Advanced Heuristics for Parallel ASP Instantiation

Simona Perri[1], Francesco Ricca[1], and Marco Sirianni[1]

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

Abstract

The evaluation of ASP programs is traditionally carried out in two steps. The first is called instantiation or grounding, and consists on the computation of a ground program equivalent to the input one that, in turn, is evaluated by using a backtracking search algorithm in the second phase. Instantiation is important for the efficiency of the whole evaluation, might becomes a bottleneck in common situations, and is particularly crucial when huge input data has to be dealt with. Notably, performance improvements can be obtained by developing parallel systems, which exploit modern multi-core multi-processor machines.

In this paper, we describe a dynamic heuristics for load balancing and granularity control devised for improving parallel instantiation systems. We implemented the new technique in the parallel instantiator based on the DLV system, and conducted an experimental analysis that confirms its efficacy.

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, that was devised for single-processor machines, is unable to exploit the power of SMP architectures. Recently [2, 3, 4, 5, 6, 7, 8], the parallel evaluation technology has been exploited for implementing faster evaluation systems in the field of Answer Set Programming (ASP). ASP [9, 10, 11, 12, 13, 14] is a declarative approach to programming proposed in the area of nonmonotonic reasoning and logic programming which features a declarative nature combined with a relatively high expressive power [15, 16].

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 expensive (see [14, 16]); 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. [17, 18, 19, 20]), have evidenced the practical need for

Proceedings of the 16th International RCRA workshop (RCRA 2009):
Experimental Evaluation of Algorithms for Solving Problems with Combinatorial Explosion
Reggio Emilia, Italy, 11-12 December 2009

faster instantiators. It is easy to see that the exploitation of SMP systems in the grounding process can bring significant performance improvements. Indeed, an effective technique for the parallel instantiation of ASP programs exploiting SMP systems was proposed in [7].

However, the efficacy of this method was limited to programs with many rules, since, roughly, it allows for instantiating independent rules in parallel; but, a rewriting technique has been proposed in [8] that modifies the input program in such a way that the technique of [7] becomes applicable also in case of programs with few rules. The basic idea of [8] is to rewrite input rules at execution time in order to induce a form of or-parallelism. This can be obtained, given a rule r, by "splitting" the extension of one single body predicate p of r in several parts. Each part is associated with a different temporary predicate; and, for each of those predicates, say p_i, a new rule, obtained by replacing p with p_i, is produced. The so-created rules are instantiated in parallel in place of r by exploiting the parallel algorithm of [7] (a trivial realign step gets rid of the new names to obtain the intended output). This rewriting technique can be exploited by any available parallel ASP instantiator [3, 7], and it was successfully implemented [8] in a parallel ASP instantiator based on DLV [15]. Here the number of splits per rule was set to a global user-defined value, and the same value is used for each rule in input. Clearly, this simple strategy does not work well might in several cases. Indeed, if each process receives a "too small" amount of work, then the costs added by parallel execution may become larger than the benefits (because of thread creation and scheduling overhead); on the other hand, if the amount of work assigned to threads is "too large", then a resulting bad workload distribution will reduce the advantages of parallel evaluation. In this paper, we propose an advanced heuristics that is able to improve the efficiency of the parallel evaluation by automatically determining, rule by rule, the amount of work that has to be assigned to each parallel instantiator. Moreover, we implemented our heuristics in the parallel ASP instantiator of [8], and we report here the results of an experimental analysis that confirms the efficacy of the proposed method.

2 Answer Set Programming

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

Syntax. A variable or a constant is a *term*. An *atom* is $a(t_1, ..., t_n)$, where a is a *predicate* of arity n and $t_1, ..., t_n$ are terms. A *literal* is either a *positive literal* p or a *negative literal* `not` p, where p is an atom. 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 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 $b_1, ..., b_k$, `not` $b_{k+1}, ...,$ `not` b_m is the *body* of r. A rule without head literals (i.e. $n = 0$) is usually referred to as an *integrity constraint*. If the body is empty (i.e. $k = m = 0$), it is called a *fact*. $H(r)$ denotes the set $\{a_1, ..., a_n\}$ of the head atoms, and by $B(r)$ the set $\{b_1, ..., b_k,$

$\texttt{not}\ b_{k+1}, \ldots, \texttt{not}\ b_m\}$ of the body literals. $B^+(r)$ (resp., $B^-(r)$) denotes the set of atoms occurring positively (resp., negatively) in $B(r)$. A rule r is *safe* if each variable appearing in r appears also in some positive body literal of r.

