On Probing and Multi-Threading in PLATYPUS

J. Gressmann\textsuperscript{1} and T. Janhunen\textsuperscript{2} and R. Mercer\textsuperscript{3} and T. Schaub\textsuperscript{1,4} and S. Thiele\textsuperscript{1} and R. Tichy\textsuperscript{1}

Abstract. The PLATYPUS approach offers a generic platform for distributed answer set solving, accommodating a variety of different architectures for distributing the search for answer sets across different processes and different search modes for modifying search behaviour. We describe two major extensions of PLATYPUS. First, we present its probing mode which provides a controlled non-linear traversal of the search space. Second, we present its new multi-threading architecture allowing for intra-process distribution. Both contributions are underpinned by experimental results illustrating their computational impact.

1 INTRODUCTION

The success of Answer Set Programming (ASP) has been greatly enhanced by the availability of highly efficient ASP-solvers \cite{8,11}. But, more complex applications are requiring computationally more powerful devices. Distributing parts of the search space among cooperating sequential solvers performing independent searches can provide increased computational power. We have proposed a generic approach to distributed answer set solving, called PLATYPUS \cite{5}.

The PLATYPUS approach differs from other pioneering work in distributed answer set solving \cite{3,10}, by accommodating in a single design a variety of different architectures for distributing the search for answer sets over different processes. The resulting platform, platypus, allows one to exploit the increased computational power of clustered and/or multi-processor machines via different types of inter- and intra-process distribution techniques like MPI \cite{7}, Unix' fork mechanism, and (as discussed in the sequel) multi-threading. In addition, the generic approach permits a flexible instantiation of all parts of the design.

More precisely, the PLATYPUS design incorporates two distinguishing features: First, it modularises (and is thus independent of) the propagation engine (currently exemplified by \texttt{models}' and \texttt{nomore+}' expansion procedures). Second, the search space is represented explicitly. This representation allows a flexible distribution scheme to be incorporated, thereby accommodating different distribution policies and architectures. The two particular contributions discussed in this paper take advantage of these two aspects of the generic design philosophy. The first extension to PLATYPUS, probing, refines the encapsulated module for propagation. Probing is akin to restart in the SAT solving framework \cite{4}. The introduction of probing demonstrates one aspect of the flexibility in our PLATYPUS design: by having a modularised generic design, we can easily specify parts of the generic design to give different computational properties to the platypus system. Our second improvement to platypus is the integration of multi-threading into our software package \cite{9}. Multi-threading expands the implemented architectural options for delegating the search space and adds several new features to platypus: (1) the single- and multi-threaded versions can take advantage of new hardware innovations such as multi-core processors, as well as primitives to implement lock-free data structures, (2) a hybrid architecture which allows the mixing of inter- and intra-process distribution, and (3) the intra-process distribution provides a lighter parallelisation mechanism than forking.

We highlight our two contributions, probing and multi-threading, by focusing on the appropriate aspects of the abstract PLATYPUS algorithm reproduced from \cite{5} below. As well, their computational impact is exposed in data provided by a series of experiments.

2 THE PLATYPUS APPROACH

In ASP, a logic program \( \Pi \) is associated with a set of answer sets, \( AS(\Pi) \), which are distinguished models of the rules in \( \Pi \). We do not elaborate, but refer the reader to \cite{2} for a formal introduction to ASP. For computing answer sets, we rely on partial assignments, mapping atoms in an alphabet \( \mathcal{A} \) 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. In general, a partial assignment \( (X, Y) \) aims at capturing a subset of the answer sets of \( \Pi \), viz. \( AS(X,Y)(\Pi) = \{ Z \in AS(\Pi) \mid X \subseteq Z, Z \cap Y \neq \emptyset \} \).

To begin, we recapitulate the major features of the PLATYPUS approach \cite{5}. To enable a distributed search for answer sets, the search space is decomposed by means of partial assignments. This method works because partial assignments that differ with respect to defined atoms represent different parts of the search space. To this end, Alg.

