PLATYPUS: A platform for distributed answer set solving

Jean Gressmann[1], Tomi Janhunen[2], Robert E. Mercer[3], Torsten Schaub[1⋆], Sven Thiele[1], and Richard Tichy[1,3]

[1] Institut für Informatik, Universität Potsdam, Postfach 900327, D-14439 Potsdam, Germany
[2] Helsinki University of Technology, Department of Computer Science and Engineering, Laboratory for Theoretical Computer Science, P.O. Box 5400, FI-02015 TKK, Finland
[3] Computer Science Department, Middlesex College, The University of Western Ontario, London, Ontario, Canada N6A 5B7

Abstract. We propose a model to manage the distributed computation of answer sets within a general framework. This design incorporates a variety of software and hardware architectures and allows its easy use with a diverse cadre of computational elements. Starting from a generic algorithmic scheme, we develop a platform for distributed answer set computation, describe its current state of implementation, and give some experimental results.

1 Introduction

The success of Answer Set Programming (ASP) has been greatly boosted by the availability of highly efficient ASP solvers [1, 2]. However, its expanding range of application creates an increasing demand for more powerful computational devices. We address this by proposing a generic approach to distributed answer set solving that permits exploitation of the increasing availability of clustered and/or multi-processor machines.

We observe that the search strategies of most current answer set solvers naturally decompose into a deterministic and a non-deterministic part, borrowing from the well-known DPLL satisfiability checking algorithm [3]. While the non-deterministic part is usually realized through heuristically driven *choice* operations, the deterministic one is normally based on advanced *propagation* operations, often amounting to the computation of Fitting's [4] or well-founded semantics [5]. Roughly, the idea is: starting with an empty (partial) assignment of truth values to atoms, successively apply propagation and choice operations, gradually extending a partial assignment, until finally a total assignment, expressing an answer set, is obtained. The overall approach is made precise in Algorithm 1, which closely follows `smodels` [1].[4,5] When called with SMODELS$((\emptyset, \emptyset))$, it computes all answer sets of a logic program via backtracking. A partial assignment is represented as a pair (X, Y) of sets of atoms, in which X and Y contain those atoms assigned true and false, respectively. Informally, propagation is done with the EXPAND function (in Line 1); choices are done with CHOOSE (in Line 4).

⋆ Affiliated with the School of Computing Science at Simon Fraser University, Burnaby, Canada.
[4] `dlv` works analogously yet tuned to disjunctive programs.
[5] We use `typewriter` font when referring to actual systems.

The first **if**-statement accounts for invalid assignments indicating an inconsistency (in Line 2), and the second, for the case of a total assignment representing an answer set (in Line 3). Otherwise, a case-analysis is performed on the chosen atom,[6] assuming it to be true in Line 5 and false in Line 6, respectively.

Algorithm 1: SMODELS

```
Global  : A logic program $\Pi$ over alphabet $\mathcal{A}$.
Input   : A partial assignment $(X, Y)$.
Output  : Print all answer sets of $\Pi \cup \{\leftarrow not\ A \mid A \in X\} \cup \{\leftarrow A \mid A \in Y\}$.
begin
1   $(X', Y') \leftarrow$ EXPAND($(X, Y)$)
2   if $X' \cap Y' \neq \emptyset$ then return
3   if $X' \cup Y' = \mathcal{A}$ then
        print $X'$
        return
4   $A \leftarrow$ CHOOSE($\mathcal{A} \setminus (X' \cup Y')$)
5   SMODELS($(X' \cup \{A\}, Y')$)
6   SMODELS($(X', Y' \cup \{A\})$)
end
```

Our approach takes advantage of this idea by relying on an encapsulated module for propagation. For sake of comparability, this is currently embodied by `smodels`' expansion procedure. Unlike `smodels`, however, we are interested in distributing parts of the search space, as invoked by the two recursive calls in Algorithm 1. To this end, we propose a general approach, based on pioneering work in distributed tree search, that accommodates a variety of different architectures for distributing the search for answer sets over different processes and processors, respectively. Distributed tree search in ASP solvers [6–8] has been significantly influenced by the general-purpose backtracking package, DIB [9], the culmination of a decade of research in distributed tree search. Also, much work has been carried out in the area of parallel logic programming, among which our work is particularly analogous to or-parallelism; see [10, 11] for surveys of this field. However, an important difference is that concurrent prolog implementations seek to parallelize query evaluation, whereas our goal is to distribute the search for answer sets. The latter is more closely related to distributed satisfiability checking (see e.g. [12, 13]), although differing in the sense that it typically suffices for satisfiability checking to find only one satisfying assignment. An early attempt to compute answer sets in parallel was made in [14] by using the model generation theorem prover MGTP.

