Parallel Algorithm Configuration

Frank Hutter, Holger H. Hoos and Kevin Leyton-Brown

University of British Columbia, 2366 Main Mall, Vancouver BC, V6T 1Z4, Canada
{hutter,hoos,kevinlb}@cs.ubc.ca

Abstract. State-of-the-art algorithms for solving hard computational problems often expose many parameters whose settings critically affect empirical performance. Manually exploring the resulting combinatorial space of parameter settings is often tedious and unsatisfactory. Automated approaches for finding good parameter settings are becoming increasingly prominent and have recently lead to substantial improvements in the state of the art for solving a variety of computationally challenging problems. However, running such automated algorithm configuration procedures is typically very costly, involving many thousands of invocations of the algorithm to be configured. Here, we study the extent to which parallel computing can come to the rescue. We compare straightforward parallelization by multiple independent runs with a more sophisticated method of parallelizing the model-based configuration procedure SMAC. Empirical results for configuring the MIP solver CPLEX demonstrate that near-optimal speedups can be obtained with up to 16 parallel workers, and that 64 workers can still accomplish challenging configuration tasks that previously took 2 days in 1–2 hours. Overall, we show that our methods make effective use of large-scale parallel resources and thus substantially expand the practical applicability of algorithm configuration methods.

1 Introduction

Heuristic algorithms are often surprisingly effective at solving hard combinatorial problems. However, heuristics that work well for one family of problem instances can perform poorly on another. Recognizing this, algorithm designers tend to parameterize some of these design choices so that an end user can select heuristics that work well in the solver's eventual domain of application. The way these parameters are chosen often makes the difference between whether a problem ends up being "easy" or "hard", with differences in runtime frequently spanning multiple orders of magnitude.

Traditionally, the problem of finding good parameter settings was left entirely to the end user (and/or the algorithm designer) and was solved through manual experiments. Lately, automated tools have become available to solve this *algorithm configuration (AC) problem*, making it possible to explore large and complex parameter spaces (involving up to more than seventy parameters with numerical, categorical, ordinal and conditional values; *e.g.*, [1, 2, 3]). Procedures for automated algorithm configuration have advanced steadily over the past few years, and now can usually outperform default settings determined by human experts even in very large and challenging configuration scenarios. However, automated algorithm configuration is computationally expensive: AC procedures involve repeatedly running the algorithm to be configured (the so-called *target algorithm*) with different parameter settings, and hence consume orders of magnitude more computing time than a single run of the target algorithm.

Substantial amounts of computing time are now readily available, in the form of powerful, multi-core consumer machines, large compute clusters and on-demand commodity computing resources, such as Amazon's Elastic Compute Cloud (EC2). Nevertheless, the computational cost of AC procedures remains a challenge—particularly because the wall-clock time required for automated algorithm configuration can quickly become a limiting factor in real-world applications and academic studies. As ongoing increases in the amount of computing power per cost unit are now almost exclusively achieved by means of parallelization, the effective use of parallel computing resources becomes an important issue in the use and design of AC procedures.

Most AC procedures are inherently sequential in that they iteratively perform target algorithm runs, learn something about which parameters work well, and then perform new runs taking this information into account. Nevertheless, there are significant opportunities for parallelization, in two different senses. First, because all state-of-the-art AC procedures are randomized, the entire procedure can be run multiple times in parallel, followed by selection of the best configuration thus obtained. Second, a finer-grained form of parallelism can be used, with a centralized process distributing sets of target algorithm runs over different processor cores. Indeed, the literature on algorithm configuration already contains examples of both the first [1, 4, 2, 3] and second [4, 5] forms of parallelization.

Here, we present a thorough investigation of parallelizing automated algorithm configuration. We explore the efficacy of parallelization by means of multiple independent runs for two state-of-the-art algorithm configuration procedures, PARAMILS [4] and SMAC [6], investigating both how to make the best use of wall-clock time (what if parallel resources were free?) and CPU time (what if one had to pay for each CPU hour?). We present a method for introducing fine-grained parallelism into model-based AC procedures and apply it to SMAC. We evaluate the performance of the resulting distributed AC procedure, D-SMAC, as in contrast to and in combination with parallelization by means of multiple independent runs. Overall, we found that D-SMAC outperformed independent parallel runs and achieved near-perfect speedups when using up to 16 parallel worker processes. Using 64 parallel workers, we still obtained 21-fold to 52-fold speedups and were thus able to solve AC problems that previously took 2 days in 1-2 hours.

