The DLV parallel grounder

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

Department of Mathematics, University of Calabria, 87036 Rende (CS), Italy
{perri,ricca,sirianni}@mat.unical.it

1 Introduction

Answer Set Programming (ASP) [1, 2], a purely declarative programming paradigm based on nonmonotonic reasoning and logic programming. The high expressive power of ASP has been profitably exploited for developing advanced applications belonging to several fields, from Artificial Intelligence [3–7] to Information Integration [8], and Knowledge Management [9]. Interestingly, these applications of ASP recently have stimulated some interest also in industry [10].

The idea of answer set programming is to represent a given computational problem by a logic program the answer sets of which correspond to solutions, and then, use an answer set solver to find such solutions [2]. The main construct of the language of ASP is the logic rule.[1] ASP rules allow (in general) both disjunction in the head and nonmonotonic negation in the body. Importantly, ASP is declarative, and the ASP encoding of a variety of problems is very concise, simple, and elegant [3, 9, 13, 7].

As an example, consider the well-known 3-colorability problem. Given a graph $G = (V, E)$, assign each node one of three colors (say, red, green, or blue) such that adjacent nodes always have distinct colors. First of all, we represent G by introducing suitable facts: $vertex(v).\ \forall v \in V$, and $edge(v_1, v_2).\ \forall (v_1, v_2) \in E$; then, an ASP program solving the 3-colorability problem is the following:

$$(r_1) \quad col(X, red) \vee col(X, green) \vee col(X, blue) \text{ :- } vertex(X).$$
$$(r_2) \quad \text{ :- } edge(X, Y), col(X, C), col(Y, C).$$

The " :- " symbol can be read as "if", thus rule r_1 expresses that each node must either be colored red, green, or blue; due to minimality of answer sets, a node cannot be assigned more than one color. Rule r_2 acts as an integrity constraint and disallows situations in which two vertices connected by an edge are associated with the same color. Intuitively, an empty head is false, thus rule r_2 has the effect of discarding models in which the conjunction is true.

The computation of an ASP program is a two step process; the first step of evaluation is called instantiation and amounts to computing a program $\mathcal{P}'$ semantically equivalent to $\mathcal{P}$, but not containing any variable; after, $\mathcal{P}'$ is evaluated by using propositional backtracking search techniques.

The instantiation of ASP program is a crucial task for efficiency, and is particularly relevant when huge input data has to be dealt with. In this scenario, significant perfor-

[1] For introductory material on ASP, we refer to [2, 11, 9, 12].

mance improvements can be obtained by exploiting modern multi-core/multi-processor SMP machines, featuring several CPU in the same case.

In this paper, we report on the implementation of DLV$^{par}_{Gr}$ a parallel instantiator based on the state of the art ASP system DLV [14]. The resulting instantiator features three levels of parallelism [15, 16], as well as load-balancing and granularity control heuristics, which allows for effectively exploiting the processing power offered by modern multi-core/multi-processor SMP machines. The result of an experimental analysis, which was carried out on publicly-available benchmarks already exploited for evaluating the performance of instantiation systems, are also reported.

2 Parallelization Techniques and Implementation Issues

In this Section we briefly present the main new features of DLV$^{par}_{Gr}$ and, then, we enlighten some pragmatically-relevant issues regarding its implementation.

System Features. The system enjoys a three level parallel instantiation technique, enhanced with granularity control and load balancing heuristics. More in detail, the first level of parallelism allows for instantiating in parallel independent subprograms of the program in input $\mathcal{P}$. The division of $\mathcal{P}$ is performed accordingly to the *Dependency Graph*, a graph that, intuitively, describes the dependencies among the predicates of $\mathcal{P}$. Clearly, this level of parallelism is useful when the input program contains parts that are independent.

The second level of parallelism allows for evaluating in parallel rules within a given subprogram. Among these rules, we distinguish two different types, recursive and non recursive ones (also called exit rules). First, all exit rules are concurrently instantiated; then, all recursive rules are processed in parallel performing several iterations according to a seminaive schema[17]. The second level of parallelism is particulary useful when the number of rules in the subprograms is large.

The third level, allows for the parallel evaluation of a single rule. In particular it allows for subdividing the extension of a body predicate and evaluating the "split" parts in parallel. This level is crucial for the parallelization of programs with few rules, where the first two levels are almost not applicable.

