Available online at www.sciencedirect.com
ScienceDirect
Journal of Algorithms: Algorithms in Cognition, Informatics and Logic 63 (2008) 34–54
www.elsevier.com/locate/jalgor

Experimenting with parallelism for the instantiation of ASP programs ☆

F. Calimeri *, S. Perri, F. Ricca

Dipartimento di Matematica, Università della Calabria, 87036 Rende (CS), Italy

Received 26 October 2007

Available online 7 March 2008

Abstract

In the last few years, microprocessor technologies have been moving towards multi-core architectures, in order to improve performance as well as reduce power consumption. This makes real Symmetric MultiProcessing (SMP) available even on non-dedicated machines, and paves the way to the development of better performing software. Notably, the recent application of Answer Set Programming (ASP) in different emerging areas, such as knowledge management or information extraction/integration, shows that performance is a crucial issue also for ASP systems. Among the tasks performed by such systems, the instantiation process, which consists of generating a variable-free program equivalent to the input one, is one of the most expensive from a computational viewpoint, especially in the case of huge input data. In this paper a new strategy exploiting parallelism for the instantiation of ASP programs is proposed. An implementation of this strategy and its integration with the grounding module of the DLV system is discussed. The results of an experimental analysis are also presented, which confirm that the strategy is effective in making ASP instantiation more efficient.
© 2008 Elsevier Inc. All rights reserved.

Keywords: Answer Set Programming; Disjunctive Logic Programming; Instantiation; Parallelism

1. Introduction

Symmetric MultiProcessing (SMP) [1,2] is a computer architecture where two or more identical processors connect to a single shared main memory resource allowing simultaneous multithread execution. In the past, only servers and workstations could take advantage of it. However, recently, technology has moved to multicore/multiprocessor architectures also for entry-level systems and PCs; this permits the enjoyment of the benefits of parallel processing on a large scale. These benefits include better workload balances, enhanced performance, improved scalability, not only for systems that run many processes simultaneously, but also for single (i.e., multithreaded) applications. This technology might be profitably exploited in the field of Answer Set Programming (ASP). Indeed, recent applications

☆ Supported by M.I.U.R. within projects "Potenziamento e Applicazioni della Programmazione Logica Disgiuntiva" and "Sistemi basati sulla logica per la rappresentazione di conoscenza: estensioni e tecniche di ottimizzazione".
* Corresponding author.
E-mail addresses: calimeri@mat.unical.it (F. Calimeri), perri@mat.unical.it (S. Perri), ricca@mat.unical.it (F. Ricca).

0196-6774/$ – see front matter © 2008 Elsevier Inc. All rights reserved.
doi:10.1016/j.jalgor.2008.02.003

of ASP in different emerging areas, such as knowledge management or information extraction/integration [3–5], have evidenced the practical need for faster ASP systems.

ASP is a declarative approach to programming proposed in the area of nonmonotonic reasoning and logic programming [6–12].[1] The main advantage of ASP is its high declarative nature combined with a relatively high expressive power [13,14]; but this comes at the price of a high computational cost, which makes the implementation of efficient ASP systems a difficult task. Some effort has been made to this end, and, after some pioneering work [15,16], there are nowadays a number of systems that support ASP and its variants [13,17–26]. The kernel modules of such systems operate on a ground instantiation of the input program, i.e. a program that does not contain any variable, but is semantically equivalent to the original input [27]. Consequently, any given program $\mathcal{P}$ first undergoes the so-called instantiation process computing from $\mathcal{P}$ an equivalent ground program $\mathcal{P}'$. This preprocessing phase is computationally very expensive;[2] thus, having an efficient instantiation procedure is, in general, a key feature of ASP systems. Many optimization techniques have been proposed for this purpose [29–31]; nevertheless, the performance of instantiators is still not acceptable in many cases, especially when the input data are significantly large (real-world instances, for example, may count hundreds of thousands of tuples).

This paper proposes a new strategy for the parallel instantiation of ASP programs, allowing the performance of instantiators to be improved by exploiting the power of multiprocessor computers. This strategy takes advantage of some structural properties of the input program in order to reduce the usage of mutex-locks and thus the time spent by concurrency-control mechanisms. In particular, the strategy is two-fold, focusing on two different aspects of the instantiation process. On the one hand, it examines the structure of the input program P, splits it into modules whose union is equivalent to P, and, according to the interdependencies between the modules, decides which of them can be processed in parallel. On the other hand, it parallelizes the evaluation within each module by allowing the concurrent instantiation of rules.

The proposed strategy is implemented into DLV, one of the two most popular ASP instantiators (the other being Lparse [32,33]), building a new parallel ASP instantiator. An experimental activity is performed to evaluate the impact of the method on the instantiation process. The results of the experiments are very positive, and confirm the effectiveness of this technique, especially in the evaluation of programs with a large amount of input data. The benchmark programs, as well as the binaries used for our experiments, are available at http://www.mat.unical.it/parallel/parallel_bench_07.zip.

It is worthwhile noting that the results can profitably be exploited by other systems, which do not have their own instantiators like, e.g., ASSAT [23], Cmodels [24,34], Smodels [20,21,32], NoMore [25], Clasp [22], etc.[3] Indeed, these systems can use this instantiator to obtain the ground program, and then apply their own procedures for the evaluation of the ground program.[4]

The remainder of the paper is structured as follows. Section 2 outlines some basic notions of Answer Set Programming; Section 3 describes how ASP programs are instantiated by the DLV system; Section 4 presents the parallel instantiation strategy; Section 5 discusses the results of the experiments carried out in order to evaluate the proposed technique; finally, Section 6 is devoted to related works, and Section 7 draws some conclusions.

2. Answer set programming

A formal definition of syntax and semantics of answer set programs is provided next.

[1] The term "Answer Set Programming" was introduced by Vladimir Lifschitz in his invited talk at ICLP'99 to denote a declarative programming methodology. Concerning terminology, ASP is sometimes used in a somewhat broader sense, referring to any declarative formalism, which represents solutions as sets. However, the more frequent understanding is the one adopted in this article, which dates back to [6]. Moreover, since ASP is the most prominent branch of logic programming in which rule heads may be disjunctive, sometimes the term Disjunctive Logic Programming can be found referring explicitly to ASP. Yet other terms for ASP are A-Prolog and Stable Logic Programming. For introductory material on ASP, we refer to [9–12].

[2] Producing the ground instantiation may require an exponential time in the size of the input program. In order to grasp the intuition consider the following program containing only one nonground rule: $obj(0).obj(1).disp(X_1,\ldots,X_n) :- obj(X_1),\ldots,obj(X_n)$. It is easy to see that the corresponding ground instantiation contains 2^n+2 rules. For more details about complexity results the reader may refer to [7,13,14,28].

[3] Obviously, systems not supporting disjunction can exploit our instantiator for nondisjunctive programs only.

[4] Remember that the instantiator can deal also with normal nondisjunctive programs.

2.1. Syntax

A variable or a constant is a *term*.[5] An *atom* is $a(t_1, \ldots, t_n)$, where a is a *predicate* of arity n and $t_1, \ldots, t_n$ are terms. A *literal* is either a *positive literal* p or a *negative literal* `not` p, where p is an atom. A *disjunctive rule* (*rule*, for short) r is a formula

$$a_1 \vee \cdots \vee a_n :- b_1, \ldots, b_k, \texttt{not}\ b_{k+1}, \ldots, \texttt{not}\ b_m, \tag{1}$$

where $a_1, \ldots, a_n, b_1, \ldots, b_m$ are atoms and $n \geqslant 0$, $m \geqslant k \geqslant 0$. The disjunction $a_1 \vee \cdots \vee a_n$ is the *head* of r, while the conjunction $b_1, \ldots, b_k, \texttt{not}\ b_{k+1}, \ldots, \texttt{not}\ b_m$ is the *body* of r. A rule without head literals (i.e. $n = 0$) is usually referred to as an *integrity constraint*.[6] A rule having precisely one head literal (i.e. $n = 1$) is called a *normal rule*. If the body is empty (i.e. $k = m = 0$), it is called a *fact*.

$H(r)$ denotes the set $\{a_1, \ldots, a_n\}$ of the head atoms, and by $B(r)$ the set $\{b_1, \ldots, b_k, \texttt{not}\ b_{k+1}, \ldots, \texttt{not}\ b_m\}$ of the body literals. $B^+(r)$ (respectively, $B^-(r)$) denotes the set of atoms occurring positively (respectively, negatively) in $B(r)$. A rule r is *safe* if each variable appearing in r appears also in some positive body literal of r.

An *ASP program* $\mathcal{P}$ is a finite set of safe rules. A `not`-free (respectively, $\vee$-free) program is called *positive* (respectively, *normal*). A term, an atom, a literal, a rule, or a program is *ground* if no variables appear in it. Accordingly with the database terminology, a predicate occurring only in *facts* is referred to as an *EDB* predicate, all others as *IDB* predicates; the set of facts of $\mathcal{P}$ is denoted by $EDB(\mathcal{P})$.

2.2. Semantics

Let $\mathcal{P}$ be a disjunctive logic program. The *Herbrand universe* of $\mathcal{P}$, denoted as $U_\mathcal{P}$, is the set of all constants appearing in $\mathcal{P}$. In case no constant appears in $\mathcal{P}$, an arbitrary constant ψ is added to $U_\mathcal{P}$. The *Herbrand base* of $\mathcal{P}$, denoted as $B_\mathcal{P}$, is the set of all ground atoms constructible from the predicate symbols appearing in $\mathcal{P}$ and the constants of $U_\mathcal{P}$.

Given a rule r occurring in $\mathcal{P}$, a *ground instance* of r is a rule obtained from r by replacing every variable X in r by $\sigma(X)$, where σ is a substitution mapping the variables occurring in r to constants in $U_\mathcal{P}$. $ground(\mathcal{P})$ denotes the set of all the ground instances of the rules occurring in $\mathcal{P}$.

Interpretations and models. An *interpretation* for $\mathcal{P}$ is a set of ground atoms, that is, an interpretation is a subset I of $B_\mathcal{P}$. A ground positive literal A is *true* (respectively, *false*) with respect to I if $A \in I$ (respectively, $A \notin I$). A ground negative literal `not` A is *true* with respect to I if A is false with respect to I; otherwise `not` A is false with respect to I.

Let r be a ground rule in $ground(\mathcal{P})$. The head of r is *true* with respect to I if $H(r) \cap I \neq \emptyset$. The body of r is *true* with respect to I if all body literals of r are true with respect to I (i.e., $B^+(r) \subseteq I$ and $B^-(r) \cap I = \emptyset$) and is *false* with respect to I otherwise. The rule r is *satisfied* (or *true*) with respect to I if its head is true with respect to I or its body is false with respect to I.

A *model* for $\mathcal{P}$ is an interpretation M for $\mathcal{P}$ such that every rule $r \in ground(\mathcal{P})$ is true with respect to M. A model M for $\mathcal{P}$ is *minimal* if no model N for $\mathcal{P}$ exists such that N is a proper subset of M. The set of all minimal models for $\mathcal{P}$ is denoted by $\mathrm{MM}(\mathcal{P})$.

Answer sets. Given a ground program $\mathcal{P}$ and an interpretation I, the *reduct* of $\mathcal{P}$ with respect to I is the subset $\mathcal{P}^I$ of $\mathcal{P}$, which is obtained from $\mathcal{P}$ by deleting rules in which a body literal is false with respect to I.

