Automatic Construction of Parallel Portfolios via Algorithm Configuration

Marius Lindauer[a], Holger Hoos[b], Kevin Leyton-Brown[b], Torsten Schaub[c,d]

[a] *University of Freiburg, Germany*
[b] *University of British Columbia, Vancouver, Canada*
[c] *University of Potsdam, Germany*
[d] *INRIA Rennes, France*

Abstract

Since 2004, increases in computational power described by Moore's law have substantially been realized in the form of additional cores rather than through faster clock speeds. To make effective use of modern hardware when solving hard computational problems, it is therefore necessary to employ parallel solution strategies. In this work, we demonstrate how effective parallel solvers for propositional satisfiability (SAT), one of the most widely studied NP-complete problems, can be produced automatically from any existing sequential, highly parametric SAT solver. Our *Automatic Construction of Parallel Portfolios* (ACPP) approach uses an automatic algorithm configuration procedure to identify a set of configurations that perform well when executed in parallel. Applied to two prominent SAT solvers, *Lingeling* and *clasp*, our ACPP procedure identified 8-core solvers that significantly outperformed their sequential counterparts on a diverse set of instances from the *application* and *hard combinatorial* category of the 2012 SAT Challenge. We further extended our ACPP approach to produce parallel portfolio solvers consisting of several different solvers by combining their configuration spaces. Applied to the component solvers of the 2012 SAT Challenge gold medal winning SAT Solver *pfolioUZK*, our ACPP procedures produced a significantly better-performing parallel SAT solver.

Keywords: Algorithm Configuration; Parallel SAT Solving; Algorithm Portfolios; Programming by Optimization; Automated Parallelization

1. Introduction

Over most of the last decade, additional computational power has come primarily in the form of increased parallelism. As a consequence, effective parallel solvers are increasingly key to solving computationally challenging problems.

Email addresses: `lindauer@informatik.uni-freiburg.de` (Marius Lindauer), `hoos@cs.ubc.ca` (Holger Hoos), `kevinlb@cs.ubc.ca` (Kevin Leyton-Brown), `torsten@cs.uni-potsdam.de` (Torsten Schaub)

Unfortunately, the manual construction of parallel solvers is non-trivial, often requiring fundamental redesign of existing, sequential approaches, as identified by Hamadi and Wintersteiger [33] as the challenge of *Starting from Scratch*. It is thus very appealing to employ generic methods for the construction of parallel solvers from inherently sequential sources as a first step. Indeed, the prospect of a substantial reduction in human development cost means that such approaches can have a significant impact, even if their performance does not reach that of special-purpose parallel designs—just as high-level programming languages are useful, even though compiled software tends to fall short of the performance that can be obtained from expert-level programming in assembly language. One promising approach for parallelizing sequential algorithms is the design of parallel algorithm portfolios – sets of solvers that are run in parallel on a given instance of a decision problem, such as SAT, until the first of them finds a solution [40, 27].

In this work,[1] we study generic methods for solving a problem we call Automatic Construction of Parallel Portfolios (ACPP): automatically constructing a static[2] parallel solver from a sequential solver or a set of sequential solvers. This task can be understood as falling within the *programming by optimization* paradigm [35] in that it involves the design of software in which many design decisions have been deliberately left open during the development process (here exposed as parameters of SAT solvers) to be made automatically later (here by means of an automated algorithm configurator) in order to obtain optimized performance for specific use cases. Hence, all that is required by our ACPP methods is a sequential solver whose configuration space contains complementary configurations.

We study three variants of the ACPP problem. First, we consider building parallel portfolios starting from a single, highly parametric sequential solver design. However, for well-studied problems (e.g., SAT), there often exist a wide range of different solvers that contribute to the state of the art (see, e.g., [74]). Complementarities among such solvers can be exploited by algorithm portfolios, whether driven by algorithm selection (like SATzilla [73]) or by parallel execution (such as *ppfolio* [64] or *pfolioUZK* [71]). Thus, the second problem we consider is leveraging such complementarities within the context of the ACPP problem, generating a parallel portfolio based on a design space induced from a set of multiple (possibly parametrized) solvers. Finally, some parallel solvers already exist; these have the advantage that they can increase performance by communicating intermediate results—notably, learned clauses—between different processes. The third problem we study is constructing parallel portfolios from a set containing both sequential and parallel solvers.

We investigate three methods for solving the ACPP problem.

1. Global simultaneously configures all solvers in a k-solver parallel portfo-

[1]This paper extends a 2012 workshop publication [38].

[2]In contrast to parallel algorithm selection systems [54, 55, 56], we do not dynamically select solvers on a per-instance base but automatically construct a static portfolio.

lio, representing this ACPP problem as a single-algorithm configuration problem with a design space corresponding to the kth Cartesian power of the design space of the given sequential solver. This has the advantages of simplicity and comprehensiveness (no candidate portfolios are omitted from the design space) but the disadvantage that the size of the design space increases exponentially with k, which quickly produces extremely difficult configuration problems.

2. *Hydra* is a method for building portfolio-based algorithm selectors from a single, highly parameterized solver [72]. It proceeds iteratively. In the first round, it aims to find a configuration that maximizes overall performance on the given dataset. In the $i + 1$st round, it aims to find a configuration that maximizes marginal contribution across the configurations identified in the previous i rounds. In the original version of *Hydra*, these marginal contributions were calculated relative to the current selector; in the latest version of *Hydra*, they are determined relative to an idealized, perfect selector [42]. The wall-clock performance of a perfect selector across i solvers (also known as *virtual best solver*) is the same as the wall-clock performance of the same i solvers running in parallel; thus, the same general idea can be used to build parallel portfolios. (Building a parallel portfolio in this way has the added advantage that no instance features are required, since there is no need to select among algorithms.) We introduce some enhancements to this approach for the parallel portfolio setting (discussed in Section 3.1.2), and refer to our method as parHydra.
3. Some parallel solvers only achieve strong performance when running on more than one core; such solvers will not be found by a greedy approach like parHydra, which only adds one configuration at a time and does not recognize interaction effects that arise between different threads of a parallel solver. To overcome this problem, we introduce a new method called parHydra$_b$, which augments parHydra to train b solvers per iteration. This method trades off the computational benefit of parHydra's greedy approach with the greater coverage of Global.

We evaluated our ACPP methods on SAT. We chose this domain because it is highly relevant to academia and industry and has been widely studied. We thus had access to a wide range of strong, highly parametric solvers and were assured that the bar for demonstrating efficacy of parallelization strategies was appropriately high. We note that our approach is not limited to SAT solvers and can be directly applied to other domains. To evaluate our methods in the single-solver setting, we studied both *Lingeling* and *clasp*: prominent, highly parametric state-of-the-art solvers for SAT. *Lingeling* won a gold medal in the application (wall-clock) track of the 2011 SAT Competition and *clasp* placed first in the hard combinatorial track of the 2012 SAT Challenge. To evaluate our methods for generating parallel portfolios involving multiple solvers, we started with the set of solvers included by *pfolioUZK*, a parallel portfolio solver based on several solvers in their default configurations that won the gold medal in the parallel track of the 2012 SAT Challenge. This set includes *Plingeling*, a parallel solver.

Our results demonstrate that parHydra transforms single solvers into parallel portfolios both well and robustly. Its performance on standard 8-core CPUs compared favourably with that of hand-crafted parallel SAT solvers. For the generation of parallel algorithm portfolios based on a set of both parallel and sequential solvers, we found that parHydra$_b$ was best among the alternatives we considered, notably outperforming *pfolioUZK*. More detailed experimental results and open-source code are available at `http://www.cs.uni-potsdam.de/acpp`.

2. Background and Related Work

We now survey related work on parallel SAT solving and algorithm portfolios.

2.1. Background: SAT Solving

The Boolean satisfiability problem (SAT) is to decide whether it is possible to assign truth values (true, false) to the variables in a given propositional formula F such that F becomes true. If such an assignment exists, F is called satisfiable, otherwise F is called unsatisfiable. A complete SAT solver takes as an input a formula F, typically in conjunctive normal form (a conjunction of disjunctions of variables and their negations) and determine a satisfiable assignment or prove that none exists. An incomplete SAT solver can find satisfying assignments, but not prove unsatisfiability.

Most state-of-the-art complete SAT solvers are based on conflict-driven clause learning (CDCL; [58]). Their parameters control variable selection for branching decisions, clause learning and restart techniques. State-of-the-art incomplete SAT solvers use stochastic local search (SLS; [39]), and their parameters control the selection of the variable whose value is modified in each local search step as well as the diversification and additional intensification strategies. Furthermore, there exist several preprocessing techniques (e.g., [21]) to simplify formulas and their parameters control how long and how aggressive preprocessing will be used – too much preprocessing can remove important structural information and hence, it can increase the hardness of formulas. The efficacy of SAT solvers depends on multiple heuristic components whose basic functions and the interplay between them are controlled by parameters. Some parameters are categorical (e.g., choice between different search strategies in SLS), while many others are integer- or real-valued (e.g., the damping factor used in computing heuristic variable scores in CDCL).

Parallel SAT solvers have received increasing attention in recent years. *ManySAT* [30, 31, 29] was one of the first parallel SAT solvers. It is a static portfolio solver that uses clause sharing between its components, each of which is a manually configured, CDCL-type SAT solver based on *MiniSat* [22]. *PeneLoPe* [5, 23] is based on *ManySAT* and adds several policies for importing and exporting clauses between the threads. *Plingeling* [12, 13, 14, 15, 16] is based on a similar design; its version 587, which won a gold medal in the application track of the 2011 SAT Competition (with respect to wall clock time on SAT+UNSAT instances), and the 2012 version *ala*, share unit clauses as well as equivalences between their component solvers. Similarly, *CryptoMiniSat* [66], which won silver in the application track of the 2011 SAT Competition, shares unit and binary clauses. *clasp* [26] is a state-of-the-art solver for SAT, ASP and PB that supports parallel multithreading (since version 2.0.0) for search space splitting and/or competing strategies, both combinable with a portfolio approach. *clasp* shares unary, binary and ternary clauses, and (optionally) offers a parameterized mechanism for distributing and integrating (longer) clauses. Finally, *ppfolio* [64] is a simple, static parallel portfolio solver for SAT without clause sharing that uses *CryptoMiniSat*, *Lingeling*, *clasp*, *TNM* [70] and *march_hi* [34] in their default configurations as component solvers, and that won numerous medals at the 2011 SAT Competition. Like the previously mentioned portfolio solvers for SAT, *ppfolio* was constructed manually, but uses a very diverse set of high-performance solvers as its components. *pfolioUZK* [71] follows the same idea as used for *ppfolio* but uses other component solvers; it won the parallel track of the 2012 SAT Challenge. On one hand, ACPP can be understood as automatically replicating the (hand-tuned) success of solvers like *ManySAT*, *Plingeling*, *CryptoMiniSat* or *clasp*, which are inherently based on different configurations of a single parametric solver; on the other, it is also concerned with automatically producing effective parallel portfolios from multiple solvers, such as *ppfolio* and *pfolioUZK*, while exploiting the rich design spaces of these component solvers.

Katsirelos et al. [47] showed that an effective parallelisation of a CDCL SAT solver does not merely hinge on picking a good clause sharing strategy, since it is not straightforward to obtain shorter resolution proofs by parallelisation without essential changes of the underlying sequential reasoning mechanism. Our ACPP does not aim at parallelising the resolution proof, but rather runs multiple algorithms and algorithm configurations in parallel, in order to maximise the probability that at least one of them finds a short proof quickly.

