TLP **13** (2): 253–278, 2013. © Cambridge University Press 2012
doi:10.1017/S1471068411000652 First published online 25 January 2012

Parallel instantiation of ASP programs: techniques and experiments

SIMONA PERRI, FRANCESCO RICCA, and MARCO SIRIANNI
Dipartimento di Matematica, Università della Calabria, 87030 Rende, Italy
(*e-mail:* {perri,ricca,sirianni}@mat.unical.it)

submitted 28 November 2010; revised 16 April 2011; accepted 31 August 2011

Abstract

Answer-Set Programming (ASP) is a powerful logic-based programming language, which is enjoying increasing interest within the scientific community and (very recently) in industry. The evaluation of Answer-Set Programs is traditionally carried out in two steps. At the first step, an input program $\mathscr{P}$ undergoes the so-called instantiation (or grounding) process, which produces a program $\mathscr{P}'$ semantically equivalent to $\mathscr{P}$, but not containing any variable; in turn, $\mathscr{P}'$ is evaluated by using a backtracking search algorithm in the second step. It is well-known that instantiation is important for the efficiency of the whole evaluation, might become a bottleneck in common situations, is crucial in several real-world applications, and is particularly relevant when huge input data have to be dealt with. At the time of this writing, the available instantiator modules are not able to exploit satisfactorily the latest hardware, featuring multi-core/multi-processor Symmetric MultiProcessing technologies. This paper presents some parallel instantiation techniques, including load-balancing and granularity control heuristics, which allow for the effective exploitation of the processing power offered by modern Symmetric MultiProcessing machines. This is confirmed by an extensive experimental analysis reported herein.

KEYWORDS: Answer-Set Programming, Instantiation, Parallelism, Heuristics

1 Introduction

Answer-Set Programming (ASP) (Gelfond and Lifschitz 1991; Eiter *et al.* 1997; Lifschitz 1999; Marek and Truszczyński 1999; Gelfond and Leone 2002; Baral 2003) is a purely declarative programming paradigm based on nonmonotonic reasoning and logic programming. The language of ASP is based on rules, allowing (in general) for both disjunction in rule heads and nonmonotonic negation in the body. The idea of ASP is to represent a given computational problem by a logic program such that its answer sets correspond to solutions, and then, use an answer-set solver to find such solutions (Lifschitz 1999). The main advantage of ASP is its declarative nature combined with a relatively high expressive power (Dantsin *et al.* 2001; Leone *et al.* 2006); but this comes at the price of an elevated 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 (Bell *et al.* 1994; Subrahmanian *et al.* 1995), there are nowadays a number of systems that support ASP and its

https://doi.org/10.1017/S1471068411000652 Published online by Cambridge University Press

variants (Anger *et al.* 2001; Simons *et al.* 2002; Lierler and Maratea 2004; Lin and Zhao 2004; Janhunen *et al.* 2006; Leone *et al.* 2006; Gebser *et al.* 2007a). The availability of efficient systems made ASP profitably exploitable in real-world applications (Lembo *et al.* 2002; Aiello and Massacci 2001; Baral and Uyan 2001) and, recently, also in industry (Ielpa *et al.* 2009; Grasso *et al.* 2010, 2011).

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. Therefore, an input program $\mathscr{P}$ first undergoes the so-called instantiation process, which produces a ground program $\mathscr{P}'$ semantically equivalent to $\mathscr{P}$. This phase, which is fundamental for some real-world applications where huge amounts of input data have to be handled, is computationally expensive (see Eiter *et al.* 1997; Dantsin *et al.* 2001); 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 (Faber *et al.* 1999; Leone *et al.* 2001, 2004); nevertheless, the performance of instantiators can be further improved 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) Symmetric MultiProcessing (SMP) (Stallings 1998) machines featuring several CPUs. 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 can be enjoyed 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 exploit fully the computational power offered by modern machines.

This paper presents[1] some advanced techniques for the parallel instantiation of ASP programs, implemented in an instantiator system allowing the exploitation of the computational power offered by multi-core/multi-processor machines. The system is based on the state-of-the-art ASP instantiator of the DLV system (Leone *et al.* 2006); moreover, it extends the work of (Calimeri *et al.* 2008) by introducing a number of relevant improvements: (*i*) an additional third stage of parallelism for the instantiation of every single rule of the program, (*ii*) dynamic load balancing, and (*iii*) granularity control strategies based on computationally cheap heuristics. In this way, the efficacy of the system is no longer limited to programs with many rules (as in Calimeri *et al.* 2008), and also the particularly (common and) difficult-to-parallelize class of programs with few rules is handled in an effective way. Moreover, we developed a new implementation supporting a richer input language (e.g., aggregates) and technical improvements in thread management.

An extensive experimental activity is also reported, which was carried out on a variety of publicly available benchmarks already exploited for evaluating the

[1] Preliminary results have been presented in (Perri *et al.* 2008, 2010).

https://doi.org/10.1017/S1471068411000652 Published online by Cambridge University Press

performance of instantiation systems (Leone *et al.* 2006; Gebser *et al.* 2007c; Denecker *et al.* 2009). A comparison with (Calimeri *et al.* 2008) shows that the new techniques both combine with the previous ones and allow for a parallel evaluation even in cases where previous techniques were not applicable.

A scalability analysis demonstrates that the new parallel instantiator behaves very well in all the considered instances: 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 (its efficiency remains stable when the size of the input problem grows). Importantly, the system offers a very good performance already when only two CPUs are enabled (i.e., for the majority of the commercially available hardware at the time of this writing) and efficiency remains at a good level when more CPUs are enabled.

The remainder of the paper is structured as follows: Section 2 outlines some basic notions of Answer-Set Programming; Section 3 describes the employed parallel instantiation strategies; Sections 4 and 5 present heuristics, load balancing, and granularity control; Section 6 discusses the results of the experiments; finally, Section 7 is devoted to related works, and Section 8 draws some conclusions.

2 Answer-set programming

In this section, we provide a formal definition of syntax and semantics of ASP programs.

Syntax. A variable or a constant is a *term*. An *atom* is $p(t_1,\ldots,t_n)$, where p is a *predicate* of arity n and $t_1,\ldots,t_n$ are terms. A *literal* is either a *positive literal* p or a *negative literal* `not` p, where p is an atom. A *(disjunctive) rule* r has the following form:

$$a_1 \vee \ldots \vee a_n \text{ :- } b_1,\ldots,b_k,\texttt{not } b_{k+1},\ldots,\texttt{not } b_m, \qquad n \geqslant 0,\ m \geqslant k \geqslant 0,$$

where $a_1,\ldots,a_n,b_1,\ldots,b_m$ are atoms. The disjunction $a_1 \vee \ldots \vee a_n$ is the *head* of r, while the conjunction $b_1,\ldots,b_k,\texttt{not } b_{k+1},\ldots,\texttt{not } b_m$ is the *body* of r.

We denote by $H(r)$ the set $\{a_1,\ldots,a_n\}$ of the head atoms, and by $B(r)$ the set $\{b_1,\ldots,\ b_k,\ \texttt{not } b_{k+1},\ldots,\texttt{not } b_m\}$ of the body literals. $B^+(r)$ (resp., $B^-(r)$) denotes the set of atoms occurring positively (resp., negatively) in $B(r)$.

A rule having precisely one head literal (i.e., $n = 1$) is called a *normal rule*. If the body is empty (i.e., $k = m = 0$), it is called a *fact*, and we usually omit the :- sign. A rule without head literals (i.e. $n = 0$) is usually referred to as an *integrity constraint*.[2] A rule r is *safe* if each variable appearing in r appears also in some positive body literal of r.

An *ASP program* $\mathscr{P}$ is a finite set of safe rules. A `not` -free (resp., $\vee$-free) program is called *positive* (resp., *normal*). A term, an atom, a literal, a rule, or a program is *ground* if no variables appear in it. A predicate p is referred to as an *EDB* predicate if, for each rule r having in the head an atom whose name is $p \in H(r)$, r is a fact; all others predicates are referred to as *IDB* predicates. The set of facts in which

[2] Note that a constraint is a shorthand for $false$:- $b_1,\ldots,b_k,$ `not` $b_{k+1},\ldots,$ `not` b_m., and it is also assumed that a rule bad :- $false$, `not` bad is in the program, where *false* and *bad* are special symbols appearing nowhere else in the original program.

https://doi.org/10.1017/S1471068411000652 Published online by Cambridge University Press

EDB predicates occur, denoted by $EDB(\mathscr{P})$, is called *Extensional Database (EDB)*, the set of all other rules is the *Intensional Database (IDB)*.

Semantics. Let $\mathscr{P}$ be an ASP program. The *Herbrand universe* of $\mathscr{P}$, denoted as $U_{\mathscr{P}}$, is the set of all constants appearing in $\mathscr{P}$. In the case when no constant appears in $\mathscr{P}$, an arbitrary constant ψ is added to $U_{\mathscr{P}}$. The *Herbrand base* of $\mathscr{P}$, denoted as $B_{\mathscr{P}}$, is the set of all ground atoms constructible from the predicate symbols appearing in $\mathscr{P}$ and the constants of $U_{\mathscr{P}}$. Given a rule r occurring in a program $\mathscr{P}$, 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 $U_{\mathscr{P}}$. We denote by $ground(\mathscr{P})$ the set of all the ground instances of the rules occurring in $\mathscr{P}$.

An *interpretation* for $\mathscr{P}$ is a set of ground atoms, that is, an interpretation is a subset I of $B_{\mathscr{P}}$. 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 `not` A is true w.r.t. I if A is false w.r.t. I; otherwise `not` A is false w.r.t. I. Let r be a rule in $ground(\mathscr{P})$. 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 otherwise the body of r is false w.r.t. I. 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 $\mathscr{P}$ is an interpretation M for $\mathscr{P}$ such that every rule $r \in ground(\mathscr{P})$ is true w.r.t. M. A model M for $\mathscr{P}$ is *minimal* if there is no model N for $\mathscr{P}$ such that N is a proper subset of M. The set of all minimal models for $\mathscr{P}$ is denoted by $\mathrm{MM}(\mathscr{P})$.

In the following, the semantics of ground programs is first given, then the semantics of general programs is given in terms of the answer sets of its instantiation.