Clearly, it can happen that a rule is particulary easy to be instantiated; in this case, applying the third level of parallelism, can be useless, or in the worst case, it can even slow down the instantiation of the rule. To this aim, a *granularity control* heuristics has been defined, which is based on the dynamic evaluation of the "weight" of a rule *r*; if the heuristic value is less than a given threshold, the level of parallelism is not applied. The "weight" is evaluated combining two estimations: (i) the size of the join of the body predicates of *r* (which should give an idea of the number of ground rules produced); and, (ii) an estimation of the number of comparison done for instantiating *r*. The same heuristic values are exploited for defining a *load balancing* strategy which allows for keeping the size of each single split of the body predicate large enough, in order to not introduce delays, but also sufficiently small so that it is unlikely that one split requires an instantiation time significantly larger than the others. If a rule passes the granularity control, load balancing is applied: according to the computed heuristic values, the number of splits is determined for the given rule; if this number is larger than

a given threshold, a finer redistribution of the workload is performed in the very last part of the computation. A more detailed description of these techniques and heuristics can be found in [15, 16].

Implementation issues. The parallel instantiator DLV_{Gr}^{par} enhances the instantiator module of DLV [14] by introducing the three levels of parallelism and the heuristics described in the previous section. The system architecture is based on the repeated appliction of the traditional producer and consumers paradigm. Each level of parallelism receives tasks from the previous level (but the first one which receives its tasks directly from the main application process), and dispenses tasks to the subsequent one. In multi-threaded applications, the minimization of both mutual exclusive parts of code and lock contentions delays is fundamental for efficiency; also the number of threads used by the application should carefully managed, to both limit the costs of data structure initialization and avoid starvation problems. Our system makes use of a fixed number of threads. In particular, a certain number of threads is spawned at the beginning for each level of parallelism. Master threads (at each level) push tasks in the buffer of consumers. The dimension of the consumer's task buffer is important. Indeed, if it is not sufficiently large, there could be delays due to lock contention between consumer and producers, and also delays due to the fact that a producer has to search for a consumer buffer which is still not full. Clearly if the task buffer is too large, there will be a waste of computational resources. The optimum value for this buffer size depends on the machine at hand and, thus, it can be provided as an input parameter.

As the instantiation process is carried out in parallel, there will be the production of ground rules simultaneously; this may introduce lock contention on the output stream. To this aim we introduced output buffers (one for each thread) to store ground rules; when a given number of ground rules has been produced the output buffers are flushed. Also in this case, the output buffers size can be set by the user as an input parameter to be adapted for optimizing performance on the machine at hand.

3 System usage and options

In this Section we describe the usage of the grounder DLV_{Gr}^{par} and the available options. DLV_{Gr}^{par} can be invoked as follows:

```
./DLV^par_Gr  [-THC= <>] [-THR= <>]  [-THS= <>]
 [-FlushFactor= <>] [-TaskBufferSize= <>]
 [-SequentialJoinThreshold= <>] [-SequentialMatchThreshold= <>]
 [-redistributionThreshold= <>] [filename [filename [...]]]
```

Beside the standard DLV options, DLV_{Gr}^{par} introduces the ones that are relevant system parameters for optimizing the parallel instantiation process on the machine ad hand. In more details:

(i) "-THC","-THR", and "-THS" indicate the number of threads spawned for component parallelism, rule parallelism, and single rule parallelism, respectively. A rule of thumb for establishing these values is to keep the first two small and the third

significantly larger, but still at a reasonable rate (as too many threads may significantly slow down the computation). Default values are 8, 8, 256 suitable for a machine equipped with 8 cores.
(ii) "-FlushFactor" indicates to overll number of ground rules that can be stored in output buffers before printing them on standard output. The optimal value mostly depends on the number of processors available, default value is 100.
(iii) "-TaskBufferSize" is the maximum number of tasks that may be scheduled for the *component*, *rule*, and *single rule* level of parallelism at the same time. Default value is 8.
(iv) "-SequentialJoinThreshold" and "-SequentialMatchThreshold" are meant to tune the granularity control heuristics; the two values specify the thresholds used in the granularity control heuristic. Default values are respectively 100 and 10000.
(v) "-redistributionThreshold" is meant to tune the redistribution heuristics; the value specifies the number of tasks of single rule parallelism that may be scheduled for instantiating a rule, before applying the redistribution heuristics. Default value is 8.