2 Methods for Parallelizing Algorithm Configuration

In this section, we describe two methods for parallelizing algorithm configuration: performing multiple independent runs and distributing target algorithms runs in the model-based configuration procedure SMAC.

2.1 Multiple Independent Runs

Any randomized algorithm with sufficiently large variation in runtime can usefully be parallelized by a simple and surprisingly powerful method: performing multiple independent runs (see, *e.g.*, [7, 8, 9]). In particular, it has been shown that the runtime of certain classes of local search procedures closely follows an exponential distribution [10], implying that optimal speedups can be achieved by using additional processor cores. Likewise,

some complete search procedures have been shown to exhibit heavy-tailed runtime distributions, in which case multiple independent runs (or, equivalently, random restarts) can yield parallelization speedups greater than the number of parallel processes [8].

Multiple independent runs have also been used routinely in algorithm configuration (although we are not aware of any existing study that characterizes the runtime distributions of these procedures). In our research on PARAMILS, we have adopted the policy of performing 10 to 25 parallel independent runs, returning the configuration found by the run with best training performance [1, 4, 2, 3], which can be formalized as follows:

Definition 1 (k-fold independent parallel version of configurator $\mathcal{C}$) *The k-fold independent parallel version of configurator $\mathcal{C}$, denoted $k \times \mathcal{C}$, is the configurator that executes k runs of $\mathcal{C}$ in parallel, and whose incumbent at each time t is the incumbent of the run with the best training performance at time t.*

For a given time budget, PARAMILS may not be able to evaluate target algorithm performance on all given training instances. In previous work, in such cases, we would sometimes measure training performance on the entire training set at the end of each of the independent configuration runs, selecting the final configuration based on those data. To keep computational costs manageable, we did not do this in the work described here. Also in previous work, we have observed that PARAMILS runs occasionally stagnate at rather poor configurations, and that in such cases $k\times$PARAMILS can dramatically improve performance. However, to the best of our knowledge, this effect has never been quantified for PARAMILS, nor for any other AC procedure.

2.2 D-SMAC: SMAC with Distributed Target Algorithm Runs

Any state-of-the-art AC procedure could in principle be parallelized by distributing target algorithm runs over multiple cores. However, we are only aware of two examples from the literature describing AC solvers that implement such fine-grained parallelism. The first is the genetic algorithm GGA [5]; however, GGA always uses eight local workers, regardless of machine architecture, and is unable to distribute runs on a cluster. Second, in our own PARAMILS variant BASICILS [4], target algorithm runs were distributed over a cluster with 110 CPUs [11]. This, however, took advantage of the fact that BASICILS performs a large number of runs for *every* configuration considered, and the same fact explains why our standard PARAMILS variant FOCUSEDILS typically outperforms BASICILS. [1] To the best of our knowledge, the effect of the number of parallel processes on overall performance has not been studied for any of these configurators.

Here, we present a general and principled method for adding fine-grained parallelization to SMAC, a recent model-based AC procedure [6]. SMAC is the focus of our current work, because (1) it achieves state-of-the-art performance for AC [6] and (2) its explicit model of algorithm performance promises to be useful beyond merely finding good configurations (*e.g.*, for selecting informative problem instances or for gaining deeper insights into the impact of parameter settings on target algorithm performance).

SMAC operates in 4 phases (see Algorithm 1). First, it initializes its data and incumbent configuration $\boldsymbol{\theta}_{inc}$—the best configuration seen thus far—using algorithm runs

[1] The latest implementation of iterated F-Race [12] also supports parallelization of target algorithm runs, but this feature has not (yet) been described in the literature.

Algorithm 1: Sequential Model-Based Algorithm Configuration (SMAC)

$\mathbf{R}$ keeps track of all performed target algorithm runs and their performances (*i.e.*, SMAC's training data); $\mathcal{M}$ is SMAC's model; and $\vec{\boldsymbol{\Theta}}_{new}$ is a list of promising configurations.