We start by developing an iterative enhancement of Algorithm 1 that is based on an explicit representation of the search space. Whenever the system environment allows us to delegate a part of this search space, it may be transferred to another computational device. Although our early efforts have focused on `smodels` as the computing engine, we differ from [7] and [8] in that their design philosophy is to build distributed versions

[6] In fact, `smodels` chooses a literal, thereby dynamically deciding the order between the two calls in Line 5 and Line 6.

of `smodels`, whereas our approach (1) modularizes (and is thus independent of) the propagation engine, and (2) incorporates a flexible distribution scheme, accommodating different distribution policies and distribution architectures, for instance. Regarding the latter, the current system supports a multiple process (by forking) and a multiple processor (by MPI [15]) architecture. A multi-threaded variant is currently under development. The multi-process and multi-threaded architectures can also be run on a multi-processor environment to achieve real speed-ups (compared to a one-processor environment).

2 Definitions and notation

A *logic program* is a finite set of rules of the form

$$p_0 \leftarrow p_1, \ldots, p_m, \mathit{not}\ p_{m+1}, \ldots, \mathit{not}\ p_n \ , \tag{1}$$

where $n \geq m \geq 0$, and each p_i $(0 \leq i \leq n)$ is an *atom* in some alphabet $\mathcal{A}$. Given a rule r as in (1), we let $\mathit{head}(r)$ denote the *head* (set), $\{p_0\}$, of r and $\mathit{body}^+(r) = \{p_1, \ldots, p_m\}$ and $\mathit{body}^-(r) = \{p_{m+1}, \ldots, p_n\}$, the sets of positive and negative *body* literals, respectively. Also, we allow for *integrity constraints*, where $\mathit{head}(r) = \emptyset$. The *reduct*, Π^X, of a program Π *relative to* a set X of atoms is defined as

$$\Pi^X = \{\mathit{head}(r) \leftarrow \mathit{body}^+(r) \mid r \in \Pi, \mathit{body}^-(r) \cap X = \emptyset\} \ .$$

Then,[7] a set X of atoms is an *answer set* of a program Π if X is a $\subseteq$–minimal model of Π^X. We use $AS(\Pi)$ for denoting the set of all answer sets of a program Π.

As an example, consider program Π, consisting of rules

$$\begin{array}{lll} p \leftarrow \mathit{not}\ q, & r \leftarrow p, & s \leftarrow r, \mathit{not}\ t, \\ q \leftarrow \mathit{not}\ p, & r \leftarrow q, & t \leftarrow r, \mathit{not}\ s. \end{array} \tag{2}$$

Program Π has 4 answer sets, $\{p, r, s\}$, $\{p, r, t\}$, $\{q, r, s\}$, and $\{q, r, t\}$. Adding integrity constraint $\leftarrow q, r$ eliminates the two last sets.

For computing answer sets, we rely on *partial assignments*, mapping atoms in $\mathcal{A}$ onto true, false, or undefined. We represent such assignments as pairs (X, Y) of sets of atoms, in which X contains all true atoms and Y all false ones. An answer set X is then represented by the total assignment $(X, \mathcal{A} \setminus X)$. In general, a partial assignment (X, Y) aims at capturing a subset of the answer sets of a program Π, viz.

$$AS_{(X,Y)}(\Pi) = \{Z \in AS(\Pi) \mid X \subseteq Z, Z \cap Y \neq \emptyset\} \ .$$

3 The PLATYPUS approach

A key observation leading to our approach is that once the program along with its alphabet is fixed, the outcome of Algorithm 1 depends only on the partial assignment

[7] We use this definition since it easily includes integrity constraints. Note that any integrity constraint $\leftarrow \mathit{body}(r)$ can be expressed as $x \leftarrow \mathit{body}(r), \mathit{not}\ x$ by using a new atom x.

given as **Input**. As made precise in the **Output** field, the resulting answer sets are then uniquely determined. Moreover, partial assignments provide a straightforward way for partitioning the search space, as witnessed by Lines 5 and 6 in Algorithm 1. In fact, incompatible partial assignments represent different parts of the search space.

For illustration, let us compute the answer sets of Program Π, given in (2). Starting with SMODELS$((\emptyset, \emptyset))$ forces us to choose immediately among the undefined atoms in $\{p, q, r, s, t\}$ because $(\emptyset, \emptyset)$ cannot be extended by EXPAND. Choosing p makes us call SMODELS first with $(\{p\}, \emptyset)$ and then with $(\emptyset, \{p\})$. This case-analysis partitions the search space into two subspaces, the one containing all assignments making p true and the other with all assignments making p false. Following up the first call with $(\{p\}, \emptyset)$, the latter gets expanded to $(\{p, r\}, \{q\})$ before a choice must be made among $\{s, t\}$. Again, the search space becomes partitioned, and we obtain two total assignments, $(\{p, r, s\}, \{q, t\})$ and $(\{p, r, t\}, \{q, s\})$, whose first components are printed as answer sets. The two remaining answer sets are obtained analogously, but with the roles of p and q interchanged.

