TLP **12** (4–5): 525–545, 2012. © Cambridge University Press 2012
doi:10.1017/S1471068412000166

Multi-threaded ASP solving with clasp

MARTIN GEBSER, BENJAMIN KAUFMANN
and TORSTEN SCHAUB*
Institut für Informatik, Universität Potsdam

Abstract

We present the new multi-threaded version of the state-of-the-art answer set solver *clasp*. We detail its component and communication architecture and illustrate how they support the principal functionalities of *clasp*. Also, we provide some insights into the data representation used for different constraint types handled by *clasp*. All this is accompanied by an extensive experimental analysis of the major features related to multi-threading in *clasp*.

1 Introduction

The increasing availability of multi-core technology offers a great opportunity for further improving the performance of solvers for Answer Set Programming (ASP; Baral 2003). This paper describes how we redesigned and reimplemented the award-winning[1] ASP solver *clasp* (Gebser *et al.* 2007b) in order to leverage the power of today's multi-core shared memory machines by supporting parallel search. To this end, we chose a coarse-grained, task-parallel approach via shared memory multi-threading. This has led to the *clasp 2* series supporting a single- and a multi-threaded variant sharing a common code base. *clasp* allows for parallel solving by search space splitting and/or competing strategies. While the former involves dynamic load balancing in view of highly irregular search spaces, both modes aim at running searches as independently as possible in order to take advantage of enhanced sequential algorithms. In fact, a portfolio of solver configurations cannot only be used for competing but also in splitting-based search. The latter is optionally combined with global restarts to escape from uninformed initial splits.

For promoting the scalability of parallel search, all major routines of *clasp 2* are lock-free. Also, we enforced a clear distinction between read-only, shared, and thread-local data and incorporated accordingly optimized representations. This is implemented by means of Intel's Threading Building Blocks (TBB) for providing platform-independent threads, atomics, and concurrent containers. Currently, *clasp*

* Affiliated with Simon Fraser University, Canada, and Griffith University, Australia.

[1] The multi-threaded variant of *clasp 2* won the first place in the *Crafted/UNSAT* and the second place in the *Crafted/SAT+UNSAT* category, respectively, at the *2011 SAT competition* in terms of number of solved instances and wall-clock time. In addition, *clasp 2* was among the three genuine parallel solvers participating in the *32 cores track* (restricted to benchmarks from the *Application* category; the fourth solver used a portfolio, including *clasp 1.3*). Also, *clasp 2* participated "out of competition" at the *2011 ASP competition*, which was dominated by the single-threaded variant of *clasp 2*.

https://doi.org/10.1017/S1471068412000166 Published online by Cambridge University Press

supports up to 64 configurable (non-hierarchic) threads. Apart from parallel search, another major extension of previous versions of *clasp* regards the exchange of recorded nogoods. While unary, binary, and ternary nogoods are always shared among all threads, longer ones can optionally be exchanged, configurable at the sender as well as at the receiver side. In fact, *clasp* provides different measures estimating the quality of shared nogoods as well as various heuristics and filters for controlling their integration. For instance, the sharing of a nogood can be subject to the number of distinct decision levels associated with its literals. Conversely, the integration of a nogood may depend on its satisfaction and/or scores in host heuristics.

In view of the wide distribution of *clasp*, we put a lot of effort into transferring the entire functionality from the sequential, viz. *clasp* series 1.3, to the parallel setting. For one, this concerned *clasp*'s reasoning modes (cf. Gebser *et al.* 2011a), including enumeration, projected enumeration, intersection and union of models, and optimization. Moreover, we extended *clasp*'s language capacities by allowing for solving weighted and/or partial MaxSAT (Li and Manyà 2009) as well as Boolean optimization (Marques-Silva *et al.* 2011) problems. Finally, it goes without saying that *clasp*'s basic infrastructure has also significantly evolved with the new design; e.g. the preprocessing capacities of *clasp* were extended with blocked clause elimination (Järvisalo *et al.* 2010), and its conflict analysis has been significantly improved by on-the-fly subsumption (Han and Somenzi 2009).

In what follows, we focus on describing the multi-threaded variant of *clasp 2*. To this end, the next section provides a high-level view on modern parallel ASP solving. The general component and communication architecture of the new version of *clasp* are presented in Sections 3 and 4. Section 5 details the design of data structures underlying the implementation of *clasp 2*. Parallel search features of *clasp 2* are empirically assessed in Section 6. Finally, Sections 7 and 8 discuss related work and the achieved results, respectively.

2 Parallel ASP solving

We presuppose some familiarity with search procedures for (Boolean) constraint solving, that is, Davis-Putnam-Logemann-Loveland (DPLL; Davis and Putnam 1960; Davis *et al.* 1962) and Conflict-Driven Constraint Learning (CDCL; Marques-Silva and Sakallah 1999; Zhang *et al.* 2001). In fact, (sequential) ASP solvers like *smodels* (Simons *et al.* 2002) adopt the search pattern of DPLL based on systematic chronological backtracking, or like *clasp* (series 1.3) apply lookback techniques from CDCL, which include conflict-driven learning and non-chronological backjumping. In what follows, we primarily concentrate on CDCL and principal points for its parallelization in the *clasp 2* series.

In order to solve the basic decision problem of solution existence, CDCL first extends a given (partial) *assignment* via deterministic (unit) propagation. Importantly, every derived literal is "forced" by some *nogood* (set of literals that must not jointly be assigned), which would be violated if the literal's complement were assigned. Although propagation aims at forgoing nogood violations, assigning

https://doi.org/10.1017/S1471068412000166 Published online by Cambridge University Press

```
while work available
    while no (result) message to send
        communicate                          // exchange information with other solver instances
        propagate                            // deterministically assign literals
        if no conflict then
            if all variables assigned then send solution
            else decide                      // non-deterministically assign some literal
        else
            if root-level conflict then send unsatisfiable
            else if external conflict then send unsatisfiable
            else
                analyze                      // analyze conflict and add conflict constraint
                backjump                     // unassign literals until conflict constraint is unit
    communicate          // exchange results with (and receive work from) other solver instances
```

Fig. 1. High-level algorithm for multi-threaded Conflict-Driven (Boolean) Constraint Learning.

a literal forced by one nogood may lead to the violation of another nogood; this situation is called *conflict*. If the conflict can be resolved (the violated nogood contains backtrackable literals), it is analyzed to identify a conflict constraint. The latter represents a "hidden" conflict reason that is recorded and guides backjumping to an earlier stage such that the complement of some formerly assigned literal is forced by the conflict constraint, thus triggering propagation. Only when propagation finishes without conflict, a (heuristically chosen) literal can be assigned at a new *decision level*, provided that the assignment at hand is partial, while a *solution* (total assignment not violating any nogood) has been found otherwise. The eventual termination of CDCL is guaranteed (cf. Zhang and Malik 2003; Ryan 2004), by either returning a solution or encountering an unresolvable conflict (independent of unforced decision literals).

