Towards a Fully-Parallel DLV System

Simona Perri, Francesco Ricca, and Marco Sirianni

Department of Mathematics
University of Calabria
87030 Rende, Italy
perri,ricca,sirianni@mat.unical.it

Abstract

In this paper we report on the first attempts to customize and exploit in DLV, a state-of-the-art Answer Set Programming system, both novel and existing parallelization methods for the propositional search phase. These techniques, combined with the recently proposed strategies for parallel instantiation, will pave the way for obtaining a fully parallel DLV system.

The results of an experimental analysis are also reported, showing the impact of parallel techniques on the performance of the DLV system.

1 Introduction

Answer Set Programming (ASP) [16, 26] is a purely declarative programming paradigm based on nonmonotonic reasoning and logic programming.

The idea of answer set programming is to represent a given computational problem by a logic program the answer sets of which correspond to solutions, and then, use an answer set solver to find such solutions [26]. The main advantage of ASP is its high declarative nature combined with a relatively high expressive power [22, 9]; but this comes at the price of a high computational cost, which makes the implementation of efficient ASP systems a difficult task.

However, the development of efficient ASP systems [22, 20, 19, 24, 35, 28, 15, 27, 25, 2, 1], and among them the state-of-the-art system DLV [22], made ASP an ideal tool for developing complex real-world applications, a fact confirmed by the increasing number of ASP applications in both the fields of AI and Knowledge Management [17, 32, 8, 21].

At the time of this writing, the majority of the available ASP implementations is not able to take advantage during all the steps of the computation from the latest hardware, featuring multi-core/multi-processor SMP (Symmetric MultiProcessing) technologies also for entry-level systems and PCs.

In particular, concerning the DLV system, a first step in this direction has been recently done by applying parallelism to the preliminary program instantiation phase [6, 29]; however model generation and checking, the two remaining phases of the computation, still rely on serial algorithms. On

Proceedings of the 17th International RCRA workshop (RCRA 2010):
Experimental Evaluation of Algorithms for Solving Problems with Combinatorial Explosion
Bologna, Italy, June 10–11, 2010

the contrary, for some other ASP systems, specialized versions exploiting parallelism in the propositional phase have been developed, but still rely on a serial instantiation[14, 10, 18, 30].[1]

In this paper we report on the first attempts to customize and exploit in DLV both novel and existing parallelization methods for the propositional search phase in order to obtain a fully parallel version of the DLV system. We start by focusing on the computation of a single answer set, and try to combine well-known parallel lookahead techniques [3] with a multi-heuristics parallel search strategy.

The results of an experimental analysis are also presented, showing the impact of parallel techniques on the performance of the DLV system.

2 The DLV System

In this Section, first we provide a short description of the syntax and the semantics of the kernel language of DLV which is disjunctive datalog under the answer sets semantics [16]; then we outline the general architecture of the system; finally, we focus on the computation performed by the *Model Generator*, the module of the system subject of the improvements presented in this paper.

2.1 The Core Language

Syntax. A variable or a constant is a *term*. An *atom* is $a(t_1, \ldots, t_n)$, where a 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. Given a literal L, we define its *complementary literal* `not` $.L$ as follows: `not` $.L = p$ if L is of the form `not` p, otherwise `not` $.L =$ `not` p.

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*.

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.

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.

[1] Actually, some studies on parallel instantiation were carried out in [3] which remained at a preliminary stage.

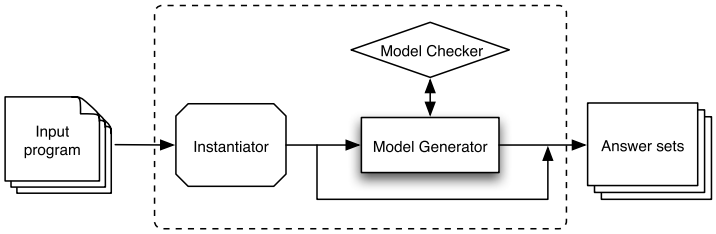

Figure 1: General architecture of the DLV system.