An *ASP program* $\mathcal{P}$ is a finite set of safe rules. An atom, a literal, a rule, or a program is *ground* if no variables appear in it. Accordingly with the database terminology, a predicate occurring only in *facts* is referred to as an *EDB* predicate, all others as *IDB* predicates; the set of facts of $\mathcal{P}$ is denoted by $EDB(\mathcal{P})$.

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

Given a rule r occurring in $\mathcal{P}$, a *ground instance* of r is a rule obtained from r by replacing every variable X in r by $\sigma(X)$, where σ is a substitution mapping the variables occurring in r to constants in $U_\mathcal{P}$; $ground(\mathcal{P})$ denotes the set of all the ground instances of the rules occurring in $\mathcal{P}$.

An *interpretation* for $\mathcal{P}$ is a set of ground atoms, that is, an interpretation is a subset I of $B_\mathcal{P}$. A ground positive literal A is *true* (resp., *false*) w.r.t. I if $A \in I$ (resp., $A \notin I$). A ground negative literal $\texttt{not}\ A$ 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 ground program $\mathcal{P}$ and an interpretation I, the *reduct* of $\mathcal{P}$ w.r.t. I is the subset $\mathcal{P}^I$ of $\mathcal{P}$, which is obtained from $\mathcal{P}$ by deleting rules in which a body literal is false w.r.t. I. Note that the above definition of reduct, proposed in [21], simplifies the original definition of Gelfond-Lifschitz (GL) transform [9], but is fully equivalent to the GL transform for the definition of answer sets [21].

Let I be an interpretation for a program $\mathcal{P}$. I is an *answer set* (or stable model) for $\mathcal{P}$ if $I \in \mathrm{MM}(\mathcal{P}^I)$ (i.e., I is a minimal model for the program $\mathcal{P}^I$) [22, 9]. The set of all answer sets for $\mathcal{P}$ is denoted by $ANS(\mathcal{P})$.

3 Parallel Instantiation of ASP Programs

In this Section we briefly recall some recently proposed techniques ([7, 8]) for the parallel instantiation of ASP Programs. 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 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. A detailed description of these techniques is out of the scope of this paper. For further details, we refer the reader to [7, 8].

3.1 Components Level

The first level of parallelism, called *Components Level*, has been described in [7] and, essentially, it consists in 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 p such that q occurs in the head of r and p occurs in a positive literal of the body of r.

The graph $G_{\mathcal{P}}$ induces a subdivision of $\mathcal{P}$ into subprograms (also called *modules*) allowing for a modular evaluation. We say that a rule $r \in \mathcal{P}$ *defines* a predicate p if p appears in the head of r. For each strongly connected component (SCC) [1]C of $G_{\mathcal{P}}$, the 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 other, then the instantiation of the corresponding program modules can be performed simultaneously, because the instantiation of A does not require

[1] A strongly connected component of a directed graph is a maximal subset of the vertices, such that every vertex is reachable from every other vertex.

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. Note that, for the sake of clarity, a simplified version of the technique presented in [7] has been described. The original one is quite more involved and takes into account also negative dependencies among predicates. Many details have been omitted since they do not give additional insight for the comprehension of the idea underlying the technique.

3.2 Rules Level

Concerning the second level of parallelism, the *Rules Level*, in [7] a technique has been presented allowing for concurrently evaluating the rules within each module. According to this technique, rules are evaluated following a semi-naïve schema [23] 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. $NS = \emptyset$).

3.3 Single Rule Level

The techniques described above, concerning the first two levels of parallelism, are very effective when handling with long programs, as confirmed also by the experimental analysis conducted in [7]. However, when the input program consists of few rules, their efficacy is drastically reduced, and there are cases where components and rules parallelism is not exploitable at all.

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

colorability problem:

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

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).

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 [8] which allows for parallelizing the evaluation of a rule on the base of a dynamic rewriting of the program. Oversimplifying, the basic idea of [8] 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 [8]:

$$(c_1) \quad \text{ :- } edge_1(X, Y), col(X, C),\ col(Y, C).$$
$$(c_2) \quad \text{ :- } edge_2(X, Y), col(X, C),\ col(Y, C).$$
$$\ldots$$
$$(c_n) \quad \text{ :- } edge_n(X, Y), col(X, C),\ col(Y, C).$$

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 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.