For distributing the computation of answer sets, the idea is to decompose the search space by means of partial assignments. For instance, instead of invoking a single process via SMODELS$((\emptyset, \emptyset))$, we may initiate two independent ones by calling SMODELS on $(\{p\}, \emptyset)$ and $(\emptyset, \{p\})$, possibly even on different machines. Although this static distribution of the search space results in a fair division of labor, such a balance is hardly achievable in general. To see this, consider the choice of r instead of p, resulting in calling SMODELS with $(\{r\}, \emptyset)$ and $(\emptyset, \{r\})$. While the former process gets to compute all 4 answer sets, the latter terminates almost immediately, since the EXPAND function yields an invalid assignment.[8] In such a case a dynamic redistribution of the search space is clearly advantageous. That is, once the second process is terminated, the first one may delegate some of its remaining tasks to the second one.

To this end, we propose a general approach that accommodates a variety of different modes for distributing the search for answer sets over different processes and/or processors. We start by developing an iterative enhancement of Algorithm 1 that is based on an explicit representation of the search space in terms of partial assignments. Algorithm 2 gives our generic PLATYPUS[9] algorithm for distributed answer set solving. A principal goal in its design is to allow for as much *generality* as possible. Specific instances contain trade-offs, for example, arbitrary access to the search space versus compact spatial representations of it. Another major design goal is *minimal communication* in terms of message size. To this end, PLATYPUS relies on the omnipresence of the given logic program Π along with its alphabet $\mathcal{A}$ as global parameters. Communication is limited to passing partial assignments as representatives of parts of the search space.

The only input variable S delineates the initial search space given to a specific instance of PLATYPUS. S is thus a set of partial assignments over alphabet $\mathcal{A}$. Although this explicit representation offers an extremely flexible access to the search space, it must be handled with care since it grows exponentially in the worst case. Without

[8] This nicely illustrates that the choice of the branching atom is more crucial in a distributed setting.

[9] *platypus*, small densely furred aquatic monotreme of Australia and Tasmania having a broad bill and tail and webbed feet.

Algorithm 2: PLATYPUS

```
Global : A logic program $\Pi$ over alphabet $\mathcal{A}$.
Input  : A nonempty set $S$ of partial assignments.
Output : Print a subset of the answer sets of $\Pi$ (cf. (3)).

repeat
1   $(X, Y) \leftarrow$ CHOOSE($S$)
2   $S \leftarrow S \setminus \{(X, Y)\}$
3   $(X', Y') \leftarrow$ EXPAND($(X, Y)$)
4   if $X' \cap Y' = \emptyset$ then
5       if $X' \cup Y' = \mathcal{A}$ then
6           print $X'$
        else
7           $A \leftarrow$ CHOOSE($\mathcal{A} \setminus (X' \cup Y')$)
8           $S \leftarrow S \cup \{\ (X' \cup \{A\}, Y'),\ (X', Y' \cup \{A\})\ \}$
9       $S \leftarrow$ DELEGATE($S$)
until $S = \emptyset$
```

Line 9, Algorithm 2 computes all answer sets in $\bigcup_{(X,Y)\in S} AS_{(X,Y)}(\Pi)$, or equivalently,

$$\bigcup_{(X,Y)\in S} AS(\Pi \cup \{\leftarrow not\ A \mid A \in X\} \cup \{\leftarrow A \mid A \in Y\})\ . \tag{3}$$

With Line 9, a subset of this set of answer sets is finally obtained (from a specific PLATYPUS instance). Clearly, depending on which parts of the search space are removed by delegation (see below), this algorithm is subject to incomplete and redundant search behaviour, unless an appropriate delegation strategy is used.

A PLATYPUS instance iterates until its local search space has been processed. Before detailing the loop's body, let us fix the formal behavior of the functions and procedures used by PLATYPUS (in order of appearance).

CHOOSE: Given a set[10] X, CHOOSE(X) gives some $x \in X$.

EXPAND: Given a partial assignment (X, Y), EXPAND$((X, Y))$ computes a partial assignment (X', Y') such that

1. $X \subseteq X'$ and $Y \subseteq Y'$,
2. $AS_{(X',Y')}(\Pi) = AS_{(X,Y)}(\Pi)$,
3. if $X' \cap Y' = \emptyset$ and $X' \cup Y' = \mathcal{A}$, then $AS_{(X',Y')}(\Pi) = \{X'\}$,

and furthermore (X', Y') can be closed under propagation principles such as those based on well-founded semantics and contraposition [1].[11]

DELEGATE: Given a set X, DELEGATE(X) returns a subset $X' \subseteq X$.

Both functions CHOOSE and DELEGATE are in principle non-deterministic selection functions. As usual, CHOOSE is confined to a single element, whereas DELEGATE

[10] The elements of X are arbitrary in view of Lines 1 and 7.

[11] In practice, propagation may even go beyond well-founded semantics, as for instance with `smodels`'s lookahead.

selects an entire subset. In sum, PLATYPUS adds two additional sources of non-determinism. While the one in Line 1 is basically a "*don't care*" choice, the one in Line 9 must be handled with care since it removes assignments from the local search space. The EXPAND function hosts the deterministic part of Algorithm 2; it is meant to be accomplished by an off-the-shelf system that is used as a black-box providing both sufficiently firm as well as efficient propagation operations which aim to reduce the remaining local search space resulting from choice operations.

