Towards Distributed Computation of Answer Sets[⋆]

Marco De Bortoli, Federico Igne, Fabio Tardivo, Pietro Totis, Agostino Dovier, and Enrico Pontelli

Dept DMIF, University of Udine, Udine, Italy
Dept CS, New Mexico State University, Las Cruces, NM

Abstract. Answer Set Programming (ASP) is a logic programming language widely used in non monotonic automated reasoning. Thanks to its popularity, in the last years there has been a great interest towards the developing of efficient solvers, required to deal with complex problems, like planning and NP problem solving. These solvers can be unable to deal with programs that are "grounded" on huge amount of data, possibly resident in different sites. To address this problem, in this paper we present a distributed approach to ASP problems, which involves all the phases of the overall solving process: from a distributed grounder (which can also be used as a solver for stratified programs) to two different techniques to deal with the pure solving phase (for non-stratified program too), both of them using the non-standard graph coloring algorithm to characterize answer sets. We show three proposals for solving the issue, two of them developed with the high-level framework Apache Spark, while the the third one is a C++ direct implementation of the first one.

Keywords: Logic programming. ASP solving. Distributed computation.

1 Introduction

The Answer Set Programming (ASP) language has become very popular in the last years thanks to the availability of more and more efficient solvers (e.g., Clingo [10] and DLV [1]). It is based on the stable model semantics from Gelfond and Lifschitz [11], introduced to resemble the human reasoning process; together with its simple syntax, this make ASP a very intuitive language to be used. Like most logic languages, ASP solving process is split into two phases: the *grounding*, namely the transformation of the normal program in a so-called ground program, which is the equivalent propositional logic program where each rule is instantiated over the domain of its variables. The second phase consists in the real solving process, which alternates non deterministic guesses and

[⋆] The work is partially supported by the Uniud PRID ENCASE, by GNCS-INdAM, by two grants from University of Udine for "Tesi all'estero" and by Graz University of Technology.

deterministic propagation to find the solutions, starting from the ground program. As described, e.g., in [5], ASP has some important weakness when dealing with real world complex problems, like planning [8, 16], which generates huge ground programs. The grounding phase is in fact a strong limitation when dealing with problems which generates a great amount of rules, especially if it is a in-memory computation. This kind of programs leads to two issues, one regarding the grounding itself and one regarding the computation of its stable models, both limited by the amount of resource of the machine. Even if in literature there is a fair interest towards the parallelization of stable models computation, the single-machine multithreading applied to this field still has the memory limitation issue.

To address this problem, we present in this paper our working project in this direction, namely three distributed tools that exploit the shared resources of a distributed system to deal with such programs, thus overcoming the limitation of a single machine.

In chronological order, the first tool is a solver called mASPreduce [12] developed with the Apache distributed framework Spark, which uses the MapReduce paradigm to distribute the computation. The second one is a solver for stratified programs, still developed with Spark, which can be used also as a grounder, called STRASP. Finally, the last solver we present, namely DASC (Distributed Answer Set Coloring), makes use of the Coloring Algorithm on which mASPreduce is built. The Coloring Algorithm [13] is a non-standard technique for finding stable models in terms of different colorings of a graph built over the ASP program to solve. It was chosen because the graph is a data structure suitable to distribution. The difference is that now it is implemented using the boost library for C++ in order to lower the implementation level and to have more control on the communication stage between the nodes of the cluster.

The paper is organized as follows. In Section 2 we explain the Graph Coloring Algorithm for solving, implemented in different ways by both mASPreduce and DASC. From Section 3 to 5 we present the three tools. Some experimental results and comparison between them are reported in Section 6. The reader can find our conclusions in Section 7.

2 Graph Coloring

We briefly present the Coloring Algorithm for computation of answer sets [13], used by both mASPreduce and DASC solvers. In order to understand the following, we expect from the reader a basic knowledge about ASP syntax.

