Distributed Answer Set Coloring: Stable Models Computation via Graph Coloring

Marco De Bortoli
Institute for Software Technology, Graz University of Technology, Graz, Austria
Dept DMIF, University of Udine, Udine, Italy
Dept CS, New Mexico State University, Las Cruces, NM, USA
mbortoli@ist.tugraz.at

Answer Set Programming (ASP) is a famous logic language for knowledge representation, which has been really successful in the last years, as witnessed by the great interest into the development of efficient solvers for ASP. Yet, the great request of resources for certain types of problems, as the planning ones, still constitutes a big limitation for problem solving. Particularly, in the case the program is grounded before the resolving phase, an exponential blow up of the grounding can generate a huge ground file, infeasible for single machines with limited resources, thus preventing even the discovering of a single non-optimal solution. To address this problem, in this paper we present a distributed approach to ASP solving, exploiting distributed computation benefits in order to overcome the just explained limitations. The here presented tool, which is called Distributed Answer Set Coloring (DASC), is a pure solver based on the well-known Graph Coloring algorithm. DASC is part of a bigger project aiming to bring logic programming into a distributed system, started in 2017 by Federico Igne with mASPreduce and continued in 2018 by Pietro Totis with a distributed grounder. In this paper we present a low level implementation of the Graph Coloring algorithm, via the Boost and MPI libraries for C++. Finally, we provide a few results of the very first working version of our tool, at the moment without any strong optimization or heuristic.

I want to thank Fabio Tardivo, Agostino Dovier and Enrico Pontelli for their support during the development of the tool.

1 Introduction and problem description

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 [8] and DLV [1]). It is based on the stable model semantics from Gelfond and Lifschitz [9], introduced to resemble the human reasoning process; together with its simple syntax, this makes ASP a very intuitive language to be used. Like most logic languages, the ASP solving process is split into two phases: the *grounding*, namely the transformation of the normal program into 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 deterministic propagation to find the solutions, starting from the ground program. As described, e.g., in [3], ASP has some important weakness when dealing with real-world complex problems, like planning [5, 16], which generates huge ground programs. The grounding phase is in fact a strong limitation when dealing with problems that generate a great amount of rules, especially if it is an 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 resources of the machine. Even if in literature there is a fair interest towards the parallelization of stable models computation, the single-machine multithreading

B. Bogaerts, E. Erdem, P. Fodor, A. Formisano,
G. Ianni, D. Inclezan, G. Vidal, A. Villanueva,
M. De Vos, F. Yang (Eds.): International
Conference on Logic Programming 2019 (ICLP'19).
EPTCS 306, 2019, pp. 441–451, doi:10.4204/EPTCS.306.60

© Marco De Bortoli
This work is licensed under the
Creative Commons Attribution License.

applied to this field still has the memory limitation issue. To address this problem, we present in this paper a distributed ASP solver, called Distributed Answer Set Coloring (DASC), which automatically splits a ground program over a network, using the resources of a single computational node to process only a portion of the original program. To accomplish this, we use the Graph Coloring algorithm, which represents the program as a graph, and its stable models as different colorings of its vertices.
DASC is not the first attempt of this sort: it was born with the purpose of lowering the implementation level of the mASPreduce solver, a tool developed by Federico Igne for his Master thesis [10], in order to address its performance. mASPreduce is indeed developed with the distribution framework Spark from Apache, which, although it is a very powerful and expressive framework for distributed programming, gives to the user very low control over the communication flow, and it is the real performance killer during such kind of distributed computations.
We handled this communication control problem by developing DASC with C++, using the MPI library for messages handling and the Parallel Boost Graph Library to represent the distributed graph to color.
This research summary is organised as follows. In Section 2, we give some basic notations about ASP semantics and the Graph Coloring algorithm, and we see some related work in literature. Then we present the main details of our tool in Section 3. Some experimental results and comparison between DASC, mASPreduce and the state-of-the-art Clingo solver are reported in Section 4. The reader can find our conclusions and future work in Section 5.