The technique in [8] solves both these issues, indeed, rules are rewritten at execution time, thus dynamically distributing the workload among processing units, and an 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 predicate p of r (chosen according to an 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; when their evaluation is completed, a realign step gets rid of the new names in order to obtain the same output of the original algorithm. Hereafter, we refer to

the number of split rules as *split number*, and to the size of the extensions of each split predicate as *split size*.

4 Heuristics for 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 parallel grounder described in [8] implements a naive strategy: each rule is rewritten in a globally fixed (specified by the user) number of splits. The number of splits allowed for each rule is (usually) the main source of concurrently running threads (roughly, the number of running threads is bounded by the number of generated split rules) and it directly determines the split size and, thus the "amount of work" assigned to threads. It is easy to see that this choice might be not the best one in several cases. 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. Moreover, it is not possible nor desirable, to let the user assessing a possible size of the split in order to obtain a balanced workload distribution, especially considering that it strictly depends by the rule at hand (and different rules in the same programs may require different split sizes); rather, a better policy for load balancing and granularity control is necessary. Despite being crucial in distributed parallel architectures (like, e. g. , clusters), in our setting (i.e., shared memory processor), developing a sophisticated granularity-control strategy is not essential, as also observed in [24]; rather it is sufficient to set the split size to an adequate value for each rule. Clearly, 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). Importantly, the number of running threads has to be controlled in or-

der to save resources. In order to satisfy both requirements, (i) we modified the implementation of [8] so that the user can set the number of concurrently running threads; and, (ii) we devised 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. In detail, our method computes an 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 exploits $\mathcal{W}(r)$ to select the more appropriate split size among six settings: small, medium, large, extra-large, equally-sized split (i.e. the old technique), and no split (i.e. sequential evaluation). The choice is made by comparing $\mathcal{W}(r)$ with five empirically-determined thresholds (w_{seq}, w_{es}, w_{el}, w_l, w_m). Basically, the criterion is to evaluate "very easy" rules sequentially (if $\mathcal{W}(r) < w_{seq}$), since the overhead introduced by threads is higher their expected evaluation time (granularity control); "easy rules", whose computation can still exploit some parallelism, are evaluated using an equally-sized split (that is, the technique on [8]) for minimizing the overheads (if $\mathcal{W}(r) < w_{es}$); whereas, for harder and harder rules, smaller and smaller split sizes are employed for obtaining a finer distribution of work. $\mathcal{W}(r)$ is obtained by combining (actually summing) 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). We now detail how the two components of $\mathcal{W}(r)$ 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 [25] 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 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 [25].

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 the body of r.

5 Experiments

In order to assess the impact of the proposed heuristics, we implemented it in the system of [8], and carried out an experimental activity.

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. Experiments were performed on a collection of benchmark programs already used for assessing ASP instantiator performance ([15, 26]). In particular, we considered the following well-known problems: Ramsey Numbers, 3-Colorability, Hamiltonian Path and Reachability.

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/heur09.zip`. Rather, to help the understanding of the results, some information is given on the number of rules of each program.

5.1 Benchmark Problems and Data

For the experiments, we considered encodings belonging to a particularly difficult-to-parallelize class i.e. ASP encodings with few rules.[2] Note 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 for each problem five instances of increasing size; and, for obtaining more significant results, we considered instances where the instantiation time is non negligible.

Ramsey Numbers. The Ramsey number $ramsey(k, m)$ is the least integer n such that, no matter how the edges of the complete undirected graph (clique) with n nodes are colored using two colors, say red and blue, there is a red clique with k nodes (a red k-clique) or a blue clique with m nodes (a blue m-clique).The encoding of this problem 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 $ramsey(k, m)$.

3-Colorability. This well-known problem asks for an assignment of three colors to the nodes of a graph, in such a way that adjacent nodes always have different colors. The encoding of this problem consists of one rule and one constraint. Three simplex graphs were generated with the Stanford GraphBase library [27], by using the function $simplex(n, n, -2, 0, 0, 0, 0)$, ($n \in \{150, 170, 190, 210, 230\}$).

[2] The good behavior of the system on easy-to-parallelize instances (where superlinear speedups have to be expected) and program with many rules has already been reported in [7].