A *labeled graph* is a pair (G, ℓ) where $G = (V, E)$ is a directed graph and $\ell : E \to \mathcal{L}$ is a map from edges to a set of labels $\mathcal{L} = \{0, 1\}$ (intuitively 0 will represent a positive dependency and 1 a negative dependency). (G, ℓ) can be represented by the triple (V, E_0, E_1), where $E_i = \{e \in E \mid \ell(e) = i\}$ for $i = 0, 1$. Given a labeled graph $G = (V, E_0, E_1)$, an *i–subgraph* of G for $i = 0, 1$ is a subgraph of the graph $G_i = (V, E_i)$—i.e. a graph $G' = (W, F)$ s.t. $W \subseteq V$, and $F \subseteq E_i \cap (W^2)$. If $x, y \in V$, an *i–path* is a path from x to y in the graph G_i.

Let Π be a ground logic program; its *rule dependency graph* ($\mathcal{RDG}$) $\Gamma_\Pi = (\Pi, E_0, E_1)$ is the labeled graph where nodes are the (number of) program rules and

$$E_0 = \{(r, r') \mid r, r' \in \Pi, head(r) \in body^+(r')\}$$
$$E_1 = \{(r, r') \mid r, r' \in \Pi, head(r) \in body^-(r')\}$$

A *(partial/total) coloring* of Γ_Π is a partial/total map $C : \Pi \to \{\oplus, \ominus\}$, where $\oplus$ and $\ominus$ are two colors. We will denote $C_\oplus = \{r \mid r \in \Pi, C(r) = \oplus\}$ and $C_\ominus = \{r \mid r \in \Pi, C(r) = \ominus\}$, and a (partial) coloring as $(C_\oplus, C_\ominus)$. Let us define with $\mathbb{C}_\Pi$ the set of all (partial) colorings, and define a partial order over $\mathbb{C}_\Pi$ as follows: let C, C' be partial coloring of Γ_Π. We say that $C \sqsubseteq C'$ iff $C_\oplus \subseteq C'_\oplus$ and $C_\ominus \subseteq C'_\ominus$. The empty coloring $(\emptyset, \emptyset)$ is the *bottom* of the partial order $\mathbb{C}_\Pi$. Colors represent *enabling* ($\oplus$) and *disabling* ($\ominus$) of rules. Intuitively, we are interested in finding all possible subsets of *generating rules*, leading us to all the possible answer sets of a logic program.

Definition 1 (Generating Rules of an answer set). *Given a set of atoms X from a program Π, the set $\mathcal{R}_\Pi(X)$ of generating rules is given by*

$$\mathcal{R}_\Pi(X) = \{r \in \Pi \mid body^+(r) \subseteq X, body^-(r) \cap X = \emptyset\}.$$

Let Π be a logic program, and let Γ_Π be the corresponding $\mathcal{RDG}$. We define the notion of *admissible coloring* as follows: if $X \in AS(\Pi)$, then $C = (R_\Pi(X), \Pi \setminus R_\Pi(X))$ is an *admissible coloring* of Γ_Π (i.e., all the rules satisfied by X are colored positively, and the other rules negatively). Moreover, $head(C_\oplus) = X$ [13]. We denote by $AC(\Pi)$ the set of all admissible colorings of Γ_Π.

By definition, admissible colorings are total and one-to-one with answer sets. As shown in [13], for computing them we have to visit the space of partial colorings. Of course we are interested in partial colorings that will lead us to a total admissible coloring.

Let Π be a program and C a coloring of $\Gamma_\Pi = (\Pi, E_0, E_1)$. For $r \in \Pi$:

- r is *supported* in (Γ_Π, C), if $body^+(r) \subseteq \{head(r') \mid (r', r) \in E_0, r' \in C_\oplus\}$;
- r is *unsupported* in (Γ_Π, C), if there is $q \in body^+(r)$ s.t. $\{r' \mid (r', r) \in E_0, head(r') = q\} \subseteq C_\ominus$;
- r is *blocked* in (Γ_Π, C), if there exists $r' \in C_\oplus$ s.t. $(r', r) \in E_1$;
- r is *unblocked* in (Γ_Π, C), if $r' \in C_\ominus$ for all $(r', r) \in E_1$.

We also define the sets of supported $S(\Gamma, C)$, unsupported $\bar{S}(\Gamma, C)$, blocked $B(\Gamma, C)$, and unblocked $\bar{B}(\Gamma, C)$ rules. By definition, $S(\Gamma, C) \cap \bar{S}(\Gamma, C) = \emptyset$ and $B(\Gamma, C) \cap \bar{B}(\Gamma, C) = \emptyset$. With C total coloring, a rule is unsupported or unblocked iff it is not supported or blocked, respectively. This is not true, in general, for partial colorings.

