Parallel Instantiation in DLV[⋆]

Simona Perri, Francesco Ricca, and Marco Sirianni

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

Abstract. Answer Set Programming (ASP) is a purely-declarative logic programming language allowing for disjunction and nonmonotonic negation. The evaluation of ASP Programs is traditionally carried out in two steps. In the first step an input program $\mathcal{P}$ undergoes the so-called instantiation (or grounding) process, which produces a program $\mathcal{P}'$ semantically equivalent to $\mathcal{P}$, but not containing any variable; in turn, $\mathcal{P}'$ is evaluated by using a backtracking search algorithm in the second step. This paper presents a new parallel version of the instantiator of DLV, featuring new load-balancing and granularity control heuristics, which is able to exploit the processing power offered by multi-core/multi-processor SMP machines.

1 Introduction

Answer Set Programming (ASP) [1, 2] is a purely declarative programming paradigm based on nonmonotonic reasoning and logic programming. The language of ASP is based on logic rules; a *disjunctive rule* (*rule*, for short) r is a formula $a_1 \ \vee \ \cdots \ \vee \ a_n$:– $b_1, \cdots, b_k,$ `not` $b_{k+1}, \cdots,$ `not` b_m. where $a_1, \cdots, a_n, b_1, \cdots, b_m$ are atoms (possibly containing variables) and $n \geq 0$, $m \geq k \geq 0$. An ASP program is a set of rules. The semantics of an ASP program was originally given in [1] and is based on the Gelfond-Lifschitz transformation. Basically, the idea of answer set programming is to represent a given computational problem by a logic program the answer sets of which correspond to solutions, and then, use an answer set solver to find such solutions [2]. In the following we assume the reader to be familiar with basic logic programming terminology and ASP.

The main advantage of ASP is its high declarative nature combined with a relatively high expressive power [3, 4]; but this comes at the price of a high computational cost, which makes the implementation of efficient ASP systems a difficult task. Some effort has been made to this end, and, after some pioneering work, there are nowadays a number of systems that support ASP and its variants [3, 5–12].

Traditionally, the kernel modules of such systems operate on a ground instantiation of the input program, i.e. a program that does not contain any variable, but is semantically equivalent to the original input [13]. 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 expensive

⋆ Partially supported by the Regione Calabria and EU under POR Calabria FESR 2007-2013 within the PIA project of DLVSYSTEM s.r.l..

(see [4]); and, nowadays, it is widely recognized that having an efficient instantiation procedure is crucial for the performance of the entire ASP system. Many optimization techniques have been proposed for this purpose [14–16]; nevertheless, the performance of instantiators is still not acceptable in many cases, especially when the input data are significantly large (real-world instances, for example, may count hundreds of thousands of tuples).

In this scenario, significant performance improvements can be obtained by exploiting modern multi-core/multi-processor SMP [17] machines, featuring several CPU in the same case. In the past only expensive servers and workstations supported this technology; whereas, at the time of this writing, most of the personal computers systems and even laptops, are equipped with (at least one) dual-core processor. This means that the benefits of true parallel processing are enjoyable also in entry-level systems and PCs. However, traditional ASP instantiators were not developed with multi-processor/multi-core hardware in mind, and are unable to fully exploit the computational power offered by modern machines.

This paper presents a system for the parallel instantiation of ASP Programs, which is able to exploit the computational power offered by multi-core/multi-processor machines for obtaining a faster instantiation. The system is based on the state-of-the-art ASP instantiator of the DLV system [3]; moreover it extends the recently-proposed [18] techniques for parallel ASP instantiation by introducing a number of relevant improvements: (i) parallelism is exploited in three different stages of the computation[1] (component level, rule level, single rule level); and (ii) dynamic load balancing and granularity control strategies based on computationally-cheap heuristics are supported. In this way, the efficacy of the system is no-more limited to programs with many rules (as in [18]), and also the particularly (common and) difficult-to-parallelize class of programs with few rules is handled in an effective way.

An experimental activity is also reported, that was carried out on a variety of publicly-available benchmarks already exploited for evaluating the performance of instantiation systems. The results are very promising: superlinear speedups are observed in the case of easy-to-parallelize problem instances; and, nearly optimal efficiencies are measured in the case of hard-to-parallelize problem instances.

The remainder of the paper is structured as follows: Section 2 describes the employed parallel instantiation strategies; Section 3 discusses the results of the experiments carried out in order to evaluate the performance of the system; finally, Section 4 is devoted to related works, and Section 5 draws some conclusions.

2 Parallel Instantiation of ASP Programs

In this section we briefly describe the employed techniques for the parallel instantiation of ASP Programs first; and then, we describe the dynamic load balancing and granularity control strategy employed in the system.

In particular, we show that according to such techniques, three levels of parallelism can be exploited during the instantiation process, namely, components, rules and single

[1] Preliminary results have been presented in [19].

rule level. The first level allows for instantiating in parallel subprograms of the program in input and it is especially useful when handling programs containing parts which are, somehow, independent. The second one, the rules level, allows for the parallel evaluation of rules within a given subprogram and it is thus useful when the number of rules in the subprograms is high. The third one, the single rule level, allows for the parallel evaluation of a single rule and it is thus crucial for the parallelization of programs with few rules, where the first two levels are almost not applicable.

The first two levels were first employed in [18] while the third one was preliminarily presented in [19]. A detailed description of these techniques is out of the scope of this paper. For further details, we refer the reader to [18, 19].