Given a *ground* program $\mathscr{P}$ and an interpretation I, the *reduct* of $\mathscr{P}$ w.r.t. I is the subset $\mathscr{P}^I$ of $\mathscr{P}$ obtained by deleting from $\mathscr{P}$ the rules in which a body literal is false w.r.t. I.

Note that the above definition of reduct, proposed in Faber *et al.* (2004), simplifies the original definition of Gelfond–Lifschitz (GL) transform (Gelfond and Lifschitz 1991), but is fully equivalent to the GL transform for the definition of answer sets (Faber *et al.* 2004).

Let I be an interpretation for a ground program $\mathscr{P}$. I is an *answer set* (or stable model) for $\mathscr{P}$ if $I \in \mathrm{MM}(\mathscr{P}^I)$ (i.e., I is a minimal model for the program $\mathscr{P}^I$) (Faber *et al.* 2004). The set of all answer sets for $\mathscr{P}$ is denoted by $ANS(\mathscr{P})$.

3 Parallel instantiation

In this section, we describe an algorithm for the instantiation of ASP programs, which exploits parallelism in three different steps of the instantiation process. In particular, the algorithm employs techniques presented in (Calimeri *et al.* 2008) and integrates them with a novel strategy which has a larger application field, covering many situations in which the previous techniques do not apply. More in detail, the parallel instantiation algorithm allows for three levels of parallelism: *components*, *rules,* and *single-rule* level. The first level allows for instantiating in parallel subprograms of the program in input: it is especially useful when handling programs containing parts that are, somehow, independent. The second one allows for the parallel evaluation of rules within a given subprogram: it is useful when

https://doi.org/10.1017/S1471068411000652 Published online by Cambridge University Press

the number of rules in the subprograms is large. The third new one, allows for the parallel evaluation of a single rule: it is crucial for the parallelization of programs with few rules, where the first two levels are almost not applicable. In the following, we first provide an overview of our approach to the parallel instantiation process, giving an intuition of the three aforementioned levels and then we illustrate the instantiation algorithm.

3.1 Overview of the approach

Components Level (Calimeri *et al.* 2008). The first level of parallelism, called *Components Level*, consists in dividing the input program $\mathscr{P}$ into subprograms, according to the dependencies among the IDB predicates of $\mathscr{P}$, and by identifying which of them can be evaluated in parallel. More in detail, each program $\mathscr{P}$ is associated with a graph, called the *Dependency Graph* of $\mathscr{P}$, which, intuitively, describes how IDB predicates of $\mathscr{P}$ depend on each other. For a program $\mathscr{P}$, the *Dependency Graph* of $\mathscr{P}$ is a directed graph $G_{\mathscr{P}} = \langle N, E \rangle$, where N is a set of nodes and E is a set of arcs. N contains a node for each IDB predicate of $\mathscr{P}$, and E contains an arc $e = (p, q)$ if there is a rule r in $\mathscr{P}$ such that q occurs in the head of r and p occurs in a positive literal of the body of r.

The graph $G_{\mathscr{P}}$ induces a subdivision of $\mathscr{P}$ into subprograms (also called *modules*) allowing for a modular evaluation. We say that a rule $r \in \mathscr{P}$ *defines* a predicate p if p appears in the head of r. For each strongly connected component[3] (SCC) C of $G_{\mathscr{P}}$, the set of rules defining all the predicates in C is called *module* of C. Moreover, a partial ordering among the SCCs is induced by $G_{\mathscr{P}}$, defined as follows: for any pair of SCCs A, B of $G_{\mathscr{P}}$, 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 $G_{\mathscr{P}}$ from A to B.

According to such definitions, the instantiation of the input program $\mathscr{P}$ 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 (Calimeri *et al.* 2008). Indeed, this gives the possibility of computing ground instances of rules containing only atoms that can possibly be derived from $\mathscr{P}$ (thus avoiding the combinatorial explosion that 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 that 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 the data produced by the instantiation of B and vice versa. The dependency

[3] 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.

https://doi.org/10.1017/S1471068411000652 Published online by Cambridge University Press

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 can be further parallelized.

Rules Level (Calimeri *et al.* 2008). The second level of parallelism, called the *Rules Level*, allows for concurrently evaluating the rules within each module. A rule r occurring in the 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*. Rules are evaluated following a semi-naïve schema (Ullman 1989) 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 in which, at each iteration n, the instantiation of all the recursive rules is performed concurrently and by exploiting only the significant information derived during iteration $n-1$.

Single Rule Level. The first two levels of parallelism are effective when handling large 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. For instance, the following program $\mathscr{P}$ 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 $\mathscr{P}$. 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.*

In the following, we present a strategy for parallelizing the evaluation of a rule. The idea is to partition the extension of a single-rule literal (hereafter called *split* literal) into a number of subsets. Thus, the rule instantiation is divided into a number of smaller similar tasks each of which considers as extension of the split literal only one of those subsets. For instance, the evaluation of rule (c) in the previous example can be performed in parallel by partitioning the extension of one of its literals, let it be *edge*, into n subsets, thus obtaining n instantiation tasks for (c), working with different ground instances of *edge*. Note that, in general, several body literals are possible candidates to be split up (for instance, in the case of (c), *col* can be split up instead of *edge*) and the choice of the most suitable 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

https://doi.org/10.1017/S1471068411000652 Published online by Cambridge University Press

```
// First level of parallelism
Procedure Components_Instantiator(𝒫: Program; G_𝒫: DependencyGraph; var Π: GroundProgram)
begin
    var S: SetOfAtoms; var C: SetOfPredicates;
    S = EDB(𝒫); Π := ∅;
    while G_𝒫 ≠ ∅ do        // until there are components to be processed
        take a SCC C from G_𝒫 that does not depend on other SCCs of G_𝒫
        Spawn(Rules_Instantiator, 𝒫, C, S, Π, G_𝒫)
    end while
end;

// Second level of parallelism
Procedure Rules_Instantiator(𝒫: Program; C: Component; var S: SetOfAtoms;
                             var Π: GroundProgram, var G_𝒫: DependencyGraph)
begin
    var ΔS, 𝒩S: SetOfAtoms;
    ΔS := ∅; 𝒩S := ∅ ;
    for each r ∈ Exit(C, 𝒫) do              // evaluation of exit rules
        ℐ_r = Spawn (SingleRule_Instantiator, r, S, ΔS, 𝒩S, Π);
    for each r ∈ Exit(C, 𝒫) do              // synchronization barrier
        join_with_thread(ℐ_r);
    do
        ΔS := 𝒩S; 𝒩S := ∅ ;
        for each r ∈ Recursive(C, 𝒫) do     // evaluation of recursive rules
            ℐ_r = Spawn (SingleRule_Instantiator, r, S, ΔS, 𝒩S, Π);
        for each r ∈ Recursive(C, 𝒫) do     // synchronization barrier
            join_with_thread(ℐ_r);
        S := S ∪ ΔS;
    while 𝒩S ≠ ∅                            // until no new information can be derived
    Remove C from G_𝒫;                      // to process C only once
end Procedure;
```

Fig. 1. Components and rules parallelism.

cases, even worse than the time required to instantiate the original encoding). Note also that, the partitioning of the extension of the split literal has to be performed at run-time. Indeed, if the predicate to split is an IDB predicate, as in the case of *col*, the partitioning can be made only when the extension of the IDB predicate has already been computed; in our example, only after the evaluation of rule (r).[4]

3.2 The algorithms

The algorithms for the three levels of parallelism mentioned above are shown in Figures 1, and 2. They repeatedly apply a pattern similar to the classical producer–consumers problem. A *manager* thread (acting as a producer) identifies the tasks that can be performed in parallel and delegates their instantiation to a number of *worker* threads (acting as consumers). More in detail, the *Components_Instantiator* procedure (see Fig. 1), acting as a manager, implements the first level of parallelism, that is the parallel evaluation of program modules. It receives as input both a program $\mathscr{P}$ to be instantiated and its Dependency Graph $G_{\mathscr{P}}$; and it outputs a set of ground rules Π, such that $ANS(\mathscr{P}) = ANS(\Pi \cup EDB(\mathscr{P}))$. First of all, the algorithm

[4] An example of workload distribution is reported in Appendix C.

https://doi.org/10.1017/S1471068411000652 Published online by Cambridge University Press

```
// Third level of parallelism

// A split is virtually identified by four iterators to S and ΔS identifying ranges of instances.
struct VirtualSplit { AtomsIterator S_begin, S_end, ΔS_begin, ΔS_end; }

Procedure SingleRule_Instantiator(r: Rule; S: SetOfAtoms; ΔS: SetOfAtoms; 𝒩S: SetOfAtoms;
                                  var Π: GroundProgram)
    var s := numberOfSplits(B(r), S, ΔS); // according to load balancing and granularity control
    SelectLiteralToSplit(L,B(r),s);       // according to a heuristics
    var vector<VirtualSplit> Splits[s];   // create virtual splits
    SplitExtension(L, s, S, ΔS, Splits);  // distribute extension of L
    for each sp in Splits                 // spawn threads running InstantiateRule for each split
        ℐ_sp = Spawn (InstantiateRule, r, L, sp, S, ΔS, 𝒩S, Π);
    for each sp in Splits do              // synchronization barrier
        join_with_thread(ℐ_sp);
end Procedure;

Procedure SplitExtension(L: Literal; s:integer; S: SetOfAtoms; ΔS: SetOfAtoms;
                         var vector<VirtualSplit> Splits)
/*Given a literal L to split, a number s of splits to be produced, virtually partitions the extension
  of L stored in S and ΔS by determining a vector Splits of VirtualSplit structures identifying
  ranges of instances to be considered in each split.
*/

Procedure InstantiateRule(r: rule; L: Literal; sp: VirtualSplit; S: SetOfAtoms; ΔS: SetOfAtoms;
                          var 𝒩S: SetOfAtoms; var Π: GroundProgram)
/* Given S and ΔS builds all the ground instances of r, adds them to Π, and add to 𝒩S
   the new head atoms of the generated ground rules. For L only the ground atoms belonging to
   the ranges {S_Begin,S_End} and {ΔS _begin,ΔS _end} indicated by sp are used .
*/
```

Fig. 2. Single-rule parallelism.

creates a new set S of atoms that will contain the subset of the Herbrand base significant for the instantiation; more in detail, S will contain, for each predicate p in the program, the extension of p, that is, the set of all the ground atoms having the predicate name of p (significant for the instantiation).