2 Background and Related Work

2.1 Answer Set Programming

We provide a quick review of the basic concepts of *Answer Set Programming (ASP)*. We assume familiarity with logic programming.

A *normal rule* r is of the form

$$h \leftarrow a_1, \ldots, a_n, \texttt{not}\ b_1, \ldots, \texttt{not}\ b_m. \tag{1}$$

where $0 \leq n, m$, h is a positive literal, for $1 \leq i \leq n$, a_i is a positive literal, and for $1 \leq j \leq m$, $\texttt{not}\ b_j$ is a negative literal. A rule where $m = n = 0$ is called a *fact*, and a rule where $h = \bot$ (representing false) is called a *constraint*. A *normal logic program* Π is a finite set of normal rules and $\textsc{Atoms}(\Pi)$ is the set of all atoms of the alphabet occurring in Π. Given a normal rule r of a program Π, we define $head(r) = h$, $body^+(r) = \{a_1, \ldots, a_n\}$, $body^-(r) = \{b_1, \ldots, b_m\}$ and $body(r) = \{a_1, \ldots, a_n, \texttt{not}\ b_1, \ldots, \texttt{not}\ b_m\}$. Given a program Π, we can write $head(\Pi) = \{head(r) \mid r \in \Pi\}$ (similar for *body*, $body^+$ and $body^-$).

We call a program Π with $body^-(\Pi) = \emptyset$ and without constraints a *definite* logic program. A definite logic program always admits a minimum Herbrand model, denoted by $Cn(\Pi)$.

We say that a set X of atoms satisfies a rule r (in symbols $X \vDash r$) if $head(r) \in X$ whenever $body^+(r) \subseteq X$ and $body^-(r) \cap X = \emptyset$, and that X satisfies a program Π if for all $r \in \Pi$ $(X \vDash r)$.

Given a program Π and a set $X \subseteq \textsc{Atoms}(\Pi)$, we define the *reduct* Π^X of Π w.r.t. X as $\Pi^X = \{head(r) \leftarrow body^+(r) \mid r \in \Pi, body^-(r) \cap X = \emptyset\}$. X is an *answer set* of Π iff $Cn(\Pi^X) = X$. $AS(\Pi)$ denotes the set of all answer sets of a program Π.

The operator $Cn(\cdot)$ can also be characterized as the least fixpoint of the *immediate consequence operator* $T_\Pi(X) = \{head(r) \mid r \in \Pi, body^+(r) \subseteq X\}$. Iterated applications of T_Π can be defined as $T_\Pi^0(X) = X$ and $T_\Pi^{i+1}(X) = T_\Pi(T_\Pi^i(X))$, for $i \geq 1$. For a definite logic program Π, it can be proven that $Cn(\Pi) = \bigcup_{i \geq 0} T_\Pi^i(\emptyset)$. Therefore, X is an answer set of P iff $\bigcup_{i \geq 0} T_{\Pi^X}^i(\emptyset) = X$.

Definition 2.1 (Generating Rules of an answer set)**.** Given a set of atoms X from a program Π, the set $\mathscr{R}_\Pi(X)$ of generating rules is given by

$$R_\Pi(X) = \{r \in \Pi \,|\, body^+(r) \subseteq X, body^-(r) \cap X = \emptyset\}.$$

$R_\Pi(X)$ is uniquely identified by an answer set X. Given a program Π, a set $X \subseteq \textsc{Atoms}(\Pi)$ is an *answer set* of Π iff $Cn(R_\Pi(X)^\emptyset) = X$ [12]—let us observe that $R_\Pi(X)^\emptyset$ is the reduct of $R_\Pi(X)$ w.r.t. the empty set of atoms.