Figure 1 provides a high-level view on the parallelization of CDCL-style search in *clasp*. We first note that entering the inner search loop relies on the availability of work. In fact, when search spaces to investigate in parallel are split up by means of *guiding paths* (Zhang *et al.* 1996), a solver instance must acquire some spare guiding path before it can start to search. In this case, all (decision) literals of the guiding path are assigned up to the solver's *root level*, precluding them from becoming unassigned upon backtracking/backjumping. Apart from search space splitting, parallelization of *clasp* can be based on *algorithm portfolios* (Gomes and Selman 2001), running different solving strategies competitively on the same search space. Once a solver instance is working on some search task, it combines deterministic propagation with communication. The latter includes nogood exchange with other solver instances, work requests from idle solvers (asking for a guiding path), and

https://doi.org/10.1017/S1471068412000166 Published online by Cambridge University Press

external conflicts raised to abort the current search.[2] An external conflict or an (unresolvable) root-level conflict likewise make a solver instance stop its current search, and the same applies when a solution is found. In such a case, the respective result is communicated (in the last line of Fig. 1), and a new search task may be received in turn.

As mentioned in the introductory section, the infrastructure of *clasp* also allows for conducting sophisticated reasoning modes like enumeration and optimization in parallel. This is accomplished via enriched message protocols, e.g. (upper) bounds are exchanged in addition to nogoods when performing parallel optimization, while an external conflict (raised upon finding the first solution) switches competing solvers of an algorithm portfolio into enumeration mode based on guiding paths. In fact, search space splitting and algorithm portfolios can be applied exclusively or be combined to flexibly orchestrate parallel solvers.

In the following sections, we detail the parallel architecture and underlying implementation techniques of *clasp 2*. Regarding data structures, it is worthwhile to note that unit propagation over "long" nogoods (involving more than three literals) relies on a *two-watched-literals* approach (Moskewicz *et al.* 2001), monitoring two references to unassigned literals for triggering propagation once the second last literal becomes assigned. We also presuppose basic familiarity with parallel computing concepts, such as race conditions, atomic operations, (dead- and spin-) locks, semaphores, etc. (cf. Herlihy and Shavit 2008).

3 Component architecture

To explain the architecture and functioning of the new version of *clasp*, let us follow the workflow underlying its design. To this end, consider *clasp*'s architectural diagram given in Figure 2. Although *clasp* also accepts other input formats, like (extended) *dimacs*, *opb*, and *wbo* for describing Boolean satisfiability (SAT; Biere *et al.* 2009) and optimization problems, we detail its functioning for computing answer sets of (propositional) logic programs, as output by grounders like *gringo* (Gebser *et al.* 2011a) or *lparse* (Syrjänen). Similarly, we concentrate on the multi-threaded setting, neglecting the single-threaded one.

At the start, only the main thread is active. Once the logic program is read in, it is subject to several preprocessing stages, all conducted by the main thread. At first, the program is (by default) simplified while identifying equivalences among its constituents (Gebser *et al.* 2008). The simplified program is then transformed into a compact representation in terms of Boolean constraints (whose core is generated from the completion (Clark 1978) of the simplified program). After that, the constraints are (optionally) subject to further, mostly SAT-based preprocessing (Eén and Biere 2005; Järvisalo *et al.* 2010). Such techniques are more involved in our ASP setting because variables relevant to unfounded-set checking, optimization, or

[2] For instance, a solver instance may discover unconditional unsatisfiability (even when using guiding paths; cf. Ellguth *et al.* 2009) and then inform others about the needlessness of performing further work.

https://doi.org/10.1017/S1471068412000166 Published online by Cambridge University Press

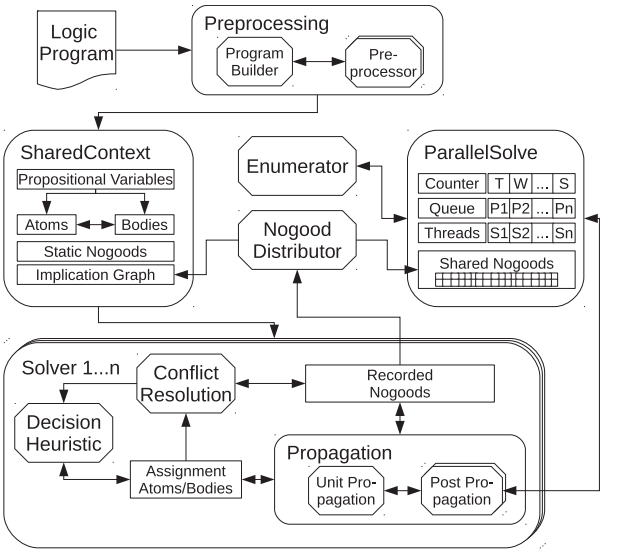


Fig. 2. Multi-threading architecture of *clasp 2*.

part of complex (i.e. cardinality and weight) constraints cannot be simply eliminated. Note that both preprocessing steps identify redundant variables that can be expressed in terms of the relevant ones included in the resulting set of constraints.

The outcomes of the preprocessing phase are stored in a SharedContext object that is initialized by the main thread and shared among all participating threads. Among others, this object contains

- the set of relevant Boolean variables together with type information (e.g. atom, body, aggregate, etc.),
- a symbol table, mapping (named) atoms from the program to internal variables,
- the positive atom-body dependency graph, restricted to its strongly connected components,
- the set of Boolean constraints, among them nogoods, cardinality and weight constraints, minimize constraints, and
- an implication graph capturing inferences from binary and ternary nogoods.[3]

The richness of this information is typical for ASP, and it is much sparser in a SAT setting.

After its initialization in association with a "master solver," further (solver) threads are (concurrently) attached to the SharedContext, where its constraints are "cloned." Notably, each constraint is aware of how to clone itself efficiently (cf. Section 5 on implementation details). Moreover, the Enumerator and NogoodDistributor objects are used globally in order to coordinate various model enumeration modes and nogood exchange among solver instances. We detail their functioning in Section 4.

[3] ASP problems usually yield a large majority of binary nogoods due to program completion (Clark 1978). Also note that unary nogoods capture initial problem simplifications that need not be rechecked during search.

https://doi.org/10.1017/S1471068412000166 Published online by Cambridge University Press

Each thread contains one Solver object, implementing the algorithm in Figure 1. Each Solver stores

- local data, including assignment, watch lists, constraint database, etc.,
- local strategies, regarding heuristics, restarts, constraint deletion, etc.,

and it uses the NogoodDistributor to share recorded nogoods. A solver assigns variables either by (deterministic) propagation or (non-deterministic) decisions. Motivated by the nature of ASP problems,[3] each solver propagates first binary and ternary nogoods (shared through the aforementioned implication graph), then longer nogoods and other constraints, before it finally applies any available post propagators.

Post propagators constitute another important new feature of *clasp 2*, providing an abstraction easing *clasp*'s extensibility with more elaborate propagation mechanisms. For this, each solver maintains a list of post propagators that are consecutively processed after unit propagation. For instance, failed-literal detection and unfounded-set checking are implemented in *clasp 2* as post propagators. Similarly, they are used in the new version of *clasp*'s extension with constraint processing, *clingcon* (Gebser *et al.* 2009), to realize theory propagation. Post propagators are assigned different priorities and are called in priority order. Typically, we distinguish three priority classes:

- *single* post propagators are deterministic and only extend the current decision level.
  Unfounded-set checking is a typical example.
- *multi* post propagators are deterministic and may add or remove decision levels.
  Failed-literal detection is a typical example.
- *complex* post propagators may or may not be deterministic.
  Nogood exchange is an example for this (see below).

Moreover, parallelism is also handled by means of post propagators, as described next.