2.1 Parallel Instantiation Techniques

Components Level. The first level of parallelism, called *Components Level* essentially consists on dividing the input program $\mathcal{P}$ into subprograms, according to the dependencies among the IDB predicates of $\mathcal{P}$, and by identifying which of them can be evaluated in parallel. More in detail, each program $\mathcal{P}$ is associated with a graph, called the *Dependency Graph* of $\mathcal{P}$, which, intuitively, describes how IDB predicates of $\mathcal{P}$ depend on each other. For 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) [2]C of $G_{\mathcal{P}}$, the set of rules defining all the predicates in C is called *module* of C. A rule r occurring in a *module* of a component C (i.e., defining some predicate $\in C$) 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*. Moreover, a partial ordering among the SCCs is induced by $G_{\mathcal{P}}$, 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 $G_{\mathcal{P}}$ from A to B.

According to such definitions, the instantiation of the input program $\mathcal{P}$ can be carried out by separately evaluating its modules; if the evaluation order of the modules respects the above mentioned partial ordering then a small ground program is produced. Indeed, this gives the possibility to compute ground instances of rules containing only atoms which can possibly be derived from $\mathcal{P}$ (thus, avoiding the combinatorial explosion which can be obtained by naively considering all the atoms in the Herbrand Base).

Intuitively, this partial ordering guarantees that a component A precedes a component B if the program module corresponding to A has to be evaluated before the one of B (because the evaluation of A produces data which are needed for the instantiation of B). Moreover, the partial ordering allows for determining which modules can be evaluated in parallel. Indeed, if two components A and B, do not depend on each

[2] 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.

other, then the instantiation of the corresponding program modules can be performed simultaneously, because the instantiation of A does not require the data produced by the instantiation of B and vice versa. The dependency among components is thus the principle underlying the first level of parallelism. At this level subprograms can be evaluated in parallel, but still the evaluation of each subprogram is sequential.[3]

Rules Level. The second level of parallelism, called the *Rules Level*, allows for concurrently evaluating the rules within each module. According to this technique, rules are evaluated following a semi-naïve schema [20] and the parallelism is exploited for the evaluation of both exit and recursive rules. More in detail, for the instantiation of a module M, first all exit rules are processed in parallel by exploiting the data (ground atoms) computed during the instantiation of the modules which M depends on (according to the partial ordering induced by the dependency graph). Only afterward, recursive rules are processed in parallel several times by applying a semi-naïve evaluation technique. At each iteration n, the instantiation of all the recursive rules is performed concurrently and by exploiting only the significant information derived during iteration $n-1$. This is done by partitioning significant atoms into three sets: ΔS, S and NS. NS 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 NS are empty; while S contains all the information previously derived in the instantiation process. At the beginning of each new iteration, NS is assigned to ΔS, i.e. the new information derived during iteration n is considered as significant information for iteration $n+1$. Then, the recursive rules are processed simultaneously and each of them uses the information contained in the set ΔS; at the end of the iteration, when the evaluation of all rules is terminated, the set ΔS is added to the set S (since it has already been exploited). The evaluation stops whenever no new information has been derived (i.e. $NS = \emptyset$).

Single Rule Level. The techniques described above, concerning the first two levels of parallelism, were firstly employed in [18] and are very effective when handling long programs. However, when the input program consists of few rules, their efficacy is drastically reduced, and there are cases where components and rules parallelism are not exploitable at all.

Consider for instance the following program $\mathcal{P}$ encoding the well known 3-colorability problem:

$$\begin{array}{ll} (r) & col(X, red) \lor col(X, yellow) \lor col(X, green) \text{ :- } node(X). \\ (c) & \text{:- } edge(X, Y), col(X, C),\ col(Y, C). \end{array}$$

The two levels of parallelism described above have no effects on the evaluation of $\mathcal{P}$. Indeed, this encoding consists of only two rules which have to be evaluated sequentially, since, intuitively, the instantiation of (r) produces the ground atoms with predicate col which are necessary for the evaluation of (c).

[3] Note that, for the sake of clarity, a simplified version of the technique presented in [18] has been described.

For the instantiation of this kind of programs a third level is necessary for the parallel evaluation of each single rule, which is therefore called *Single Rule Level*. To this aim, a strategy has been presented in [19] which allows for parallelizing the evaluation of a rule on the base of a dynamic rewriting of the program. Oversimplifying, the basic idea of single rule level parallelism consists in rewriting the program rules into a number of new rules whose evaluation can be performed simultaneously by applying the techniques described above.

For instance, rule (c) in the previous example can be rewritten as follows:

$$\begin{array}{ll}(c_1) & \text{:-}\ edge_1(X,Y), col(X,C),\ col(Y,C).\\ (c_2) & \text{:-}\ edge_2(X,Y), col(X,C),\ col(Y,C).\\ \ldots & \\ (c_n) & \text{:-}\ edge_n(X,Y), col(X,C),\ col(Y,C).\end{array}$$

by *splitting* the set of ground atoms with predicate $edge$ (also called the *extension* of $edge$), into a number of subsets. The obtained rules can be evaluated in parallel and the instantiation produced is equivalent (modulo renaming) to the original one. However, in general, many ways for rewriting a program may exist (for instance, in the case of (c), col can be split up instead of $edge$) and the choice of the literal to split has to be carefully made, since it may strongly affect the cost of the instantiation of rules. 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). Moreover, if the predicate to be split is an IDB predicate (as in the case col) a static rewriting would lead to quite complex encodings possibly requiring a slower instantiation; in this case a rewriting performed at running time is more suitable, since it can be applied when the extension of the IDB predicate has already been computed.

