Efficient Parallel ASP Instantiation via Dynamic Rewriting[⋆]

Simona Perri, Francesco Ricca, and Saverio Vescio

Dipartimento di Matematica, Università della Calabria, 87030 Rende, Italy
{perri,ricca,vescio}@mat.unical.it

Abstract. Answer Set Programming (ASP) is a powerful formalism for knowledge representation and reasoning. The computation of most ASP systems follows a two-phase approach: an instantiation (or grounding) phase generates a variable-free program which is then evaluated by propositional algorithms in the second phase. The instantiation process may be very expensive, especially for real-world problems, where huge input data are often to be dealt with.
A method that exploits the capabilities of multi-processor machines for improving instantiation performance has been recently proposed. This method, implemented in the grounding module of the ASP system DLV, proved to be effective especially when dealing with programs consisting of many rules.
In this paper, a dynamic rewriting of input rules is proposed that enhances the efficacy of the parallel evaluation also in the case of programs with very few rules. The effect of the technique is twofold: on the one hand, a kind of or-parallelism is induced by rewriting each rule at running time; on the other hand, the workload is dynamically distributed among processing units according to an heuristics.
Dynamic rewriting was implemented, and an experimental analysis was conducted that confirms the effectiveness of the technique. In particular, the new parallel implementation always outperforms the (sequential) DLV instantiator, and compared with the previous parallel method offers a very relevant gain especially in the case of programs with very few rules.

1 Introduction

In the last few years, multi-core/multi-processor architectures have become standard, thus making Symmetric MultiProcessing (SMP) [1] common also for entry-level systems and PCs. The principle behind SMP architectures is very simple: two or more identical processors connect to a single shared main memory, enabling simultaneous multithread execution. Such technology has been recently exploited with profit in the field of Answer Set Programming (ASP).

ASP is a declarative approach to programming proposed in the area of nonmonotonic reasoning and logic programming [2–7] which features a high declarative nature combined with a relatively high expressive power [8, 9]. There are nowadays a number of systems that support ASP and its variants [8, 10–17]. The kernel modules of ASP

[⋆] 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.”

systems work on a ground instantiation of the input program. Thus, an input program $\mathcal{P}$ first undergoes the so-called instantiation process, which produces a program $\mathcal{P}'$ semantically equivalent to $\mathcal{P}$, but not containing any variable. This phase is computationally very expensive (see [7, 9]); thus, having an efficient instantiation procedure is, in general, crucial for the performance of the entire ASP systems. Indeed, recent applications of ASP in different emerging areas (see e.g., [18–21]), have evidenced the practical need for faster and scalable ASP instantiators.

In [22] a technique for the parallel instantiation of ASP programs was proposed, allowing for the performance of instantiators to be improved by exploiting the power of multiprocessor computers. The technique takes advantage of some structural properties of input programs in order to reduce the usage of concurrency control mechanisms [1], and, thus, the so-called parallel overhead. The strategy focuses on two different aspects of the instantiation process: on the one hand, it examines the structure of the input program $\mathcal{P}$, splits it into modules (or sub-programs) 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 of rules within each module. This strategy has been implemented into the instantiator module of the ASP system DLV [8], thus obtaining a parallel ASP instantiator.

This parallel system proved to be effective especially in the instantiation of programs consisting of several rules with a large amount of input data [22]. However, it is not fully exploitable in case of programs with few rules. The reason for this behavior can be easily understood by considering the following disjunctive encoding for the well-known 3-Colorability problem:

$$(r) \quad col(X, red) \vee col(X, yellow) \vee col(X, green) \;\text{:-}\; node(X).$$
$$(c) \quad \text{:-}\; col(X, C),\; col(Y, C),\; edge(X, Y).$$

Predicates $node$ and $edge$ represent the input graph; rule (r) guesses the possible colorings of the graph, and the constraint (c) imposes that two adjacent nodes cannot have the same color.

In this case, the technique proceeds by first instantiating (r), thus computing the extension of col, and then, only once this is done, by processing the constraint (c). Thus, such encoding does not allow the existing technique to make the evaluation parallel at all. However, one may provide different encodings (with more rules) for the same problem, which are more amenable for the technique. In general, this would require the user to know *how* the evaluation process works, while writing a program: clearly, such a requirement is not desirable for a fully declarative system. Nevertheless, an automatic rewriting of the input program for an equivalent one, whose evaluation can be made more parallel, could make this optimization process transparent to the user.

In this paper, a dynamic rewriting of input rules is proposed that enhances the efficacy of the existing parallel evaluation technique, especially in the case of programs with very few rules. The basic idea is to rewrite input rules at execution time in order to induce a form of Or-parallelism [23–26]. This can be obtained, given a rule r, by "splitting" the extension of one single body predicate p of r in several parts. Each part is associated with a different temporary predicate; and, for each of those predicates, say p_i, a new rule, obtained by replacing p with p_i, is produced. The so-created rules will be instantiated in parallel in place of r; when they are done, a realign step gets rid of the new names in order to obtain the same output of the original algorithm.

However, the choice of the most convenient predicate to split is not trivial; indeed, a "bad" split might reduce or neutralize the benefits of parallelism, thus making the overall time consumed by the parallel evaluation not optimal (and, in some corner case, even worse than the time required to instantiate the original encoding). Thus, an heuristic has also been proposed to select the "best" predicate to split in order to minimize the parallel execution time. Summarizing, the contributions of this paper are the following:

- A technique is presented for rewriting rules of ASP programs in such a way that they can be evaluated in parallel; rules are rewritten at execution time, thus dynamically distributing the workload among processing units.
- An heuristic for selecting the most convenient way for rewriting a rule is proposed aiming at minimizing the parallel time.
- An implementation of the dynamic rewriting was done into the parallel version of the DLV instantiator.
- An experimental analysis was conducted for assessing the technique.