Initially, $S = EDB(\mathscr{P})$, and $\Pi = \emptyset$. Then, the manager checks whether some SCC C can be instantiated; in particular, it checks whether there is some other component C' that has not been evaluated yet and such that C depends on C'. As soon as a component C is processable, a new thread is created, by a call to threading function *Spawn*, running procedure *Rules_Instantiator*.

Procedure *Rules_Instantiator* (see Fig. 1), implementing the second level of parallelism, takes as input, among the others, the component C to be instantiated and the set S; for each atom a belonging to C, and for each rule r defining a, it computes the ground instances of r containing only atoms that can possibly be derived from $\mathscr{P}$. At the same time, it updates the set S with the atoms occurring in the heads of the rules of Π. To this end, each rule r in the program module of C is processed by calling procedure *SingleRule_Instantiator*.

It is worth noting that exit rules are instantiated by a single call to *SingleRule_Instantiator*, whereas recursive ones are processed several times according to a semi-naïve evaluation technique (Ullman 1989), where at each iteration n only the

https://doi.org/10.1017/S1471068411000652 Published online by Cambridge University Press

significant information derived during iteration $n-1$ is used. This is implemented by partitioning significant atoms into three sets: ΔS, S, and $\mathcal{N}S$. $\mathcal{N}S$ 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 $\mathcal{N}S$ are empty; the exit rules contained in the program module of C are evaluated and, in particular, one new thread identified by $\mathcal{I}_r$ for each exit rule r, running procedure *SingleRule_Instantiator*, is spawned. Since the evaluation of recursive rules has to be performed only when the instantiation of all the exit rules is completed, a synchronization barrier is exploited. This barrier is encoded (à la POSIX) by several calls to threading function *join_with_thread* forcing *Rules_Instantiator* to wait until all *SingleRule_Instantiator* threads are done. Then, recursive rules are processed (do-while loop). At the beginning of each iteration, $\mathcal{N}S$ is assigned to ΔS, i.e., the new information derived during iteration n is considered as significant information for iteration $n+1$. Then, for each recursive rule, a new thread is spawned, running procedure *SingleRule_Instantiator*, which receives as input S and ΔS; when all threads terminate (second barrier), ΔS is added to S (since it has already been exploited). The evaluation stops whenever no new information has been derived (i.e., $\mathcal{N}S = \emptyset$). Eventually, C is removed from $G_{\mathscr{P}}$.

The third level of parallelism (see Fig. 2), concerning the parallel evaluation of each single rule, is then implemented by Procedure *SingleRule_Instantiator*, which given the sets S and ΔS of atoms that are known to be significant up to now, builds all the ground instances of r and adds them to Π. Initially, *SingleRule_Instantiator* selects, according to a heuristics for load balancing (see Section 5) the number s of parts in which the evaluation has to be divided; then, *SingleRule_Instantiator* heuristically selects a positive literal to split in the body of r, say L (see Section 4). A call to function *SplitExtension* (detailed in Appendix A) partitions the extension of L (stored in S and ΔS) into s equally sized parts, called splits. In order to avoid useless copies, each split is virtually identified by means of iterators over S and ΔS, representing ranges of instances. More in detail, for each split, a VirtualSplit is created containing two iterators over S (resp., ΔS), namely S_begin and S_end (resp., ΔS_begin and ΔS_end), indicating the instances of L from S (resp., ΔS) that belong to the split. Note that, in general, a split may contain ground atoms from both S and ΔS. Once the extension of the split literal has been partitioned, then a number of threads running procedure *InstantiateRule*, are spawned. *InstantiateRule*, given S and ΔS builds all the ground instances of r that can be obtained by considering as extension of the split literal L only the ground atoms indicated by the iterators in the virtual split at hand. *SingleRule_Instantiator* terminates (last barrier) once all splits are evaluated.

The correctness of the algorithm follows from the consideration that, whatever the split literal L, the union of the outputs of all the s concurrent *InstantiateRule* procedures is the same as the output produced by a single call to *InstantiateRule* working with the entire extension of L ($s = 1$). Note that, if the split predicate is recursive, its extension may change at each iteration. This is considered in our approach by performing different splits of recursive rules at each iteration. This

https://doi.org/10.1017/S1471068411000652 Published online by Cambridge University Press

ensures that at each iteration, the virtual splits are updated according to the actual extension of the literal to split.

In addition, this choice has a relevant side-effect: at each iteration, the workload is dynamically re-distributed among instantiators, thus inducing a form of dynamic load balancing in case of the evaluation of recursive rules.

4 Selection of the literal to split

Concerning the selection of the literal to split, the choice has to be carefully made, since it may strongly affect the cost of the instantiation of rules. It is well-known that this cost strongly 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. However, the choice of the split literal may influence the time spent on instantiating each split rule, whatever the join order. In the light of these observations, we have devised a new heuristics for selecting the split literal *given an optimal ordering* (which can be obtained as explained in Leone *et al.* (2001).

Intuitively, suppose we have a rule r containing n literals in the body in a given order, and suppose that any body literal allows for the target number of splits, say s, then: to obtain work for s threads, it is sufficient to split the first literal (whatever the join order); nonetheless, moving forward, say splitting the third literal, would cause a replicate evaluation of the join of the first two literals in each split thread possibly increasing parallel time. It is easy to see that the picture changes if all/some body literal does not allow for the target number of splits, in this case one should estimate the cost of splitting a literal different from the first and select the best possible choice.

In the following, we first introduce some metrics for estimating the work needed for instantiating a given rule, and then we describe the new heuristics. In detail, we use the following estimation for determining the size of the joins of the body literals: given two relations R and S, with one or more common variables, the size of $R \bowtie S$ can be estimated as follows:

$$T(R \bowtie S) = \frac{T(R) \cdot T(S)}{\prod_{X \in var(R) \cap var(S)} \max\{V(X,R), V(X,S)\}}, \tag{1}$$

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. Given an evaluation order of body literals, one can repeatedly apply this formula to pairs of body predicates for estimating the size of the join of a body. A more detailed discussion on this estimation can be found in Ullman (1989).

Let r be a rule with n body literals $L_1, L_2, \ldots, L_n$, where L_i precedes L_j for each $i < j$ in a given evaluation order, an estimation of the cost of instantiating the first k literals in $B(r)$ is:

$$\mathscr{C}(k) = \begin{cases} 0, & \text{if } k < 2, \\ T(L_1) \cdot T(L_2), & \text{if } k = 2, \\ \mathscr{C}(k-1) + T(L_1 \bowtie \cdots \bowtie L_{k-1}) \cdot T(k), & \text{if } k > 2. \end{cases} \tag{2}$$

https://doi.org/10.1017/S1471068411000652 Published online by Cambridge University Press

Table 1. *Number of splits and costs of the instantiation tasks, minimal values in boldface*

s	s^1	s^2	s^3	s^4	$\mathscr{C}^1$	$\mathscr{C}^2$	$\mathscr{C}^3$	$\mathscr{C}^4$
5	5	5	5	5	**11,400**	11,400	12,200	17,000
50	20	50	50	50	2,850	**1,140**	2,120	8,000
100	20	50	100	100	2,850	**1,140**	1,560	7,500
500	20	50	500	500	2,850	1,140	**1,112**	7,100

Now, let s be the number of splits to be performed; the following is an estimation of the work of the instantiation tasks obtained by the split of the ith literal L_i:

$$\mathscr{C}^i = \frac{\mathscr{C}(n) - \mathscr{C}(i-1)}{s^i} + \mathscr{C}(i-1), \quad 1 \leqslant i \leqslant n, \tag{3}$$

where s^i is equal to s (if the extension of L_i is sufficiently large) or the maximum number of splits allowed by L_i. Intuitively, if L_i is the split literal, the work of each instantiation task is composed of two parts: a part to be performed serially, common to all tasks, which consists in the instantiation of the first $i-1$ literals, whose cost is represented by $\mathscr{C}(i-1)$; and the instantiation of the remaining literals, which is divided among the s^i tasks, whose cost is represented by $\frac{\mathscr{C}(n)-\mathscr{C}(i-1)}{s^i}$. The estimation $\mathscr{C}^i$ can be used for determining the split literal, by choosing the one with minimum cost.

Note that, in the search for the best one, we can skip over each body literal L_k, with $k > j$, if L_j allows for s splits since $\mathscr{C}^j \leqslant \mathscr{C}^k$ holds. Indeed, if $n = 2$, $\mathscr{C}^1 = \mathscr{C}^2$; while for $n \geqslant 3$, $k = j + 1$ and $s^j = s^k = s$ (worst case), we have that

$$\mathscr{C}^k = \frac{\mathscr{C}(n) - \mathscr{C}(k-1)}{s^k} + \mathscr{C}(k-1) = \frac{\mathscr{C}(n) - \mathscr{C}(j)}{s} + \mathscr{C}(j).$$

By applying (2):

$$\mathscr{C}^k = \frac{\mathscr{C}(n) - \mathscr{C}(j-1)}{s} - \frac{Q}{s} + \mathscr{C}(j-1) + Q = C^j + \frac{s-1}{s} \cdot Q,$$

where $Q = T(L_1 \bowtie \cdots \bowtie L_{j-1}) \cdot T(j)$. Thus, by induction, if L_j allows for s splits, $\mathscr{C}^j \leqslant \mathscr{C}^k$, for $k > j$. Intuitively, this can be explained by considering that splitting a literal L_k after one allowing for s splits L_j has the effect of evaluating serially the join of literals between L_k and L_j thus leading to a greater evaluation time. Clearly, even a literal L whose extension cannot be split in s parts can be chosen, provided that L allows for a minor (estimated) work for each instantiation task. Moreover, if $s^1 = s$ (L_1 allows for s splits), it holds that $\mathscr{C}^1 \leqslant \mathscr{C}^i$, for each $1 < i \leqslant n$; in this case, L_1 can be chosen without computing any cost.