\textbf{Algorithm 1: PLATYPUS} \texttt{Global} : A logic program \( \Pi \) over alphabet \( \mathcal{A} \).
\texttt{Input} : A nonempty set \( S \) of partial assignments.
\texttt{Output} : Print a subset of the answer sets of \( \Pi \).
\begin{verbatim}
repeat
1 (X, Y) ← CHOOSE(S)
2 S ← S \ {(X,Y)}
3 (X', Y') ← EXPAND((X,Y))
4 if X' ∩ Y' = \emptyset then
5 if X' ∪ Y' = \mathcal{A} then print X' else
6 A ← CHOOSE(\mathcal{A} \setminus (X' ∪ Y'))
7 S ← S \ { (X' ∪ \{A\}, Y'), (X', Y' ∪ \{A\}) } 
8 S ← DELEGATE(S)
until S = \emptyset
\end{verbatim}

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

\footnotesize
\begin{tabular}{ll}
1 Universit"at Potsdam, Postfach 900327, D-14439 Potsdam, Germany.\
2 Helsinki University of Technology, P.O. Box 5400, FI-02015 TKK, Finland.\
3 University of Western Ontario, London, Ontario, Canada N6A 5B7.\
4 Affiliated with Simon Fraser University, Burnaby, Canada.
\end{tabular}
logic program II and its alphabet \(\mathcal{A}\) as global parameters. Communication between \textsc{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 \textsc{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 in \(\bigcup_{(X,Y) \in S} AS_{(X,Y)}(\Pi)\). With Line 8 each \textsc{Platypus} instance generates a subset of the answer sets. \textsc{Choose} and \textsc{Delegate} are in principle non-deterministic selection functions: \textsc{Choose} yields a single element, \textsc{Delegate} communicates a subset of \(S\) to a \textsc{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 \textsc{Expand} function hosts the deterministic part of Algorithm 1. This function is meant to be implemented with an off-the-shelf ASP-expander that is used as a black-box providing both sufficiently strong as well as efficient propagation operations. See [5] for details.

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 [10] in representing partial assignments only by atoms\(^5\) 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 \((X, Y) = \textsc{Expand}(X_c, Y_c)\). Consequently, the expansion of assignment \((X, Y)\) to \((X', Y')\) in Line 3 does not affect the representation of the search space in \(S\).\(^6\) Furthermore, the design includes the option of using a choice proposed by the \textsc{Expand} component for implementing Line 6. Additionally, the currently used expanders, \texttt{smmodels} and \texttt{nomore++}, also supply a \textit{polarity}, indicating a preference for assigning true or false.

Each \textsc{Platypus} process has an explicit representation of its (part of the) search space in its variable \(S\). 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 \texttt{smmodels} or \texttt{nomore++}, whose search space contains a single branch at a time, this tree normally contains several independent branches. The \textit{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 \((X_c, Y_c)\).

3 PROBING

The explicit representation of the (partial) search space, although originally devised to enable the use of a variety of strategies for delegating parts of the search space in the distributed setting, appears to be beneficial in some sequential contexts, as well. Of particular interest, when looking for a single answer set, is limiting fruitless searches in parts of the search tree that are sparsely populated with answer sets. In such cases, it seems advantageous to leave a putatively sparsely populated part and continue at another location in the search space. In \textsc{Platypus}, this decision is governed by two command line options, \#\(c\) and \#\(j\). A part of the search is regarded as fruitless, whenever the number of conflicts (as encountered in Line 4) exceeds the value of \#\(c\). The corresponding conflict counter\(^7\) \(c\) is incremented each time a conflict is detected in Line 4. The counter \(c\) is reset to zero whenever an answer set is found in Line 5 or the active branch in \(S\) is switched (and thus the expander is reinitialised; see Algorithm 2). The number of jumps in the search space is limited by \#\(j\); each jump changes the active branch in the search space. We use a binary exponential back-off (cf. [12]) scheme to heed unsuccessful jumps. The idea is as follows. At first, probing initiates a jump in the search space whenever the initial conflict limit \#\(c\) is reached. If no solution is found after \#\(j\) jumps, then the problem appears to be harder than expected. In this case, the permissible number of conflicts \#\(c\) is doubled and the allowed number of jumps \#\(j\) is halved. The former is done to prolong systematic search, the latter to reduce gradually to zero the number of jumps in the search space. We refer to this treatment of the search space as \textit{probing}. Probing is made precise in Algorithm 2, which is a refinement of the \textsc{Choose} operation in Line 1 of Algorithm 1. Note that probing continues until the parameter \#\(j\) becomes zero. When probing stops, search proceeds in the usual depth-first manner by considering only one branch at a time by means of the expander’s stack. Clearly, this is also the case during the phases when the conflict limit has not been reached (\(c \leq \#c\)).

At the level of implementation, the expander must be reinitialised whenever the active branch of the search space changes. Reinitialisation is unnecessary when extending the active branch by the choice (obtained in Line 6) in Line 7 of Algorithm 1 or when backtracking is possible in case a conflict occurs or an answer set is obtained. In the first case, the expander’s choice (that is, an atom with a truth value) is simply pushed on top of the expander’s stack (and marked as a possible backtracking point). At the same time, the active branch in \(S\) is extended by the choice and a copy of the active branch, extended by the complementary choice, is added to \(S\). See [6] for details.