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 p is *true* (resp., *false*) w.r.t. I if $p \in I$ (resp., $p \notin I$). A ground negative literal `not` p is *true* w.r.t. I if p is false w.r.t. I; otherwise `not` p 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 [12], simplifies the original definition of Gelfond-Lifschitz (GL) transform [16], but is fully equivalent to the GL transform for the definition of answer sets [12].

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 $\mathcal{P}^I$) [31, 16]. The set of all answer sets for $\mathcal{P}$ is denoted by $ANS(\mathcal{P})$.

2.2 Architecture

An outline of the general architecture of the system is depicted in Fig.1.

Upon startup, the input specified by the user is parsed and transformed into the internal data structures of the system. In general, an input program $\mathcal{P}$ contains variables, and the first step of the DLV computation, performed by the *Instantiator* module, is to eliminate these variables, generating a

```
bool ModelGenerator ( Interpretation& I )
{
      I = Propagate ( I );
      if ( I == ℒ ) return false; (* inconsistency *)
      if ( "no atom is undefined in I" )
        return IsAnswerSet(I);
      Literal L;
      if ( ! Select(I, L) ) return False;
      if ( MG ( I ∪ {L} ) return True;
      else return MG ( I ∪ {not .L} );
};
```

Figure 2: Computation of Answer Sets

ground instantiation $ground(\mathcal{P})$ of $\mathcal{P}$. This process, called *instantiation* (or *grounding*), is much more than a simple variables-elimination; it allows to evaluate relevant programs fragments, and produces a ground program which has precisely the same answer sets as the theoretical instantiation, but it is sensibly smaller in size. Moreover, if the input program is disjunction-free and stratified, then its evaluation is completely done by the instantiator which computes the single answer set.

The subsequent computations, which constitute the non-deterministic part of the DLV system, are then performed on $ground(\mathcal{P})$ by both the *Model Generator* and the *Model Checker*. Roughly, the former produces some "candidate" answer set, whose stability is subsequently verified by the latter. The computation performed by the Model Generator exploits a backtracking search algorithm, which works directly on the ground instantiation of the input program. When the Model Generator produces an answer set candidate, the Model Checker verifies whether it is an answer set for the input program. Note that, the task performed by the Model Checker is as hard as the problem solved by the Model Generator for disjunctive programs, while it is trivial for non-disjunctive programs. However, there is also a class of disjunctive programs, called Head-Cycle-Free programs [4], for which the task solved by the Model Checker is provably simpler, which is exploited in the system algorithms. Finally, once an answer set has been found, it is printed and possibly the Model Generator resumes in order to look for further answer sets.

2.3 The Model Generator

The computation performed by the Model Generator, is outlined in Figure 2. Note that the description here is quite simplified, since the details of the actual implementation are not important for this paper. For instance, the algorithm presented here decides whether an answer set exists, but it can be straightforwardly extended to compute all or a fixed number of answer

sets as the DLV system does. A more detailed description can be found in [11].

The *ModelGenerator* function is first called with parameter I set to the empty interpretation. If the program $\mathcal{P}$ has an answer set, then the function returns true setting I to the computed answer set; otherwise it returns false. The Model Generator is similar to the Davis-Putnam procedure employed by SAT solvers. It first calls a function *Propagate*, which returns the extension of I with the literals that can be deterministically inferred (or the set of all literals $\mathcal{L}$ upon inconsistency). This function is similar to a unit propagation procedure employed by SAT solvers, but exploits the peculiarities of ASP for making further inferences (e.g., it exploits the knowledge that every answer set is a minimal model). If *Propagate* does not detect any inconsistency, an undefined[2] literal L is selected according to a heuristic criterion (by a call to the *Select* procedure) and *ModelGenerator* is called on $I \cup \{L\}$ and on $I \cup \{\texttt{not}\ .L\}$. The literal L corresponds to a *branching variable* in SAT solvers. And indeed, like for SAT solvers, the selection of a "good" literal L is crucial for the performance of an ASP system. To this end DLV employes look-ahead heuristics [13] where the system looks ahead by tentatively assuming each undefined literal L and its complement. Then, the heuristic value of L (which is a measure of the "quality" of the resulting interpretations) is exploited to select the next branching literal.