ParallelSolve controls concurrent solving with up to 64 individually configurable threads. When attaching a solver to the SharedContext, ParallelSolve associates a thread with the solver and adds dedicated post propagators to it. One high-priority post propagator is added for message handling and another, very low-priority post propagator is supplied for integrating information stemming from models[4] and/or shared nogoods.

For controlling parallel search, ParallelSolve maintains a set of atomic message flags:

- *terminate* signals the end of a computation,
- *interrupt* forces outside termination (e.g. when the user hits Ctrl+C),

[4] This can regard an enumerated model to exclude, intersect, or union, as well as objective function values.

https://doi.org/10.1017/S1471068412000166 Published online by Cambridge University Press

- *sync* indicates that all threads shall synchronize, and
- *split* is set during splitting-based search whenever at least one thread needs work.

These flags are used to implement *clasp*'s two major search strategies:

- *splitting-based search* via distribution of guiding paths and dynamic load balancing via a split-request and -response protocol, and
- *competition-based search* via freely configurable solver portfolios.

Notably, solver portfolios can also be used in splitting-based search, that is, different guiding paths may be solved with different configurations.

4 Communication architecture

A salient transverse aspect of the architecture of *clasp 2* is its communication infrastructure, used for implementing advanced reasoning procedures. To begin with, the ParallelSolve object keeps track of threads' load, particularly in splitting-based search. Moreover, the Enumerator controls enumeration-based reasoning modes, while the NogoodDistributor handles the exchange of recorded nogoods among solver threads. These communication-intense components along with fundamental implementation techniques are detailed below in increasing order of complexness.

4.1 Thread coordination

The basic communication architecture of *clasp* relies on message passing, efficiently implemented by lock-free atomic integers. On the one hand, globally shared atomic counters are stored in ParallelSolve. For instance, all aforementioned control flags are stored in a single shared atomic integer. On the other hand, each thread has a local message counter hosted by the message handling post propagator (see above). Message passing builds upon two basic methods: `postMessage()` and `hasMessage()`. Posting a message amounts to a *Compare-And-Swap*[5] (CAS) on an atomic integer, and checking for messages (via specialized post propagators) is equivalent to an atomic read. Of particular interest is communication during splitting-based search. This is accomplished via a lock-free work queue, an atomic work request counter, and a work semaphore in ParallelSolve. Initially, the work queue only contains the empty guiding path, and all threads "race" for this work package by issuing a work request. A work request first tries to pop a guiding path from the work queue and returns upon success. Otherwise, the work request counter is incremented and a split request is posted, which results in raising the *split* flag. Afterwards, a `wait()` is tried on the work semaphore.[6] If `wait()` fails because the number of idle threads now equals the total number of threads, the requesting thread posts a *terminate* message and wakes up all waiting threads. Otherwise, the thread is blocked until new work

[5] Conditional writing is performed as atomic CPU instruction to achieve synchronization in multi-threading.

[6] See `http://en.wikipedia.org/wiki/Semaphore_(programming)` in case of unfamiliarity with the working of semaphores.

https://doi.org/10.1017/S1471068412000166 Published online by Cambridge University Press

arrives. On the receiver side, the message handling post propagator of each thread checks whether the *split* flag has been set. If so, and provided that the thread at hand has work to split, its message handler proceeds as follows. At first, it decrements the work request counter. (Note that the message handler thus declares the request as handled before actually serving it in order to minimize over-splitting.) If the work request counter reached 0, the message handler also resets the *split* flag. Afterwards, the search space is split and a (short) guiding path is pushed to the work queue in ParallelSolve. At last, the message handler signals the work semaphore and hence eventually wakes up a waiting thread.

Splitting-based search usually suffers from uninformed early splits of the search space. To counterbalance this, ParallelSolve supports an advanced global restart scheme based on a two-phase strategy. In the first phase, threads vote upon effectuating a global restart based on some given criterion (currently, number of conflicts); however, individual threads may veto a global restart. For instance, this may happen in enumeration when a first model is found during this first restarting phase. Once there are enough votes, a global restart is initiated in the second phase. For this, a *sync* message is posted and threads wait until all solvers have reacted to this message. The last reacting thread decides on how to continue. If no veto was issued, the global restart is executed. That is, threads give up their guiding paths, the work queue is cleared, and the initial (empty) guiding path is again added to the work queue. Otherwise, the restart is abandoned, and the threads simply continue with their current guiding paths.

If splitting-based search is not active (i.e. during competition-based search), the work queue initially contains one (empty) guiding path for each thread, and additional work requests simply result in the posting of a *terminate* message.

4.2 Nogood exchange

Given that each thread implements conflict-driven search involving nogood learning, the corresponding solvers may benefit from a controlled exchange of their recorded information. However, such an interchange must be handled with great care because each individual solver may already learn exponentially many nogoods, so that their additional sharing may significantly hamper the overall performance.

To differentiate which nogoods to share, *clasp 2* pursues a hybrid approach regarding both nogood exchange and storage. As described in Section 3, the binary and ternary implication graph (as well as the positive atom-body dependency graph) are shared among all solver threads. Otherwise, each solver maintains its own local nogood database. The sharing of these nogoods is optional, as we detail next.

The actual exchange of nogoods is controlled in *clasp* by separate distribution and integration components for carefully selecting the spread constraints. This is supported by thread-local interfaces along with the global NogoodDistributor (see Fig. 2). All components rely on interfaces abstracting from the specific sharing mechanism used underneath.

The distribution of nogoods is configurable in two ways. First, the exported nogoods can be filtered by their *type*, viz. conflict, loop, or short (i.e. binary and

https://doi.org/10.1017/S1471068412000166 Published online by Cambridge University Press

ternary), or be exhaustive or inhibited. The difference between globally sharing short nogoods (via their implication graph) and additionally "distributing" them lies in the proactiveness of the process. While the mere sharing leaves it to each solver to discover nogoods added by others, their explicit distribution furthermore communicates this information through the standard distribution process. Second, the export of nogoods is subject to their respective number of distinct decision levels associated with the contained literals, called the *Literal Block Distance* (LBD; Audemard and Simon 2009). Fewer distinct decision levels are regarded as advantageous since they are prone to prune larger parts of the search space. This criterion has empirically shown to be rather effective and largely superior to a selection by length.

The integration of nogoods is likewise configurable in two ways. The first criterion captures the *relevance* of a nogood to the local search process. First, the state of a nogood is assessed by checking whether it is satisfied, violated, open (i.e. neither satisfied nor violated), or unit w.r.t. the current (partial) assignment. While violated and unit nogoods are always considered relevant, open nogoods are optionally passed through a filter using the solver's current heuristic values to discriminate the relevance of the candidate nogood to the current solving process. Finally, satisfied nogoods are either ignored or considered open depending on the configuration of the corresponding filter and their state relative to the original guiding path. The second integration criterion is expressed by a *grace period* influencing the size of the local import queue and thereby the minimum time a nogood is stored. Once the local import queue is full, the least recently added nogood is evicted and either transferred to the thread's nogood database (where it becomes subject to the thread's nogood deletion policy) or immediately discarded. Currently, two modes are distinguished. The thread transfers either all or only "heuristically active" nogoods from its import queue while discarding all others.