Note that the above definition of reduct, proposed in [35], simplifies the original definition of Gelfond and Lifschitz (GL) transform [6], but is fully equivalent to the GL transform for the definition of answer sets [35].

[5] Function symbols are not allowed.

[6] Note that, a constraint is a shorthand for *false* :– $b_1, \ldots, b_k$, `not` $b_{k+1}, \ldots,$ `not` b_m, and it is also assumed that a rule *bad* :– *false*, `not` *bad* is in the program, where *false* and *bad* are special symbols appearing nowhere else in the original program.

Definition 1. (See [6,36].) Let I be an interpretation for a program $\mathcal{P}$. I is an *answer set* for $\mathcal{P}$ if $I \in \mathrm{MM}(\mathcal{P}^I)$ (i.e., I is a minimal model for the program $\mathcal{P}^I$).[7] The set of all answer sets for $\mathcal{P}$ is denoted by $ANS(\mathcal{P})$.

Example 1. Given the general program $\mathcal{P}_1 = \{a \text{ :- } c, \texttt{not } b.,\ b \vee c \text{ :- } \texttt{not } a.,\ b \text{ :- } \texttt{not } c, \texttt{not } a.\}$ and the interpretation $I = \{b\}$, the reduct $\mathcal{P}_1^I$ is $\{b \vee c \text{ :- } \texttt{not } a.,\ b \text{ :- } \texttt{not } c, \texttt{not } a.\}$. As it is easy to see, I is a minimal model of $\mathcal{P}_1^I$, and, for this reason, it is also an answer set of $\mathcal{P}_1$. Now consider $J = \{a\}$. The reduct $\mathcal{P}_1^J$ is empty, that is $\mathcal{P}_1^J = \emptyset$. Therefore, J is not a minimal model of $\mathcal{P}_1^J$ and, thus, it is not an answer set of $\mathcal{P}_1$. It can be easily verified that I is the only answer set of $\mathcal{P}_1$.

3. The DLV instantiator

In this section a description is provided of the DLV instantiator. Given an input program $\mathcal{P}$, it efficiently generates a ground instantiation that has the same answer sets as the full one, but is much smaller in general [13]. Note that the size of the instantiation is a crucial aspect for efficiency, since the answer set computation takes an exponential time (in the worst case) in the size of the ground program received as input (i.e., produced by the instantiator). In order to generate a small ground program equivalent to $\mathcal{P}$, the DLV instantiator generates ground instances of rules containing only atoms which can possibly be derived from $\mathcal{P}$, and thus avoiding the combinatorial explosion which can be obtained by naively considering all the atoms in the Herbrand base [30]. This is obtained by taking into account some structural information of the input program, concerning the dependencies among IDB predicates.

The *dependency graph* of $\mathcal{P}$ is now defined, which, intuitively, describes how predicates depend on each other.

Definition 2. Let $\mathcal{P}$ be a program. The *dependency graph* of $\mathcal{P}$ is a directed graph $G_{\mathcal{P}} = \langle N, E \rangle$, where N is a set of nodes and E is a set of arcs. N contains a node for each IDB predicate of $\mathcal{P}$, and E contains an arc $e = (p, q)$ if there is a rule r in $\mathcal{P}$ such that q occurs in the head of r and p occurs in a positive literal of the body of r.

The graph $G_{\mathcal{P}}$ induces a subdivision of $\mathcal{P}$ into subprograms (also called *modules*) allowing for a modular evaluation. We say that a rule $r \in \mathcal{P}$ *defines* a predicate p if p appears in the head of r. For each strongly connected component (SCC)[8] C of $G_{\mathcal{P}}$, the set of rules defining all the predicates in C is called *module* of C and is denoted by $\mathcal{P}_C$.[9]

More in detail, a rule r occurring in a module $\mathcal{P}_C$ (i.e., defining some predicate $q \in C$) is said to be *recursive* if there is a predicate $p \in C$ occurring in the positive body of r; otherwise, r is said to be an *exit rule*.

Example 2. Consider the following program $\mathcal{P}$, where a is an EDB predicate:

$$p(X, Y) \vee s(Y) \text{ :- } q(X),\ q(Y),\ not\ t(X, Y). \qquad q(X) \text{ :- } a(X).$$
$$p(X, Y) \text{ :- } q(X),\ t(X, Y). \qquad t(X, Y) \text{ :- } p(X, Y),\ s(Y).$$

Graph $G_{\mathcal{P}}$ is illustrated in Fig. 1; the strongly connected components of $G_{\mathcal{P}}$ are $\{s\}$, $\{q\}$ and $\{p, t\}$. They correspond to the three following modules:

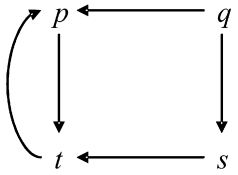
- $\{p(X, Y) \vee s(Y) \text{ :- } q(X),\ q(Y),\ not\ t(X, Y).\}$;
- $\{q(X) \text{ :- } a(X).\}$;
- $\{p(X, Y) \text{ :- } q(X),\ t(X, Y).\ p(X, Y) \vee s(Y) \text{ :- } q(X),\ q(Y),\ not\ t(X, Y).\ t(X, Y) \text{ :- } p(X, Y),\ s(Y).\}$.

Moreover, the first and second module do not contain recursive rules, while the third one contains one exit rule, namely $p(X, Y) \vee s(Y) \text{ :- } q(X),\ q(Y),\ not\ t(X, Y)$, and two recursive rules.

[7] Answer sets are also called stable models.

[8] We briefly recall here that a strongly connected component of a directed graph is a maximal subset of the vertices, such that every vertex is reachable from every other vertex.

[9] Note that, since integrity constraints are considered as rules with exactly the same head (which is a special symbol appearing nowhere in the program), they all belong to the same module.



$G_{\mathcal{P}}$

Fig. 1. Dependency graph.

```
Procedure Instantiate (P: Program; G_P: DependencyGraph;
                       var Π: GroundProgram)
begin
  var S: SetOfAtoms;
  var C: SetOfPredicates;
  S = EDB(P); Π := ∅;
  while G_P ≠ ∅ do
    Remove a SCC C from G_P without incoming edges;
    InstantiateComponent(P, C, S, Π);
  end while
end Procedure;

Procedure InstantiateComponent (P: Program; C: Component;
var S: SetOfAtoms; var Π: GroundProgram)
begin
  var ΔNS: SetOfAtoms;
  var ΔS: SetOfAtoms;
  ΔNS := ∅; ΔS := ∅;
  for each r ∈ Exit(C, P); do
    InstantiateRule(r, S, ΔS, ΔN, Π);
  end for
  do
    ΔS := ΔNS; ΔNS = ∅;
    for each r ∈ Recursive(C, P); do
      InstantiateRule(r, S, ΔS, ΔNS, Π);
    S := S ∪ ΔS;
  while ΔNS ≠ ∅
end Procedure;
```

Fig. 2. The DLV instantiation procedure.

The dependency graph[10] induces a partial ordering among its SCCs, defined as follows: for any pair of SCCs A, B of $G_{\mathcal{P}}$, we say that B *directly depends on* A (denoted $A \prec B$) if there is an arc from a predicate of A to a predicate of B; and, B *depends* on A if $A \prec_s B$, where $\prec_s$ denotes the transitive closure of relation $\prec$.

Example 3. Consider the dependency graph $G_{\mathcal{P}}$ shown in Fig. 1; it is easy to see that component $\{p, t\}$ depends on components $\{s\}$ and $\{q\}$, while $\{s\}$ depends only on $\{q\}$.

This ordering can be exploited to single out an ordered sequence $C_1, \ldots, C_n$ of SCCs of $G_{\mathcal{P}}$ (which is not unique, in general) such that whenever C_j depends on C_i, C_i precedes C_j in the sequence (i.e. $i < j$). Intuitively, this partial ordering allows the evaluation of the program one module at time, so that all data needed for the instantiation of a module C_i have been already generated by the instantiation of the modules preceding C_i.

A description of the instantiation process based on this principle follows.

The procedure *Instantiate* shown in Fig. 2 takes as input both a program $\mathcal{P}$ to be instantiated and the dependency graph $G_{\mathcal{P}}$, and outputs a set Π of ground rules containing only atoms which can possibly be derived from $\mathcal{P}$, such that

[10] It is worth remembering that, according to Definition 2, the dependency graph does not take into account negative dependencies.

$ANS(\mathcal{P}) = ANS(\Pi \cup EDB(\mathcal{P}))$. As already pointed out, the input program $\mathcal{P}$ is divided into modules corresponding to the SCCs of the dependency graph $G_{\mathcal{P}}$. Such modules are evaluated one at a time according to an ordering induced by the dependency graph.

The algorithm creates a new set of atoms S that will contain the subset of the Herbrand Base significant for the instantiation. Initially, $S = EDB(\mathcal{P})$, and $\Pi = \emptyset$. Then, a strongly connected component C, with no incoming edge, is removed from $G_{\mathcal{P}}$, and the program module corresponding to C is evaluated by invoking *InstantiateComponent*. This ensures that modules are evaluated one at a time so that whenever $C_1 \prec_s C_2$, $\mathcal{P}_{C_1}$ is evaluated before $\mathcal{P}_{C_2}$. The *Instantiate* procedure runs on until all the components of $G_{\mathcal{P}}$ have been evaluated.

Example 4. Let $\mathcal{P}$ be the program of Example 2. The unique component of $G_{\mathcal{P}}$ having no incoming edges is $\{q\}$. Thus the program module $\mathcal{P}_q$ is evaluated first. Then, once $\{q\}$ has been removed from $G_{\mathcal{P}}$, $\{s\}$ becomes the (unique) component of $G_{\mathcal{P}}$ having no incoming edge and is therefore taken. Once $\{s\}$ has been evaluated and thus removed from $G_{\mathcal{P}}$, $\{p, t\}$ is processed at last, completing the instantiation process.

Procedure *InstantiateComponent*, in turn, takes as input the component C to be instantiate and S, and for each atom a belonging to C, and for each rule r defining a, computes the ground instances of r containing only atoms which can possibly be derived from $\mathcal{P}$. At the same time, it updates the set S with the atoms occurring in the heads of the rules of Π. To this end, each rule r in the program module of C is processed by calling procedure *InstantiateRule*. This, given the set of atoms which are known to be significant up to now, builds all the ground instances of r, adds them to Π, and marks as significant the head atoms of the newly generated ground rules. It is worth noting that a disjunctive rule r may appear in the program modules of two different components. In order to deal with this, before processing r, *InstantiateRule* checks whether it has been already grounded during the instantiation of another component. This ensures that a rule is actually processed only within one program module.

Concerning recursive rules, they are processed several times according to a semi-naïve evaluation technique [37], where at each iteration n only the significant information derived during iteration $n-1$ has to be used. This is implemented by partitioning significant atoms into three sets: ΔS, S, and $\mathcal{N}S$. $\mathcal{N}S$ is filled with atoms computed during current iteration (say n); ΔS contains atoms computed during previous iteration (say $n-1$); and, S contains the ones previously computed (up to iteration $n-2$). Initially, ΔS and $\mathcal{N}S$ are empty, and the exit rules contained in the program module of C are evaluated by a single[11] call to procedure *InstantiateRule*; then, the recursive rules are evaluated (do-while loop). At the beginning of each iteration, $\mathcal{N}S$ is assigned to ΔS, i.e. the new information derived during iteration n is considered as significant information for iteration $n+1$. Then, *InstantiateRule* is invoked for each recursive rule r, and, at the end of each iteration, ΔS is added to S (since it has already been exploited). The procedure stops whenever no new information has been derived (i.e. $\mathcal{N}S = \emptyset$).

