which nodes are considered is the same as in a sequential implementation of the algorithm.

The number of agents among which the set of unexplored literals is split is selected at run-time. This allows the user to decide between a computation based on Horizontal Parallelism, useful if the answer set(s) are expected to be found with little backtracking, and a computation based on Vertical Parallelism, useful if more backtracking is expected.

6.2.3 Experimental Results

For our tests on Parallel Lookahead, we have used a subset of the benchmarks available from the Smodels' web site [5]: *(1)* `color`: c-colorability (4 colors, 300 nodes), *(2)* `pigeon`: put N pigeons in M holes with at most one pigeon in a

[4] Our version of `choose_literal` runs `expand` on the two partial answer sets before returning them.

[5] `http://www.tcs.hut.fi/pub/smodels/tests/lp-csp-tests.tar.gz`

hole ($N = 24, M = 24$), *(3)* `queens`: N-queens problem ($N = 14$), and *(4)* `schur`: put N items in B boxes such that, for any $X, Y \in \{1, \ldots, N\}$: items labeled X and $2X$ are in different boxes, and if X and Y are in the same box, then $X + Y$ is in a different box ($N = 35, B = 15$).

Fig. 15. Times for Parallel Lookahead

Fig. 16. Speedups for Parallel Lookahead

	`ParLook`	Smodels
Color	27.120	3.800
Pigeon	5.350	4.430
Queen	6.680	1.120
Schur	37.160	5.540

Fig. 18. Times of `ParLook` (54 agents) and Smodels

Fig. 17. Speedups for Parallel Lookahead

`ParLook` has been run on a distributed memory platform, and employs Model Copying. The tests consisted of finding one answer set for each of the above programs. Since, for all of these programs, an answer set can be found with a comparatively small amount of backtracking, the engine was run so that Horizontal Parallelism was given a higher priority than Vertical Parallelism.

The experiments show, in general, a consistently good speedup for all programs. The speedup measured for `queens` is indeed surprising. It is interesting to note that `queens` requires (with *Smodels*) the highest amount of backtracking. We conjecture that the speedup observed is the result of the combined application of both types of parallelism.

Although in this paper we are interested in investigating the issues involved in automatic exploitation of parallelism, rather than in relating to state-of-the-art sequential engines, we have included in Figure 18 a short comparison with Smodels. The table reports the times obtained with Smodels on the benchmark programs above, and the times obtained with `ParLook` using the maximum number of processors available. Smodels was run will all the available optimizations turned on (including lookahead). Notice that our prototype's performance reasonably close to that of Smodels, considering that many important heuristics present in Smodels are not implemented in `ParLook`.

7 Optimizations

7.1 Parallel Grounding

In this section we present some preliminary results on the development of a parallel grounding engine. The problem is important, since there are ASP programs whose grounding phase requires a significant amount of time. On the other hand, as shown in this section, the problem can be easily addressed via parallelism.

7.1.1 Parallelizing Lparse

The first phase of the execution is characterized by the grounding of the input program. Although for most interesting programs the majority of the execution time of the engine is spent in the actual computation of models, the execution of the local grounding can still require a non-negligible amount of time. For this reason we have decided to investigate simple ways to exploit parallelism also from the preprocessing phase.

The structure of the local grounding process, as illustrated in [62], is based on taking advantage of the strong range restriction to individually ground each rule in th
the task of grounding the different rules to different agents, as in Fig. 19. The **forall** indicated in the algorithm represents a parallel computation: the different iterations are independent of each other. The actual solution adopted in our system is based on the use of a *distribution function* which statically computes a partition of the program Π (after removing all rules defining the domain predicates) and assigns the elements of the partition to the available computing agents. The choice of performing a

function ParallelGround(Π)
  $\Pi_G = \{a \mid \texttt{a is instance of domain predicate}\}$
  $\Pi = \Pi \setminus \Pi_G$
  forall $R^i \in \Pi$
    $R^i_G = \texttt{GroundRule}(R^i)$
  endall
  $\Pi_G = \bigcup R^i_G$
end

Fig. 19: Parallel Preprocessing

static assignment is dictated by *(i)* the large amount of work typically generated, and *(ii)* the desire to avoid costly dynamic scheduling in a distributed memory context. The various computing agents provide as result the ground instantiations of all the rules in their assigned component of the partition of Π. The partitioning of Π is performed in a way to attempt to balance the load between processors. The heuristic used in this context assigns a weight to each rule (an estimation of the number of instances based on the size of the relations of the domain predicates in the body of the rule) and attempts to distribute balanced weight to each agent. Although simplistic in its design, the heuristics have proven effective in the experiments performed.