As an example, suppose that we have to instantiate the constraint $:\!-a(X,Y)$, $b(Y,Z)$, $c(Z,X)$, $d(V,Z)$. Suppose also that the extensions of the body literals are $T(a) = 20$, $T(b) = 50$, $T(c) = 1{,}000$, $T(d) = 1{,}000$, and that the estimations of the costs of instantiating the first i literals with $1 \leqslant i \leqslant 4$ are the following: $\mathscr{C}(1) = 0$, $\mathscr{C}(2) = 1{,}000$, $\mathscr{C}(3) = 7{,}000$, $\mathscr{C}(4) = 57{,}000$. Table 1 shows the estimations $\mathscr{C}^i$ of the works of the instantiation tasks obtained by the split of the ith literal with $1 \leqslant i \leqslant 4$, by varying the target number s of splits. In particular, the first column shows the

https://doi.org/10.1017/S1471068411000652 Published online by Cambridge University Press

target number of splits, the following four columns show the maximum number of splits s^i allowed for each literal, and the remaining four columns show the costs $\mathscr{C}^i$ computed according to the s^i values; in boldface we outline, for each target number of splits, the minimal values of $\mathscr{C}^i$. It can be noted that, in our example, increasing the value of s corresponds to different choices of the literal to split. Moreover, in each row, the choice is always restricted to the first i literals, where the ith literal is the first one allowing for the target number s of splits; indeed, $\mathscr{C}^i \leqslant \mathscr{C}^k$, for each $k > i$. Furthermore, even a literal that does not allow for s splits can be chosen; this is the case for $s = 100$, where the chosen literal is $b(Y, Z)$, which allow for 50 splits. Notice that the choice of the literal to split may be influenced by the body ordering in some cases, which in turn considerably affects the serial evaluation time (which is the amount to be divided by parallel evaluation). For example, all body orderings having $d(V, Z)$ as first literal have d as chosen literal, since its extension is sufficiently large to allow the four target numbers of splits considered. However, if such orderings determine a higher evaluation cost w.r.t. the body order exploited in the serial evaluation, then the effect of parallel evaluation could be overshadowed. Thus, we apply the selection of the literal to split after body reordering with a strategy that minimizes the heuristic cost of instantiating the body.

Summarizing, our heuristics consists in determining an ordering of the body literals exploiting the already assessed technique described in Leone *et al.* (2001) and splitting the first literal in the body if it allows for the target number of splits (without computing any cost). Otherwise, the estimations of the costs are determined and the literal allowing for the minimum one is chosen.

5 Load balancing and granularity control

An advanced parallelization technique has to deal with two important issues that strongly affect the performance of a real implementation: 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 corner cases, adopting a sequential evaluation might be preferable).

As an example, consider the case in which we are running the instantiation of a rule r on a two-processor machine and, by applying the technique for Single-Rule parallelism described above, the instantiation of r is divided into two smaller tasks, by partitioning the extension of the split predicate of r into two subsets with, approximatively, the same size. Then, each of the two tasks 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 into equal parts does not ensure that the workload is equally spread among threads.

https://doi.org/10.1017/S1471068411000652 Published online by Cambridge University Press

However, if we consider a larger number of splits, a further subdivision of the workload would be obtained, and, the inactivity time would be more likely limited.

Clearly, it is crucial to guarantee that the parallel evaluation of a number of tasks is not more time-consuming than their serial evaluation (granularity control); and that an unbalanced workload distribution does not introduce significant delays and limits the overall performance (load balancing).

Granularity Control. Our method for granularity control is based on the use of a heuristic value $\mathscr{W}(r)$, which acts as a litmus paper indicating the amount of work required for evaluating each rule of the program, and so, its "hardness," just before its instantiation. $\mathscr{W}(r)$ denotes the value $\mathscr{C}(n)$ (see Section 4), for each rule r having n body literals.

At the rules level, rather than assigning each rule to a different thread, a set of rules R is determined and assigned to a thread. R is such that the total work for instantiating its rules is enough for enjoying the benefits of scheduling a new thread. In practice, R is constructed by iterating on the rules of the same component, and stopping if either $\sum_{r \in R} \mathscr{W}(r) > w_{seq}$ or when no further rules can be added to R, where w_{seq} is an empirically-determined threshold value. At the single-rule level, a rule r is scheduled for parallel instantiation (i.e., its evaluation can be divided into smaller tasks that can be performed in parallel) if $\mathscr{W}(r) > w_{seq}$; otherwise, for r, the third level of parallelism it is not applied at all.

Note that, for simplifying the presentation of the algorithms in Section 3, we have not considered the management of the granularity control in the second level of parallelism, which would have added noisy technical details and made the description more involved. However, they can be suitably adapted by modifying procedure *Rules_Instantiator* (see Fig. 1) in order to build a set of "easy" rules, and by adding a *SetOfRules_Instantiator* procedure, which instantiates each rule in the set. Note also that, granularity control in the third level of parallelism is obtained by setting the number of splits of a given rule to 1.

Load Balancing. In our approach, load balancing exploits different factors. On the one hand, in the case of the evaluation of recursive rules, a dynamic load redistribution of the extension of the split literal at each iteration is performed. On the other hand, the extension of the split literal is divided by a number which is greater than the number of processors (actually, a multiple of the number of processors is enough) for exploiting the preemptive multitasking scheduler of the operating system. Moreover, in case of "very hard" rules, a finer distribution is performed in the last splits. In particular, when a rule is assessed to be "hard" by comparing the estimated work (the value $\mathscr{W}(r)$ described above) with another empirically determined threshold ($\mathscr{W}(r) > w_{hard}$), a finer work distribution (exploiting a unary split size) is performed for the last $s - n_p$ splits, where s is the number of splits 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.

https://doi.org/10.1017/S1471068411000652 Published online by Cambridge University Press

6 Experiments

The parallelization techniques described in the previous sections were implemented in the instantiation module of DLV (Leone *et al.* 2006). In order to assess the performance of the resulting parallel instantiator, we carried out an extensive experimental activity, reported in this section. In particular, (*i*) we compared the previous techniques (components and rules parallelism) with the new technique (single-rule parallelism); and (*ii*) we conducted a scalability analysis on the instantiator considering the effects of increasing both the number of available processors and the size of the instances. Before discussing the results, a description of the implemented system and some benchmark data are given.

6.1 Implementation in DLV

We implemented our parallel techniques by extending the instantiator module of DLV. The system is implemented in the C++ language by exploiting the Linux POSIX Thread API, shipped with the GCC 4.3.3 compiler. The actual implementation of the algorithms reported in Section 3 adopts a producer–consumers pattern in which the total number of threads spawned in each of the three levels of parallelism is limited to a fixed number which is user-defined. This is obtained by adapting procedures described in Section 3.1, in such a way that spawn commands are replaced by push operations in three different shared buffers (one for each level of parallelism); moreover, thread joins ensuring the completion of a given task (e.g., evaluation of all splits of the same rule) are replaced by proper condition statements (e.g., counting semaphores). In this way, worker threads are recycled, and continuously pop working tasks from the buffers, up to the end of the instantiation process. The main motivation for this technical variant is limiting thread creation overhead to a single initialization step. In addition, the implementation allows for separately enabling/disabling the three levels of parallelism by command line options.

6.2 Benchmark problems and data

In our experiments, several well-known combinatorial problems as well as real-world problems are considered. These benchmarks have already been used for assessing ASP instantiator performance (Leone *et al.* 2006; Gebser *et al.* 2007b; Denecker *et al.* 2009). Many of them are particularly difficult to parallelize due to the compactness of their encodings; note that concise encodings 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 (where the instantiation time is non-negligible) of increasing difficulty for each problem, except for the Hamiltonian Path and 3-Colorability problem, for which generators are available, and we could generate several instances of increasing size.

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.`

https://doi.org/10.1017/S1471068411000652 Published online by Cambridge University Press

mat.unical.it/ricca/downloads/parallelground10.zip. Rather, to help the understanding of the results, both the description of problems and some information on the number of rules of each program are reported below.

n-Queens. The 8-queens puzzle is the problem of putting eight chess queens on an 8×8 chessboard so that none of them is able to capture any other using the standard chess queen's moves. The n-queens puzzle is the more general problem of placing n queens on an $n\times n$ chessboard ($n \geqslant 4$). The encoding consists of one rule and four constraints. Instances were considered having $n \in \{37, 39, 41, 43, 45\}$.

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 $ramsey(k, m)$.

Clique. A clique in an undirected graph $G = (V, E)$ is a subset of its vertices such that every two vertices in the subset are connected by an edge. We considered the problem finding a clique in a given input graph. Five graphs of increasing size were considered.

Timetabling. The problem of determining a timetable for some university lectures which have to be given in a week to some groups of students. The timetable must respect a number of given constraints concerning availability of rooms, teachers, and other issues related to the overall organization of the lectures. Many instances were provided by the University of Calabria; they refer to different numbers of student groups $g \in \{15, 17, 19, 21, 23\}$.

Sudoku. Given an $N\times N$ grid board, where N is a square number $N = M^2$, fill it with integers from 1 to N so that each row, each column, and each of the N $M\times M$ boxes contains each of the integers from 1 to N exactly once. Suppose the rows are numbered 1 to N from left to right, and the columns are numbered 1 to N from top to bottom. The boxes are formed by dividing the rows from top to bottom every M rows and dividing the columns from left to right every M columns. Encoding and instances were used for testing the competitors solvers in the ASP Competition 2009 (Denecker *et al.* 2009). For assessing our system, we considered the instances {*sudoku.in*5, *sudoku.in*6, *sudoku.in*7, *sudoku.in*9, *sudoku.in*10}, where $N = 25$.

Golomb Ruler. A Golomb ruler is an assignment of marks to integer positions along a ruler so that no pair of two marks is the same distance from each other. The number of marks is the order of the ruler. The first mark is required to be at position 0 and the position of the highest mark is the length of the ruler. The problem is finding the shortest ruler of a given order. Encoding and instances have been used for testing the competitors solvers in the ASP Competition 2009 (Denecker *et al.* 2009). Instances are described by a couple (m, p), where m is the number of marks and p is the number of positions: we considered the values (10, 125), (13, 150), (14, 175),(15, 200), and (15, 225).

https://doi.org/10.1017/S1471068411000652 Published online by Cambridge University Press

Reachability. Given a directed graph $G = (V, E)$, 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 edges in E. The encoding of this problem consists of one exit rule and a recursive one. Five trees were generated with a pair (number of levels, number of siblings): (9,3), (7,5), (14,2), (10,3), and (15,2), respectively.

Food. The problem here is to generate plans for repairing faulty workflows. That is, starting from a faulty workflow instance, the goal is to provide a completion of the workflow such that the output of the workflow is correct. Workflows may comprise many activities. Repair actions are compensation, (re)do, and replacement of activities. A single instance was provided related to a workflow containing 63 predicates, 56 components, and 116 rules.