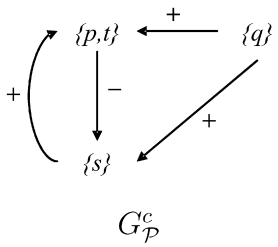
Proposition 5. *(See [38].) Let $\mathcal{P}$ be an ASP program, and Π be the ground program generated by the algorithm Instantiate. Then $ANS(\mathcal{P}) = ANS(\Pi \cup EDB(\mathcal{P}))$ (i.e. $\mathcal{P}$ and $\Pi \cup EDB(\mathcal{P})$ have the same answer sets).*

4. Parallel instantiation techniques

In this section a new instantiation algorithm is described that computes a ground version of a given program $\mathcal{P}$ by exploiting parallelism. This new algorithm takes advantage of some structural properties of the input program $\mathcal{P}$ in order to reduce the usage of mutex-locks in the main data structures, thus reducing the time spent in lock-contentions. This is obtained combining two strategies that aim at parallelizing two different steps of the instantiation process. In the first one, program modules are evaluated in parallel in such a way that race conditions can be avoided *without using mutex-locks*. In the second one, the instantiation of a single module is made parallel by concurrently instantiating its rules. Even in this case, the use of mutex-locks can be avoided by suitably parallelizing each iteration of the semi-naïve algorithm.

For the sake of clarity, these two techniques are described separately.

[11] Since no recursive atom occurs in the body of exit rules, a single call to *InstantiateRule* is sufficient for completely evaluating them.



$G^c_{\mathcal{P}}$

Fig. 3. Component graph.

4.1. Parallelizing the program instantiation

The parallel instantiation of the input program $\mathcal{P}$ is based on a pattern similar to the classical producer–consumers problem. A *manager* thread (acting as a producer) identifies the components of the dependency graph of $\mathcal{P}$ that can run in parallel, and delegates their instantiation to a number of *instantiator* threads (acting as consumers) that exploit the same *InstantiateComponent* function introduced in Section 3.

Once the general idea has been given, some formal definitions are introduced in order to detail the proposed technique. First of all, a new graph is defined, called *component graph*, whose nodes correspond to the strongly connected components of the dependency graph $G_{\mathcal{P}}$. Then, the definition is given of a partial ordering among the nodes of the component graph, which extends the partial order $\prec_s$ defined in Section 3. Please note how, with a small abuse of notation, strongly connected components of $G_{\mathcal{P}}$ and corresponding nodes of the component graph will be referred to indifferently.

Definition 3. Given a program $\mathcal{P}$, let $G_{\mathcal{P}}$ be the corresponding dependency graph. The *component graph* of $\mathcal{P}$ is a directed labelled graph $G^c_{\mathcal{P}} = \langle N, E, lab\rangle$, where N is a set of nodes, E is a set of arcs, and $lab : E \to \{+, -\}$ is a function assigning to each arc a label. N contains a node for each strongly connected component of $G_{\mathcal{P}}$; E contains an arc $e = (B, A)$, with $lab(e) = \text{“}+\text{”}$, if there is a rule r in $\mathcal{P}$ such that $q \in A$ occurs in the head of r and $p \in B$ occurs in a positive literal of the body of r; E contains an arc $e = (B, A)$, with $lab(e) = \text{“}-\text{”}$, if there is a rule r in $\mathcal{P}$ such that $q \in A$ occurs in the head of r and $p \in B$ occurs in a negative literal of the body of r, and there is no arc $e' = (B, A)$, with $lab(e') = \text{“}+\text{”}$.

Example 6. Given the program $\mathcal{P}$ of Example 2 its component graph is illustrated in Fig. 3.

Definition 4. For any pair of nodes A, B of $G^c_{\mathcal{P}}$, A *precedes* B in $G^c_{\mathcal{P}}$ (denoted $A \preceq_{\|} B$) if there is a *path* in $G^c_{\mathcal{P}}$ from A to B; and A *strictly precedes* B (denoted $A \prec_{\|} B$), if $A \preceq_{\|} B$ and $B \not\preceq_{\|} A$.

Example 7. Given the component graph illustrated in Fig. 3, it easy to see that the node $\{s\}$ precedes $\{p, t\}$, while $\{q\}$ strictly precedes $\{s\}$.

This ordering guarantees that a node A strictly precedes a node B if the program module corresponding to A has to be evaluated before the one corresponding to B.[12] It is now shown that this ordering is stronger than the one induced by the relation $\prec_s$ on the components of $G_{\mathcal{P}}$.

Lemma 8. *Let $\mathcal{P}$ be a program, and $G^c_{\mathcal{P}}$ and $G_{\mathcal{P}}$ be the component and dependency graphs of $\mathcal{P}$, respectively. Then, for any pair C_1 and C_2 of $G^c_{\mathcal{P}}$ nodes, if $C_1 \prec_s C_2$, then $C_1 \preceq_{\|} C_2$.*

Proof. Suppose that there are two components C_1, C_2 such that $C_1 \prec_s C_2$, while $C_1 \not\preceq_{\|} C_2$. If the latter holds, then there is no path in $G^c_{\mathcal{P}}$ from C_1 to C_2, and by definition of $\prec_s$, $C_1 \not\prec_s C_2$ holds, thus leading to a contradiction. □

[12] Note that the presence of negative arcs in $G^c_{\mathcal{P}}$ only determines a preference among the admissible orderings induced by the dependency graph, thus it does not affect the correctness of the overall instantiation process.

```
Procedure Parallel_Instantiate (P: Program; G^c_P: ComponentGraph;
                                var Π: GroundProgram);
begin
  var U: SetOfComponents; var D: SetOfComponents; var R: SetOfComponents;
  var S: SetOfAtoms; var C: SetOfPredicates;

  D = ∅; R = ∅; U = nodes(G^c_P)
  S = EDB(P); Π := ∅;
  while (U ≠ ∅)
    for all C ∈ U;
      if (canRun(C, U, R, G^c_P))
        begin
          R = R ∪ {C};
          Spawn(Instantiator, P, C, U, R, D, S, Π);
        end if
end

Procedure Instantiator (P: Program; C: Component; var U: SetOfComponents;
                        var R: SetOfComponents; var D: SetOfComponents;
                        var S: SetOfAtoms; var Π: GroundProgram);
begin
  InstantiateComponent(P, C, S, Π);
  D = D ∪ {C};
  R = R − {C};
  U = U − {C};
end
```

Fig. 4. The parallel instantiation procedures.

The parallel instantiation procedures exploiting this ordering can now be described. As previously pointed out, use is made of some threads: a *manager*, and a number of *instantiators* running the procedures *Parallel_Instantiate* and *Instantiator* reported in Fig. 4, respectively.

The *Parallel_Instantiate* procedure receives as input both a program $\mathcal{P}$ to be instantiated and its component graph $G^c_{\mathcal{P}}$; it outputs a set of ground rules Π, such that $ANS(\mathcal{P}) = ANS(\Pi \cup EDB(\mathcal{P}))$. First of all, the sets S, and Π are initialized as in the *Instantiate* procedure. Moreover, three new sets of components are created: $\mathcal{U}$ (which stands for *Undone*) represents the components of $\mathcal{P}$ that have still to be processed, $\mathcal{D}$ (which stands for *Done*) those that have already been instantiated, and $\mathcal{R}$ (which stands for *Running*) those currently being processed.

Initially, $\mathcal{D}$ and $\mathcal{R}$ are empty, while $\mathcal{U}$ contains all the nodes of $G^c_{\mathcal{P}}$. The manager checks, by means of function *canRun* described below, whether components in $\mathcal{U}$ can be instantiated. As soon as some C is processable,[13] it is added to $\mathcal{R}$, and a new instantiator thread is spawned in order to instantiate C by exploiting the *InstantiateComponent* function defined in Section 3. Once the instantiation of C has been completed, C is moved from $\mathcal{R}$ to $\mathcal{D}$, and deleted from $\mathcal{U}$. The manager thread goes on until all the components have been processed (i.e., $\mathcal{U} = \emptyset$). Obviously, an implementation of this algorithm must ensure mutually exclusive access to auxiliary control structures, like $\mathcal{U}$, $\mathcal{D}$ and $\mathcal{R}$, in order to obtain a correct execution.[14]

The function *canRun*, as the name suggests, checks whether a component C can be *safely evaluated* (i.e. without requiring "mutexes" in the main data structures) by exploiting the following definition:

Definition 5. Let $\mathcal{U}$ be the set of components which still have to be processed. We say that $C \in \mathcal{U}$ *can run* if $\forall A \in \mathcal{U}$ at least one of the following conditions holds:

[13] One may think of adopting some kind of strategy in order to determine the schedule that maximizes the parallelism among modules. However, the choice of avoiding the computation of an "a priori" schedule is reasonable. Indeed, the time taken by the instantiation of each module cannot be predicted; in addition, even supposing to have an estimation, the corresponding problem of finding the best scheduling is known to be intractable (see problem SS13 in [39]).

[14] Moreover, an efficient implementation of this algorithm must exploit and an implementation of the shared buffer and classical busy-waiting avoidance techniques [1], in order to call *canRun* only when it is really needed.

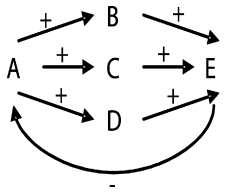


Fig. 5. An example of component graph.

(i) $C \prec_{\parallel}$;
(ii) $C \preceq_{\parallel}$ and $\nexists A' \in \mathcal{U}$ such that there is an arc $e = (A', C)$ of $G^c_{\mathcal{P}}$ with $lab(e) =$ "$+$", and $\forall K \in \mathcal{R}$, neither is there an arc $e' = (K, C)$ of $G^c_{\mathcal{P}}$, nor is there an arc $e'' = (C, K)$ of $G^c_{\mathcal{P}}$.
(iii) $C \not\preceq_{\parallel}$ and $A \not\preceq_{\parallel} C$ (i.e., C and A are not comparable).

Basically, this definition ensures that (i) a component C is not evaluated before all the components strictly preceding C (with respect to the partial ordering defined above) have been processed; and, (ii) if C appears in a cycle of $G^c_{\mathcal{P}}$ then it is selected only if it has no positive incoming edges and does not directly depend on some currently running component. Intuitively, the conditions of Definition 5 checked by function *canRun* guarantee the correctness, since they respect the dependencies of $G_{\mathcal{P}}$ (as for the standard instantiation algorithm defined in Section 3). Moreover, the conditions checked by *canRun* allow one to save resources and reduce the time spent in lock-contentions. Interestingly, condition (ii) of Definition 5 allows one to run in parallel even components appearing in cycles of $G^c_{\mathcal{P}}$ (i.e., components that are, somehow, interdependent).