Both distribution and integration are implemented as dedicated (complex) post propagators, based upon a global distribution scheme implemented via an efficient lock-free *Multi-Read-Multi-Write* (MRMW) list situated in ParallelSolve.[7] Distribution roughly works as follows. When the solver of Thread i records a nogood that is a candidate for sharing, it is first integrated into the thread-local nogood database. In addition, the nogood's reference counter is set to the total number of threads plus one, and its target mask to all threads except i. At last, Thread i appends the shared nogood to the aforementioned MRMW list.

Conversely upon integration, Thread j traverses the MRMW list, thereby ignoring all nogoods whose target mask excludes j. Depending on the state of a nogood, the aforementioned filters decide whether a nogood is relevant or not. All relevant nogoods are integrated into the search process of Thread j and added to its local import queue. The reference counter of each nogood is decremented by each thread moving its read pointer beyond it. In addition, the sharing thread i decrements a nogood's reference counter whenever it no longer uses it. Hence, the reference

[7] This choice is motivated by the fact that we aim at optimizing *clasp* for desktop computers, still mostly possessing few genuine processing units. Other strategies are possible and an active subject of current research.

https://doi.org/10.1017/S1471068412000166 Published online by Cambridge University Press

counter of a shared nogood can only drop to zero once it is no longer addressed by any read pointer. This makes it subject to deletion.

Notably, the shared representation of a nogood is only created when the nogood is actually distributed. Otherwise, its optimized (single-threaded) representation is used. Upon integration, the "best" representation is selected, for instance, short nogoods are copied while longer ones are physically shared (see Section 5 for implementation details).

4.3 Complex reasoning modes

In addition to model printing, all enumeration-based reasoning modes of *clasp 2* are controlled by the global Enumerator (see Fig. 2). These reasoning modes include regular and projected model enumeration, intersection and union of models, uniform and hierarchical (multi-criteria) optimization as well as combinations thereof, like computing the intersection of all optimal models.

As already mentioned, one global Enumerator is shared among all threads and is protected by a lock. Whenever applicable, it hosts global constraints, like minimize constraints, that are updated whenever a model is found. Additionally, the Enumerator adds a local enumeration-specific constraint to each solver for storing thread-local data, e.g. current optima (see below). Once a model is found, a dedicated message *update-model* is send to all threads, but threads only react to the most recent one.

In fact, enumeration is combinable with both search strategies described in Section 3, either by applying dedicated enumeration algorithms taking advantage of guiding paths or by using solution recording in a competitive setting. The latter setting exploits the infrastructure for nogood exchange in order to distribute solutions among solver threads. Once a solution is converted into a nogood, it can be treated as usual, except that its integration is imperative and that it is exempt from deletion. However, this approach suffers from exponential space complexity in the worst case. Unlike this, splitting-based enumeration runs in polynomial space, following a distributed version of the enumeration algorithm introduced in (Gebser *et al.* 2007a). In order to avoid uninformed splits at the beginning, all solver threads may optionally start in a competitive setting. Once the first model is found, the Enumerator enforces splitting-based search among all solver threads and disables global restarts. In addition to the distribution of disjoint guiding paths, backtrack levels (see Gebser *et al.* 2007a) are dealt with locally in order to guarantee an exhaustive and duplicate-free enumeration of all models.

In optimization, solver threads cooperate in enumerating one better model after another until no better one is found, so that the last model is optimal. Whenever a better model is found, its objective value is stored in the Enumerator. The threads react upon the following *update-model* message by integrating the new value into their local minimize constraint representation[8] and thus into the search processes of their

[8] While the literals of a minimize constraint are stored globally, corresponding upper bounds are local to threads, and changes are communicated through the Enumerator.

https://doi.org/10.1017/S1471068412000166 Published online by Cambridge University Press

solvers. Minimize constraints provide methods for efficiently re-computing their state after an update, so that restarting search is unnecessary in most cases. An innovative feature of *clasp 2* is hierarchical optimization (Gebser *et al.* 2011b), build on top of uniform optimization. Hierarchical optimization allows for solving multi-criteria optimization problems by considering criteria according to their respective priorities. Such an approach is much more involved than standard branch-and-bound-based optimization because it must recover from several unsatisfiable subproblems, one for each criterion. This is accomplished by dynamic minimize constraints that may be disabled and reinitialized during search. Accordingly, nogoods learned under minimize constraints must be retracted once the constraint gets disabled. Another benefit of such dynamic constraints is that we may decrease the (upper) bound in a non-uniform way, and successively re-increase it upon unsatisfiability. Hierarchical optimization allows for gaining an order of magnitude on multi-criteria problems, as witnessed in Linux configuration (Gebser *et al.* 2011c).

Also, brave and cautious reasoning, computing the union and intersection of all models, respectively, are implemented through a global constraint within the Enumerator. Whenever a new model is found, the constraint is intersected with the model (or its complement).

5 Implementation

A major design goal of *clasp 2* was to leverage the power of today's multi-core shared memory machines, while keeping the resulting overhead low so that the single-threaded variant does not suffer from a significant loss in performance. In particular, we aimed at empowering physical sharing of constraints and data while avoiding false sharing, locking, and communication overhead. To this end, our design foresees a clear distinction between three types of data representations, viz.

- *read-only* data providing lock- and wait-free sharing (without deadlocks and races),
- *shared* data being subject to concurrent updates via CAS or locks (admitting races), and
- *thread-local* data being private to each thread and thus not sharable (avoiding deadlocks and races).

Let us make this more precise by detailing the data representations of the various types of constraints used in *clasp*. Constraints are typically separated into a thread-local and a (possibly shared) read-only part. While the former usually contains search-specific and thus dynamic data, the latter typically comprises static data not being subject to change.

As mentioned above, the **implication graph** is shared among all threads and stores inferences from binary and ternary nogoods. The corresponding data structure is separated into two parts. On the one hand, a static read-only part is initialized during preprocessing; it stores two vectors, `bin(l)` and `tern(l)`, for each literal `l`. The former contains literals being forced once `l` becomes true. Similarly, the latter stores binary clauses being activated when `l` becomes true. For better data locality,

https://doi.org/10.1017/S1471068412000166 Published online by Cambridge University Press

`bin(l)` and `tern(l)` are actually stored in one memory block. On the other hand, the dynamic part supports concurrent updates for storing and distributing short recorded nogoods. To this end, it includes, for each literal `l`, an atomic pointer, `learnt(l)`, to a linked list of `CACHE_LINE_SIZE`-sized memory blocks. Each such memory block contains a fixed-size array of binary and ternary nogoods. This setting guarantees that propagation over `learnt(l)` is efficient and does not need any locks (given that short clauses are never removed). Moreover, we rely on fine-grained spinlocks to enable efficient updates of fixed-size arrays.