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. A number of simplex graphs were generated with the Stanford GraphBase library (Knuth 1994), by using the function $simplex(n, n, -2, 0, 0, 0, 0)$, where $80 \leqslant n \leqslant 250$.

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. Simons 2000), with n nodes with $1,000 \leqslant n \leqslant 12,000$.

The machine used for the experiments is a two-processor Intel Xeon "Woodcrest" (quad core) 3 GHz machine with 4 MB of L2 Cache and 4 GB of RAM, running Debian GNU Linux 4.0. 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 (parsing) is obviously the same both for parallel and non-parallel version. In order to obtain more trustworthy results, each single experiment was repeated five times and the average of performance measures is reported.

6.3 Effect of single-rule parallelism

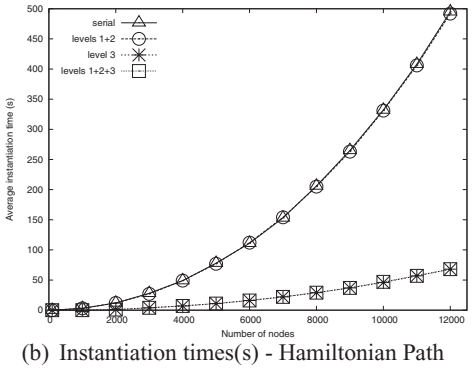
In this section, we report the results of an experimental analysis aimed at comparing the effects of the single-rule parallelism with the first two levels. More in detail, we considered four versions of the instantiator: (*i*) `serial` where parallel techniques are not applied, (*ii*) $\texttt{levels}_{1+2}$ where components and rules parallelism are applied, (*iii*) $\texttt{level}_{3}$ where only the single-rule level is applied, and (*iv*) $\texttt{levels}_{1+2+3}$ in which all the three levels are applied. Results are shown in Figures 3(a)–(c), where the two graphs report the average instantiation times for the Hamiltonian Path problem and 3-Colorability; while the table reports the average instantiation times in seconds for the remaining benchmarks.[5]

[5] Results in the form of tables for Hamiltonian Path and 3-Colorability are reported in Appendix B.

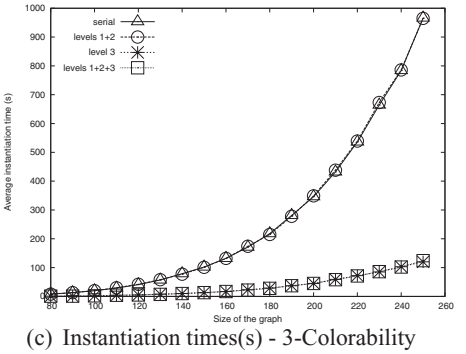
https://doi.org/10.1017/S1471068411000652 Published online by Cambridge University Press

Problem	serial	levels_{1+2}	level_3	levels_{1+2+3}
$queen_1$	4.64 (0.00)	2.19 (0.06)	0.71 (0.01)	0.69 (0.01)
$queen_2$	5.65 (0.00)	3.29 (0.51)	0.89 (0.01)	0.86 (0.02)
$queen_3$	6.83 (0.00)	3.85 (0.50)	1.08 (0.00)	1.03 (0.02)
$queen_4$	8.19 (0.00)	4.50 (0.48)	1.25 (0.01)	1.22 (0.00)
$queen_5$	9.96 (0.00)	4.72 (0.11)	1.45 (0.02)	1.43 (0.01)
$ramsey_1$	258.52 (0.00)	131.61 (0.23)	40.53 (0.21)	36.23 (0.34)
$ramsey_2$	328.68 (0.00)	168.03 (1.38)	51.35 (0.25)	46.09 (0.33)
$ramsey_3$	414.88 (0.00)	211.60 (0.55)	68.49 (0.10)	58.06 (0.20)
$ramsey_4$	518.28 (0.00)	265.58 (2.82)	83.32 (0.24)	75.19 (2.41)
$ramsey_5$	643.65 (0.00)	327.25 (0.79)	103.14 (0.44)	92.28 (0.65)
$clique_1$	16.06 (0.00)	15.88 (0.20)	3.34 (0.04)	2.35 (0.01)
$clique_2$	29.98 (0.00)	29.97 (0.01)	4.41 (0.12)	4.34 (0.07)
$clique_3$	49.11 (0.00)	49.18 (0.05)	7.11 (0.03)	7.09 (0.02)
$clique_4$	78.05 (0.00)	78.70 (0.35)	11.35 (0.14)	11.29 (0.11)
$clique_5$	119.48 (0.00)	118.66 (0.07)	17.08 (0.14)	17.09 (0.16)
$timetab_1$	15.48 (0.00)	7.28 (0.04)	2.35 (0.04)	2.29 (0.01)
$timetab_2$	17.49 (0.00)	8.30 (0.07)	2.61 (0.01)	2.61 (0.02)
$timetab_3$	21.65 (0.00)	10.22 (0.05)	3.22 (0.04)	3.20 (0.01)
$timetab_4$	17.75 (0.00)	8.24 (0.04)	2.61 (0.01)	2.64 (0.05)
$timetab_5$	23.69 (0.00)	11.09 (0.01)	3.52 (0.01)	3.50 (0.03)
$sudoku_1$	5.42 (0.00)	4.15 (0.04)	0.98 (0.01)	0.88 (0.01)
$sudoku_2$	9.87 (0.00)	7.75 (0.01)	1.59 (0.02)	1.51 (0.02)
$sudoku_3$	10.28 (0.00)	8.01 (0.05)	1.62 (0.00)	1.57 (0.01)
$sudoku_4$	10.56 (0.00)	8.38 (0.01)	1.75 (0.02)	1.63 (0.03)
$sudoku_5$	11.08 (0.00)	8.25 (0.04)	1.67 (0.02)	1.63 (0.05)
gol_ruler_1	6.58 (0.00)	6.32 (0.07)	0.96 (0.01)	0.94 (0.02)
gol_ruler_2	13.74 (0.00)	12.57 (0.04)	1.87 (0.04)	1.84 (0.09)
gol_ruler_3	24.13 (0.00)	22.67 (0.05)	3.29 (0.06)	3.25 (0.13)
gol_ruler_4	40.64 (0.00)	37.50 (0.10)	5.44 (0.21)	5.51 (0.10)
gol_ruler_5	62.23 (0.00)	59.04 (0.04)	8.32 (0.12)	8.36 (0.17)
$reach_1$	52.21 (0.00)	52.07 (0.07)	8.25 (0.06)	8.28 (0.01)
$reach_2$	147.34 (0.00)	148.34 (0.01)	22.60 (0.16)	22.67 (0.18)
$reach_3$	258.01 (0.00)	240.17 (0.13)	39.59 (0.29)	39.57 (0.44)
$reach_4$	522.09 (0.00)	517.97 (0.59)	77.21 (0.20)	77.52 (0.31)
$reach_5$	1072.00 (0.00)	1069.86 (1.04)	160.66 (0.21)	160.31 (0.25)
$Food$	684.95 (1.19)	0.18 (0.15)	104.6 (1.01)	0.08 (0.01)

(a) Average instantiation times in seconds (standard deviation)



(b) Instantiation times(s) - Hamiltonian Path



(c) Instantiation times(s) - 3-Colorability

Fig. 3. Impact of parallelization techniques.

More in detail, in Figure 3(a), the first column reports the problem considered, whereas the next columns report the results for the four instantiators. Looking at the third column in the table, benchmarks can be classified in three different groups according to their behaviors: the benchmarks in which the first two levels of parallelism apply, those where these first two levels apply marginally, and those

https://doi.org/10.1017/S1471068411000652 Published online by Cambridge University Press

where they do not apply at all. In the first group, we have the n-Queens problem, Ramsey Numbers, and Timetabling, where $\texttt{levels}_{1+2}$ are about twice faster than `serial`; however, considering that the machine on which we ran the benchmarks has eight cores available, $\texttt{levels}_{1+2}$ are not able to exploit all the computational power at hand. The reason is that the encodings of these benchmarks either have a small number of rules (n-Queens, Ramsey Numbers), or they show an intrinsic dependency among components/rules (Timetabling), that limits the efficacy of the first two levels of parallelism. These considerations explain also the behavior of the other two groups of benchmarks. More in detail, for the second group (which contains only Sudoku), a small improvement is obtained due to few rules which are evaluated in parallel, while the benchmarks belonging to the third group, whose encodings have very few interdependent rules (e.g., Reachability), proved hard to parallelize. Looking at the graphs, Hamiltonian Path and 3-Colorability clearly belong to the third group, indeed the lines of `serial` and $\texttt{levels}_{1+2}$ overlap.

A special case is the Food problem, showing an impressive performance, which proved 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 $\texttt{levels}_{1+2}$ 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 latter evaluates the inconsistent constraint later on.

Concerning the behavior of $\texttt{level}_3$, we notice that it always performs very well (always more than 7.5× faster than `serial`), and in all cases but Food, it outperforms $\texttt{levels}_{1+2}$. Basically, the third level of parallelism applies to every single rule, and thus it is effective on all problems, even those with very small encodings. In the case of Food, even if $\texttt{level}_3$ is about 7x faster than `serial`, it evaluates rules in the same order than as `serial`, thus recognizing the inconsistent constraint later than $\texttt{levels}_{1+2}$.

The good news is that the three levels of parallelism always combine (even in the case of Food). This can be easily seen by looking at the last column of table and at the two graphs. Note that most of the advantages are due to the third level of parallelism. Indeed, in the graphs, the lines for $\texttt{level}_3$ and $\texttt{levels}_{1+2+3}$ overlap, and $\texttt{levels}_{1+2+3}$ show only marginal gains w.r.t. $\texttt{level}_3$, in the benchmarks where $\texttt{levels}_{1+2}$ applies.

