Parallel Instantiation of ASP Programs*

F. Calimeri, S. Perri, and F. Ricca

Dipartimento di Matematica, Università della Calabria, 87036 Rende (CS), Italy
{calimeri, perri, ricca}@mat.unical.it

Abstract. Most of the Answer Set Programming (ASP) systems are endowed with an instantiation module, which generates a new program equivalent to the input one, but not containing variables. The instantiation process may be computationally expensive, especially for solving real-world problems, where large amounts of data have to be processed: this has been confirmed by recent applications of ASP in different emerging areas, such as knowledge management or information extraction/integration, where also scalability has been recognized as a crucial issue.
In this paper we present a new strategy for the parallel instantiation, that allows to improve both performances and scalability of ASP systems by exploiting the power of multiprocessor computers. Indeed, in the last few years, the microprocessors technologies have been moving to multi-core architectures; this makes the real Symmetric MultiProcessing (SMP) finally available even on non-dedicated machines, and paves the way to the development of more scalable softwares.
We have implemented such approach into the ASP system DLV, and carried out an experimental analysis which confirms the validity of the proposed strategy, expecially for real-world applications.

1 Introduction

ASP is a declarative approach to programming proposed in the area of nonmonotonic reasoning and logic programming [1–3]. The main advantage of ASP is its high expressiveness; unfortunately, this comes at the price of a high computational cost, which has made the implementation of efficient ASP systems a difficult task. Several efforts have been spent to this end, and, after some pioneering work [4, 5], a number of modern systems are now available. The most widespread ones are DLV [6], GnT [7], and Cmodels-3 [8]; many other support various fragments of the ASP language. The kernel modules of ASP systems operate on a ground instantiation of the input program, i.e. a program that does not contains any variable, but is semantically equivalent to the original input [9]. 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; thus, having a good and scalable instantiation procedure is, in general, a key feature of ASP systems.

* 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."

Many optimization techniques have been proposed for this purpose [10–12]; nevertheless, performances of instantiators are still not acceptable, especially in case of real-world problems, where the input data may be huge and also scalability is crucial. Indeed, the recent application of ASP in different emerging areas, such as knowledge management or information extraction/integration [13–15], have confirmed the practical need of more performant and scalable ASP systems.

Besides the other techniques, a technology that might be profitably exploited in the field of Answer Set Programming (ASP) is Symmetric MultiProcessing (SMP). SMP 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 took advantage of it. However, recently, technology has moved to multi-core/multi-processor architectures also for entry-level systems and PCs; this permitted to enjoy the benefits of parallel processing on a large scale. These benefits include better workload balances, enhanced performances, improved scalability, not only for systems that run many processes simultaneously, but also for single (i.e., multithreaded) applications.

In this paper we present a brand new strategy for the parallel instantiation, allowing to improve both performances and scalability of ASP systems by exploiting the power of multiprocessor computers. The proposed technique exploits some structural properties of the input program in order to detect subprograms of $\mathcal{P}$ that can be evaluated in parallel minimizing the usage of concurrency-control mechanisms, and thus minimizing the "parallel overhead". We implemented it in an experimental version of the DLV system, and performed several experiments which confirmed the effectiveness of our technique especially in the evaluation of real-world problem instances.

The remainder of the paper is structured as follows: in Section 2 we introduce syntax and semantics of ASP; in Section 3 we briefly describe the instantiation procedures of DLV system; in Section 4 we present our parallel instantiation algorithm; in Section 5 we report and discuss the results of the experiments carried out in order to evaluate the proposed technique; in Section 6, eventually, we look at related works and draw the conclusions.

2 Answer Set Programming

We next provide a formal definition of syntax and semantics of answer set programs.

2.1 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* not p, where p is an atom.

A *disjunctive rule* (*rule*, for short) r is a formula

$$a_1 \lor \cdots \lor a_n \text{ :- } b_1, \cdots, b_k, \texttt{ not } b_{k+1}, \cdots, \texttt{ not } b_m. \quad (1)$$

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*. 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*.

We denote by $H(r)$ 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. A `not` -free (resp., $\vee$-free) program is called *positive* (resp., *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.