The results of the experiments show that the new parallel implementation always outperforms the (sequential) DLV instantiator, and, compared with the previous parallel method, offers a very relevant gain especially in case of programs with few rules.

2 Answer Set Programming

In this section, we briefly recall syntax and semantics of Answer Set Programming.

Syntax. A variable or a constant is a *term*. An *atom* is $a(t_1, ..., t_n)$, where a is a *predicate* of arity n and $t_1, ..., t_n$ are terms. A *literal* is either a *positive literal* p or a *negative literal* $\texttt{not}\ p$, where p is an atom. A *disjunctive rule* (*rule*, for short) r is a formula $a_1 \ \vee \ \cdots \ \vee \ a_n \ \text{:--}\ b_1, \cdots, b_k,\ \texttt{not}\ \ b_{k+1}, \cdots,\ \texttt{not}\ \ b_m.$ where $a_1, \cdots, a_n, b_1, \cdots, b_m$ are atoms and $n \geq 0$, $m \geq k \geq 0$. The disjunction $a_1 \ \vee \ \cdots \ \vee \ a_n$ is the *head* of r, while the conjunction $b_1, ..., b_k,\ \texttt{not}\ \ b_{k+1}, ...,\ \texttt{not}\ \ b_m$ is the *body* of r. A rule without head literals (i.e. $n = 0$) is usually referred to as an *integrity constraint*. If the body is empty (i.e. $k = m = 0$), it is called a *fact*.

$H(r)$ denotes the set $\{a_1, ..., a_n\}$ of the head atoms, and by $B(r)$ the set $\{b_1, ..., b_k, \texttt{not}\ b_{k+1}, \ldots, \texttt{not}\ b_m\}$ of the body literals. $B^+(r)$ (resp., $B^-(r)$) denotes the set of atoms occurring positively (resp., 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. 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 $EDB(\mathcal{P})$.

Semantics. Let $\mathcal{P}$ be a program. The *Herbrand Universe* and the *Herbrand Base* of $\mathcal{P}$ are defined in the standard way and denoted by $U_{\mathcal{P}}$ and $B_{\mathcal{P}}$, respectively.

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 $\mathcal{P}$.

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* (resp., *false*) w.r.t. I if $A \in I$ (resp., $A \notin I$). A ground negative literal `not` A is *true* w.r.t. I if A is false w.r.t. I; otherwise `not` A is false w.r.t. I. Let r be a ground rule in $ground(\mathcal{P})$. The head of r is *true* w.r.t. I if $H(r) \cap I \neq \emptyset$. The body of r is *true* w.r.t. I if all body literals of r are true w.r.t. I (i.e., $B^+(r) \subseteq I$ and $B^-(r) \cap I = \emptyset$) and is *false* w.r.t. I otherwise. The rule r is *satisfied* (or *true*) w.r.t. I if its head is true w.r.t. I or its body is false w.r.t. I.

A *model* for $\mathcal{P}$ is an interpretation M for $\mathcal{P}$ such that every rule $r \in ground(\mathcal{P})$ is true w.r.t. 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 $\mathrm{MM}(\mathcal{P})$.

Given a ground program $\mathcal{P}$ and an interpretation I, the *reduct* of $\mathcal{P}$ w.r.t. 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 w.r.t. I. Note that the above definition of reduct, proposed in [27], simplifies the original definition of Gelfond-Lifschitz (GL) transform [2], but is fully equivalent to the GL transform for the definition of answer sets [27].

Let I be an interpretation for a program $\mathcal{P}$. I is an *answer set* (or stable model) for $\mathcal{P}$ if $I \in \mathrm{MM}(\mathcal{P}^I)$ (i.e., I is a minimal model for the program $\mathcal{P}^I$) [28, 2]. The set of all answer sets for $\mathcal{P}$ is denoted by $ANS(\mathcal{P})$.

3 Parallel Instantiation of ASP programs

In this Section a sketchy description of the parallel instantiation algorithm of [22] is provided. A detailed discussion about this technique is out of the scope of this paper; for further insights we refer the reader to [22].

Given an input program $\mathcal{P}$, the algorithm efficiently generates a ground instantiation of the input program that has the same answer sets as the full one, but is much smaller in general. In order to generate a small ground program equivalent to $\mathcal{P}$, the parallel instantiator computes 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. This is done by taking into account some structural information of the input program, concerning the dependencies among IDB predicates.

In particular, each program $\mathcal{P}$ is associated with a graph, called the *Dependency Graph* of $\mathcal{P}$, which, intuitively, describes how predicates depend on each other. More in detail, given a program $\mathcal{P}$, 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)[1] C of $G_{\mathcal{P}}$, the

[1] 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.

set of rules defining all the predicates in C is called *module* of C and is denoted by $\mathcal{P}_c$. 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*. As an example, consider the following program $\mathcal{P}$, where a is an EDB predicate, and its dependency graph $G_{\mathcal{P}}$:

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

$$\begin{tikzcd}
p \arrow[d] & q \arrow[l] \arrow[d] \\
t \arrow[u, bend left] & s \arrow[l]
\end{tikzcd}$$

the strongly connected components of $G_{\mathcal{P}}$ are $\{s\}$, $\{q\}$ and $\{p, t\}$. They correspond to the three following modules: $\{\ p(X,Y) \vee s(Y) \,\text{:-}\, q(X), q(Y), not\ t(X,Y).\ \}$, $\{\ q(X) \,\text{:-}\, a(X).\ \}$, and $\{\ p(X,Y) \,\text{:-}\, q(X), t(X,Y).\ \ p(X,Y) \vee s(Y) \,\text{:-}\, q(X), q(Y), not\ t(X,Y).\ \ t(X,Y) \,\text{:-}\, p(X,Y), s(Y).\}$ Note that 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.

The dependency graph 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 if there is an arc from a predicate of A to a predicate of B; and, B *depends* on A if there is a path in the Dependency Graph from A to B.