Looking ahead is a comparatively costly operation, the computation of this kind of heuristics can be very expensive, since the number of literals to be "looked-ahead" may be very large; thus, the lookahead step is a good candidate for exploiting parallelism [3]. Moreover, no heuristics measure fits perfectly all cases, and it might even be the case that the costs of the lookahead largely overcome its benefits; parallelism might help also in this case by allowing for concurrently exploring different search paths.

3 Pushing Parallelism in the Model Generator

In this Section we present the first attempts to exploit parallelism in the Model Generator. In this preliminary work, we focus on the computation of a single answer set and we investigate two directions: first, we act on the evaluation of the heuristically best literal, by making use of techniques of parallel look-ahead; then we exploit a multi-heuristics parallel search strategy taking advantage of different heuristic criteria.

[2]The interpretations built during the computation are 3-valued, that is a literal can be true, false or undefined (if its value has not been set yet) w.r.t. to an interpretation I.

3.1 Parallel Lookahead

We now sketch a framework which exploits parallelism for the selection of branching literals, according to a heuristic criterion.

The general procedure evaluating the heuristically best literal is shown in Figure 3. Roughly, some threads are spawned which run function Lookahead (calls to *pthread_create*). Each thread takes a different undefined literal A to perform a look-ahead step by considering first $I \cup \{A\}$ and then $I \cup \{\texttt{not } .A\}$. Lookahead, either calculates the two heuristic values (results are stored in I_A^+ and I_A^-) by calling function Propagate, or if an inconsistency is encountered ($\mathrm{I}_\mathrm{A}^+ = \mathcal{L}$ or $\mathrm{I}_\mathrm{A}^- = \mathcal{L}$), stores the complement of the propagated literal in the *detChoices* set. Once all threads have terminated (they reach the barrier), literals in *detChoices*, which can be deterministically assumed, are propagated. If an inconsistency arises at this point, then no literal can be chosen at this level (the assumption of both A and its complement `not` $.A$ leads to inconsistency) and the function Select returns false, in order to cause a backtracking.[3] Otherwise, the best literal according to a given heuristic criterion h_C is selected among the candidate ones.

3.2 Multi-Heuristics Parallel Search Strategy

The heuristics for the selection of the branching literal dramatically affects the performance of an ASP system; and, obviously, there is no heuristics performing well in all cases, rather a heuristics can be more suitable than another one for a given problem typology, or even for a specific problem instance. In order to obtain an ASP solver which is able to always adopt the best known heuristics, we exploit a multi-heuristics strategy. More in detail, we run a number of Model Generator instances, each one adopting a different branching criterion. Since we are interested in looking for just one answer set, such a strategy corresponds to exploring different paths in the search space, concurrently. The computation stops when the first instance of Model Generator terminates, thus ensuring the best possible performance for the given problem instance.

In the following we briefly recall the exploited heuristic criteria. For fast prototyping, we considered the ones already supported by the DLV system. Further details can be found in [13].

Heuristic $\mathbf{h_{satz}}$. This is an extension of the branching rule adopted in the system SATZ [23] to the framework of DLP. The basic idea is to prefer literals introducing a higher number of "short" (that is, where few undefined literals occur) unsatisfied rules. Intuitively, the introduction of a high number of short unsatisfied rules is preferred because it creates more