Input : Target algorithm with parameter configuration space $\boldsymbol{\Theta}$; instance set Π; cost metric $\hat{c}$

Output : Optimized (incumbent) parameter configuration, $\boldsymbol{\theta}_{inc}$

$[\mathbf{R}, \boldsymbol{\theta}_{inc}] \leftarrow$ *Initialize*($\boldsymbol{\Theta}$, Π)
repeat
  $\mathcal{M} \leftarrow$ *FitModel*($\mathbf{R}$)
  $\vec{\boldsymbol{\Theta}}_{new} \leftarrow$ *SelectConfigurations*($\mathcal{M}$, $\boldsymbol{\theta}_{inc}$, $\boldsymbol{\Theta}$)
  $[\mathbf{R}, \boldsymbol{\theta}_{inc}] \leftarrow$ *Intensify*($\vec{\boldsymbol{\Theta}}_{new}$, $\boldsymbol{\theta}_{inc}$, $\mathbf{R}$, Π, $\hat{c}$)
until *total time budget for configuration exhausted*
return $\boldsymbol{\theta}_{inc}$

from an initial design. Then it iterates between learning a new model, selecting new configurations based on that model and performing additional runs to compare these selected configurations against the incumbent.

The selection of new configurations is performed by optimizing a desirability function $d(\boldsymbol{\theta})$ defined in terms of the model's predictive distribution for $\boldsymbol{\theta}$. This desirability function serves to address the exploration/exploitation tradeoff between learning about new, unknown parts of the parameter space and intensifying the search locally in the best known region. Having found an incumbent with training performance $f_{\min}$, SMAC uses a classic desirability function measuring the expected positive improvement over f_{min}, $\mathbb{E}[I(\boldsymbol{\theta})] = \mathbb{E}[\max\{0, f_{\min} - f(\boldsymbol{\theta})\}]$. Many other desirability functions have been defined, such as the probability of improvement $\mathbb{P}[I(\boldsymbol{\theta}) > 0]$ [13], generalizations of expected improvement $\mathbb{E}[I^g(\theta)]$ for $g > 1$ [14], and the optimistic confidence bound $(-\mu_{\boldsymbol{\theta}} + \lambda\sigma_{\boldsymbol{\theta}})$ for $\lambda > 0$ [13, 15].[2] High values of all of these desirability functions reward low predictive mean (to encourage minimization of the performance metric) and high predictive variance (to encourage exploration of new regions).

Several methods have been proposed for identifying multiple desirable inputs to be evaluated in parallel. Ginsbourger et al. [16] introduced the multipoints expected improvement criterion, as well as the "constant liar approach": greedily select one new input $\boldsymbol{\theta}$, using expected improvement, hallucinate that its response equals the current model's predictive mean $\mu_{\boldsymbol{\theta}}$, refit the model, and iterate. Jones [13] demonstrated that maximizing the optimistic confidence bound $(-\mu_{\boldsymbol{\theta}} + \lambda\sigma_{\boldsymbol{\theta}})$ with different values of λ yields a diverse set of points whose parallel evaluation is useful.

In our distributed version of SMAC we follow this latter approach (slighly deviating from it by sampling λ uniformly at random from an exponential distribution with mean 1 instead of using a fixed set of values for λ), since it also allows for the selection step to be parallelized: each of k workers can sample a value for λ and then optimize $(-\mu_{\boldsymbol{\theta}} + \lambda\sigma_{\boldsymbol{\theta}})$ independently. Surprisingly, although we originally chose it in order to facilitate parallelization, in our experiments we found that this modified desirability function sometimes substantially improved SMAC's performance and never substantially

[2] For maximization problems, this desirability function is $(\mu_{\boldsymbol{\theta}} + \lambda\sigma_{\boldsymbol{\theta}})$ and is called the *upper confidence bound (UCB)*.

Algorithm 2: Distributed Sequential Model-Based Algorithm Configuration (D-SMAC)

$\mathbf{Q}$ is a queue of target algoritm runs to be executed; $\mathbf{A}$ is a set of runs currently assigned to workers; $\mathbf{R}$ keeps track of all executed runs and their performances (*i.e.*, SMAC's training data); $\mathcal{M}$ is SMAC's model, and $\vec{\boldsymbol{\Theta}}_{new}$ is a list of promising configurations. *Initialize* performs $\sqrt{k}$ runs for the default configuration and one run each for other configurations from a Latin Hypercube Design.