2.2. Related Work

Well before there was widespread interest in multi-core computing, the potential benefits offered by parallel algorithm portfolios were identified in seminal work by Huberman et al. [40]. Their notion of an algorithm portfolio is inspired by the use of portfolios of assets for risk management in finance and amounts to running multiple algorithms concurrently and independently on the same instance of a given problem, until one of them produces a solution. Gomes et al. [27] further investigated conditions under which such portfolios outperform their component solvers. Both lines of work considered prominent constraint programming problems (graph colouring and quasigroup completion), but neither presented methods for automatically constructing portfolio solvers. Parallel portfolios have since made practical impact, both in cases where the allocation of computational resources to algorithms in the portfolio is static [63, 77] and where the component solvers contained in a portfolio or the resources assigned to them can change over time [24].

A closely related notion of algorithm portfolios first saw practical application in this domain as the basis for algorithm selectors such as SATzilla [59, 73] and many conceptually related methods (see, e.g., [49]). In this context, a portfolio is a set of candidate algorithms for a given problem from which one or more solvers are selected to be run, based on characteristics of the problem instance to be solved.

p3S [45, 54] and *parCSHC* [55, 56] were the first methods to automatically select a parallel portfolio (in the case of *p3S*, actually, a parallel algorithm schedule) from a given set of SAT solvers on a per-instance basis. *p3S* [54] is a parallel extension of the sequential algorithm selector *3S* [45]. Similar to the sequential version, *p3S* uses k-nearest neighbour clustering to determine the k training instances closest in the feature space to a new instance to be solved, and computes a per-instance parallel algorithm schedule based on the runtime data of these instances using Integer Linear Programming (ILP; [62, 65]). In contrast to our ACPP method, which trains the portfolio offline, the ILP problem within *p3S* has to be solved online for each new instance to determine a well-performing parallel portfolio. This quickly becomes very time-consuming as the number of available solvers grows and as more CPU cores are considered. *parCSHC* was specially designed for the SAT Competition. It always statically and independently runs 4 threads of the parallel SAT solver *Plingeling*, 1 thread of the sequential SAT solver *CCASat*, and three solvers selected on a per-instance basis. These latter solvers are selected by models that are trained on application, hard-combinatorial and random SAT instances, respectively. Other approaches to the per-instance selection of parallel portfolios that have emerged since our own are *sunny-cp2* [2], which selects a parallel algorithm schedule, and *claspfolio 2* [52], which implements several extensions of sequential algorithm selectors to select a parallel portfolio.

One thing that all of these methods have in common—whether parallel, selection-based or both—is that they build a portfolio from a relatively small candidate set of distinct algorithms. While, in principle, these methods could also be applied given a set of algorithms expressed implicitly as the configurations of one parametric solver, in practice, they are useful only when the set of candidates is relatively small. The same limitation applies to existing approaches that combine algorithm selection and scheduling, notably *CPHydra* [61], which also relies on cheaply computable features of the problem instances to be solved and selects multiple solvers to be run in parallel. Two further, conceptually related approaches are *aspeed* [36] and *MIPSAT* [60], which compute (parallel) algorithm schedules by taking advantage of the modelling and solving capacities of Answer Set Programming (ASP [10, 25]) and Mixed Integer Programming (MIP; [62, 65]), respectively.

Recently, automatic algorithm configuration has become increasingly effective, with the advent of high-performance methods such as *ParamILS* [43], *GGA* [3], *irace* [53] and *SMAC* [41]. As a result, there has been recent interest in automatically identifying useful portfolios of configurations from large algorithm design spaces. As before, such portfolio-construction techniques were first demonstrated to be practical in the case of portfolio-based algorithm selectors. We have already discussed one key method for solving this problem: *Hydra* [72], which greedily constructs a portfolio by configuring solvers iteratively, changing the configurator's objective function at each iteration to direct it to maximize marginal contribution to the portfolio. Another key method is *ISAC* [46], which clusters instances based on features and runs the configurator separately for each cluster. Malitsky et al. [57] extended *ISAC*'s scope to the construction of portfolios from a set of different solvers. However, there are three differences between the construction of sequential portfolios and of static parallel portfolios.

1. Whereas we know how many algorithms we need for a parallel portfolio when running exactly one solver per processor core (i.e., the size of the portfolio is limited to the number of processor cores available), the potential size of the portfolio is unlimited in the sequential case, since we may not select all solvers to run.
2. A sequential portfolio solver must somehow select component solvers (which can result in making the wrong decision), while static parallel solvers run the entire portfolio in parallel and thus achieve nearly the same performance as the portfolio's virtual best solver. We note that both approaches are bounded by the performance of the virtual best solver.
3. Using several cores in parallel introduces overhead which should be considered in the configuration process.

3. Parallel Portfolio Configuration from a Single Sequential Solver

We begin by considering the problem of automatically producing a parallel portfolio solver from a single, highly-parametric sequential solver; this closely resembles the problem (manually) addressed by the developers of solvers like *ManySAT*, *Plingeling*, *CryptoMiniSat* and *clasp*. First, we define our three ACPP methods. Next, we illustrate the performance of our ACPP portfolio solvers based on *Lingeling* and *clasp* and analyze the empirical scalability of our trained ACPP solvers. Finally, in the case where clause sharing is in the design space of the component solvers, we extend our ACPP solvers with clause sharing and investigate how much further performance can be achieved by this extension.

3.1. Approach

We now describe three methods for automatically constructing parallel portfolios from a single parametric solver. We use C to denote the configuration space of our parametric solver, $c \in C$ to represent individual configurations, and I to refer to the given set of problem instances. Our goal is to optimize (without loss of generality, to minimize) performance according to a given metric m. (In our experiments, we minimize penalized average runtime, PAR10.[3]) We use a k-tuple $c_{1:k} = (c_1, \dots, c_k)$ to denote a parallel portfolio with k component solvers. The parallel portfolio's full configuration space is $C^k = \prod_{l=1}^{k}\{(c) \mid c \in C\}$, where the product of two configuration spaces X and Y is defined as $\{x||y \mid x \in X, y \in Y\}$,

[3]PARX penalizes each timeout with X times the given cutoff time [43].

Algorithm 1: Portfolio Configuration Procedure GLOBAL

```
Input  : parametric solver with configuration space $C$; desired number $k$
         of component solvers; instance set $I$; performance metric $m$;
         configurator $AC$; number $n$ of independent configurator runs;
         total configuration time $t$
Output : parallel portfolio solver with portfolio $\hat{c}_{1:k}$
1 for $j := 1 \dots n$ do
2     obtain portfolio $c^{(j)}_{1:k}$ by running $AC$ on configuration space
      $\prod_{l=1}^{k}\{(c) \mid c \in C\}$ on $I$ using $m$ for time $t/n$
3 choose $\hat{c}_{1:k} \in \arg\min_{c^{(j)}_{1:k} \mid j \in \{1 \dots n\}} m(c^{(j)}_{1:k}, I)$ that achieved best
  performance on $I$ according to $m$
4 return $\hat{c}_{1:k}$
```

with $x||y$ denoting the concatenation (rather than nesting) of tuples. Let AC denote a generic algorithm configuration procedure; in our experiments, we used *SMAC* [41]. Following established best practices (see [41]), we performed n independent runs of AC, obtained configured solvers $c^{(j)}$ with $j \in \{1 \dots n\}$ and retained the configured solver $\hat{c}$ which achieved the best performance on instance set I according to metric m. By t we denote the overall time budget available for producing a parallel portfolio solver.

*3.1.1. Simultaneous configuration of all component solvers (*GLOBAL*)*

Our first portfolio configuration method is the straightforward extension of standard algorithm configuration to the construction of a parallel portfolio (see Algorithm 1). Specifically, if the given solver has ℓ parameters, we treat the portfolio $c_{1:k}$ as a single algorithm with $\ell \cdot k$ parameters inducing a configuration space of size $|C|^k$, and configure it directly. As noted above, we identify a single configuration as the best of n independent runs of AC. These runs can be performed in parallel, meaning that this procedure requires wall clock time t/n if n machines – one for each AC run – with k cores are available. The used CPU time will be the given time budget t for Lines 1 and 2 in Algorithm 1 and some small overhead ϵ to choose the best portfolio in Line 3. The scalability of this approach is limited by the fact that the global configuration space C^k to which AC is applied grows exponentially with k. However, given a powerful configurator, a moderate value of k and a reasonably sized C, this simple approach can be effective, especially when compared to manual parallel portfolio construction.

*3.1.2. Iterative configuration of component solvers (*PARHYDRA*)*

The key problem with GLOBAL is that C^k may be so large that AC cannot effectively search it. We thus consider an extension of the *Hydra* methodology to the ACPP problem, which we dub PARHYDRA (see Algorithm 2). This method has the advantage that it adds and configures component solvers one at a time.

Algorithm 2: Portfolio Configuration Procedure parHydra

```
Input : parametric solver with configuration space $C$; desired number $k$
        of component solvers; instance set $I$; performance metric $m$;
        configurator $AC$; number $n$ of independent configurator runs;
        total configuration time $t$
Output: parallel portfolio solver with portfolio $\hat{c}_{1:k}$
1 for $i := 1 \dots k$ do
2     for $j := 1 \dots n$ do
3         obtain portfolio $c^{(j)}_{1:i} := \hat{c}_{1:i-1} || c^{(j)}$ by running $AC$ on configuration
          space $\{\hat{c}_{1:i-1}\} \times \{(c) \mid c \in C\}$ and initial incumbent $\hat{c}_{1:i-1} || c_{init}$ on
          $I$ using $m$ for time $t/(k \cdot n)$
4     let $\hat{c}_{1:i} \in \arg\min_{c^{(j)}_{1:i} | j \in \{1 \dots n\}} m(c^{(j)}_{1:i}, I)$ be the configuration which
      achieved best performance on $I$ according to $m$
5     let $c_{init} \in \arg\min_{c^{(j)} | j \in \{1 \dots n\}} m(\hat{c}_{1:i} || c^{(j)}, I)$ be the configuration that
      has the largest marginal contribution to $\hat{c}_{1:i}$
6 return $\hat{c}_{1:k}$
```

The key idea is to use AC only to configure the component solver added in the given iteration, leaving all other components clamped to the configurations that were determined for them in previous iterations. The procedure is greedy in the sense that in each iteration i, it attempts to add a component solver to the given portfolio $\hat{c}_{1:i-1}$ in a way that myopically optimizes the performance of the new portfolio $\hat{c}_{1:i}$ (Line 4). While the sets of n independent configurator runs in Line 2 can be performed in parallel (as in Global), the choice of the best-performing configuration $\hat{c}_{1:i}$ must be made after each iteration i, introducing a modest overhead compared to the cost of the actual configuration runs.

A disadvantage of the original *Hydra* approach is that it discards any intermediate results learned during configuration when it proceeds to the next iteration. In particular, configurations that were examined but not selected may turn out to be useful later on. We thus introduce a new idea here—which, indeed, can also be applied to the construction of portfolio-based algorithm selectors—as follows. We identify the unselected configuration $c^{(j)} \neq \hat{c}_{i:i}$ with the best marginal contribution to the current portfolio $\hat{c}_{1:i}$ (Line 5), and use it to initialize the configuration procedure in the next iteration (Line 3). This idea helps when using different initial configurations in each iteration more quickly guides the configuration procedure to complementary parts of the configuration space.