[3] The process is described here in a simplified way for reason of presentation. In the actual implementation, the number of Lookahead threads can be fixed and in case of inconsistency, threads are stopped as soon as the propagation of both L and `not` $.L$ fails.

```
bool Select(Interpretation& I, Literal& L)
{
    Interpretation I_A^+, I_A^-; ListOfThreadsID thList = ∅;
    ThreadSafeSetOfLiterals DetChoices = ∅;
    foreach Literal A undefined w.r.t. I do      (* spawn lookahead threads *)
        ThreadID id;
        pthread_create(id, Lookahead, I, A, detChoices);
        th_list.add(id);
    foreach ThreadID thread ∈ thList do          (* barrier *)
        pthread_join(thread);
    I = Propagate(I ∪ detChoices);               (* assume literals that can be deterministically propagated *)
    if (I == L) return false;
    L = NULL;                                    (* compute the heuristically best literal *)
    foreach Literal A undefined w.r.t. I do
        if (L == NULL)
            L = A;                               (* first literal, no comparison *)
        else if (L <_hC A)                       (* compare A against L w.r.t. the heuristics *)
            L = A;
    return true;
}

void Lookahead( Interpretation I, Literal A, ThreadSafeSetOfLiterals& detChoices)
{
    I_A^+ = Propagate(I ∪ {A});                  (* look-ahead for A *)
    if (I_A^+ == L)
        detChoices.add(not .A);
    I_A^- := Propagate(I ∪ {not .A});            (* look-ahead for not .A *)
    if (I_A^- == L)
        detChoices.add(A);
}
```

Figure 3: Framework for the selection of the branching literal in DLV

and stronger constraints on the interpretation so that a contradiction can be found earlier.

Heuristic $\mathbf{h_{std}}$. This is the heuristics used as default in the DLV system. It is based on a peculiar property of answer sets, the *supportdness*: for each true atom A of an answer set I, there exists a rule r of the program such that the body of r is true w.r.t. I and A is the only true atom in the head of r [13]. Since an ASP system must eventually converge to a supported interpretation, it makes sense to keep the interpretations "as much supported as possible" during the intermediate steps of the computation. To this end, the heuristics takes into account the value of several counters obtained at the end of the propagation: number of *UnsupportedTrue (UT)* atoms, i.e., atoms which are true in the current interpretation but still miss a supporting rule; the total number $Sat(L)$ of rules which are satisfied; the degree of supportedness, that is the average number of supporting rules for a true

atom. The heuristics $\mathbf{h_{std}}$ of DLV considers these measures in a prioritized way, to favor atoms yielding interpretations with fewer UT atoms (which should more likely lead to a supported model). If the UT counters are equal, then the heuristics considers the total number $Sat(L)$ of rules which are satisfied. And, finally, for hard programs (non Head Cycle Free [4]), $\mathbf{h_{std}}$ considers atoms leading to an higher degree of supportedness in order to drive the computation toward supported models having higher chances to be answer set, with the goal of reducing the overall number of the expensive calls to the Model Checker.

4 Experiments

In this Section we report the results of an experimental analysis carried out on a prototypical version of the DLV system exploiting the above described techniques. Parallelism is introduced in the system by exploiting the POSIX pthreads standard libraries.

4.1 Benchmark Problems and Instances

We considered three well-known problems which are usually exploited for evaluating Model Generator performances:

- **3SAT** is a special case of SAT, one of the best researched problems in AI, which amounts to verify if it exists a truth assignment which satisfies a CNF formula having at most tree variables per clause. The instances for 3SAT were randomly generated by using a tool by Selman and Kautz [33]. For each size we generated 20 such instances, where we kept the ratio between the number of clauses and the number of variables at 4.3, which is near the cross-over point for random 3SAT [7].

- **HAMPATH** 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 instances for HAMPATH were generated by a tool by Patrik Simons (cf. [34]). For each problem size n we generated 20 instances, always assuming node 1 as the starting node.

- **STRATCOMP** Is the problem of computing companies that are "strategic" according with the following definition [5]. A holding owns companies $C(1), \ldots, C(c)$, each of which produces some goods. Some of these companies may jointly control another one. Now, some companies should be sold, under the constraint that all goods can still be produced, and that no company is sold which would still be controlled by the holding afterwards. A company is strategic, if it belongs to

a *strategic set*, which is a minimal set of companies satisfying these constraints.

For STRATCOMP, we randomly generated 20 instances for each problem size n, with n companies and n products.