The preprocessor has been implemented as part of our ASP system, and it is designed to be compatible in input/output formats with the *lparse* preprocessor used in *Smodels*. The preprocessor makes use of an internal representation of the program based on structure sharing—the input rule acts as skeleton and the different instantiations are described as environments for such skeleton. The remaining data structures are essentially identical to those described for the *lparse* system [62]. The implementation of the preprocessor, developed on a Beowulf system, has been organized as a master-slave structure, where the master agent is in charge of computing the program partition while the slaves are in charge of grounding the rules in each partition.

7.1.2 Experimental Results

We have analyzed the performance of the parallel preprocessor by comparing its execution speed with varying number of processors. The parallel preprocessor is in its first prototype and it is very unoptimized (compared to *lparse* we have observed differences in speed ranging from 4% to 48%). Nevertheless, the current implementation was mostly meant to represent a proof of concept concerning the feasibility of extracting parallelism from the preprocessing phase.

The first interesting result that we have observed is that the rather embarrassingly parallel structure of the computation allowed us to make the parallel overhead (i.e., the added computation cost due to the exploitation of parallelism) almost negligible. This can be seen in Fig. 20, which compares the execution times for a direct sequential implementation of the grounding algorithm with the execution times using a single agent in the parallel preprocessor. In no cases we have observed overhead higher than 4.1%. Very good speedups have been observed in each benchmark containing a sufficient number of rules to keep the agents busy. Fig. 21 shows the preprocessing time for two benchmarks using different numbers of processors. Note that for certain benchmarks the speedup is slightly lower than linear due to slightly unbalanced distribution of work between the agents—in the current scheme we are simply relying on a static partitioning without any additional load balancing activities.

Fig. 20. Preprocessing Overhead (Pigeon, Coloring)

Fig. 21. Parallel Execution of the Preprocessor

7.2 Other Optimizations

Further research is needed in order to improve the efficiency of the system. Different types of improvements can be identified.

(1) *Design improvements*, aimed at decreasing the overhead due to communications. Improvements will probably need to be focused on the selection of the correct work sharing model, for which the hybrid method is a good candidate. The development of better scheduling techniques will also be important to achieve a higher efficiency.
(2) Optimization of the heuristic function used to find the "best" literal for `choose_literal`, in order to exploit the features of the parallel implementation: we are currently using a heuristic function close to the one used in *Smodels*, designed for sequential implementations.
(3) Improvements aimed at making the system able to self-adapt according the type of logic program whose answer sets are to be found. Research has to be conducted on techniques for selecting the correct balance between Vertical Parallelism and Horizontal Parallelism depending on the task to be performed.

8 Conclusions and Future Work

In this paper we have presented an overview of the current effort in developing technology for the parallel execution of answer set programs. ASP has quickly become a leading paradigm for the high-level development of applications in areas such as planning and scheduling.

The investigation has led to the identification of two major forms of parallelism—horizontal parallelism and vertical parallelism—that can be automatically exploited from the commonly used execution model for ASP. We have illustrated the major issues behind the exploitation of these forms of parallelism and described some possible solutions. These solutions have been integrated in actual prototypes and the paper reported the performance results obtained. The results accomplished demonstrates that good parallel performance is possible, especially for programs that are rich of non-determinism and for programs where heavy pruning is realized through lookahead.

The current research has highlighted the inherent difficulties in the efficient exploitation of horizontal parallelism from ASP programs. The experiments conducted indicates that exploitation of horizontal parallelism is heavily hampered by some key aspects of ASP execution:

(1) *granularity:* the steps performed during the execution of the `expand` operation are very fine grained;
(2) *dependencies:* the activities required to expand a partial model require intense interactions—each worker needs to have an up-to-date view of the partial answer set in order to effectively progress the expansion;
(3) *irregularity:* traditional partitioning techniques (e.g., partitioning based on predicates) lead to unbalanced computations and/or increased communication overheads.