In our system, rules are rewritten at execution time, thus dynamically distributing the workload among processing units, and a heuristics is used for determining the literal to split. More in detail, the strategy works as follows: a rule r is rewritten at execution time by splitting the extension of one single body (either EDB or IDB) predicate p of r (chosen according to a heuristics) in several parts. Each part is associated with a different temporary predicate; and, for each of those predicates, say p_i, a new rule called *split rule*, obtained by replacing p with p_i, is produced. The so-created rules will be instantiated in parallel in place of r (temporary predicate names are recognized when output is produced and replaced with original names in order to obtain the same output of the standard algorithm). Hereafter, we refer to the number of split rules as *split number* (or, equivalently, *number of splits*), and to the size of the extensions of each split predicate as *split size*.

Concerning the selection of the literal to be 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. A pragmatic choice is to select an optimal ordering and split the first literal in this order. Note that, since the instantiation of a body rule basically follows a nested-tuple strategy proceeding from left to right, splitting on the first literal minimizes the number of repeated match operations (see [19] for further insights). Since the ordering problem has already been investigated and an effective strategy [15] 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.

2.2 Load Balancing and Granularity Control

An advanced implementation of a parallel system has to deal with two important issues that strongly affect the performance: load balancing and granularity control. 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).

In this respect, the number of splits allowed for each rule directly determines the split size and, thus the “amount of work” assigned to different threads. As an example, consider the case in which we are running on a two processor machine the instantiation of a rule r and that, by applying dynamic rewriting, r is rewritten into two split rules. Assume also that the extension of the split predicate of r is divided into two subsets with, approximatively, the same size. Then, each split rule will be processed by a thread; and the two threads will possibly run separately on the two available processors. For limiting the inactivity time of the processors, it would be desirable that the threads terminate their execution almost at the same time. Unfortunately, this is not always the case, because subdividing the extension of the split predicate in equal parts does not ensure that the workload is equally spread between threads. However, if we consider a larger number of split, a further subdivision of the workload will be implied, and, the inactivity time would be more likely limited.

Clearly, it is crucial to guarantee that the parallel instantiation of a rule is not more time-consuming than its serial instantiation; and that an unbalanced workload distribution does not introduce significant delays and limits the overall performance. Nevertheless, it is necessary to control the number of threads, in order to save system resources. In order to satisfy all these requirements, (i) we imposed a limit to the number of concurrently running threads which is user-defined (an adequate setting is a multiple of the number of available CPUs); and, (ii) we devised and tuned a heuristics that selects dynamically the size of the split depending on the rule at hand (and different rules in the same programs may be assigned to different split sizes).

In detail, our method computes a heuristic value $\mathcal{W}(r)$ that acts as a litmus paper indicating the amount of work required for evaluating each rule r of the program, and so, its “hardness”, just before its instantiation; then, it uses $\mathcal{W}(r)$ to decide an appropriate split size. In particular, the size of the split should be sufficiently large to avoid thread management overhead (granularity control); and sufficiently small to exploit the preemptive multitasking scheduler of the operating system for obtaining a good workload distribution (load balancing).

Granularity Control is obtained by comparing, before instantiating each rule r, $\mathcal{W}(r)$ to an empirically-determined threshold w_{seq}; if $\mathcal{W}(r) > w_{seq}$ then the rule is scheduled for parallel instantiation, otherwise a sequential instantiation is performed. The idea is that: it is more convenient to perform a sequential instantiation of "very easy" rules since the overhead introduced by threads might be larger than their expected evaluation time.

Load Balancing is obtained by splitting rules in equally-sized splits. In particular, each rule is split by dividing the extension of the first predicate by a number which is multiple of the number of processors. This strategy resulted to be sufficient in most cases, but required a refinement in the case of "very hard" rules. In particular, when a rule is assessed to be "hard" by comparing the estimated work with another empirically-determined threshold ($\mathcal{W}(r) > w_{eq}$), a finer work distribution (exploiting a unary split size) is performed for the last $s - n_p$ splits, where s is the number of split and n_p is the number of processors. The intuition here is that, if a rule is hard to instantiate then it is more likely that its splits are also hard, and thus an uneven distribution of the splits to the available processors in the last part of the computation might cause a sensible loss of efficiency. Thus, further subdividing the last "hard" splits, may help to distribute in a finer way the workload in the last part of the computation.

Computation of heuristic values. $\mathcal{W}(r)$ is obtained by combining two estimations: $\mathcal{J}(r)$ and $\mathcal{C}(r)$. First, note that computing all the possible instantiations of a rule is equivalent to calculate all the answers of a conjunctive query. Thus, we considered $\mathcal{J}(r)$ that is an estimation of the size of the join corresponding to the evaluation of the body of r. Moreover, since in the instantiation of rules with several join variables the running time is mostly due to variable matching, we considered $\mathcal{C}(r)$ that is an estimation of the number of comparisons made by the instantiation algorithm (roughly, we considered $\mathcal{C}(r)$ because even producing a small output might require a considerable amount of time due to many matching failures). The two components of $\mathcal{W}(r)$ are estimated as follows:

– *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 [20] as follows:

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

where $T(R)$ is the number of tuples in R, and $V(X,R)$ (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 pair of body predicates according to a given evaluation order for computing $\mathcal{J}(r)$. The interested reader can find a more detailed discussion on this estimation in [20].