Input : Target algorithm with parameter configuration space $\boldsymbol{\Theta}$; instance set Π; cost metric $\hat{c}$; number of workers, k

Output : Optimized (incumbent) parameter configuration, $\boldsymbol{\theta}_{inc}$

$\mathbf{Q} \leftarrow$ *Initialize*($\boldsymbol{\Theta}$, Π, $2k$)
$\mathbf{A} \leftarrow \emptyset$; Move first k runs from $\mathbf{Q}$ to $\mathbf{A}$ and start the workers
repeat
  Wait for workers to finish, move the finished runs from $\mathbf{A}$ to $\mathbf{R}$
  $\vec{\boldsymbol{\Theta}}_{new} \leftarrow \{\boldsymbol{\theta} \mid \mathbf{R}$ received at least one new run with $\boldsymbol{\theta}\}$
  $[\mathbf{Q}, \boldsymbol{\theta}_{inc}] \leftarrow$ *Intensify*($\vec{\boldsymbol{\Theta}}_{new}$, $\boldsymbol{\theta}_{inc}$, $\mathbf{Q}$, $\mathbf{R}$, Π, $\hat{c}$)
  Move first k runs from $\mathbf{Q}$ to $\mathbf{A}$ and start the workers
  if $|Q| < k$ **then**
    $\mathcal{M} \leftarrow$ *FitModel*($\mathbf{R}$)
    $\vec{\boldsymbol{\Theta}}_{new} \leftarrow$ *SelectConfigurations*($\mathcal{M}$, $\boldsymbol{\theta}_{inc}$, $\boldsymbol{\Theta}$, $k - |Q|$)
    $[\mathbf{Q}, \boldsymbol{\theta}_{inc}] \leftarrow$ *Intensify*($\vec{\boldsymbol{\Theta}}_{new}$, $\boldsymbol{\theta}_{inc}$, $\mathbf{Q}$, $\mathbf{R}$, Π, $\hat{c}$)
until *total time budget for configuration exhausted*
return $\boldsymbol{\theta}_{inc}$

degraded it compared to the expected improvement criterion we used previously (see Table 2 in Section 4).

The simplest way to parallelize SMAC would be to maintain the structure of Algorithm 1, but to execute each major component in parallel, synchronizing afterwards. The initialization can be parallelized easily, as it consists of randomly chosen target algorithm runs. Model fitting also parallelizes, as SMAC uses random forest models: each tree can be learned independently, and even subtrees of a single tree are independent. Gathering k desirable and diverse configurations can be parallelized as described above, and one can also parallelize the comparison of these configurations against the incumbent. We experimented with this approach, but found that when running on a compute cluster, it suffered from high communication overhead: learning the model requires the full input data, and optimizing the desirability function requires the model. Furthermore, the model learning phase is unlikely to parallelize perfectly, since a constant fraction of the time for building a regression tree is spent at the root of the tree. While we can still parallelize perfectly *across* trees, we typically only use 10 trees in practice and are interested in scaling to much larger numbers of parallel processes.

The parallelized version of SMAC we present here (dubbed D-SMAC) is therefore based on a different approach, slightly changing the structure of SMAC to bypass the chokepoint wherein more workers are available than can meaningfully be used to learn the model. Algorithm 2 illustrates the new control flow. The important difference is that D-SMAC maintains a queue of algorithm runs that is replenished whenever its

current state drops below the number of runs than can be handled in one iteration by the parallel processes available. The intensification step—which compares challengers to the incumbent—now merely queues up runs rather than executing them. The benefit is that a master process can learn the model and select desirable new configurations *while* worker processes are performing target algorithm runs (typically the most expensive operation in SMAC). The master could also execute target runs or parallelize model learning and selecting configurations as necessary, to further balance load with the workers. In our current implementation, we simply use a separate processor for SMAC's master thread; since the model overhead was low in our experiments, these master threads spent most of their time idling, and we started several master processes on a single CPU.[3]

We employed a lightweight solution for distributing runs on compute clusters. The D-SMAC master writes command line call strings for target algorithm runs to designated files on a shared file system. Worker jobs submitted via the respective cluster's queueing software (in our case, Torque) listen on the designated files, carry out the requested target run, and write the resulting performance to designated output files to be read by the master. On the cluster we used, we found the overhead for this job dispatch mechanism to be comparable to starting jobs on the local machine. Larger-scale deployment of D-SMAC would benefit from the use of an experimental framework such as HAL [17] or EDACC [18].