The above defined notions can be used to define an operational semantics to compute the stable models of a logic program. We will only give an overview of the characterization implemented in our solver. For a deeper analysis of several other operational characterizations, we refer the reader to [13].

Let Γ be the $\mathcal{RDG}$ of a logic program Π and C be a partial coloring of Γ. The coloring operator $\mathcal{D}_\Gamma^\odot : \mathbb{C} \to \mathbb{C}$, where $\odot \in \{\oplus, \ominus\}$, is defined as follows:

1. $\mathcal{D}_\Gamma^\oplus = (C_\oplus \cup \{r\}, C_\ominus)$ for some $r \in S(\Gamma, C) \setminus (C_\oplus \cup C_\ominus)$;
2. $\mathcal{D}_\Gamma^\ominus = (C_\oplus, C_\ominus \cup \{r\})$ for some $r \in S(\Gamma, C) \setminus (C_\oplus \cup C_\ominus)$.

Operator $\mathcal{D}_\Gamma^\odot$ will be used to encode a branching path in the visit of the coloring tree, in fact, representing a non-deterministic choice (restricting our choice to the supported rules). Since *support* is a local property of a node (it only depends on information coming from the neighborhood), the coloring operator can be efficiently applied.

Let Γ be the $\mathcal{RDG}$ of a logic program Π and C be a (partial) coloring of Γ. Let us define the operators $\mathcal{P}_\Gamma, \mathcal{T}_\Gamma, \mathcal{V}_\Gamma : \mathbb{C} \to \mathbb{C}$ as follows

$$\begin{aligned}
&\mathcal{P}_\Gamma(C) = (C_\oplus \cup (S(\Gamma, C) \cap \bar{B}(\Gamma, C)), C_\ominus \cup (\bar{S}(\Gamma, C) \cup B(\Gamma, C)))\\
&\mathcal{T}_\Gamma(C) = (C_\oplus \cup (S(\Gamma, C) \setminus C_\ominus), C_\ominus)\\
&\mathcal{V}_\Gamma(C) = (C_\oplus, \Pi \setminus V)
\end{aligned}$$

where $V = \mathcal{T}_\Gamma^*(C_\oplus)$ and $\mathcal{T}_\Gamma^*(C)$ is the $\sqsubseteq$-smallest coloring containing C and closed under $\mathcal{T}_\Gamma$. A coloring C is closed under the operator *op* if $C = op(C)$. Finally, let $(\mathcal{PV})_\Gamma^*(C)$ be the $\sqsubseteq$-smallest coloring containing C and being closed under $\mathcal{P}_\Gamma$ and $\mathcal{V}_\Gamma$.

Theorem 21 (Operational Answer Set Characterization, III) *Let Γ be the $\mathcal{RDG}$ of a logic program Π and let C be a total coloring Γ. Then, C is an admissible coloring of Γ iff there exists a* coloring sequence *$C^0, C^1, \ldots, C^n$ such that: (1) $C^0 = (\mathcal{PV})_\Gamma^*((\emptyset, \emptyset))$, (2) $C^{i+1} = (\mathcal{PV})_\Gamma^*(\mathcal{D}_\Gamma^\odot(C^i))$ for some $\odot \in \{\oplus, \ominus\}$ and $0 \leq i < n$, (3) $C^n = C$.*

Given an admissible coloring C (point 3), *head*$(C_\oplus)$ returns its corresponding answer set. The proof of the above theorem can be found in [13] and a general introduction of so-called ASP computation is given in [14].

3 The mASPreduce solver

The mASPreduce solver is based on the MapReduce distributed programming paradigm first introduced in [6]. It is designed to analyze and process large data sets, and recent implementations of the model [7] are usually executed on clusters to take full advantage of the parallel nature of the architecture. A preliminary version was presented in [12].

A bare-bones implementation of MapReduce works on a generic collection of homogeneus data, and provides a basic interface consisting of two methods:

1. map$(\cdot)$ that maps a function over a collection of objects. It outputs a collection of "key-value" tuples;

2. `reduce`(·) that takes as input a collection of key-value pairs and merges the values of all entries with the same key. The merging operation is user-defined.