– *Number of comparisons:* an approximation of the number of comparisons done for instantiating a rule r is:

$$\mathcal{C}(r) = \sum_{X \in \mathcal{X}(r)} \prod_{L \in \mathcal{L}(r,X)} V(X,L)$$

where $\mathcal{X}(r)$ is the set of variables that appear in at least two literals in the body of r, $\mathcal{L}(R, X)$ 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 the sum of the product of the number of distinct values assumed by each join variable in r.

3 Experiments

In order to assess the performance of our parallel instantiator we carried out an experimental activity, reported in this section.

The machine used for the experiments is a two-processor Intel Xeon “Woodcrest” (quad core) 3GHz machine with 4MB of L2 Cache and 4GB of RAM, running Debian GNU Linux 4.0. We measured the efficiency of the instantiator, and test its behavior when the number of available CPUs is between 2 and 8. To do so, we exploited the available hardware, enabling several fixed numbers of CPUs. We enabled/disabled the CPUs by running the following bash Linux commands:

```
echo 0 >> /sys/devices/system/cpu/cpu-n/online
```

that disables the $cpu - n$ CPU; and

```
echo 1 >> /sys/devices/system/cpu/cpu-n/online
```

to re-enable the same CPU.

Since our techniques focus on instantiation, all the results of the experimental analysis refer only to the instantiation process rather than the whole process of computing answer sets; in addition, the time spent before the grounding stage (parser and preliminary operations) is obviously the same both for parallel and non-parallel version. In order to obtain more trustworthy results, each single experiment has been repeated five times.

In the following, we briefly describe both benchmark problems and data. In order to meet space constraints, encodings are not presented but they are available, together with the employed instances, and the binaries, at `http://www.mat.unical.it/ricca/downloads/parallelground09.zip`. Rather, to help the understanding of the results, some information is given on the number of rules of each program.

3.1 Benchmark Problems and Data

The benchmark problems can be grouped into two different classes: the first class is composed of some well know combinatorial problems, namely Ramsey Numbers, 3-Colorability, Hamiltonian Path, Reachability, and N-Queens. These benchmark problems have been already used for assessing ASP instantiator performance ([3, 21]). Problems belonging to this class are, indeed, particularly difficult to parallelize due to the compactness of their encodings; note also that, such kind of programs are quite common given the declarative nature of the ASP language which allows to compactly encode even very hard problems. About data, we considered five instances of increasing difficulty for each problem, except for the Hamiltonian Path problem, for which we considered thirteen instances of increasing size; and, for obtaining more significant results, we considered instances where the instantiation time is non negligible.

	Instantiation time							
Problem	serial	2 proc	3 proc	4 proc	5 proc	6 proc	7 proc	8 proc
$queens_1$	4.31 (0,03)	2.28 (0.01)	1.55 (0.04)	1.18 (0.01)	0.97 (0.02)	0.81 (0.01)	0.71 (0.01)	0.63 (0.01)
$queens_2$	5.43 (0,01)	2.80 (0.01)	1.89 (0.00)	1.44 (0.01)	1.17 (0.01)	1.00 (0.01)	0.87 (0.01)	0.78 (0.02)
$queens_3$	6.66 (0,05)	3.40 (0.01)	2.29 (0.01)	1.76 (0.01)	1.45 (0.07)	1.20 (0.01)	1.04 (0.01)	0.93 (0.02)
$queens_4$	7.96 (0,03)	4.25 (0.26)	2.83 (0.12)	2.10 (0.02)	1.70 (0.01)	1.43 (0.01)	1.26 (0.00)	1.11 (0.00)
$queens_5$	9.48 (0,04)	4.87 (0.01)	3.30 (0.03)	2.48 (0.01)	2.02 (0.01)	1.70 (0.01)	1.48 (0.02)	1.32 (0.01)
$ramsey_1$	377.36 (0.05)	194.20 (0.48)	129.61 (0.22)	97.95 (0.48)	78.37 (0.15)	65.85 (0.15)	56.91 (0.26)	50.44 (0.20)
$ramsey_2$	485.88 (0.13)	251.09 (0.49)	167.49 (0.30)	126.34 (0.25)	101.47 (0.31)	85.07 (0.23)	73.48 (0.40)	65.27 (0.85)
$ramsey_3$	616.81 (0.21)	319.17 (0.59)	212.29 (0.41)	159.95 (0.31)	129.00 (0.73)	107.96 (0.28)	93.21 (0.35)	82.07 (0.18)
$ramsey_4$	790.51 (0.17)	405.63 (0.73)	270.15 (0.64)	203.77 (0.24)	163.75 (0.32)	137.36 (0.67)	118.94 (0.42)	104.31 (0.25)
$ramsey_5$	944.18 (0.09)	485.88 (0.74)	323.48 (0.25)	243.63 (0.60)	195.89 (0.24)	164.51 (0.46)	141.69 (0.48)	124.76 (0.45)
$3col_1$	87.29 (0.08)	40.65 (0.28)	27.30 (0.52)	21.14 (0.18)	17.03 (0.14)	14.52 (0.04)	12.74 (0.09)	11.41 (0.10)
$3col_2$	145.50 (0.10)	67.61 (1.56)	45.15 (0.77)	35.47 (1.01)	27.59 (0.30)	23.75 (0.32)	20.37 (0.42)	18.45 (0.28)
$3col_3$	247.71 (0.23)	114.08 (3.72)	73.12 (2.56)	56.85 (1.46)	43.90 (0.66)	38.31 (1.14)	33.08 (0.95)	29.06 (0.43)
$3col_4$	375.72 (0.15)	171.26 (5.87)	112.64 (1.74)	87.46 (2.09)	70.94 (1.00)	59.82 (1.33)	50.83 (0.55)	45.16 (0.53)
$3col_5$	612.98 (0.19)	270.80 (14.10)	174.56 (3.44)	133.72 (2.43)	106.02 (0.87)	90.48 (3.39)	79.32 (1.82)	68.60 (1.85)
$reach_1$	74.30 (0.10)	32.05 (0.22)	21.29 (0.05)	16.27 (0.09)	13.24 (0.13)	11.17 (0.08)	9.69 (0.06)	8.55 (0.02)
$reach_2$	224.52 (0.37)	93.69 (0.35)	62.60 (0.06)	47.35 (0.04)	38.02 (0.22)	32.10 (0.21)	27.64 (0.05)	24.44 (0.12)
$reach_3$	325.58 (0.18)	137.52 (0.30)	92.00 (0.16)	69.18 (0.09)	55.80 (0.04)	46.82 (0.19)	40.38 (0.08)	35.63 (0.17)
$reach_4$	731.09 (0.32)	306.14 (1.12)	203.63 (0.17)	153.39 (0.49)	123.02 (0.34)	103.12 (0.31)	89.30 (0.29)	78.25 (0.15)
$reach_5$	1431.54 (0.66)	591.13 (1.62)	393.29 (0.34)	295.59 (0.32)	237.49 (0.61)	197.96 (0.38)	170.79 (0.49)	149.70 (0.40)
$timetabling_1$	46.80 (0.23)	20.99 (0.37)	14.03 (0.19)	10.63 (0.14)	8.55 (0.05)	7.22 (0.10)	6.33 (0.03)	5.59 (0.03)
$timetabling_2$	58.93 (0.39)	26.15 (0.25)	17.40 (0.28)	13.32 (0.20)	10.83 (0.11)	8.99 (0.11)	7.87 (0.09)	6.96 (0.07)
$WorkflowRepair$	684.95 (1.19)	0.22 (0.15)	0.08 (0.01)	0.07 (0.01)	0.06 (0.01)	0.06 (0.00)	0.06 (0.00)	0.07 (0.00)