Another way that parHydra differs from the original *Hydra* methodology is that it runs entire portfolios on each instance considered during configuration. Because we target multicore machines, we consider these computational resources to be available without cost. While *Hydra* explicitly modifies the performance metric in each round, parHydra thus achieves the same modification implicitly, optimizing marginal contribution to the existing portfolio because only the ith

	wall clock time	CPU time
Global	$t/n + \epsilon$	$t + n \cdot k \cdot \epsilon$
parHydra	$t/n + k \cdot \epsilon$	$\sum_{i=1}^{k} i \cdot (\frac{t}{k} + n \cdot \epsilon)$

Table 1: Required wall clock time and CPU time of Global and parHydra for a configuration budget t, desired number k of component solvers, n algorithm configurator runs, $n \cdot k$ available CPU cores, and a small overhead ϵ for evaluating the performance of a parallel portfolio.

element of the portfolio is available to be configured in the ith iteration. Because parHydra only runs portfolios of size i in iteration i, if there is a cost to CPU cycles, we achieve some savings relative to Global in iterations $i < k$. If the overhead for the evaluation of the portfolios after each iteration is bounded by ϵ, the CPU cycles used in parHydra are bounded by $\sum_{i=1}^{k} i \cdot (\frac{t}{k} + n \cdot \epsilon)$ as compared to $t + n \cdot k \cdot \epsilon$ for Global. If $k > 1$ and $\frac{t}{k} > \epsilon$, parHydra will use fewer CPU cycles than Global. This is particularly important if ACPP is used on commercial cloud infrastructure, where saving CPU cycles means saving money. Table 1 gives an overview about the required wall clock time and CPU time for Global and parHydra.

Obviously, for $k > 1$, even if we assume that AC finds optimal configurations in each iteration, the parHydra procedure is not guaranteed to find a globally optimal portfolio. For instance, since the configuration found in the first iteration will be optimized to perform well on average on all instances I, the configuration added in the second iteration will then specialize to some subset of I. A combination of two configurations that are both specialized to different sets of instances may perform better; however, the configuration tasks in each parHydra iteration will be much easier than those performed by Global for even a moderately sized portfolio, giving us reason to hope that under realistic conditions, parHydra might perform better than Global, especially for large configuration spaces C and for comparatively modest time budgets t.

*3.1.3. Independent configuration of component solvers (*Clustering*)*

We also investigated adapting the *ISAC* approach [46, 57] to the ACPP setting. Specifically, we identified clusters in a space of instance features, ran a configurator to identify a configuration that performed well on each cluster, and combined these configurations into a parallel portfolio. However, our experiments (see on-line Appendix A) showed that this approach achieved consistently worse performance than Global and parHydra. In particular, we identified two main issues. First, normalization of instance features was very important; we struggled to determine a way of normalizing that produced good clusterings across different solvers. Second, we did not consistently observe that clusters of instances that were distinct in feature space necessarily led to solver configurations with complementary performance (which, obviously, is necessary for good performance in the ACPP setting). Thus, we do not further consider this approach in what follows.

3.2. Experiments

To empirically evaluate our methods for solving the ACPP problem, we applied GLOBAL and PARHYDRA to two state-of-the-art SAT solvers: *clasp* and *Lingeling*. Specifically, we compared our automatically configured parallel portfolios alongside performance-optimized sequential solvers, running on eight processor cores. Furthermore, we investigated the scalability of PARHYDRA by assessing the performance of our portfolio after each iteration, thereby also assessing the slowdown observed for increasing number of component solvers due to hardware bottlenecks. Finally, we integrated our configured portfolio based on *clasp* into *clasp*'s flexible multithreading architecture and configured the clause sharing policy to investigate the influence of clause sharing on our trained ACPP solvers.

3.2.1. Scenarios

We compared six evaluation scenarios for each solver. We denote the default configuration of a single-process solver as DEFAULT-SP and that of a multi-process solver with 8 processes and without clause sharing as DEFAULT-MP(8); DEFAULT-MP(8)+CS denotes the additional use of clause sharing, which is activated by default in both *Plingeling* and *clasp*. We contrasted these solver versions with three versions obtained using automated configuration: CONFIGURED-SP denotes the best (single-process) configuration obtained from configurator runs on a given training set, while GLOBAL-MP(8) and PARHYDRA-MP(8) represent the 8-component portfolios obtained using our GLOBAL and PARHYDRA methods. We chose this portfolio size to reflect widely available multi-core hardware, as used, for example, in the 2013 SAT Competition and also supported by the Amazon EC2 cloud (CC2 instances). We note that our approach is not inherently limited to eight cores and can be expected to scale to higher degrees of parallelism as long as sufficiently many complementary configurations can be found in the given design space.

3.2.2. Solvers

We applied our approach to the SAT solvers *clasp* version 2.1.3 [26] and *Lingeling* version ala [14]. We chose these two solvers because they were demonstrated to achieve state-of-the-art performance on combinatorial and industrial SAT instances in the 2012 SAT Challenge and therefore, represent an appropriately high bar for demonstrating the efficacy of our ACPP approach. Furthermore, both solvers are suitable for ACPP because they are highly parameterized; *clasp* has 81 parameters and *Lingeling* has 118. Hence, the configuration space for 8 processes has 648 parameters for *clasp* and 944 parameters for *Lingeling*.

We ruled out from our study other state-of-the-art parameterized solvers like *glucose* that have no parallelized counterpart for comparison with our automatically constructed solvers. We did not study *Plingeling*, the "official" parallel version of *Lingeling*, because it lacks configurable parameters for individual threads. We also disregarded the native parallel version of *clasp*, because *clasp*'s clause sharing mechanism, which cannot be turned off, results in highly non-deterministic runtime behaviour, rendering the configuration process much more difficult. Instead, we investigated the impact of clause sharing in a separate experiment. We executed all automatically constructed parallel portfolios via a simple wrapper script that runs a given number of solver instances independently in parallel and without communication between the component solvers.

3.2.3. Instance Sets

We conducted our experiments on instances from the *application* and *hard combinatorial* tracks of the 2012 SAT Challenge. Our configuration experiments made use of disjoint training and test sets, which we obtained by randomly splitting both instance sets into subsets with 300 instances each.[4]

To ensure that our experiments would complete within a feasible amount of time, we made use of an instance selection technique [37] on our training set to obtain a representative and effectively solvable subset of 100 instances for use with a runtime cutoff time of 180 seconds. We did this by (i) removing instances that we judged too easy and too hard from the instance set, (ii) clustering the instances in the feature space, and (iii) subsampling the instance set to ensure approximately equal coverage of the different clusters and normally distributed runtimes. As a reference for the selection process, we used the base features of *SATzilla* [73] and employed *SINN* [76], *Lingeling* [14], *glucose* [6], *clasp* [26] and *CCASat* [18] as representative set of state-of-the-art solvers, following [37].

3.2.4. Resource Limits and Hardware

We chose a cutoff time of 180 seconds for algorithm configuration on the training set and 900 seconds for evaluating solvers on the test set (as in the 2012 SAT Challenge). Additionally, we performed three repetitions of each solver and test instance run and report the median of those three runs. We restricted all solver runs (on both training and test sets) to use at most 12 GB of memory (as in the 2012 SAT Challenge). If a solver terminated because of memory limitations, we recorded it as a timeout. We performed all solver and configurator runs on Dell PowerEdge R610 systems with 48GB RAM and two Intel Xeon E5520 CPUs with four cores each (2.26GHz and 8MB Cache), running 64-bit Scientific Linux (2.6.18-348.6.1.el5).

3.2.5. Configuration Experiments

We performed configuration using *SMAC* (version 2.04.01) [41], a state-of-the-art algorithm configurator. *SMAC* allows the user to specify the initial incumbent, as required in the context of our parHydra approach (see Lines 2 and 5 of Algorithm 2). We specified PAR10 as our performance metric, and gave *SMAC* access to the base features of *SATzilla* [73]. (*SMAC* builds performance models internally; it can operate without instance features, but often performs

[4]A random split into training and test set is often used in machine learning to obtain unbiased performance estimates. However, such a simple split has a higher variance in its performance estimation than using a cross validation. Because of the large amount of CPU resources needed for our experiments, we could not afford to measure the performance of our ACPP methods on more splits, for example, based on cross validation.

	Lingeling (*application*)			*clasp* (*hard combinatorial*)		
Solver Set	#TOs	PAR10	PAR1	#TOs	PAR10	PAR1
DEFAULT-SP	72	2317	373	137	4180	481
CONFIGURED-SP	68	2204	368	140	4253	473
DEFAULT-MP(8)	64	2073	345	96	2950	358
DEFAULT-MP(8)+CS	**53***	**1730***	**299***	**90***	**2763***	**333***
GLOBAL-MP(8)	**52***	**1702***	**298***	98	3011	365
PARHYDRA-MP(8)	**55***†	**1788***†	**303***†	**96***†	**2945***†	**353***†

Table 2: Runtime statistics on the test set from *application* and *hard combinatorial* SAT instances achieved by single-processor (SP) and 8-processor (MP8) versions. DEFAULT-MP(8) was *Plingeling* in case of *Lingeling* and `clasp -t 8` for *clasp* where both use clause sharing (CS). The performance of a solver is shown in boldface if it was not significantly different from the best performance, and is marked with an asterisk (*) if it was not significantly worse than DEFAULT-MP(8)+CS (according to a permutation test with 100 000 permutations and significance level $\alpha = 0.05$). The best ACPP portfolio on the training set is marked with a dagger (†).

better when they are available.) To enable fair performance comparisons, in the case of CONFIGURED-SP ($n = 80$) and GLOBAL-MP(8) ($n = 10$) we allowed 80 hours of configuration time and 2 hours of validation time to determine the best-performing portfolio on the training instances from our 10 independent configuration runs, which amounts to a total of 6560 CPU hours for $k = 8$. For PARHYDRA-MP(8), we allowed for 10 hours of configuration time and 2 hours of validation time (ϵ) per configurator run ($n = 10$) in each iteration, amounting to a total of 3360 CPU hours (see Section 3.1.2). When using a cluster of dedicated machines with 8-core CPUs, each of these solver versions could be produced within 96 hours of wall-clock time.

3.2.6. Results and Interpretation

To evaluate our ACPP solvers, we present the number of timeouts (#TOs), PAR10 and PAR1 based on the median performance of the three repeated runs for each solver–test instance pair in Table 2. The best ACPP portfolio on the training set is marked with a dagger (†) to indicate that we would have chosen this portfolio if we had to make a choice only based on training data. Furthermore, we applied a statistical test (a permutation test with 100 000 permutations and significance level $\alpha = 0.05$) to the (0/1) timeout scores, the PAR10 scores and the PAR1 scores to determine whether performance differences between the solvers were significant. In Table 2, performance of a given solver is indicated in bold face if it was not significantly different from the performance of the best solver. We use an asterisk (*) to indicate that a given solver's performance was not significantly worse than the performance of DEFAULT-MP(8)+CS—the official parallel solver with clause sharing produced by experts.

Table 2 summarizes the results of our experiments with *Lingeling* and *clasp*. Running a configurator to obtain an improved, single-processor solver (CONFIGURED-SP) made a statistically insignificant impact on performance.

We thus believe that these default configurations are nearly optimal, reflecting the status of *Lingeling* and *clasp* as state-of-the-art solvers. With *Lingeling* as the component solver, Global-MP(8) produced the best-performing portfolio. There was no significant difference on any of these scores between parHydra-MP(8), Global-MP(8) and Default-MP(8)+CS. However, the portfolio performance of Default-MP(8) (*Plingeling* with deactivated clause sharing) was significantly worse than the performance of all other parallel portfolios and not even significantly better than Configured-SP in terms of timeout scores or PAR10 scores. Note that *Plingeling* (without clause sharing) builds a parallel portfolio only in a degenerate sense, simply using different random seeds and thus making different choices in the default phase [14]. Hence, it is not surprising that *Plingeling* without clause sharing performed significantly worse than *Plingeling* with clause sharing.

With *clasp* as the component solver, the portfolio constructed by parHydra-MP(8) was the best ACPP solver and matched (up to statistically insignificant differences) the performance of Default-MP(8)+CS (the expert-constructed portfolio solver with clause sharing) according to all metrics we considered, despite incurring six more timeouts. All other ACPP solvers fell short of this (high) bar; however, the portfolios of Global-MP(8) performed as well as the default portfolio of *clasp* without clause sharing (Default-MP(8)). All parallel solvers significantly outperformed the single-threaded versions of clasp.