3 Experimental Setup

Our configuration experiments in this paper focus on the optimization of the solution quality that the mixed integer solver CPLEX can achieve in a fixed runtime. Specifically, we employed the five solution quality AC scenarios introduced in [2], as well as one additional scenario described below. All of these AC scenarios use a lexicographic objective function that first minimizes the number of instances for which no feasible solution was found, and then breaks ties by the average optimality gap. To use this objective function in SMAC and D-SMAC (whose modelling step requires scalar objective functions), we counted the "optimality gap" of runs that did not find a feasible solution as $10^{10}\%$. For a configuration scenario with test instance set $\mathcal{S}$ and fixed time limit per CPLEX run L, we defined the *test performance* of a configuration run R as the average optimality gap CPLEX achieved on $\mathcal{S}$ in runs with time limit L when using the incumbent parameter configuration of R.

Throughout our experiments, in order to study the test performance of $k \times \mathcal{C}$, the k-fold independent parallel version of AC procedure $\mathcal{C}$, we employed a bootstrap analysis. Given a large population $\mathcal{P}$ of independent runs of $\mathcal{C}$, we evaluated $k \times \mathcal{C}$ by repeatedly drawing k runs of $\mathcal{C}$ from $\mathcal{P}$ (with repetitions) and computing the test performance of the best of the k runs (best in terms of training performance). This process yielded a bootstrap distribution of test performance; we plot the median of this distribution at each time step, show boxplots for the final state and carry out a Mann-Whitney U-test for differences across different configurators (or different parallelization options).

[3] The model overhead grows with the number of data points, meaning that for long enough configuration runs it could become a chokepoint. In such settings, the master could delegate model learning to a slave, and update its model whenever the slave completed this work.

Parameter type	# parameters of this type	# values considered	Total # configurations
Boolean	6 (7)	2	$1.90 \cdot 10^{47}$
Categorical	45 (43)	3–7	
Integer	18	5–7	
Continuous	7	5–8	

Benchmark	Description	# instances		Default performance	
		training	test	% infeasible	mean opt. gap when feasible
MIK	Mixed integer knapsack [19]	60	60	0%	0.142%
CLS	Capacitated lot-sizing [20]	50	50	0%	0.273%
REGIONS200	Combinatorial winner determination [21]	1000	1000	0%	1.87%
CORLAT	Wildlife corridor [22]	1000	1000	28%	4.43%
MASS	Multi-activity shift scheduling [23]	50	50	64%	1.91%
RCW	Spread of red-cockaded woodpecker [24]	1000	1000	0%	49%

Table 1. Overview of CPLEX parameters and MIP benchmark sets used.

To carry out a robust bootstrap analysis of $k \times \mathcal{C}$, a population of roughly $3 \cdot k$ runs of $\mathcal{C}$ is required. Since we wanted to evaluate the benefit of up to 64 independent runs, we had to run each configurator 200 times on each configuration scenario. As a result, we carried out over 5000 configuration runs, more than in all of our previously published works on AC combined. Note that each individual configuration run involved thousands of target algorithm runs. In total, the experiments for this paper (including offline validation) took roughly 20 CPU years.

To fit within this time budget, we kept the original, relatively small configuration budget for the five AC scenarios taken from [2]: five hours per AC run and ten seconds per CPLEX run. Since the machines we used[4] are a (surprisingly constant) factor of just above 2 times faster than the machines used in [2], we divided both the runtime for configuration runs and for individual CPLEX runs by 2 to keep the characteristics of the AC scenarios as similar as possible to previously published work.

For the same reason, we used exactly the same parameter configuration space of CPLEX 12.1, and the same mixed integer problems (MIPs) as in the original scenarios from [2]. Briefly, we considered 76 parameters that directly affect the performance of CPLEX. We carefully kept all parameters fixed that change the problem formulation (*e.g.*, numerical precision parameters). The 76 parameters we selected affect all aspects of CPLEX. They include 12 preprocessing parameters; 17 MIP strategy parameters; 11 cut parameters; 9 MIP limits parameters; 10 simplex parameters; 6 barrier optimization parameters; and 11 further parameters. Table 1 gives an overview of these parameters and of the MIP benchmarks we used; full details can be found in [2].