2.2 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 $U_{\mathcal{P}}$. We denote by $ground(\mathcal{P})$ 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 $\texttt{not}\ A$ is *true* w.r.t. I if A is false w.r.t. I; otherwise $\texttt{not}\ A$ 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 program $\mathcal{P}$ and an interpretation I, the *Gelfond-Lifschitz (GL) transformation* of $\mathcal{P}$ w.r.t. I, denoted $\mathcal{P}^I$, is the set of positive rules

$$\mathcal{P}^I = \{\ a_1 \vee \cdots \vee a_n \text{ :- } b_1, \cdots, b_k \mid a_1 \vee \cdots \vee a_n \text{ :- } b_1, \cdots, b_k, \texttt{not}\ b_{k+1}, \cdots, \texttt{not}\ b_m \text{ is in ground}(\mathcal{P}) \text{ and } b_i \notin I, \text{ for all } k < i \leq m\}$$

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 positive program $\mathcal{P}^I$)*[16, 1]*. The set of all answer sets for $\mathcal{P}$ is denoted by $ANS(\mathcal{P})$.

3 The DLV Instantiator

In this section we provide a description of the DLV instantiator. Given an input program $\mathcal{P}$, it efficiently generates a ground instantiation that has the same answer sets as the

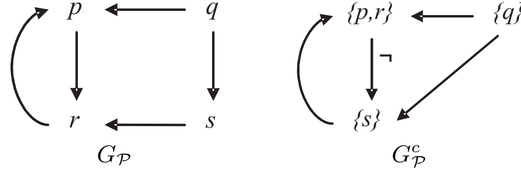

Fig. 1. Dependency and Component Graphs.

full one, but is much smaller in general [6]. Note that the size of the instantiation is a crucial aspect for the efficiency of ASP systems, since the answer set computation takes an exponential time 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[11]. This is obtained by taking into account some structural information of the input program, concerning the dependencies among IDB predicates.

We give now the definition the *Dependency Graph* of $\mathcal{P}$, which, intuitively, describes how predicates depend on each other.

Definition 1. 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 $G_{\mathcal{P}}$ naturally induces a partitioning of $\mathcal{P}$ into subprograms (also called *modules*) which allows 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. A *module* of $\mathcal{P}$ is the set of rules defining all the predicates contained in a particular maximal strongly connected component (SCC) of $G_{\mathcal{P}}$. Intuitively, a module includes (among others) all rules defining mutually dependent predicates.

Example 1. Consider the following program $\mathcal{P}$, where a is an EDB predicate:

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

The graph $G_{\mathcal{P}}$ is illustrated in Figure 1; moreover, the strongly connected components of $G_{\mathcal{P}}$ are $\{s\}$, $\{q\}$ and $\{p, r\}$. They correspond to the three following modules:

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

It is possible to single out an ordered sequence $C_1, \ldots, C_n$ of SCC components of $G_{\mathcal{P}}$ (which is not unique, in general) such that the evaluation of the program module

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

Fig. 2. The DLV Instantiation Procedure.

corresponding to component C_i depends only on the evaluation of the components C_j such that $i < j$ ($1 \leq i < n$, $1 < j \leq n$). Basically, this follows from the definition of SCC which corresponds to a maximal subset of mutually dependent predicates. Intuitively, this ordering allows one to evaluate 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.

We sketch now a description of the instantiation process based on this principle, omitting details on how a single module is grounded and providing a general idea of the whole process.

The procedure *Instantiate* shown in Figure 2 takes as input a program $\mathcal{P}$ to be instantiated and the dependency graph $G_{\mathcal{P}}$ and outputs a set of true atoms *T* and a set of ground rules containing only atoms which can possibly be derived from $\mathcal{P}$, such that $ANS(T \cup \Pi) = ANS(\mathcal{P})$. As previously pointed out, the input program $\mathcal{P}$ is partitioned in modules corresponding to the maximal strongly connected components of the dependency graph $G_{\mathcal{P}}$. Such modules are evaluated one at a time starting from those that do not depend on other components, according to the ordering induced by the dependency graph.

More in detail, the algorithm initially creates a new set of atoms I that will contain the subset of the Herbrand Base relevant for the instantiation. Initially, $T = EDB(\mathcal{P})$, $I = EDB(\mathcal{P})$, and $\Pi = \emptyset$. Then, a strongly connected component C, which has no incoming edge, is removed from $G_{\mathcal{P}}$, and the program module corresponding to C is evaluated by invoking *InstantiateComponent* which uses an improved version of the generalized semi-naive technique [17] for the evaluation of (recursive) rules.

Roughly, *InstantiateComponent* takes as input the component C to be instantiated, the sets *T* and I, 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 both the set *T* with the newly generated ground atoms already recognized as true, and the set I with the atoms occurring in the heads of the rules of Π. The algorithm runs on until all the components of $G_{\mathcal{P}}$ have been evaluated.

It can be shown that, given a program $\mathcal{P}$, the ground program $\Pi \cup T$ generated by the algorithm *Instantiate* is such that $\mathcal{P}$ and $\Pi \cup T$ have the same answer sets.