In the search for better solutions to this problem, we have initiated an investigation aimed at developing horizontal parallel models for ASP where partitioning is driven by the syntactic and semantic properties of the program. This effort is facilitated by the rich collection of theoretical results that have been developed over the years in the context of stable models and answer set semantics [8]. We are currently exploring one main property of answer set programs to drive horizontal parallel execution: *splitting*. *Splitting* is a property of logic programs under the answer set semantics originally studied by Lifschitz and Turner [39,25]. Given a logic program P, a *splitting set* of P is a set of atoms U with the following property: for each rule r in the program, if the head of r belongs to U, the all the atoms in the body of r belong to U. A splitting set U of P suggests a partitioning of the program in two parts: the set of all the rules whose head is in U (*bottom*—$b_U(P)$) and the set of all rules whose head is not in U (*top*—$t_U(P)$). The *Splitting Theorem* [39] guarantees that each answer set of P can be computed by first computing each answer set of $b_U(P)$ and then using such answer sets to determine the answer sets of $t_U(P)$. Splitting can be generalized to obtain a splitting of a program in n layers with the same property.

Our current effort is aimed at viewing the computation of answer sets as a *pipelined computation*, where the different stages of the pipeline corresponds to the different components of a program splitting. The pipeline allows data movement in both directions, since each layer of the pipeline can support the computation of both the preceding as well as the consecutive layers. We are currently developing a prototypical implementation of an engine based on this view of horizontal parallelism. We expect that this approach can be significant for large size programs with a regular splitting structure—this is, for example, the case of answer set programs obtained from planning applications [9].

Acknowledgments

The authors wish to thank G. Gupta, S. Tran, and M. Gelfond for their help. The authors would also like to acknowledge the anonymous reviewers for their thorough work and their insightful suggestions. This work is partially supported by NSF grants CCR9875279, CCR9900320, CDA9729848, EIA0130887, EIA0220590 and HRD9906130, by NASA grant NCC9-157, and by United Space Alliance grant 26-3502-21 and contract COC6771311.

References

[1] S. Abiteboul and V. Vianu. Fixpoint Extensions of First-order Logic and Datalog-like Languages. In *Symposium on Logic in Computer Science*, pages 71–89. IEEE Computer Society, 1989.

[2] K.A.M. Ali and R. Karlsson. The Muse Or-parallel Prolog Model and its Performance. In *1990 N. American Conf. on Logic Prog.*, pages 757–776. MIT Press, 1990.

[3] K. Apt, H. Blair, and A. Walker. Towards a Theory of Declarative Knowledge. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*. Morgan Kaufmann, 1989.

[4] K.R. Apt and R.N. Bol. Logic Programming and Negation: A Survey. *Journal of Logic Programming*, 19/20, 1994.

[5] Y. Babovich. CModels. `www.cs.utexas.edu/users/tag/cmodels.html`, 2002.

[6] M. Balduccini and M. Gelfond. Diagnostic Reasoning with A-Prolog. *Theory and Practice of Logic Programming*, 3(4–5):425–461, 2003.

[7] M. Balduccini and M. Gelfond. Logic Programs with Consistency-Restoring Rules. In *AAAI Spring Symposium*. AAAI/MIT Press, 2003.

[8] C. Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, 2003.

[9] C. Baral and M. Gelfond. Logic Programming and Knowledge Representation. *Journal of Logic Programming*, 19/20:73–148, 1994.

[10] C. Baral and M. Gelfond. Reasoning Agents in Dynamic Domains. Technical report, University of Texas at El Paso, 1999.

[11] C. Bell, A. Nerode, R. Ng, and V.S. Subrahmanian. Implementing Stable Semantics by Linear Programming. In *Logic Programming and Non-monotonic Reasoning*, pages 23–42. MIT Press, 1993.

[12] G. Brewka. Logic Programming with Ordered Disjunction. In *AAAI/IAAA*, pages 100–105. AAAI/MIT Press, 2002.

[13] W. Chen and D. S. Warren. Computation of Stable Models and its Integration with Logical Query Processing. *Transactions on Knowledge and Data Engineering*, 8(5):742–757, 1996.

[14] P. Cholewinski, V.W. Marek, and M. Truszczyński. Default Reasoning System DeReS. In *International Conference on Principles of Knowledge Representation and Reasoning*, pages 518–528. Morgan Kauffman, 1996.

[15] K.L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*. Plenum, 1978.

[16] W.F. Clocksin and H. Alshawi. A Method for Efficiently Executing Horn Clause Programs Using Multiple Processors. *New Generation Computing*, 5:361–376, 1988.

[17] A. Dekhtyar and V.S. Subrahmanian. Hybrid Probabilistic Programs. *Journal of Logic Programming*, 43(3):187–250, 2000.