Overall, parHydra-MP(8) was the only ACPP solver that matched the performance of Default-MP(8)+CS on both domains. parHydra-MP(8)'s portfolio had also the best training performance and therefore, out of the ACPP solvers, we would choose it. However, while Default-MP(8)+CS uses clause sharing, parHydra-MP(8) does not. This is surprising, because the performance of *Plingeling* and *clasp* without clause sharing was significantly worse than with clause sharing. Thus, parHydra-MP(8) was the best performing method among those that did not perform clause sharing.

3.2.7. Scalability and Overhead

Although 8-core machines have become fairly common, 4-core machines are still more commonly used as desktop computers. Furthermore, Asin et al. [4] observed that parallel portfolios scale sublinearly in the number of cores—in part, because component solvers share the same CPU cache. Therefore, we investigated how the performance of our automatically constructed portfolios scaled with the number of processors. The parHydra approach has the advantage that the portfolio is extended by one configuration at each iteration, making it easy to perform such scaling analysis.

Table 3 shows the test-set performance of parHydra-MP(i) after each iteration. First of all, parHydra-MP(1) was able to find a better performing configuration than Default-SP for *clasp*. In contrast, parHydra-MP(1) found a poorly performing configuration for *Lingeling* in comparison to Default-SP, and had to compensate in subsequent iterations. For both solvers, the largest performance improvement occurred between the first and second iterations, with the number of timeouts reduced by 17 for *Lingeling* and 18 for *clasp*. In later

	Lingeling (*application*)			*clasp* (*hard combinatorial*)		
Solver	#TOs	PAR10	PAR1	#TOs	PAR10	PAR1
Default-SP	72	2317	373	137	4180	481
parHydra-MP(1)	82	2594	380	136	4136	464
parHydra-MP(2)	65	2086	331	118	3607	421
parHydra-MP(3)	60	**1933**	313	115	3515	410
parHydra-MP(4)	**56**	**1874**	**308**	115	3507	402
parHydra-MP(5)	**58**	**1878**	312	105	3219	384
parHydra-MP(6)	60	1935	315	103	3161	380
parHydra-MP(7)	59	1902	309	102	3126	372
parHydra-MP(8)	**55**	**1788**	**303**	**96**	**2945**	**353**

Table 3: Runtime statistics of parHydra-MP(i) after each iteration i (test set). The performance of a solver is shown in boldface if it was not significantly different from the best performance, (according to a permutation test with 100 000 permutations and significance level $\alpha = 0.05$).

iterations, performance can stagnate or even drop: e.g., parHydra-MP(5) solves two more instances than parHydra-MP(6) with *Lingeling*. This may in part reflect hardware limitations: as the size of a portfolio increases, more processes compete for fixed memory (particularly, cache) resources.

We investigated the influence of these hardware limitations on the performance of our parallel solvers by constructing portfolios consisting of identical copies of the same solver. In particular, we replicated the same configuration multiple times with the same random seed; clearly, this setup should result in worsening performance as portfolio size increases, because each component solver does exactly the same work but shares hardware resources. (We note that these experiments are particularly sensitive to the underlying hardware we used.) To compare directly against Table 3, we used the configurations found in the first iteration of parHydra-MP(1). In Table 4, we see that hardware limitations did seem to impact the portfolio of *Lingeling* solvers; e.g., a single *Lingeling* configuration solved 10 more instances than eight such configurations running in parallel on an eight-core machine. In contrast, the performance of *clasp* varied only slightly as duplicate solvers were added. Based on the results in [1], we suspected that this overhead arose because of memory issues, noting that we evaluated *clasp* on *hard combinatorial* instances with an average size of 1.4 MB each, whereas we evaluated *Lingeling* on *application* instances with an average size of 36.7 MB. We confirmed that *clasp*'s portfolio also did experience overhead on instances with large memory consumption, and that *Lingeling* produced nearly no overhead on instances with low memory consumption.

An interesting further observation is that *Lingeling* and *clasp* performed best if two copies of the same configuration ran in parallel, and that running only one copy was worse than two copies. We speculate that this is caused by cache effects known to affect multi-core computations with more than one CPU. For example, the operating system may move a solver from one CPU to another, which may result in the loss of data in the CPU cache. However, if two solvers run on two CPUs, the operating system might run each of them on its own CPU

	Lingeling (*application*)			*clasp* (*hard combinatorial*)		
# Processes	#TOs	PAR10	PAR1	#TOs	PAR10	PAR1
1	82	2594	380	136	4136	464
2	**79**	**2509**	**376**	**134**	**4079**	**461**
3	**79**	**2509**	**376**	135	4106	451
4	85	2677	382	135	4107	452
5	86	2707	385	135	4108	463
6	89	2793	390	135	4110	465
7	90	2820	390	135	4110	465
8	92	2877	393	136	4139	467

Table 4: Runtime statistics of *Lingeling* and *clasp* with parallel runs of the same configuration on all instances in the corresponding test sets. The performance of a solver is shown in boldface if it was not significantly different from the best performance (according to a permutation test with 100 000 permutations and significance level $\alpha = 0.05$).

without moving them.

3.2.8. Algorithm Configuration of Clause Sharing

Our previous experiments did not allow our component solvers to share clauses, despite evidence from the literature that this can be very helpful [31]. The implementation of clause sharing is a challenging task; for example, if too many clauses are shared, the overhead caused by clause sharing may exceed the benefits [50]. Furthermore, the best clause sharing policy varies across instance sets and it is a tedious and time-consuming task to manually determine an effective clause sharing policy. A combination of ACPP and clause sharing will not completely compensate for human efforts to implement effective clause sharing, but ACPP can help developers to automatically determine well-performing clause sharing policies. In the following, we investigate the application of clause sharing to our ACPP portfolio. Since there are many possible clause sharing policies, we used algorithm configuration to identify effective clause sharing policies. This can be understood as an additional instrument for improving the performance of ACPP portfolios in cases where clause sharing is available.

To study the impact of clause sharing on our ACPP procedures, we relied upon the clause sharing infrastructure provided by *clasp* [26], which has a relatively highly parametrized clause sharing policy (10 parameters) and allows for the configuration of each component solver. *Plingeling*, on the other hand, does not support the configuration of each component solver. As before, we considered the *hard combinatorial* instance set.

We started with the portfolio identified by parHydra-MP(8). *clasp*'s multi-threading architecture performs preprocessing before threading is used. Hence, we ignored the preprocessing parameters identified in the parHydra-MP(8) portfolio, adding them again to the configuration space as global parameters. Since the communication of clause sharing induces greater variation in solving behaviour, we used 50 CPU hours as the configurator's time budget.

Table 5 shows the performance of *clasp*'s default portfolio with clause sharing, Default-MP(8)+CS; the portfolio originally returned by parHydra, which

clasp variant	#TOs	PAR10	PAR1
Default-MP(8)	96	2950	358
Default-MP(8)+CS	**90**	**2763**	**333**
parHydra-MP(8)	96	2945	**353**
parHydra-MP(8)+defCS	**90**	**2777**	**347**
parHydra-MP(8)+confCS	**88**	**2722**	**346**

Table 5: Runtime statistics of *clasp*'s parHydra-MP(8) portfolio with default clause sharing (defCS) and configured clause sharing (confCS) on the test instances of the *hard combinatorial* set. The performance of a solver is shown in boldface if its performance was at least as good as that of any other solver, up to statistically insignificant differences (according to a permutation test with 100 000 permutations and significance level $\alpha = 0.05$).

does not perform clause sharing, parHydra-MP(8); the application of *clasp*'s default clause sharing and preprocessing settings to the original parHydra portfolio, parHydra-MP(8)+defCS; and the parHydra portfolio with newly configured clause sharing and preprocessing settings, parHydra-MP(8)+confCS. As confirmed by these results, the use of clause sharing led to significant performance gains; furthermore, while the additional gains through configuring the clause sharing and preprocessing mechanisms were too small to reach statistical significance, parHydra-MP(8)+confCS solved two more instances than Default-MP(8)+CS and parHydra-MP(8)+defCS.

We note that there is potential for performance to be improved even further if clause sharing were configured alongside the portfolio itself. For example, *clasp*'s default portfolio contains configurations that are unlikely to solve instances directly, but that generate useful clauses for other clasp instances.[5] Clearly, our methodology for configuring clause sharing will not identify such configurations. Configuration of clause sharing can be directly integrated in Global and parHydra by adding the corresponding parameters to the configuration space, because the solvers actually run in parallel. However, since *clasp* with clause sharing is highly non-deterministic, the configuration process would require a larger time budget for constructing the portfolio. In a similar vein, some results in the literature indicate that the collaboration of SAT solvers via clause sharing performs better if the solvers use similar strategies, e.g., the same solver with a fixed configuration runs several times in parallel but with different seed (*cf. Plingeling*). If the configuration of the portfolio is performed alongside the configuration of the clause sharing policy, such homogeneous portfolios would also belong to the configuration space of our ACPP methods. We plan to investigate other approaches in future work.

3.2.9. Conclusion

Given a solver with a rich design space (such as *Lingeling* and *clasp*), all our ACPP methods were able to generate 8-core parallel solvers that significantly

[5]Personal communication with the main developer of *clasp*, Benjamin Kaufmann.

outperformed their sequential counterparts. We have thus demonstrated that our ACPP methods are able to automatically build parallel portfolio solvers, without the need for costly, hand-crafted parallel implementations. However, our scalability analysis indicates that hardware restrictions lead to substantial overhead as more processor cores are used, and the scalability of our ACPP methods depends on the richness of the given sequential solver's design spaces and the existence of complementary designs within these spaces. We were also able to verify that clause sharing can be used to further improve the performance of an ACPP solver, especially when configuration is performed alongside the component solver instances.

4. Parallel Portfolio Configuration with Multiple Sequential Solvers

So far, we have shown that our procedures were able to construct effective parallel portfolios based on single solvers with rich design spaces. There is considerable evidence from the literature and from SAT competitions that strong portfolios can also be built by combining entirely different solvers in their default configurations (see, e.g., SATzilla [73], *ppfolio* [64] and *pfolioUZK* [71]). For instance, *ppfolio* was obtained simply by combining the best solvers from the previous competition into a parallel portfolio. *pfolioUZK* included more state-of-the-art solvers from 2011 and relied on additional experiments to find the best combination of solvers in a portfolio. Neither portfolio considers the configuration space of the component solvers and therefore both can be seen as simple baselines for other parallelization approaches, including ours. However, *ppfolio* and *pfolioUZK* use *Plingeling* as a portfolio component. Since we aim to investigate the strength of our ACPP methods without additional human expert knowledge on parallel solving, we first consider only sequential solvers as the basis for our ACPP approach. This section and the following section investigates the extension of our automatic techniques to the construction of portfolios based on the configuration spaces spanned by such solver sets.

4.1. Approach

As long as all of our component solvers are sequential, we can simply use the ACPP procedures defined in Section 3. We can accommodate the multi-solver setting by introducing a solver choice parameter for each portfolio component (see Figure 1), and ensuring that the parameters of solver $a \in A$ are only active when the solver choice parameter is set to use a. This is implemented by using conditional parameters (see the PCS format of the Algorithm Configuration Library [44]). Similar architectures were used by *SATenstein* [48] and *Auto-WEKA* [67].

We have so far aimed to create portfolios with size equal to the number of available processor cores. But as observed in Section 3.2.7, each component solver used within a parallel portfolio incurs some overhead. A similar observation was made by the developer of *pfolioUZK* (personal communication) and prompted the decision for *pfolioUZK* to use only 7 components on an 8-core platform.