In the case that a conflict occurs or an answer set is obtained, the active branch in \(S\) is replaced by the branch corresponding to the expander’s stack after backtracking. If it exists, this is the largest branch in \(S\) that equals a subbranch of the active branch after switching the truth value of its leaf element. If backtracking is impossible, the active branch is chosen by means of the given policy \(P\) (at present, a largest, a smallest, or a random assignment). If this, too, is impossible\(^7\), each thread has its own conflict and jump counters.

\footnotesize
\begin{itemize}
  \item \(^5\) Assignments are not restricted to atoms, as used when using \texttt{nomore++}.
  \item \(^6\) Accordingly, the tests in Lines 4 and 5 must be handled with care; see [5].
  \item \(^7\) Each thread has its own conflict and jump counters.
\end{itemize}
threads in the core object, and it shows the number of partial assignments in the thread delegation queue of the master thread. Slave threads share their search space automatically among themselves as long as one thread has some work left. A slave thread running out of work (reaching an empty search space $S$) checks the availability of work via the idle thread counter and if possible removes a partial assignment from the thread delegation queue. Otherwise, it waits until new work is assigned to it. Another slave thread can become aware of the existence of an idle thread by noting that the idle thread counter exceeds zero during one of its periodic checks. If this is the case, it splits off a part of its local search space according to a distribution policy, puts the partial assignment that represents the subspace into the thread delegation queue, and decrements the idle thread counter.

As this may happen simultaneously in several working slave threads, more partial assignments can end up in the thread delegation queue than there exist idle slaves. These extras are used subsequently by idle threads.

When all slave threads are idle (i.e. the idle thread counter equals the number of slave threads) the master thread initiates communication via the distribution object to acquire more work from other Platypus processes. To this end, the master thread periodically queries the associated distribution object for work until it either gets some work or is requested to terminate. Once work is available, the master thread adds it to the thread delegation queue, decrements the idle thread counter,9 and wakes up a slave thread. The awoken slave thread will find the branch there, take it out, and start working again. From there on, the core enters its normal thread-to-thread mode of work sharing. Conversely, when a Platypus process receives notification that another process has run out of work, it attempts to delegate a piece of its search space. To this end, it sets the other-process-needs-work flag of the master thread in its core object. All slave threads noticing this flag clear the flag and delegate a piece of their search space according to the delegation policy by adding it to the remote delegation queue. The master thread takes one branch out of the queue and forwards it to the requesting Platypus process (via the distribution object). Because of the multi-threaded nature any number of threads can end up delegating. Items left in the remote delegation queue are used by the master thread to fulfil subsequent requests for work by other Platypus processes or work requests by its slave threads. The conceptual difference between the thread delegation and the remote delegation queues is that the former handle intra-core delegations, while the latter deal with extra-core delegation, although non-delegated work can return to the core. This is reflected by the fact that master and slave threads are allowed to insert partial assignments into the thread delegation queue, whereas only slave threads remove items from this queue. In contrast, only the master thread is allowed to eliminate items from the remote delegation queue, while insertions are performed only by slave threads.

An important aspect of the multi-threaded core implementation is the use of lock-free data structures for synchronising communication among master and slave threads. This is detailed in [6].

5 EXPERIMENTAL RESULTS

The following experiments aim to provide some indication of the computational value of probing and multi-threading. All experiments were conducted with some fixed parameters: (i) smodels (2.28) was used as propagation engine and for delivering the (signed) choice in Line 6 of Algorithm 1, (ii) the choice in Line 1 of Algorithm 1 was

---

8 Option \#n can be zero, indicating the use of all branches.

9 The inserting thread is responsible for decrementing the idle thread counter.
fixed to the problem of selecting assignments with the largest number of unassigned atoms, (iii) all such selections were done in a deterministic way by setting command-line option #n to 1 (cf. Section 2). All tests were done with plattypus version 0.2.2 [9]. They reflect average times of 5 runs for finding the first or all answer sets, resp., of the considered instances. Timing excludes parsing and printing. The data was obtained on a quad processor (4 Opteron 2.2GHz processors, 8 GB shared RAM) under Linux.