[18] Y. Dimopoulos, B. Nebel, and J. Koehler. Encoding Planning Problems in Nonmonotonic Logic Programs. In *European Workshop on Planning*, pages 169–181, 1997.

[19] W.F. Dowling and J.H. Gallier. Linear-time Algorithms for Testing the Satisfiability of Propositional Horn Formulae. *Journal of Logic Programming*, 3:267–289, 1984.

[20] D. East and M. Truszczyński. Datalog with Constraints. In *Proceedings of the National Conference on Artificial Intelligence*, pages 163–168. AAAI/MIT Press, 2000.

[21] T. Eiter, W. Faber, N. Leone, G. Pfeifer, and A. Polleres. Answer Set Planning under Action Cost. In *European Conference on Logics in AI (JELIA)*. Springer Verlag, 2002.

[22] T. Eiter, G. Gottlob, and N. Leone. Abduction from Logic Programs: Semantics and Complexity. *Theoretical Computer Science*, 189(1-2):129–177, 1997.

[23] T. Eiter, N. Leone, C. Mateis, G. Pfeifer, and F. Scarcello. The KR System `dlv`: Progress Report, Comparisons, and Benchmarks. In *International Conference on Principles of Knowledge Representation and Reasoning*, pages 406–417, 1998.

[24] O. El-Khatib and E. Pontelli. Parallel Evaluation of Answer Sets Programs Preliminary Results. In *Workshop on Parallelism and Implementation of Logic Programming*, 2000.

[25] S. Erdogan and V. Lifschitz. Definitions in Answer Set Programming. In *Logic Programming and Non-monotonic Reasoning*, Springer Verlag, pp. 114–126, 2004.

[26] E. Erdem, V. Lifschitz, L. Nakhleh, and D. Ringe. Reconstructing the Evolutionary History of Indo-European Languages using Answer Set Programming. In *Practical Aspects of Declarative Languages*, pages 160–176. Springer Verlag, 2003.

[27] R. Finkel, V. Marek, N. Moore, and M. Truszczyński. Computing Stable Models in Parallel. In A. Provetti and S.C. Tran, editors, *Proceedings of the AAAI Spring Symposium on Answer Set Programming*, pages 72–75, Cambridge, MA, 2001. AAAI/MIT Press.

[28] M. Gelfond. The USA-Advisor: a Case Study in Answer Set Programming. In *Logics in Artificial Intelligence*, pages 566–568. Springer Verlag, 2002.

[29] M. Gelfond and V. Lifschitz. Classical Negation in Logic Programs and Disjunctive Databases. New Generation Computing, pages 365–385. 1991.

[30] M. Gelfond and V. Lifschitz. The Stable Model Semantics for Logic Programs. In *International Symposium on Logic Programming*, pages 1070–1080. MIT Press, 1988.

[31] M. Gelfond and V. Lifschitz. Action Languages. In *Electronic Notes in Artificial Intelligence*, 3(6), 1998.

[32] G. Gupta. *Multiprocessor Execution of Logic Programs*. Kluwer Academic Press, Dordrecht, 1994.

[33] G. Gupta and E. Pontelli. Optimization Schemas for Parallel Implementation of Nondeterministic Languages and Systems. In *International Parallel Processing Symposium*, Los Alamitos, CA, 1997. IEEE Computer Society.

[34] G. Gupta, E. Pontelli, M. Carlsson, M. Hermenegildo, and K.M. Ali. Parallel Execution of Prolog Programs: a Survey. *ACM Transactions on Programming Languages and Systems*, 23(4):472–602, 2001.

[35] K. Heljanko and I. Niemela. Bounded LTL Model Checking with Stable Models. In *International Conference on Logic Programming and Non-monotonic Reasoning*. Springer Verlag, 2001.

[36] R. Krishnamurthy and S.A. Naqvi. Non-deterministic Choice in Datalog. In *International Conference on Data and Knowledge Bases*, pages 416–424. Morgan Kaufmann, 1988.

[37] V. Lifschitz. Answer Set Programming and Plan Generation. Artificial Intelligence, Vol. 138, pages 39–54. 2002.

[38] V. Lifschitz. Answer Set Planning. In *Logic Programming and Non-monotonic Reasoning*, pages 373–374. Springer Verlag, 1999.

[39] V. Lifschitz and H. Turner. Splitting a logic program. In *Proceedings of the Eleventh International Conference on Logic Programming*, pages 23–37. MIT Press, 1994.

