On Probing and Multi-Threading in PLATYPUS

J. Gressmann[1] and **T. Janhunen**[2] and **R. Mercer**[3] and **T. Schaub**[1,4] and **S. Thiele**[1] and **R. Tichy**[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 [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 [5].

The PLATYPUS approach differs from other pioneering work in distributed answer set solving [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 [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 `smodels`' and `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 [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 [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 [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 Π is associated with a set of *answer sets*, $AS(\Pi)$, which are distinguished models of the rules in Π. We do not elaborate, but refer the reader to [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 Π, 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 [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, 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

Algorithm 1: PLATYPUS

Global : A logic program Π over alphabet $\mathcal{A}$.
Input : A nonempty set S of partial assignments.
Output: Print a subset of the answer sets of Π.

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 $S = \emptyset$

[1] Universität 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.

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 in $\bigcup_{(X,Y)\in S} AS_{(X,Y)}(\Pi)$. 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 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 $(X, Y) = \text{EXPAND}((X_c, Y_c))$. 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 EXPAND component for implementing Line 6. Additionally, the currently used expanders, `smodels` and `nomore++`, also supply a *polarity*, indicating a preference for assigning true or false.

Each `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 `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).

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 `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 *probing*. Probing is made precise in Algorithm 2, which is a refinement of the 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 ($c \leq \#c$).

Algorithm 2: CHOOSE (in Line 1 of Algo. 1) in *probing* mode.

Global : Positive integers $\#c, \#j$, supplied via command line.
Integers c, j, initially $c = 0$ and $j = \#j$.
Selection policy $\mathcal{P}$, supplied via command line.
Input : A set S of assignments with active assignment $b \in S$.
Output: A partial assignment.

// Counter c is incremented by one in Line 4 of Algorithm 1.
if $(c \leq \#c)$ *or* $(\#j = 0)$ **then return** b ; *// no jumping*
else *// jumping*
  $c \leftarrow 0$
  $j \leftarrow j - 1$
  if $(j = 0)$ **then**
    $\#c \leftarrow (\#c \times 2)$
    $\#j \leftarrow (\#j \ div\ 2)$
    $j \leftarrow \#j$
  let $b' \leftarrow \text{SELECT}(\mathcal{P}, S)$ **in**
    $\langle$*make b' the active partial assignment in S*$\rangle$
    return b'

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 $\mathcal{P}$ (at present, a largest, a smallest, or a random assignment). If this, too, is impossi-

[5] Assignments are not restricted to atoms, as used when using `nomore++`.
[6] Accordingly, the tests in Lines 4 and 5 must be handled with care; see [5].
[7] Each thread has its own conflict and jump counters.

ble, S must be empty and the PLATYPUS instance terminates.

The policy-driven selection of a branch, expressed in Algorithm 2 by SELECT($\mathcal{P}, S$), is governed by another command line option[8] $\#n$ and works in two steps. First, among all branches in S, the $\#n$ best ones, $b_1, \ldots, b_{\#n}$, are identified according to policy $\mathcal{P}$. To be precise, let p be a mapping of branches to ordinal values, used by $\mathcal{P}$ for evaluating branches. For every $b \in \{b_1, \ldots, b_{\#n}\}$ and $b' \in S \setminus \{b_1, \ldots, b_{\#n}\}$, we then have that $p(b) \leq p(b')$. Then, a branch b is randomly selected from $\{b_1, \ldots, b_{\#n}\}$. This random selection from the best $\#n$ branches counteracts the effect of a rigid policy by arbitrarily choosing some close alternatives.

To see that probing guarantees completeness, it is sufficient to see that no partial assignment is ever eliminated from the search space. Also, when probing, the number of different branches in the search space S cannot exceed twice the number of initially permitted jumps, viz. $2 \times \#j$. For instance, if the command line option sets $\#j$ to 13, we may develop at most $13+6+3+1$ different branches in S, which is bound by 2×13. Thereby, a branch is considered as different if it is not obtainable from another's subbranch by switching the assigned truth value of a single atom (i.e. if it is not a backtracking point).

4 THREAD ARCHITECTURE

This section details the multi-threaded architecture extension to the `platypus` platform which adds the capacity to do intra-process distribution delegation to the existing inter-process capabilities, which are optionally realised via Unix' forking mechanism or MPI [7] (described in [5]). This richer architecture now permits hybrid delegation methods, for instance, delegating `platypus` via MPI on a cluster of multi-processor machines, with delegation among the multi-processors of each machine accomplished with multi-threading.

The architecture is split into more or less two parts: the *core* and the *distribution* components. A core encapsulates the search for answer sets, and the DELEGATE function is encapsulated in a distribution component. The core and distribution components have well-defined interfaces that localise the communication between the components. This design allows us to incorporate, for instance, single- and multi-threaded cores, as well as inter-process distribution schemes, like MPI and forking, with ease.