Problem	Serial	Old Technique	Heuristics	Gain	Speedup	Efficiency
$ramsey_1$	380.33 (0.93)	85.96 (6.36)	54.05 (0.51)	261,21%	704%	0.88
$ramsey_2$	491.18 (1.94)	113.36 (2.39)	67.69 (0.53)	292,34%	726%	0.90
$ramsey_3$	624.43 (2.05)	148.04 (7.26)	85.92 (0.53)	304,96%	727%	0.91
$ramsey_4$	794.30 (1.76)	181.36 (0.99)	108.72 (0.22)	292,62%	731%	0.91
$ramsey_5$	951.61 (1.50)	213.19 (4.10)	131.79 (0.98)	275.7%	722%	0.90
$3col_1$	96.37 (2.01)	12.90 (0.12)	11.56 (0.21)	86,6%	834%	1.04
$3col_2$	156.13 (4.27)	21.64 (0.95)	19.13 (0.19)	94.66%	816%	1.02
$3col_3$	257.32 (1.85)	33.90 (0.48)	29.97 (0.28)	99,54%	859%	1.07
$3col_4$	391.06 (3.44)	53.59 (1.12)	46.76 (0.34)	106,59%	836%	1.05
$3col_5$	595.58 (7.56)	77.37 (0.42)	67.48 (0.61)	112.82%	852%	1.10
$hampath_1$	209.56 (1.54)	30.48 (0.66)	29.03 (0.27)	34,34%	722%	0.90
$hampath_2$	266.35 (2.32)	37.38 (0.66)	35.47 (0.15)	38.37%	751%	0.94
$hampath_3$	328.54 (3.76)	45.51 (0.46)	43.13 (0.48)	39.84%	762%	0.95
$hampath_4$	406.55 (2,89)	56.97 (2.45)	53.90 (0.16)	40,65%	754%	0.94
$hampath_5$	501.4 (2.11)	69.14 (1.86)	65.44 (0.17)	41.00%	766%	0.96
$reach_1$	64.73 (1.05)	8.62 (0.10)	8.53 (0.04)	7.92%	759%	0.95
$reach_2$	191.52 (1.50)	24.61 (0.26)	24.68 (0.17)	2,53%	779%	0.97
$reach_3$	281.82 (1.98)	36.06 (0.29)	36.01 (0.25)	1.09%	783%	0.98
$reach_4$	613.94 (3.95)	79.25 (0.39)	78.97 (0.18)	2,26%	783%	0.97
$reach_5$	1216.62 (12.55)	151.79 (0.31)	151.49 (0.22)	1.59%	803%	1.00

Table 1: Benchmark Results: average instantiation times in seconds (standard deviation), percentage gain w.r.t the old instantiator, speedup and relative efficiency.

Reachability. Given a finite directed graph $G = (V, A)$, we want to compute all pairs of nodes $(a, b) \in V \times V$ (i) such that b is reachable from a through a nonempty sequence of arcs in A. The encoding of this problem consists of one exit rule and a recursive one. Tree trees were generated [28] having pair (number of levels, number of siblings): (9,3),(7,5),(14,2),(10,3) and (15,2), respectively.

Hamiltonian Path. A classical NP-complete problem in graph theory, which can be expressed as follows: given a directed graph $G = (V, E)$ and a node $a \in V$ of this graph, does there exist a path in G starting at a and passing through each node in V exactly once. The encoding of this problem consists of several rules, one of these is recursive. Instances were generated, by using a tool by Patrik Simons (cf. [29]), having 5800, 6500, 7200, 8000 and 8800 nodes, respectively.

5.2 Impact of The New Heuristics

In order to prove the efficacy of the method that is the subject of this work, we compared the performance of the instantiator equipped with the heuristics with the previous version implementing a simple dynamic rewriting. The results of the experiments are summarized in Table 1, where Columns 2-4 report the times obtained by the serial instantiator, the previous parallel instantiator, and the parallel instantiator enhanced with the heuristics, respectively; in addition, Column 5 shows the percentage gain obtained by the version with heuristics w.r.t the previous one (the speedup of the version with the heuristics w.r.t. serial execution minus the

speedup of the plain parallel version w.r.t. serial execution), and Columns 6-7 show the speedup and the relative efficiency, respectively.[3]

First of all, we notice that the version with the heuristics is basically the best performer and always overcomes the results obtained by the previous one with a percentage gain ranging from 1% in Reachability up to 300% in Ramsey. Such good results are mainly due to the selection of different split sizes for different rules in the same program.