To study whether our findings for the short configuration runs above translate to longer runs of the most recent CPLEX version (12.3) on more challenging benchmark sets, we also carried out experiments on a new configuration scenario. The MIP instances in this scenario come from the domain of computational sustainability; they model the spread of the endangered red-cockaded woodpecker (RCW), conditional on decisions about certain parcels of land to be protected. We generated 2000 instances using the generator from [24] (using the five hardest of their eleven maps). CPLEX 12.3's default configuration could solve 7% of these instances in two minutes and 75% in one hour.

[4] All of our experiments were carried out on the Westgrid Orcinus cluster (http://www.westgrid.ca/), comprising 384 nodes with two Intel X5650 six-core 2.66 GHz processors each.

Scenario	Unit	Median test performance			Median test performance		
		PILS	SMAC	d-SMAC(1)	25×PILS	25×SMAC	25×d-SMAC(1)
CLS	[0.1%]	2.36	2.43	**2.00**	1.38	1.41	**1.35**
CORLAT	[10^8%]	17.6	3.17	**2.95**	4.20	0.82	**0.72**
MIK	[0.01%]	6.56	6.59	**2.78**	**0.44**	2.08	0.73
Regions200	[1%]	**1.69**	1.8	1.83	**0.85**	1.16	1.14
MASS	[10^9%]	6.40	3.68	**3.47**	4.00	2.36	**2.29**

Table 2. Statistics for baseline comparison of configuration procedures. We show median test performances achieved by the base AC procedures (left), and their k-fold parallel independent run versions with $k = 25$ (recall that test performance is the average optimality gap across test instances, counting runs with infeasible solutions as a gap of 10^{10}%). We bold-faced entries for configurators that are not significantly worse than the best configurator for the respective configuration space, based on a Mann-Whitney U test.

The objective in our RCW configuration scenario was to minimize the optimality gap CPLEX could achieve within two minutes, and the AC budget was two days.

Throughout our experiments, we accounted for the inherent runtime overheads for building and using models, but we did not count the constant overhead of starting jobs (either as part of the per-run budget or of the configuration budget), since this can be reduced to almost zero in a production system. We computed the wall clock time for each iteration of D-SMAC as the maximum of the master's model learning time and the maximum of the CPU times of the parallel algorithm runs it executed in parallel.

4 Experiments

We studied the parallelization speedups obtained by using multiple independent runs and by using fine-grained parallelism in D-SMAC. As a side result, we were able to show for the first time that SMAC (in its sequential version) achieves state-of-the-art performance for optimizing a measure of solution quality that can be obtained in a fixed time (rather than minimizing the runtime required to solve a problem).

4.1 Multiple Independent Runs

First, we assessed the baseline performance of the three sequential AC procedures we used: PARAMILS, SMAC, and D-SMAC(1). PARAMILS has been shown to achieve state-of-the-art performance for the five configuration scenarios we study here [2], and Table 2 demonstrates that SMAC and D-SMAC(1) perform competitively, making all procedures natural candidates for parallelization. The right part of Table 2 compares the performance of the multiple independent run versions 25×PARAMILS, 25×SMAC, and 25×D-SMAC, showing that PARAMILS benefitted more from multiple runs than the two SMAC versions. The raw data (not shown) explains this: the variance of PARAMILS's performance was higher than for either SMAC version.

Table 3 quantifies the speedups gained by multiple independent runs of the AC procedures. For the two versions of SMAC, speedups were consistent and sometimes near-perfect with up to 4 independent runs. Due to larger performance variation between

Scenario	PARAMILS			SMAC			D-SMAC(1)		
	$1\rightarrow 4\times$	$4\rightarrow 16\times$	$16\rightarrow 64\times$	$1\rightarrow 4\times$	$4\rightarrow 16\times$	$16\rightarrow 64\times$	$1\rightarrow 4\times$	$4\rightarrow 16\times$	$16\rightarrow 64\times$
CLS	5.02	2.87	1.66	5.72	2.33	1.50	1.92	2.09	1.75
CORLAT	12.1	4.75	4.22	2.45	2.10	1.15	3.93	2.31	1.00
MIK	8.29	3.10	2.29	3.22	3.45	4.01	2.37	2.91	1.02
Regions200	5.59	3.65	2.94	3.04	1.49	1.76	3.14	3.08	2.39
MASS	4.00	5.78	1.00	1.62	1.44	1.36	2.24	1.49	1.00