For illustrating the advantage of probing, we have chosen the search for one Hamiltonian cycle in clumpy graphs, proposed in [13] as a problem set being problematic for systematic backtracking. These benchmarks are available at [9]. Table 1 contrasts different settings for numbers of conflicts #c (10, 50, 100, 200) and jumps #j (32, 64, 128, 256, 512), resp., running the single-threaded core. For comparison, we also provide the corresponding standard times 10 [9] and the ones for single-threaded plattypus without probing in the columns labelled sm and sr. The remaining columns are labelled with the command line options used, viz. #c, #j. A blank entry represents a timeout after 240 seconds. First of all, we notice that the systems using standard depth-first-search are unable to solve 12 instances within the given time limit, whereas when using probing, apart from a few exceptions, all instances are solved. We see that plattypus without probing does best 8 times, as indicated in boldface, and worst 24 times, whereas standard times all give worst 24 times. Compared to each specific probing configuration, plattypus without probing performs better among 9 to 15 (smalls, 6 to 8) times out of 38. In fact, there seems to be no clear pattern indicating a best probing configuration. However, looking at the lower part of Table 1, we observe that plattypus without probing (smalls) times out 12 times, while probing still gives a solution under all but three configurations. In all, we see that probing allows for a significant speed-up for finding the first answer set. This is particularly valuable whenever answer sets are hard to find with a systematic backtracking procedure, as witnessed by the entries in the lower part of Table 1. However, probing has generally no positive effect when computing all answer sets. Also, on more common benchmarks (cf. [1]) probing rarely kicks in since the conflict counter is earlier reset to zero whenever an answer set is found.

The computational impact of probing is even more significant when using multi-threading, 12 where further speed-ups are observed on 20 benchmarks, most of which are among the more substantial ones in the lower part of Table 1. The most substantial one is observed on clumpy graph 09.09.04 which is solved in 4.66 and 4.26 seconds, resp., when setting #c, #j to 10,512 and using 4 and 8 slave threads, resp. Interestingly, even the multi-threaded variant without probing cannot solve the last seven benchmarks within the time limit, except for clumpy 09.09.07, which plattypus with 4 slave threads is able to solve in 13.8 seconds. This illustrates that probing and multi-threading are two complementary techniques that can be used for accelerating the performance of standard ASP-solvers. A way to tackle benchmarks that are even beyond the reach of probing with multi-threading is to use randomisation via command-line option #n.

Table 2 displays the effect of multi-threading, when computing all answer sets. For consistency, we have taken a subset of the asparagus benchmarks [1] in [5], used when evaluating the speed-ups obtained with the (initial) forking and MPI variant of plattypus. Comparing the sum of the average times, the current plattypus variant running multi-threading is 2.64 times faster than its predecessor using forking, as reported in [5]. 13 More in detail, the columns reflect the times of plattypus run with the multi-threaded core restricted

---

10 These times are only indicative since they include printing one answer set.
11 The six cases differ by only 0.01sec which is due to slightly different timing methods (see Footnote 10).

12 All tests on multi-threaded with and without probing are provided at [9].
13 The forking tests in [5] were also run on the same machine.
to 1, 2, 3, and 4 slave threads (probing disabled). When looking at each benchmark, the 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. For instance, the smallest speed-up is observed on schur-11-5 (1.52, 1.73, 1.75); among the highest speed-ups, we find schur-19-4 (2.17, 3.43, 4.75) and pigeon-7-11 (2.24, 3.43, 4.6). The average speed-ups observed on this set of benchmarks is 1.96, 2.89, and 3.75. If we weight the average speed-ups with the respective average running times, we obtain even a slightly super-linear speed-up: 2.07, 3.18, 4.24. Such super-linear speed-ups are observed primarily on time-demanding benchmark, the more clear-cut the speed-up.

6 DISCUSSION

At the heart of the PLATYPUS design is its generality and modularity. These two features allow a great deal of flexibility in any instantiation of the algorithm, making it unique among related approaches. Up to now, this flexibility was witnessed by the possibility to use different off-the-shelf solvers, different process-oriented distribution mechanisms, and a variety of choice policies. In this paper we have presented two significant configurable enhancements to platypus. First, we have described its probing mode, relying on an explicit yet restricted representation of the search space. This provides us with a global view of the search space and allows us to have different threads working on different subspaces. Although probing does not primarily aim at a sequential setting, we have experimentally demon-

