Increasing Parallelism while Instantiating 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. One of the most hard tasks performed by Answer Set Programming (ASP) systems is instantiation, which consists of generating variable-free programs equivalent to those given as input. The efficiency of this task is crucial for ASP systems performance especially in case of real-world applications where huge inputs are processed.
We recently proposed a method that exploits the capabilities of multi-processor machines for the instantiation. This method confirmed to be effective especially when dealing with programs consisting of many rules. Here, we report some preliminary results on a rewriting-based strategy that makes the existing technique exploitable even in case of programs with few rules.

1 Introduction

In the last few years, multi-core/multi-processor architectures become standard, thus making Symmetric MultiProcessing (SMP) [1] common also for entry-level systems and PCs. In SMP architectures two or more identical processors connect to a single shared main memory, enabling simultaneous multithread execution. Such technology might be profitably exploited also in the field of Answer Set Programming (ASP): indeed, recent applications of ASP in different emerging areas (see e.g., [2–8]), have evidenced the practical need for faster and scalable ASP systems.

ASP is a declarative approach to programming proposed in the area of nonmonotonic reasoning and logic programming [9–15] which features a high declarative nature combined with a relatively high expressive power [16, 17]; unfortunately, this comes at the price of a high computational cost. The kernel modules of ASP 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; thus, having an efficient instantiation procedure is, in general, a key feature of ASP systems.

In [18] we proposed a technique for the parallel instantiation of ASP programs, allowing 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 mutex-locks [1], and thus the

* 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.”

time spent by concurrency-control mechanisms. 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 and, according to the interdependencies between the modules, decides which of them can be processed in parallel; on the other hand, it parallelizes the evaluation within each module. The proposed strategy has been implemented into the instantiator module of the ASP system DLV [16], thus obtaining a parallel ASP instantiator. This new system is effective especially in the evaluation of programs consisting of several rules with a large amount of input data [18].

Here we present the basic principles concerning a rewriting-based strategy that aims at improving the system performance even when dealing with programs consisting of few rules. Basically, input programs are rewritten in such a way that the instantiation of each rule is split into different jobs that can be done in parallel. Results of a preliminary experimental activity are also presented.

2 Parallel Instantiation of ASP Programs

In this Section the parallel instantiation algorithm of [18], which relies on the DLV ("standard") instantiation procedure, is briefly described. A detailed discussion about the DLV instantiator and the details of the parallel instantiation technique are out of the scope of this short paper; for further insights we kindly refer the reader to [16, 18, 19].

Roughly, the instantiation module of DLV splits up a given program $\mathcal{P}$ into subprograms called *modules*. Each of these modules corresponds to a strongly connected component (SCC) of a particular graph, called *dependency graph* ($G_{\mathcal{P}}$), which, intuitively, describes how predicates depend on each other. The DLV instantiator processes them, one at a time, according to a (partial) ordering induced by $G_{\mathcal{P}}$, which ensures that all data needed for the instantiation of a module have been already generated by the instantiation of the modules preceding it. A procedure called *InstantiateComponent* is in charge of instantiating modules, while a procedure called *InstantiateRule* builds all the ground instances of a given rule r. A single call to *InstantiateRule* is sufficient for completely evaluating non-recursive rules, while recursive ones are processed several times according to a semi-naïve evaluation technique [20].

The procedure presented in [18] combines two strategies: the first one for the parallel evaluation of different modules, while the second for the concurrent instantiation of rules within a module. Both strategies avoid the use of mutex-locks: the former by properly choosing the modules to be evaluated in parallel; the latter by suitably parallelizing each iteration of the semi-naïve algorithm. The idea is that if there are no two threads in read/write (nor write/write) conflict on the same data structure, then no synchronization is needed. This allows one to drastically reduce the so-called parallel overhead.

Parallelizing the Program Instantiation. The parallel instantiation of an input program $\mathcal{P}$ is based on classical producer-consumers pattern. A *manager* thread (producer) identifies the components that can be processed at a given time, and delegates their instantiation to a number of *instantiator* threads (consumers). The choice of the components to be processed in parallel is made according to the above-mentioned partial ordering. Intuitively, a component C from the bunch of components to be instantiated

is given by the manager to the instantiators only if all the information needed has already been computed.