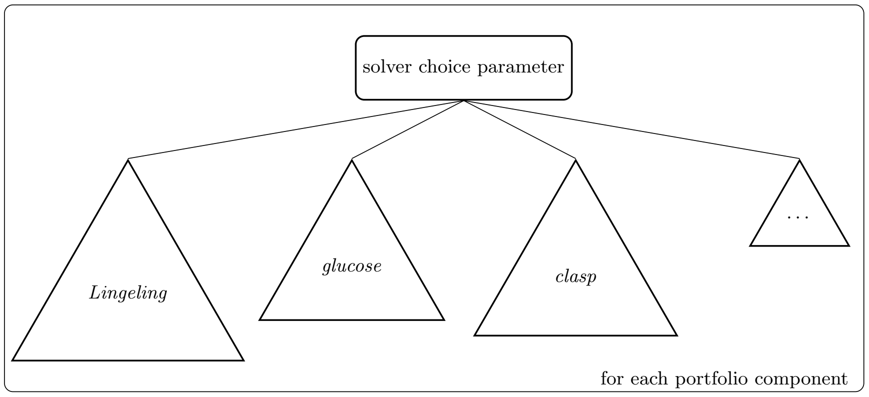

Figure 1: Using a solver choice parameter, we can specify a single configuration space that spans multiple solvers.

To allow our portfolios to make the same choice, we included "none" as one of choices available for each portfolio component.

4.2. Experiments

While we would presumably have obtained the strongest parallel solver by allowing our portfolio to include a very wide range of modern SAT solvers, this would have made it difficult to answer the question how our automated methods compare to human expertise in terms of the performance of the parallel portfolios thus obtained. In particular, we were interested in *pfolioUZK* [71], a parallel solver that won the parallel track of the 2012 SAT Challenge with *application* instances. To compare our automatic methods with the manual efforts of *pfolioUZK*'s authors, we thus chose the same set of solvers they considered as the basis for our experiments.

4.2.1. Solvers

pfolioUZK uses *satUZK*, *Lingeling*, *TNM*, and *MPhaseSAT_M* on the same core in its sequential version (Default-SP) and uses *satUZK*, *glucose*, *contrasat* and *Plingeling* with 4 threads and clause sharing in its 8-process parallel version (Default-MP(8)+CS). In all cases, solvers are used in their default configurations. However, in designing *pfolioUZK* [71], Wotzlaw et al. considered the following, larger set of component solvers:

- *contrasat* [69]: 15 parameters
- *glucose* 2.0 [6]: 10 parameters for *satelite* preprocessing and 6 for *glucose*
- *Lingeling* 587 [13]: 117 parameters
- *march_hi* 2009 [34]: 0 parameters
- *MPhaseSAT_M* [19]: 0 parameters
- *satUZK* [28]: 1 parameter

8-Processor Parallel Solver	#TOs	PAR10	PAR1
pfolioUZK-ST	150	4656	606
pfolioUZK-MP(8)+CS	**35**	**1168**	**223**
Global-MP(8)(*pfolioUZK* w/o *Plingeling*)	44	1463	275
parHydra-MP(8)(*pfolioUZK* w/o *Plingeling*)	**39**†	**1297**†	244†

Table 6: Runtime statistics for 8-processor parallel solvers on the *application* test set. The performance of a solver is shown in boldface if it was not significantly different from the best performance (according to a permutation test with 100 000 permutations at significance level $\alpha = 0.05$). The best ACPP portfolio on the training set is marked with a dagger (†).

- *sparrow2011* [68]: 0 parameters[6]
- *TNM* [51]: 0 parameters

Overall, the configuration space we considered has 150 parameters for each portfolio component (including the top-level parameter used to select a solver), and thus 1200 parameters for an 8-component parallel portfolio.

4.2.2. Instances and Setup

We evaluated *pfolioUZK* as well as our Global and parHydra approaches on the same 300 *application* test instances of the 2012 SAT Challenge as used before. Otherwise, our experimental setup was as described in Section 3.2.

4.2.3. Results and Interpretation

The first part of Table 6 shows the results of *pfolioUZK* in its sequential and parallel versions. Recall that *pfolioUZK* uses *Plingeling* with clause sharing as a component solver. Sequential *pfolioUZK* experienced 115 more timeouts than its parallel version; indeed, it was only ranked 16th in the sequential application track of the 2012 SAT Challenge.

The second part of Table 6 summarizes the performance of our ACPP solvers (which do not use *Plingeling* as a component solver). parHydra-MP(8) performed best; indeed, there was no significant difference between parHydra-MP(8) and *pfolioUZK*-MP(8) in terms of timeout and PAR10 scores. This indicates that our ACPP approach was indeed able to match the performance of parallel portfolios manually constructed by experts, even with the disadvantage of being prohibited from using *Plingeling* and thus clause sharing. Global-MP(8) performed significantly worse than *pfolioUZK*-MP(8), but not significantly worse than parHydra-MP(8) in terms of timeout and PAR10 scores.

Although we allowed our portfolio-building procedures to choose "none" for any component solver, this option was never selected.

[6]Although *sparrow2011* should be parameterized [68], the source code and binary provided with *pfolioUZK* does not expose any parameters.

4.2.4. Conclusion

We have demonstrated that by exploiting the configuration spaces of a set of complementary solvers, even-better-performing ACPP solvers can be obtained, compared to those constructed from a single parametric SAT solver such as *Lingeling* (compare Table 2 and Table 6). To produce such an ACPP solver, we did not need to modify our ACPP methods, but instead used conditionals in our configuration space to distinguish between the design spaces of the individual solvers. Although we did not use parallel solvers with clause sharing (such as *Plingeling*) in our portfolio, our parHydra method was able to generate a parallel solver without clause sharing that nevertheless performed as well as *pfolioUZK*.

5. Parallel Portfolio Configuration with Multiple Sequential and Parallel Solvers

Our results reported in Section 3.2.8 confirm the intuition that clause sharing is an important ingredient of high-performance parallel solvers. This section extends the scope of our ACPP methods to allow inclusion of parallel solvers that perform clause sharing as portfolio components. This way, we combine our automatic methods with the human expert knowledge inherent in existing clause sharing mechanisms to boost performance even further.

5.1. Approach: parHydra$_b$

To add parallel solvers as components in our ACPP approach, we consider each of them by adding multiple copies of the same solver, where each copy represents one thread of the parallel solver. Thereby, we mark parameters that have to be joined to be used across different cores; for example, the number of threads of a parallel solver. In contrast to other approaches that use scheduling (e.g., [54]), we do not have to decide on which core a solver runs, but only how many cores it will utilize.

The parHydra approach has a drawback when used to configure parallel SAT solvers. This can be seen when considering the solvers *Lingeling* and *Plingeling*. First of all, the components of *Plingeling* are not parameterized, and we can only choose the number of threads it is assigned. If the portfolio can also consist of configured versions of *Lingeling*, which subsumes single-core *Plingeling*, and the configurator is run for long enough, there is no reason for the parHydra approach to choose *Plingeling* as a component, unless *Plingeling* already belongs to the previous iteration's portfolio (in which case the benefits of clause sharing can make themselves felt). Obviously then, an argument by induction shows that *Plingeling* will *never* be preferred by parHydra, revealing a disadvantage of its greedy optimization strategy. In contrast, Global does not have this problem, but has difficulties dealing with the large configuration space encountered here.

To overcome both of these limitations and effectively interpolate between parHydra and Global, we introduce a new approach, which we call parHydra$_b$

Algorithm 3: Portfolio Configuration Procedure parHydra$_b$

```
Input  : set of parametric solvers $a \in A$ with configuration space $C_a$;
         desired number $k$ of component solvers; number $b$ of component
         solvers simultaneously configured per iteration; instance set $I$;
         performance metric $m$; configurator $AC$; number $n$ of
         independent configurator runs; total configuration time $t$
Output : parallel portfolio solver with portfolio $\hat{c}_{1:k}$
1 $i := 1$
2 while $i < k$ do
3     $i' := i + b - 1$
4     for $j := 1..n$ do
5         obtain portfolio $c^{(j)}_{1:i'} := \hat{c}_{1:i-1} || c^{(j)}_{i:i'}$ by running $AC$ on configuration
          space $\{\hat{c}_{1:i-1}\} \times (\prod_{l=i}^{i'} \bigcup_{a \in A} \{(c) \mid c \in C_a\})$ and initial incumbent
          $\hat{c}_{1:i-1} || c_{init}$ on $I$ using $m$ for time $t \cdot b/(k \cdot n)$
6     let $\hat{c}_{1:i'} \in \arg\min_{c^{(j)}_{1:i'} | j \in \{1 \ldots n\}} m(c^{(j)}_{1:i'}, I)$ be the configuration that
      achieved best performance on $I$ according to $m$
7     let $c_{init} \in \arg\min_{c^{(j)}_{i:i'} | j \in \{1 \ldots n\}} m(\hat{c}_{1:i'} || c^{(j)}_{i:i'}, I)$ be the configuration that
      has the largest marginal contribution to $\hat{c}_{1:i'}$
8     $i := i + b$
9 return $\hat{c}_{1:k}$
```

(Algorithm 3). In brief, unlike parHydra, parHydra$_b$ simultaneously configures b processes in each iteration. Specifically, in Lines 2 and 3, parHydra$_b$ iterates up to the desired number of component solvers with a step size of b; in Line 5, the algorithm configurator is used to find a portfolio of b configurations with b times the configuration time budget and adds them to the current portfolio $c^{(j)}_{1:i'}$. After the n independent runs of the algorithm configurator (Line 4 and 5), the best performing portfolio $\hat{c}_{1:i'}$ is selected in Line 6, and in Line 7, the initial incumbent for the next iteration is selected based on the marginal contribution to the currently selected portfolio. The parameter b controls the size of the configuration space in each iteration. Since the configuration space grows exponentially with b but we allow configuration time to grow only linearly, the algorithm configurator has a harder task under parHydra$_b$ than under parHydra. However, for sufficiently small b, this additional cost can be worthwhile, because of parHydra$_b$'s reduced tendency to stagnate in local minima.

5.2. Experiments

We used the set of solvers described in Section 4.2, with the addition of *Plingeling*. We added parHydra$_b$ to the set of ACPP methods considered and allowed $b \in \{2, 4\}$. We use the same setup as before, except that we allowed a 20-hour configuration budget per configured process, twice as much as before, to take into consideration the greater variation in solving behaviour of *Plingeling* which induces a harder configuration task.

We compared our results to a variety of state-of-the-art solvers from the 2012 SAT Challenge on this benchmark set. We considered two state-of-the-art sequential solvers: *glucose* (2.1) [6] (winner of the single-engine application track—like all other competition results cited below, in the 2012 SAT Challenge); and *SATzilla-App* [75], which is *SATzilla* trained on application instances (winner of the sequential portfolio application track). We also considered the following high-performance parallel solvers[7]:

- *clasp* (2.1.3) [26];
- *Plingeling* (ala) [14] and *Plingeling* (aqw) [15][8];
- *ppfolio* [64] (bronze medal in the parallel track);
- *PeneLoPe* [5] (silver medal in the parallel track);
- and again *pfolioUZK* [71] (winner of the parallel track).