In analogy, longer **nogoods** are separated into two parts, called head and tail. The head part is always thread-local and is referenced in the owning thread's watch lists. It stores two watched literals, one cache literal, and some extra dynamic data, like nogood activity. The cache literal provides a (potential) spare watched literal, in case one of the two original ones is assigned. That is, upon updating the watched literals, the cache literal is inspected before a costly visit of the literals in the (possibly shared) tail part is engaged.[9] Further contents of the head part depend on whether a nogood is shared. If not, the nogood stores its unshared tail part, including the nogood's size and remaining literals, together with the head in one continuous memory block. Otherwise, the head points to a read-only shared tail object containing the nogood's literals, an (atomic) reference counter, and further static data, like the size of the nogood. The separation into a dynamic thread-local and a static read-only shared part is motivated by the fact that sharing only needs to replicate the search-specific state of a nogood, like its watched literals and activity. Notably, although a more local representation of shared nogoods would be possible, it is important to avoid storing dynamic data of different threads in the same coherence block (e.g. a cache line); otherwise, writes of one thread lead to (logically) unnecessary coherence operations in other threads. Our separation of data ensures that thread-local data of different threads is never stored together and thus avoids such "false sharing." Regarding representation, *clasp* employs the following policies. Short nogoods of up to five literals are never physically shared, but completely stored in thread-local head parts for improving access locality. Original problem nogoods are physically shared in the presence of multiple threads, except if copying (instead of sharing) of problem nogoods is enforced. Finally, recorded nogoods are only shared on demand, as described in Section 4.

Analogously to nogoods, **weight constraints** have a thread-local part storing current assignments (to enclosed literals) and the corresponding sum of weights as well as a shared part storing size, literals, weights, and a reference counter. The shared part of a **minimize constraint** (cf. Section 4) in addition includes priority levels of literals, and thread-local parts contain current (upper) bounds.

Finally, **unfounded-set checking** also relies on a bipartite data representation. As mentioned above, it is implemented as a dedicated post propagator utilizing the (read-only) shared strongly connected components of a program's positive atom-body dependency graph (cf. Section 3). This is again counterbalanced by a

[9] The *Watched Literal Reference Lists* of *miraxt* (Schubert *et al.* 2009) follow a similar approach.

https://doi.org/10.1017/S1471068412000166 Published online by Cambridge University Press

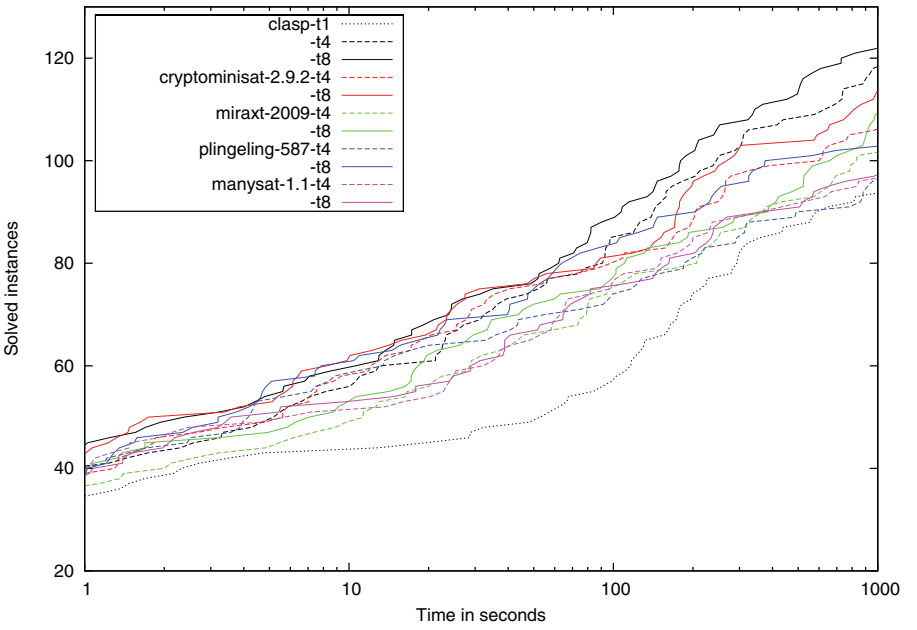


Fig. 3. (Colour online) Number of solved instances per time for *clasp 2* and other multi-threaded SAT solvers.

thread-local part storing assignment-specific data, like source pointers (cf. Simons *et al.* 2002).

6 Experiments

We conducted two series of experiments, the first comparing *clasp 2* to other multi-threaded CDCL-based (SAT) solvers and the second assessing the impact of different parallel search features. In fact, efforts to parallelize CDCL have so far concentrated on the area of SAT, and thus we compare *clasp* (version 2.0.5) to the following multi-threaded SAT solvers: *cryptominisat* (version 2.9.2; Soos *et al.* 2009), *manysat* (version 1.1; Hamadi *et al.* 2009b), *miraxt* (version 2009; Schubert *et al.* 2009), and *plingeling* (version 587f; Biere 2011). While *miraxt* performs search space splitting via guiding paths, the three other solvers let different configurations of an underlying sequential SAT solver compete with one another. Furthermore, nogood exchange among individual threads is either confined to short nogoods, only unary (*plingeling*) or binary ones as well (*cryptominisat*), performed adaptively (*manysat*; cf. Hamadi *et al.* 2009a), or exhaustive in view of a shared nogood database (*miraxt*). The solvers were run on a Linux machine with two Intel Quad-Core Xeon E5520 2.27GHz processors, imposing a limit of 1000 (or 1200) seconds wall-clock time per solver and benchmark instance in the first (or second) series of experiments.[10]

Our first series of experiments evaluates the performance of *clasp* in comparison to other multi-threaded SAT solvers. To this end, we ran the aforementioned solvers on

[10] The benchmark suites are available at `http://www.cs.uni-potsdam.de/clasp`.

https://doi.org/10.1017/S1471068412000166 Published online by Cambridge University Press

160 benchmark instances from the Crafted category at the 2011 SAT competition.[11] The plot in Figure 3 displays numbers of solved instances (on the y-axis) as a function of time (in log scale on the x-axis). As (sequential) baseline, we include *clasp* running one thread in the configuration submitted to the 2011 SAT competition. This configuration is contrasted with four- and eight-threaded variants of the considered parallel SAT solvers, using a prefabricated portfolio (`clasp --create-template`) for competing threads of *clasp*. First of all, we observe in Figure 3 that all multi-threaded solvers complete more instances than sequential *clasp* when given sufficient time (more than 10 seconds). This is unsurprising because the available CPU time roughly amounts to the product of wall-clock time and number of threads, given that our benchmark machine offers sufficient computing resources for concurrent thread execution. In fact, we further observe that each multi-threaded solver benefits from running more (eight instead of four) threads. However, the increase in the number of solved instances is solver-specific and rather small with *manysat*, which mainly duplicates its fixed portfolio of four configurations in the transition to eight threads (changing only the random seed used in the branching heuristics). Unlike this, the other multi-threaded solvers complete between five (*clasp*) and eight (*cryptominisat*, *miraxt*, and *plingeling*) more instances in the time limit when doubling the number of threads. These improvements are significant because harnessing additional computing resources for parallel search is justified when it makes instances accessible that are hard (or unpredictable) to solve sequentially.[12] Comparing the performance of multi-threaded *clasp* to other SAT solvers shows that *clasp* is very competitive, thus emphasizing the (low-level) efficiency of its parallel infrastructure. But please take into account that Crafted benchmarks are closer to ASP problems, which *clasp* is originally designed for, than those in SAT competitions' Application category, to which the other four SAT solvers are tailored. Finally, although solver portfolios (as used in *ppfolio*) proved to be powerful at the 2011 SAT competition, we do not include them in our experiments because their diverse members are run in separation, thus not utilizing multi-threading for parallelization.