Each `platypus` process hosts an instance of the core, the core object, which cooperates with one instance of the distribution component, the distribution object. Communication is directed from core to distribution objects and is initiated by the core object. During execution the major flow of control lies with the core objects.

The multi-threaded core works according to the master/slave principle. The master coordinates a number of slave threads. Each slave thread executes the PLATYPUS algorithm on its thread-local search space. The master thread handles communication (through the distribution object) with other `platypus` processes on behalf of the slave threads. Communication between the master thread and its slave threads is based on counters and queues: Events of interest (e.g. statistics, answer sets, etc.) are communicated by the slaves to the master by incrementing the appropriate counter or adding to the respective queue. The master thread periodically polls the counters and queues for any change. The search ends (followed by termination of the `platypus` program) if there is agreement among the distribution objects that either all participating processes are in need of work (indicating all the work is done) or the requested number of answer sets is computed. In the core, the *idle thread counter* of the master thread serves two purposes: It indicates the number of idle slave 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.

clumpy	*sm*	*st*	10,32	10,64	10,128	10,256	10,512	50,32	50,64	50,128	50,256	50,512	100,32	100,64	100,128	100,256	100,512	200,32	200,64	200,128	200,256	200,512
06,06,02	0.01	**0.01**	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01
06,06,03	0.10	0.10	0.05	0.05	0.05	0.05	0.05	0.07	0.07	0.07	0.07	0.07	0.11	0.11	0.11	0.11	0.11	0.17	0.16	0.16	0.16	0.16
06,06,04	0.61	0.63	0.08	0.08	0.08	0.08	0.08	0.14	0.14	0.14	0.14	0.14	0.24	0.24	0.24	0.24	0.24	0.34	0.34	0.34	0.34	0.34
06,06,05	6.30	6.61	1.24	1.79	0.95	0.84	0.84	0.78	0.66	0.66	0.66	0.66	0.96	0.96	0.96	0.96	0.96	2.29	2.14	2.14	2.14	2.14
06,06,06	0.38	0.39	0.05	0.05	0.05	0.05	0.05	0.04	0.04	0.04	0.04	0.04	0.06	0.06	0.06	0.06	0.06	0.10	0.10	0.10	0.10	0.10
06,06,07	0.04	**0.03**	0.14	0.14	0.14	0.14	0.14	0.08	0.08	0.08	0.08	0.08	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03
06,06,08	0.08	0.08	0.01	0.01	0.01	0.01	0.01	0.02	0.02	0.02	0.02	0.02	0.02	0.02	0.02	0.02	0.02	0.03	0.03	0.03	0.03	0.03
06,06,09	11.3	11.8	0.47	0.52	0.62	0.62	0.62	1.07	1.01	1.01	1.01	1.01	2.23	2.06	2.06	2.06	2.06	3.06	3.46	3.46	3.46	3.46
06,06,10	0.06	0.05	0.03	0.03	0.03	0.03	0.03	0.02	0.02	0.02	0.02	0.02	0.03	0.03	0.03	0.03	0.03	0.05	0.05	0.05	0.05	0.05
07,07,01	0.02	**0.01**	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01
07,07,02	0.05	**0.04**	0.61	0.74	0.71	0.71	0.71	1.76	1.45	1.45	1.45	1.45	2.01	2.92	2.91	2.91	2.90	0.04	0.04	0.04	0.04	0.04
07,07,03	8.98	9.60	18.7	9.56	14.5	3.75	3.26	4.79	4.72	16.9	6.11	6.05	5.02	33.8	18.4	9.71	10.3	23.3	9.75	22.1	14.5	14.5
07,07,04	1.37	1.38	0.98	2.05	2.01	3.49	3.38	1.57	1.79	1.54	1.54	1.53	2.87	2.19	2.19	2.20	2.19	2.76	3.30	3.30	3.30	3.28
07,07,05	0.03	**0.02**	0.04	0.04	0.04	0.04	0.04	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.02	0.02	0.02	0.02	0.02
07,07,06	**0.38**	**0.38**	0.41	0.38	0.38	0.38	0.38	0.61	0.61	0.61	0.61	0.61	0.69	0.69	0.69	0.69	0.69	0.86	0.86	0.86	0.86	0.86
07,07,07	0.04	**0.03**	0.08	0.08	0.08	0.08	0.08	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03
07,07,08	0.11	0.11	0.11	0.11	0.11	0.11	0.11	0.10	0.10	0.10	0.10	0.10	0.10	0.10	0.10	0.10	0.10	0.14	0.14	0.14	0.14	0.14