Example 9. Consider the component graph of Fig. 5. All the nodes of the graph are involved in a cycle; thus, the evaluation of each component is somehow dependent on the evaluation of each other. However, condition (ii) of Definition 5 allows A to be selected for first evaluation. While A is running, no other component can be processed, because all conditions (i)–(iii) are violated for all of them. Once the instantiation of A has ended, component B can run, because it satisfies condition (ii). At the same time, by virtue of condition (ii), also components C and D can run. Then, component E can be evaluated only when the instantiations of all B, C and D have been completed.

It is worth noting that, thanks to Definition 5, procedure *ParallelInstantiate* never gets stuck in a deadlock situation (where no component can be safely evaluated). Indeed, whenever there is some "undone" component (i.e. $\mathcal{U} \neq \emptyset$) and no component is currently being processed (i.e. $\mathcal{R} = \emptyset$), at least one component, say C, *can run*. More explicitly, if there is a component $C' \in \mathcal{U}$ such that there is no component $C'' \in \mathcal{U}$ preceding C', then $C = C'$ can run (condition (i) or (iii)); on the contrary, if there is no such C' then there must be a cycle in $G^c_{\mathcal{P}}$ containing at least a negative arc (otherwise all nodes appearing in the cycle would collapse in a unique SCC); thus, let C be the component this negative arc enters into: C *can run* according to condition (ii).

The following demonstrates the correctness of the algorithm and outlines its main properties. To this end, some notation is introduced.

Let C be a component and $\mathcal{P}_C$ its corresponding module, then: I_C denotes the instantiator thread working on C (i.e. the one calling function *InstantiateComponent* with input C); $write(I_C)$ (respectively $read(I_C)$) denotes the set of predicates occurring in the head (respectively body) of rules in $\mathcal{P}_C$. Intuitively, $write(I_C)$ and $read(I_C)$ represent the sets of ground atoms created by I_C and read by I_C in procedure *Instantiate_Rule*, respectively.

It is easy to see that, for each pair I_X, I_Y of instantiator threads spawned by *Parallel_Instantiate* and working on components X and Y, respectively, $write(I_X) \cap read(I_Y) = \emptyset$. Indeed, according to Definition 5, if component X can run in parallel with component Y, then any predicate p occurring in the body of some rule in $\mathcal{P}_X$ cannot appear in the head of some rule of $\mathcal{P}_Y$, and vice versa. In case of nondisjunctive programs, it always holds also that $write(I_X) \cap write(I_Y) = \emptyset$. In general, for disjunctive programs, this does not always hold. Consider, for instance, the following example, where g is an *EDB* predicate:

$$a(X) \vee b(X) :- g(X, Y).$$
$$c(X) :- b(X),\ a(X).$$

$b(X) :- \mathit{not}\ c(X),\ g(X, X).$

The SCCs are $\{a\}$, $\{b\}$ and $\{c\}$, and, according to Definition 5, the program modules corresponding to $\{a\}$ and $\{b\}$ can be evaluated in parallel. Nevertheless, in this case, $\mathit{write}(I_{\{a\}}) \cap \mathit{write}(I_{\{b\}}) = \{b\}$, because the disjunctive rule belongs to both the modules corresponding to $\{a\}$ and $\{b\}$. However, since *InstantiateRule* checks whether a rule has already been taken into account (see Section 3), only the very first instantiator processing the rule actually instantiates it. Thus, at most one instantiator can write into the intersection of the writing sets.

It is worth pointing out that the presence of negative arcs in the component graph plays a central role for ensuring the above-mentioned condition. Indeed, if negative arcs are not considered, both components $\{b\}$ and $\{c\}$ can run in parallel. Thus, the instantiator working on $\{c\}$ may write in the portion of S, storing information about predicate c, which is also needed by the instantiator working on $\{b\}$.

Summarizing, an instantiator can never write where some other is reading/writing. This consideration has two noticeable consequences. On the one hand, if the actual implementation of set S of procedure *Parallel_Instantiate* is such that ground atoms are stored in different containers grouped by their predicate names, no "mutex" locks are needed to protect S, which is the shared structure mostly accessed and modified during the instantiation phase. On the other hand, whatever the schedule of the components evaluation followed by the *Parallel_Instantiate* algorithm, at least a serial schedule leading to the same output exists. Indeed, whatever the execution schedule, as soon as an instantiator thread is spawned, no other concurrently-running instantiator can modify its input data (and, thus its output). Note that considering negative arcs in the component graph is crucial to guarantee what has been discussed above. Indeed, two components share predicates (and thus may be in read/write conflict) also when one negatively depends on the other.[15]

The foregoing allows consideration of only serial executions for demonstrating the correctness of the algorithm.

Theorem 10. *Let* Π *and* $\Pi_{\|}$ *be the ground programs generated by algorithms Instantiate and Parallel_Instantiate, respectively. Then* $\mathit{ANS}(\Pi_{\|} \cup \mathit{EDB}(\mathcal{P})) = \mathit{ANS}(\Pi \cup \mathit{EDB}(\mathcal{P}))$.

Proof. It is now shown that each ordered sequence $\mathcal{S} = \langle C_1, \ldots, C_n \rangle$ of components, corresponding to a serial schedule of instantiator threads in procedure *Parallel_Instantiate*, is one of the admissible orders of execution followed by algorithm *Instantiate*.

It should be remembered that $\mathcal{S}$ is admissible for algorithm *Instantiate* if for each pair of component C_i, C_j in $\mathcal{S}$ such that $C_i \prec_s C_j$, $i < j$ holds.

Suppose that, by contradiction, $\mathcal{S}$ is not admissible. Then, there would be a pair of components C_i, C_j such that $C_i \prec_s C_j$ with $j < i$. Since it is assumed that the execution is sequential, when $C_j \in \mathcal{U}$ can run we have that $\mathcal{R} = \emptyset$ and, from Definition 5 (since $C_i \in \mathcal{U}$), one of these conditions holds: (i) $C_j \prec_{\|} C_i$; or (ii) $C_j \preceq_{\|} C_i$ and no positive arc of $G^c_{\mathcal{P}}$ enters C_j; or (iii) $C_i \not\preceq_{\|} C_j$ and $C_j \not\preceq_{\|} C_i$. Moreover, in accord with Lemma 8, $C_i \preceq_{\|} C_j$. Thus, conditions (i) and (iii) are clearly violated, and (ii) must hold (otherwise C_j cannot run). Therefore, there is a cycle involving C_j and C_i in $G^c_{\mathcal{P}}$, and the arc entering C_j is negative. Hence, $C_i \not\prec_s C_j$ which contradicts the hypotheses. □

4.2. Parallelizing the instantiation of a component

The previous section showed an algorithm for the instantiation of a program $\mathcal{P}$, which allows for the parallel evaluation of its modules. This algorithm is now improved by exploiting parallelism also for the instantiation of each single module. This can be done by the new procedure *Parallel_InstantiateComponent* (see Fig. 6), which, given a component C, instantiates the corresponding module $\mathcal{P}_C$ by processing its rules in parallel.

As *InstantiateComponent* (see Section 3), the new procedure takes as input the component C to be instantiated, and the set of significant atoms S; then, for each rule r in $\mathcal{P}_C$, computes its ground instances containing only atoms which can possibly be derived from $\mathcal{P}$, and adds them to Π. Moreover, the ground atoms occurring in the heads of newly generated ground rules are added to S.

[15] Actually, thanks to the presence of the negative arcs, it is also guaranteed that each disjunction-free stratified program is completely evaluated (i.e., the output of the algorithm is precisely the set of facts which are true in the unique answer set of the program). This property is inherited by the standard instantiation algorithm of the DLV system, which also takes care of them when components are serially scheduled. For more details about the standard instantiation procedure the interested reader may refer to [13,38].

```
Procedure Parallel_InstantiateComponent (P: Program; C: Component;
                                         var S: SetOfAtoms;
                                         var Π: GroundProgram)
begin
  var NS: Map⟨Rule, SetOfAtoms⟩;
  var ΔS: SetOfAtoms;
  ΔS := ∅;
  for each r ∈ P_C do NS(r) := ∅;
  for each r ∈ Exit(C, P); do
    I_r = Spawn(InstantiateRule, r, S, ΔS, NS(r), Π);
  for each r ∈ Exit(C, P); do
    join_with_thread(I_r);
  do
    ΔS := ⋃_{r∈P_C} NS(r);
    for each r ∈ P_C do NS(r) := ∅;
    for each r ∈ Recursive(C, P); do
      I_r = Spawn(InstantiateRule, r, S, ΔS, NS(r), Π);
    for each r ∈ Recursive(C, P); do
      join_with_thread(I_r);
    S := S ∪ ΔS;
  while ⋃_{r∈P_C} NS(r) ≠ ∅
end Procedure;
```

Fig. 6. The *Parallel_InstantiateComponent* procedure.

Parallel_InstantiateComponent first evaluates all the exit rules. To this end, one new thread for each exit rule, running procedure *InstantiateRule* (see Section 4.1), is spawned. Only once all the threads are done, recursive rules are processed. In this case, the execution of each single iteration of the semi naïve algorithm is parallelized. More in detail, at each iteration of the corresponding do-while loop (see Fig. 6) the following steps are performed:

- $\mathcal{N}S$ and ΔS are updated;
- for each recursive rule, a new thread is spawned running procedure *InstantiateRule* which receives as input S and ΔS;
- wait for all threads to terminate (by exploiting function *join_with_thread* of Fig. 6);[16]
- S, which constitutes the input for the next iteration, is updated.

Intuitively, *Parallel_InstantiateComponent*, when invoked with the same input as *InstantiateComponent* defined in Section 3, produces the same output.

Proposition 11. *Let S_s, Π_s (respectively S_p, Π_p) denote the sets S, Π after a call to procedure InstantiateComponent (respectively, Parallel_InstantiateComponent). Then $S_s = S_p$, and $\Pi_s = \Pi_p$.*

Proof. Procedures *InstantiateComponent* and *Parallel_InstantiateComponent* receive the set of significant atoms S and the set of ground rules Π, and update them, by only adding new elements. Locally, they define two new sets of atoms ΔS_s (respectively ΔS_p) and $\mathcal{N}S_s$ (respectively $\mathcal{N}S_p$), initially empty ($\mathcal{N}S_p$ is actually split into several sets, one for each rule). In both cases, when evaluating exit rules, each call to *InstantiateRule* is given the same input; in fact, S is trivially the same, and $\Delta S_s = \Delta S_p = \emptyset$. Since the inputs to *InstantiateRule* coincide in the two cases, also the outputs are the same. On the one hand, the union of the resulting sets from the split of $\mathcal{N}S_p$ in the parallel procedure trivially corresponds to $\mathcal{N}S_s$ of the standard one; on the other hand, the order in which rules are added to Π is irrelevant. The outcome is the same also after processing recursive rules. Indeed, the only difference in this case is that sets S and ΔS_s (respectively ΔS_p) change during the evaluation; however, they are updated by both the procedures at the end of each iteration and, in particular, *Parallel_InstantiateComponent* does that only when all the threads are done. □

[16] Following the POSIX threads style, function *join_with_thread*(T) causes the current thread to wait until the one identified by T finishes. In the for-loop this function is called for each spawned instantiator, acting as a barrier.

It is worth noting that this technique, similarly to the previous one, does not need mutex locks to protect S, since it is updated only from *Parallel_InstantiateComponent* at the end of each iteration when no other thread can work on it.