The first two benchmarks (3SAT and HAMPATH) are well-known NP-complete problems, while the third (STRATCOMP) is a Σ_2^P-complete problem.

4.2 Experimental Results

The machine used for the experiments is a multi-processor Intel Xeon "Woodcrest" (quad core) 3GHz machine with 4MB of L2 Cache and 4GB of RAM, running Debian GNU Linux 4.0.

We have allowed at most 600 seconds (ten minutes) of execution time for each instance. The experimentation has been stopped (for each system) at the size at which some instance exceeded this time limit.

The results of our experiments are displayed in the graphs of Figure 4. For each problem domain we report two graphs, describing the behavior of the two tested parallel techniques: In both graphs the horizontal axis reports a parameter representing the size of the instance, while on the vertical axis we report the running time (expressed in seconds) averaged over the instances of the same size we have run.

Since our techniques focus on model generation, all the results of the experimental analysis refer only to the process of computing answer sets of ground programs.

Evaluation of Parallel Lookahead. In order to assess the impact of the parallel lookahead we considered a number of variants of the same prototype:

- **dl.X.satz** DLV with lookahead exploiting $\mathbf{h_{satz}}$ and $X = [1, 2, 3]$ threads.
- **dl.X.std** DLV with lookahead exploiting $\mathbf{h_{std}}$ and $X = [1, 2, 3]$ threads.

The results shown in Figure 4(a), in Figure 4(c) and in Figure 4(e) report the performance of those variants on the considered instances of 3SAT, HAMPATH and STRATCOMP, respectively.

Concerning 3SAT, the best heuristics is clearly $\mathbf{h_{satz}}$, the entire group of dl.X.satz performed better than dl.X.std group, solving all the generated instances in less time; whereas all the dl.X.std were stopped when considering instances having more than 300 variables. Moreover, it can be noted that dl.2.satz is the absolute best variant in this domain, and dl.2.std is the best among the ones exploiting $\mathbf{h_{std}}$. The lower performance of variants

with three workers[4] is due to the larger amount of time spent by threads in synchronization (perhaps a technological problem of our prototype that can be probably overcome by improving the implementation).

As far as HAMPATH is concerned, the results are clearly in favor of the group exploiting the standard heuristics. Here the standard heuristics, which takes into account peculiar properties of ASP programs, has an edge on the sat-based one. Indeed, all the dl.X.satz variants were stopped before reaching 100 nodes, while the best versions equipped with standard heuristics could solve instances having up to 120 nodes. Unfortunately here the effect of parallel lookahead is overshadowed by an internal optimization of DLV, which considers as selectable literals only a subset the available ones called PT literals (c.f.r. [13]), thus reducing the work to be divided among workers; this results in an emphasized effect of synchronization overhead which still remains acceptable (<0.9sec).

For STRATCOMP it is evident that the standard heuristics is the best, and that the effect of parallel lookahead made dl.2.std and dl.3.std to be the best variants (still dl.2 shows less overhead compared with dl.3), which were capable to solve instances having up to 290 companies; whereas, the $\mathbf{h_{satz}}$-based systems were stopped before reaching 150 companies, and dl.1.std reaches at most 170 companies.

Summarizing, results clearly show that the benefits of parallel lookahead are maximized when at most two workers are employed, and the technique can bring interesting speedups on hard instances.

Evaluation of the Multi-Heuristics Approach. The results reported above, as one can expect, show that there is no heuristics performing well in all cases (e.g. $\mathbf{h_{satz}}$ is the best for 3SAT, while in the other two domains $\mathbf{h_{std}}$ is the winner). In order to evaluate the second strategy proposed in this paper, we considered a variant implementing the multi-heuristics approach and exploiting the best parallel lookahead settings (named dl2.MultiHeu). We compared dl2.MultiHeu with the best performer in each domain among the ones considered before; the result are shown in Figure 4(b), in Figure 4(d) and in Figure 4(f), which reports the performance of those variants on 3SAT, HAMPATH and STRATCOMP, respectively.