The user defines a MapReduce program as a sequence of map/reduce calls; modern implementations provide additional operators usually built on top of the primitives map/reduce (e.g., filters, fixpoints).

We used Apache Spark as a state-of-the-art in-memory MapReduce framework to implement the solver [17]. The framework relies on the concept of *Resilient Distributed Dataset*: a RDD is an immutable, fault-tolerant distributed collection of objects, a read-only, partitioned collection of records, organized into logical partitions, that may be located and processed on different nodes of the network. Spark abstracts from the underlying storage system, making it virtually compatible with any kind of filesystem. We built our system on top of the Hadoop Distributed File System (HDFS), since it is natively compatible with Spark.

As a further astraction over MapReduce, Spark offers two different families of high-level operators:

1. *transformations*, which create a new dataset from an existing one;
2. *actions*, which aggregate elements of a RDD with a custom function and return a single results to the caller.

In this case, `map` is a simple example of transformation because it executes a user-defined function on each node and returns a new RDD; `reduce` is an action that aggregates all the elements of the RDD using the provided function. Transformations are *lazily* executed, in that they do not compute their results until an action needs to be executed.

We implemented the rule dependency graph and the graph coloring algorithm for the computation of answer sets using the GraphX module [18] of Spark.

Apart from giving access to a complete distributed graph implementation, GraphX also provides a *Pregel API* [15]. Pregel is a *programming model* for large-scale graph problems and for fix-point computations over graphs. A typical Pregel computation consists of a sequence of *supersteps*. Within each superstep, vertices in a graph may interact with their neighbours sending messages. Each vertex analyzes the set of messages received in the previous superstep (if any) and alter its property according to a user-defined function; then, it can send new messages to the neighbourhood. A Pregel computation stops when a superstep does not generate any new message (or when other meta-conditions, such as a maximum number of iterations, are met).

The software is written in Scala and the $\mathcal{RDG}$ of a logic program is implemented as a subclass of `Graph[VD,ED]`, GraphX built-in class that gives access to the *property graph*. The original `Graph[VD,ED]` class is parametrized over vertex and edge property labels (`VD` and `ED` respectively). In the case of the `RDG` class, nodes keep track of rules and edges keep track of *support* and *blockage* relations between rules. `RDG` is in turn enclosed in a wrapper class which keeps track of the *atom table* (provided by the grounder) and the answer sets computed so far.

As described before, the $\mathcal{RDG}$ coloring process alternates between two phases:

1. *Non-deterministic coloring*: it is basically encoded as a visit over the colorings tree. Different heuristics will change the way we visit the tree, achieving better performances on different programs;
2. *Deterministic coloring*: deterministic propagation of the colors to avoid a blind visit of the colorings tree.

The main recursive procedure encodes the non-deterministic search and the back-propagation process to compute all possible answer sets. At any time, it goes through the vertices and (randomly) choose an uncolored *supported* rule. If none exists, we reached an admissible total coloring and we can build the corresponding answer set. In case an uncolored supported rule r exists, the computation branches. In the first case, r is colored with $\oplus$ and the color is deterministically propagated; finally, the function calls itself on the new graph. The second case is similar, but r is colored with $\ominus$.

Deterministic and non-deterministic operators are implemented as map/reduce routines, while fixpoint operators are implemented using Pregel.

4 Exploiting stratification

Stratified programs reflect the following intuition: at the point of the inference process where a rule is used, each negative reference should regard only atoms such that a complete information about their foundedness is available. This entails that a program is stratified if and only if it is possible to define an ordering on the evaluation of the rules such that whenever we encounter a *naf* literal its membership to the answer set is already determined.

The stable model is unique and can be obtained by iterating minimum fixpoint procedures at each strata [2]. The experiments presented in this section are aimed to test the effectiveness of the Apache Spark approach in absence of non determinism (hence, handling of backtracking).

[9, 4, 3] offer an overview of the way the computation of an answer set can be structured in order to exploit *SMP* (Symmetric Multi-Processing) architectures. Three levels of parallelism can be distinguished on a program $\mathcal{P}$, of which we implemented the first two: Component level parallelism, Rules level parallelism and Single Rule level parallelism. The reader can find more details in [4].