DELEGATE permits some answer set computation tasks embodied in S to be assigned to other processes and/or processors. The assignments returned in Line 9 have not been delegated and thus remain in S. The removed assignments are either dealt with by other PLATYPUS instances or even algorithms other than PLATYPUS. The elimination of search space constituents is a delicate operation insofar as we may lose completeness or termination. For example, an implementation of DELEGATE that does not reassign all removed constituents is incomplete. Accordingly, passing certain constituents around forever would lead to non-termination.[12]

For illustration, we present some concrete specifications of DELEGATE, given in Algorithms 3 and 4. The common idea is that the system wide number of PLATYPUS

```
Algorithm 3: DELEGATE$_1$
Global  : Two integers $k, n$ indicating the current and maximum number of PLATYPUS
          instances.
Input   : A set $S$ of partial assignments.
Output  : A subset of $S$.
begin
    while $(k < n) \wedge (S \neq \emptyset)$ do
        $(X,Y) \leftarrow$ CHOOSE($S$)
        $S \leftarrow S \setminus \{(X,Y)\}$
        $k \leftarrow k+1$
        distribute PLATYPUS($\{(X,Y)\}$)
    return $S$
end
```

instances is limited (by n). Variable k holds the current number of PLATYPUS instances. Accordingly, in this specific setting, k must be declared in Algorithm 2 as a global variable and decremented after each execution of the **repeat** loop. In Algorithm 3, the delegation procedure tries to maximize the global number of PLATYPUS instances. Without external interference (a changing k), DELEGATE$_1$ tries to produce $(n-k)$ new PLATYPUS instances. Each instance is created[13] following one of a variety of strategies, for instance, taking into account temporal or structural criteria on the partial assignments in S as well as system-specific balance criteria. Algorithm 4 is less greedy insofar as it "removes"[14] a subset from S and creates only a single new PLATYPUS instance. As

[12] In fact, this cannot happen in a pure PLATYPUS setting, since at least one element is removed by each PLATYPUS instance in Line 2.

[13] To be precise, MPI instances are merely reinitialized; they live throughout the computation.

[14] Given a set X, SPLIT(X) returns a partition (X_1, X_2) of X.

Algorithm 4: DELEGATE$_2$

```
Global : Two integers $k, n$ indicating the current and maximum number of PLATYPUS
         instances.
Input  : A set $S$ of partial assignments.
Output : A subset of $S$.
begin
    if $k < n$ then
        $(S, D) \leftarrow$ SPLIT($S$)
        $k \leftarrow k + 1$
        distribute PLATYPUS($D$)
    return $S$
end
```

with CHOOSE previously, SPLIT can be guided by various strategies, e.g. trying to share the remaining search space equally among processes/processors in order to minimize communication costs that result from delegation operations. Also, numerous mixtures of both strategies can be envisaged. Both delegation procedures guarantee completeness. To see this, it is enough to observe that every assignment and thus every part of the search space is investigated by one PLATYPUS instance in one way or another. Also, duplicate solutions are avoided by having exactly one solver investigate each part of the search space.

k/n	PLATYPUS I	PLATYPUS II	PLATYPUS III
$1/3$	$\{\underline{(\emptyset, \emptyset)}\}$ $(\emptyset, \emptyset)$ $\{(\{\underline{s}\}, \emptyset), \overline{(\emptyset, \{\underline{s}\})}\}$		
$2/3$	$\{\underline{(\{s\}, \emptyset)}\}$ $(\{s, r\}, \{t\})$ $\{(\{s, r, \underline{q}\}, \{t\}), \overline{(\{s, r\}, \{t, \underline{q}\})}\}$	$\{\underline{(\emptyset, \{s\})}\}$ $(\emptyset, \{s\})$ $\{(\{\underline{p}\}, \{s\}), (\emptyset, \{s, \underline{p}\})\}$	
$3/3$	$\{\underline{(\{s, r, q\}, \{t\})}\}$ $(\boldsymbol{\{s, r, q\}}, \{t, p\})$ $\emptyset$	$\{\underline{(\{p\}, \{s\})}, (\emptyset, \{s, p\})\}$ $(\boldsymbol{\{p, r, t\}}, \{s, q\})$ $\{(\emptyset, \{s, p\})\}$	$\{\underline{(\{s, r\}, \{t, q\})}\}$ $(\boldsymbol{\{s, r, p\}}, \{t, q\})$ $\emptyset$
$1/3$		$\{\underline{(\emptyset, \{s, p\})}\}$ $(\boldsymbol{\{r, t, q\}}, \{s, p\})$ $\emptyset$	
$0/3$			

Table 1. Three PLATYPUS instances computing the answer sets of Π_2.