07,07,09	0.40	0.40	0.08	0.08	0.08	0.08	0.08	0.35	0.35	0.35	0.35	0.35	0.35	0.35	0.35	0.35	0.35	0.55	0.55	0.55	0.55	0.55
07,07,10	124.5	126.4	15.8	6.32	2.17	1.96	1.97	31.7	13.4	6.01	5.27	5.27	59.3	72.0	9.49	8.74	8.74	18.8	21.5	20.4	14.1	14.1
08,08,01			5.07	1.64	2.44	4.68	5.23	22.5	2.84	3.21	3.22	3.20	10.9	4.81	4.76	4.72	4.68	45.1	15.4	10.3	10.2	10.0
08,08,02			7.04	11.1	2.42	2.44	2.43	8.01	6.22	5.61	6.64	6.61	23.0	12.0	9.74	9.05	8.98	44.0	15.5	13.7	13.8	13.7
08,08,03			14.8	9.39	13.1	5.31	5.52	61.9	84.9	7.57	14.0	13.1	105.8	51.8	9.17	8.71	8.66	32.8	205.8	15.9	15.3	15.3
08,08,05	36.7	37.0	231.2		16.1	33.6	43.6	176.6	24.1	36.1	53.5	96.5	48.3	29.2	47.7	84.1	129.2	70.0	39.4	87.3	189	240
08,08,06	8.15	8.22	0.05	0.05	0.05	0.05	0.05	0.10	0.10	0.10	0.10	0.10	0.16	0.17	0.17	0.17	0.16	0.26	0.26	0.26	0.26	0.26
08,08,07	4.17	4.10	0.44	0.44	0.44	0.44	0.43	1.23	1.24	1.23	1.23	1.23	0.48	0.48	0.48	0.48	0.47	0.89	0.90	0.90	0.90	0.89
08,08,08			0.85	71.6	14.5	6.33	13.5	2.16	1.73	1.73	1.72	1.72	3.69	2.77	2.77	2.77	2.76	6.40	4.76	4.76	4.77	4.75
08,08,09			1.29	0.87	0.88	0.88	0.87	1.07	1.08	1.08	1.08	1.07	2.03	2.03	2.03	2.03	2.02	3.02	3.04	3.03	3.03	3.02
08,08,10	**1.66**	**1.67**	17.3	11.5	4.24	4.37	4.02	1.87	2.24	2.24	2.24	2.23	4.93	2.72	2.72	2.72	2.72	5.97	7.41	7.41	7.40	7.37
09,09,01	24.9	25.0	0.34	0.34	0.34	0.34	0.34	0.10	0.10	0.10	0.10	0.10	0.11	0.11	0.11	0.11	0.11	0.12	0.12	0.12	0.12	0.12
09,09,02			1.66	1.82	2.84	2.64	2.63	0.85	0.85	0.85	0.85	0.84	1.48	1.49	1.49	1.49	1.48	2.31	2.32	2.33	2.32	2.31
09,09,03			13.3	4.24	7.33		74.3	0.82	0.82	0.82	0.82	0.82	1.67	1.68	1.68	1.68	1.68	2.51	2.52	2.52	2.52	2.51
09,09,04			143.8				50.9					81.6					95.7					
09,09,05			2.60	2.08	2.66	2.66	2.66	4.03	3.98	4.68	4.68	4.67	3.96	4.80	4.81	4.80	4.79	6.49	6.32	6.31	6.33	6.31
09,09,06			4.00	2.59	159.6	6.40	5.89	11.5	8.62	5.51	5.51	5.50	7.35	21.5	6.45	6.46	6.44	12.8	20.1	17.4	17.4	17.4
09,09,07			0.75	28.4	3.23	3.01	3.01	2.16	2.03	2.04	2.03	2.03	3.05	3.07	3.07	3.06	3.05	6.70	5.95	5.95	5.95	5.90
09,09,09			0.73	0.71	0.71	0.71	0.71	1.95	2.40	2.40	2.40	2.39	3.91	3.50	3.51	3.50	3.48	12.5	9.68	9.67	9.69	9.63

Table 1. Experimental results for *probing* (with the single-threaded core).

fixed to the policy 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 `platypus` version 0.2.2 [9]. They reflect average times of 5 runs for finding the first or all answer sets, resp., of the considered instance. 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 `smodels` times[10] and the ones for single-threaded `platypus` without probing in the columns labelled *sm* and *st*. 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 for a few exceptions, all instances are solved. We see that `platypus` without probing does best 8 times, as indicated in boldface, and worst 24 times, whereas `smodels` does best 2 times[11] and worst 24 times. Compared to each specific probing configuration, `platypus` without probing performs better among 9 to 15 (`smodels`, 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 `platypus` without probing (`smodels`) 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 3 and 4 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 `platypus` 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 `platypus`. Comparing the sum of the average times, the current `platypus` variant running multi-threading is 2.64 times faster than its predecessor using forking, as reported in [5].[13] In more detail, the columns reflect the times of `platypus` 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-threading with and without probing are provided at [9].