The picture here is very clear, dl2.MultiHeu is able to move forward the limit of the the largest instance solved in 10 minutes. Still the synchronization overhead, paid to stop the concurrent model generators remains evident, but the dramatic advantage of selecting the best possible criterion per instance allows for solving larger instances in less time. For instance dl2.MultiHeu solved instances with up to 160 nodes in HAMPATH where the competitor stopped at 120 nodes; and dl2.MultiHeu solved up to 3000 companies vs 2900. The overall performance leap becomes dramatic when

[4]We tried also variants with more than three workers confirming this statement; results have been omitted for the sake of readability.

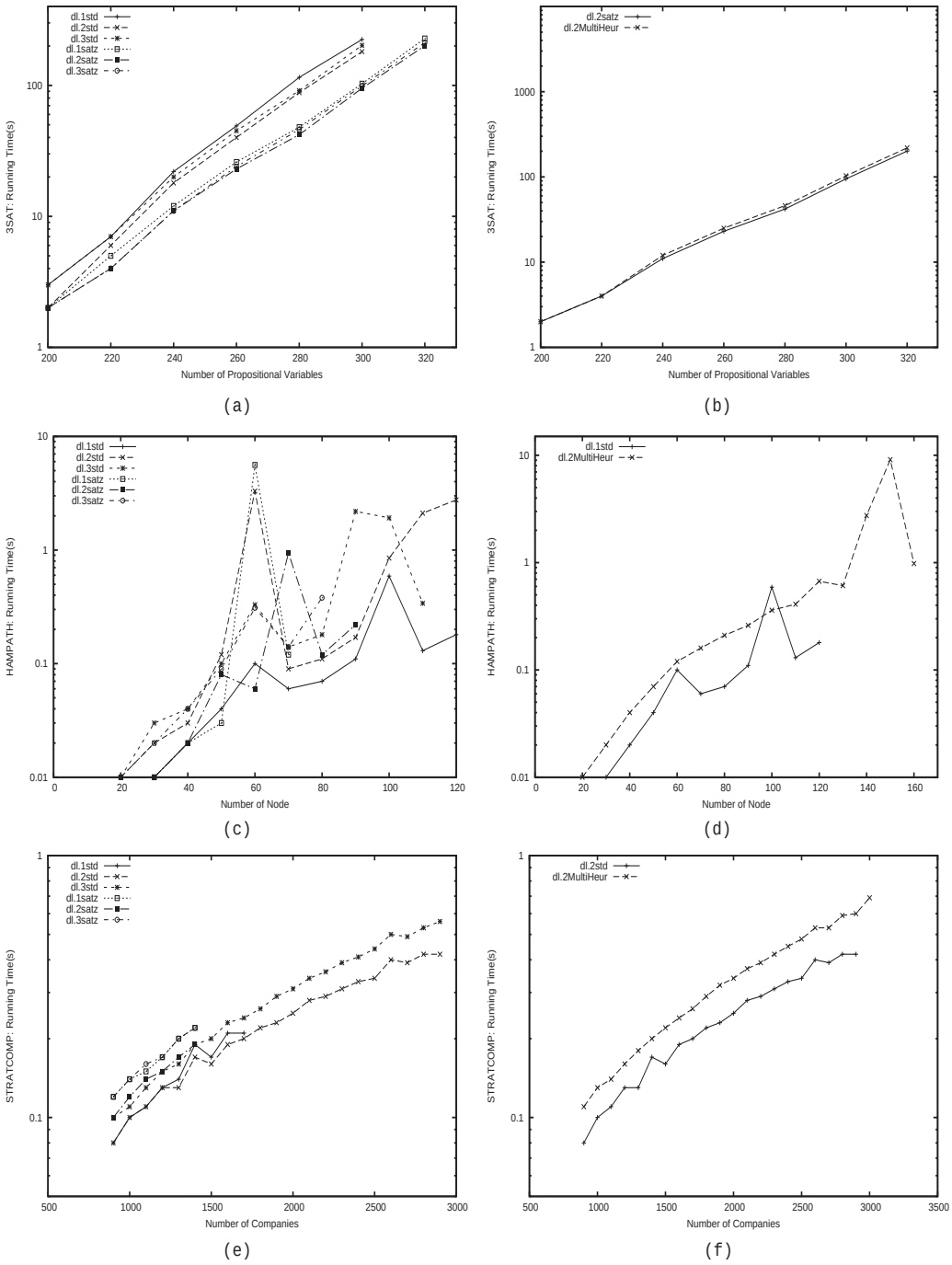

Figure 4: Experimental Results.