To illustrate, let us compute the answer sets of the program given in (2) in an environment with at most 3 PLATYPUS instances, using the delegation procedure in Algorithm 4. The **distribute** procedure is used for creating new processes. Our instance of SPLIT transfers $\lfloor |S|/2 \rfloor$ assignments to another PLATYPUS instance. These assignments are overlined in Table 1, while the ones chosen in Lines 1 and 7 in Algorithm 2 are underlined. The **print** of an answer set (in Line 5) is indicated by boldface letters. A cell in Table 1 represents an iteration in Algorithm 2. In each cell the first entry is S after Line 1, the second presents the result of the EXPAND function, and the last one is S before Line 9. Once the environment has been initialized, setting $n = 3$ and $k = 1$ among other things, the first PLATYPUS instance is invoked with $\{(\emptyset, \emptyset)\}$. After the first iteration, its search space contains $(\{s\}, \emptyset)$, while $(\emptyset, \{s\})$ is used to create a second instance of PLATYPUS. Also, variable k is incremented by DELEGATE$_2$. After one more iteration, each PLATYPUS instance could potentially create yet another instance. Since the maximum number of processes is limited to 3, only one of them is able to create a new PLATYPUS instance. Once this is done k equals 3, which prevents DELEGATE$_2$ from creating any further processes. In our case, PLATYPUS I manages to delegate $(\{s, r\}, \{t, q\})$ while blocking any delegation by PLATYPUS II. The three processes output their answer sets. Whereas the first and third terminate, having emptied their search spaces, the second one iterates once more to compute the fourth and last answer set. (No delegation is initiated since $\lfloor |S|/2 \rfloor = \lfloor 1/2 \rfloor = 0$.)

4 The `platypus` platform

Current technology provides a large variety of software and hardware mechanisms for distributed computing. Among them, we find single- and multi-threaded processes, multiple processes, as well as multiple processors, sometimes combined with multiple processes and threads.

The goal of the `platypus` platform is to provide an easy and flexible use of these architectures for ASP. To begin with, we have implemented a multiple process and a multiple processor variant of `platypus`. A multi-threaded variant is currently under development. To enable the generality of the approach, the `platypus` system is designed in a strictly modular way. The central module consists of a *black-box* providing the functionality of the EXPAND function. This module provides a fixed interface that permits wrapping different off-the-shelf propagation engines. A second module deals with the search space given by variable S in Algorithms 2, 3, and 4. Last but not least, distribution is handled by a dedicated control module fixing the respective implementation of the **distribute** operation. That is, this module controls forking, threading or MPI. With this module, each variant is equipped with a common set of distribution policies; currently all of them are realized through controllers being variations of DELEGATE$_2$ (cf. Algorithm 4), where SPLIT is replaced by CHOOSE. Hence, the currently implemented policies vary the strategy of the CHOOSE operation. Each policy depends upon the underlying distribution capabilities of the software and hardware architectures.

The multiple process variant is implemented with the `UNIX fork` mechanism, which creates a child process managed by the same operating system that controls the parent process. The forking policy requires a local controller in each process to perform the delegation task. Communication among the processes is accomplished via shared memory. The forking policy provides an easily manageable framework to test design alternatives and to experiment with low-level distribution policy decisions, such as resource saturation caused by having too many processes. The multiple processor variant runs on a cluster and relies on *The Message Passing Interface* (MPI [15]) library to perform the distribution to a process controlled by another operating system on a distinct processor. As with forking, we rely on a local controller in each process to control the delegation task; in addition, we use a global controller to manage the processes on the separate processors and the communication among processes.

To reduce the size of partial assignments and thus of passed messages, we follow [8] in storing only atoms whose truth values were assigned by a choice operation (cf. atom A in Lines 7 and 8 of Algorithm 2). Given an assignment (X, Y) along with the subsets $X_c \subseteq X$ and $Y_c \subseteq Y$ of atoms treated in this way, we have $(X, Y) = \text{EXPAND}((X_c, Y_c))$. Accordingly, some care must be taken when implementing the tests in Lines 4 and 5. To this end, the current design foresees two signals provided by the EXPAND module. The search space module (1) must support multiple access modes for accommodating the varying choice policies in Lines 1 and 9 (or better the subsequent delegation procedures) of Algorithm 2, and (2) must be handled with care, since it may grow exponentially without appropriate restrictions. So that, at this stage of the project, we can focus on issues arising from our distribution-oriented setting, the current implementation is based on the design decision that a non-distributing `platypus` instance must correspond to a traditional solver, as given in Algorithm 1. This has the advantage that we obtain a "bottom-line" solver instance that we can use for comparison with state-of-the-art solvers as well as all distributed `platypus` instances for measuring the respective trade-offs. To this end, we restrict the search space (in S) to a single branch of the search tree and implement the "local" choice operation in Line 1 of Algorithm 2 through a LIFO strategy. In this way, the "local" view of the search space can be realized by stack-based operations. Unlike this, the second access to the search space, described in the delegation procedures 3 and 4 is completely generic. This allows us to integrate various delegation policies (cf. Section 5). Following the above decision, our design also foresees the option of using a choice proposed by a given EXPAND module for implementing Line 7 in Algorithm 2, provided that this is supported by the underlying propagation engine.