6.4 Scalability of the approach

We conducted a scalability analysis on the instantiator $\texttt{levels}_{1+2+3}$ which exploits all the three parallelism levels. Moreover, we considered the effects of increasing both the size of the instances and the number of available processors (from one up to eight CPUs).[6] The results of the analysis are summarized in Table 2, and Figure 4,

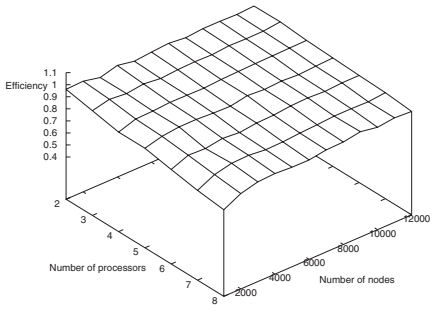
[6] Available processors can be disabled (respectively, enabled) by running the bash Linux commands: *echo 0 >> /sys/devices/system/cpu/cpu-n/online* (resp. *echo 1 >> /sys/devices/system/cpu/cpu-n/online*).

https://doi.org/10.1017/S1471068411000652 Published online by Cambridge University Press

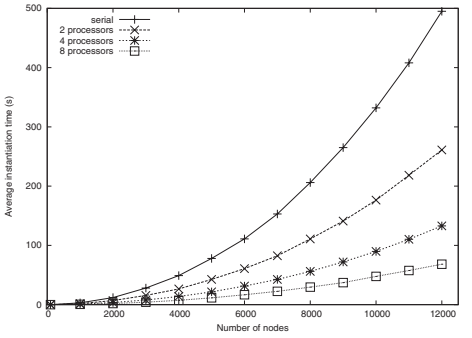
https://doi.org/10.1017/S1471068411000652 Published online by Cambridge University Press

Table 2. *Benchmark results: average instantiation times in seconds (standard deviation), efficiency*

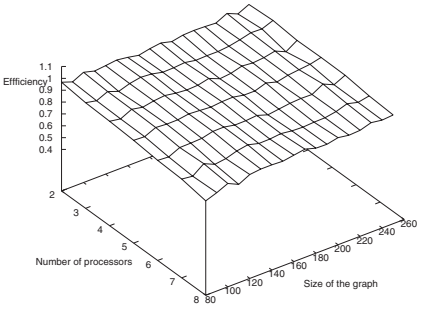
	Average instantiation time (standard deviation)								Efficiency						
Problem	serial	2 proc	3 proc	4 proc	5 proc	6 proc	7 proc	8 proc	2 proc	3 proc	4 proc	5 proc	6 proc	7 proc	8 proc
$queen_1$	4.64 (0.00)	2.53 (0.01)	1.71 (0.01)	1.31 (0.01)	1.07 (0.00)	0.91 (0.01)	0.78 (0.01)	0.69 (0.01)	0.98	0.97	0.95	0.93	0.91	0.91	0.90
$queen_2$	5.65 (0.00)	3.11 (0.00)	2.11 (0.01)	1.60 (0.00)	1.31 (0.01)	1.11 (0.01)	0.97 (0.00)	0.86 (0.02)	0.99	0.97	0.96	0.94	0.92	0.91	0.90
$queen_3$	6.83 (0.00)	3.79 (0.01)	2.57 (0.01)	1.97 (0.01)	1.60 (0.01)	1.35 (0.02)	1.17 (0.01)	1.03 (0.02)	0.99	0.97	0.95	0.94	0.92	0.91	0.91
$queen_4$	8.19 (0.00)	4.54 (0.00)	3.06 (0.01)	2.35 (0.01)	1.90 (0.01)	1.62 (0.02)	1.41 (0.01)	1.22 (0.00)	0.99	0.98	0.96	0.95	0.92	0.91	0.92
$queen_5$	9.96 (0.00)	5.57 (0.19)	3.68 (0.01)	2.81 (0.02)	2.26 (0.02)	1.92 (0.00)	1.69 (0.01)	1.43 (0.01)	0.97	0.97	0.96	0.95	0.93	0.94	0.94
$ramsey_1$	258.52 (0.00)	131.72 (0.08)	89.04 (0.41)	67.10 (0.46)	55.14 (0.93)	46.62 (0.19)	39.98 (0.12)	36.23 (0.34)	0.98	0.97	0.96	0.94	0.92	0.92	0.89
$ramsey_2$	328.68 (0.00)	167.47 (0.16)	112.97 (0.94)	85.90 (0.15)	70.64 (1.74)	58.70 (0.82)	51.21 (0.18)	46.09 (0.33)	0.98	0.97	0.96	0.93	0.93	0.92	0.89
$ramsey_3$	414.88 (0.00)	210.98 (0.38)	142.85 (0.68)	108.00 (0.38)	88.13 (0.51)	74.83 (0.22)	65.25 (0.59)	58.06 (0.20)	0.98	0.97	0.96	0.94	0.92	0.91	0.89
$ramsey_4$	518.28 (0.00)	264.69 (1.82)	178.67 (2.39)	137.42 (1.89)	111.09 (2.15)	95.27 (2.02)	81.45 (0.45)	75.19 (2.41)	0.98	0.97	0.94	0.93	0.91	0.91	0.86
$ramsey_5$	643.65 (0.00)	327.06 (0.36)	222.81 (0.20)	169.37 (0.86)	135.94 (0.17)	115.78 (0.92)	101.21 (1.33)	92.28 (0.65)	0.98	0.96	0.95	0.95	0.93	0.91	0.87
$clique_1$	16.06 (0.0)	8.51 (0.13)	5.84 (0.08)	4.45 (0.17)	3.64 (0.04)	3.08 (0.1)	2.67 (0.03)	2.35 (0.01)	0.94	0.92	0.90	0.88	0.87	0.86	0.85
$clique_2$	29.98 (0.0)	15.92 (0.23)	10.69 (0.18)	8.27 (0.09)	6.77 (0.11)	5.71 (0.10)	4.94 (0.40)	4.34 (0.07)	0.94	0.93	0.91	0.89	0.88	0.87	0.86
$clique_3$	49.11 (0.00)	25.81 (0.41)	17.31 (0.06)	13.39 (0.20)	10.92 (0.20)	9.23 (0.02)	7.98 (0.03)	7.09 (0.02)	0.95	0.95	0.92	0.90	0.89	0.88	0.87
$clique_4$	78.05 (0.00)	41.68 (0.07)	27.91 (0.28)	21.10 (0.02)	17.33 (0.20)	14.60 (0.04)	12.76 (0.06)	11.29 (0.11)	0.94	0.93	0.92	0.90	0.89	0.87	0.86
$clique_5$	119.48 (0.00)	62.87 (0.13)	42.62 (0.15)	32.46 (0.04)	26.14 (0.21)	22.24 (0.00)	19.14 (0.00)	17.09 (0.16)	0.95	0.93	0.92	0.91	0.90	0.89	0.87
$timetab_1$	15.48 (0.00)	7.98 (0.10)	5.41 (0.02)	4.16 (0.00)	3.37 (0.00)	2.93 (0.05)	2.59 (0.03)	2.29 (0.01)	0.97	0.95	0.93	0.92	0.88	0.85	0.84
$timetab_2$	17.49 (0.00)	9.26 (0.34)	6.30 (0.23)	4.68 (0.01)	3.89 (0.04)	3.41 (0.14)	2.92 (0.04)	2.61 (0.02)	0.94	0.93	0.93	0.90	0.85	0.86	0.84
$timetab_3$	21.65 (0.00)	11.12 (0.03)	7.54 (0.02)	5.98 (0.23)	4.84 (0.09)	4.08 (0.05)	3.64 (0.02)	3.20 (0.01)	0.97	0.96	0.91	0.89	0.88	0.85	0.85
$timetab_4$	17.75 (0.00)	9.32 (0.36)	6.13 (0.02)	4.75 (0.06)	3.86 (0.02)	3.33 (0.01)	3.01 (0.15)	2.64 (0.05)	0.95	0.97	0.93	0.92	0.89	0.84	0.84
$timetab_5$	23.69 (0.00)	12.16 (0.01)	8.28 (0.01)	6.35 (0.02)	5.34 (0.20)	4.47 (0.03)	3.94 (0.03)	3.50 (0.03)	0.97	0.95	0.93	0.89	0.88	0.86	0.85
$sudoku_1$	5.42 (0.00)	2.84 (0.01)	2.14 (0.21)	1.54 (0.00)	1.29 (0.02)	1.10 (0.00)	0.98 (0.02)	0.88 (0.01)	0.95	0.84	0.88	0.84	0.82	0.79	0.77
$sudoku_2$	9.87 (0.00)	5.09 (0.02)	3.53 (0.02)	2.72 (0.04)	2.25 (0.01)	1.90 (0.02)	1.68 (0.03)	1.51 (0.02)	0.94	0.94	0.91	0.89	0.87	0.83	0.82
$sudoku_3$	10.28 (0.00)	5.45 (0.17)	3.63 (0.01)	2.81 (0.02)	2.31 (0.01)	1.96 (0.01)	1.78 (0.00)	1.57 (0.01)	0.97	0.93	0.91	0.88	0.87	0.84	0.82
$sudoku_4$	10.56 (0.00)	5.50 (0.03)	3.80 (0.03)	2.92 (0.03)	2.41 (0.02)	2.03 (0.02)	1.81 (0.04)	1.63 (0.03)	0.96	0.93	0.90	0.88	0.87	0.83	0.81
$sudoku_5$	11.08 (0.00)	5.52 (0.12)	3.73 (0.01)	2.93 (0.11)	2.35 (0.01)	2.05 (0.04)	1.82 (0.03)	1.63 (0.05)	1.00	0.99	0.95	0.94	0.90	0.87	0.85
gol_ruler_1	6.58 (0.00)	3.34 (0.01)	2.26 (0.00)	1.73 (0.02)	1.42 (0.02)	1.24 (0.02)	1.06 (0.03)	0.94 (0.02)	0.99	0.97	0.95	0.93	0.88	0.89	0.88
gol_ruler_2	13.74 (0.00)	6.63 (0.02)	4.60 (0.18)	3.41 (0.04)	2.86 (0.10)	2.43 (0.04)	2.11 (0.02)	1.84 (0.09)	1.04	1.00	1.01	0.96	0.94	0.93	0.93
gol_ruler_3	24.13 (0.00)	12.11 (0.02)	8.15 (0.06)	6.34 (0.06)	5.06 (0.10)	4.34 (0.17)	3.79 (0.05)	3.25 (0.13)	1.00	0.99	0.95	0.95	0.93	0.91	0.93
gol_ruler_4	40.64 (0.00)	20.27 (0.05)	13.51 (0.11)	10.35 (0.10)	8.64 (0.19)	7.13 (0.25)	6.35 (0.31)	5.51 (0.10)	1.00	1.00	0.98	0.94	0.95	0.91	0.92
gol_ruler_4	62.23 (0.00)	31.54 (0.29)	21.30 (0.16)	16.03 (0.09)	12.95 (0.20)	11.03 (0.27)	9.67 (0.15)	8.36 (0.17)	0.99	0.97	0.97	0.96	0.94	0.92	0.93
$reach_1$	52.21 (0.00)	29.52 (0.36)	20.41 (0.25)	15.38 (0.02)	12.73 (0.16)	10.81 (0.03)	9.63 (0.05)	8.28 (0.01)	0.86	0.85	0.85	0.82	0.80	0.77	0.79
$reach_2$	147.34 (0.00)	84.93 (0.35)	57.14 (0.07)	43.56 (0.07)	35.16 (0.19)	29.90 (0.02)	26.02 (0.10)	22.67 (0.18)	0.87	0.86	0.85	0.84	0.82	0.81	0.81
$reach_3$	258.01 (0.00)	144.36 (0.60)	97.06 (0.39)	74.88 (0.25)	61.05 (0.21)	52.38 (0.09)	45.64 (0.21)	39.57 (0.44)	0.87	0.87	0.86	0.85	0.82	0.81	0.82
$reach_4$	522.09 (0.00)	301.44 (0.48)	201.90 (0.47)	153.00 (0.28)	123.83 (0.16)	104.74 (0.54)	90.44 (0.10)	77.52 (0.31))	0.87	0.86	0.85	0.84	0.83	0.82	0.84
$reach_5$	1072.00 (0.00)	618.68 (0.71)	412.29 (0.05)	311.96 (0.98)	253.22 (0.98)	213.31 (0.85)	185.08 (0.26)	160.31 (0.25)	0.87	0.87	0.86	0.85	0.84	0.83	0.84
Food	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)	0.08 (0.01)	1556	2853	2446	2283	1902	1630	1223