<table>
<thead>
<tr>
<th>problem</th>
<th>mt #1</th>
<th>mt #2</th>
<th>mt #3</th>
<th>mt #4</th>
</tr>
</thead>
<tbody>
<tr>
<td>color-5-10</td>
<td>1.53</td>
<td>0.84</td>
<td>0.62</td>
<td>0.53</td>
</tr>
<tr>
<td>color-5-15</td>
<td>0.93</td>
<td>1.19</td>
<td>0.88</td>
<td>0.73</td>
</tr>
<tr>
<td>ham_comp_8</td>
<td>3.56</td>
<td>1.99</td>
<td>1.38</td>
<td>1.01</td>
</tr>
<tr>
<td>ham_comp_9</td>
<td>85.2</td>
<td>43.6</td>
<td>29.0</td>
<td>22.5</td>
</tr>
<tr>
<td>pigeon-7-8</td>
<td>1.38</td>
<td>0.73</td>
<td>0.57</td>
<td>0.48</td>
</tr>
<tr>
<td>pigeon-7-9</td>
<td>4.22</td>
<td>2.19</td>
<td>1.46</td>
<td>1.17</td>
</tr>
<tr>
<td>pigeon-7-10</td>
<td>13.2</td>
<td>6.31</td>
<td>4.12</td>
<td>3.08</td>
</tr>
<tr>
<td>pigeon-7-11</td>
<td>36.5</td>
<td>16.3</td>
<td>10.6</td>
<td>7.94</td>
</tr>
<tr>
<td>pigeon-7-12</td>
<td>86.2</td>
<td>39.9</td>
<td>25.8</td>
<td>19.0</td>
</tr>
<tr>
<td>pigeon-8-9</td>
<td>11.6</td>
<td>5.77</td>
<td>3.80</td>
<td>2.84</td>
</tr>
<tr>
<td>pigeon-8-10</td>
<td>48.5</td>
<td>22.3</td>
<td>14.2</td>
<td>10.4</td>
</tr>
<tr>
<td>pigeon-9-10</td>
<td>128.4</td>
<td>61.8</td>
<td>39.5</td>
<td>29.4</td>
</tr>
<tr>
<td>schur-14-4</td>
<td>1.00</td>
<td>0.63</td>
<td>0.47</td>
<td>0.42</td>
</tr>
<tr>
<td>schur-15-4</td>
<td>2.38</td>
<td>1.30</td>
<td>0.91</td>
<td>0.73</td>
</tr>
<tr>
<td>schur-16-4</td>
<td>4.04</td>
<td>2.14</td>
<td>1.41</td>
<td>1.11</td>
</tr>
<tr>
<td>schur-17-4</td>
<td>9.13</td>
<td>4.58</td>
<td>3.04</td>
<td>2.28</td>
</tr>
<tr>
<td>schur-18-4</td>
<td>16.7</td>
<td>8.34</td>
<td>5.31</td>
<td>3.92</td>
</tr>
<tr>
<td>schur-19-4</td>
<td>39.3</td>
<td>18.1</td>
<td>11.5</td>
<td>8.28</td>
</tr>
<tr>
<td>schur-20-4</td>
<td>44.1</td>
<td>21.9</td>
<td>13.8</td>
<td>10.1</td>
</tr>
<tr>
<td>schur-11-5</td>
<td>0.56</td>
<td>0.37</td>
<td>0.32</td>
<td>0.32</td>
</tr>
<tr>
<td>schur-12-5</td>
<td>1.49</td>
<td>0.83</td>
<td>0.63</td>
<td>0.54</td>
</tr>
<tr>
<td>schur-13-5</td>
<td>5.69</td>
<td>2.90</td>
<td>1.97</td>
<td>1.51</td>
</tr>
<tr>
<td>schur-14-5</td>
<td>18.6</td>
<td>9.05</td>
<td>6.00</td>
<td>4.42</td>
</tr>
</tbody>
</table>

Table 2. Experimental results on multi-threading.

strated its computational value on a specific class of benchmarks, which is problematic for standard ASP-solvers. Unlike restart strategies in SAT, which usually draw on learnt information [4], probing keeps previously abandoned parts of the search space, so that they can be revisited subsequently. Probing offers a non-linear exploration of the search space that can be randomised while remaining complete, a search strategy that no other native ASP-solver offers.

Second, we have presented platypus' multi-threaded architecture. Multi-threading complements the previous process-oriented distribution schemes of platypus by providing further intra-process distribution capacities. This is of great practical value since it allows us to take advantage of recent hardware developments, offering multi-core processors. In a hybrid setting, consisting of clusters of such machines, we may use multi-threading for distribution on the multi-core processors, while distribution among different workstations is done with previously established distribution techniques in platypus, like MPI. Furthermore, the modular implementation of the core and distribution component allow for easy modifications in view of new distribution concepts, like grid computing, for instance. The platypus platform is freely available on the web [9].

For more details and related work, we refer the reader to [6].

ACKNOWLEDGMENTS. Research at Potsdam was supported by DFG (SCHA 550/6-4), and at U.W.O. by NSERC (Canada) and SHARCNET. We are grateful to C. Anger, M. Brain, M. Gebser, B. Kaufmann, and the referees for many helpful suggestions.

REFERENCES


14 That is, the traversal of the search space does not follow a given strategy like depth-first search.