Finally, let us detail some issues of the current implementation. `platypus` is written in `C++`. Its major EXPAND module is based on the `smodels` API. This also allows us to take advantage of `smodels`' heuristics in view of Line 7 in Algorithm 2 (see above). Accordingly, this module requires `lparse` for parsing logic programs. All experiments reported in Section 5 are conducted with this implementation of EXPAND. However, for guaranteeing modularity, we have also implemented other EXPAND modules, among them the one of the `nomore++` system [16]. While the distribution architecture of a `platypus` instance must be fixed at compile time, the respective parameters, like constant n in Algorithm 3, or delegation policies, such as the kind of the delegated choice point, are set via command line options at run-time. More details are given in the experimental section.

5 Experimental results

To illustrate the feasibility of our approach, we present in Tables 2 and 3 a selection of experimental results obtained with the multi-process and the multi-processor versions of `platypus`. As a point of reference, we mention that on all these tests `smodels`

platypus	-p1 -ls	-p2 -ls	-p3 -ls	-p4 -ls
color-5-10	6.44 (0)	3.54 (21.1)	2.55 (44.7)	2.12 (69.2)
color-5-15	349.07 (0)	178.70 (36.5)	120.24 (63.2)	91.15 (99.8)
hamcyc-8	4.35 (0)	2.61 (30.9)	1.94 (55.4)	1.66 (87.1)
hamcyc-9	105.88 (0)	54.59 (40.6)	36.73 (88.3)	28.13 (134.8)
pigeon-7-8	1.92 (0)	1.60 (38.5)	1.33 (68.5)	1.22 (92.7)
pigeon-7-9	7.44 (0)	4.49 (45.2)	3.35 (84.8)	2.83 (115.8)
pigeon-7-10	24.21 (0)	12.86 (49.0)	9.04 (99.3)	7.24 (142.6)
pigeon-7-11	71.40 (0)	34.93 (55.9)	23.73 (111.8)	18.34 (165.2)
pigeon-7-12	177.02 (0)	85.71 (60.5)	57.53 (124.2)	44.04 (193.3)
pigeon-8-9	18.83 (0)	9.99 (46.3)	7.09 (94.7)	5.73 (138.8)
pigeon-8-10	87.45 (0)	43.23 (49.5)	29.22 (112.3)	22.33 (163.4)
pigeon-9-10	227.72 (0)	107.14 (60.3)	71.14 (123.8)	53.56 (189.2)
schur-11-5	1.50 (0)	1.15 (18.1)	0.82 (25.4)	0.71 (34.9)
schur-12-5	5.26 (0)	3.09 (19.6)	2.23 (34.9)	1.80 (48.0)
schur-13-5	22.53 (0)	11.78 (20.6)	8.07 (37.9)	6.22 (56.9)
schur-14-5	74.51 (0)	37.80 (19.9)	25.60 (46.0)	19.25 (68.0)
schur-14-4	2.93 (0)	1.88 (16.4)	1.52 (41.9)	1.17 (46.0)
schur-15-4	8.02 (0)	4.54 (20.4)	3.23 (41.3)	2.55 (55.5)
schur-16-4	14.14 (0)	7.64 (24.1)	5.28 (43.9)	4.13 (62.3)
schur-17-4	32.50 (0)	16.92 (22.9)	11.50 (48.1)	8.82 (68.3)
schur-18-4	62.72 (0)	31.23 (21.8)	20.77 (52.8)	15.75 (74.7)
schur-19-4	132.30 (0)	65.99 (22.6)	44.02 (54.2)	33.24 (77.7)
schur-20-4	164.24 (0)	80.70 (26.1)	53.61 (60.5)	40.09 (75.0)

Table 2. Data obtained for computing *all* answer sets with the multi-process version of `platypus`.

is on average 1.62 times faster than the multi-process version of `platypus` limited to one process. Apart from `platypus`' early stage of development, a certain overhead is created by "double bookkeeping" due to the strict encapsulation of the EXPAND module. In this way, we trade-off some speed for our modular design.

The multi-process data were generated using the forking architecture limited to 1 to 4 active processes on a quad processor under `Linux`, comprised of 4 Opteron 2.2GHz processors with 8 GB shared RAM. The multi-processor tests ran on a cluster of 5 Pentium III 1GHz PCs under `Linux` with 1 GB RAM each, with 1 to 4 active nodes and one extra node serving as master.[15] All of our timing results reflect the average elapsed time (in seconds) of the launching process/processor, respectively, over 20 runs, each computing *all* answer sets. Timing excludes parsing and printing time. Similarly, the number in parentheses indicates the average number of forks/messages passed, respectively.