Intuitively, this partial ordering guarantees that a node A precedes a node B if the program module corresponding to A has to be evaluated before the one of B. Moreover, if two components do not depend on each other, they can be evaluated in parallel.

The parallel instantiation algorithm exploits this partial ordering in order to both produce a small instantiation and identify modules that can be evaluated in parallel. It follows a pattern similar to the classical producer-consumers problem. A *manager* thread (acting as a producer) identifies the components of the dependency graph of the input program $\mathcal{P}$ that can run in parallel, and delegates their instantiation to a number of *instantiator* threads (acting as consumers).

The *Parallel_Instantiate* procedure, shown in Figure 1, acts as a manager. It receives as input both a program $\mathcal{P}$ to be instantiated and its Dependency Graph $G_{\mathcal{P}}$; and it outputs a set of ground rules Π, such that $ANS(\mathcal{P}) = ANS(\Pi \cup EDB(\mathcal{P}))$. First of all, the algorithm creates a new set of atoms S that will contain the subset of the Herbrand Base significant for the instantiation; more in detail, S will contain, for each predicate p in the program, the extension of p, that is, the set of all the ground atoms having the predicate name of p (significant for the instantiation).

Initially, $S = EDB(\mathcal{P})$, and $\Pi = \emptyset$. Then, the manager checks, whether some SCC C can be instantiated; in particular, it checks if there is some other component C' such that C depends on C' and C' has not been evaluated yet. As soon as a component C is processable, a new *ComponentInstantiator* thread is spawned for instantiating C.

Procedure *ComponentInstantiator*, in turn, takes as input, among the others, the component C to be instantiated and the set S; for each atom a belonging to C, and for each rule r defining a, it 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

```
**Procedure** *Parallel_Instantiate* ($\mathcal{P}$: Program; $G_{\mathcal{P}}$: DependencyGraph; **var** $\Pi$: GroundProgram)
**begin**
    **var** $S$: SetOfAtoms; **var** $C$: SetOfPredicates;
    $S = EDB(\mathcal{P})$; $\Pi := \emptyset$;
    **while** $G_{\mathcal{P}} \neq \emptyset$ **do**
        take a SCC $C$ from $G_{\mathcal{P}}$ that can run in parallel
        *Spawn*(*ComponentInstantiator*, $\mathcal{P}, C, S, \Pi, G_{\mathcal{P}}$)
    **end while**
**end**;
**Procedure** *ComponentInstantiator* ($\mathcal{P}$: Program; $C$: Component;**var** $S$: SetOfAtoms;
                                    **var** $\Pi$: GroundProgram; **var** $G_{\mathcal{P}}$: DependencyGraph)
**begin**
    **var** $\mathcal{N}S$: SetOfAtoms; **var** $\Delta S$: SetOfAtoms;
    $\Delta S := \emptyset; \mathcal{N}S := \emptyset$ ;
    **for each** $r \in Exit(C, \mathcal{P})$; **do**
        $\mathcal{I}_r$ = *Spawn* $(InstantiateRule, r, S, \Delta S, \mathcal{N}S, \Pi)$;
    **for each** $r \in Exit(C, \mathcal{P})$; **do**
        *join_with_thread*($\mathcal{I}_r$);
    **do**
        $\Delta S := \mathcal{N}S; \mathcal{N}S := \emptyset$ ;
        **for each** $r \in Recursive(C, \mathcal{P})$; **do**
            $\mathcal{I}_r$ = *Spawn* $(InstantiateRule, r, S, \Delta S, \mathcal{N}S, \Pi)$;
        **for each** $r \in Recursive(C, \mathcal{P})$; **do**
            *join_with_thread*($\mathcal{I}_r$);
        $S := S \cup \Delta S$;
    **while** $\mathcal{N}S \neq \emptyset$
    Remove $C$ from $G_{\mathcal{P}}$;
**end Procedure**;
**Procedure** *InstantiateRule* ($r$: rule; $S$: SetOfAtoms; $\Delta S$: SetOfAtoms
                                **var** $\mathcal{N}S$: SetOfAtoms**var** $\Pi$: GroundProgram)
/* Given $S$ and $\Delta S$, builds all the ground instances of $r$, adds them to $\Pi$, and add to $\mathcal{N}S$
   the head atoms of the newly generated ground rules. */
```

Fig. 1. The Parallel Instantiation Procedures.

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 rules.

It is worth noting that, exit rules are instantiated by a single call to *InstantiateRule*, whereas recursive ones are processed several times according to a semi-naïve evaluation technique [29], 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; the exit rules contained in the program module of C are evaluated and, in particular, one new thread for each exit rule, running procedure *InstantiateRule*, is spawned. Only once all the threads are done, recursive rules are processed (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, for each recursive rule, a new thread is spawned, running procedure *InstantiateRule*, which receives as input S and $\varDelta S$; when all threads terminate, $\varDelta S$ is added to S (since it has already been exploited). The evaluation stops whenever no new information has been derived (i.e. $\mathcal{N}S = \emptyset$). Eventually, component C is removed from $\varPi$.

Proposition 1. [22] Let $\mathcal{P}$ be an ASP program, and $\varPi$ be the ground program generated by the algorithm *Parallel_Instantiate*. Then $ANS(\mathcal{P}) = ANS(\varPi \cup EDB(\mathcal{P}))$ (i.e. $\mathcal{P}$ and $\varPi \cup EDB(\mathcal{P})$ have the same answer sets). □

4 Parallel Instantiation via Dynamic Rewriting

In this section, a rewriting technique is described that enhances the parallel instantiation algorithm of the previous Section.