4 The Parallel Instantiation Procedure

In this Section we describe the new instantiation algorithm that computes a ground version of a given program $\mathcal{P}$ by exploiting parallelism. It takes advantage of some structural properties of the input program $\mathcal{P}$ in order to detect the modules that can be evaluated in parallel *without using "mutexes"* in the main data structures.

Roughly, 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 be 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, we introduce some formal definition in order to detail the proposed technique. First of all, we define a new graph, called *Component Graph*, whose nodes correspond to the strongly connected components of the Dependency Graph $G_{\mathcal{P}}$. Then, we give the definition of a partial ordering among the nodes of $G^c_{\mathcal{P}}$. Please note as, with a small abuse of notation, we will indifferently refer to components of $G_{\mathcal{P}}$ and corresponding nodes of the Component Graph.

Definition 2. 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 $lab : E \rightarrow \{+,-\}$ is a function assigning to each arc a label. N contains a node for each (maximal) strongly connected component of $G_{\mathcal{P}}$; E contains an arc $e = (B, A)$ 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 (resp., negative) literal of the body of r; $lab(e) =$"+" (resp., $lab(e) =$"-").

Definition 3. For any pair of nodes A, B of $G^c_{\mathcal{P}}$, A *precedes* B (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 $B \npreceq A$.

Example 2. Consider the program $\mathcal{P}$ of Example 1. The component graph of $\mathcal{P}$ is illustrated in Figure 1. It easy to see that the node $\{p, r\}$ precedes $\{s\}$, while $\{q\}$ strictly precedes $\{s\}$.

Basically, 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.[1]

We are now ready to describe the parallel instantiation procedures exploiting this ordering. As previously pointed out, we make use of some threads: a *manager*, and a number of *instantiators* running the procedures $Manager$ and $Instantiator$ reported in Figure 3, respectively.

[1] 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 Manager (𝒫: Program; G^c_𝒫: ComponentGraph;
                   var T: SetOfAtoms; var Π: GroundProgram);
begin
    var 𝒰:SetOfComponents ; var 𝒟:SetOfComponents; var ℛ:SetOfComponents;
    var I: SetOfAtoms; var C: SetOfPredicates;

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

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

Fig. 3. The Parallel Instantiation Procedures.

The $Manager$ procedure takes as input both a program $\mathcal{P}$ to be instantiated and its Component Graph $G^c_{\mathcal{P}}$; it outputs both a set T of true atoms and a set of ground rules Π, such that $ANS(T \cup \Pi) = ANS(\mathcal{P})$.

First of all, the sets T, I, and Π are initialized like in the standard DLV Instantiator. 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 $canBeRun$ described below, whether components in $\mathcal{U}$ can be instantiated. As soon as some C is processable, 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., $\mathcal{U} = \emptyset$).

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

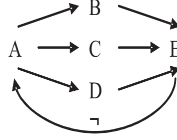

Fig. 4. An example of Component Graph.

Definition 4. Let $\mathcal{U}$ be the set of components which still have to be processed. We say that $C \in U$ *can be run* if $\forall A \in \mathcal{U}$ at least one of the following conditions holds:

(i) $C \prec A$;
(ii) $C \preceq A$ and $\nexists A' \in \mathcal{U}$ s.t. exists an arc $e = (A', C)$ of $G^c_{\mathcal{P}}$ with $lab(e) =$"+", and $\forall K \in \mathcal{R}$, do not exist arcs e', e'' of $G^c_{\mathcal{P}}$ such that $e' = (R, C)$ and $e'' = (C, R)$.

Basically, this definition ensures that (i) a component C is not evaluated before all the components strictly preceding C (w.r.t. 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. The two conditions of Definition 4 checked by function *canBeRun* guarantee the correctness, since they respect the dependencies of $G_{\mathcal{P}}$ (as for the standard instantiation algorithm defined in Section 3).

Moreover, *canBeRun* singles out components that can be evaluated in parallel without using "mutex" locks in the data structures that implement the sets T and I; this allows us to save resources and reduce the time spent in lock-contentions. It is easy to see that two components C_1 and C_2 are "selected" only if any predicate p occurring in the body of some rule of C_1 does not appear in the head of some rule of C_2, and vice versa. If the data structures implementing the sets T and I properly store the ground atoms in different containers (i.e., one for each predicate name, as in DLV), then no "mutex" lock is needed to protect them: during the evaluation of a rule, an instantiator thread may write in the container of an atom a only when a rule defining a is processed; thus, it will never write in a location being accessed by another instantiator.