System Availability. The system (32bit ELF executable) can be downloaded at `http://www.mat.unical.it/ricca/downloads/parallelground10.zip`.

4 Experimental analysis

We carried out an experimental analysis to assess the performance of the instantiator. We considered some well-known combinatorial problems which have been already used for assessing ASP instantiator performance [14, 3, 7], namely *N-Queens*, *Ramsey Number*, *Golomb Ruler*, *Max Clique* and *3Colorability*.[2] Note that these benchmarks are particulary difficult to parallelize because of the compactness of their encodings (a common property of ASP programs due to the declarative nature of the language). About intances, we considered five instance of increasing difficulty for each of the first 5 problem, whereas for 3Colorability we generated graphs having from 80 to 260 nodes. The machine used for the experiments is a two-processor Intel Xeon "Woodcrest" (quad core) 3GHz machine with 4MB of L2 Cache and 4GB of RAM, running Debian GNU Linux 4.0. In order to obtain more trustworthy results, each single experiment has been repeated five times and the average of performance measures are reported. In order to study the performance of the system when the number of available processor increases, the system was run on the selected benchmarks with 1,2,3,4,5,6,7, and 8 CPU enabled.[3]

The results of the analysis are summarized in Fig. 1. In particular, the table in Fig. 1a reports the instantiation times and standard deviation elapsed, as the number of enabled CPUs grows from 1 to 8, for solving the problem instances of all the considered benchmarks; whereas the average efficiency[4] for the same problems is reported in Fig. 1b.

[2] Encodings and instances can be downloaded at `http://www.mat.unical.it/ricca/downloads/parallelground10.zip`.

[3] The CPU n was disabled/enabled by running the linux command $echo\ 0/1 >> /sys/devices/system/cpu/cpu-n/online$.

[4] Efficiency is given by the run time of the sequential algorithm divided by the runtime of the parallel one times the number of processors.

	Average instantiation time (standard deviation)							
Problem	serial	2 proc	3 proc	4 proc	5 proc	6 proc	7 proc	8 proc
$queens_1$	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)
$queens_2$	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)
$queens_3$	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)
$queens_4$	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)
$queens_5$	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)
$ramsey_1$	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)
$ramsey_2$	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)
$ramsey_3$	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)
$ramsey_4$	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)
$ramsey_5$	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)
$clique_1$	16.06 (0.00)	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)
$clique_2$	29.98 (0.00)	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)
$clique_3$	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)
$clique_4$	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)
$clique_5$	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)
$golomb\_ruler_1$	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)
$golomb\_ruler_2$	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)
$golomb\_ruler_3$	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)
$golomb\_ruler_4$	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)
$golomb\_ruler_4$	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)
$3 - col_1$	8.61 (0.00)	4.41 (0.02)	3.02 (0.02)	2.29 (0.03)	1.86 (0.03)	1.59 (0.02)	1.38 (0.03)	1.26 (0.03)
$3 - col_2$	30.92 (0.00)	15.74 (0.27)	10.78 (0.19)	7.97 (0.02)	6.35 (0.02)	5.37 (0.03)	4.77 (0.20)	4.28 (0.30)
$3 - col_3$	78.64 (0.00)	40.25 (0.19)	26.66 (0.42)	20.31 (0.19)	16.24 (0.34)	13.44 (0.13)	11.56 (0.10)	10.36 (0.15)
$3 - col_4$	177.28 (0.00)	89.29 (0.65)	60.65 (0.58)	44.91 (0.24)	36.34 (0.26)	30.54 (0.51)	26.35 (0.06)	22.70 (0.37)
$3 - col_5$	347.97 (0.00)	178.98 (4.07)	123.22 (2.62)	90.09 (1.79)	71.84 (1.48)	61.42 (2.13)	53.32 (0.52)	45.88 (1.12)

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

	Efficiency						
Problem	2 proc	3 proc	4 proc	5 proc	6 proc	7 proc	8 proc
$queens$	0.98	0.97	0.96	0.94	0.92	0.92	0.91
$ramsey$	0.98	0.97	0.95	0.94	0.92	0.91	0.88
$clique$	0.94	0.93	0.91	0.90	0.89	0.87	0.86
$golomb\_ruler$	1.00	0.99	0.97	0.95	0.93	0.91	0.92
$3 - col$	0.98	0.96	0.97	0.96	0.95	0.94	0.93

