PLATYPUS: A platform for distributed answer set solving

Jean Gressmann[*] Tomi Janhunen[†] Robert E. Mercer[‡][§] Torsten Schaub[*][¶] Sven Thiele[*] Richard Tichy[*][‡]

The success of Answer Set Programming (ASP) has been greatly enhanced by the availability of highly efficient ASP-solvers [1, 5, 8, 10]. But, more complex applications with their significantly larger search spaces are requiring computationally more powerful search engines. Distributing parts of the search space among cooperating sequential solvers performing independent searches can provide increased computational power. Our approach to distributed answer set solving differs in philosophy from other pioneering work in distributed answer set solving [3, 9], by accommodating in a single design a variety of architectures for distributing the search for answer sets over different processes. Concentrating on the stack-based architecture of the well-known DPLL-based ASP solvers, the resulting platform, `platypus`, permits a reasonably straightforward way to connect `platypus`'s different types of inter- and intra-process distribution techniques (like MPI [7], Unix' fork mechanism, and multi-threading) to the ASP's solver via its API, thereby allowing one to exploit the increased computational power of clustered and/or multi-processor machines seamlessly with only a small amount of programming effort. In addition, the generic approach permits a flexible instantiation of all parts of the design.

The PLATYPUS design incorporates two distinguishing features: First, it modularises (and is thus independent of) the DPLL-style propagation engine. Currently, we have successfully integrated `smodels`'[10] and `nomore++`' [1] expansion procedures with `platypus`. Second, the search space is represented explicitly. This representation allows a flexible distribution scheme to be incorporated, thereby accommodating different distribution policies and architectures. At this time, we have incorporated three assignment-based distribution policies and one randomized yet complete distribution policy (called probing), and three architectures: MPI, forking, and multi-threading.

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 [2]. 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 [11]. 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. 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.

Now, we present the major features of the PLATYPUS approach [6]. To enable a distributed search for answer sets, the search space is decomposed by means of partial assignments. This method works because

[*]Institut für Informatik, Universität Potsdam, Postfach 900327, D-14439 Potsdam, Germany

[†]Helsinki University of Technology, Department of Computer Science and Engineering, Laboratory for Theoretical Computer Science, P.O. Box 5400, FI-02015 TKK, Finland

[‡]Computer Science Department, Middlesex College, The University of Western Ontario, London, Ontario, Canada N6A 5B7

[§]Affiliated with the David R. Cheriton School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada N2L 3G1

[¶]Affiliated with the School of Computing Science, Simon Fraser University, Burnaby, B.C., Canada V5A 1S6

Algorithm 1: 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 $\Pi$.