Table 1. Benchmark Results: instantiation times in seconds (standard deviation)

In the following we briefly describe the benchmark problems belonging to the first class, and report quantitative information on the encodings, and the size of the benchmarks data. In particular:

– the encoding of *Ramsey Numbers* consists of one rule and two constraints; for the experiments, the problem was considered of deciding whether, for $k = 7$, $m = 7$, and $n \in \{31, 32, 33, 34, 35\}$, n is the Ramsey number $ramsey(k, m)$.
– The encoding of *3-Colorability* consists of one rule and one constraint; three simplex graphs were generated with the Stanford GraphBase library [22], by using the function $simplex(n, n, -2, 0, 0, 0, 0)$, ($n \in \{150, 170, 190, 210, 230\}$).
– The encoding of *Reachability* consists of one exit rule and a recursive one; three trees were generated [23] having pair (number of levels, number of siblings): (9,3),(7,5), (14,2), (10,3) and (15,2), respectively.
– The encoding of *Hamiltonian Path* consists of several rules, one of these is recursive; instances were generated, by using a tool by Patrik Simons (cf. [24]), having 100, 1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000, 10000, 11000 and 12000 nodes, respectively.
– The encoding of *n-Queens* consists of one rule and four constraints; instances were considered having $n \in \{37, 39, 41, 43, 45\}$.

The second class of problems is composed of problems taken from some practical application of ASP, namely *Timetabling*, and *WorkflowRepair*. Timetabling is the problem of determining a timetable for some university lectures that have to be given in a week to some groups of students (the considered instances were provided by the University of Calabria); WorkflowRepair is the problem of generating plans for repairing faulty workflows. Workflows may comprise many activities. Repair actions are compensation, (re)do and replacement of activities (the considered instance contains 63 predicates, 56 components and 116 rules).

3.2 Experimental Results

The results of the experimental activities on the benchmark problems presented above are summarized in Table 1-2, and Figure 1-2. In order to study the performance of the system when the number of available processors increases, system was run on our 8-core machine in eight different settings where 2,3,4,5,6,7 and 8 CPU were respectively enabled. The instantiation times were measured and reported in Table 1 and Figure 2(b); whereas the (relative) efficiency of the system is reported in Table 2 and Figure 2(a). The results obtained in the case of Hamiltonian Path, which is the only problem for which we considered more than five instances, are reported separately in Figure 2. Finally, Figure 1 reports the average efficiency of the system for the problems of Table 1.[4]

The overall picture is very positive: the performance of the system is nearly optimal in most cases and efficiencies above 1 are measured in four domains out of seven. As one would expect, the efficiency of the system both slightly decreases when the number of processors increases -still remaining at a good level-, and rapidly increases going from very small instances (execution times below 2s) to larger ones (see Figure 1 and Figure 2).