Parallelizing the Instantiation of a Program Module. Within a single module, each rule is processed by one thread. First, all non-recursive rules are concurrently evaluated, then, as soon as all of them are done, recursive ones are processed. In particular, at the end of each single iteration of the semi-naïve algorithm, instantiators synchronize in such a way that common structures (like, e.g. current partial interpretation) can be safely updated by the manager, and next iteration starts.

3 Parallelization of Rule Instantiation: Ideas and Experiments

The technique described above makes parallel the execution of two different steps of the instantiation process: the evaluation of program modules and the instantiation of rules within each module. However, it is not fully exploitable in case of programs with few components and few rules. Consider, for instance, 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, and then by processing the constraint c only once the extension of col has been computed.

Thus, such encoding does not allow the existing technique to make the evaluation parallel at all. However, as it is easy to see, 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 work, while writing a program: clearly, such a requirement is not desirable for a declarative system. Nevertheless, an automatic rewriting of the input program into an equivalent one, whose evaluation can be made more parallel, could make transparent this optimization process to the user. For instance, the following is an alternative encoding for the 3-Colorability problem which can be obtained by automatically rewriting the original one:

$$(r) \quad col(X, red) \vee col(X, yellow) \vee col(X, green) \text{ :- } node(X).$$
$$(c_1) \quad \text{:- } col(X, C),\ col(Y, C),\ edge1(X, Y).$$
$$(c_2) \quad \text{:- } col(X, C),\ col(Y, C),\ edge2(X, Y).$$

The set of edges is *split up* into two (equally sized) 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, but the computation now can be carried out in parallel by two different instantiators. Obviously, this rewriting strategy can be straightforwardly extended for allowing more than two instantiators to work in parallel.

This rewriting technique somewhat coincides with the so-called *Or-parallelism* [21–23], which is here simulated by splitting obtained via rewriting, without a drastic and involved modification of a system implementation. In general, this idea allows one to "split" any encoding; but there are different, sometimes many, ways to rewrite a program. For instance, another possible encoding for 3-Colorability could be obtained by

splitting predicate $color$ into $color_1 \dots color_n$.[1] Or, if possible, one can also consider to split also two (or more) body predicates, like both $color$ and $edge$; in this case, the rewritten program requires also a number of rules, the body of each containing a join between $color_i$ and $edge_j$, $1 \leq i, j \leq n$. The choice of the most convenient is not trivial, and must be made according to several factors. For instance, the instantiation exploits, for the evaluation of each rule, clever techniques based on join ordering [20, 24] and backjumping [25]. A "bad" split might reduce or neutralize the benefits provided by these techniques, 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). Intuitively, the join ordering techniques establish the body order according to several facts, as the (estimated) size (number of instances) of body predicates. While rewriting a rule r, according to the split of a body predicate p into $p_1, \dots, p_n$, a number of rules $r_1, \dots, r_n$ is obtained with the same shape as r, but with body predicates $p_1, \dots, p_n$ smaller in size w.r.t. p. Thus, the body orderings of $r_1, \dots, r_n$ may differ from the one of r, possibly significantly affecting the instantiation time. These considerations are confirmed by some experiments reported in the following section.

Experiments. In order to check the viability of the rewriting for increasing parallelism and to evaluate the effects of different splits on performance, we have carried out some preliminary experiments. In particular, we considered three well-known problems, whose standard encodings in disjunctive ASP are not suitable for parallel evaluation with the technique of [18], namely 3-Colorability, Reachability (compute the transitive closure of a given graph), and Same Generation (given a parent-child relationship, i.e. a tree, find pairs of persons belonging to the same generation). For each problem we analyzed different splits, but for space reason we refrain from reporting the corresponding encodings here.