Interestingly, condition (ii) of Definition 4 allows one to run in parallel even components appearing in cycles of $G^c_{\mathcal{P}}$ (i.e., components that are, somehow, interdependent). This can be illustrated by the following example:

Example 3. Consider the Component Graph of Figure 4. 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 4 allows to select A to be evaluated first. While A is running, no other component can be processed, because both conditions (i) and (ii) are violated for all of them. Once the instantiation of A has ended, component B can be run, because it satisfies condition (ii). At the same time, by virtue of condition (ii), also components C and D can be run. Then, component E can be evaluated only when the instantiations of all B, C and D have been completed.

It is important noting that the actual implementation is more involved, but only because of technical reasons. First of all, the auxiliary control structures (like $\mathcal{U}$, $\mathcal{D}$ and $\mathcal{R}$) are properly protected by "mutex" locks, and the busy waiting is properly avoided.

Finally, we also have to deal with additional structures which allow the user to set the maximum number of instantiator threads. We do not believe that these technical issues may help to get a better insight, but they are rather lengthy in description; for this reason, we refrain from discussing them here.

5 Experiments

In order to consistently evaluate the parallel grounding technique described in Section 4, we have implemented it as an extension of the DLV system, and performed some experiments. We took into account several problems belonging to different applications, ranging from classical ASP benchmarks to "real-word" applications.

We have compared the prototype with the official DLV [18] release[2] on which it is based. In addition, we considered the maximum number of concurrent instantiator threads as a parameter; thus, we deal with the following versions of DLV:

- **dl.release**: the original DLV system release without parallel grounding;
- **dl.th**X: the modified DLV system with X indipendent working threads (X ranges from 1 to 4).

All the binaries have been built with GCC 3.4 (the same used to build the original DLV release), statically linking the Posix Thread Library.

Experiments have been performed on a machine equipped with two Intel Xeon HT (single core) CPUs clocked at 3.60GHz with 1 MB of Level 2 Cache and 3GB of RAM, running Debian GNU Linux (kernel 2.4.27-2-686-smp). This machine is capable of *simultaneously* run (i.e., each thread executed on a different processing unit) at most 4 threads; with more, the system performs poorly because of the preemptive thread scheduling overhead. This has ben confirmed by the experiments; thus, we decided to omit here the results obtained by allowing more than 4 concurrent instantiator threads. Limiting to 4 concurrent instantiator threads does not eliminate the effects of preemption, but, reasonably, they become negligible; indeed, we ran the tests on an "unloaded" machine, and the operating system always tries to schedule active threads on free CPUs.

Time measurements have been performed by means of the `time` command shipped with the above cited version of Debian GNU Linux. Unfortunately, we could not consider the total CPU times[3], because, in case of multi-threaded applications, they result as the sum of the time spent by the process on *each* processing unit (e.g., when a process fully exploits simultaneously two processors for 5 minutes, the total reported CPU time is 10 minutes). We decided to overcome the problem by considering the so called *real* time, based on the system wall-clock time. Obviously, this measure is less accurate than the total CPU time, since it unavoidably includes the time spent by other processes in the system (even by unrelated operating system routines). In order to obtain more reliable information, we have repeated each test three times, and provide here both average and standard deviation of the results.

[2] Official DLV release, July 14th 2006.

[3] The sum of *user* and *system* time; we refer the reader to *time* manual pages for a detailed description of these quantities [19].

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

5.1 Benchmark Programs

We provide here a brief description of the problems considered for the experiments. In order to meet the space constraints, we refrain from showing the encodings (consider that some are automatically generated, and are very long and involved). However, they are available at `http://www.mat.unical.it/parallel/cilc_inst.tar.gz`.

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.

Ancestor. Given a *parent* relationship over a set of persons, find the genealogy tree of each one. It is a classical deductive database problem exploiting recursive rules.

Knowledge Discovery. Given an ontology and a text document, an ASP program classifies the document w.r.t. the ontology. Basically, the goal is to associate the document contents to one or more concepts in the given ontology: a document is associated to the concepts it deals with. Problems have been provided by the company EXEURA s.r.l. [20].

Player. A data integration problem. Given some tables containing discording data, find a repair where some key constraints are satisfied. The problem was originally defined within the EU project INFOMIX [15].

Hypertree Decomposition. Compute a k-width complete hypertree decomposition [21] of a given query Q in a given predicate P.

ETL Workflow. In general, ETL stands for Extraction Transformation and Loading. 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). We considered the encoding of three different steps, automatically generated by a software working on some american insurance data. Problems have been provided by the company EXEURA s.r.l. [20].