A special case is the WorkflowRepair problem, showing an impressive efficiency (always above 3200), which revealed to be a case very easy to parallelize. This behavior can be explained by a different scheduling of the constraints performed by the serial version and the parallel one. In particular, this instance is inconsistent (there is a constraint always violated) and both versions stop the computation as soon as they recognize this fact. The scheduling performed by the parallel version allows the identification of this situation before the serial one since constraints are evaluated in parallel, while the serial version evaluates the inconsistent constraint later on. The super-linear speedup is already evident with two processors and efficiency peaks when three processors are enabled, after the execution times remain almost the same (see the last row of Table 1) since the execution is stopped basically at the same time.

The granularity control mechanism resulted to be effective in the Queens problem, where all the considered instances required less than 10 seconds of serial execution time. Indeed, the “very easy” disjunctive rule was always sequentially-evaluated in all the instances. Since the remaining constraints strictly depend on the result of the evaluation of the disjunctive rule, the unavoidable presence of a sequential part limited the final efficiency to a still acceptable 0.9 in the case of 8 processors.

A similar scenario can be observed in the case of Ramsey Numbers, where the positive impact of the load balancing heuristics becomes very evident. In fact, since the encoding is composed of few “very easy” rules and two “very hard” constraints, the heuristics selects a sequential evaluation for the rules, and dynamically applies the

[4] We did not report here the size of the ground programs produced by the compared implementations because we verified that they are basically the same (for both parallel and serial version).

	Efficiency						
Problem	2 proc	3 proc	4 proc	5 proc	6 proc	7 proc	8 proc
$queens_1$	0.95	0.93	0.91	0.89	0.89	0.87	0.86
$queens_2$	0.97	0.96	0.94	0.93	0.91	0.89	0.87
$queens_3$	0.97	0.96	0.94	0.91	0.92	0.91	0.89
$queens_4$	0.94	0.94	0.95	0.94	0.93	0.90	0.90
$queens_5$	0.97	0.96	0.96	0.94	0.93	0.92	0.90
$ramsey_1$	0.97	0.97	0.96	0.96	0.96	0.95	0.94
$ramsey_2$	0.97	0.97	0.96	0.96	0.95	0.94	0.93
$ramsey_3$	0.97	0.97	0.96	0.96	0.95	0.95	0.94
$ramsey_4$	0.97	0.98	0.97	0.97	0.96	0.95	0.95
$ramsey_5$	0.97	0.97	0.97	0.96	0.96	0.95	0.95
$3col_1$	1.07	1.07	1.03	1.03	1.00	0.98	0.96
$3col_2$	1.08	1.07	1.03	1.05	1.02	1.02	0.99
$3col_3$	1.09	1.13	1.09	1.13	1.08	1.07	1.07
$3col_4$	1.10	1.11	1.07	1.06	1.05	1.06	1.04
$3col_5$	1.13	1.17	1.15	1.16	1.13	1.10	1.12
$reach_1$	1.16	1.16	1.14	1.12	1.11	1.10	1.09
$reach_2$	1.20	1.20	1.19	1.18	1.17	1.16	1.15
$reach_3$	1.18	1.18	1.18	1.17	1.16	1.15	1.14
$reach_4$	1.19	1.20	1.19	1.19	1.18	1.17	1.17
$reach_5$	1.21	1.21	1.21	1.21	1.21	1.20	1.20
$timetabling_1$	1.11	1.11	1.10	1.09	1.08	1.06	1.05
$timetabling_2$	1.13	1.13	1.11	1.09	1.09	1.07	1.06
$WorkflowRepair$	4119.59	7552.58	6473.64	6042.07	5035.06	4315.76	3236.82

Table 2. Benchmark Results: efficiency

finer distribution of the last splits for the constraints. As a result, the system produces a well-balanced work subdivision, that allows for obtaining steady results with an average efficiency greater than 0.9 in all tested configurations (see Figure 1).

The very good performance (by looking at Figure 1 it can be noted that the average efficiency is always greater than 1 in this case) obtained in the case of Reachability is due to the dynamic workload distribution made in case of recursive rules.

Here the system benefits of the fact that instances are redistributed (with possibly different split sizes) at each different iteration of the semi naïve algorithm, and, still, the granularity control has some positive effect when the iteration of recursive rules has to compute very little domains. The load balancing method demonstrated to be effective also for 3-Colorability and the real-world Timetabling problem, where the performance of the system results to be good and stable thanks to a well-balanced distribution of the work.

The behavior of the system for instances of varying sizes was analyzed in more detail in the case of Hamiltonian Path. This was made possible by the availability of a generator (cf. [24]) that allowed for controlling the size of the generated instances. Looking at Figure 2(a) it is evident that the efficiency of the system rapidly reaches a good level (greater than 0.9) moving from small instances (requiring less that 2s of execution) to larger ones, and remains stable (the surface plotted in Figure 2(a) forms a sort of plateau). The corresponding gains are visible by looking at Figure 2(b), where, e.g. an instance of 10000 nodes is instantiated in 638.27 seconds by the serial system and in 87.11 seconds by the parallel one with 8-processor enabled.