Table 3. Speedups achieved by using independent parallel runs of various AC procedures $\mathcal{C}$. We give the speedups of $4 \times \mathcal{C}$ over $\mathcal{C}$, $16 \times \mathcal{C}$ over $4 \times \mathcal{C}$, and $64 \times \mathcal{C}$ over $16 \times \mathcal{C}$. The speedup of procedure $\mathcal{C}_1$ over procedure $\mathcal{C}_2$ is defined as the time allocated to $\mathcal{C}_2$ divided by the time $\mathcal{C}_1$ required to reach (at least) $\mathcal{C}_2$'s final solution quality. We do not report speedups of $16 \times \mathcal{C}$ and $64 \times \mathcal{C}$ over $\mathcal{C}$ directly since $\mathcal{C}$ often found very poor results in the small configuration budget allowed, the time to find which is not indicative of a procedure's ultimate performance.

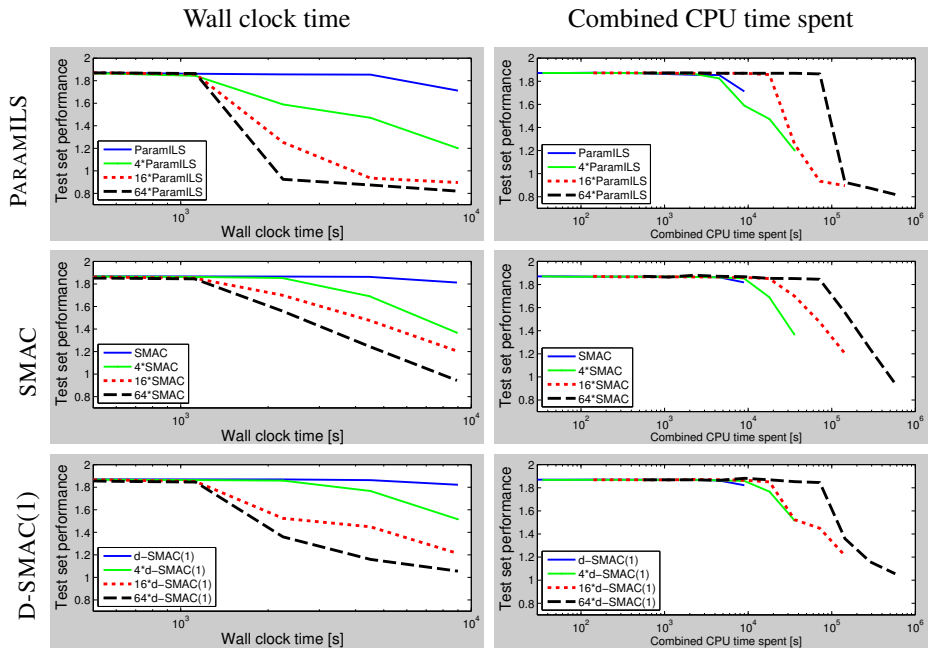


Fig. 1. Evaluation of k-fold parallel independent run versions of PARAMILS, SMAC, and D-SMAC(1) on benchmark set Regions200. For each configurator C, k*C denotes $k\times$C.

independent runs, the parallelization speedups obtained for PARAMILS were more pronounced: perfect or higher-than-perfect speedups were observed for all scenarios with up to 4 independent runs, and the speedup factor obtained when moving from 4 to 16 independent parallel runs was still almost 4.