5. Experiments

In order to check the validity of the proposed techniques, they have been implemented into the grounding engine of the DLV system, and the resulting parallel instantiator was run on a collection of benchmark programs taken from different domains.

The system was built with gcc 4.1.2, dynamically linking the Posix Thread Library. Experiments were performed on a machine equipped with two Intel Xeon HT (single core) processors clocked at 3.60 GHz with 1 MB of Level 2 Cache and 3 GB of RAM, running Debian GNU Linux (kernel 2.4.27-2-686-smp).

The following instantiators were compared:

- ST: a single-threaded grounding engine;
- MT: a multi-threaded grounding engine.[17]

Since our techniques focus on instantiation, we report here the instantiation time rather than the total time needed in order to compute answer sets; in addition, the time spent before the grounding stage is obviously the same both for parallel and non-parallel version. In order to obtain more trustworthy results, each single experiment has been repeated five times, and we report both the average and standard deviation of the results.

In the following, we describe the benchmark problems, and finally report and discuss the results of the experiments.

5.1. Benchmark problems and data

We have considered several problems whose encodings are significantly hard to instantiate. In particular, depending on their origin, they can be grouped into two classes: "classical" and "real-world" problems. The first class contains *Hamiltonian path*, *Ramsey numbers*, and *n-Queens*; all of them are well-known combinatorial problems which have been widely used in the evaluation of the ASP systems (see for instance [13,21,40,41]). The second class, on the other hand, counts problems arising in practical applications of ASP, namely *Timetabling*, *Hilex*, *Food*, *Cristal*, *Decomp*, *Player* and *Insurance Workflow*.

We next provide a description of the problems; between parentheses are reported the names as they appear in the tables, if shortened for space reasons. We also provide the encodings of all the "classical" ones, omitting those of "real-world". Indeed, in such cases, encodings are long and very involved (some times automatically generated), and showing them here would have not given additional insight. Nevertheless, the full encodings, as well as the benchmark instances and the binaries used for experiments, are available at http://www.mat.unical.it/parallel/parallel_bench_07.zip.

Hamiltonian path (*HamPath*). This is a classical NP-complete problem in graph theory, and can be expressed as follows: given a directed graph $G = (V, E)$ and a node $a \in V$ of this graph, does there exist a path in G starting at a and passing through each node in V exactly once?

Suppose that the graph G is specified by using facts over predicates *node* (unary) and *arc* (binary), and the starting node a is specified by the predicate *start* (unary). Then, the following program $\mathcal{P}_{hp}$ solves the problem *Hampath*:

r_1: $inPath(X, Y) \vee outPath(X, Y) \text{ :- } start(X),\ arc(X, Y).$

r_2: $inPath(X, Y) \vee outPath(X, Y) \text{ :- } reached(X),\ arc(X, Y).$

r_3: $reached(X) \text{ :- } inPath(Y, X).$

[17] Actually, this implementation allows the user to set the maximum number of allowed concurrent instantiator threads. Here the results obtained are reported for when this parameter is set to the number of simultaneous (i.e., executed in a different processing unit) threads/processes allowed by the hardware (i.e. a two-processor Intel Xeon HT machine). Note that several possible values of this parameter were experimented, and, as expected, no significant differences in performance were obtained when the parameter exceeds this number.

c_1: :– $inPath(X, Y)$, $inPath(X, Y1)$, $Y <> Y1$.

c_2: :– $inPath(X, Y)$, $inPath(X1, Y)$, $X <> X1$.

c_3: :– $node(X)$, not $reached(X)$, not $start(X)$.

The two disjunctive rules r_1 and r_2 guess a subset S of the arcs to be in the path, while the rest of the program checks whether S constitutes a Hamiltonian path. Here, an auxiliary predicate *reached* is used, which is associated with the guessed predicate *inPath* using the last rule. In turn, through the second rule, the predicate *reached* influences the guess of *inPath*, which is made somehow inductively: initially, a guess on an arc leaving the starting node is made by the first rule, followed by repeated guesses of arcs leaving from reached nodes by the second rule, until all reached nodes have been handled. The first two constraints ensure that the set of arcs S selected by *inPath* meets the following requirements, which any Hamiltonian path must satisfy: (i) there must not be two arcs starting at the same node, and (ii) there must not be two arcs ending in the same node. The third constraint enforces that all nodes in the graph are reached from the starting node in the subgraph induced by S. It is easy to see that, given a set of facts $\mathcal{F}$ for *node*, *arc*, and *start*, the program $\mathcal{P}_{hp} \cup \mathcal{F}$ has an answer set if and only if the corresponding graph has a Hamiltonian path.

The instances for the experiments were generated using a tool by Patrik Simons (cf. [42]). For each problem size n 10 instances were generated, always assuming node 1 as the starting node. Sizes go from 100 to 10000 nodes.

Ramsey numbers (*Ramsey*). The Ramsey number $ramsey(k, m)$ is the least integer n such that, no matter how the edges of the complete undirected graph (clique) with n nodes are colored using two colors, say red and blue, there is a red clique with k nodes (a red k-clique) or a blue clique with m nodes (a blue m-clique). There are Ramsey numbers for all pairs of positive integers k and m [43]. Next a program $\mathcal{P}$ is shown that allows the decision about whether a given integer n is *not* the Ramsey number $ramsey(3, 4)$. By varying the input number n, $ramsey(3, 4)$ can be determined, as described below. Let $\mathcal{F}$ be the collection of facts for input predicates *node* and *edge* encoding a complete graph with n nodes. $\mathcal{P}$ is the following program:

r_1: $blue(X, Y) \vee red(X, Y)$:– $edge(X, Y)$.

c_1: :– $red(X, Y)$, $red(X, Z)$, $red(Y, Z)$.

c_2: :– $blue(X, Y)$, $blue(X, Z)$, $blue(Y, Z)$,
$blue(X, W)$, $blue(Y, W)$, $blue(Z, W)$.

Intuitively, the disjunctive rule r_1 guesses a color for each edge. The first constraint eliminates the colorings containing a red clique with 3 nodes, and the second constraint eliminates the colorings containing a blue clique with 4 nodes. The program $\mathcal{P} \cup \mathcal{F}$ has an answer set if and only if there is a coloring of the edges of the complete graph on n nodes containing no red clique of size 3 and no blue clique of size 4. Thus, if there is an answer set for a particular n, then n is *not* $ramsey(3, 4)$, that is, $n < ramsey(3, 4)$. On the other hand, if $\mathcal{P} \cup \mathcal{F}$ has no answer set, then $n \geqslant ramsey(3, 4)$. Thus, the smallest n such that no answer set is found is the Ramsey number $ramsey(3, 4)$.

For the experiments, the problem was considered of deciding, for varying k, m, and n, whether n is the Ramsey number $ramsey(k, m)$. Note that there is no a single, uniform encoding to solve all instances, rather there is one program for each instance. In particular, for checking that n is the Ramsey number $ramsey(k, m)$, the first constraint contains $\binom{k}{2}$ atoms with predicate *red* and the second constraint contains $\binom{m}{2}$ atoms with predicate *blue*.

***n*-Queens.** The 8-queens puzzle is the problem of putting eight chess queens on an 8×8 chessboard such that none of them is able to capture any other using the standard chess queen's moves (in chess, a queen can move as far as she pleases, horizontally, vertically, or diagonally). The colour of the queens is meaningless. A solution requires that no two queens share the same row, column, or diagonal. The n-queens puzzle is the more general problem of placing n queens on an $n \times n$ chessboard ($n \geqslant 4$).

If rows are represented by means of facts $row(1), \ldots, row(N)$, the 8-queens problem can be encoded as follows.

r_1: $q(X, 1) \vee q(X, 2) \vee q(X, 3) \vee q(X, 4)$
$\vee q(X, 5) \vee q(X, 6) \vee q(X, 7) \vee q(X, 8)$:– $row(X)$.

$$c_1\colon\ :\!-q(X,Y),\ q(X,Z),\ Y \neq Z.$$
$$c_2\colon\ :\!-q(X,Y),\ q(Z,Y),\ X \neq Z.$$
$$c_3\colon\ :\!-q(X_1,Y_1),\ q(X_2,Y_2),\ X_2 = X_1 + K,\ Y_1 = Y_2 + K,\ K > 0.$$
$$c_4\colon\ :\!-q(X_1,Y_1),\ q(X_2,Y_2),\ X_2 = X_1 + K,\ Y_2 = Y_1 + K,\ K > 0.$$

Queens are represented by atoms of the form $q(X, Y)$. Atom $q(X, Y)$ is true if a queen is placed on the chess board at row X and column Y. The disjunctive rule guesses the position of the queens; then constraints c_1 and c_2 assert that two queens cannot be on the same row or on the same column, respectively. Constraints c_3 and c_4 assert that two queens cannot be on the same diagonal (from top left to bottom right (constraint c_3) and from top right to bottom left (constraint c_4)). Note that, as for *Ramsey*, there is no a single, uniform encoding to solve all instances, rather there is one program for each instance. In particular, what really changes is the head of the first disjunctive rule, in order to meet the number of queens to be placed.

For the experimental evaluation different problems of sizes ranging from 25 to 37 were considered.

Timetabling. The problem was considered of determining a timetable for some university lectures that have to be given in a week to some groups of students. The timetable must respect a number of given constraints concerning availability of rooms, teachers, and other issues related to the overall organization of the lectures. Many instances were provided by the University of Calabria; they refer to different numbers of student groups.

Hilex. The problem consists in recognizing and extracting meaningful information from unstructured web documents. This is done by combining both syntactic and semantic information, through the use of domain ontologies. A preprocessor transforms the input documents into ASP facts, extraction rules are translated into ASP, and information extraction amounts to reasoning on an ASP program, which is executed by DLV. The single problem instance (containing 1526 predicates, 2682 rules, 1368 components) was provided by the company EXEURA s.r.l. [44].

Food. The problem here is to generate plans for repairing faulty workflows. That is, starting from a faulty workflow instance, the goal is to provide a completion of the workflow such that the output of the workflow is correct. Workflows may comprise many activities. Repair actions are compensation, (re)do and replacement of activities. Two distinct instances were provided related to two different workflows, one containing 63 predicates, 56 components and 116 rules, and another one having 78 predicates, 132 rules and 58 components.

Cristal. Cristal (Cooperative Repositories & Information System for Tracking Assembly Lifecycle) is a deductive databases application that involves complex knowledge manipulations. The main purpose is to manage the gathering of production data during the ongoing construction of the electromagnetic calorimeter of the compact muon solenoid, at the European Centre for Nuclear Research (CERN) [45]. The unique instance provided was used for the experiments, which features 16 predicates, 20 rules and 16 components.

Decomp. To circumvent the high complexity of query decompositions, in [46] a new concept of decomposition and its associated notion of width were introduced, which are called hypertree decomposition and hypertree-width, respectively. The benchmark problem presented here consists in computing a k-width complete hypertree decomposition of a query Q in a predicate P. A single instance of the problem was provided, containing 17 predicates, 20 rules and 19 components.

Player. A data integration problem. Given some tables containing discording data, find a repair where some key constraints are satisfied. The single problem instance used for these tests was originally defined within the EU project INFOMIX [5]. The instance contains 10 predicates, 13 rules and 12 components.