Cristal. A deductive databases application that involves complex knowledge manipulations. The application was originally developed at CERN [22].

Timetabling. A real timetable problem from the faculty of Science of the University of Calabria. We have considered for the evaluation the programs that the faculty exploited for two different academic years.

The above-mentioned problems can be roughly divided into two classes, with respect to their structure. One contains problems having very “dense” dependency graphs (meaning that there are only few components), like 3-Colorability or Ancestor. The other contains problems featuring several rules belonging to independent components of the dependency graph, like ETLs or Timetablings. Therefore, the parallel instantiation of the first might only moderately be profitable, while the latter should be easier grounded in parallel; having both classes of problems allows one to get a sharpen picture of the behavior of our prototype.

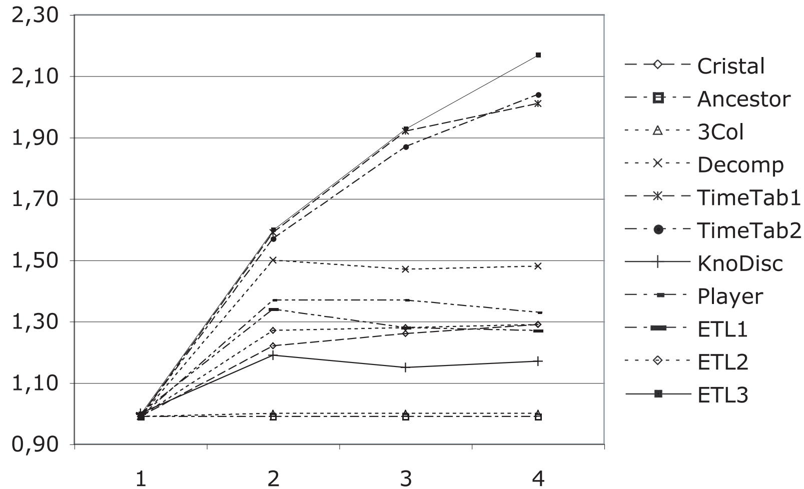

Fig. 5. Average CPU usage.

5.2 Results

The results of the experimental activities are summarized in Table 1 and Table 2, showing average total execution times and average instantiation times, respectively, both in seconds. The first consists of the overall time spent by DLV, from the invocation by the OS to the end of the instantiation (including parsing, output printing, etc.)[4], while the second takes into account exactly the time spent by the instantiation itself.

We wanted also to outline the amount of parallel computation resources actually exploited; to this extent, in Figure 5 we report the average CPU usages[5]. The graph draws the CPU usage w.r.t. to the number of allowed threads: "1.00" means that the process "took" a single CPU, while higher values mean an higher rate of parallel execution. It is important recalling also CPU usages take into account the *overall* time spent by DLV.

We discuss now the results. At a first glance, it is easy to notice as, apart from few exceptions, as the number of allowed threads increases, the performances get better. However, among all the considered problems, one can observe different behaviors.

First of all, 3col and Ancestor do not enjoy any improvement. This was expected, since their dependency graphs are almost composed by a single huge component, thus preventing the possibility of taking advantage from our parallel techniques. Indeed, they settle at the bottom of the graph of Figure 5. Nevertheless, times are not affected by noticeable overheads, even w.r.t. the official DLV release.

[4] This means, both the parallel instantiation and the other non-parallel phases. It is worthwhile stating that the the component graph is computed by the old DLV version as well as the new one; in addition, we verified that the time spent in the computation is actually negligible.

[5] Computed as $((User + System)/Real)$; see [19]. Please note that this does not rate parallelism, but the exploitation of the CPUs, thus giving us an idea about how much the execution has actually been parallelized, even if it is machine-dependent.

It's then possible to identify a set of instances (namely: Knowledge discovery, ETL1, ETL2, Player, Decomp) that, as soon as the number of allowed threads moves from one to two, show an appreciable gain in instantiation time, all above 20% (e.g.: Knowledge Discovery passes from 2.06s to 1.86s; ETL1 from 64.35s to 53.93s). Then, the advantage does not grow, in practice, with the number of allowed threads; as an example, allowing more than three threads for solving Knowledge discovery is useless (1.86s, 1.53s and 1.65s with two, three and four threads, respectively). Almost the same observation can be made for ETL1, ETL2 and Decomp, while Player still exhibits a little performance gain even with four threads. Indeed, looking at the graph of Figure 5, all these instances show an almost flat pattern over 2 allowed threads.