The first part of Table 7 summarizes the performance results for these solvers: first the sequential solvers in their default configurations (DEFAULT-SP), then the parallel solvers using clause sharing in their default configurations (DEFAULT-MP(8)+CS), and finally our ACPP solvers based on the component solvers of *pfolioUZK*. As already discussed, the performance of the sequential *pfolioUZK* did not achieve state-of-the-art performance; this distinction goes to *glucose* for a single solver, and *SATzilla* for a portfolio-based algorithm selector.

pfolioUZK and *clasp* performed significantly better than *ppfolio*, *PeneLoPe* and *Plingeling*; we observed no significant performance difference between *pfolioUZK* and *clasp* in terms of any of the scores we measured. (Even with further, extensive experiments, we have not been able to determine why *clasp* performed significantly worse than *pfolioUZK* and *Lingeling* in the 2012 SAT Challenge.)

parHYDRA$_4$-MP(8) produced the best parallel portfolio solver overall, which turned out to be significantly faster than *pfolioUZK*. The portfolio solvers produced by parHYDRA-MP(8) and parHYDRA$_2$-MP(8) exhibited no significant performance differences from *pfolioUZK*. Furthermore, parHYDRA$_4$-MP(8) also solved more instances than *Plingeling*(aqw), although *Plingeling*(aqw) won the 2013 SAT competition and the solvers in parHYDRA$_4$-MP(8) were mostly published in 2011, which gives *Plingeling*(aqw) an advantage of two additional years of development.

[7]We did not consider the parallel algorithm selection solvers *p3S* and *parCSHC*, since the only versions available are optimized for a mixed set of SAT instances (application, handcrafted and random) and there is no trainable version available. Therefore, we had no way of performing a fair comparison between those methods and our ACPP portfolios.

[8]The work we describe in this study took more than a year. In the meantime, the 2013 SAT Competition took place and the new *Plingeling* version aqw won the gold medal in the parallel track.

Solver	#TOs	PAR10	PAR1
Single threaded solvers: DEFAULT-SP			
pfolioUZK-ST	150	4656	606
glucose-2.1	55	1778	293
SATzilla-2012-*APP*	38	1289	263
Parallel solvers with default config: DEFAULT-MP(8)			
Plingeling(ala)+CS	53	1730	299
PeneLoPe+CS	49	1563	240
ppfolio+CS	46	1506	264
clasp+CS	37	**1203**	**204**
pfolioUZK-MP8+CS	35	1168	223
Plingeling(aqw)+CS	**32**	**1058**	**194**
ACPP solvers including a parallel solver			
PARHYDRA-MP(8)(*pfolioUZK*)	34	1143	225
PARHYDRA$_2$-MP(8)(*pfolioUZK*)	32	1082	218
PARHYDRA$_4$-MP(8)(*pfolioUZK*)	**29**†	**992**†	**209**†
GLOBAL-MP(8)(*pfolioUZK*)	35	1172	227

Table 7: Comparison of parallel solvers with 8 processors on the test set of *application*. The performance of a solver is shown in boldface if its performance was at least as good as that of any other solver, up to statistically insignificant differences (according to a permutation test with 100 000 permutations at significance level $\alpha = 0.05$). The best ACPP portfolio on the training set is marked with a dagger (†).

Taking a closer look at these portfolio solvers, PARHYDRA$_2$-MP(8), PARHYDRA$_4$-MP(8) and GLOBAL-MP(8) allocated three cores to *Plingeling*. As expected, PARHYDRA-MP(8) did not include *Plingeling* in its portfolio; however, it did include three variants of *Lingeling*. All four portfolio solvers used at most seven processes by selecting “none” on one process; GLOBAL-MP(8) selected “none” twice.

5.3. Comparison with Sequential Portfolio Solvers

As illustrated in Table 7, our ACPP portfolios outperformed *SATzilla*—the winning sequential portfolio solver of the SAT Challenge 2012. However, *SATzilla* used a different set of component solvers. Therefore, one might wonder how well a sequential portfolio solver could perform when using our ACPP methods to obtain a configured portfolio. For all sequential portfolio solvers, such as algorithm selection or scheduling systems, without communication between the components, the best possible performance is achieved by the virtual best solver (VBS). We thus compared such a VBS to our ACPP method. Specifically, we assessed the performance of all components of our best-performing parallel portfolio that does not use any parallel solvers: PARHYDRA-MP(8)(*pfolioUZK* w/o *Plingeling*) (see Table 6). In contrast to PARHYDRA-MP(8)(*pfolioUZK* w/o *Plingeling*), which gave rise to 39 timeouts, the VBS of PARHYDRA-MP(8)(*pfolioUZK* w/o *Plingeling*)'s components gave rise to 35 timeouts. This performance difference arises due to hardware overhead, as discussed earlier. Comparing this VBS performance with our PARHYDRA$_4$-MP(8) with 29 timeouts (see Table 7), we conclude that no sequential portfolio solver would have been able to outperform

	Plingeling(ala)			parHydra$_4$-MP(8)		
#Processes	#TOs	PAR10	PAR1	#TOs	PAR10	PAR1
4	27	938	209	34	1137	219
8	30	1009	199	22	766	172
12	28	950	194	22	761	167
16	28	949	193	25	845	25

Table 8: Comparing *Plingeling* (ala) and parHydra$_4$-MP(8) with increasing number of cores where parHydra$_4$-MP(8) with more than 8 cores used more threads for *Plingeling*.

our parHydra$_4$-MP(8) portfolios. parHydra$_4$-MP(8) has a speedup of 1.18 on PAR10 (VBS: 1173 vs. parHydra$_4$-MP(8): 992) and 1.09 on PAR1 (VBS: 228 vs. parHydra$_4$-MP(8): 209).

5.4. Scaling to more than 8 cores

Our ACPP methods are able to take advantage of an arbitrary number of cores, as long we can find a sufficient number of complementary solver configurations within the given configuration space. The comparison of parHydra-MP(8) with only *Lingeling* (Section 3.2) and with the solvers of *pfolioUZK* demonstrated that a more extensive configuration space with several solvers can lead to better performance (compare Table 2 and 7). However, parHydra$_4$-MP(8)(*pfolioUZK*) used only 7 out of 8 available CPU cores. This indicates that the configuration space of parHydra$_4$-MP(8)(*pfolioUZK*) was relatively exhausted, to the point where running a further solver produced less benefit than incurring additional hardware overhead.

Looking at the training performance of parHydra$_4$-MP(8)(*pfolioUZK*), we note that the improvement between the first and second iterations (first and last four components, respectively) of parHydra$_4$-MP(8) was less than 10%. The performance improvement achieved by the more fine-grained parHydra$_2$-MP(8)(*pfolioUZK*) between its third and fourth iterations was even lower, less than 3%. Indeed, the majority of our *SMAC* runs (7 out of 10) found similarly performing portfolios after their last iterations (with a difference of less than 1 CPU seconds), and one of these 7 portfolios showed the overall best performance on our training set. Therefore, given the configuration space we studied, we do not expect the potential for substantial performance improvements by leveraging more than 8 cores.

Using a parallel solver with clause sharing in our ACPP portfolios, we expect that performance could always be improved by increasing the number of parallel threads. Therefore, we studied the effect of increasing the number of parallel threads of *Plingeling* (ala) in parHydra$_4$-MP(8)(*pfolioUZK*) by using more than 8 cores. Since the machines we used for our previous experiments had only 8 cores, we used another cluster for the following experiment, consisting of machines with 64GB memory and two Intel Xeon E5-2650v2 8-core CPUs with 2.60GHz and 20 MB L2 cache each, running 64-bit Ubuntu 14.04 LTS.

Table 8 shows the scalability of *Plingeling* (ala) and parHydra$_4$-MP(8)(*pfolioUZK*) in steps of 4 processes, since parHydra$_4$-MP(8)(*pfolioUZK*) also adds 4 components at a time. On this new hardware, we observed that hardware overhead influenced performance less than in our previous experiments. parHydra$_4$-MP(8)(*pfolioUZK*) reached a performance peak at 12 processes and performed worse when using all 16 cores. Furthermore, parHydra$_4$-MP(8)(*pfolioUZK*) did not solve more instances when using additional *Plingeling* threads; we note that the original parHydra$_4$-MP(8) already used 3 threads for *Plingeling*. However, the average runtime (PAR1) of parHydra$_4$-MP(8)(*pfolioUZK*) slightly improved between 8 and 12 cores. Running only *Plingeling* had similar effects; *Plingeling* performance improved as cores were added up to 12 and then stagnated.

Based on these results, we conjecture that the number of CPU cores at which hardware overhead becomes important is higher on newer hardware; indeed, perhaps future hardware architectures will permit running even larger parallel portfolios on one machine without significant hardware overhead. We also observe that adding a reasonable number of additional threads to *Plingeling* did not substantially improve the performance of parHydra-MP(8)(*pfolioUZK*) .

5.4.1. Conclusion

Using our extended parHydra$_b$ method and a parallel solver with clause sharing, we were able to automatically generate an ACPP solver that outperformed *pfolioUZK* and reached the performance level of *Plingeling*(aqw), which is based on considerably more advanced solving strategies than are used in the baseline portfolio from *pfolioUZK*. This shows that the combination of our automatic ACPP methods and expert knowledge can be used not only to generate efficient parallel solvers, but also to automatically (albeit slightly) improve *Plingeling*(aqw), the 2013 state of the art in parallel SAT solving.

6. Conclusions and Future Work

In this work, we demonstrated that sequential algorithms can be combined automatically and effectively into parallel portfolios, following an approach we call Automatic Construction of Parallel Portfolios (ACPP). This approach enables solver developers to leverage parallel resources without having to be concerned with synchronization, race conditions or other difficulties that arise in the explicit design of parallel code. Of course, inherently parallel solving techniques (e.g., based on clause sharing) can further improve the performance of our ACPP portfolios. In this view, ACPP can also be used to support a human developer by determining a well-performing parallel portfolio which can provide a base for (i) adding clause sharing, (ii) identifying complementary configurations or (iii) as starting point for further manual fine-tuning and development of new techniques.

We investigated two different ACPP procedures: (i) configuration in the joint configuration space of all portfolio components (Global); ; and (ii) iteratively adding one or more component solvers at a time (parHydra). We assessed these procedures on widely studied classes of satisfiability problems: the *application* and *hard combinatorial* tracks of the 2012 SAT Challenge. Overall, we found that PARHYDRA was the most practical method. The configuration space of GLOBAL grows exponentially with the size of the portfolio; thus, while in principle it subsumes the other methods, in practice, it tended to find worse portfolios than PARHYDRA within available time budgets. In contrast to GLOBAL, PARHYDRA was able to find well-performing portfolios on all of our domains; using *pfolioUZK*'s solvers on *application* instances, it even was able to reach the performance level of *Plingeling*(aqw), which won the 2013 parallel track. We expect that as additional highly parametric SAT solvers become available, PARHYDRA will produce even stronger parallel portfolios.

In future work, it would be interesting to investigate how information exchange strategies such as clause sharing can be integrated more deeply into our procedures. This could be done, e.g., by combining our ACPP approach with *HordeSAT* [9], a modular, massively parallel SAT solver with clause sharing that can make use of arbitrary CDCL solvers. Since parameters governing such information exchange are global (rather than restricted to an individual component solver), we also intend to investigate improved methods for handling global portfolio parameters. Finally, we plan to investigate ways of reusing previously-trained portfolios for building new ones, for instance, in cases where the instance set changes slightly or new solvers become available.

Acknowledgments

M. Lindauer was supported by the DFG (German Research Foundation) under Emmy Noether grant HU 1900/2-1 and project SCHA 550/8-3, H. Hoos and K. Leyton-Brown by NSERC Discovery Grants, and T. Schaub by the DFG under project SCHA 550/8-3, respectively.

[1] Aigner, M., Biere, A., Kirsch, C., Niemetz, A., Preiner, M., 2013. Analysis of portfolio-style parallel SAT solving on current multi-core architectures. In: Proceeding of the Fourth International Workshop on Pragmatics of SAT (POS'13).

[2] Amadini, R., Gabbrielli, M., Mauro, J., 2015. A multicore tool for constraint solving. In: Yang, Q., Wooldridge, M. (Eds.), Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence (IJCAI'15). AAAI Press, pp. 232–238.