Table 1
Ramsey—Average grounding times (standard deviations within parentheses)

Problem	ST	MT	Gain (%)
Ramsey$(5, 5) \neq 21$	3.41 (0.00)	2.16 (0.64)	36.7
Ramsey$(5, 5) \neq 23$	4.41 (0.01)	2.53 (0.03)	42.6
Ramsey$(5, 5) \neq 25$	8.23 (0.01)	4.57 (0.14)	44.5
Ramsey$(5, 6) \neq 23$	13.16 (0.08)	11.40 (0.12)	13.4
Ramsey$(5, 6) \neq 25$	22.67 (0.12)	19.33 (0.11)	14.7
Ramsey$(5, 6) \neq 28$	43.84 (0.17)	38.24 (0.27)	12.8
Ramsey$(6, 6) \neq 23$	21.88 (0.09)	12.02 (0.34)	45.1
Ramsey$(6, 6) \neq 25$	37.13 (0.15)	20.64 (0.19)	44.4
Ramsey$(6, 6) \neq 28$	74.09 (0.13)	41.53 (0.21)	43.9
Ramsey$(6, 7) \neq 25$	84.50 (0.06)	70.30 (0.76)	16.8
Ramsey$(6, 7) \neq 28$	209.59 (0.54)	180.11 (2.01)	14.1
Ramsey$(6, 7) \neq 26$	370.17 (0.20)	322.66 (0.59)	12.8
Ramsey$(7, 7) \neq 28$	348.17 (3.03)	197.34 (1.15)	43.3
Ramsey$(7, 7) \neq 30$	– (–)	353.50 (3.41)	–
Ramsey$(7, 7) \neq 32$	– (–)	546.08 (5.21)	–

Table 2
n-Queens—Average grounding times (standard deviations within parentheses)

Problem	ST	MT	Gain (%)
25-*queens*	1.83 (0.00)	1.24 (0.13)	32.2
27-*queens*	2.52 (0.08)	1.60 (0.07)	36.5
29-*queens*	3.26 (0.00)	2.05 (0.04)	37.1
31-*queens*	4.24 (0.05)	2.66 (0.08)	37.3
33-*queens*	5.38 (0.03)	3.54 (0.17)	34.2
35-*queens*	6.76 (0.01)	4.29 (0.12)	36.5
37-*queens*	8.41 (0.08)	5.50 (0.25)	34.6

Insurance workflow (*Insurance*). Here the goal is to emulate, by means of an ASP program, the execution of a workflow, in which each step constitutes a transformation to be applied to some data (in order to query for and/or extract implicit knowledge). A single problem instance was provided by the company EXEURA s.r.l. [44]; it has been automatically generated by a software working on several American insurance data, and features 55 predicates, 53 rules and 52 components.

5.2. Experimental results and discussion

The results of the experimental activities on the benchmark problems presented above are summarized in Tables 1–4, and Fig. 7. The tables report, in seconds, average instantiation times and standard deviation for the two versions tested; the last column shows percentage gains. For every instance, we allowed a maximum running (real-)time of 600 seconds. In Fig. 7, the line corresponding to an instantiator stops whenever a problem instance was not solved within the allowed time limit; in the tables, this information is represented by means of the symbol '−'.

It can be observed that, overall, performance improvements of up to 45% were obtained, close to the theoretical maximum gain of 50% which one could expect from a bi-processor machine. More detailed considerations follow.

Concerning classical problems, the parallel system enjoys improvements up to 42% (*HamPath*, see Fig. 7), 45.1% (*Ramsey*, see Table 1), 37.3% (*n-queens*, see Table 2). As it is easy to see looking at the encodings, the program structures in this class of problem consist of only a few components; thus, the system takes advantage of the parallel rule evaluation within each module rather than of the parallel evaluation of the program modules. It is worth noting that, while evaluating *HamPath* and *n-queens*, MT shows an almost uniform behavior; this is not the case of the evaluation of *Ramsey*, where quite different improvements can be observed: a set of instances gain about 15%, while the rest about 43%. This is due to the nature of the encodings, which, as already pointed out in Section 5.1, are actually different as the integers m and n vary. The hard part of the computation concerns the two constraints c_1 and c_2. In particular, instances of *Ramsey*$(5, 5)$, *Ramsey*$(6, 6)$ and *Ramsey*$(7, 7)$ allow better performance, because the "size"

Table 3
Timetabling—Average grounding times (standard deviations within parentheses)

Problem	ST	MT	Gain (%)
Timetabling 3*c*	4.67 (0.01)	3.47 (0.10)	25.7
Timetabling 4*c*	6.89 (0.01)	4.99 (0.13)	27.6
Timetabling 5*c*	9.60 (0.07)	6.65 (0.08)	30.7
Timetabling 7*c*	15.92 (0.06)	10.61 (0.33)	33.4
Timetabling 9*c*	23.80 (0.20)	14.64 (0.42)	38.5
Timetabling 11*c*	34.34 (0.42)	20.20 (0.68)	41.2
Timetabling 13*c*	47.46 (1.11)	27.54 (0.60)	42.0
Timetabling 15*c*	62.81 (0.88)	35.46 (0.58)	43.5

Table 4
Real-world problems—Average grounding times
(standard deviations within parentheses)

Problem	ST	MT	Gain (%)
Hilex	67.93 (0.58)	47.87 (1.03)	29.5
Food-1	3.92 (0.07)	0.58 (0.02)	85.2
Food-2	146.55 (1.38)	131.93 (1.00)	10.0
Cristal	3.67 (0.01)	3.12 (0.04)	15.0
Decomp	8.10 (0.23)	6.02 (0.06)	25.7
Player	6.41 (0.08)	3.60 (0.22)	43.8
Insurance	106.78 (0.40)	75.43 (2.87)	29.4

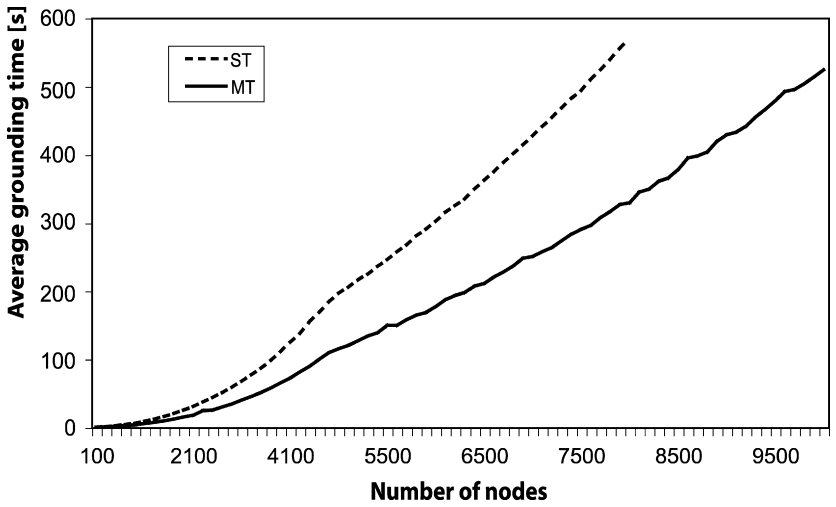


Fig. 7. Average grounding times for *HamPath*.

of the two constraints is the same, and thus two instantiators, one for each constraint, taking the same time, work in parallel for their whole job. This is not the case of *Ramsey*(5, 6) and *Ramsey*(6, 7), where one constraint takes more time than the other.

Moreover, these results show that, on this kind of problem, the parallel instantiator scales better than the non-parallel one. Indeed, looking at Table 1, we see that ST could not solve two out of three instances of *Ramsey*(7, 7) within the allowed time, while MT did. The same happens when looking at Fig. 7: when dealing with more than 8000 nodes, ST was not able to solve all instances of *HamPath* in time, while MT was successful with all of them.

Facing real-world problems, on the other hand, MT shows unsteady improvements when moving from a problem to another. In particular, there are two problems, *Food*-2 and *Cristal*, where despite their structures allowing the parallel evaluation of the program modules, the gain obtained is not as good as expected. This strange phenomenon was investigated, and with the use of suitable profiling tools, it was discovered that it is actually a technological issue,

concerning the standard STL [47] multithreaded memory allocator. In fact, DLV heavily relies on STL data structures; these exploit a memory allocator function that suffers from a dramatic performance degradation when linked against a multithreaded executable [48]. This problem negatively affects the results of all the experiments; sometimes, it may reduce the benefits provided by parallelism, as in the case of *Food*-2 and *Cristal*. This technological problem can be fixed, as indicated for instance in [48]; the implementation takes quite some time and it could be the subject of a future work.

Coming to the rest of the real-world problems, instead, the parallel system performed much more regularly. Here, in general, the programs structures consist of many components, usually containing few rules. Thus, the parallel evaluation of the different program modules pays more than the evaluation of the rules of each module in parallel. Improvements (see Table 4) here go from 25.7% (*Decomp*) to 85.2% (*Food*-1). The impressive speed-up for *Food*-1 can be explained with a different scheduling of the components performed by ST and MT. In particular, this instance is inconsistent (there is a constraint always violated) and both the executables stop the computation as soon as they recognize this fact. The scheduling performed by MT allows the identification of this situation before ST (MT evaluates the component containing constraints in parallel with other components whose evaluation is stopped immediately). A special note can be written for *Timetabling*. It is worth noting that improvements in the evaluation of its instances (see Table 3) get better as their size increases. This is due to the fact that there is a common part of the computation, which is exactly the same for the single-threaded and the multi-threaded systems; whose hardness is always the same, whatever the size of the instance. Thus, for small instances its weight is significant, and seems to overshadow the advantages produced by the parallel techniques; this effect disappears as the size of the instances grows, thus allowing improvements of up to 43.5% to be achieved (see again Table 3).

6. Related work

The exploitation of parallel techniques for evaluating ASP programs is, as matter of fact, not completely new [41, 49–51]. However, most of the contributions focuses on the model generation task, which is a distinct phase of ASP computation, carried out after the instantiation (whose parallelization is the subject of this work), and thus not directly related to this approach.

Nevertheless, some preliminary studies about the parallelization of the instantiation phase in ASP computation were carried out in [52], as one of the aspects of the attempt to introduce parallelism in non-monotonic reasoning systems. However, there are many differences with respect to this approach. First of all, the technique described in [52] was designed for the ASP instantiator Lparse [32,33], which, importantly, adopts an instantiation strategy quite different from the one exploited by DLV. More in detail, Lparse accepts logic programs respecting *domain restrictions*. This condition enforces each rule variable to occur in a positive body literal, called domain literal, which (i) is not mutually recursive with the head, and (ii) is not unstratified nor (transitively) depends on an unstratified literal (see [33] for details). This makes Lparse unable to accept many programs which are, on the contrary, accepted by DLV. On the other hand, these restrictions on the rules allow Lparse to employ a simple instantiation strategy (basically, a nested loop that scans the extensions of the domain predicates occurring in the body of r, and generates its ground instances accordingly). Such strategy may lead to the generation of several useless rules inasmuch as they may contain non-domain body literals not derivable by the program. The DLV instantiator, instead, incorporates several database optimization techniques, and builds the domains dynamically; consequently, the instantiation generated by DLV is generally a (sometimes very smaller) subset of that generated by Lparse [13]. Note that, since the ground program is the input of model generation, a computationally expensive [13] task, the size of the produced instantiation strongly affects the overall time required to compute the answer sets.