The semantics discussed above assumes that the atoms in the program are ground, i.e., they do not contain any variables. Intuitively, an atom/rule containing variables is a shorthand for the set of all possible ground instances obtained by consistently replacing each variable with all possible elements of the Herbrand universe. The process of rewriting a rule with variables into the equivalent set of rules without variables is referred to as *grounding*.

2.2 The Graph Coloring Algorithm

We briefly present the Coloring algorithm for the computation of answer sets [12], used by both mASPreduce and DASC solvers.

A *labeled graph* is a pair (G,ℓ) where $G = (V,E)$ is a directed graph and $\ell : E \to \mathscr{L}$ is a mapping from edges to the set $\mathscr{L} = \{0,1\}$ of labels (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 ($\mathscr{RDG}$)* $\Gamma_\Pi = (\Pi,E_0,E_1)$ is a labeled graph where nodes are the 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 mapping $C : \Pi \to \{\oplus,\ominus\}$, 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 $\mathbb{C}_\Pi$ be 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 sets of *generating rules*, leading us to all the possible answer sets of a logic program.

Let Π be a logic program, and let Γ_Π be the corresponding $\mathscr{RDG}$. 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 whose bodies are satisfied by X are colored positively, and the other rules negatively) such that $head(C_\oplus) = X$ [12]. We denote by $AC(\Pi)$ the set of all admissible colorings of Γ_Π.

By definition, admissible colorings are total and correspond one-to-one with answer sets. As shown in [12], 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 a 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 [12].

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

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

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

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

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

Theorem 2.1 (Operational Answer Set Characterization, III)**.** *Let* Γ *be the* $\mathscr{RDG}$ *of a logic program* Π *and let* C *be a total coloring of* 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)* $C^0 = (\mathscr{PV})_\Gamma^*((\emptyset,\emptyset))$*, (2)* $C^{i+1} = (\mathscr{PV})_\Gamma^*(\mathscr{D}_\Gamma^\odot(C^i))$ *for some* $\odot \in \{\oplus, \ominus\}$ *and* $0 \le i < n$*, (3)* $C^n = C$*.*

Given an admissible coloring C, $head(C_\oplus)$ returns its corresponding answer set. The proof of the above theorem can be found in [12] and a general introduction of so-called ASP computation is given in [13].

2.3 A survey on parallel solving

In this section we give a brief overview of existing parallelization techniques for ASP solving. For more details, we refer the reader to [6].

2.3.1 Parallel Grounding

We are going to start by presenting a multi-level parallel approach for grounding, made up of three phases: the Component Level Parallelism, the Rule Level Parallelism and the Single-Rule Level Parallelism [2].

In the Component Level Parallelism, the dependency graph of the ASP program is split into several partitions (or components), according to the Strongly Connected Components (SCC) of the graph.
Then, we can make a grounder work on each component separately, in order to process all of them in parallel.
In the second phase, the rules of each module are grounded in parallel, following some guidelines in order to not do invalidate the result. In fact, the rules are split into two groups, the exit rules and the recursive rules, and the former are instantiated first. For more details see the original paper [6].
Then, the third phase consists of splitting a rule body atom, selected by a heuristic procedure, and partitioning its extension between several threads.

2.3.2 Parallel Solving

Almost all the NP-problem solving techniques, including ASP solvers, are based on a search process. It can be seen as a tree (called search tree), in which each internal node represents a non-deterministic choice, each leaf is either a solution or a inconsistency, and the edges between different nodes are the deterministic propagation part which leads from a non-deterministic guess to the following one.
The general idea to parallelize this process is to assign to different threads (or computational nodes) the processing of the several subtrees generated along the search [14, 7].
However, this approach has two main problems:

- the knowledge/information collected till the moment a subtree T is split into more parts (for instance, a subtree for each of T's children) has to be replicated, possibly causing a memory blow up;
- to improve performance, solvers use heuristics which collect information from a specific branch in order to reuse it on another branches. This requires communication between the different parallel processes.

To address the latter, various techniques have been developed, divided into two groups: task sharing and scheduling. For more details, we refer the reader to the original paper [6].