The first column of Tables 2 and 3 lists the benchmarks, largely taken from the benchmarking site at [17]. Columns 2 to 5 give the forking architecture results, and columns 6 to 9 contain the data obtained from the cluster using MPI. The first row pro-

[15] Note that the former processor type is much faster than the latter.

platypus	-p1 -ls	-p2 -ls	-p3 -ls	-p4 -ls
color-5-10	28.18 (4)	14.57 (119.0)	9.99 (245.1)	7.74 (385.4)
color-5-15	1632.91 (4)	821.83 (120.3)	549.68 (258.4)	413.75 (616.5)
hamcyc-8	16.59 (4)	8.89 (146.8)	6.10 (312.4)	4.77 (454.3)
hamcyc-9	407.43 (4)	202.16 (219.8)	135.96 (585.8)	102.80 (993.6)
pigeon-7-8	6.37 (4)	4.01 (82.8)	3.23 (176.6)	2.87 (243.3)
pigeon-7-9	25.53 (4)	13.99 (131.8)	10.18 (251.6)	8.54 (349.1)
pigeon-7-10	84.28 (4)	42.92 (167.3)	30.08 (335.3)	24.01 (546.9)
pigeon-7-11	238.62 (4)	117.46 (198.3)	81.07 (476.1)	62.73 (750.0)
pigeon-7-12	590.83 (4)	291.03 (219.3)	197.33 (581.1)	150.43 (972.6)
pigeon-8-9	62.70 (4)	32.22 (146.3)	22.71 (281.8)	18.44 (486.0)
pigeon-8-10	282.19 (4)	139.72 (176.8)	95.51 (453.8)	73.75 (753.7)
pigeon-9-10	695.56 (4)	341.25 (214.0)	230.35 (586.1)	175.02 (1024.1)
schur-11-5	5.45 (4)	3.19 (40.5)	2.67 (83.7)	2.27 (133.6)
schur-12-5	18.85 (4)	9.88 (30.3)	7.19 (81.9)	5.95 (195.6)
schur-13-5	78.90 (4)	40.16 (48.5)	27.36 (104.5)	21.30 (247.5)
schur-14-5	254.78 (4)	129.08 (70.5)	86.70 (106.8)	66.05 (297.1)
schur-14-4	10.26 (4)	5.55 (27.5)	4.38 (112.9)	3.77 (187.2)
schur-15-4	27.83 (4)	14.56 (44.5)	10.24 (102.7)	8.52 (251.3)
schur-16-4	48.56 (4)	24.92 (56.3)	17.27 (122.8)	14.16 (291.6)
schur-17-4	113.29 (4)	57.61 (65.8)	39.47 (164.9)	30.36 (291.3)
schur-18-4	206.20 (4)	103.74 (47.5)	70.19 (172.5)	53.92 (369.6)
schur-19-4	450.43 (4)	225.75 (75.3)	151.40 (226.6)	113.60 (306.2)
schur-20-4	539.21 (4)	270.21 (70.0)	180.97 (237.8)	135.70 (335.9)

Table 3. Data obtained for computing *all* answer sets with the multi-processor version of `platypus`.

vides the command line options with which `platypus` was invoked. The `-p` option indicates the maximum number of processes or processors, respectively (n in Algorithms 3 and 4). And `-l` stands for the delegation policy. All listed tests are run in `s`*hallow* mode, delegating the smallest among all of the delegatable partial assignments available to the delegation procedure. The current system also supports a `d`*eep*, `m`*iddle*, and `r`*andom* mode. Delegating small partial assignments is theoretically reasonable since they represent the putatively largest parts of the search space, thus each delegated `platypus` instance will be given the largest task to perform, hence minimizing the amount of delegation. Our early experiments and similar observations reported in [8] support this view. In fact, selecting the largest assignment (via option `-ld`) results in much more forking/message passing and much poorer performance. For instance, using the forking architecture, we get on average for schur-20-4 with options `-p4 -ld` a time of 155.43s and a count of 322 forks.

The results in Tables 2 and 3 are indicative of what is expected from distributed computation. When looking at each benchmark, the forking and MPI experiments show a qualitatively consistent 2-, 3-, and 4-times speed-up when doubling, tripling, and quadrupling the number of processors, with only minor exceptions. The more substantial the benchmark, the more clear-cut the speed-up. A more global and more quantitative sense of the speed-up is provided by the sum of times[16] for all benchmarks for each `-p` setting compared to "`-p 1`". These ratios are: 1, 1.98, 2.93, 3.85, and 1, 1.99, 2.96, 3.88 for forking and MPI, respectively. With this in mind, we observe on computationally undemanding benchmarks, like hamcyc-8, pigeon-7-8, or schur-11-5 no real gain. In fact, our overall experiments show that the less substantial the benchmark, the more insignificant the speed-up as the overhead caused by distribution dominates the actual search time. Similarly, the sum of messages increases from "`-p 2`" to "`-p 3`" and to "`-p 4`" by 2.03, 2.95, and by 2.41, 4.16, respectively for forking and MPI. To interpret the number of messages in Tables 2 and 3, the one in the forking results reflects the number of delegations, whereas the number of messages in our MPI setting include $2(n+1)$ start-up and shut-down messages plus 5 handshaking messages for each delegation.

6 Summary

Conceptually, the PLATYPUS approach offers a general and flexible framework for managing the distributed computation of answer sets, incorporating a variety of different software and hardware architectures for distribution. The major design decisions were to minimize the number and size of the messages passed by appeal to partial assignments and to abstract from a serial ASP propagation system. The latter allows us to take advantage of efficient off-the-shelf engines, developed within the ASP community for the non-distributive case.