The second series of experiments assesses parallel search features of *clasp* on a broad collection of 1435 benchmark instances, stemming from the 2009 ASP and SAT competitions as well as the 2006 and 2008 SAT races. To begin with, the plot in Figure 4 compares different parallel search strategies, viz. portfolio of competing threads (PORT), search space splitting via guiding paths (GP), splitting-based search with a portfolio of different configurations (PORT+GP), and the previous setting augmented with global restarts (PORT+GP+GR). Note that the PORT mode matches the *clasp* setup that has already been used above, and that up

[11] From the whole collection of 300 competition benchmarks, the 160 selected instances could be solved with *ppfolio* (Roussel 2011), the (wall-clock time) winner in the Crafted category at the 2011 SAT competition, within 1000 seconds. Without this preselection, plenty (more) runs of the considered solvers would not finish in the time limit, and running the experiments would have consumed an order of magnitude more time.

[12] The speedup (in terms of wall-clock time) of eight-threaded over single-threaded *clasp* is about 1.5, which may seem low, but the eight-threaded variant completes 31 instances (with unknown sequential solving time) more.

https://doi.org/10.1017/S1471068412000166 Published online by Cambridge University Press

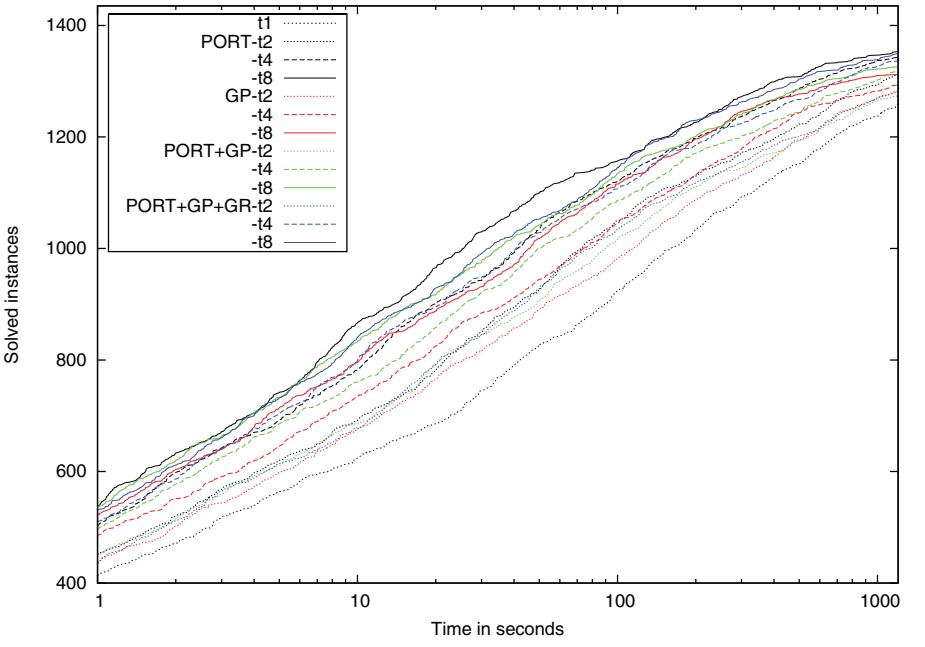


Fig. 4. (Colour online) Number of solved instances per time for different parallel search strategies of *clasp 2*.

to ten restarts (according to the geometric policy $500*1.5^i$) are performed globally with the PORT+GP+GR mode. As in our first experiments, we observe that all multi-threaded *clasp* modes dominate the baseline of running a single thread. Similarly, each mode benefits from more threads, where the transition from two to four threads is particularly significant with portfolio approaches (e.g. 32 more instances completed with PORT). In fact, the latter dominate the GP mode relying on a uniform *clasp* (default) configuration, especially when the number of threads is greater than two. This indicates the difficulty of making fair splits in view of irregular search spaces, while running different configurations in parallel improves the chance of success (cf. Hyvärinen *et al.* 2011). Although the robustness of splitting-based search is somewhat enhanced by running different configurations (PORT+GP) and additionally applying global restarts to refine uninformed splits (PORT+GP+GR), its combinations with guiding paths could not improve over the plain PORT mode. However, it would be interesting to scale this experiment further up (on a machine with more than eight cores) in order to investigate whether a portfolio becomes saturated at some point, so that combinations with search space splitting would be natural to exploit greater parallelism.

Finally, Figure 5 plots the performances of *clasp* (PORT mode) w.r.t. nogood exchange policies. Given that the binary and ternary implication graph is always shared among all threads, the difference between the NO and SHORT modes is that short nogoods are recorded "silently" with NO and proactively communicated with SHORT (cf. Section 4.2). The LBD-2 and -4 modes further extend SHORT by additionally distributing "long" nogoods whose LBD does not exceed 2 or 4,

https://doi.org/10.1017/S1471068412000166 Published online by Cambridge University Press

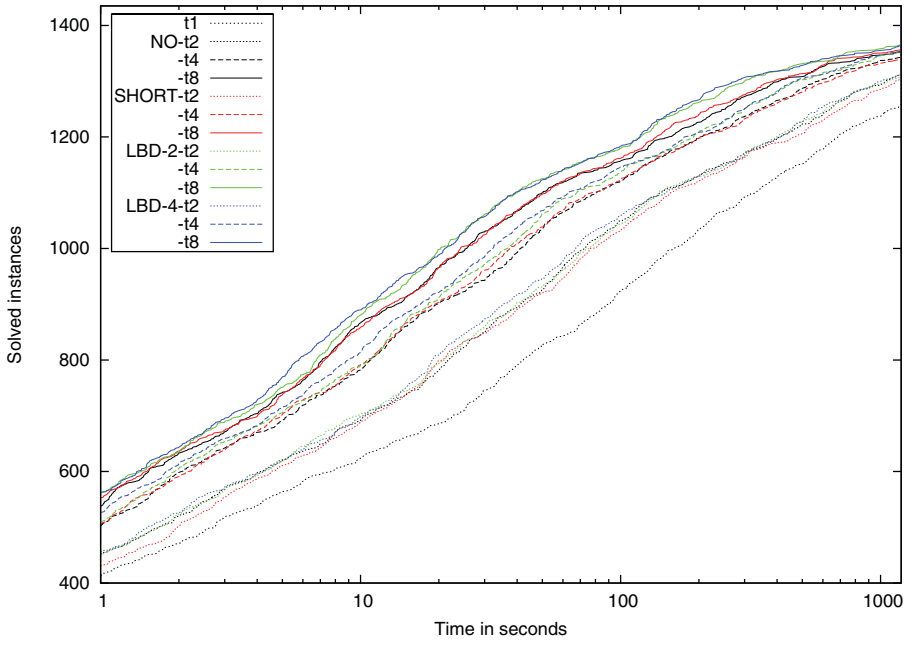


Fig. 5. (Colour online) Number of solved instances per time for different nogood exchange policies of *clasp 2*.

respectively, independent of the nogood size in terms of literals. While the amount of solved instances is primarily influenced by the number of threads, different nogood exchange policies are responsible for gradual differences between *clasp* variants running the same number of threads. With four and eight threads, the LBD modes are more successful than NO and SHORT, especially in the time interval from 10 to a few hundred seconds. This shows that the exchange of information helps to reduce redundancies between the search processes of individual threads; it further supports the conjecture in (Audemard and Simon 2009) that "our measure [LBD] will also be very useful in the context of parallel SAT solvers." Interestingly, even when running eight threads, the performances of LBD-2 and -4 modes are close to each other, with a slight tendency towards LBD-4. Our experiments do thus not exhibit bottlenecks due to the additional exchange of nogoods with LBD 3 and 4. However, more exhaustive experiments are required (and part of our ongoing work) to find a good trade-off between number of threads and LBD limit for exchange. Ultimately, dynamic measures like those suggested in (Hamadi *et al.* 2009a) are indispensable for self-adapting nogood exchange to different problem characteristics, and adding such measures to *clasp* is a subject to future work.