More in details, in the case of the Reachability problem, the two parallel instantiators show very similar behaviors. Indeed the heuristics suggests for this problem to use the biggest split size (equally-split size) for most of the rules, which corresponds to the fixed setting imposed by the previous implementation and which already allowed very good results with a speed up of about 800% (and, thus, an efficiency of about 1). However, the heuristics still gives a little benefit thanks to the effects of the granularity control, which allows to compute sequentially very easy rules, thus avoiding some overhead for threads creation and scheduling.

Similar considerations hold for the Hamiltonian Path problem, even if, here, the effects of the heuristics are more evident. In this case, the system benefits of the fact that the heuristics may dynamically assign to the same recursive rule different split sizes in different iterations. In particular, the heuristics suggests splits sizes mainly varying between large and equally-split size, and, still, the granularity control has some positive effect when the iteration of recursive rules has to compute very little domains.

The positive impact of the heuristics becomes very evident in the case of the Ramsey Number problem. 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 the smallest split size possible for the constraints. As a result, the system produces a well-balanced work subdivision, that allows for improving its overall performance, reaching a speedup of 730% in the best case, thus resulting in a percentage gain w.r.t the previous system of about 300%.

Similar considerations hold for 3-Colorability. As for Ramsey Number, the encoding is composed of few "easy" rules, and an "hard" constraint; the heuristics selects equally-split size for the rules and a small split size for the constraint, which leads to a percentage gain of about 100% w.r.t. the previous instantiator and a speedup of more than 800%.

6 Related Work

Several works about parallel techniques for the evaluation of ASP programs have been proposed, focusing on both the propositional (model search) phase [5, 6, 4, 2],

[3] 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); thus, the good behavior (see [15]) of the grounding module of DLV (that is able to produce an output that is sensibly smaller than the theoretical ground instantiation) is preserved on its parallel version.

and the instantiation phase [3, 7]. 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 [3], as one of the aspects of the attempt to introduce parallelism in non-monotonic reasoning systems. 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 [1, 30], while the one described in [3] is actually a Beowulf [31] cluster working in local memory. Moreover, the parallel instantiation strategy of [3] 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 [3] *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 Lparse instantiator.

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 [32, 33, 34, 35, 36] (for a detailed comparison between the two approaches see [8]). Focusing on the *heuristics* employed on parallel databases, we mention [36] and [37]. In [36] is described an heuristics for balancing the distribution of load in the parallel evaluation of PARULEL, a language similar to Datalog. Here, load balancing is done by a manager server that records the execution times at each site, and exploits this information for distributing the load. In [37] the proposed heuristics were devised for both minimizing communication costs and choosing an opportune site for processing sub-queries among various network-connected database systems. In both cases, the proposed heuristics were devised and tuned for dealing with data distributed in several sites and their application to similar architectures might be neither viable nor straightforward.

7 Conclusions

In this paper, an advanced heuristics for load balancing and granularity control in the parallel instantiation of ASP programs has been proposed. The heuristics has been implemented in the parallel instantiator of [7, 8] based on the DLV system, and an experimental analysis has been conducted on hard-to-parallelize problem instances which confirms the efficacy of the method for improving the performance of the system. In particular, the parallel instantiator equipped with the new heuristics always improves the results obtained by the old version; and compared with the previous parallel method offers a very relevant gain especially in case of programs

with hard-to-instantiate rules/constraints.

As far as future work is concerned, we are experimenting for obtaining a finer tuning of the heuristics; and we are working on a procedure for the automatic calibration of the heuristics thresholds. Moreover, we are assessing the impact of the heuristics on a larger set of benchmarks.