Some Motivation. As already pointed out, there are problem encodings that do not allow the instantiation technique described in Section 3, to make the evaluation parallel at all; the following encoding of the 3-Colorability problem, is an example for that:

$(r)\;\; col(X, red) \,\vee\, col(X, yellow) \,\vee\, col(X, green) \text{ :- } node(X).$
$(c)\;\; \text{:- } col(X, C),\; col(Y, C),\; edge(X, Y).$

However, one may provide different encodings for the same problem, which are more amenable for the application of the technique. Oversimplifying, since each rule of the input program is processed by one processing unit, one may think of rewriting it into an equivalent program containing several rules. For instance, the following is a possible rewriting for the constraint (c), of the 3-Colorability encoding reported above:

$(c_1)\;\; \text{:- } col(X, C),\; col(Y, C),\; edge1(X, Y).$
$(c_2)\;\; \text{:- } col(X, C),\; col(Y, C),\; edge2(X, Y).$

The set of edges is *split up* into two subsets, represented by predicates $edge1$ and $edge2$. The evaluation of constraints (c_1) and (c_2) is equivalent to the evaluation of the original constraint c, modulo renaming, but the computation now can be carried out in parallel by two different processing units (instantiators).

This rewriting strategy can be straightforwardly extended for allowing more than two instantiators to work in parallel, and it can be generalized in order to deal with any program. However, there are different, sometimes many, ways to apply it. For instance, another possible encoding for 3-Colorability could be obtained by working on a literal whose predicate name is col, and by introducing new predicates $col_1 \ldots col_n$ (obtained by distributing the extension of col). Hereafter, with a small abuse of notation we indicate as extension of a literal l the extension of the predicate p corresponding to l (having the same name as l). Note that, differently from $edge$, col is not an EDB predicate (it occurs in the head of rule (r)); thus, in this case, a rewriting preserving the original semantics, would require to further modify the original program. Indeed, the extension of col is not known a-priori; thus, the split of col has to be induced by the split of the predicates it depends on by means of other rules. This may lead to an intricate rewriting of the entire program (not only rules to be split) and a possibly slower instantiation.

However, the extension of the predicate to be split is known during the instantiation when the rule is taken for evaluating it. Thus, if the rewriting is performed at execution time, a rule can be split without involving the entire program. Moreover, it can be easily automatized in order to be transparently applied. Note that, this strategy somewhat induces a form of *Or-parallelism* [23–26], which is here simulated via rewriting.

Dynamic Rewriting. The enhanced parallel instantiation algorithms are now described in detail that are based on the idea described above.

Procedure *ComponentInstantiator* used in the algorithm of Section 3 is replaced by a new one, called *ComponentInstantiator_Rew*, reported in Figure 2. It takes as input the component C to be instantiated and the set of significant atoms S; and for each atom a belonging to C, and for each rule r defining a, it computes the ground instances of r.

At the beginning, the new set of atoms ΔS and $\mathcal{N}S$ (which initially are empty) are created; then, exit rules are evaluated. More in detail, each of them is rewritten into a set of new exit rules which are added to $\mathcal{P}$. This is done by calling the procedure *Rewrite* which is detailed in the following. At this point, a thread running *InstantiateRule* is spawned for each exit rule. Only when all the threads are done, function *Realign* (i) restores S and ΔS by removing all the ground atoms inserted by procedure *Rewrite*, (ii) properly formats the output ground rules in such a way that "split predicates" do not appear; (iii) deletes from the program $\mathcal{P}$ all the rules containing split predicates and reintroduces the original ones. Now, as in *ComponentInstantiator*, recursive rules are evaluated according to a semi-naïve schema. Also in this case, rules are first rewritten and the output is "realigned". However, since recursive rules are processed several times, this strategy is applied at each iteration of the do-while loop.

One may think that, recursive rules could be "split" only once, but this choice is not correct in the general case. Indeed, if the split predicate is recursive, its extension may change at each iteration; hence, the distribution made during the rewriting step could not be sufficient to compute all the ground instances of the original rule. In addition, this choice has a relevant side-effect: at each iteration the workload is dynamically redistributed among instantiators, thus inducing a dynamic load balancing. Note that this feature intervenes just in case of the evaluation of recursive rules, which are often the most time consuming part of the computation.

Procedure *Rewrite* is now described in detail. It receives as input: the rule r to be "split", the sets S and ΔS containing the extensions of the body predicates, and the program $\mathcal{P}$. *Rewrite* first selects, according to an heuristics, a positive literal to split, say l, in the body of r; then, it replaces r in $\mathcal{P}$ by a set of rules r_i ($i = 1, \ldots, k$). Each r_i is obtained from r by substituting l with a new literal l_i having a fresh[2] new predicate name built by concatenating i to the name of l. This is done by function *SplitRules*, whereas procedure *Distribute* creates the extension of the new literals l_i ($i = 1, \ldots, k$) by uniformly distributing the extension of l (both S and ΔS are affected).

Concerning the selection of the literal to split, the choice has to be carefully made, since it may strongly affect the cost of the instantiation of rules; a good heuristics should minimize it. It is well-known that this cost strictly depends on the order of evaluation of body literals, since computing all the possible instantiations of a rule is equivalent to computing all the answers of a conjunctive query joining the extensions of literals of the rule body. However, the choice of the split literal may influence the time spent

[2] If a predicate with this name already exists, another string which does not appear elsewhere in the program is appended to the name.

on instantiating each split rule, whatever the join order. In order to help the intuition, suppose that the rule $r : h(X) : -a(X), b(X), c(X)$. has to be split in ten parts, and that the size of the extensions of a, b, and c are 10, 20, and 30, respectively. The following table reports the number of operations (i.e. comparisons) needed to instantiate a single split rule of r in the worst case, by varying the literal to split (on the columns) and by considering three different body orders (on the rows). Note that, any other order is equivalent to one of the table w.r.t. the number of operations.

order/split	split a	split b	split c
ABC	**620**	**620**	800
ACB	**630**	900	**630**
CBA	1200	**660**	**660**

Looking at the table, some considerations can be made. First of all, the order ABC is the most efficient; moreover, in each order, the number of operations is minimum when one of the first two literals is split. Note also that, the size of the extension alone is not a good discriminant for choosing the literal to split. Indeed, the table shows that splitting on c (which has the largest extension) is always a bad choice; whereas there is an order (CBA) in which, even if the split literal is the smallest (a) the number of operations is the highest. Similar considerations still hold if more precise estimations of costs are made. According to these considerations, the heuristics proposed here consists of selecting an optimal ordering and splitting the first literal in this order. Such heuristics tries, on the one hand, to minimize the overall (sequential) execution time; and, on the other hand, to distribute the workload in order to minimize the parallel execution time.