[3] Ansótegui, C., Sellmann, M., Tierney, K., 2009. A gender-based genetic algorithm for the automatic configuration of algorithms. In: Gent, I. (Ed.), Proceedings of the Fifteenth International Conference on Principles and Practice of Constraint Programming (CP'09). Vol. 5732 of Lecture Notes in Computer Science. Springer-Verlag, pp. 142–157.

[4] Asin, R., Olate, J., Ferres, L., 2013. Cache performance study of portfolio-based parallel CDCL SAT solvers. CoRR abs/1309.3187 (v1).

[5] Audemard, G., Hoessen, B., Jabbour, S., Lagniez, J.-M., Piette, C., 2012. Penelope, a parallel clause-freezer solver. In: [7], pp. 43–44, available at `https://helda.helsinki.fi/handle/10138/34218`.

[6] Audemard, G., Simon, L., 2012. Glucose 2.1. in the SAT challenge 2012. In: [7], pp. 23–23, available at `https://helda.helsinki.fi/handle/10138/34218`.

[7] Balint, A., Belov, A., Diepold, D., Gerber, S., Järvisalo, M., Sinz, C. (Eds.), 2012. Proceedings of SAT Challenge 2012: Solver and Benchmark Descriptions. Vol. B-2012-2 of Department of Computer Science Series of Publications B. University of Helsinki, available at `https://helda.helsinki.fi/handle/10138/34218`.

[8] Balint, A., Belov, A., Heule, M., Järvisalo, M. (Eds.), 2013. Proceedings of SAT Competition 2013: Solver and Benchmark Descriptions. Vol. B-2013-1 of Department of Computer Science Series of Publications B. University of Helsinki.

[9] Balyo, T., Sanders, P., Sinz, C., 2015. HordeSat: A massively parallel portfolio SAT solver. In: Heule, M., Weaver, S. (Eds.), Proceedings of the International Conference on Theory and Applications of Satisfiability Testing (SAT'15). Vol. 9340 of Lecture Notes in Computer Science. Springer-Verlag, pp. 156–172.

[10] Baral, C., 2003. Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press.

[11] Belov, A., Diepold, D., Heule, M., Järvisalo, M. (Eds.), 2014. Proceedings of SAT Competition 2014: Solver and Benchmark Descriptions. Vol. B-2014-2 of Department of Computer Science Series of Publications B. University of Helsinki.

[12] Biere, A., 2010. Lingeling, Plingeling, PicoSAT and PrecoSAT at SAT race 2010. Tech. Rep. 10/1, Institute for Formal Models and Verification. Johannes Kepler University.

[13] 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.

[14] Biere, A., 2012. Lingeling and friends entering the SAT challenge 2012. In: [7], pp. 33–34, available at `https://helda.helsinki.fi/handle/10138/34218`.

[15] Biere, A., 2013. Lingeling, plingeling and treengeling entering the sat competition 2013. In: [8], pp. 51–52.

[16] Biere, A., 2014. Yet another local search solver and lingeling and friends entering the SAT competition 2014. In: [11], pp. 39–40.

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

[18] Cai, S., Luo, C., Su, K., 2012. CCASAT: Solver description. In: [7], pp. 13–14, available at https://helda.helsinki.fi/handle/10138/34218.

[19] Chen, J., 2011. Phase selection heuristics for satisfiability solvers. CoRR abs/1106.1372 (v1).

[20] Cimatti, A., Sebastiani, R. (Eds.), 2012. Proceedings of the Fifteenth International Conference on Theory and Applications of Satisfiability Testing (SAT'12). Vol. 7317 of Lecture Notes in Computer Science. Springer-Verlag.

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

[22] Eén, N., Sörensson, N., 2004. An extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (Eds.), Proceedings of the Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT'03). Vol. 2919 of Lecture Notes in Computer Science. Springer-Verlag, pp. 502–518.

[23] G. Audemard, B. H., Jabbour, S., Lagniez, J., Piette, C., 2014. PeneLoPe in SAT competition 2014. In: [11], pp. 58–59.

[24] Gagliolo, M., Schmidhuber, J., 2006. Learning dynamic algorithm portfolios. Annals of Mathematics and Artificial Intelligence 47 (3-4), 295–328.
URL http://www.springerlink.com/content/g10248526jq91k52/

[25] Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T., 2012. Answer Set Solving in Practice. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan and Claypool Publishers.

[26] Gebser, M., Kaufmann, B., Schaub, T., 2012. Multi-threaded ASP solving with clasp. Theory and Practice of Logic Programming 12 (4-5), 525–545.

[27] Gomes, C., Selman, B., 2001. Algorithm portfolios. Artificial Intelligence 126 (1-2), 43–62.

[28] Grinten, A., Wotzlaw, A., Speckenmeyer, E., Porschen, S., 2012. satUZK: Solver description. In: [7], pp. 54–55, available at https://helda.helsinki.fi/handle/10138/34218.

[29] Guo, L., Hamadi, Y., Jabbour, S., Sais, L., 2010. Diversification and intensification in parallel SAT solving. In: Cohen, D. (Ed.), Proceedings of the Sixteenth International Conference on Principles and Practice of Constraint Programming (CP'10). Vol. 6308 of Lecture Notes in Computer Science. Springer-Verlag, pp. 252–265.

[30] Hamadi, Y., Jabbour, S., Sais, L., 2009. Control-based clause sharing in parallel SAT solving. In: [17], pp. 499–504.

[31] Hamadi, Y., Jabbour, S., Sais, L., 2009. ManySAT: a parallel SAT solver. Journal on Satisfiability, Boolean Modeling and Computation 6, 245–262.

[32] Hamadi, Y., Schoenauer, M. (Eds.), 2012. Proceedings of the Sixth International Conference Learning and Intelligent Optimization (LION'12). Vol. 7219 of Lecture Notes in Computer Science. Springer-Verlag.

[33] Hamadi, Y., Wintersteiger, C., 2013. Seven challenges in parallel SAT solving. AI Magazine 34, 99–106.

[34] Heule, M., Dufour, M., van Zwieten, J., van Maaren, H., 2004. March_eq: Implementing additional reasoning into an efficient look-ahead SAT solver. In: Hoos, H., Mitchell, D. (Eds.), Proceedings of the Seventh International Conference on Theory and Applications of Satisfiability Testing (SAT'04). Vol. 3542 of Lecture Notes in Computer Science. Springer-Verlag, pp. 345–359.

[35] Hoos, H., 2012. Programming by optimisation. Communications of the ACM 55, 70–80.

[36] Hoos, H., Kaminski, R., Schaub, T., Schneider, M., 2012. aspeed: ASP-based solver scheduling. In: Dovier, A., Santos Costa, V. (Eds.), Technical Communications of the Twenty-eighth International Conference on Logic Programming (ICLP'12). Vol. 17. Leibniz International Proceedings in Informatics (LIPIcs), pp. 176–187.

[37] Hoos, H., Kaufmann, B., Schaub, T., Schneider, M., 2013. Robust benchmark set selection for boolean constraint solvers. In: Pardalos, P., Nicosia, G. (Eds.), Proceedings of the Seventh International Conference on Learning and Intelligent Optimization (LION'13). Vol. 7997 of Lecture Notes in Computer Science. Springer-Verlag, pp. 138–152.

[38] Hoos, H., Leyton-Brown, K., Schaub, T., Schneider, M., 2012. Algorithm configuration for portfolio-based parallel SAT-solving. In: Coletta, R., Guns, T., O'Sullivan, B., Passerini, A., Tack, G. (Eds.), Proceedings of the First Workshop on Combining Constraint Solving with Mining and Learning (CoCoMile'12). pp. 7–12.

[39] Hoos, H., Stützle, T., 2004. Stochastic Local Search: Foundations and Applications. Elsevier/Morgan Kaufmann.

[40] Huberman, B., Lukose, R., Hogg, T., 1997. An economic approach to hard computational problems. Science 275, 51–54.

[41] Hutter, F., Hoos, H., Leyton-Brown, K., 2011. Sequential model-based optimization for general algorithm configuration. In: Proceedings of the Fifth International Conference on Learning and Intelligent Optimization

(LION'11). Vol. 6683 of Lecture Notes in Computer Science. Springer-Verlag, pp. 507–523.

[42] Hutter, F., Hoos, H., Leyton-Brown, K., 2014. Submodular configuration of algorithms for portfolio-based selection. Tech. rep., Department of Computer Science, University of British Columbia, to appear.

[43] Hutter, F., Hoos, H., Leyton-Brown, K., Stützle, T., 2009. ParamILS: An automatic algorithm configuration framework. Journal of Artificial Intelligence Research 36, 267–306.

[44] Hutter, F., López-Ibáñez, M., Fawcett, C., Lindauer, M., Hoos, H., Leyton-Brown, K., Stützle, T., 2014. AClib: a benchmark library for algorithm configuration. In: Pardalos, P., Resende, M., Vogiatzis, C., Walteros, J. (Eds.), Proceedings of the Eigth International Conference on Learning and Intelligent Optimization (LION'14). Vol. 8426 of Lecture Notes in Computer Science. Springer-Verlag, pp. 36–40.

[45] Kadioglu, S., Malitsky, Y., Sabharwal, A., Samulowitz, H., Sellmann, M., 2011. Algorithm selection and scheduling. In: Lee, J. (Ed.), Proceedings of the Seventeenth International Conference on Principles and Practice of Constraint Programming (CP'11). Vol. 6876 of Lecture Notes in Computer Science. Springer-Verlag, pp. 454–469.

[46] Kadioglu, S., Malitsky, Y., Sellmann, M., Tierney, K., 2010. ISAC – instance-specific algorithm configuration. In: Coelho, H., Studer, R., Wooldridge, M. (Eds.), Proceedings of the Nineteenth European Conference on Artificial Intelligence (ECAI'10). IOS Press, pp. 751–756.

[47] Katsirelos, G., Sabharwal, A., Samulowitz, H., Simon, L., 2013. Resolution and parallelizability: Barriers to the efficient parallelization of SAT solvers. In: desJardins, M., Littman, M. (Eds.), Proceedings of the Twenty-Seventh National Conference on Artificial Intelligence (AAAI'13). AAAI Press.

[48] KhudaBukhsh, A., Xu, L., Hutter, F., Hoos, H., Leyton-Brown, K., 2009. SATenstein: Automatically building local search SAT solvers from components. In: [17], pp. 517–524.

[49] Kotthoff, L., 2012. Algorithm selection for combinatorial search problems: A survey. Tech. rep., University College Cork.

[50] Lazaar, N., Hamadi, Y., Jabbour, S., Sebag, M., 2012. Cooperation control in parallel SAT solving: a multi-armed bandit approach. Tech. rep., INRIA. URL `http://hal.inria.fr/hal-00733282`

[51] Li, C., Wei, W., Li, Y., 2012. Exploiting historical relationships of clauses and variables in local search for satisfiability. In: [20], pp. 479–480.

[52] Lindauer, M., Hoos, H., , Hutter, F., 2015. From sequential algorithm selection to parallel portfolio selection. In: Proceedings of the International Conference on Learning and Intelligent Optimization (LION'15). pp. 1–16.

[53] López-Ibáñez, M., Dubois-Lacoste, J., Stützle, T., Birattari, M., 2011. The irace package, iterated race for automatic algorithm configuration. Tech. rep., IRIDIA, Université Libre de Bruxelles, Belgium.
URL `http://iridia.ulb.ac.be/IridiaTrSeries/IridiaTr2011-004.pdf`

[54] Malitsky, Y., Sabharwal, A., Samulowitz, H., Sellmann, M., 2012. Parallel SAT solver selection and scheduling. In: Milano, M. (Ed.), Proceedings of the Eighteenth International Conference on Principles and Practice of Constraint Programming (CP'12). Vol. 7514 of Lecture Notes in Computer Science. Springer-Verlag, pp. 512–526.