Unfortunately, we note that something strange happens when we look ad total average times: for Decomp and Knowledge discovery, all the *dl.thX* executables show a clear degradation in the performances if compared with the DLV release. We have investigated this strange phenomenon, and discovered that it is actually a technological issue, concerning the standard STL [23] multithreaded memory allocator. In fact, the DLV system 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 [24]. In our case, this sometimes neutralizes all the benefits provided by parallelism. Fortunately, in case of ETL1 and ETL2, this does not waste all the gain, which still stands on about 23% also in the total execution times (ETL1, for instance, moves from 64.99s to 50.4s). This technological problem can be fixed, as indicated for instance in [24]; since the implementation takes quite some time, we left it as a future work.

Finally, a last set of instances (namely: ETL3, Cristal, Timetabling 1, Timetabling 2), clearly exhibits a performance gain growing as the number of allowed threads increases. Instantiation times improvements go from about 18% for Cristal (which passes from 4.17s with one thread to 3.58s with four) to about 40% for Timetabling 1 (from 10.64s to 6.42s). In the graph of Figure 5 the patterns related to these problems are monotonically increasing (in particular, when four threads are allowed, the exploited cpu usage grows up to a factor of 2). These benefits still survive in the total execution times for all instances, apart from Timetabling 2. The difference with this is due to the same memory allocation drawbacks previously discussed[6].

Summarizing, best improvements have been observed within the last set of problems; it is worthwhile noting that all of them come from concrete applications, thus confirming that our approach can be profitably exploited to improve performances of ASP while dealing with real world problems.

For the sake of completeness, it is important noting that the Intel Hyper Threading (HT) technology [25] (implemented by the machine exploited for the experiments) works by duplicating certain sections of the processor, but not all the main execution resources. Basically, each processor pretends to be two "logical" processors in front of the host operating system which, thus, can schedule four threads or processes simultaneously (two processes/threads per CPU); however, these will compete for some important execution resources (e.g., the cache). Consequently, there is only an approximation

[6] Intuitively, these drawbacks increase their weight when memory allocation functions are more frequently invoked.

Problem	dl.release	dl.th1	dl.th2	dl.th3	dl.th4
$Cristal$	4.04 (0.07)	4.17 (0.08)	4.23 (0.08)	4.41 (0.12)	4.66 (0.36)
$Ancestor$	51.78 (0.54)	52.15 (0.48)	52.23 (0.48)	52.68 (0.35)	52.16 (0.64)
$3Col$	15.83 (0.35)	15.71 (0.05)	14.90 (0.30)	15.40 (0.69)	15.32 (0.35)
$Decomp$	6.91 (0.00)	8.83 (0.01)	7.41 (0.77)	6.81 (0.07)	7.23 (0.25)
$TimeTab_1$	7.27 (0.10)	12.14 (0.04)	9.56 (0.56)	8.50 (0.06)	7.93 (0.34)
$TimeTab_2$	12.42 (0.46)	12.85 (0.42)	10.98 (1.99)	8.59 (0.55)	10.78 (3.43)
$KnoDisc$	3.41 (0.00)	5.29 (0.00)	5.09 (0.00)	4.76 (0.00)	4.87 (0.15)
$Player$	6.87 (0.02)	7.00 (0.02)	5.52 (0.47)	5.56 (0.46)	5.34 (0.06)
ETL_1	64.72 (0.35)	64.99 (0.61)	54.58 (2.77)	49.61 (0.61)	50.40 (0.11)
ETL_2	64.25 (0.22)	64.26 (0.42)	49.70 (0.56)	49.89 (0.13)	49.92 (0.53)
ETL_3	182.38 (0.38)	186.72 (0.43)	126.82 (10.21)	118.57 (1.53)	120.97 (2.42)

Table 1. Average Real Execution Times (standard deviations within parentheses).