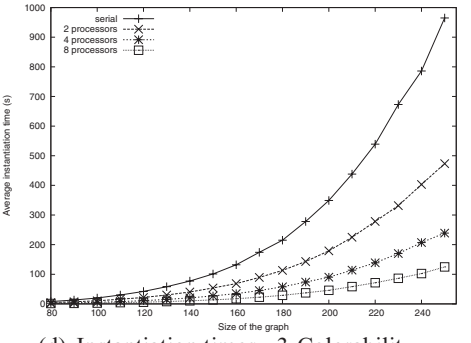
(a) Efficiency - Hamiltonian Path



(b) Instantiation times - Hamiltonian Path



(c) Efficiency - 3-Colorability



(d) Instantiation times - 3-Colorability

Fig. 4. Hamiltonian Path and 3-Colorability: average instantiation times (s) and efficiency. (a) Efficiency – Hamiltonian Path; (b) Instantiation times – Hamiltonian Path; (c) Efficiency – 3-Colorability; (d) Instantiation times – 3-Colorability.

where both the average instantiation times and the efficiencies are reported. As before, the graphs show results for Hamiltonian Path and 3-Colorability, while the results of the remaining problems are reported in the table. The overall picture is very positive: the performance of the instantiator is very good in all cases and average efficiencies vary in between 0.85 and 0.95 when all the available CPUs are enabled. 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 (<2 s) to larger ones.

The granularity control mechanism resulted to be effective in the *n*-Queens problem, where all the considered instances required less than 10 s of serial execution time. Indeed, the "very easy" disjunctive rule was always sequentially evaluated in all the cases. Since the remaining constraints are strictly determined by the result of the evaluation of the disjunctive rule, the unavoidable presence of a sequential part limited the final efficiency to a remarkable 0.9 in the case of eight processors. A similar scenario can be observed in the case of Ramsey Numbers, where the positive impact of the load balancing and granularity control heuristics becomes very evident. In fact, since the encoding is composed of few "very easy" disjunctive rules and two "very hard" constraints, the heuristics selects a sequential evaluation

https://doi.org/10.1017/S1471068411000652 Published online by Cambridge University Press

for the rules, and dynamically applies the 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 or almost equal to 0.9 in all tested configurations. Analogously, for Clique, which has a short encoding consisting of only three easy rules, for which granularity control schedules a serial execution, and one "hard" constraint which can be split and thus evaluated in parallel.

A good performance is also obtained in the case of Reachability. This problem is made up of only two rules; the first one is caught by granularity control which schedules its serial execution. The second one is a heavy recursive rule that requires several iterations to be grounded. In this case, a good load balancing is obtained thanks to the redistributions applied (with possibly different split sizes) at each iteration of the semi-naïve algorithm.

The instantiator is effective also in Golomb Ruler, Timetabling, and Sudoku, where the performance results appear to be good also thanks to a well-balanced workload distribution.

About Food, a super-linear speedup (due to the first levels of parallelism) is already evident with two-processors and efficiency peaks when three processors are enabled, where the execution times becomes negligible. The behavior of the system for instances of varying sizes was analyzed in more detail in the case of Hamiltonian Path and 3-Colorability; this was made possible by the availability of generators.

Looking at Figures 4(a) and 4(c), it is evident that the efficiency of the system rapidly reaches a good level (ranging from 0.9 up to 1), moving from small instances (requiring less than 2 s) to larger ones, and remains stable (the surfaces are basically plateaux). The corresponding gains are visible by looking at Figures 4(b) and 4(d), where, e.g., a Hamiltonian Path (3-Colorability) instance is evaluated in 332.78 s (965.36 s) by the serial system, and requires only 68.26 s (124.70 s) with $\texttt{levels}_{1+2+3}$ with eight-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, 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 are 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 eight CPUs are available.

7 Related work

Several works about parallel techniques for the evaluation of ASP programs have been proposed, focusing on both the propositional (model search) phase (Finkel *et al.* 2001; Pontelli and El-Khatib 2001; Gressmann *et al.* 2005; Ellguth *et al.* 2009), and the instantiation phase (Balduccini *et al.* 2005; Calimeri *et al.* 2008). 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

https://doi.org/10.1017/S1471068411000652 Published online by Cambridge University Press

studies were carried out in Balduccinni *et al.* (2005) 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 (Stallings 1998), while the one described in Balduccini *et al.* (2005) is actually a Beowulf cluster working in local memory. Moreover, the parallel instantiation strategy of Balduccini *et al.* (2005) is applicable only to a subset of the program rules (those not defining domain predicates), and is, in general, unable to exploit parallelism fruitfully in the case of programs with a small number of rules. Importantly, the parallelization strategy of Balduccini *et al.* (2005) *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 evaluation) while the instantiation process is running. Note also that our parallelization techniques could be adapted for improving other ASP instantiators like Lparse (Niemelä and Simons 1997) and Gringo (Gebser *et al.* 2007b). Concerning other related works, it is worth remembering that, the Single-Rule parallelism employed in our system is related to the *copy and constrain* technique for parallelizing the evaluation of deductive databases (Wolfson and Silberschatz 1988; Ganguly *et al.* 1990; Wolfson and Ozeri 1990; Dewan *et al.* 1994; Zhang *et al.* 1995). In many of the mentioned works (dating back to 1990s), only restricted classes of Datalog programs are parallelized; whereas, the most general ones (reported in Wolfson and Ozeri 1990; Zhang *et al.* 1995) are applicable to normal Datalog programs. Clearly, none of them consider the peculiarities of disjunctive programs and unstratified negation. More in detail, Wolfson and Ozeri (1990) provide the theoretical foundations for the *copy and constrain* technique, whereas Zhang *et al.* (1995) enhance it in such a way that the network communication overhead in distributed systems can be minimized. The copy and constrain technique works as follows: rules are replicated with additional constraints attached to each copy; such constraints are generated by exploiting a hash function and allow for selecting a subset of the tuples. The obtained restricted rules are evaluated in parallel. 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 Wolfson and Ozeri (1990) and Zhang *et al.* (1995), copied rules are generated and statically associated to instantiators according to a hash function which is independent of the current instance in input. In contrast, 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 (Zhang *et al.* 1995) of choosing a 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 us to modify either the standard instantiation procedure (for efficiently selecting the tuples from the predicate extensions according to added constraints) or to incur a possible non-negligible overhead due to their evaluation. Focusing on the *heuristics* employed on parallel databases, we mention Dewan *et al.* (1994) and Carey and Lu (1986).

https://doi.org/10.1017/S1471068411000652 Published online by Cambridge University Press

In Dewan *et al.* (1994), a heuristics is described 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 according to predictive dynamic load balancing (PDLB) protocols that "update and reorganize the distribution of workload at runtime by modifying the restrictions on versions of the rule program" (Dewan *et al.* 1994). In Carey and Lu (1986), 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 at several sites and their application to other architectures might be neither viable nor straightforward.

8 Conclusions

In this paper, we present some advanced techniques for the parallel instantiation of ASP programs, and a parallel ASP instantiator based on the DLV system. In particular, we have proposed and implemented a three-level parallelization technique, dynamic load balancing, and granularity control strategies. An experimental analysis outlines significant performance improvements, larger applicability w.r.t. existing approaches, as well as nearly optimal efficiency and steady scalability of the implemented instantiator.

As far as future work is concerned, we are studying other techniques for further exploiting parallelism in ASP systems, considering also the other phases of the computation. Automatic determination of heuristics thresholds is also under investigation.

Acknowledgements

This work has been partially supported by Regione Calabria and EU under POR Calabria FESR 2007–2013 within the PIA project of DLVSYSTEM s.r.l.

References

Aiello, L. C. and Massacci, F. 2001. Verifying security protocols as planning in logic programming. *ACM Transactions on Computational Logic 2,* 4, 542–580.

Anger, C., Konczak, K. and Linke, T. 2001. NoMoRe: A system for non-monotonic reasoning. In *Proc. of the 6th International Conference on Logic Programming and Nonmonotonic Reasoning*. LPNMR '01, vol. 2173. Springer-Verlag, Berlin, 406–410.

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

Baral, C. 2003. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, Cambridge, UK.

Baral, C. and Uyan, C. 2001. Declarative specification and solution of combinatorial auctions using logic programming. In *Proc. of the 6th International Conference on Logic Programming and Nonmonotonic Reasoning*. LPNMR '01, vol. 2173. Springer-Verlag, Berlin, 186–199.

https://doi.org/10.1017/S1471068411000652 Published online by Cambridge University Press

Bell, C., Nerode, A., Ng, R. T. and Subrahmanian, V. 1994. Mixed integer programming methods for computing nonmonotonic deductive databases. *Journal of the ACM 41*, 1178–1215.