Since the ordering problem has already been investigated and an effective strategy [31] has already been successfully implemented in DLV, it was decided to adopt it.

This choice has also another important consequence: since all the factors the heuristics is based on are always already computed during the computation, its implementation does not introduce any overhead.

Proposition Let $\mathcal{P}$ be an ASP program, and Π be the ground program generated by the algorithm *Parallel_Instantiate* where procedure *ComponentInstantiator_Rew* is used instead of *ComponentInstantiator*; then $ANS(\mathcal{P}) = ANS(\Pi \cup EDB(\mathcal{P}))$.

Proof. (sketch) This follows from Proposition 1 if *ComponentInstantiator_Rew* produces the same output of *ComponentInstantiator* when invoked on the same input. First, observe that the ground instances of exit rules produced by the two procedures are the same modulo a renaming, that is performed by the realign step. Concerning recursive rules, their evaluation is obtained by applying several times the same algorithm used for the exit ones, where the output of each iteration is used as input for the next one. Thus, since the realign step is performed at the end of each iteration the thesis follows. $\square$

5 Experiments

In order to check the validity of the dynamic rewriting, it was implemented into the parallel grounding engine of [22]. The resulting system was compared with the previous one on a collection of benchmark programs taken from different domains.

Both problems whose encodings cannot be evaluated in parallel with the existing technique were considered, and problems where the old technique applies. All of them have already been used for assessing ASP instantiators performance ([8, 32, 33]).

```
**Procedure** *ComponentInstantiator_Rew* ( $\mathcal{P}$: Program; $C$: Component; **var** $S$: SetOfAtoms;
                                   **var** $\Pi$: GroundProgram, **var** $G_{\mathcal{P}}$: DependencyGraph)
**begin**
    **var** $\Delta S, \mathcal{N}S$: SetOfAtoms;
    $\Delta S := \emptyset; \mathcal{N}S := \emptyset$ ;
    **for each** $r \in Exit(C, \mathcal{P})$; **do**
        *Rewrite* $(r, S, \Delta S, \mathcal{P}$ );   // *this will add new exit rules*
    **for each** $r \in Exit(C, \mathcal{P})$; **do**
        $\mathcal{I}_r$ = *Spawn* $(InstantiateRule, r, S, \Delta S, \mathcal{N}S, \Pi)$;
    **for each** $r \in Exit(C, \mathcal{P})$; **do**
        *join_with_thread*($\mathcal{I}_r$);
    *Realign*($\Pi$,S,$\Delta S$);
    **do**
        $\Delta S := \mathcal{N}S; \mathcal{N}S := \emptyset$ ;
        **for each** $r \in Recursive(C, \mathcal{P})$; **do**
            *Rewrite* $(r, S, \Delta S, \mathcal{P}$ );   // *this will add new recursive rules*
        **for each** $r \in Recursive(C, \mathcal{P})$; **do**
            $\mathcal{I}_r$ = *Spawn* $(InstantiateRule, r, S, \Delta S, \mathcal{N}S, \Pi)$;
        **for each** $r \in Recursive(C, \mathcal{P})$; **do**
            *join_with_thread*($\mathcal{I}_r$);
        *Realign*($\Pi$,S,$\Delta S$,$\mathcal{P}$);
        $S := S \cup \Delta S$;
    **while** $\mathcal{N}S \neq \emptyset$
    Remove $C$ from $G_{\mathcal{P}}$;
**end Procedure**;

**Procedure** *Rewrite* ($r$: Rule; **var** $S$: SetOfAtoms; **var** $\Delta S$: SetOfAtoms;**var** $\mathcal{P}$: Program)
**begin**
    Select $l \in B(r)$;   //*according to an heuristics*
    $\mathcal{P}$ = $\mathcal{P} \cup$ *SplitRules*($r, l$);
    $\mathcal{P}$ = $\mathcal{P} \setminus \{r\}$;
    *Distribute*($l$,$S$,$\Delta S$);
**end Procedure**;

Program **Function** *SplitRules* ($r$: Rule; $l$: Literal)
/*
  *Given rule $r$, returns a program containing rules $r_i$ ($i = 1, \ldots, k$) obtained from $r$*
  *by replacing $l$ with a new literal $l_i$ having a fresh new name built by concatenating $i$*
  *to the name of $l$.*
*/

**Procedure** *Distribute* ($l$: Literal; **var** $S$: SetOfAtoms; **var** $\Delta S$: SetOfAtoms)
**begin**
    **for each** $a \in S$; **do**
        **if** *$a$ has the same name of $l$*
            **index** $s\_id$ = DetectSplit($a$);
            // *create atom $a_{s\_id}$ whose name is built by concatenating $s\_id$*
            // *to the name of $a$ and add it to $S$.*;
            $S = S \cup \{a_{s\_id}\}$;
**end Procedure**;

**Procedure** *Realign* ($\Pi$: GroundProgram; **var** $S$: SetOfAtoms; **var** $\Delta S$: SetOfAtoms;
               **var** $\mathcal{P}$: Program)
/*
  *For each ground rule $r \in \Pi$, if $B(r)$ contains a split literal $l_i$ replace its name with*
  *the original one; restore $S$ and $\Delta S$ by removing all the ground atoms inserted*
  *by function Distribute; replace rules originated by SplitRules with the original ones.*
*/
```

Fig. 2. The Parallel Instantiation Procedures enhanced by Dynamic Rewriting.