5 The DASC solver

The DASC solver has been developed with the purpose of improving the poor performance and scaling of mASPreduce, caused by the limitation of the high-level framework Spark (as witnessed by the results on stratified programs explained in the next section). To reach this goal, we opted for a C++ implementation, with the help of the Parallel Boost Graph Library (briefly, PBGL) for the distributed graph data structure, and the boost MPI library for the communication stage. Thanks to the latter, we have complete control over the messages sent on the network and the synchronization between the different computational nodes. Since the bad scaling of mASPreduce resides on the communication stage, our optimization starts from that.

The way PBGL distributes the graph is pretty straightforward: vertices are divided between the computational nodes in Round Robin way, stored in a node list, and each unit keeps track of the edges connected to its local vertices with adjacency lists. Vertex properties are stored in a property map.

5.1 Design choices

The first and most visible change with respect to mASPreduce is a modification of the $\mathcal{RDG}$ structure, which has two noticeable effects: it is more suitable to address the $notify\_change$ implementation of propagation, explained in the next subsection, and it potentially reduces the number of edges of an unbound number, at the cost of doubling up the nodes. From now on, we refer to such a graph as $\mathcal{RDG}$'.

Definition 2 (New Rule Dependency Graph: $\mathcal{RDG}$'). *Given a logic program Π we define the $\mathcal{RDG}$' Γ as the graph (V, E_0, E_1, E_2) where*

- $V = \Pi \cup atoms(\Pi)$
- $E_0 = \{(a, r) \mid a \in atoms(\Pi) \wedge r \in \Pi, a \in body^+(r)\}$
- $E_1 = \{(a, r) \mid a \in atoms(\Pi) \wedge r \in \Pi, a \in body^-(r)\}$
- $E_2 = \{(r, a) \mid r \in \Pi \wedge a \in atoms(\Pi), head(r) = a\}$

The reason why this graph is more suitable to our algorithm is that we rely only on information local to a node to decide whether the latter is supported or blocked.
For instance, a rule r is blocked if we are sure that in the actual coloring an atom a belonging to $body^-(r)$ does not belong to the answer set, i.e, for all rules r' such that $a = head(r')$, then $r' \in \mathcal{C}_\ominus$. To perform this check without forcing r to query all its neighbors, we could use a counter for each atom in $body^-(r)$ to keep count of how many r' were disabled. Since it is not a good idea to keep variable size data structures inside a node, we opted to use atom nodes, each one with its own single counter.

The other reason to choose this $\mathcal{RDG}$ structure is that it can strongly decrease the number of edges, which is a very good point in a distributed graph: fewer edges between different computational nodes, less amount of communication.

To address performance and reduce communication, a completely different strategy was developed in DASC to implement the propagation operators.

MapReduce paradigm has a big downside when dealing with a distributed system. Querying a neighbor stored in another computational node is a very expensive operation, and this situation always happens, even if the considered vertex would never be touched by the actual propagation. We refer to nodes connected to other computational units as border nodes; since MapReduce relies on the fact that each node queries all of its neighbors, this implies that also in the case of a local propagation, which theoretically does not need to send any message on the network, edges connected to border nodes are crossed, causing useless traffic inside the cluster.

To fix the problem, the idea is to develop an algorithm in which only the nodes really affected by the actual propagation (plus their neighbors) are touched: we will refer to this implementation as *notify_change* algorithm, since it will be duty of an affected node to notify its neighbors of an eventual change in its coloring state, and not the opposite. The reader can find the pseudocode in Figure 1.

```
void notify_change(node v) {
  //v is eventually colored by propagation operators
  if (v has just been colored) then
    notify_change(u) for all u such that there is a edge (v,u);
  else
    return;
}
```

Fig. 1. *notify_change* pseudocode

6 Testing

In Figure 2 the reader can find a quick comparison between DASC and mASPreduce. Each DASC test is executed with all possible combinations of distribution options.

STRASP solver supports both the maximal stratified subprogram and the fully stratified program resolution; moreover, we implemented a first naive form of distributed grounding, by exploiting the component and rule level parallelism in order to distribute among the available nodes the computation and resolution of the ground predicates.