dl2.MultiHeu is compared with the original dl1.std: 300 vs 320 propositional variables for 3SAT; 120 vs 160 nodes for HAMPATH; and 1700 vs 3000 companies for STRATCOMP.

5 Conclusion

In this paper we reported on the first attempts to introduce parallelism in the DLV system. In particular, we focused on the Model Generator module and we experimented with two parallel techniques: parallel lookahead and multi-heuristic search strategy. The results show that great benefits can be already obtained by exploiting simple parallelization strategies in the expensive phase of the answer sets computation. Nevertheless our work is still in a preliminary stage: much has to be done in order to obtain an efficient and solid implementation; moreover, more sophisticated parallelization techniques, specifically conceived for DLV have to be designed, probably requiring stronger modifications to the original system implementation. This is subject of future work. Once a more stable parallel system will be obtained, we plan to make a larger experimental activity considering more domains, more techniques (both novel and well-known), and involving comparison with other already well-assessed parallel systems.

Acknowledgements

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

References

[1] C. Anger, M. Gebser, T. Linke, A. Neumann, and T. Schaub. The nomore++ Approach to Answer Set Solving. In *Logic for Programming, Artificial Intelligence, and Reasoning, 12th International Conference, LPAR 2005*, LNCS 3835, pp. 95–109. Dec. 2005.

[2] C. Anger, K. Konczak, and T. Linke. `NoMoRe`: A System for Non-Monotonic Reasoning. In *LPNMR'01*, LNCS 2173, pp. 406–410. Sept. 2001.

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

[4] R. Ben-Eliyahu and R. Dechter. Propositional Semantics for Disjunctive Logic Programs. *AMAI*, 12:53–87, 1994.

[5] M. Cadoli, T. Eiter, and G. Gottlob. Default Logic as a Query Language. *IEEE TKDE*, 9(3):448–463, 1997.

[6] F. Calimeri, S. Perri, and F. Ricca. Experimenting with Parallelism for the Instantiation of ASP Programs. *Journal of Algorithms in Cognition, Informatics and Logics*, 63(1–3):34–54, 2008.

[7] J. M. Crawford and L. D. Auton. Experimental Results on the Crossover Point in Random 3SAT. *AI*, 81(1–2):31–57, Mar. 1996.

[8] C. Cumbo, S. Iiritano, and P. Rullo. Reasoning-Based Knowledge Extraction for Text Classification. In *Proceedings of Discovery Science, 7th International Conference*, pp. 380–387, Padova, Italy, Oct. 2004.

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

[10] E. Ellguth, M. Gebser, M. Gusowski, B. Kaufmann, R. Kaminski, S. Liske, T. Schaub, L. Schneidenbach, and B. Schnor. A simple distributed conflict-driven answer set solver. In *LPNMR*, LNCS 5753, pp. 490–495. 2009.

[11] W. Faber. *Enhancing Efficiency and Expressiveness in Answer Set Programming Systems*. PhD thesis, TU Wien, 2002.

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

[13] W. Faber, N. Leone, G. Pfeifer, and F. Ricca. On look-ahead heuristics in disjunctive logic programming. *AMAI*, 51(2–4):229–266, 2007.

[14] R. A. Finkel, V. W. Marek, N. Moore, and M. Truszczynski. 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*, pp. 72–76, Mar. 2001.

[15] M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub. Conflict-driven answer set solving. In *IJCAI 2007*, pp. 386–392. Jan. 2007.

