An Heuristics for Load Balancing and Granularity Control in the Parallel Instantiation of Disjunctive Logic Programs

Simona Perri, Francesco Ricca, and Marco Sirianni

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

Abstract. In this paper we present a dynamic heuristics that allows for improving the performance of a parallel instantiator algorithm based on the DLV system. In this system, each rule is rewritten in several "splits" of the same size that are assigned to a number of parallel instatiator subprocesses. The new heuristics allows for dynamically determining an optimal amount of work that has to be assigned to each parallel instantiator, and, thus, it improves the overall efficiency of the parallel evaluation. We implemented our heuristics and performed an experimental analysis that confirms the efficacy of the proposed method.

1 Introduction

In the last few years, entry-level computer systems have started to implement multi-core/multi-processor SMP (Symmetric MultiProcessing) architectures. In a modern SMP computer two or more identical processors can connect to a single shared main memory, and the operating system supports multithreaded programs for exploiting the available CPUs [1]. However, most of the available software was devised for single-processor machines and is unable to exploit the power of SMP architectures. Recently [2, 3], such technology has been exploited for implementing faster evaluation systems 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 [5–10] which features a declarative nature combined with a relatively high expressive power [11, 12].

Traditionally, the kernel modules of ASP systems work on a ground instantiation of the input program. Therefore, 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 [10, 12]); thus, having an efficient instantiation procedure is, in general, crucial for the performance of the entire ASP systems. Moreover, some recent applications of ASP (see e. g. , [13–16]), have evidenced the practical need for faster instantiators. It is easy to see that the exploitation of SMP technology in the grounding process can bring significant performance improvements. Indeed, an effective technique for the parallel instantiation of ASP programs was proposed in [2]. However, the efficacy of this method was limited to programs with many rules, since it roughly allows for instantiating independent rules in parallel; but, in [3] a rewriting technique has been proposed that modifies the input program in such a way that the technique of [2] becomes applicable also in case of programs with few rules. The following example explains the idea behind this rewriting

(the reader is referred to [3] for a detailed description of the technique). Consider the program encoding of the well-known 3-colorability problem:

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

In this case, the technique of [2] is unable to produce a parallel evaluation, since it proceeds by first instantiating (r) (thus, computing the extension of col), and, then, only once this is done, by processing the constraint (c).

However, one may rewrite this program in an equivalent one which is more amenable for parallel evaluation. Basically, since each single 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 program can be rewritten as follows [3]:

(r_1) $col(X, red) \vee col(X, yellow) \vee col(X, green) \text{ :- } node_1(X).$
(r_2) $col(X, red) \vee col(X, yellow) \vee col(X, green) \text{ :- } node_2(X).$
...
(r_n) $col(X, red) \vee col(X, yellow) \vee col(X, green) \text{ :- } node_n(X).$
(c_1) $\text{ :- } edge_1(X, Y), col(X, C),\ col(Y, C).$
...
(c_n) $\text{ :- } edge_n(X, Y), col(X, C),\ col(Y, C).$

The sets of nodes and edges are *split up* into subsets by splitting the extension of predicates $node$ and $edge$, respectively. The resulting program is equivalent to the original one modulo renaming, but its instantiation can be done in parallel.

This rewriting was implemented [3] in a parallel ASP instantiator based on DLV [11]. In the first version of the system the number of rule splits was set to a global user-defined value (the same for each rule in input). However, this naïve strategy is not optimal in most cases. Indeed, if each instantiator receives a "very small" amount of work, then the costs added by parallel execution are larger than the benefits (because of the overhead introduced by thread creation and scheduling); on the other hand, if the amount of work assigned to threads is "too big" then a resulting bad workload distribution will reduce the advantages of parallel evaluation. Note also that, the optimal setting may be different for each rule.

In this paper, we propose a dynamic heuristics that is able to improve the overall efficiency of the parallel evaluation of [3] by automatically determining, rule by rule, an optimal amount of work that has to be assigned to each parallel instantiator. Moreover, we implemented our heuristics, and we report here the results of an experimental analysis that confirms the efficacy of the proposed method.

2 An Heuristics for Load Balancing and Granularity Control

A real implementation of a parallel system has to deal with two important issues that strongly affect the performance: load balancing and granularity. Indeed, if the workload is not uniformly distributed to the available processors then the benefits of parallelization are not fully obtained; moreover, if the "amount" of work assigned to each parallel processing unit is too small then the (unavoidable) overheads due to creation and scheduling of parallel tasks might overcome the advantages of parallel evaluation (in a corner case, adopting a sequential evaluation might be preferable).