References

[1] Stallings, W.: Operating systems (3rd ed.): internals and design principles. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1998)

[2] 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 (March 2001) 174–180

[3] Balduccini, M., Pontelli, E., Elkhatib, O., Le, H.: Issues in parallel execution of non-monotonic reasoning systems. Parallel Computing **31**(6) (2005) 608–647

[4] 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 (September 2005) 227–239

[5] 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 (March 2001) 72–76

[6] 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

[7] 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

[8] 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)

[9] Gelfond, M., Lifschitz, V.: Classical Negation in Logic Programs and Disjunctive Databases. New Generation Computing **9** (1991) 365–385

[10] Lifschitz, V.: Answer Set Planning. In: (ICLP'99) 23–37

[11] 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

[12] Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press (2003)

[13] Gelfond, M., Leone, N.: Logic Programming and Knowledge Representation – the A-Prolog perspective . Artificial Intelligence **138**(1–2) (2002) 3–38

[14] Eiter, T., Gottlob, G., Mannila, H.: Disjunctive Datalog. ACM Transactions on Database Systems **22**(3) (September 1997) 364–418

[15] 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) (July 2006) 499–562

[16] Dantsin, E., Eiter, T., Gottlob, G., Voronkov, A.: Complexity and Expressive Power of Logic Programming. ACM Computing Surveys **33**(3) (2001) 374–425

[17] 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 (June 2005) 915–917

[18] Curia, R., Ettorre, M., Gallucci, L., Iiritano, S., Rullo, P.: Textual Document Pre-Processing and Feature Extraction in OLEX. In: Proceedings of Data Mining 2005, Skiathos, Greece (2005)

[19] Massacci, F.: Computer Aided Security Requirements Engineering with ASP Non-monotonic Reasoning, ASP and Constraints, Seminar N 05171. Dagstuhl Seminar on Nonmonotonic Reasoning, Answer Set Programming and Constraints (September 2005)

[20] Ruffolo, M., Leone, N., Manna, M., Saccà, D., Zavatto, A.: Exploiting ASP for Semantic Information Extraction. In de Vos, M., Provetti, A., eds.: Proceedings ASP05 - Answer Set Programming: Advances in Theory and Implementation, Bath, UK (July 2005) 248–262

[21] Faber, W., Leone, N., Pfeifer, G.: Recursive aggregates in disjunctive logic programs: Semantics and complexity. In JELIA 2004. LNCS 3229, (2004) 200–212

[22] Przymusinski, T.C.: Stable Semantics for Disjunctive Programs. New Generation Computing **9** (1991) 401–424

[23] Ullman, J.D.: Principles of Database and Knowledge-Base Systems, Volume I. Computer Science Press (1988)

[24] Lopez, P., Hermenegildo, M., Debray, S.: A methodology for granularity-based control of parallelism in logic programs. J. Symb. Comput. **21**(4-6) (1996) 715–734

[25] Ullman, J.D.: Principles of Database and Knowledge Base Systems. Computer Science Press (1989)

[26] Gebser, M., Liu, L., Namasivayam, G., Neumann, A., Schaub, T., Truszczyński, M.: The first answer set programming system competition. In: LMNMR'07. LNCS 4483, (2007) 3–17

[27] Knuth, D.E.: The Stanford GraphBase : A Platform for Combinatorial Computing. ACM Press, New York (1994)

[28] Terracina, G., Leone, N., Lio, V., Panetta, C.: Experimenting with recursive queries in database and logic programming systems. TPLP **8** (2008) 129–165

[29] Simons, P.: Extending and Implementing the Stable Model Semantics. PhD thesis, Helsinki University of Technology, Finland (2000)

[30] Tanenbaum, A.S., Woodhull, A.S.: Operating Systems Design and Implementation (3rd Edition). Prentice-Hall, Inc., Upper Saddle River, NJ, USA (2005)

[31] : The Beowulf Cluster Site <URL:http://www.beowulf.org>.

[32] 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 (June 1988) 329–336

[33] Wolfson, O., Ozeri, A.: A new paradigm for parallel and distributed rule-processing. In: SIGMOND Conference 1990, New York, USA (1990) 133–142

[34] Ganguly, S., Silberschatz, A., Tsur, S.: A Framework for the Parallel Processing of Datalog Queries. In: SIGMOND Conference 1990, Atlantic City, NJ, 1990.(1990) 143–152

[35] Zhang, W., Wang, K., Chau, S.C.: Data Partition and Parallel Evaluation of Datalog Programs. IEEE TKDE **7**(1) (1995) 163–176

[36] Dewan, H. M., Stolfo, S. J., Hernndez, M., Hwang, J.: Predictive dynamic load balancing of parallel and distributed rule and query processing. In: Proc. of ACM SIGMOD'86, 277-288

[37] Carey, M.J., Lu, H.: Load balancing in a locally distributed db system. SIGMOD Rec. **15**(2) (1986) 108–119