2.3.3 Related Work

Our solver is part of a bigger project aiming to bring logic programming into a distributed system, started in 2017 by Federico Igne with mASPreduce [10, 11] and continued in 2018 by Pietro Totis with the distributed grounder STRASP [17].
Both of them are implemented in Scala, with the distribution framework Spark from Apache. The main difference between them lies in their purposes:

- mASPreduce is a pure solver, and it is a high level implementation of the Graph Coloring algorithm. Spark provides support for automatic distribution and parallelization via the MapReduce technique [4]: as a consequence, all the Graph Coloring operators are implemented as MapReduce routines. Although the operators seem pretty suitable to be encoded in this way, this leads to a network overhead, seriously affecting the performance;

- STRASP is a multi-purpose distributed tool: it can be used as a grounder for any kind of ASP programs, or as a solver for definite and stratified programs, like most of the grounders.
  It makes use of the two first levels of parallelism during the grounding phase: the Component Level Parallelism and the Rule Level Parallelism.
  Unfortunately, it suffers of the same performance issues as the previous system, because of the underlying framework.

In view of this, we chose for a different strategy: since communication handling is as important as the solving algorithm in a distributed environment, we developed from scratch a new implementation for the Graph Coloring algorithm, using low level languages and frameworks, such as C++, the Boost libraries and the MPI library for communication. This has led to a considerable performance improvement, as the reader can see in Section 4.

3 Distributed Answer Set Coloring

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. 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 a Round Robin way, stored in a node list, and each unit keeps track of the edges connected to its local vertices with adjacency lists. The user can choose between different data structures to implement both the node list and the adjacency lists (vector or list), and he can also set a property map, which we use to store vertex characteristics (color, supported, blocked, ...).

Anyway, this naive graph partitioning leads to poor communication performance, caused by the high size of the cut (with cut we mean the set of edges which connect two vertices stored in different computational units).
For this reason, we developed a greedy redistribution algorithm, which, for each pair of computational nodes, keeps swapping the two vertices that contribute more to the cut until a fixpoint is reached, i.e., until the swapping operation increases the cut size.

3.1 Design choices

The first and most visible change with respect to mASPreduce is a modification of the $\mathscr{RDG}$ structure, which has two noticeable effects: it is more suitable to address the *notify_change* implementation of propagation, explained later in this section, and it can considerably reduces the number of edges, at the cost of doubling up the nodes. From now on, we refer to such a graph as $\mathscr{RDG}$'.

Definition 3.1 (New Rule Dependency Graph: $\mathscr{RDG}$'). Given a logic program Π, we define the $\mathscr{RDG}$' Γ as the graph (V, E_0, E_1, E_2) where

- $V = \Pi \cup atoms(\Pi)$

- positive edges E_0: they have the same meaning they had in $\mathscr{RDG}$, but the source node must be an atom a and the destination node a rule r:

$$E_0 = \{(a,r) \mid a \in atoms(\Pi)\ ,\ r \in \Pi, a \in body^+(r)\}$$

- negative edges E_1: they have the same meaning they had in $\mathscr{RDG}$, but the source node must be an atom a and the destination node a rule r:

$$E_1 = \{(a,r) \mid a \in atoms(\Pi)\ ,\ r \in \Pi, a \in body^-(r)\}$$

- head edges E_2: they link each rule with its head (if it has any):

$$E_2 = \{(r,a) \mid r \in \Pi\ ,\ a \in atoms(\Pi), head(r) = a\}$$

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 unblocked 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 $head(r') = a$, then $r' \in \mathscr{C}_\ominus$. To perform this check without forcing r to query all its neighbors, we could use a counter for each atom in $body^-(r)$ to count 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 $\mathscr{RDG}$ structure is that it can strongly decrease the number of edges, which is a very good point in a distributed graph: the fewer edges between different computational nodes, the less amount of communication.
To explain this property, the reader can imagine a logic program with n rules with the same head a, which in turn is present in the body of m rules. To represent a's dependencies, an $\mathscr{RDG}$ would need $n*m$ edges (from each of n rules to each of m rules), while an $\mathscr{RDG}$' can instead obtain the same result with only $n+m$ edges (from each of n rules to the atom node a, and from the atom node a to each of m rules).