The parallel grounder described in [3] implements a naïve strategy: basically each rule is rewritten in a globally fixed number of "splits" (specified by the user). Here, a crucial role is played by the number of splits allowed for each rule, which is (usually) the main source of concurrently running threads, and it directly determines the "amount" of work (and, thus, the "size of the split") assigned to instantiators. It is easy to see that the best possible fixed setting for the number of splits might be not optimal, since the evaluation of different rules in the same program may require significantly different execution times. In our setting, (i. e. shared memory processor) developing a sophisticated granularity-control strategy is not essential (as also observed in [4]) as for other parallel architectures (like, e. g. , clusters); rather it is sufficient to set the size of the splits for each rule to an adequate value. The size of the split should be sufficiently large to avoid thread management overhead; and sufficiently small to exploit the preemptive multitasking scheduler of the operating system for obtaining a good workload distribution. Clearly, also the number of running threads has to be controlled.

In order to satisfy both requirements, in our implementation: (i) we let the user set the number of concurrently running thread (an adequate value is given by a multiple of the number of available CPUs so that preemptive multitasking is exploited for load balancing); and (ii) we implemented and tuned an heuristics that allows for selecting an optimal split size for each rule. Note that the second task is not trivial, since the time needed for evaluating each rule is not known a priori.

More in detail, our method computes an estimation $e(r)$ of the amount of work required for evaluating each rule r of the program (just before r has to be instantiated) then, it exploits $e(r)$ for associating to r its split size among three empirically-defined values: small, large, and no split (i.e. sequential evaluation) [1];. Basically, very easy rules are evaluated sequentially, since the overhead introduced by threads is higher than their expected evaluation time (granularity control); whereas, for hard rules a small split size is employed for obtaining a finer distribution of work; finally, easy rules, whose computation can still exploit some parallelism, are evaluated using a large split size for minimizing the overheads. The estimation $e(r)$ is obtained by combining (actually summing) two factors: the number of comparisons made by our algorithm and the number of operations needed to compute and print the output ("size of the corresponding join"). The latter is motivated by the similarities between the rule instantiation process and the evaluation of a join in a database system. In the following, we describe how the two factors have been estimated.

Size of the join. The size of the join between two relations R and S with one or more common variables can be estimated, according to [17], as follows:

$$T\left(R \bowtie S\right) = \frac{T\left(R\right) \cdot T\left(S\right)}{\prod_{X \in var(R) \cap var(S)} \max\left\{V\left(X,R\right), V\left(X,S\right)\right\}}$$

where $T\left(R\right)$ is the number of tuples in R, and $V\left(X,R\right)$ (called selectivity) is the number of distinct values assumed by the variable X in R. For joins with more relations one can repeatedly apply this formula to couple of body predicates according to a given evaluation order.

[1] Note that in case of recursive rules, a new distribution is performed each time they are processed (according semi-naïve evaluation [2]); thus obtaining a dynamic load balancing.

Problem	small	medium	large	Heuristics
$3col_1$	11. 50 (0. 11)	8. 57 (0. 01)	**8. 57 (0. 07)**	8. 91 (0. 04)
$3col_2$	14. 42 (0. 15)	**11. 68 (0. 13)**	11. 89 (0. 05)	11. 29 (0. 20)
$3col_3$	23. 01 (0. 23)	19. 68 (1. 06)	**19. 57 (0. 04)**	18. 46 (0. 19)
$reach_2$	60. 73 (0. 17)	40. 73 (0. 18)	**39. 92 (0. 32)**	39. 70 (0. 12)
$reach_3$	129. 78 (0. 77)	84. 67 (0. 30)	**82. 78 (0. 32)**	82. 44 (0. 52)
$ramsey_1$	**44. 00 (0. 77)**	91. 04 (0. 65)	95. 92 (0. 12)	43. 19 (0. 31)
$ramsey_2$	**72. 38 (0. 40)**	170. 51 (1. 28)	236. 82 (2. 76)	71. 96 (0. 21)
$ramsey_3$	**112. 66 (0. 71)**	286. 64 (0. 74)	294. 51 (2. 51)	111. 02 (0. 13)

Table 1. Result for different size of the split - result for heuristics choice.

Number of comparisons. An approximation of the number of comparison done for instantiating a rule r is: $C(r) = \sum_{x \in X(r)} \prod_{l \in L(r,x)} V(x, l)$, where $X(r)$ is the set of variables that appear in at least two literals in the body of r, $L(r, x)$ is the set of body literals in which x occurs; and $V(x, l)$ is the selectivity of x in the extension of l. Roughly, the number of comparisons is approximated by sum of the products of the number of distinct values assumed by each join variable in the body of r.

3 Experiments

We implemented our heuristics and tested its efficacy in a collection of benchmark programs already used for assessing ASP instantiators performance ([11, 18, 19]). In particular, we considered the following well-known problems: *3-colorability, Reachability* and *Ramsey Numbers*; for space reasons we do not report the encodings (they are available at `http://www.mat.unical.it/ricca/downloads/cilc09.zip`), for a detailed problems description refer to [11, 19, 3]. The machine used for the experiments is a two-processor Intel Xeon "Woodcrcaratteriest" (quad core) 3GHz machine with 4MB of L2 Cache and 4GB of RAM, running Debian GNU Linux 4. 0. We set the number of concurrent splits to 32 (the quadruple of available processors). Actually, an experimental analysis (not reported here for space reasons) confirmed that this fixed setting is optimal for the available hardware.