[55] Malitsky, Y., Sabharwal, A., Samulowitz, H., Sellmann, M., 2013. Algorithm portfolios based on cost-sensitive hierarchical clustering. In: Rossi, F. (Ed.), Proceedings of the Twenty-third International Joint Conference on Artificial Intelligence (IJCAI'13). IJCAI/AAAI, pp. 608–614.

[56] Malitsky, Y., Sabharwal, A., Samulowitz, H., Sellmann, M., 2013. Parallel lingeling, ccasat, and csch-based portfolio. In: [8], pp. 26–27.

[57] Malitsky, Y., Sellmann, M., 2012. Instance-specific algorithm configuration as a method for non-model-based portfolio generation. In: Beldiceanu, N., Jussien, N., Pinson, E. (Eds.), CPAIOR. Vol. 7298 of Lecture Notes in Computer Science. Springer-Verlag, pp. 244–259.

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

[59] Nudelman, E., Leyton-Brown, K., Andrew, G., Gomes, C., McFadden, J., Selman, B., Shoham, Y., 2003. Satzilla 0.9, solver description, International SAT Competition.

[60] Núnez, S., Borrajo, D., López, C., 2013. Mipsat. In: [8], pp. 59–60.

[61] O'Mahony, E., Hebrard, E., Holland, A., Nugent, C., O'Sullivan, B., 2008. Using case-based reasoning in an algorithm portfolio for constraint solving. In: Bridge, D., Brown, K., O'Sullivan, B., Sorensen, H. (Eds.), Proceedings of the Nineteenth Irish Conference on Artificial Intelligence and Cognitive Science (AICS'08).

[62] Papadimitriou, C., Steiglitz, K., 1982. Combinatorial Optimization: Algorithms and Complexity. Prentice-Hall, Upper Saddle River, NJ, USA.

[63] Petrik, M., Zilberstein, S., 2006. Learning static parallel portfolios of algorithms. In: Proceedings of the International Symposium on Artificial Intelligence and Mathematics (ISAIM 2006).

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

[65] Schrijver, A., 1986. Theory of Linear and Integer Programming. John Wiley & sons, New York, NY, USA.

[66] Soos, M., Nohl, K., Castelluccia, C., 2009. Extending SAT solvers to cryptographic problems. In: Kullmann, O. (Ed.), Proceedings of the Twelfth International Conference on Theory and Applications of Satisfiability Testing (SAT'09). Vol. 5584 of Lecture Notes in Computer Science. Springer-Verlag, pp. 244–257.

[67] Thornton, C., Hutter, F., Hoos, H., Leyton-Brown, K., 2013. Auto-WEKA: Combined selection and hyperparameter optimization of classification algorithms. In: Proceedings of the 19th International Conference on Knowledge Discovery and Data Mining (KDD'13). pp. 847–855.

[68] Tompkins, D., Balint, A., Hoos, H., 2011. Captain Jack – new variable selection heuristics in local search for SAT. In: Sakallah, K., Simon, L. (Eds.), Proceedings of the Fourteenth International Conference on Theory and Applications of Satisfiability Testing (SAT'11). Vol. 6695 of Lecture Notes in Computer Science. Springer-Verlag, pp. 302–316.

[69] van Gelder, A., 2012. Contrasat - a contrarian SAT solver. Journal on Satisfiability, Boolean Modeling and Computation 8 (1/2), 117–122.

[70] Wei, W., Li, C., 2009. Switching between two adaptive noise mechanism in local search for SAT. Available at `http://home.mis.u-picardie.fr/~cli/EnglishPage.html`.

[71] Wotzlaw, A., van der Grinten, A., Speckenmeyer, E., Porschen, S., 2012. pfolioUZK: Solver description. In: [7], p. 45, available at `https://helda.helsinki.fi/handle/10138/34218`.

[72] Xu, L., Hoos, H., Leyton-Brown, K., 2010. Hydra: Automatically configuring algorithms for portfolio-based selection. In: Fox, M., Poole, D. (Eds.), Proceedings of the Twenty-fourth National Conference on Artificial Intelligence (AAAI'10). AAAI Press, pp. 210–216.

[73] Xu, L., Hutter, F., Hoos, H., Leyton-Brown, K., 2008. SATzilla: Portfolio-based algorithm selection for SAT. Journal of Artificial Intelligence Research 32, 565–606.

[74] Xu, L., Hutter, F., Hoos, H., Leyton-Brown, K., 2012. Evaluating component solver contributions to portfolio-based algorithm slelectors. In: [20], pp. 228–241.

[75] Xu, L., Hutter, F., Shen, J., Hoos, H., Leyton-Brown, K., 2012. SATzilla2012: Improved algorithm selection based on cost-sensitive classification models. In: [7], pp. 57–58, available at `https://helda.helsinki.fi/handle/10138/34218`.

[76] Yasumoto, T., 2012. Sinn. In: [7], pp. 61–61, available at `https://helda.helsinki.fi/handle/10138/34218`.

[77] Yun, X., Epstein, S., 2012. Learning algorithm portfolios for parallel execution. In: [32], pp. 323–338.

Appendix A. Clustering Approach

Algorithm 4: Portfolio Configuration Procedure Clustering

```
Input  : parametric solvers with configuration space $C$; desired number $k$
         of component solvers; instance set $I$; performance metric $m$;
         configurator $AC$; number $n$ of independent configurator runs;
         total configuration time $t$; feature normalizer $FN$; cluster
         algorithm $CA$; features $f(i)$ for all instances $i \in I$
Output : parallel portfolio solver with portfolio $\hat{c}_S$
1 normalize features with $FN$ into feature space $f'$
2 cluster instances with $CA$ in normalized feature space $f'$ into $k$ clusters $S$
3 foreach $s \in S$ do
4     for $j := 1..n$ do
5         obtain configuration $c_s^{(j)}$ by running $AC$ with configuration space
          $C$ on $I_s$ using $m$ for time $t/(k \cdot n)$, where $I_s$ denotes all instances in
          cluster $s$
6     let $\hat{c}_s \in \arg\min_{c_s^{(j)} | j \in \{1 \ldots n\}} m(c_s^{(j)}, I)$ be the configuration which
      achieved best performance on $I$ according to $m$
7 let $\hat{c}_S$ be the portfolio consisting the configurations for each clusters
8 return $\hat{c}_S$
```

ISAC [46, 57] is a second method for automatically designing portfolio-based algorithm selectors. It works by clustering a set of instances in a given (normalized) instance feature space and then independently configuring the given highly parameterized algorithm on each instance cluster (see Algorithm 4). We adapted *ISAC* to the ACPP problem by generalizing it in two ways. First, *ISAC* uses a linear normalization of the features, whereas we leave this decision as a parameter open to the user, allowing linear, standard (or so-called z-score), or no normalization. In general the best normalization strategy may vary between feature sets, and there is no way to assess cluster quality before configuration experiments are complete. Second, we controlled the number of clusters via a parameter, allowing us to set it to the number of cores targeted by the parallel portfolio. Hence, we do not have to use a clustering method to determine how many clusters to choose (e.g., *ISAC* uses g-means). To avoid suggesting that *ISAC*'s authors endorsed these changes, we refer to the resulting method using the neutral moniker Clustering.

Table A.9 shows results of Clustering in addition to Table 2. We note that Clustering-MP(8) clusters the training instances based on instance features; thus, normalizing these features in different ways can result in different instance clusters. There is no way to assess cluster quality before configuration experiments are complete; one can only observe the distribution of the instances in the clusters. For example, the instances in the training set of the *application* distribution for Clustering-None-MP(8) were distributed across clusters of

	Lingeling (*application*)			*clasp* (*hard combinatorial*)		
Solver Set	#TOs	PAR10	PAR1	#TOs	PAR10	PAR1
CLUSTERING-NONE-MP(8)	**47***	**1571***	**302***	107	3257	368
CLUSTERING-LINEAR-MP(8)	61	1970	323	114	3476	398
CLUSTERING-ZSCORE-MP(8)	**51***	**1674***	**297***	99	3035	362

Table A.9: Runtime statistics on the test set from *application* and *hard combinatorial* SAT instances achieved by CLUSTERING with different feature normalization strategies, CLUSTERING-NONE-MP(8): no normalization, CLUSTERING-LINEAR-MP(8): linear normalization ([0,1]), CLUSTERING-ZSCORE-MP(8): z-score normalization. The performance of a solver is shown in boldface if it was not significantly different from the best performance, and is marked with an asterisk (*) if it was not significantly worse than DEFAULT-MP(8)+CS (according to a permutation test with 100 000 permutations and significance level $\alpha = 0.05$).

sizes 2, 2, 3, 11, 13, 18, 21, and 30; we observed qualitatively similar distributions for CLUSTERING-LINEAR-MP(8) and CLUSTERING-ZSCORE-MP(8). This is potentially problematic, because running a configurator on sets of 2 or 3 instances can lead to overfitting and produce configurations whose performance does not generalize well to new instances. One reason for these small clusters could be related to our instance selection technique (see Section 3.2.3), which reduced the number of training instances to speed up the configuration process. However, the instance selection technique we used already provides a mechanism to improve the distribution of the instances in the feature space. Kadioglu et al. [46] described how *ISAC* removes such small clusters by merging them into larger clusters. However, in the case of parallel portfolios, the number of clusters is fixed, because the number of clusters has to match the desired portfolio size, in order to ensure maximal utilization of the given parallel computing resources.

For both solvers, linear feature normalization (CLUSTERING-LINEAR-MP(8)) produced clusters that were insufficiently complementary, and hence led to relatively poor performance. (We note that linear normalization is used in *ISAC*.) Using clustering without feature normalization (CLUSTERING-NONE-MP(8)) led to surprisingly strong performance in the case of *Lingeling* on the *application* instances, but failed to reach the performance of DEFAULT-MP(8)+CS for *clasp* on the *hard combinatorial* scenario. Similarly, the use of z-score normalization (CLUSTERING-ZSCORE-MP(8)) did not produce portfolios that consistently reached the performance of DEFAULT-MP(8)+CS.

Table A.10 shows results of CLUSTERING in addition to Table 6. All CLUSTERING approaches performed significantly worse than the best ACPP approach (PARHYDRA-MP(8)).

As we previously observed with portfolios based on *Lingeling*, CLUSTERING-NONE-MP(8) (no feature normalization) performed best among the CLUSTERING approaches. However, this time, CLUSTERING-ZSCORE-MP(8) performed worse than CLUSTERING-LINEAR-MP(8). This indicates that the quality of the clusters depends not only on the instance set but also on the configuration space of the portfolio (which, indeed, is disregarded by the CLUSTERING approach).

8-Processor Parallel Solver	#TOs	PAR10	PAR1
Clustering-None-MP(8)(*pfolioUZK* w/o *Plingeling*)	42	1390	256
Clustering-Linear-MP(8)(*pfolioUZK* w/o *Plingeling*)	48	1581	285
Clustering-Zscore-MP(8)(*pfolioUZK* w/o *Plingeling*)	52	1676	272

Table A.10: Runtime statistics for 8-processor parallel solvers on the *application* test set. The performance of a solver is shown in boldface if it was not significantly different from the best performance (according to a permutation test with 100 000 permutations at significance level $\alpha = 0.05$).

The Clustering approach cannot be effectively applied to sets of component solvers that include parallel solvers. When the configuration of each component solver is performed independently of all other solvers, there is no way to direct a configurator to consider synergies between solvers, such as those arising from clause sharing. Therefore, an unparameterized, parallel solver with clause sharing, such as *Plingeling*, will never be selected. Thus, we did not consider a variant of Clustering in the experiments of Section 5.2.