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 "Woodcrest" (quad core) processors clocked at 3.00GHz with 4 MB of Level 2 Cache and 4GB of RAM, running Debian GNU Linux 4.0.

The implementation allows the user for setting the number of splits as an input argument. For our experiments, this number was set to 8 which coincides with the number of available processors (if the extension of the split literal contains x instances where $x < 8$ then exactly x split rules are geenrated). Actually, an experimental analysis (not reported here for space reasons) confirmed that this fixed setting is optimal. The total time needed to instantiate the inputs was measured. In order to obtain more trustworthy results, each single experiment was repeated three times, and both the average and standard deviation of the results are reported. In the following, the benchmark problems are described, and finally, the results of the experiments are reported and discussed.

5.1 Benchmark Problems and Data

A brief description of the problems considered for the experiments follows. In order to meet space constraints, encodings are not presented but they are available at *http://www.mat.unical.it/parallel/parallel_bench_08.tar.gzip*. To help the understanding of the results some information is given on the number of rules of each program. About data, we considered for each problem three instances of increasing size.

3-Colorability. This well-known problem asks for an assignment of three colors to the nodes of a graph, in such a way that adjacent nodes always have different colors. The encoding of this problem consists of one rule and one constraint. Three simplex graphs were generated with the Stanford GraphBase library [34], by using the function $simplex(n, n, -2, 0, 0, 0, 0)$, ($n \in \{140, 150, 170\}$).

Reachability. Given a finite directed graph $G = (V, A)$, we want to compute all pairs of nodes $(a, b) \in V \times V$ such that b is reachable from a through a nonempty sequence of arcs in A. The encoding of this problem consists of one exit rule and a recursive one. Tree graphs were generated [35] having pair (number of levels, number of siblings): (12,2), (14,2), and (10,3), respectively.

Hamiltonian Path. 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? The encoding of this problem consists of several rules, one of these is recursive. Instances were generated, by using a tool by Patrik Simons (cf. [36]), having 5800, 6500 and 7200 nodes, respectively.

Player. A data integration problem [18]. Given some tables containing discording data, find a repair where some key constraints are satisfied. The encoding of this problem consists of several rules, and one constraint. The considered randomly generated databases have 32000, 39000, 45500 tuples.

n-Queens. The 8-queens puzzle is the problem of putting eight chess queens on an 8x8 chessboard such that none of them is able to capture any other using the standard chess queen's moves. The n-queens puzzle is the more general problem of placing n queens on an nxn chessboard ($n \geq 4$). The encoding consists of one rule and four constraints. Instances were considered having $n \in \{37, 39, 41\}$.

Problem	serial	no_split	split	split vs no_split
$3col_1$	60.76 (0.58)	60.79 (0.10)	9.24 (0.09)	657%
$3col_2$	92.00 (0.55)	91.97 (0.06)	13.28(0.10)	692%
$3col_3$	171.30 (0.29)	171.36 (0.20)	24.80 (0.29)	690%
$reach_1$	14.20 (0.03)	14.26 (0.02)	2.24 (0.08)	634%
$reach_2$	268.13 (0.31)	267.98 (0.60)	35.12(0.14)	763%
$reach_3$	802.35 (10.74)	805.47 (0.40)	101.51(0.62)	790%
$hampath_1$	229.43 (0.13)	141.02 (0.29)	33.35 (0.87)	423%
$hampath_2$	303.66 (0.92)	185.83 (0.56)	45.49 (0.35)	409%
$hampath_3$	377.85 (0.48)	231.89 (1.07)	57.73 (0.47)	402%
$queens_1$	4.79 (0.01)	2.29 (0.04)	0.76 (0.03)	301%
$queens_2$	5.89 (0.01)	2.79 (0.02)	0.93 (0.01)	300%
$queens_3$	7.16 (0.01)	3.38 (0.03)	1.08 (0.02)	312%
$player_1$	141.94 (2.02)	61.16 (0.08)	20.78 (0.26)	296%
$player_2$	289.72 (0.62)	126.48 (2.19)	41.95 (0.15)	301%
$player_3$	481.65 (11.23)	209.88 (1.15)	69.84 (0.16)	300%

Table 1. Effect of the enhanced technique - Average Times and Standard Deviation.

5.2 Experimental Results and Discussion

The performance of the compared systems is summarized in Table 1. In particular, the first three columns report the average instantiation times (and standard deviation) for the serial instantiation, the old parallel instantiator and the new one, respectively; the last column shows percentage gains given by the new technique w.r.t. the old one.

First of all, notice that for 3-Colorability and Reachability the old technique does not apply, thus the time reported in column "no_split" coincides with the serial execution time for those problems; conversely, the time required by the instances of Hamiltonian Path, n-Queens and Player already benefits of the presence of more than one processor.

It is worth noting that, where the old technique has no effect, the best performance is near to the theoretical maximum obtainable with eight processors. For example, in $reach_3$ the execution time changes from 805 seconds to 101 (i.e. gain about 790%).

The better performance obtained in the case of Reachability (w.r.t 3-Colorability) is due to the dynamic workload distribution made in case of recursive rules.

Looking at the remaining problems the picture is still very good: the introduction of the dynamic rewriting always allows to significantly improve performance. Indeed, for these problems, the old technique already allows for some interesting improvements w.r.t. the serial execution. But the combination of the two techniques is always the best performer, and reaches performance gains up to 700% w.r.t. the serial version.

In particular, when comparing the old parallel instantiator with the new one, gains range from 296% of $player_1$ to 423% of $hampath_1$. Note that, the better performance obtained for Hamiltonian Path is due (as for Reachability) to the dynamic workload distribution made in the presence of recursive rules. Indeed, Hamiltonian Path and Reachability are the only ones exploiting recursion among problems in Table 1.