We assessed our encodings by exploiting the parallel instantiator of [18] on a machine equipped with two Intel Xeon HT (single core) processors clocked at 3.60GHz. In particular we compared a single-threaded grounding engine (ST in the table) against a multi-threaded grounding engine (MT).[2] Table 1 reports, for each problem, the average instantiation time spent by the two engines (each experiment has been repeated five times), for two different instances. For each problem we tested the standard encoding against two different rewritten programs; these are based on the split of predicates $edge$ and $color$, respectively, for 3-Colorability, and $edge$ and $node$ for Reachability; for Same Generation we split on $edge$ and considered two different body orderings. The results clearly show that MT always outperfoms ST while instantiating split encodings, with a gain close to 50% (the best one can obtain from a two-processor machine), while, as expected, the standard encodings do not enjoy any gain. Moreover, as discussed before, different splits produce very different behaviors. Indeed, while the two splits for Reachability lead to comparable results, for 3-Colorability and Same Generation the scenario changes. For instance, splitting $color$ instead of $edge$ in 3-Colorability

[1] Note that, differently from $edge$, $color$ is not an input predicate; splitting on it requires to split the predicates it depends on and generate new rules accordingly.

[2] The maximum number of allowed concurrent instantiator threads was set to the number of simultaneous (i.e., executed in a different CPU) threads/processes allowed by the hardware.

	3Colorability			Reachability			Same Generation		
	$NoSplit$	$Edge$	$Color$	$NoSplit$	$Edge$	$Node$	$NoSplit$	$Edge_1$	$Edge_2$
ST	46.4	45.3	202.5	15.8	14.3	14.7	45.11	377.9	22.9
MT	46.5	23.6	104.5	15.6	7.8	8.2	45.20	197.9	15.6
ST	124.2	128.6	544.3	375.2	272.7	281.7	2680.1	31996.8	449.5
MT	129.4	66.6	278.1	374.3	139.9	144.1	2674.8	16369.1	274.9

Table 1. Average Grounding Times (s).

nullifies the advantages of parallel computation, and the overall time becomes higher than the one required by the original encoding. In addition, the results for Same Generation clearly show how a split can interfere with the join ordering technique, thus leading to unexpected side-effects: splitting $edge$ caused a noticeable worsening (column $Edge_1$ of Table1), but a more deep analysis allowed to discover that the times can be drastically reduced by simply changing the body ordering (column $Edge_2$).

Summarizing, the splitting strategy produces encodings that fully enjoy the parallel technique (MT gets a 50% off against ST); importantly, by carefully choosing the predicate(s) to split, the technique presents noticeable gains when compared to non-parallel instantiation (MT with best split halves times against ST with original encoding).

Conclusion Preliminary experiments confirmed that the "split" rewriting is viable, but cannot be applied trivially. We plan to develop heuristics that allow for selecting the best predicate to split, in order to implement a smart rewriter that can be seamlessly integrated in our parallel instantiator. This will lead to take further advantage of multi-processing, especially in the case of program encodings containing very few rules.

References

1. Stallings, W.: Operating systems (3rd ed.): internals and design principles. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1998)
2. 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
3. Lembo, D., Lenzerini, M., Rosati, R.: Source Inconsistency and Incompleteness in Data Integration. In: Proceedings of the Knowledge Representation meets Databases International Workshop (KRDB-02), Toulouse France, CEUR Electronic Workshop Proceedings `http://sunsite.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-54/` (2002)
4. Soininen, T., Niemelä, I.: Developing a Declarative Rule Language for Applications in Product Configuration. In Gupta, G., ed.: Proceedings of the 1st International Workshop on Practical Aspects of Declarative Languages (PADL'99). Volume 1551 of Lecture Notes in Computer Science., Springer (1999) 305–319
5. Aiello, L.C., Massacci, F.: Verifying security protocols as planning in logic programming. ACM Transactions on Computational Logic **2**(4) (2001) 542–580