Recent comparison and benchmarks [13,53,54] showed that the more sophisticated instantiation technique is a strong point of the DLV system, as confirmed also by the results of the First Answer Set Programming System Competition [55].[18] Indeed, also thanks to the power of its instantiator, DLV was the first place winner in the MGS (Modeling, Grounding, Solving) category, also called the "Royal" category, where the overall performance of systems are measured.

The simple instantiation technique adopted by Lparse may allow for a more simple parallelization technique, as the one illustrated in [52]; there, some rules can be evaluated in parallel, without any control of dependencies

[18] See also http://asparagus.cs.uni-potsdam.de/contest/.

between predicates of the input program. Still, the parallelization schema of [52] is applicable only to the subset of the input program consisting of the rules which do not define domain predicates; the remaining rules are preliminarily processed, and their evaluation is not parallel. Note that, at a first glance, it seems that this approach could allow more parallelism, if compared to ours, but this is not the case. Indeed, also in this approach the amount of parallelization depends on the nature of the program at hand (even if in a different way with respect to our technique). More in detail, in the case of programs where most of the rules define domain predicates, parallelism may be notably limited. Consider the following program, where c and b are EDB predicates:

r_1: $h(X) :- c(X), \texttt{not}\ b(X).$

r_2: $f(X) :- b(X), \texttt{not}\ c(X).$

r_3: $a(Y) :- a(Y), \texttt{not}\ a(X),\ f(X),\ h(Y).$

It is easy to see that our instantiation technique allows for evaluating in parallel rules r_1 and r_2; on the contrary, the technique described in [52] processes sequentially the input program, since both rules r_1 and r_2, defining domain predicates, have to be evaluated before processing r_3.

Regarding system implementations, there are significant differences between the adopted technologies. This system was implemented by using POSIX threads APIs, and works in a shared memory architecture [1,2], while the one described in [52] is actually a Beowulf [56] cluster working in local memory. Beowulf is a multi-computer architecture which usually consists of one master server node, and one or more client nodes (which can be thought of as a CPU + memory package that can be plugged into the cluster) connected together via Ethernet or some other network. It is worth noting that our parallelization technique could also be exploited to implement this kind of distributed system.

The approach here is related to some extent to other works in the more general fields of logic programming and deductive databases [37,57–69]; however, the techniques are comparable to a limited extent to the one illustrated here. The literature is vast and varied, mainly centered on top-down evaluation; for a survey (centered on Prolog) the reader may refer to [68]. Basically, there are two major approaches (plus some hybrid). One relies on implicit exploitation of parallelism; this means that the parallelization occurs without any input from the programmer. The other, in contrast, concerns the design of high-level languages in order to support parallelism. These attempts start from considering the fact that the declarative appeal of logic programming unfortunately may be lost when mechanisms to explicitly control synchronization and communication (coordination) are introduced in logic programming languages. Thus, the main goal of this research efforts is to define language suitable for describing reactive process which provides both a declarative and an operational reading as in the logic paradigm (see for instance [59,70]). It is worth noting that this approach does not rely on a modification nor an extension of a language, and thus it may be included in the first group; it is fully transparent to the programmer, who still designs and implements her solution as before. The system is in charge of deciding when and where the computation may be parallelized, in order to improve its overall efficiency.

With respect to the rest of the literature, most of the techniques work on those so-called *And-*, *Or-* and *stream* parallelism, which try to transform certain sequential parts of the operational semantics into parallel operations [57, 60,63]. Also unification parallelism [71] (concerning unifications of arguments of a goal with arguments of a clause head with the same name and arity) is taken into account; nonetheless, it has not been a major focus of research. Or-parallelism arises when a subgoal can unify with the heads of more than one clause. In such a case the bodies of these clauses can be evaluated in parallel with each other: the exploration of the different alternatives in a choice point is parallelized. And-parallelism consists of the parallel solution of subgoals generated from literals in the same rule body: the resolution of distinct subgoals is parallelized (usually the literature distinguishes between independent and dependent parallelism, depending on the variable bindings which may or may not cause variable sharing among parallel tasks). Stream parallelism refers to the eager processing by a subgoal ("consumer") of an argument value, such as list, that is being constructed by another subgoal ("producer") [60].

The comparison of the techniques to ours is still not straightforward; nevertheless, some connections with the literature can still be found. The parallel technique exploited for processing exit rules within a single component (see Section 4) can be seen as a form of Or-parallelism; the process of non-exit rules (instantiated all together once all the exit rules are done) just requires to implement proper adjustments for additional synchronization (see again Section 4). In respect, many similarities can be found with to the work in [63], especially since the proposal considers Or-parallelism while exploiting semi-naïve techniques for the bottom-up evaluation of Datalog programs. However,

the independent rules are evaluated exploiting Or-parallelism, but the *stream-Or* parallelism herein described for processing dependent rules has no contact with our techniques. Basically, the information resulting from the evaluation of each rule is passed to the ones depending on it, as in a pipeline, while the evaluation processes are still running; our technique, on the contrary, allows the output of a process to be available only when it ends. In addition, the work in [63] does not consider disjunction.

Among the cited techniques, in the end, some apply to syntactically restricted classes of programs, and some require a heavy use of concurrency-control mechanisms; in addition, it is worth noting that none separates the program into components, as ours does; this allows us to minimize the use of "mutexes" in the main data structures, thus reducing the overhead introduced by the concurrency-control constructs.

7. Conclusion

In this paper a new strategy is proposed for introducing parallelism into the instantiation process of ASP programs. This strategy allows performance to be improved by taking advantage of the power of multiprocessor/multicore computers. The techniques involved wisely exploit some structural properties of the input ASP programs, and try to minimize the use of concurrency-control mechanisms in order to limit the "parallel overhead."

The proposed techniques were implemented into the grounding module of the DLV system, thus obtaining an experimental parallel ASP instantiator. A deep experimental analysis was then carried out, coming out with the confirmation that the strategy is indeed effective and causes the instantiation stage to achieve noticeable improvements, both in case of classical AI problems and real-world applications.

As far as future work is concerned, it is planned to exploit parallelism even for the instantiation of a single rule. This should enable further advantage to be taken of multiprocessing, especially in the case of program modules containing very few rules.

Acknowledgments

We would like to thank Nicola Leone and Wolfgang Faber for the useful comments, and Giovambattista Ianni for the fruitful discussions and the helpful hints. Moreover, we would like to thank the anonymous reviewers for their useful suggestions.

References

[1] W. Stallings, Operating Systems: Internals and Design Principles, 3rd ed., Prentice–Hall, Upper Saddle River, NJ, 1998.
[2] A.S. Tanenbaum, A.S. Woodhull, Operating Systems Design and Implementation, 3rd ed., Prentice–Hall, Upper Saddle River, NJ, 2005.
[3] M. Ruffolo, N. Leone, M. Manna, D. Sacca', A. Zavatto, Exploiting ASP for semantic information extraction, in: M. de Vos, A. Provetti (Eds.), Proceedings ASP05—Answer Set Programming: Advances in Theory and Implementation, Bath, UK, 2005, pp. 248–262.
[4] C. Cumbo, S. Iiritano, P. Rullo, Reasoning-based knowledge extraction for text classification, in: Proceedings of Discovery Science, 7th International Conference, Padova, Italy, 2004, pp. 380–387.
[5] N. Leone, G. Gottlob, R. Rosati, T. Eiter, W. Faber, M. Fink, G. Greco, G. Ianni, E. Kałka, D. Lembo, M. Lenzerini, V. Lio, B. Nowicki, M. Ruzzi, W. Staniszkis, G. Terracina, The INFOMIX system for advanced integration of incomplete and inconsistent data, in: Proceedings of the 24th ACM SIGMOD International Conference on Management of Data (SIGMOD 2005), ACM Press, Baltimore, MA, 2005, pp. 915–917.
[6] M. Gelfond, V. Lifschitz, Classical negation in logic programs and disjunctive databases, New Generation Comput. 9 (1991) 365–385.
[7] T. Eiter, G. Gottlob, H. Mannila, Disjunctive datalog, ACM Trans. Database Systems 22 (3) (1997) 364–418.
[8] T. Eiter, W. Faber, N. Leone, G. Pfeifer, Declarative problem-solving using the DLV system, in: J. Minker (Ed.), Logic-Based Artificial Intelligence, Kluwer Academic Publ., 2000, pp. 79–103.
[9] V. Lifschitz, Answer set planning, in: D.D. Schreye (Ed.), Proceedings of the 16th International Conference on Logic Programming (ICLP'99), The MIT Press, Las Cruces, NM, 1999, pp. 23–37.
[10] V.W. Marek, M. Truszczyński, Stable models and an alternative logic programming paradigm, in: K.R. Apt, V.W. Marek, M. Truszczyński, D.S. Warren (Eds.), The Logic Programming Paradigm—A 25-Year Perspective, Springer, 1999, pp. 375–398.
[11] C. Baral, Knowledge Representation, Reasoning and Declarative Problem Solving, Cambridge Univ. Press, 2003.
[12] M. Gelfond, N. Leone, Logic programming and knowledge representation—The A-Prolog perspective, Artificial Intelligence 138 (1–2) (2002) 3–38.
[13] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, F. Scarcello, The DLV system for knowledge representation and reasoning, ACM Trans. Comput. Log. 7 (3) (2006) 499–562.
[14] E. Dantsin, T. Eiter, G. Gottlob, A. Voronkov, Complexity and expressive power of logic programming, ACM Comput. Surveys 33 (3) (2001) 374–425.