Calimeri, F., Perri, S. and Ricca, F. 2008. Experimenting with parallelism for the instantiation of ASP programs. *Journal of Algorithms in Cognition, Informatics and Logics 63,* 1–3, 34–54.

Carey, M. J. and Lu, H. 1986. Load balancing in a locally distributed db system. In *Proc. of the 1986 International Conference on Management of Data*. SIGMOD '86, vol. 15. ACM, New York, 108–119.

Dantsin, E., Eiter, T., Gottlob, G. and Voronkov, A. 2001. Complexity and expressive power of logic programming. *ACM Computing Surveys 33,* 3, 374–425.

Denecker, M., Vennekens, J., Bond, S., Gebser, M. and Truszczyński, M. 2009. The second answer set programming competition. In *Proc. of the 10th International Conference on Logic Programming and Nonmonotonic Reasoning*. LPNMR '09, vol. 5753. Springer-Verlag, Berlin, 637–654.

Dewan, H. M., Stolfo, S. J., Hernández, M. and Hwang, J.-J. 1994. Predictive dynamic load balancing of parallel and distributed rule and query processing. In *Proc. of the 1994 International Conference on Management of Data*. SIGMOD '94, vol. 23. ACM, New York, 277–288.

Eiter, T., Gottlob, G. and Mannila, H. 1997. Disjunctive datalog. *ACM Transactions on Database Systems 22*, 364–418.

Ellguth, E., Gebser, M., Gusowski, M., Kaufmann, B., Kaminski, R., Liske, S., Schaub, T., Schneidenbach, L. and Schnor, B. 2009. A simple distributed conflict-driven answer set solver. In *Proc. of The 10th International Conference on Logic Programming and Nonmonotonic Reasoning*. LPNMR '09, vol. 5753. Springer-Verlag, Berlin, 490–495.

Faber, W., Leone, N., Mateis, C. and Pfeifer, G. 1999. Using database optimization techniques for nonmonotonic reasoning. In *Proc. of the 7th International Workshop on Deductive Databases and Logic Programming*. DDLP '99. Prolog Association of Japan, Tokyo, 135–139.

Faber, W., Leone, N. and Pfeifer, G. 2004. Recursive aggregates in disjunctive logic programs: Semantics and complexity. In *Proc. of the 9th European Conference on Artificial Intelligence*. JELIA '04, vol. 3229. Springer Verlag, Berlin, 200–212.

Finkel, R. A., Marek, V. W., Moore, N. and Truszczyński, M. 2001. Computing stable models in parallel. In *Proc. of the Symposium on Answer Set Programming, Towards Efficient and Scalable Knowledge Representation and Reasoning*. ASP'01, AAAI Press, Palo Alto, CA, 72–76.

Ganguly, S., Silberschatz, A. and Tsur, S. 1990. A framework for the parallel processing of datalog queries. In *Proc. of the 1990 International Conference on Management of Data.* SIGMOD '90, vol. 19. ACM, New York, 143–152.

Gebser, M., Kaufmann, B., Neumann, A. and Schaub, T. 2007a. Conflict-driven answer set solving. In *Proc. of the 10th International Joint Conference on Artificial Intelligence*. IJCAI '07. Morgan Kaufmann Publishers, San Mateo, CA, 386–392.

Gebser, M., Liu, L., Namasivayam, G., Neumann, A., Schaub, T. and Truszczyński, M. 2007b. The first answer set programming system competition. In *Proc. of the 9th International Conference on Logic Programming and Nonmonotonic Reasoning*. LPNMR'07, vol. 4483. Springer-Verlag, 3–17.

Gebser, M., Schaub, T. and Thiele, S. 2007c. GrinGo : A new grounder for answer set programming. In *Proc. of the 9th International Conference on Logic Programming and Nonmonotonic Reasoning*. LPNMR'07, vol. 4483. Springer-Verlag, Berlin, 266–271.

https://doi.org/10.1017/S1471068411000652 Published online by Cambridge University Press

Gelfond, M. and Leone, N. 2002. Logic programming and knowledge representation – the A-Prolog perspective. *Artificial Intelligence 138,* 1–2, 3–38.

Gelfond, M. and Lifschitz, V. 1991. Classical negation in logic programs and disjunctive databases. *New Generation Computing 9*, 365–385.

Grasso, G., Iiritano, S., Leone, N., Lio, V., Ricca, F. and Scalise, F. 2010. An asp-based system for team-building in the Gioia-Tauro seaport. In *Proc. of the 12th International Symposium on Practical Aspects of Declarative Languages*. PADL '10, vol. 5937. Springer-Verlag, 40–42.

Grasso, G., Leone, N., Manna, M. and Ricca, F. 2011. ASP at work: Spin-off and applications of the DLV system. In *Logic Programming, Knowledge Representation, and Nonmonotonic Reasoning: Essays in Honor of Michael Gelfond*, vol. 6565. Springer-Verlag, Berlin, 1–20.

Gressmann, J., Janhunen, T., Mercer, R. E., Schaub, T., Thiele, S. and Tichy, R. 2005. Platypus: A platform for distributed answer set solving. In *Proc. of the 8th International Conference on Logic Programming and Nonmonotonic Reasoning*. LPNMR '05, vol. 3662. Springer-Verlag, Berlin, 227–239.

Ielpa, S. M., Iiritano, S., Leone, N. and Ricca, F. 2009. An ASP-based system for e-tourism. In *Proc. of the 10th International Conference on Logic Programming and Nonmonotonic Reasoning*. LPNMR '09, vol. 5753. Springer-Verlag, Berlin, 368–381.

Janhunen, T., Niemelä, I., Seipel, D., Simons, P. and You, J.-H. 2006. Unfolding partiality and disjunctions in stable model semantics. *ACM Transactions on Computational Logic 7,* 1 (January), 1–37.

Knuth, D. E. 1994. *The Stanford GraphBase : A Platform for Combinatorial Computing*, vols. I–VII. ACM, New York.

Lembo, D., Lenzerini, M. and Rosati, R. 2002. Integrating inconsistent and incomplete data sources. In *Proc. of the 10th Italian Symposium on Advanced Database Systems 2005*. SEBD '05. 299–308.

Leone, N., Perri, S. and Scarcello, F. 2001. Improving ASP instantiators by join-ordering methods. In *Proc. of the 6th International Conference on Logic Programming and Nonmonotonic Reasoning*. LPNMR '01, vol. 2173. Springer-Verlag, Berlin, 280–294.

Leone, N., Perri, S. and Scarcello, F. 2004. BackJumping techniques for rules instantiation in the DLV system. In *Proc. of the 10th International Workshop on Non-monotonic Reasoning*. NMR '04. 258–266.

Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S. and Scarcello, F. 2006. The DLV system for knowledge representation and reasoning. *ACM Transactions on Computational Logic 7*, 499–562.

Lierler, Y. and Maratea, M. 2004. Cmodels-2: SAT-based answer set solver enhanced to non-tight programs. In *Proc. of the 7th International Conference on Logic Programming and Non-Monotonic Reasoning*. LPNMR '04, vol. 2923. Springer Verlag, Berlin, 346–350.

Lifschitz, V. 1999. Answer set planning. In *Proc. of the 16th International Conference on Logic Programming*. ICLP '99. The MIT Press, Cambridge, MA, USA, 23–37.

Lin, F. and Zhao, Y. 2004. ASSAT: Computing answer sets of a logic program by SAT solvers. *Artificial Intelligence 157,* 1–2, 115–137.

Marek, V. W. and Truszczyński, M. 1999. Stable models and an alternative logic programming paradigm. In *The Logic Programming Paradigm – A 25-Year Perspective*. Springer Verlag, Berlin, 375–398.

Niemelä, I. and Simons, P. 1997. Smodels – an implementation of the stable model and well-founded semantics for normal lp. In *Proc. of the 4th International Conference on Logic*

https://doi.org/10.1017/S1471068411000652 Published online by Cambridge University Press

Programming and Nonmonotonic Reasoning. LPNMR '97, vol. 1265. Springer-Verlag, Berlin, 421–430.

Perri, S., Ricca, F. and Sirianni, M. 2010. A parallel ASP instantiator based on DLV. In *Proc. of the 5th ACM SIGPLAN Workshop on Declarative Aspects of Multicore Programming*. DAMP '10. ACM, New York, 73–82.

Perri, S., Ricca, F. and Vescio, S. 2008. Efficient parallel ASP instantiation via dynamic rewriting. In *Proc. of the First Workshop on Answer Set Programming and Other Computing Paradigms*. ASPOCP '08, 16–30.

Pontelli, E. and El-Khatib, O. 2001. Exploiting vertical parallelism from answer Set programs. In *Proc. of the Symposium on Answer Set Programming, Towards Efficient and Scalable Knowledge Representation and Reasoning*. ASP'01, AAAI Press, Palo Alto, CA, 174–180.

Simons, P. 2000. *Extending and Implementing the Stable Model Semantics*. PhD thesis, Helsinki University of Technology, Finland.

Simons, P., Niemelä, I. and Soininen, T. 2002. Extending and implementing the stable model semantics. *Artificial Intelligence 138*, 181–234.

Stallings, W. 1998. *Operating Systems: Internals and Design Principles*, 3rd ed. Prentice-Hall, Englewood Cliffs, NJ.

Subrahmanian, V., Nau, D. and Vago, C. 1995. WFS + branch andbound = stable models. *IEEE Transactions on Knowledge and Data Engineering 7*, 362–377.

Ullman, J. D. 1989. *Principles of Database and Knowledge Base Systems*. Computer Science Press, New York, NY, USA.

Wolfson, O. and Ozeri, A. 1990. A new paradigm for parallel and distributed rule-processing. In *Proc. of the 1990 International Conference on Management of Data*. SIGMOD '90, vol. 19. ACM, New York, 133–142.

Wolfson, O. and Silberschatz, A. 1988. Distributed processing of logic programs. In *Proc. of the 1988 International Conference on Management of Data*. SIGMOD '88, vol. 17. ACM, New York, 329–336.

Zhang, W., Wang, K. and Chau, S.-C. 1995. Data partition and parallel evaluation of datalog programs. *IEEE Transactions on Knowledge and Data Engineering 7*, 163–176.

https://doi.org/10.1017/S1471068411000652 Published online by Cambridge University Press