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 print $X'$ else
6           $A \leftarrow$ CHOOSE($\mathcal{A} \setminus (X' \cup Y')$)
7           $S \leftarrow S \cup \{ (X' \cup \{A\}, Y'),\ (X', Y' \cup \{A\}) \}$
8       $S \leftarrow$ DELEGATE($S$)
until $S = \emptyset$
```

partial assignments that differ with respect to defined atoms represent different parts of the search space. To this end, Algorithm 1 is based on an explicit representation of the search space in terms of a set S of partial assignments, on which it iterates until S becomes empty. The algorithm relies on the omnipresence of a logic program Π and its alphabet $\mathcal{A}$ as global parameters. Communication between PLATYPUS instances is limited to delegating partial assignments as representatives of parts of the search space. The set of partial assignments provided in the input variable S delineates the search space given to a specific instance of PLATYPUS. 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 Line 8, Algorithm 1 computes all answer sets extending the partial assignments given as input. With Line 8 each PLATYPUS instance generates a subset of the answer sets. CHOOSE and DELEGATE are in principle non-deterministic selection functions: CHOOSE yields a single element, DELEGATE communicates a subset of S to a PLATYPUS instance and returns a subset of S. Clearly, depending on what these subsets are, this algorithm is subject to incomplete and redundant search behaviours. The EXPAND function hosts the deterministic part of Algorithm 1. This function is meant to be implemented with an off-the-shelf ASP-expander that provides both sufficiently strong as well as efficient propagation operations. The ASP-expander is used as a grey-box. Program communication is only through the API, but having some knowledge of the inner workings of the expander can benefit some design decisions.

We now turn to specific design issues beyond the generic description of Algorithm 1. To reduce the size of partial assignments and thus that of passed messages, we follow [9] in representing partial assignments only by propositions[1] whose truth values were assigned by choice operations (cf. atom A in Lines 6 and 7). Given assignment (X,Y) with its subsets $X_c \subseteq X$ and $Y_c \subseteq Y$ of atoms assigned by a choice operation, we have $(X,Y) =$ EXPAND$((X_c,Y_c))$. Consequently, the expansion of assignment (X,Y) to (X',Y') in Line 3 does not affect the representation of the search space in S.[2] Furthermore, the design includes the option of using a choice proposed by the EXPAND component for implementing Line 6. Additionally, the currently used expanders, `smodels` and `nomore++`, also supply a *polarity* for the choice point, indicating a preference for assigning true or false first.

Each `platypus` process has an explicit representation of its (part of the) search space in its variable

[1]Assignments are not restricted to atoms, as used when using `nomore++`.

[2]Accordingly, the tests in Lines 4 and 5 must be handled with care; see [6].

S. This set of partial assignments is implemented as a tree. Whenever more convenient, we describe S in terms of a set of assignments or a search tree and its branches. In contrast to stack-based ASP-solvers, like `smodels` or `nomore++`, whose search space contains a single branch at a time, this tree normally contains several independent branches. The *active* partial assignment (or branch) selected in Line 1, is the one being currently treated by the expander. The state of the expander is characterised by the contents of its stack, which corresponds to the active branch in the search tree. While the stack contains the full assignment (X, Y), the search tree's active branch only contains the pair of subsets (X_c, Y_c).

When looking at all benchmarks in all experiments that we have run, the multi-threading, forking, and MPI distribution techniques 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 is the benchmark, the more clear-cut becomes the speed-up.

When we weight the speed-ups by the average speed-ups with respect to average running times, we have observed a slightly super-linear speed-up. We ascribe such super-linear speed-ups, observed primarily on time-demanding benchmarks, to caching and/or shared memory effects. Finally, we note a small 2% performance degradation caused by the overhead of using the multi-threaded core.

References

[1] C. Anger, M. Gebser, T. Linke, A. Neumann, and T. Schaub. The `nomore++` approach to answer set solving. In G. Sutcliffe and A. Voronkov, editors, *Proceedings of the Twelfth International Conference on Logic for Programming Artificial Intelligence and Reasoning (LPAR'05)*, volume 3835 of *Lecture Notes in Artificial Intelligence*, pages 95–109. Springer-Verlag, 2005.

[2] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5:394–397, 1962.

[3] R. Finkel, V. Marek, N. Moore, and M. Truszczynski. Computing stable models in parallel. In A. Provetti and T. Son, editors, *Proceedings of AAAI Spring Symposium on Answer Set Programming (ASP'01)*, pages 72–75. AAAI/MIT Press, 2001.

[4] M. Fitting. Fixpoint semantics for logic programming: A survey. *Theoretical Computer Science*, 278(1-2):25–51, 2002.

[5] M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub. Conflict-driven answer set solving. In M. Veloso, editor, *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI'07)*, pages 386–392. AAAI Press/The MIT Press, 2007. Available at http://www.ijcai.org/papers07/contents.php.

[6] J. Gressmann, T. Janhunen, R. Mercer, T. Schaub, S. Thiele, and R. Tichy. Platypus: A platform for distributed answer set solving. In C. Baral, G. Greco, N. Leone, and G. Terracina, editors, *Proceedings of the Eighth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'05)*, volume 3662 of *Lecture Notes in Artificial Intelligence*, pages 227–239. Springer-Verlag, 2005.

[7] W. Gropp, E. Lusk, and R. Thakur. *Using MPI-2: Advanced Features of the Message-Passing Interface*. The MIT Press, 1999.

[8] N. Leone, W. Faber, G. Pfeifer, T. Eiter, G. Gottlob, C. Koch, C. Mateis, S. Perri, and F. Scarcello. The DLV system for knowledge representation and reasoning. *ACM Transactions on Computational Logic*, 7(3):499–562, 2006.

[9] E. Pontelli, M. Balduccini, and F. Bermudez. Non-monotonic reasoning on beowulf platforms. In V. Dahl and P. Wadler, editors, *Proceedings of the Fifth International Symposium on Practical Aspects of Declarative Languages (PADL'03)*, volume 2562 of *Lecture Notes in Artificial Intelligence*, pages 37–57, 2003.

[10] P. Simons, I. Niemelä, and T. Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138(1-2):181–234, 2002.

[11] A. van Gelder, K. Ross, and J. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650, 1991.