[40] F. Lin and Y. Zhao. ASSAT: Computing Answer Sets of Logic Programs by SAT Solvers. In *AAAI/IAAA*, pages 112–117. AAAI/MIT Press, 2002.

[41] J.W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, Heidelberg, 1987.

[42] V.W. Marek and M. Truszczyński. Stable Models and an Alternative Logic Programming Paradigm. In K.R. Apt, V.W. Marek, M. Truszcziński, and D. S. Warren, editors, *The Logic Programming Paradigm.* Springer Verlag, 1999.

[43] W. Marek, A. Nerode, and J.B. Remmel. The Stable Models of Predicate Logic Programs. In *Joint International Conference and Symposium on Logic Programming.* MIT Press, 1992.

[44] S. McIlraith, T.C. Son, and H. Zeng. Semantic Web Services. *IEEE Intelligent Systems*, march/april 2001.

[45] S. McIlraith, T.C. Son, and H. Zeng. Mobilizing the Semantic Web with DAML-Enabled Web Services. In *Proceedings of the Second International Workshop on the Semantic Web (SemWeb'01)*, 2001.

[46] J. Minker. An Overview of Nonmonotonic Reasoning and Logic Programming. *Journal of Logic Programming*, 17(2/3/4), 1993.

[47] T. Nguyen and Y. Deville. A Distributed Arc-Consistency Algorithm. *Science of Computer Programming*, 30(1–2):227–250, 1998.

[48] I. Niemela. Logic Programs with Stable Model Semantics as a Constraint Programming Paradigm. *Annals of Mathematics and AI*, (to appear).

[49] I. Niemela and P. Simons. Smodels - An Implementation of the Stable Model and Well-Founded Semantics for Normal LP. In *Logic Programming and Non-monotonic Reasoning*, pages 421–430. Springer Verlag, 1997.

[50] I. Niemela, P. Simons, and T. Soininen. Stable Model Semantics of Weight Constraint Rules. In *Logic Programming and Non-monotonic Reasoning.* Springer Verlag, 1999.

[51] E. Pontelli and G. Gupta. Efficient Backtracking in And-Parallel Implementations of Non-deterministic Languages. In T. Lai, editor, *Proceedings of the International Conference on Parallel Processing*, pages 338–345, Los Alamitos, CA, 1998. IEEE Computer Society.

[52] E. Pontelli and D. Ranjan. On the Complexity of Dependent And-Parallelism in Logic Programming. Technical report, New Mexico State University, 2002.

[53] E. Pontelli and T.C. Son. Smodels with CLP: a Treatment of Aggregates in ASP. Technical Report NMSU-CSTR-2003-008, New Mexico State University, 2003.

[54] I. V. Ramakrishnan, P. Rao, K.F. Sagonas, T. Swift, and D. S. Warren. Efficient Tabling Mechanisms for Logic Programs. In *International Conference on Logic Programming*. MIT Press, 1995.

[55] D. Ranjan, E. Pontelli, and G. Gupta. On the Complexity of Or-Parallelism. *New Generation Computing*, 17(3):285–308, 1999.

[56] D. Saccà and C. Zaniolo. Stable Models and Non-determinism in Logic Programs with Negation. In *Symposium on Principles of Database Systems*, pages 205–217. ACM Press, 1990.

[57] C. Schulte. Programming Constraint Inference Engines. In *Principles and Practice of Constraint Programming*, pages 519–533. Springer Verlag, 1997.

[58] C. Schulte. Compairing Trailing and Copying for Constraint Programming. In *International Conference on Logic Programming*, pages 275–289. MIT Press, 1999.

[59] J.C. Shepherdson. Negation in Logic Programming. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*. Morgan Kaufmann, 1989.

[60] P. Simons. *Extending and Implementing the Stable Model Semantics*. PhD thesis, Helsinki University of Technology, 2000.

[61] V.S. Subrahmanian, D. Nau, and C. Vago. WFS + Branch and Bound = Stable Models. *Transactions on Knowledge and Data Engineering*, 7(3):362–377, 1995.

[62] T. Syrjanen. Implementation of Local Grounding for Logic Programs with Stable Model Semantics. Technical Report B-18, Helsinki University of Technology, 1998.

[63] A. Van Gelder, K.A. Ross, and J.S. Schlipf. The Well-Founded Semantics for General Logic Programs. *Journal of the ACM*, 38(3):620–650, 1991.