STRASP performance as solver for stratified programs can be seen in Figure 3. We tested it by using the same logic (stratified) program, where the base stratum is composed of predicates such as `a(1..20).`. The upper strata define predicates with rules referring to the lower strata with combinations leading to an exponentially large grounding. Each test instance is obtained from the previous by increasing (+10%) the ranges of the base stratum. The comparison with Clingo shows that the two time series have similar trends, diverging only by a constant factor; unfortunately this constant is too large.

Although the shape of the graph of the running times with this approach is the same w.r.t. Clingo on the same stratified programs, the difference in terms of *constants* showed that the approach is still unfeasible. Efficiency is probably lost in the various abstraction levels that separates Apache users from the machine.

Since stratified programs can be fully evaluated at grounding time, the computation of the maximal stratified subprogram can be used as a starting point for a more general distributed grounding, by first grounding and solving the maximal stratified subprogram with the distributed procedure implemented in STRASP, and then considering the non deterministic part of the program.

Inst	Distr	1 cp unit		2 cp unit		3 cp unit		4 cp unit		5 cp unit	
		DASC	MR	DASC	MR	DASC	MR	DASC	MR	DASC	MR
1	RR	0.003	56.330	0.013	42.190	0.015	40.160	0.015	41.177	0.017	35.405
	RD	NR		0.010		0.011		0.013		0.014	
2	RR	0.048	95.697	0.14	64.315	0.19	61.767	0.16	62.845	0.19	54.144
	RD	NR		0.184		0.151		0.166		0.172	
3	RR	0.36	150.82	1.11	88.043	1.42	89.145	1.36	89.695	1.33	78.178
	RD	NR		1.226		1.393		1.115		1.232	
4	RR	1.83	SE	6.23	SE	6.18	SE	6.26	SE	5.98	SE
	RD	NR		6.118		6.401		6.226		5.256	
5	RR	7.03	to	22.84	to	23.86	to	33.60	to	24.21	to
	RD	NR		22.513		20.765		20.881		18.511	
6	RR	21.99	to	71.09	to	81.55	to	69.76	to	65.83	to
	RD	NR		71.07		83.60		66.43		68.20	
7	RR	58.90	to	188.85	to	185.45	to	212.69	to	220.73	to
	RD	NR		185.46		195.41		182.33		191.27	

Fig. 2. Comparison between DASC and mASPreduce (MR) on a set of benchmarks. For DASC, two distribution options are tested: round robin (RR) and heuristics based (RD). NR means "not relevant", SE "Spark Error", and "to" timeout. Tests from 1 to 5 computation units.

7 Conclusions and Future Work

Since this project started with mASPreduce, we made some steps forward in building a tool capable of exploiting distributed systems resources in order to manage huge size programs. Yet, we are still far from achieve the goal of large problems handling, and a lot of work has to be done to make our tools competitive with state-of-the-art solvers. Heuristics implementation and a variant of clause learning would probably be the main task to perform in order to closing the gap with them. At that point, DASC could be used to handle ground programs too big for single machine solvers.

In order to handle the overall solving process, STRASP is a good starting point for the grounding phase, even if performance are averagely 1000 time slower then Clingo in stratified programs. This gap is probably due to Spark inefficiency,

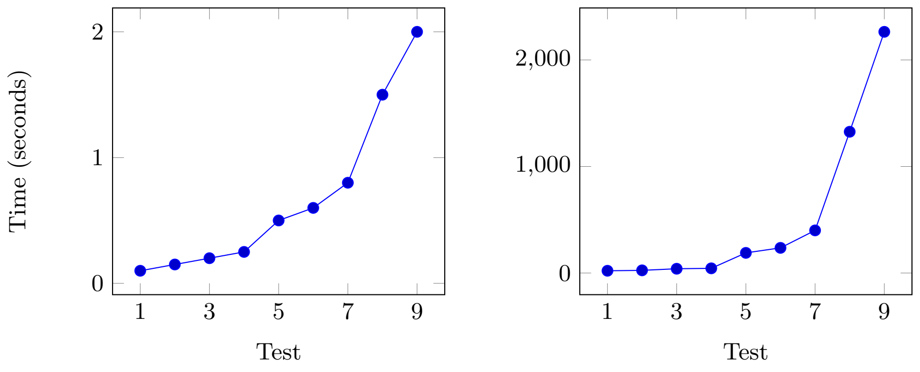

Fig. 3. Comparison between Clingo (left) and our Spark approach to stratified programs (right)

so we expect much better results with a direct implementation, how it happened between mASPreduce and DASC. However, STRASP performance can still be improved by implementing the last level (Single rule level) of parallelism, in order to handle huge programs.