Meanwhile, the `platypus` system furnishes a platform for implementing various forms of distribution. All versions of `platypus` share the same code, except for the control module matching the specific distribution architecture. The current system supports a multiple process architecture, using the `UNIX` forking mechanism, and a multiple processor architecture, running on a cluster via MPI. A multi-threaded variant is currently under development. Moreover, `platypus` supports different distribution policies, being open to further extensions through well-defined interfaces.

Finally, the encouraging results from our experiments suggest that our generic approach to the distributed computation of answer sets offers a powerful computational enhancement to classical answer set solvers. In particular, we have seen a virtually optimal speed-up on substantial benchmarks, that is, the speed-up nearly matched the number of processes or processors, respectively.

The `platypus` platform is freely available on the web [18]. The current system provides us with the necessary infrastructure for manifold future investigations into distributed answer set solving. This concerns the whole spectrum of different instances of the procedures CHOOSE and SPLIT, on the one side, and **distribute** on the other. A systematic study of different options in view of dynamic load balancing will be a major issue of future experimental research. In fact, the given set of benchmarks was chosen as a representative selection demonstrating the feasibility of our approach. In view of load balancing, it will be interesting to see how the type of benchmark (and thus the underlying search space) is related to specific distribution schemes. Also, we

[16] Of course, these ratios are biased by the more substantial benchmarks, but these are the more reasonable indicators of the speed-up.

have so far concentrated on finding *all* answer sets of a given program. When extending the system for finding some answer set(s) only, the structure of the search space will become much more important.

Acknowledgments The first, fourth, and fifth authors were supported by DFG under grant SCHA 550/6-4. All but the third and last authors were also funded by the EC through IST-2001-37004 WASP project. The third and last authors were funded by NSERC (Canada) and SHARCNET. Finally, we would like to thank Christine Bailey for her contribution to the very early development of `platypus`.

References

1. Simons, P., Niemelä, I., Soininen, T.: Extending and implementing the stable model semantics. Artificial Intelligence **138** (2002) 181–234
2. Leone, N., Faber, W., Pfeifer, G., Eiter, T., Gottlob, G., Koch, C., Mateis, C., Perri, S., Scarcello, F.: The DLV system for knowledge representation and reasoning. ACM Transactions on Computational Logic (2005) To appear.
3. Davis, M., Logemann, G., Loveland, D.: A machine program for theorem-proving. Communications of the ACM **5** (1962) 394–397
4. Fitting, M.: Fixpoint semantics for logic programming: A survey. Theoretical Computer Science **278** (2002) 25–51
5. van Gelder, A., Ross, K., Schlipf, J.: The well-founded semantics for general logic programs. Journal of the ACM **38** (1991) 620–650
6. Finkel, R., Marek, V., Moore, N., Truszczynski, M.: Computing stable models in parallel. In Provetti, A., Son, T., eds.: Proceedings of AAAI Spring Symposium on Answer Set Programming, AAAI/MIT Press (2001) 72–75
7. Hirsimäki, T.: Distributing backtracking search trees. Technical report, Helsinki University of Technology (2001)
8. Pontelli, E., Balduccini, M., Bermudez, F.: Non-monotonic reasoning on beowulf platforms. In Dahl, V., Wadler, P., eds.: Proceedings of the Fifth International Symposium on Practical Aspects of Declarative Languages. (2003) 37–57
9. Finkel, R., Manber, U.: DIB — a distributed implementation of backtracking. ACM Transactions on Programming Languages and Systems **9** (1987) 235–256
10. Gupta, G., Pontelli, E., Ali, K., Carlsson, M., Hermenegildo, M.: Parallel execution of prolog programs: a survey. ACM Transactions on Programming Languages and Systems **23** (2001) 472–602
11. Chassin de Kergommeaux, J., Codognet, P.: Parallel logic programming systems. ACM Computing Surveys **26** (1994) 295–336
12. Zhang, H., Bonacina, M., Hsiang, J.: PSATO: a distributed propositional prover and its application to quasigroup problems. Journal of Logic and Computation **21** (1996) 543–560
13. Blochinger, W., Sinz, C., Küchlin, W.: Parallel propositional satisfiability checking with distributed dynamic learning. Parallel Computing **29** (2003) 969–994
14. Inoue, K., Koshimura, M., Hasegawa, R.: Embedding negation as failure into a model generation theorem prover. In Kapur, D., ed.: Proceedings of the Eleventh International Conference on Automated Deduction. Springer-Verlag (1992) 400–415
15. Gropp, W., Lusk, E., Thakur, R.: Using MPI-2: Advanced Features of the Message-Passing Interface. The MIT Press (1999)
16. (`http://www.cs.uni-potsdam.de/nomore`)
17. (`http://asparagus.cs.uni-potsdam.de`)
18. (`http://www.cs.uni-potsdam.de/platypus`)