7 Related work

Parallel ASP solving was so far dominated by approaches distributing tree search by extending the solver *smodels* in various ways (Finkel *et al.* 2001; Hirsimäki 2001; Pontelli *et al.* 2003; Balduccini *et al.* 2005; Gressmann *et al.* 2005; Gressmann *et al.* 2006). While *smodels* applies systematic backtracking-based search, following the

https://doi.org/10.1017/S1471068412000166 Published online by Cambridge University Press

scheme of DPLL used in traditional SAT solving, *clasp* as well as modern SAT solvers are based on CDCL, relying on conflict-driven learning and backjumping. However, the clear edge of CDCL-based solvers over DPLL-based ones also brings about more sophisticated search procedures that have to be accommodated in a distributed setting. Apart from distributed constraint learning, this particularly affects the coordination of model enumeration.

The approach taken with *claspar* (Ellguth *et al.* 2009; Gebser *et al.* 2011d) can be regarded as a precursor to our present work. *claspar* is designed for a cluster-oriented setting without any shared memory. It thus aims at large-scale computing environments, where physical distribution necessitates data copying rather than sharing. In fact, *claspar* can be understood as a wrapper controlling the distribution of independent *clasp* instances via MPI (Gropp *et al.* 1999), thereby taking advantage of *clasp*'s interfaces for data exchange. However, compared to *claspar*, (quasi) instantaneous communication via shared memory enables a much closer collaboration (e.g. rapid nogood exchange) among threads in *clasp*.

Although much work has also been carried out in the area of parallel logic programming, among which or-parallelism (Chassin de Kergommeaux and Codognet 1994; Gupta *et al.* 2001) is similar to search space splitting, our work is more closely related to parallel SAT solving, tracing back to (Zhang *et al.* 1996; Blochinger *et al.* 2003). Among modern approaches to multi-threaded SAT solving, the ones of *miraxt* (Schubert *et al.* 2009) and *manysat* (Hamadi *et al.* 2009b) are of particular interest due to their complementary treatment of recorded nogoods. *miraxt* is implemented via *pthreads* and uses a globally shared nogood database. The advantage of this is that each thread sees all nogoods and can integrate them with low latency. However, given that multiple threads read and write on the database, it needs readers-writer locks. Moreover, many nogoods are actually never used by more than one thread, but still produce some maintenance overhead in each thread. *manysat* is implemented via *openmp* and uses a copying approach to nogood exchange, proscribing any physical sharing. That is, each among n solver threads has its own nogood database, and nogood exchange is accomplished by copying via $n*(n-1)$ pairwise distribution queues. While this approach performs well for a small number n of solver threads, it does not scale up due to the quadratic number of queues and excessive copying. Recent parallel SAT solvers further include *plingeling* (Biere 2011) and the multi-threaded variant of *cryptominisat* (Soos *et al.* 2009). Finally, note that, while knowledge exchange and (shared) memory access matter likewise in parallel SAT and ASP solving, the scope of the latter also stretches out over enumeration and optimization of answer sets.

8 Discussion

We have presented major design principles and key implementation techniques underlying the *clasp 2* series, thus providing the first CDCL-based ASP solver supporting parallelization via multi-threading. While its multi-threaded variant aims at leveraging the power of today's multi-core shared memory machines in parallel search, *clasp 2* has also been designed with care not to sacrifice the

https://doi.org/10.1017/S1471068412000166 Published online by Cambridge University Press

(low-level) performance of its single-threaded variant, sharing a common code base. In fact, the competitiveness of single- as well as multi-threaded *clasp 2* variants is, for instance, witnessed by their performances at the 2011 SAT competition. Beyond powerful parallel search, multi-threaded *clasp 2* allows for conducting the various reasoning modes of its single-threaded sibling, including enumeration and (hierarchical) optimization, in parallel. On the one hand, this makes the multi-threaded variant of *clasp 2* highly flexible, offering parallel solving capacities for various reasoning tasks. On the other hand, the vast configuration space of a CDCL-based solver becomes even more complex, as individual threads as well as their interaction can be configured in manifold ways. In view of this, adaptive solving strategies (e.g. regarding nogood exchange) and automatic parallel solver configuration are important issues to future work.

Acknowledgements

We are grateful to Hannes Schröder for support with experiments and to the anonymous referees for their comments. This work was partially funded by the German Science Foundation (DFG) under grant SCHA 550/8-2.

References

Audemard, G. and Simon, L. 2009. Predicting learnt clauses quality in modern SAT solvers. See Boutilier (2009), 399–404.

Balduccini, M., Pontelli, E., El-Khatib, O. and Le, H. 2005. Issues in parallel execution of non-monotonic reasoning systems. *Parallel Computing 31,* 6, 608–647.

Baral, C. 2003. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press.

Biere, A. 2011. *Lingeling and Friends at the SAT Competition 2011*. Technical Report FMV 11/1, Institute for Formal Models and Verification, Johannes Kepler University.

Biere, A., Heule, M., van Maaren, H. and Walsh, T., Eds. 2009. *Handbook of Satisfiability*. Frontiers in Artificial Intelligence and Applications, vol. 185. IOS Press.

Blochinger, W., Sinz, C. and Küchlin, W. 2003. Parallel propositional satisfiability checking with distributed dynamic learning. *Parallel Computing 29,* 7, 969–994.

Boutilier, C., Ed. 2009. *Proceedings of the Twenty-first International Joint Conference on Artificial Intelligence (IJCAI'09)*. AAAI Press.

Chassin de Kergommeaux, J. and Codognet, P. 1994. Parallel logic programming systems. *ACM Computing Surveys 26,* 3, 295–336.

Clark, K. 1978. Negation as failure. In *Logic and Data Bases*, H. Gallaire and J. Minker, Eds. Plenum Press, 293–322.

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

Davis, M. and Putnam, H. 1960. A computing procedure for quantification theory. *Journal of the ACM 7*, 201–215.

Eén, N. and Biere, A. 2005. Effective preprocessing in SAT through variable and clause elimination. In *Proceedings of the Eighth International Conference on Theory and Applications of Satisfiability Testing (SAT'05)*, F. Bacchus and T. Walsh, Eds. Lecture Notes in Computer Science, vol. 3569. Springer-Verlag, 61–75.

https://doi.org/10.1017/S1471068412000166 Published online by Cambridge University Press

Ellguth, E., Gebser, M., Gusowski, M., Kaminski, R., Kaufmann, B., Liske, S., Schaub, T., Schneidenbach, L. and Schnor, B. 2009. A simple distributed conflict-driven answer set solver. In *Proceedings of the Tenth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'09)*, E. Erdem, F. Lin and T. Schaub, Eds. Lecture Notes in Artificial Intelligence, vol. 5753. Springer-Verlag, 490–495.