We measured the performance of the system in the case of some growing (fixed) split sizes (small=1 tuple, medium=50 tuples, large=100 tuples)[2], and when the new heuristics is employed. 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 Table 1. In particular, the first three columns contain the average instantiation times (and standard deviation) for the different split sizes, while last column reports the performance obtained by applying the heuristics. The fixed setting that obtains the best result is reported in bold face for each instance.

First of all we notice that *the performance of the heuristics version is always optimal and overcomes the better fixed setting in most cases*, since it benefits from the selection of a different split size for each rule of the program.[3].

About Ramsey, since the encoding is composed of few "very easy" rules and two "hard" constraints, the best split size choice for the entire program is the smaller one (see Table 1). We verified that the version with the heuristics evaluates sequentially the

[2] More experiments have been done on different split size; we reported here the most significant.

[3] In addition, we observed that performance gains (w.r.t. serial execution) range form about 700% up to 790% in the best setting, which is near to the theoretical limit for eight-processors.

rules and selects a small split size for the constraints, thus resulting the best performer. Dual considerations hold for 3col, where the best split size for the entire program is the largest (third column), since the evaluation of rules requiring a medium/large split sizes dominates the computation time. Still, the version equipped with the heuristics, overcomes the results of the fixed split size, as it chooses a large split size for the rule, and a small split size for the constraints. The application of the heuristics to reachability (the encoding is composed of two "easy" rules, one of which is recursive) results in the selection of large split size and sequential evaluation respectively for the two input rules, and this choice results to be still more appropriate.

As far as future work is concerned, we are experimenting for a finer tuning of the heuristics, and we are considering a larger set of problems. Moreover, we planned to test the system behavior when more than 8 processors are available (probably by exploiting a simulation environment).

References

1. Stallings, W.: Operating systems (3rd ed.): internals and design principles. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1998)
2. Calimeri, F., Perri, S., Ricca, F.: Experimenting with Parallelism for the Instantiation of ASP Programs. Journal of Algorithms in Cognition, Informatics and Logics **63**(1–3) (2008) 34–54
3. Vescio, S., Perri, S., Ricca, F.: Efficient Parallel ASP Instantiation via Dynamic Rewriting. In: ASPOCP 2008, Udine, Italy (2008)
4. Lopez, P., Hermenegildo, M., Debray, S.: A Methodology for Granularity Based Control of Parallelism in Logic Programs. In: J. Symbolic Computation (1996), 22, 715-734
5. Gelfond, M., Lifschitz, V.: Classical Negation in Logic Programs and Disjunctive Databases. New Generation Computing **9** (1991) 365–385
6. Lifschitz, V.: Answer Set Planning. In Schreye, D.D., ed.: (ICLP'99) 23–37
7. 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
8. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press (2003)
9. Gelfond, M., Leone, N.: Logic Programming and Knowledge Representation – the A-Prolog perspective . Artificial Intelligence **138**(1–2) (2002) 3–38
10. Eiter, T., Gottlob, G., Mannila, H.: Disjunctive Datalog. ACM Transactions on Database Systems **22**(3) (September 1997) 364–418
11. 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
12. Dantsin, E., Eiter, T., Gottlob, G., Voronkov, A.: Complexity and Expressive Power of Logic Programming. ACM Computing Surveys **33**(3) (2001) 374–425
13. 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: (SIGMOD 2005), Baltimore, Maryland, USA, ACM Press (2005) 915–917
14. Curia, R., Ettorre, M., Gallucci, L., Iiritano, S., Rullo, P.: Textual Document Pre-Processing and Feature Extraction in OLEX. In: Data Mining VI, WIT Pres, 2005, pp. 163-173
15. Massacci, F.: Computer Aided Security Requirements Engineering with ASP Non-monotonic Reasoning, ASP and Constraints, Seminar N 05171. Dagstuhl Seminar (2005)

16. Ruffolo, M., Leone, N., Manna, M., Saccà, D., Zavatto, A.: Exploiting ASP for Semantic Information Extraction. Proceedings ASP05, Bath, UK (July 2005) 248–262
17. Ullman, J.D.: Principles of Database and Knowledge Base Systems. Computer Science Press (1989)
18. Gebser, M., Liu, L., Namasivayam, G., Neumann, A., Schaub, T., Truszczyński, M.: The first answer set programming system competition. In: LPNMR 2007,LCNCS 4483,3-17
19. Perri, S., Scarcello, F., Catalano, G., Leone, N.: Enhancing DLV instantiator by backjumping techniques. Annals of Mathematics and Artificial Intelligence **51**(2–4) (2007) 195–228