[16] M. Gelfond and V. Lifschitz. Classical Negation in Logic Programs and Disjunctive Databases. *NGC*, 9:365–385, 1991.

[17] G. Grasso, S. Iiritano, N. Leone, and F. Ricca. Some DLV Applications for Knowledge Management. In *Proceedings of the 10th International Conference on Logic Programming and Nonmonotonic Reasoning* (*LPNMR 2009*), LNCS 5753, pp. 591–597. 2009.

[18] J. Gressmann, T. Janhunen, R. E. Mercer, T. Schaub, S. Thiele, and R. Tichy. Platypus: A Platform for Distributed Answer Set Solving. In *Proceedings of Logic Programming and Nonmonotonic Reasoning, 8th International Conference (LPNMR)*, pp. 227–239, Diamante, Italy, Sept. 2005.

[19] T. Janhunen and I. Niemelä. Gnt - a solver for disjunctive logic programs. In *LPNMR-7*, LNCS 2923, pp. 331–335. Jan. 2004.

[20] T. Janhunen, I. Niemelä, D. Seipel, P. Simons, and J.-H. You. Unfolding Partiality and Disjunctions in Stable Model Semantics. *ACM TOCL*, 7(1):1–37, Jan. 2006.

[21] N. Leone, G. Gottlob, R. Rosati, T. Eiter, W. Faber, M. Fink, G. Greco, G. Ianni, E. Kałka, D. Lembo, M. Lenzerini, V. Lio, B. Nowicki, M. Ruzzi, W. Staniszkis, and G. Terracina. The INFOMIX System for Advanced Integration of Incomplete and Inconsistent Data. In *SIGMOD 2005*, pp. 915–917, Baltimore, Maryland, USA, 2005. ACM Press.

[22] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello. The DLV System for Knowledge Representation and Reasoning. *ACM TOCL*, 7(3):499–562, 2006.

[23] C. Li and Anbulagan. Heuristics Based on Unit Propagation for Satisfiability Problems. In *IJCAI 1997*, pp. 366–371, Nagoya, Japan, Aug. 1997.

[24] Y. Lierler. Disjunctive Answer Set Programming via Satisfiability. In *LPNMR'05*, LNCS 3662, pp. 447–451. Sept. 2005.

[25] Y. Lierler and M. Maratea. Cmodels-2: SAT-based Answer Set Solver Enhanced to Non-tight Programs. In *LPNMR-7*, LNCS 2923, pp. 346–350. Jan. 2004.

[26] V. Lifschitz. Answer Set Planning. In *ICLP'99*, pp. 23–37.

[27] F. Lin and Y. Zhao. ASSAT: computing answer sets of a logic program by SAT solvers. *AI*, 157(1–2):115–137, 2004.

[28] I. Niemelä, P. Simons, and T. Syrjänen. Smodels: A System for Answer Set Programming. In *NMR'2000*, Apr. 2000. Online at `http://xxx.lanl.gov/abs/cs/0003033v1`.

[29] S. Perri, F. Ricca, and M. Sirianni. A parallel asp instantiator based on dlv. In *DAMP '10: Proceedings of the 5th ACM SIGPLAN workshop on Declarative aspects of multicore programming*, pp. 73–82, New York, USA, 2010. ACM.

[30] E. Pontelli and O. El-Khatib. 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*, pp. 174–180, Mar. 2001.

[31] T. C. Przymusinski. Stable Semantics for Disjunctive Programs. *NGC*, 9:401–424, 1991.

[32] M. Ruffolo, N. Leone, M. Manna, D. Saccà, and A. Zavatto. Exploiting ASP for Semantic Information Extraction. In *Proceedings ASP05 - Answer Set Programming: Advances in Theory and Implementation*, pp. 248–262, Bath, UK, 2005.

[33] B. Selman and H. Kautz, 1997. `ftp://ftp.research.att.com/dist/ai/`.

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

[35] P. Simons, I. Niemelä, and T. Soininen. Extending and Implementing the Stable Model Semantics. *AI*, 138:181–234, 2002.