(b) Mean efficiencies

Fig. 1: Impact of parallelization techniques

Looking at both the tables it is possible to see that the performance is nearly optimal, it slightly decreases as the number of processors increases, and rapidly increases when the difficulty of the input instance grows. The granularity control heuristics has been able to catch easy rules and perform their instantiation sequentially. The load balancing mechanism resulted to be effective and produced a well-balanced workload between CPUs.

Summarizing, the parallel instantiator behaved very well in all the considered instances, indeed 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 8 CPUs are available.

References

1. Gelfond, M., Lifschitz, V.: Classical Negation in Logic Programs and Disjunctive Databases. NGC **9** (1991) 365–385
2. Lifschitz, V.: Answer Set Planning. In: ICLP'99, Las Cruces, New Mexico, USA, The MIT Press (1999) 23–37
3. Gebser, M., Liu, L., Namasivayam, G., Neumann, A., Schaub, T., Truszczyński, M.: The first answer set programming system competition. In: LPNMR'07. LNCS 4483, (2007) 3–17
4. Balduccini, M., Gelfond, M., Watson, R., Nogeira, M.: The USA-Advisor: A Case Study in Answer Set Planning. In: LPNMR 2001 . Volume 2173., (2001) 439–442
5. Baral, C., Gelfond, M.: Reasoning Agents in Dynamic Domains. In: Logic-Based Artificial Intelligence. Kluwer (2000) 257–279
6. Baral, C., Uyan, C.: Declarative Specification and Solution of Combinatorial Auctions Using Logic Programming. In: LPNMR 2001. Volume 2173., (2001) 186–199
7. Denecker, M., Vennekens, J., Bond, S., Gebser, M., Truszczyński, M.: The second answer set programming competition. In: Logic Programming and Nonmonotonic Reasoning, 10th International Conference, LPNMR 2009, Potsdam, Germany, 14-18, 2009. Proceedings. LNCS 5753, (2009) 637–654
8. Leone, N., Gottlob, G., Rosati, R., Eiter, T., Faber, W., Fink, M., Greco, G., Ianni, G., Kałka, E., Lembo, D., Lenzerini, M., Lio, V., Nowicki, B., Ruzzi, M., Staniszkis, W., Terracina, G.: The INFOMIX System for Advanced Integration of Incomplete and Inconsistent Data. In: SIGMOD 2005, Baltimore, Maryland, USA, ACM Press (2005) 915–917
9. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. CUP (2003)
10. Grasso, G., Leone, N., Manna, M., Ricca, F.: Asp at work: Spin-off and applications of the DLV system. In: Logic Programming, Knowledge Representation, and Nonmonotonic Reasoning: Essays in Honor of M. Gelfond. (2011) 1–20
11. Marek, V.W., Truszczyński, M.: Stable Models and an Alternative Logic Programming Paradigm. In: The Logic Programming Paradigm – A 25-Year Perspective. (1999) 375–398
12. Gelfond, M., Leone, N.: Logic Programming and Knowledge Representation – the A-Prolog perspective . AI **138**(1–2) (2002) 3–38
13. Eiter, T., Faber, W., Leone, N., Pfeifer, G.: Declarative Problem-Solving Using the DLV System. In: Logic-Based Artificial Intelligence. Kluwer (2000) 79–103
14. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV System for Knowledge Representation and Reasoning. ACM TOCL **7**(3) (2006) 499–562
15. Calimeri, F., Perri, S., Ricca, F.: Experimenting with Parallelism for the Instantiation of ASP Programs. Journal of Algorithms in Cognition, Informatics and Logics **63**(1–3) (2008) 34–54
16. Perri, S., Ricca, F., Sirianni, M.: A parallel asp instantiator based on DLV. In: Proceedings of the 5th ACM SIGPLAN workshop on Declarative aspects of multicore programming. DAMP '10, New York, USA, ACM (2010) 73–82
17. Ullman, J.D.: Principles of Database and Knowledge-Base Systems, Volume I. Computer Science Press (1988)