Summarizing, the parallel instantiator behaved very well in all the considered instances. It showed superlinear speedups in the case of easy-to-parallelize instances and,

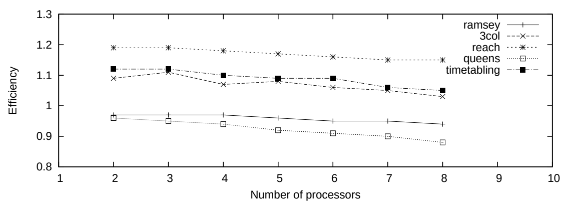


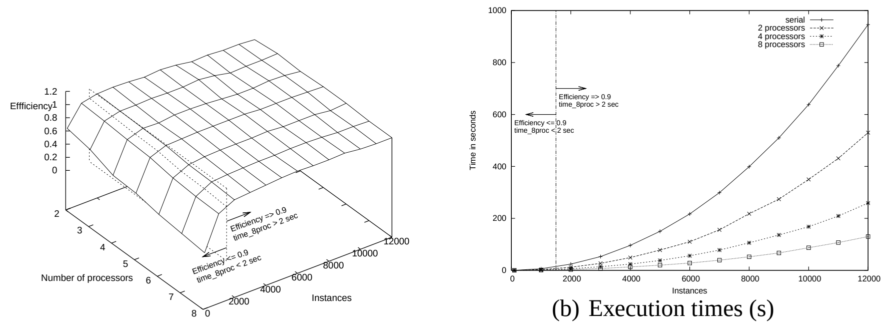
Fig. 1. Average Efficiency

in the other cases its efficiency rapidly reaches good levels and remains stable when the sizes of the input problem grow. Importantly, the system offers a very good performance already when only two CPUs were enabled (i.e. for the largest majority of the commercially-available hardware at the time of this writing) and efficiency remains at a very good level when up to 8 CPUs are available.

4 Related Work

Several works about parallel techniques for the evaluation of ASP programs have been proposed, focusing on both the propositional (model search) phase [25–29], and the instantiation phase [29, 18]. Model generation is a distinct phase of ASP computation, carried out after the instantiation, and thus, the first group of proposals is not directly related to our setting. Concerning the parallelization of the instantiation phase, some preliminary studies were carried out in [29]. However, there are crucial differences with our system regarding both the employed technology and the supported parallelization strategy. Indeed, our system is implemented by using POSIX threads APIs, and works in a shared memory architecture [17], while the one described in [29] is actually a Beowulf [30] cluster working in local memory. Moreover, the parallel instantiation strategy of [29] is applicable only to a subset of the program rules (those not defining domain predicates), and is, in general, unable to fruitfully exploit parallelism in case of programs with a small number of rules. Importantly, the parallelization strategy of [29] *statically* assigns a rule per processing unit; whereas, in our approach, both the extension of predicates and "split sizes" are dynamically computed (and updated at different iterations of the semi-naïve) while the instantiation process is running. Note also that our parallelization techniques and heuristics could be also adapted for improving the other ASP instantiators like Lparse [31] and Gringo [32].

Concerning other related works, it is worth remembering that, the dynamic rewriting technique employed in our system is related to the *copy and constrain* technique for parallelizing the evaluation of deductive databases [33–37]. In many of the mentioned works (dating back to 90's), only restricted classes of Datalog programs are parallelized; whereas, the most general ones (reported in [34, 36]) are applicable to normal Datalog programs. Clearly, none of them consider the peculiarities of disjunctive programs and unstratified negation. The technique employed in our system shares the idea of splitting the instantiation of each rule, but has several differences that allow for obtaining an effective implementation. Indeed, in [34, 36] copied rules are generated and statically



(a) Efficiency

(b) Execution times (s)

Fig. 2. Efficiency results for Hamiltonian path

associated to instantiators according to a 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 [36] 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 incur in a possible non negligible overhead due to their evaluation.

Focusing on the *heuristics* employed on parallel databases, we mention [37] and [38]. In both cases, the proposed heuristics were devised and tuned for dealing with data distributed in several sites and their application to other architectures might be neither viable nor straightforward.

5 Conclusions

In this paper, we presented a parallel ASP instantiator based on the DLV system which is able to profitably exploit state-of-the-art commercial multi-core/multi-processor hardware. The system employs several parallelization strategies and dynamic techniques for load balancing and granularity control specifically-conceived for parallel ASP instantiation. An experimental analysis has been conducted on both easy and hard-to-parallelize problem instances for assessing system performance. The results confirm that multi-core/multi-processor technology can be effectively exploited for ASP instantiation; indeed, the parallel system showed a nearly-optimal efficiency in the case of hard-to-parallelize problem instances, and superlinear speedups in other cases.

As far as future work is concerned, we are studying other techniques for further improving the single rule level parallelism. Moreover, we are assessing system performance on a larger set of benchmarks.