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

The 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. Looking at the example in Figure 1, 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 (like one involving only r_1 and r_3), 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.

4 Preliminary results

In Figure 3 the reader can find a quick comparison between DASC and mASPreduce. We tested a toy example (Figure 2) in which, by only changing the domain size of the problem, it is easy to generate

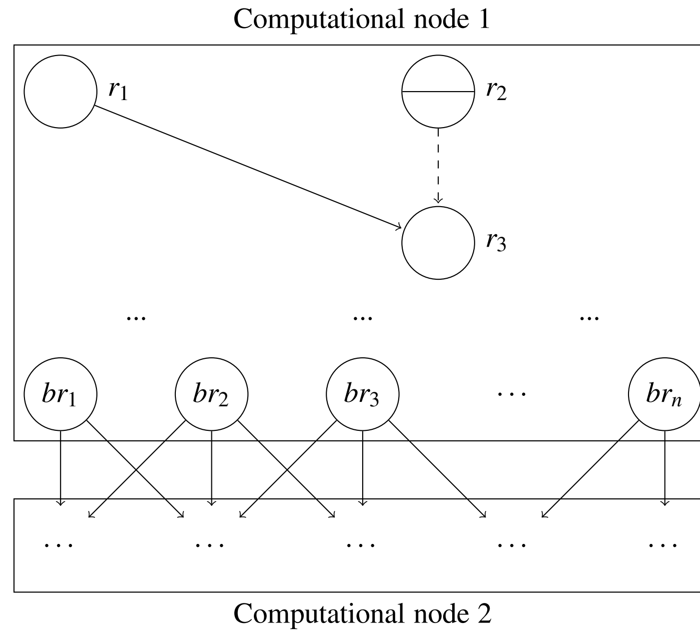

Figure 1: This figure represents the main issue of the MapReduce paradigm: during a propagation involving only nodes owned by a single machine (like r_1, r_2, r_3), the border nodes br_i do query their neighbors (owned by computational node 2), generating useless communication.

```
dom(1..n).

sel(X) :- dom(X), not nsel(X).

nsel(X) :- dom(X), not sel(X).

:- sel(X), sel(Y), X != Y.

p(X1,X2,X3,X4,X5,X6) :-
   sel(X1), sel(X2), sel(X3), sel(X4), sel(X5), sel(X6).
```

Figure 2: Toy example: by increasing *n* inside *dom(1..n)*, the number of generated ground roules grows exponentially

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	

Figure 3: Comparison between DASC and mASPreduce (MR) on a set of benchmarks. For DASC, two distribution options are tested: round robin (RR) and greedy redistribution (RD). NR means "not relevant" (we can not test a redistribution modality with 1 cp unit), SE "Spark Error", and "to" timeout. Tests involve from 1 to 5 computation units.

ground programs with a high number of ground rules: since our program does not (yet) make use of any heuristics, and we are more interested in how it deals with a huge number of nodes to distribute, this behaviour is useful to test how well a problem scales over both the number of computational nodes and the number of vertices in the RDG.
Each DASC test is executed with all possible combinations of distribution options:

- Distribution algorithm: it could be either a naive round robin distribution (standard behaviour of *boost*) or a greedy distribution algorithm which tries to minimize cut size;
- Number of computational nodes: each problem is tested using different numbers of computational units of the cluster, from 1 to 5.

We omit the Clingo (single thread) tables since almost all timings were at most 10ms.