Such good results may be further improved by applying more sophisticated technique for the distribution of the extension of the literal to split.

6 Related Work and Conclusions

In this paper a new strategy is proposed for increasing parallelism into the instantiation process of ASP programs. This strategy allows performance to be improved by performing a dynamic rewriting of the input program that, when combined with existing parallel instantiation techniques, naturally induces both a form of of *Or*-parallelism [23–26] and a dynamic load-balancing technique. The technique was implemented into the parallel DLV instantiator and an experimental analysis was conducted that confirmed the effectiveness of the technique. In particular, the new parallel implementation always outperforms the (sequential) DLV instantiator, and compared with the previous parallel method offers a very relevant gain especially in case of programs with very few rules.

Concerning related work, there are several studies about parallel techniques for the evaluation of ASP programs that focus on both the propositional (model search) phase [37–39], and the instantiation phase [40, 22]. About the latter group of proposals (which, evidently, is the only one strictly-related to this work), there are two distinct approaches: in [40] a parallelization technique was designed for the ASP instantiator Lparse [41]; whereas, in [22] the instantiator of the ASP system DLV was parallelized. Although the two approaches have several differences concerning both the input language and the exploited technology (clusters vs shared memory), both of them proceed by delegating the instantiation of rules of the program to different processing units. Thus, the method proposed in this paper, which was successfully implemented as an extension of the parallel DLV instantiator, may also be adapted to increase parallelism in case of the Lparse-based one (e.g. one might rewrite the input program, by splitting a domain predicate, just before launching the parallel computation). Regarding load-balancing, it is worth pointing out that, in [40] the distribution of work to processing units is statically determined at the beginning of the computation while, in the approach described in this paper, the work is distributed at running time.

Our work is also related to the efforts of parallelizing the evaluation of Datalog [42–45], dating back to 90's. In many of them, only restricted classes of Datalog programs are parallelized; whereas, the most general ones (reported in [43, 45]) are applicable to normal Datalog programs. Clearly, none of them consider the peculiarities of disjunctive programs and unstratified negation. More in detail, [43] provides the theoretical foundations for the so-called *copy and constrain* technique, whereas [45] enhances it in such a way that the communication overhead in distributed systems can be minimized.[3] The copy and constrain technique works as follows: rules are replicated with additional constraints attached to each copy; such constraints are generated by exploiting an hash function and allow for selecting a subset of the tuples. The obtained restricted rules are evaluated in parallel. Our technique shares the idea of splitting the instantiation of each rule, but has several differences that allow for obtaining an effective implementation. Indeed, in [43, 45] copied rules are generated and statically associated to instantiators according to an hash function which is independent from the current instance in input. Conversely, in our technique, the distribution of predicate extensions is performed dynamically, before assigning the rules to instantiators, by taking into account the "actual" predicate extensions. In this way, the non-trivial problem [45] of choosing an hash function that properly distributes the load is completely avoided in our approach. Moreover, the evaluation of conditions attached to the rule bodies during the instantiation phase would require to either modify the standard instantiation procedure (for efficiently selecting the tuples from the predicate extensions according to added constraints) or to

[3] Since the enhancements introduced in [45] are not relevant in our setting (i.e. SMP machines with *shared memory*), in the following we focus on [43]).

incur in a possible non negligible overhead due to their evaluation. As far as future work is concerned, it is planned to further study both load-balancing techniques and heuristics. A possibility is to extend to our framework dynamic load redistribution techniques like the one in [46].

References

1. Stallings, W.: Operating systems (3rd ed.): internals and design principles. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1998)
2. Gelfond, M., Lifschitz, V.: Classical Negation in Logic Programs and Disjunctive Databases. NGC **9** (1991) 365–385
3. Lifschitz, V.: Answer Set Planning. In Schreye, D.D., ed.: ICLP'99) 23–37
4. Marek, V.W., Truszczyński, M.: Stable Models and an Alternative Logic Programming Paradigm. In: The Logic Programming Paradigm-A 25-Year Perspective. (1999) 375–398
5. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. CUP (2003)
6. Gelfond, M., Leone, N.: Logic Programming and Knowledge Representation – the A-Prolog perspective . Artificial Intelligence **138**(1–2) (2002) 3–38
7. Eiter, T., Gottlob, G., Mannila, H.: Disjunctive Datalog. ACM TODS **22**(3) (1997) 364–418
8. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV System for Knowledge Representation and Reasoning. ACM TOCL **7**(3) (2006) 499–562
9. Dantsin, E., Eiter, T., Gottlob, G., Voronkov, A.: Complexity and Expressive Power of Logic Programming. ACM Computing Surveys **33**(3) (2001) 374–425
10. Janhunen, T., Niemelä, I.: Gnt - a solver for disjunctive logic programs. In: LPNMR-7. LNCS 2923, (2004) 331–335
11. Lierler, Y.: Disjunctive Answer Set Programming via Satisfiability. In: LPNMR'05. LNCS 3662, (2005) 447–451
12. Simons, P., Niemelä, I., Soininen, T.: Extending and Implementing the Stable Model Semantics. Artificial Intelligence **138** (2002) 181–234
13. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: Conflict-driven answer set solving. Proc. of IJCAI 2007, 386–392
14. Lin, F., Zhao, Y.: ASSAT: computing answer sets of a logic program by SAT solvers. Artificial Intelligence **157**(1–2) (2004) 115–137
15. Lierler, Y., Maratea, M.: Cmodels-2: SAT-based Answer Set Solver Enhanced to Non-tight Programs. In: LPNMR-7. LNCS 2923, (2004) 346–350
16. Anger, C., Konczak, K., Linke, T.: `NoMoRe`: A System for Non-Monotonic Reasoning. In: LPNMR'01. LNCS 2173, (2001) 406–410
17. Anger, C., Gebser, M., Linke, T., Neumann, A., Schaub, T.: The nomore++ Approach to Answer Set Solving. Proc. of LPAR 2005, 95–109
18. Leone, N., Gottlob, G., Rosati, R., Eiter, T., Faber, W., Fink, M., Greco, G., Ianni, G., Kałka, E., Lembo, D., Lenzerini, M., Lio, V., Nowicki, B., Ruzzi, M., Staniszkis, W., Terracina, G.: The INFOMIX System for Advanced Integration of Incomplete and Inconsistent Data. Proc. of ACM SIGMOD 2005, 915–917
19. Curia, R., Ettorre, M., Iiritano, S., Rullo, P.: Textual Document Per-Processing and Feature Extraction in OLEX. In: Proceedings of Data Mining 2005, Skiathos, Greece (2005)
20. Massacci, F.: Computer Aided Security Requirements Engineering with ASP Non-monotonic Reasoning, ASP and Constraints, Seminar N 05171. Dagstuhl Seminar (2005)
21. Ruffolo, M., Leone, N., Manna, M., Sacca', D., Zavatto, A.: Exploiting ASP for Semantic Information Extraction. Proc. of ASP05, Bath, UK (2005) 248–262