Finkel, R., Marek, V., Moore, N. and Truszczyński, M. 2001. Computing stable models in parallel. In *Proceedings of the First International Workshop on Answer Set Programming (ASP'01)*, A. Provetti and T. Son, Eds. AAAI Press, 72–76.

Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T. and Schneider, M. 2011a. Potassco: The Potsdam answer set solving collection. *AI Communications 24,* 2, 105–124.

Gebser, M., Kaminski, R., Kaufmann, B. and Schaub, T. 2011b. Multi-criteria optimization in answer set programming. In *Technical Communications of the Twenty-seventh International Conference on Logic Programming (ICLP'11)*, J. Gallagher and M. Gelfond, Eds. Leibniz International Proceedings in Informatics, vol. 11. Dagstuhl Publishing, 1–10.

Gebser, M., Kaminski, R., Kaufmann, B. and Schaub, T. 2011c. Multi-criteria optimization in ASP and its application to Linux package configuration. Available at `http://www.cs.uni-potsdam.de/wv/pdfformat/gekakasc11b.pdf`.

Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T. and Schnor, B. 2011d. Cluster-based ASP solving with claspar. In *Proceedings of the Eleventh International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'11)*, J. Delgrande and W. Faber, Eds. Lecture Notes in Artificial Intelligence, vol. 6645. Springer-Verlag, 364–369.

Gebser, M., Kaufmann, B., Neumann, A. and Schaub, T. 2007a. Conflict-driven answer set enumeration. In *Proceedings of the Ninth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'07)*, C. Baral, G. Brewka and J. Schlipf, Eds. Lecture Notes in Artificial Intelligence, vol. 4483. Springer-Verlag, 136–148.

Gebser, M., Kaufmann, B., Neumann, A. and Schaub, T. 2007b. Conflict-driven answer set solving. In *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI'07)*, M. Veloso, Ed. AAAI Press, 386–392.

Gebser, M., Kaufmann, B., Neumann, A. and Schaub, T. 2008. Advanced preprocessing for answer set solving. In *Proceedings of the Eighteenth European Conference on Artificial Intelligence (ECAI'08)*, M. Ghallab, C. Spyropoulos, N. Fakotakis and N. Avouris, Eds. IOS Press, 15–19.

Gebser, M., Ostrowski, M. and Schaub, T. 2009. Constraint answer set solving. In *Proceedings of the Twenty-fifth International Conference on Logic Programming (ICLP'09)*, P. Hill and D. Warren, Eds. Lecture Notes in Computer Science, vol. 5649. Springer-Verlag, 235–249.

Gomes, C. and Selman, B. 2001. Algorithm portfolios. *Artificial Intelligence 126,* 1–2, 43–62.

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

Gressmann, J., Janhunen, T., Mercer, R., Schaub, T., Thiele, S. and Tichy, R. 2006. On probing and multi-threading in platypus. In *Proceedings of the Seventeenth European Conference on Artificial Intelligence (ECAI'06)*, G. Brewka, S. Coradeschi, A. Perini and P. Traverso, Eds. IOS Press, 392–396.

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

https://doi.org/10.1017/S1471068412000166 Published online by Cambridge University Press

Gupta, G., Pontelli, E., Ali, K., Carlsson, M. and Hermenegildo, M. 2001. Parallel execution of Prolog programs: A survey. *ACM Transactions on Programming Languages and Systems 23,* 4, 472–602.

Hamadi, Y., Jabbour, S. and Sais, L. 2009a. Control-based clause sharing in parallel SAT solving. See Boutilier (2009), 499–504.

Hamadi, Y., Jabbour, S. and Sais, L. 2009b. ManySAT: A parallel SAT solver. *Journal on Satisfiability, Boolean Modeling and Computation 6*, 245–262.

Han, H. and Somenzi, F. 2009. On-the-fly clause improvement. See Kullmann (2009), 209–222.

Herlihy, M. and Shavit, N. 2008. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers.

Hirsimäki, T. 2001. *Distributing Backtracking Search Trees.* Technical Report, Helsinki University of Technology.

Hyvärinen, A., Junttila, T. and Niemelä, I. 2011. Partitioning search spaces of a randomized search. *Fundamenta Informaticae 107,* 2-3, 289–311.

Järvisalo, M., Biere, A. and Heule, M. 2010. Blocked clause elimination. In *Proceedings of the Sixteenth International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'10)*, J. Esparza and R. Majumdar, Eds. Lecture Notes in Computer Science, vol. 6015. Springer-Verlag, 129–144.

Kullmann, O., Ed. 2009. *Proceedings of the Twelfth International Conference on Theory and Applications of Satisfiability Testing (SAT'09)*. Lecture Notes in Computer Science, vol. 5584. Springer-Verlag.

Li, C. and Manyà, F. 2009. MaxSAT. See Biere *et al.* (2009), Chapter 19, 613–631.

Marques-Silva, J., Argelich, J., Graça, A. and Lynce, I. 2011. Boolean lexicographic optimization: Algorithms and applications. *Annals of Mathematics and Artificial Intelligence 62,* 3-4, 317–343.

Marques-Silva, J. and Sakallah, K. 1999. GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers 48,* 5, 506–521.

Moskewicz, M., Madigan, C., Zhao, Y., Zhang, L. and Malik, S. 2001. Chaff: Engineering an efficient SAT solver. In *Proceedings of the Thirty-eighth Conference on Design Automation (DAC'01)*. ACM Press, 530–535.

Pontelli, E., Balduccini, M. and Bermudez, F. 2003. Non-monotonic reasoning on Beowulf platforms. In *Proceedings of the Fifth International Symposium on Practical Aspects of Declarative Languages (PADL'03)*, V. Dahl and P. Wadler, Eds. Lecture Notes in Artificial Intelligence, vol. 2562. Springer-Verlag, 37–57.

Roussel, O. 2011. Description of ppfolio. Available at `http://www.cril.univ-artois.fr/~roussel/ppfolio/solver1.pdf`.

Ryan, L. 2004. *Efficient Algorithms for Clause-learning SAT Solvers.* Master's Thesis, Simon Fraser University.

Schubert, T., Lewis, M. and Becker, B. 2009. PaMiraXT: Parallel SAT solving with threads and message passing. *Journal on Satisfiability, Boolean Modeling and Computation 6*, 203–222.

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

Soos, M., Nohl, K. and Castelluccia, C. 2009. Extending SAT solvers to cryptographic problems. See Kullmann (2009), 244–257.

Syrjänen, T. Lparse 1.0 user's manual. Available at `http://www.tcs.hut.fi/Software/smodels/lparse.ps.gz`.

Zhang, H., Bonacina, M. and Hsiang, J. 1996. PSATO: A distributed propositional prover and its application to quasigroup problems. *Journal of Symbolic Computation 21,* 4, 543–560.

https://doi.org/10.1017/S1471068412000166 Published online by Cambridge University Press

Zhang, L., Madigan, C., Moskewicz, M. and Malik, S. 2001. Efficient conflict driven learning in a Boolean satisfiability solver. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD'01)*. 279–285.

Zhang, L. and Malik, S. 2003. Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications. In *Proceedings of the Sixth Conference on Design, Automation and Test in Europe (DATE'03)*. IEEE Computer Society, 10880–10885.

https://doi.org/10.1017/S1471068412000166 Published online by Cambridge University Press