It is clear that there is a huge performance gap between both mASPreduce with DASC, and the latter with Clingo.
Moreover, also DASC seems to do not scale very well, how the reader can notice by the fact that the tests with only one computational node are the ones which behaviour better in all the cases. The reason for such a bad scaling lies on the communication phase: apparently, the improvement for having a propagation process parallelized between more nodes is not enough to compensate the performance degradation caused by the communication between those nodes. There are other reasons to still use this distributed approach, like the fact we can theoretically handle programs too big to be contained into a single machine. Yet for the moment, such problems are too complicated to be solved by our tool within acceptable timings, but this will change in the future, through the implementation of various heuristics.
Moreover, a big part of the guilt of this behaviour is due to the initial round robin distribution from which the redistribution algorithm starts. In fact, we noticed a very few iterations were made during the graph repartition, resulting in a node distribution too similar to the initial one. We stongly believe that, by

addressing this problem, we would obtain a far better scaling.

5 Conclusion and Future Work

DASC represents a step forward in building a tool capable of exploiting distributed system resources in order to manage huge-size programs, thanks also to the fact that it is capable of solving domains that do not work properly with mASPreduce, like the ones from the 2015 ASP competition. Yet, we are still far from achieving the goal of large problems handling, and a lot of work has to be done to make our tool competitive with state-of-the-art solvers. Heuristics implementation would probably be the main task to perform in order to close the gap with them. At that point, DASC could be used to handle programs too big for single machine solvers.
Moreover, it is evident from the performance difference between our tool and mASPreduce that lowering the level of implementation paid off, together with the development of a different propagation technique, the so-called *notify_change* approach.
To confirm C++ boost improvement with respect to SPARK, in the instances in which Clingo exceeds 10ms, the minimum machine time, DASC is about 500 times slower; STRASP instead, the distributed grounder developed by Pietro Totis in his thesis using SPARK [17], capable of solving stratified programs (non-definite programs solvable without non-determinism in polynomial time), is circa 2000 times slower than Clingo.

Finally, we present below the roadmap for DASC:

- improving the initial distribution or changing the redistribution algorithm with a more sophisticated one;
- implementing multithreading. We identified two kinds of parallelization we can apply, and the program is already prepared for the first one:
  - task parallelization: in the actual state, each time a node receives a message with a distribution task to execute, this is stored inside a stack. Since we noticed during the testing that this stack can reach really huge size, a good idea would be to process these tasks in parallel, by distributing them between more threads. Of course this kind of parallelization would work only in a distributed system;
  - local parallelization: to exploit multithreading also when working in a local machine we can do the following: each time we need to propagate some information from a vertex with many outgoing edges, we can cross each edge with a different thread, in order to parallelize the various propagation branches.

  Obviously multithreading introduces some issues, like the fact we need to avoid that more threads/propagation branches read or write from the same vertex at the same time, because one of them could retrieve invalid information, but Boost provides us all the tools to address this problem, like mutex access;
- implementing heuristics, like the strong techniques used by Clingo, namely clause learning and backjumping.

References

[1] Weronika T. Adrian, Mario Alviano, Francesco Calimeri, Bernardo Cuteri, Carmine Dodaro, Wolfgang Faber, Davide Fuscà, Nicola Leone, Marco Manna, Simona Perri, Francesco Ricca, Pierfrancesco Veltri & Jessica Zangari (2018): *The ASP System DLV: Advancements and Applications*. *KI* 32(2-3), pp. 177–179, doi:10.1007/s13218-018-0533-0.

[2] Francesco Calimeri, Simona Perri & Francesco Ricca (2008): *Experimenting with parallelism for the instantiation of ASP programs*. *Journal of Algorithms* 63(1-3), pp. 34–54, doi:10.1016/j.jalgor.2008.02.003.

[3] Alessandro Dal Palù , Agostino Dovier, Enrico Pontelli & Gianfranco Rossi (2009): *GASP: Answer Set Programming with Lazy Grounding*. *Fundam. Inform.* 96(3), pp. 297–322, doi:10.3233/FI-2009-180.