Problem	dl.release	dl.th1	dl.th2	dl.th3	dl.th4
$Cristal$	3.96 (0.07)	4.03 (0.07)	3.31 (0.08)	3.33 (0.07)	3.44 (0.11)
$Ancestor$	51.75 (0.54)	52.08 (0.48)	52.16 (0.48)	52.61 (0.35)	52.08 (0.64)
$3Col$	15.67 (0.35)	15.40 (0.05)	14.59 (0.30)	15.10 (0.69)	15.01 (0.35)
$Decomp$	6.65 (0.00)	8.33 (0.01)	6.92 (0.77)	6.32 (0.07)	6.73 (0.25)
$TimeTab_1$	6.51 (0.10)	10.64 (0.04)	8.06 (0.54)	6.99 (0.05)	6.42 (0.34)
$TimeTab_2$	11.46 (0.46)	10.97 (0.42)	9.08 (1.98)	6.70 (0.55)	8.90 (3.44)
$KnoDisc$	1.88 (0.00)	2.06 (0.01)	1.86 (0.00)	1.53 (0.01)	1.65 (0.15)
$Player$	6.76 (0.02)	6.79 (0.01)	5.31 (0.47)	5.36 (0.46)	5.13 (0.07)
ETL_1	64.32 (0.35)	64.35 (0.62)	53.93 (2.77)	48.96 (0.61)	49.75 (0.11)
ETL_2	64.00 (0.22)	63.88 (0.41)	49.31 (0.56)	49.50 (0.13)	49.48 (0.57)
ETL_3	180.83 (0.37)	184.17 (0.44)	124.24 (10.19)	115.96 (1.61)	118.41 (2.42)

Table 2. Average Grounding Times (standard deviations within parentheses).

of the behavior of a true four processor machine. This effect cannot be avoided with the standard linux SMP kernel, since it is not aware of all the HT peculiarities. Thus, with pure multi-processor or multi-core machines, the performances of our technique should be even better.

6 Related Work and Conclusions

In this paper, we proposed a new technique for the parallel computation of the instantiation of ASP programs. It exploits some structural properties of the input program $\mathcal{P}$ in order to detect modules of $\mathcal{P}$ that can be evaluated in parallel.

As a matter of fact, the exploitation of parallel techniques for computing answer sets is not new [26–28]; however, our approach is not comparable with the existing ones, since the latter concern the model generation task, instead of the instantiation. Nonetheless, a lot of work has been done in the fields of logic programming and deductive databases [17, 29–39]; still, the techniques are comparable to a limited extent to the one illustrated here: some of them apply to syntactically restricted classes of programs, and some others requires an heavy usage of concurrency-control mechanisms. The only one comparable to the present is the so-called *stream* parallelism, where all the

rules are evaluated simultaneously: basically, the information resulting from the evaluation of each rule is passed to the ones depending on it, like in a pipeline. However, this scheme suffers from heavy communication overheads, while our approach minimizes the usage of "mutexes" in the main data structures, thus reducing the overhead introduced by the concurrency-control constructs.

We have implemented our strategy producing an experimental version of the DLV system, and performed several experiments on a SMP-based machine. The obtained results confirmed, on the one hand, the effectiveness of our technique, which allows one to save real (wall-clock) time, especially while evaluating real-world problem instances; on the other hand, they outlined some annoying technical issues due to the usage of the standard STL multithreaded memory allocator, which is widely considered performance-wise not optimal [24].

We plan to improve the current implementation by solving the problems concerning STL memory allocation performances; nonetheless, we want to extend our parallel grounding technique as well, in order to exploit parallelism also during the instantiation of a single component.

References

1. Gelfond, M., Lifschitz, V.: Classical Negation in Logic Programs and Disjunctive Databases. NGC **9** (1991) 365–385
2. Eiter, T., Gottlob, G., Mannila, H.: Disjunctive Datalog. ACM TODS **22**(3) (1997) 364–418
3. Eiter, T., Faber, W., Leone, N., Pfeifer, G.: Declarative Problem-Solving Using the DLV System. In Minker, J., ed.: Logic-Based Artificial Intelligence. Kluwer (2000) 79–103
4. Bell, C., Nerode, A., Ng, R.T., Subrahmanian, V.: Mixed Integer Programming Methods for Computing Nonmonotonic Deductive Databases. JACM **41** (1994) 1178–1215
5. Subrahmanian, V., Nau, D., Vago, C.: WFS + Branch and Bound = Stable Models. IEEE TKDE **7**(3) (1995) 362–377
6. 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
7. Janhunen, T., Niemelä, I., Seipel, D., Simons, P., You, J.H.: Unfolding Partiality and Disjunctions in Stable Model Semantics. ACM TOCL **7**(1) (2006) 1–37
8. Lierler, Y.: Disjunctive Answer Set Programming via Satisfiability. In: LPNMR'05. LNCS 3662
9. Eiter, T., Leone, N., Mateis, C., Pfeifer, G., Scarcello, F.: A Deductive System for Nonmonotonic Reasoning. In J. Dix and U. Furbach and A. Nerode, ed.: LPNMR'97. LNCS 1265
10. Faber, W., Leone, N., Mateis, C., Pfeifer, G.: Using Database Optimization Techniques for Nonmonotonic Reasoning. In DDLP'99, Prolog Association of Japan (1999) 135–139
11. Leone, N., Perri, S., Scarcello, F.: Improving ASP Instantiators by Join-Ordering Methods. In: LPNMR'01. LNCS 2173
12. Leone, N., Perri, S., Scarcello, F.: BackJumping Techniques for Rules Instantiation in the DLV System. In: NMR 2004. (2004) 258–266
13. Ruffolo, M., Leone, N., Manna, M., Sacca', D., Zavatto, A.: Exploiting ASP for Semantic Information Extraction. In: Proceedings ASP05 - Answer Set Programming: Advances in Theory and Implementation, Bath, UK (2005)
14. Cumbo, C., Iiritano, S., Rullo, P.: Reasoning-based knowledge extraction for text classification. In: Discovery Science. (2004) 380–387