References

1. Gelfond, M., Lifschitz, V.: Classical Negation in Logic Programs and Disjunctive Databases. NGC **9** (1991) 365–385
2. Lifschitz, V.: Answer Set Planning. In: Proceedings of the 16th International Conference on Logic Programming (ICLP'99) 23–37
3. 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
4. Dantsin, E., Eiter, T., Gottlob, G., Voronkov, A.: Complexity and Expressive Power of Logic Programming. ACM Computing Surveys **33**(3) (2001) 374–425
5. Janhunen, T., Niemelä, I.: Gnt - a solver for disjunctive logic programs. In: LPNMR-7. LNCS 2923, (2004) 331–335
6. Lierler, Y.: Disjunctive Answer Set Programming via Satisfiability. In: LPNMR'05. LNCS 3662, (2005) 447–451
7. Simons, P., Niemelä, I., Soininen, T.: Extending and Implementing the Stable Model Semantics. AI **138** (2002) 181–234
8. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: Conflict-driven answer set solving. In: IJCAI 2007,(2007) 386–392
9. Lin, F., Zhao, Y.: ASSAT: computing answer sets of a logic program by SAT solvers. AI **157**(1–2) (2004) 115–137
10. Lierler, Y., Maratea, M.: Cmodels-2: SAT-based Answer Set Solver Enhanced to Non-tight Programs. In: LPNMR-7. LNCS 2923, (2004) 346–350
11. Anger, C., Konczak, K., Linke, T.: `NoMoRe`: A System for Non-Monotonic Reasoning. In: LPNMR'01. LNCS 2173, (2001) 406–410
12. Anger, C., Gebser, M., Linke, T., Neumann, A., Schaub, T.: The nomore++ Approach to Answer Set Solving. In: Logic for Programming, Artificial Intelligence, and Reasoning, 12th International Conference, LPAR 2005. LNCS 3835, (2005) 95–109
13. Eiter, T., Leone, N., Mateis, C., Pfeifer, G., Scarcello, F.: A Deductive System for Nonmonotonic Reasoning. In: LPNMR'97. LNCS 1265, Dagstuhl, Germany, (1997) 363–374
14. 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
15. Leone, N., Perri, S., Scarcello, F.: Improving ASP Instantiators by Join-Ordering Methods. In: LPNMR'01. LNCS 2173, (2001) 280–294
16. Leone, N., Perri, S., Scarcello, F.: BackJumping Techniques for Rules Instantiation in the DLV System. In: NMR 2004. (2004) 258–266
17. Stallings, W.: Operating systems (3rd ed.): internals and design principles. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1998)
18. 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
19. Perri, S., Ricca, F., Vescio, S.: Efficient Parallel ASP Instantiation via Dynamic Rewriting. In: Proceedings of the First Workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP 2008), Udine, Italy (2008)
20. Ullman, J.D.: Principles of Database and Knowledge Base Systems. Computer Science Press (1989)
21. Gebser, M., Liu, L., Namasivayam, G., Neumann, A., Schaub, T., Truszczyński, M.: The first answer set programming system competition. In: LPNMR'07. LNCS 4483, (2007) 3–17
22. Knuth, D.E.: The Stanford GraphBase : A Platform for Combinatorial Computing. ACM Press, New York (1994)

23. Terracina, G., Leone, N., Lio, V., Panetta, C.: Experimenting with recursive queries in database and logic programming systems. TPLP **8** (2008) 129–165
24. Simons, P.: Extending and Implementing the Stable Model Semantics. PhD thesis, Helsinki University of Technology, Finland (2000)
25. Finkel, R.A., Marek, V.W., Moore, N., Truszczynski, M.: Computing stable models in parallel. In: Answer Set Programming, Towards Efficient and Scalable Knowledge Representation and Reasoning, Proceedings of the 1st Intl. ASP'01 Workshop, Stanford (2001) 72–76
26. Ellguth, E., Gebser, M., Gusowski, M., Kaufmann, B., Kaminski, R., Liske, S., Schaub, T., Schneidenbach, L., Schnor, B.: A simple distributed conflict-driven answer set solver. In: LPNMR. LNCS 5753, (2009) 490–495
27. Gressmann, J., Janhunen, T., Mercer, R.E., Schaub, T., Thiele, S., Tichy, R.: Platypus: A Platform for Distributed Answer Set Solving. In: Proceedings of Logic Programming and Nonmonotonic Reasoning, 8th International Conference (LPNMR), Diamante, Italy (2005) 227–239
28. Pontelli, E., El-Khatib, O.: Exploiting Vertical Parallelism from Answer Set Programs. In: Answer Set Programming, Towards Efficient and Scalable Knowledge Representation and Reasoning, Proceedings of the 1st Intl. ASP'01 Workshop, Stanford (2001) 174–180
29. Balduccini, M., Pontelli, E., Elkhatib, O., Le, H.: Issues in parallel execution of non-monotonic reasoning systems. Parallel Computing **31**(6) (2005) 608–647
30. : The Beowulf Cluster Site `<URL:http://www.beowulf.org>`.
31. Niemelä, I., Simons, P.: Smodels – An Implementation of the Stable Model and Well-founded Semantics for Normal Logic Programs. In: LPNMR'97. LNCS 1265, Dagstuhl, Germany, (1997) 420–429
32. Gebser, M., Schaub, T., Thiele, S.: GrinGo : A New Grounder for Answer Set Programming. In: Logic Programming and Nonmonotonic Reasoning, 9th International Conference, LPNMR 2007, 15-17, 2007, Proceedings. LNCS 4483, (2007) 266–271
33. Wolfson, O., Silberschatz, A.: Distributed Processing of Logic Programs. In: Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA (1988) 329–336
34. Wolfson, O., Ozeri, A.: A new paradigm for parallel and distributed rule-processing. In: SIGMOD Conference 1990, New York, USA (1990) 133–142
35. 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
36. Zhang, W., Wang, K., Chau, S.C.: Data Partition and Parallel Evaluation of Datalog Programs. IEEE TKDE **7**(1) (1995) 163–176
37. Dewan, H.M., Stolfo, S.J., Hernández, M., Hwang, J.J.: Predictive dynamic load balancing of parallel and distributed rule and query processing. In: Proceedings of the 1994 ACM SIGMOD international conference on Management of data, New York, USA, ACM (1994) 277–288
38. Carey, M.J., Lu, H.: Load balancing in a locally distributed db system. SIGMOD Rec. **15**(2) (1986) 108–119