[15] C. Bell, A. Nerode, R.T. Ng, V. Subrahmanian, Mixed integer programming methods for computing nonmonotonic deductive databases, J. ACM 41 (1994) 1178–1215.
[16] V. Subrahmanian, D. Nau, C. Vago, WFS + Branch and Bound = Stable Models, IEEE Trans. Knowledge and Data Engrg. 7 (3) (1995) 362–377.
[17] T. Janhunen, I. Niemelä, D. Seipel, P. Simons, J.-H. You, Unfolding partiality and disjunctions in stable model semantics, ACM Trans. Comput. Logic 7 (1) (2006) 1–37.
[18] T. Janhunen, I. Niemelä, Gnt—a solver for disjunctive logic programs, in: V. Lifschitz, I. Niemelä (Eds.), Proceedings of the 7th International Conference on Logic Programming and Non-Monotonic Reasoning (LPNMR-7), in: Lecture Notes in Artificial Intelligence, vol. 2923, Springer, 2004, pp. 331–335.
[19] Y. Lierler, Disjunctive answer set programming via satisfiability, in: C. Baral, G. Greco, N. Leone, G. Terracina (Eds.), Proceedings of Logic Programming and Nonmonotonic Reasoning—8th International Conference, LPNMR'05, Diamante, Italy, September 2005, in: Lecture Notes in Comput. Sci., vol. 3662, Springer, 2005, pp. 447–451.
[20] P. Simons, I. Niemelä, T. Soininen, Extending and implementing the stable model semantics, Artificial Intelligence 138 (2002) 181–234.
[21] I. Niemelä, P. Simons, T. Syrjänen, Smodels: A system for answer set programming, in: C. Baral, M. Truszczyński (Eds.), Proceedings of the 8th International Workshop on Non-Monotonic Reasoning (NMR'2000), Breckenridge, CO, 2000, online at http://xxx.lanl.gov/abs/cs/0003033v1.
[22] M. Gebser, B. Kaufmann, A. Neumann, T. Schaub, Conflict-driven answer set solving, in: Twentieth International Joint Conference on Artificial Intelligence (IJCAI-07), Morgan Kaufmann, 2007, pp. 386–392.
[23] F. Lin, Y. Zhao, ASSAT: Computing answer sets of a logic program by SAT solvers, Artificial Intelligence 157 (1–2) (2004) 115–137.
[24] Y. Lierler, M. Maratea, Cmodels-2: Sat-based answer set solver enhanced to non-tight programs, in: V. Lifschitz, I. Niemelä (Eds.), Proceedings of the 7th International Conference on Logic Programming and Non-Monotonic Reasoning (LPNMR-7), in: Lecture Notes in Artificial Intelligence, vol. 2923, Springer, 2004, pp. 346–350.
[25] C. Anger, K. Konczak, T. Linke, NoMoRe: A system for non-monotonic reasoning, in: T. Eiter, W. Faber, M. Truszczyński (Eds.), Proceedings of Logic Programming and Nonmonotonic Reasoning—6th International Conference (LPNMR'01), Vienna, Austria, September 2001, in: Lecture Notes in Artificial Intelligence, vol. 2173, Springer, 2001, pp. 406–410.
[26] C. Anger, M. Gebser, T. Linke, A. Neumann, T. Schaub, The nomore++ approach to answer set solving, in: G. Sutcliffe, A. Voronkov (Eds.), Logic for Programming, Artificial Intelligence, and Reasoning, 12th International Conference, LPAR 2005, in: Lecture Notes in Comput. Sci., vol. 3835, Springer, 2005, pp. 95–109.
[27] T. Eiter, N. Leone, C. Mateis, G. Pfeifer, F. Scarcello, A deductive system for nonmonotonic reasoning, in: J. Dix, U. Furbach, A. Nerode (Eds.), Proceedings of the 4th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'97), Dagstuhl, Germany, in: Lecture Notes in Artificial Intelligence, vol. 1265, Springer, 1997, pp. 363–374.
[28] M.Y. Vardi, Complexity of relational query languages, in: Proceedings of the 14th Symposium on Theory of Computation (STOC), 1982, pp. 137–146.
[29] W. Faber, N. Leone, C. Mateis, G. Pfeifer, Using database optimization techniques for nonmonotonic reasoning, in: INAP Organizing Committee (Ed.), Proceedings of the 7th International Workshop on Deductive Databases and Logic Programming (DDLP'99), Prolog Association of Japan, 1999, pp. 135–139.
[30] N. Leone, S. Perri, F. Scarcello, Improving ASP instantiators by join-ordering methods, in: T. Eiter, W. Faber, M. Truszczyński (Eds.), Logic Programming and Nonmonotonic Reasoning—6th International Conference (LPNMR'01), Vienna, Austria, in: Lecture Notes in Artificial Intelligence, vol. 2173, Springer, 2001, pp. 280–294.
[31] N. Leone, S. Perri, F. Scarcello, BackJumping techniques for rules instantiation in the DLV system, in: Proceedings of the 10th International Workshop on Non-monotonic Reasoning (NMR 2004), Whistler, BC, Canada, 2004, pp. 258–266.
[32] I. Niemelä, P. Simons, Smodels—An implementation of the stable model and well-founded semantics for normal logic programs, in: J. Dix, U. Furbach, A. Nerode (Eds.), Proceedings of the 4th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'97), Dagstuhl, Germany, in: Lecture Notes in Artificial Intelligence, vol. 1265, Springer, 1997, pp. 420–429.
[33] T. Syrjänen, Lparse 1.0 user's manual, http://www.tcs.hut.fi/Software/smodels/lparse.ps.gz, 2002.
[34] Y. Babovich, Cmodels homepage, http://www.cs.utexas.edu/users/tag/cmodels.html, since 2002.
[35] W. Faber, N. Leone, G. Pfeifer, Recursive aggregates in disjunctive logic programs: Semantics and complexity, in: J.J. Alferes, J. Leite (Eds.), Proceedings of the 9th European Conference on Artificial Intelligence (JELIA 2004), in: Lecture Notes in Artificial Intelligence, vol. 3229, Springer, 2004, pp. 200–212.
[36] T.C. Przymusinski, Stable semantics for disjunctive programs, New Generation Comput. 9 (1991) 401–424.
[37] J.D. Ullman, Principles of Database and Knowledge Base Systems, Comput. Sci. Press, 1989.
[38] W. Faber, N. Leone, S. Perri, G. Pfeifer, Efficient instantiation of disjunctive databases, Technical report DBAI-TR-2001-44, Institut für Informationssysteme, Technische Universität Wien, Austria, online at http://www.dbai.tuwien.ac.at/local/reports/dbai-tr-2001-44.pdf, November 2001.
[39] M.R. Garey, D.S. Johnson, Computers and Intractability, A Guide to the Theory of NP-Completeness, Freeman, 1979.
[40] D. East, M. Truszczyński, Propositional satisfiability in answer-set programming, in: Proceedings of Joint German/Austrian Conference on Artificial Intelligence (KI'2001), in: Lecture Notes in Artificial Intelligence, vol. 2174, Springer, 2001, pp. 138–153.
[41] R.A. Finkel, V.W. Marek, N. Moore, M. Truszczynski, Computing stable models in parallel, in: Answer Set Programming, Towards Efficient and Scalable Knowledge Representation and Reasoning, Proceedings of the 1st Internat. ASP'01 Workshop, Stanford, 2001, pp. 72–76.
[42] P. Simons, Extending and implementing the stable model semantics, PhD thesis, Helsinki University of Technology, Finland, 2000.
[43] S.P. Radziszowski, Small Ramsey numbers, Electron. J. Combin. 1 (2002), revision 9: July 15.
[44] Exeura s.r.l. homepage, http://www.exeura.it/.

[45] CRISTAL project homepage, http://proj-cristal.web.cern.ch/.
[46] G. Gottlob, N. Leone, F. Scarcello, Hypertree decompositions and tractable queries, J. Comput. System Sci. 64 (3) (2002) 579–627.
[47] A. Stepanov, M. Lee, The Standard Template Library, part of the evolving ANSI C++ standard, available at ftp://butler.hpl.hp.com/stl/, July 1995.
[48] E.D. Berger, K.S. McKinley, R.D. Blumofe, P.R. Wilson, Hoard: A scalable memory allocator for multithreaded applications, in: Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Cambridge, MA, 2000, pp. 117–128.
[49] J. Gressmann, T. Janhunen, R.E. Mercer, T. Schaub, S. Thiele, R. Tichy, Platypus: A platform for distributed answer set solving, in: Proceedings of Logic Programming and Nonmonotonic Reasoning, 8th International Conference (LPNMR), Diamante, Italy, 2005, pp. 227–239.
[50] E. Pontelli, O. El-Khatib, Exploiting vertical parallelism from answer set programs, in: Answer Set Programming, Towards Efficient and Scalable Knowledge Representation and Reasoning, Proceedings of the 1st Internat. ASP'01 Workshop, Stanford, CA, 2001, pp. 174–180.
[51] E. Pontelli, Experiments in parallel execution of answer set programs, in: Proceedings of 15th International Parallel and Distributed Processing Symposium (IPDPS), San Francisco, CA, 2001, p. 20.
[52] M. Balduccini, E. Pontelli, O. Elkhatib, H. Le, Issues in parallel execution of non-monotonic reasoning systems, Parallel Comput. 31 (6) (2005) 608–647.
[53] J. Dix, T. Eiter, M. Fink, A. Polleres, Y. Zhang, Monitoring agents using declarative planning, in: Proceedings of the 26th German Conference on Artificial Intelligence (KI2003), in: Lecture Notes in Comput. Sci., vol. 2821, Springer, 2003, pp. 646–660.
[54] O. Arieli, M. Denecker, B. Van Nuffelen, M. Bruynooghe, Database repair by signed formulae, in: D. Seipel, J.M. TurullTorres (Eds.), Foundations of Information and Knowledge Systems, Third International Symposium (FoIKS 2004), in: Lecture Notes in Comput. Sci., vol. 2942, Springer, 2004, pp. 14–30.
[55] M. Gebser, L. Liu, G. Namasivayam, A. Neumann, T. Schaub, M. Truszczyński, The first answer set programming system competition, in: C. Baral, G. Brewka, J. Schlipf (Eds.), Logic Programming and Nonmonotonic Reasoning—9th International Conference (LPNMR'07), Tempe, AZ, in: Lecture Notes in Comput. Sci., vol. 4483, Springer, 2007, pp. 3–17.
[56] The Beowulf Cluster Site, http://www.beowulf.org.
[57] K.L. Clark, S. Gregory, Parlog: Parallel programming in logic, ACM Trans. Program. Language Systems 8 (1) (1986) 1–49.
[58] O. Wolfson, A. Silberschatz, Distributed processing of logic programs, in: Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data, Chicago, IL, 1988, pp. 329–336.
[59] E.Y. Shapiro, The family of concurrent logic programming languages, ACM Comput. Surveys 21 (3) (1989) 413–510.
[60] R. Ramakrishnan, Parallelism in logic programs, Ann. Math. Artif. Intell. 3 (2–4) (1991) 295–330.
[61] O. Wolfson, Parallel evaluation of datalog programs by load sharing, J. Logic Program. 12 (3&4) (1992) 369–393.
[62] K. Inoue, M. Koshimura, R. Hasegawa, Embedding negation as failure into a model generation theorem prover, in: Proceedings of CADE-11, 11th International Conference on Automated Deduction, Saratoga Springs, NY, 1992, pp. 400–415.
[63] N. Leone, P. Restuccia, M. Romeo, P. Rullo, Expliciting parallelism in the semi-naive algorithm for the bottom-up evaluation of datalog programs, Database Technology 4 (4) (1993) 245–258.
[64] J.C. de Kergommeaux, P. Codognet, Parallel logic programming systems, ACM Comput. Surveys 26 (3) (1994) 295–336.
[65] R. Ramakrishnan, J.D. Ullman, A survey of deductive database systems, J. Logic Program. 23 (2) (1995) 125–149.
[66] W. Zhang, K. Wang, S.-C. Chau, Data partition and parallel evaluation of datalog programs, IEEE Trans. Knowledge Data Engrg. 7 (1) (1995) 163–176.
[67] M. Gaspari, A declarative language for parallel programming, Technical report UBLCS-00-X, Comput. Science Laboratory, University of Bologna, Italy, October 2000.
[68] G. Gupta, E. Pontelli, K.A.M. Ali, M. Carlsson, M.V. Hermenegildo, Parallel execution of Prolog programs: A survey, ACM Trans. Program. Language Syst. 23 (4) (2001) 472–602.
[69] Y. Wu, E. Pontelli, D. Ranjan, Computational issues in exploiting dependent and parallelism in logic programming: Leftness detection in dynamic search trees, in: Proceedings of Logic for Programming, Artificial Intelligence, and Reasoning, 12th International Conference (LPAR), Montego Bay, Jamaica, 2005, pp. 79–94.
[70] F. de Boer, C. Palamidessi, From concurrent logic programming to concurrent constraint programming, in: G. Levi (Ed.), Advances in Logic Programming Theory, Oxford Univ. Press, 1994, pp. 55–113.
[71] J. Barklund, Parallel unification, PhD thesis, Uppsala University, 1990.