22. Calimeri, F., Perri, S., Ricca, F.: Experimenting with Parallelism for the Instantiation of ASP Programs. J. of Algorithms in Cognition, Informatics and Logic (2008) **63**(1-3) 34 - 54.
23. Leone, N., Restuccia, P., Romeo, M., Rullo, P.: Expliciting Parallelism in the Semi-Naive Algorithm for the Bottom-up Evaluation of Datalog Programs. Database Technology **4**(4) (1993) 245–158
24. Gupta, G., Pontelli, E., Ali, K.A.M., Carlsson, M., Hermenegildo, M.V.: Parallel execution of prolog programs: a survey. ACM Transactions on Programming Language Systems **23**(4) (2001) 472–602
25. de Kergommeaux, J.C., Codognet, P.: Parallel Logic Programming Systems. ACM Comput. Surv. **26**(3) (1994) 295–336
26. Gupta, G., Jayaraman, B.: Analysis of Or-Parallel Execution Models. ACM Transactions on Programming Language Systems **15**(4) (1993) 659–680
27. Faber, W., Leone, N., Pfeifer, G.: Recursive aggregates in disjunctive logic programs: Semantics and complexity. In: JELIA 2004. LNCS 3229, (2004) 200–212
28. Przymusinski, T.C.: Stable Semantics for Disjunctive Programs. NGC **9** (1991) 401–424
29. Ullman, J.D.: Principles of Database and Knowledge Base Systems. Computer Science Press (1989)
30. Garey, M.R., Johnson, D.S.: Computers and Intractability, A Guide to the Theory of NP-Completeness. W.H. Freeman and Company (1979)
31. Leone, N., Perri, S., Scarcello, F.: Improving ASP Instantiators by Join-Ordering Methods. Proc of LPNMR 2001, LNCS 2173, (2001) 280–294
32. Gebser, M., Liu, L., Namasivayam, G., Neumann, A., Schaub, T., Truszczyński, M.: The first answer set programming system competition. In: LPNMR 2007, LNCS 4483, 3–17
33. Perri, S., Scarcello, F., Catalano, G., Leone, N.: Enhancing DLV instantiator by backjumping techniques. AMAI **51**(2–4) (2007) 195–228.
34. Knuth, D.E.: The Stanford GraphBase : A Platform for Combinatorial Computing. ACM Press, New York (1994)
35. Giorgio, T., Leone, N., Vincenzino, L., Panetta, C.: Experimenting with recursive queries in database and logic programming systems. TPLP **7**, Cambridge University Press (2007) 1–37
36. Simons, P.: Extending and Implementing the Stable Model Semantics. PhD thesis, Helsinki University of Technology, Finland (2000)
37. Finkel, R.A., Marek, V.W., Moore, N., Truszczynski, M.: Computing stable models in parallel. In: Proc. of ASP'01 Workshop, Stanford (2001) 72–76
38. Gressmann, J., Janhunen, T., Mercer, R.E., Schaub, T., Thiele, S., Tichy, R.: Platypus: A Platform for Distributed Answer Set Solving. Proc. of LPNMR 2005, 227–239
39. Pontelli, E., El-Khatib, O.: Exploiting Vertical Parallelism from Answer Set Programs. In: Proc. of ASP'01 Workshop, Stanford (2001) 174–180
40. Balduccini, M., Pontelli, E., Elkhatib, O., Le, H.: Issues in parallel execution of non-monotonic reasoning systems. Parallel Computing **31**(6) (2005) 608–647
41. Niemelä, I., Simons, P.: Smodels – An Implementation of the Stable Model and Well-founded Semantics for Normal Logic Programs. In: LPNMR'97. LNCS 1265, 420–429
42. Wolfson, O., Silberschatz, A.: Distributed Processing of Logic Programs. Proc. of ACM SIGMOD 1998, 329–336
43. Wolfson, O., Ozeri, A.: A new paradigm for parallel and distributed rule-processing. In: ACM SIGMOD 1990, 133–142
44. Ganguly, S., Silberschatz, A., Tsur, S.: A Framework for the Parallel Processing of Datalog Queries. In: SIGMOD Conference 1990, Atlantic City, NJ, 23-25, 1990. (1990) 143–152
45. Zhang, W., Wang, K., Chau, S.C.: Data Partition and Parallel Evaluation of Datalog Programs. IEEE TKDE **7**(1) (1995) 163–176
46. Dewan, H.M., Stolfo, S.J., Hernández, M., Hwang, J.J.: Predictive dynamic load balancing of parallel and distributed rule and query processing. Proc. of ACM SIGMOD 1994, 277–288