15. 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. In: Proceedings of the 24th ACM SIGMOD International Conference on Management of Data (SIGMOD 2005), Baltimore, Maryland, USA, ACM Press (2005) 915–917
16. Przymusinski, T.C.: Stable Semantics for Disjunctive Programs. NGC **9** (1991) 401–424
17. Ullman, J.D.: Principles of Database and Knowledge Base Systems. Computer Science Press (1989)
18. Faber, W., Pfeifer, G.: `DLV` homepage (since 1996) `http://www.dlvsystem.com/`.
19. Debian: Manual pp. about using a GNU/Linux system `http://packages.debian.org/stable/doc/manpages/`.
20. Exeura s.r.l., Homepage `http://www.exeura.it/`.
21. Gottlob, G., Leone, N., Scarcello, F.: Hypertree Decompositions and Tractable Queries. JCSS (2002)
22. `CRISTAL` project, homepage `http://proj-cristal.web.cern.ch/`.
23. Stepanov, A., Lee, M.: The Standard Template Library. (Part of the evolving ANSI C++ standard. `ftp://butler.hpl.hp.com/stl/`.) (1995)
24. Berger, E.D., McKinley, K.S., Blumofe, R.D., Wilson, P.R.: Hoard: A Scalable Memory Allocator for Multithreaded Applications. In: ASPLOS. (2000) 117–128
25. Koufaty, D., Marr, D.T.: Hyperthreading Technology in the Netburst Microarchitecture. IEEE Micro **23**(2) (2003) 56–65
26. Finkel, R.A., Marek, V.W., Moore, N., Truszczynski, M.: Computing stable models in parallel. In: Answer Set Programming. (2001)
27. Gressmann, J., Janhunen, T., Mercer, R.E., Schaub, T., Thiele, S., Tichy, R.: Platypus: A Platform for Distributed Answer Set Solving. In: LPNMR. (2005) 227–239
28. Pontelli, E., El-Khatib, O.: Exploiting Vertical Parallelism from Answer Set Programs. In: Answer Set Programming. (2001)
29. Wolfson, O., Silberschatz, A.: Distributed Processing of Logic Programs. In: SIGMOD Conference. (1988) 329–336
30. Wolfson, O.: Parallel Evaluation of Datalog Programs by Load Sharing. J. Log. Program. **12**(3&4) (1992) 369–393
31. Gupta, G., Pontelli, E., Ali, K.A.M., Carlsson, M., Hermenegildo, M.V.: Parallel execution of prolog programs: a survey. ACM Trans. Program. Lang. Syst. **23**(4) (2001) 472–602
32. Clark, K.L., Gregory, S.: Parlog: Parallel Programming in Logic. ACM Trans. Program. Lang. Syst. **8**(1) (1986) 1–49
33. Ramakrishnan, R.: Parallelism in Logic Programs. Ann. Math. Artif. Intell. **3**(2-4) (1991) 295–330
34. 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
35. de Kergommeaux, J.C., Codognet, P.: Parallel Logic Programming Systems. ACM Comput. Surv. **26**(3) (1994) 295–336
36. Wu, Y., Pontelli, E., Ranjan, D.: Computational Issues in Exploiting Dependent And-Parallelism in Logic Programming: Leftness Detection in Dynamic Search Trees. In: LPAR. (2005) 79–94
37. Inoue, K., Koshimura, M., Hasegawa, R.: Embedding Negation as Failure into a Model Generation Theorem Prover. In: CADE. (1992) 400–415
38. Ramakrishnan, R., Ullman, J.D.: A Survey of Deductive Database Systems. JLP **23**(2) (1995) 125–149
39. Zhang, W., Wang, K., Chau, S.C.: Data Partition and Parallel Evaluation of Datalog Programs. IEEE Trans. Knowl. Data Eng. **7**(1) (1995) 163–176