[13] The forking tests in [5] were also run on the same machine.

problem	mt #1	mt #2	mt #3	mt #4
color-5-10	1.53	0.84	0.62	0.53
color-5-15	60.9	31.1	20.5	15.7
ham_comp_8	3.66	1.99	1.38	1.10
ham_comp_9	85.2	43.6	29.0	22.5
pigeon-7-8	1.38	0.73	0.57	0.48
pigeon-7-9	4.22	2.19	1.46	1.17
pigeon-7-10	13.2	6.31	4.12	3.08
pigeon-7-11	36.5	16.3	10.6	7.94
pigeon-7-12	88.2	39.9	25.8	19.0
pigeon-8-9	11.6	5.77	3.80	2.84
pigeon-8-10	48.3	22.3	14.2	10.4
pigeon-9-10	128.4	61.8	39.5	29.4
schur-14-4	1.00	0.63	0.47	0.42
schur-15-4	2.38	1.30	0.91	0.73
schur-16-4	4.04	2.14	1.41	1.11
schur-17-4	9.13	4.58	3.04	2.28
schur-18-4	16.7	8.34	5.31	3.92
schur-19-4	39.3	18.1	11.5	8.28
schur-20-4	44.1	21.9	13.8	10.1
schur-11-5	0.56	0.37	0.32	0.32
schur-12-5	1.49	0.83	0.63	0.54
schur-13-5	5.69	2.90	1.97	1.51
schur-14-5	18.6	9.05	6.00	4.42

Table 2. Experimental results on *multi-threading*.

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 benchmarks and, although less significant, have also been observed in [5] when forking (which makes us ascribe them to caching effects and/or shared memory). In all, we observe that the more substantial the benchmark, the more clear-cut the speed-up.

Given that the experiments were run on a quad processor, it is worth noting that we observe no drop in performance when increasing the number of slave threads from 3 to 4, despite having a fifth (master) thread. Finally, we note that the multi-threaded core, when restricted to a single slave thread, loses on average only 2% performance compared to the single-threaded version.

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 demonstrated 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[14] 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

[1] `http://asparagus.cs.uni-potsdam.de`.
[2] C. Baral, *Knowledge Representation, Reasoning and Declarative Problem Solving*, Cambridge University Press, 2003.
[3] R. Finkel, V. Marek, N. Moore, and M. Truszczynski, 'Computing stable models in parallel', in *Proc. of AAAI Spring Symposium on Answer Set Programming (ASP'01)*, eds., A. Provetti and T. Son, pp. 72–75. AAAI/MIT Press, (2001).
[4] C. Gomes, B. Selman, and H. Kautz, 'Boosting combinatorial search through randomization', in *Proc. of the Fifteenth National Conference on Artificial Intelligence (AAAI'98)*, pp. 431–437. AAAI Press, (1998).
[5] J. Gressmann, T. Janhunen, R. Mercer, T. Schaub, S. Thiele, and R. Tichy, 'Platypus: A platform for distributed answer set solving', in *Proc. of the Eighth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'05)*, eds., C. Baral, G. Greco, N. Leone, and G. Terracina, pp. 227–239. Springer-Verlag, (2005).
[6] J. Gressmann, T. Janhunen, R. Mercer, T. Schaub, S. Thiele, and R. Tichy, 'On probing and multi-threading in platypus', in *Proc. of the Eleventh International Workshop on Nonmonotonic Reasoning*, eds., J. Dix and A. Hunter, (2006). To appear.
[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 TOCL*, (2006). To appear.
[9] `http://www.cs.uni-potsdam.de/platypus`.
[10] E. Pontelli, M. Balduccini, and F. Bermudez, 'Non-monotonic reasoning on beowulf platforms', in *Proc. of the Fifth International Symposium on Practical Aspects of Declarative Languages (PADL'03)*, eds., V. Dahl and P. Wadler, pp. 37–57. Springer-Verlag, (2003).
[11] P. Simons, I. Niemelä, and T. Soininen, 'Extending and implementing the stable model semantics', *Art. Intell.*, **138**(1-2), 181–234, (2002).
[12] A. S. Tanenbaum, *Modern Operating Systems*, Prentice Hall, 2001.
[13] J. Ward and J. Schlipf, 'Answer set programming with clause learning', in *Proc. of the Seventh International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'04)*, eds., V. Lifschitz and I. Niemelä, pp. 302–313. Springer-Verlag, (2004).

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