References

1. W. T. Adrian, M. Alviano, F. Calimeri, B. Cuteri, C. Dodaro, W. Faber, D. Fuscà, N. Leone, M. Manna, S. Perri, F. Ricca, P. Veltri, and J. Zangari. The ASP system DLV: advancements and applications. *KI*, 32(2-3):177–179, 2018.
2. K. R. Apt, H. A. Blair, and A. Walker. Towards a theory of declarative knowledge. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 89–148. Morgan Kaufmann, 1988.
3. M. Balduccini, E. Pontelli, O. El-Khatib, and H. Le. Issues in parallel execution of non-monotonic reasoning systems. *Parallel Computing*, 31(6):608–647, 2005.
4. F. Calimeri, S. Perri, and F. Ricca. Experimenting with parallelism for the instantiation of ASP programs. *J. Algorithms*, 63(1-3):34–54, 2008.
5. A. Dal Palù , A. Dovier, E. Pontelli, and G. Rossi. GASP: answer set programming with lazy grounding. *Fundam. Inform.*, 96(3):297–322, 2009.
6. J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In E. A. Brewer and P. Chen, editors, *6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004*, pages 137–150. USENIX Association, 2004.
7. J. Dean and S. Ghemawat. *MapReduce: Simplified Data Processing on Large Clusters*, volume 51, pages 107–113. ACM, Jan. 2008.
8. A. Dovier, A. Formisano, and E. Pontelli. An empirical study of constraint logic programming and answer set programming solutions of combinatorial problems. *J. Exp. Theor. Artif. Intell.*, 21(2):79–121, 2009.
9. A. Dovier, A. Formisano, and E. Pontelli. Parallel answer set programming. In Y. Hamadi and L. Sais, editors, *Handbook of Parallel Constraint Reasoning.*, pages 237–282. Springer, 2018.

10. M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub. Clingo = ASP + control: Preliminary report. *CoRR*, abs/1405.3694, 2014.
11. M. Gelfond and V. Lifschitz. Logic programs with classical negation. In *Logic Programming, Proceedings of the Seventh International Conference, Jerusalem, Israel, June 18-20, 1990*, pages 579–597, 1990.
12. F. Igne, A. Dovier, and E. Pontelli. Masp-reduce: A proposal for distributed computation of stable models. In *Technical Communications of the 34th International Conference on Logic Programming, ICLP 2018, July 14-17, 2018, Oxford, United Kingdom*, volume 64 of *OASICS*, pages 8:1–8:4, 2018.
13. K. Konczak, T. Linke, and T. Schaub. Graphs and colorings for answer set programming. *TPLP*, 6(1-2):61–106, 2006.
14. L. Liu, E. Pontelli, T. C. Son, and M. Truszczynski. Logic programs with abstract constraint atoms: The role of computations. In V. Dahl and I. Niemelä, editors, *Logic Programming, 23rd International Conference, ICLP 2007, Porto, Portugal, September 8-13, 2007, Proceedings*, volume 4670 of *LNCS*, pages 286–301. Springer, 2007.
15. G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In S. Tirthapura and L. Alvisi, editors, *Proceedings of the 28th Annual ACM Symposium on Principles of Distributed Computing, PODC 2009, Calgary, Alberta, Canada, August 10-12, 2009*, page 6. ACM, 2009.
16. T. Son and E. Pontelli. Planning for biochemical pathways: A case study of answer set planning in large planning problem instances. In M. De Vos and T. Schaub, editors, *Proceedings of the First International SEA'07 Workshop, Tempe, Arizona, USA*, volume 281 of *CEUR Workshop Proceedings*, pages 116–130, 01 2007.
17. The Apache Software Foundation. Apache Spark (website), 2018. [last accessed Feb. 2018] https://spark.apache.org/.
18. The Apache Software Foundation. GraphX (website), 2018. [last accessed Feb. 2018] https://spark.apache.org/docs/latest/graphx-programming-guide.html.