6. Bertino, E., Mileo, A., Provetti, A.: User Preferences VS Minimality in PPDL. In Buccafurri, F., ed.: Proceedings of the Joint Conference on Declarative Programming APPIA-GULP-PRODE 2003. (2003) 110–122
7. Buccafurri, F., Caminiti, G.: A Social Semantics for Multi-agent Systems. In Baral, C., Greco, G., Leone, N., Terracina, G., eds.: Logic Programming and Nonmonotonic Reasoning — 8th International Conference, LPNMR'05, Diamante, Italy. Volume 3662 of Lecture Notes in Computer Science., Springer Verlag (2005) 317–329
8. Costantini, S., Tocchio, A.: The dali logic programming agent-oriented language. In Alferes, J.J., Leite, J., eds.: Proceedings of the 9th European Conference on Artificial Intelligence (JELIA 2004). Volume 3229 of Lecture Notes in AI (LNAI)., Springer Verlag (2004) 685–688
9. Gelfond, M., Lifschitz, V.: Classical Negation in Logic Programs and Disjunctive Databases. New Generation Computing **9** (1991) 365–385
10. Eiter, T., Gottlob, G., Mannila, H.: Disjunctive Datalog. ACM Transactions on Database Systems **22**(3) (1997) 364–418
11. Eiter, T., Faber, W., Leone, N., Pfeifer, G.: Declarative Problem-Solving Using the DLV System. In Minker, J., ed.: Logic-Based Artificial Intelligence. Kluwer Academic Publishers (2000) 79–103
12. Lifschitz, V.: Answer Set Planning. In Schreye, D.D., ed.: Proceedings of the 16th International Conference on Logic Programming (ICLP'99), Las Cruces, New Mexico, USA, The MIT Press (1999) 23–37
13. Marek, V.W., Truszczyński, M.: Stable Models and an Alternative Logic Programming Paradigm. In Apt, K.R., Marek, V.W., Truszczyński, M., Warren, D.S., eds.: The Logic Programming Paradigm – A 25-Year Perspective. Springer Verlag (1999) 375–398
14. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press (2003)
15. Gelfond, M., Leone, N.: Logic Programming and Knowledge Representation – the A-Prolog perspective . Artificial Intelligence **138**(1–2) (2002) 3–38
16. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV System for Knowledge Representation and Reasoning. ACM Transactions on Computational Logic **7**(3) (2006) 499–562
17. Dantsin, E., Eiter, T., Gottlob, G., Voronkov, A.: Complexity and Expressive Power of Logic Programming. ACM Computing Surveys **33**(3) (2001) 374–425
18. Calimeri, F., Perri, S., Ricca, F.: Experimenting with Parallelism for the Instantiation of ASP Programs. Journal of Algorithms in Cognition, Informatics and Logic (2008) To appear. Available at `http://dx.doi.org/10.1016/j.jalgor.2008.02.003`.
19. Faber, W., Leone, N., Perri, S., Pfeifer, G.: Efficient Instantiation of Disjunctive Databases. Technical Report DBAI-TR-2001-44, Institut für Informationssysteme, Technische Universität Wien, Austria (2001) Online at `http://www.dbai.tuwien.ac.at/local/reports/dbai-tr-2001-44.pdf`.
20. Ullman, J.D.: Principles of Database and Knowledge Base Systems. Computer Science Press (1989)
21. Clark, K.L., Gregory, S.: Parlog: Parallel Programming in Logic. ACM Transactions on Programming Language Systems **8**(1) (1986) 1–49
22. Ramakrishnan, R.: Parallelism in Logic Programs. Annals of Mathematics and Artificial Intelligence **3**(2–4) (1991) 295–330
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. Leone, N., Perri, S., Scarcello, F.: Improving ASP Instantiators by Join-Ordering Methods. In Eiter, T., Faber, W., Truszczyński, M., eds.: Logic Programming and Nonmonotonic Reasoning — 6th International Conference, LPNMR'01, Vienna, Austria. Volume 2173 of Lecture Notes in AI (LNAI)., Springer Verlag (2001) 280–294
25. Leone, N., Perri, S., Scarcello, F.: BackJumping Techniques for Rules Instantiation in the DLV System. In: Proceedings of the 10th International Workshop on Non-monotonic Reasoning (NMR 2004), Whistler, BC, Canada. (2004) 258–266