[4] J. Dean & S. Ghemawat (2008): *MapReduce: Simplified Data Processing on Large Clusters*, pp. 107–113. 51, ACM.

[5] Agostino Dovier, Andrea Formisano & Enrico Pontelli (2009): *An empirical study of constraint logic programming and answer set programming solutions of combinatorial problems*. *JETAI* 21(2), pp. 79–121, doi:10.1080/09528130701538174.

[6] Agostino Dovier, Andrea Formisano & Enrico Pontelli (2018): *Parallel Answer Set Programming*. In Youssef Hamadi & Lakhdar Sais, editors: *Handbook of Parallel Constraint Reasoning.*, Springer, pp. 237–282, doi:10.1007/978-3-319-63516-3_7.

[7] Raphael A. Finkel, Victor W. Marek, Neil Moore & Miroslaw Truszczynski (2001): *Computing stable models in parallel*. In Provetti & Son [15]. Available at http://www.cs.nmsu.edu/%7Etson/ASP2001/18.ps.

[8] Martin Gebser, Roland Kaminski, Benjamin Kaufmann & Torsten Schaub (2014): *Clingo = ASP + Control: Preliminary Report*. *CoRR* abs/1405.3694. Available at http://arxiv.org/abs/1405.3694.

[9] M. Gelfond & V. Lifschitz (1990): *Logic Programs with Classical Negation*. In: *Logic Programming, Proceedings of the Seventh International Conference, Jerusalem, Israel, June 18-20, 1990*, pp. 579–597.

[10] Federico Igne (2017): *Analysis and development of a distributed ASP solver using MapReduce*. Master's thesis, University of Udine.

[11] Federico Igne, Agostino Dovier & Enrico Pontelli (2018): *MASP-Reduce: A Proposal for Distributed Computation of Stable Models*. In Alessandro Dal Palù, Paul Tarau, Neda Saeedloei & Paul Fodor, editors: *Technical Communications of the 34th International Conference on Logic Programming, ICLP 2018, July 14-17, 2018, Oxford, United Kingdom*, *OASICS* 64, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, pp. 8:1–8:4, doi:10.4230/OASIcs.ICLP.2018.8.

[12] Kathrin Konczak, Thomas Linke & Torsten Schaub (2006): *Graphs and colorings for answer set programming*. *TPLP* 6(1-2), pp. 61–106, doi:10.1017/S1471068405002528.

[13] Lengning Liu, Enrico Pontelli, Tran Cao Son & Miroslaw Truszczynski (2007): *Logic Programs with Abstract Constraint Atoms: The Role of Computations*. In Verónica Dahl & Ilkka Niemelä, editors: *Logic Programming, 23rd International Conference, ICLP 2007, Porto, Portugal, September 8-13, 2007, Proceedings*, *LNCS* 4670, Springer, pp. 286–301, doi:10.1145/321978.321991.

[14] Enrico Pontelli & Omar El-Khatib (2001): *Exploiting Vertical Parallelism from Answer Set Programs*. In Provetti & Son [15]. Available at http://www.cs.nmsu.edu/%7Etson/ASP2001/24.ps.

[15] Alessandro Provetti & Tran Cao Son, editors (2001): *Answer Set Programming, Towards Efficient and Scalable Knowledge Representation and Reasoning, Proceedings of the 1st Intl. ASP'01 Workshop, Stanford, CA, USA, March 26-28, 2001*.

[16] Tran Cao Son & Enrico Pontelli (2007): *Planning for biochemical pathways: A case study of answer set planning in large planning problem instances*. In Marina De Vos & Torsten Schaub, editors: *Proceedings of the First International SEA'07 Workshop, Tempe, Arizona, USA*, *CEUR Workshop Proceedings* 281, pp. 116–130.

[17] Pietro Totis (2018): *A distributed ASP solver for stratified programs*. Master's thesis, University of Udine.