Figures 1 and 2 visualize the speedups achieved for two representative configuration scenarios. As the left column of Figure 1 shows, for benchmark set Regions200 additional independent runs yielded consistent speedups in wall clock time for all configurators. The right column shows that, as the runtime spent in a PARAMILS or D-SMAC(1) run

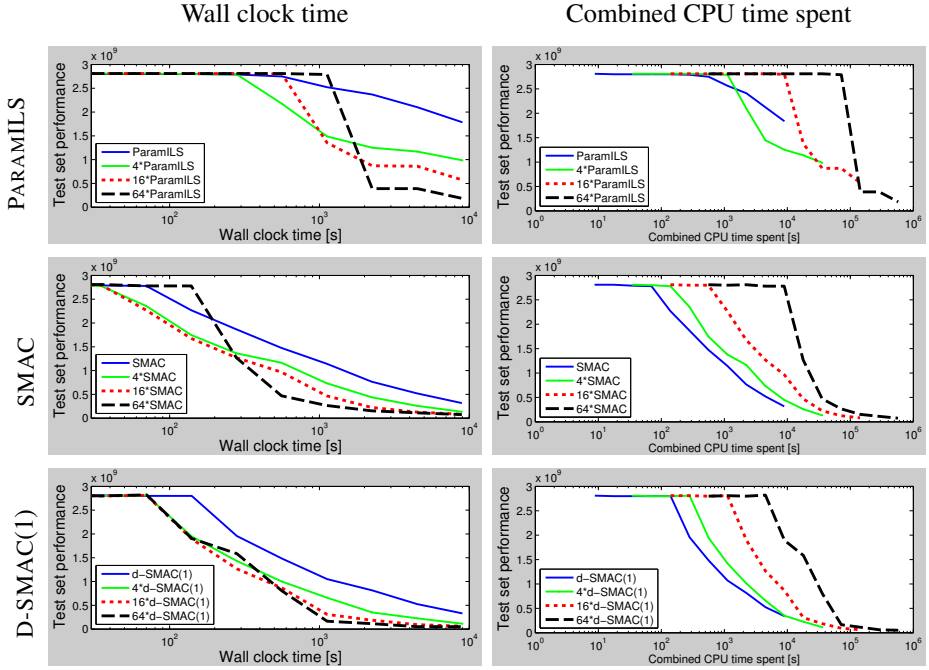


Fig. 2. Evaluation of k-fold parallel independent run versions of PARAMILS, SMAC, and D-SMAC(1) on benchmark set CORLAT.

increases for this benchmark set, $k\times$PARAMILS and $k\times$D-SMAC(1) tend to use their combined CPU time about as well as their respective sequential versions with a k-fold larger time budget. Figure 2 visualizes the results for benchmark set CORLAT, showing an interesting effect by which $k\times$PARAMILS and $k\times$SMAC can actually perform *worse* in their early phases as k increases. This effect is due to the fact that training performance can be *negatively* correlated with progress in the early phase of the search;[5] it is clearly visible as the crossing of lines in Figure 2 (left column). Another interesting effect for this scenario is that $4\times$PARAMILS achieved higher-than-perfect speedups (visible as the crossing of lines for PARAMILS in the right column of Figure 2).

4.2 Distributed SMAC

We now evaluate the parallelization speedups obtained by D-SMAC with a varying number of parallel worker processes. As shown in Table 4, these speedups were greater

[5] PARAMILS starts from the default configuration, which finds a feasible solution for 72% of the instances. The configuration scenario's objective function heavily penalizes target algorithm runs that do not find a feasible solution and no configuration is found that finds a feasible solution for *all* training instances. Thus, any configuration run that has made enough progress will have a worse training performance than configuration runs that are still stuck having done only a few successful runs on the default. The larger we grow k in $k\times$PARAMILS, the more likely it is that one of the runs will be stuck at the default up to any given time (having seen only successful runs for the default), making $k\times$PARAMILS's incumbent the default configuration.

Scenario	$1 \rightarrow 4\times$	$4 \rightarrow 16\times$	$16 \rightarrow 64\times$	$1 \rightarrow 4\times$	$1 \rightarrow 16\times$	$1 \rightarrow 64\times$
CLS	6.22	2.87	2.55	6.22	16.2	41.2
CORLAT	4.04	3.35	1.95	4.04	13.7	27.3
MIK	2.40	4.70	1.56	2.40	11.9	21.3
Regions200	2.61	10.9	1.25	2.61	41.3	52.3
MASS	2.21	3.44	2.41	2.21	9.76	21.5

Table 4. Wall clock speedups over D-SMAC(1) with different numbers of distributed workers in SMAC. The speedup of procedure $\mathcal{C}_1$ over procedure $\mathcal{C}_2$ is defined as the time allocated to $\mathcal{C}_2$ divided by the time $\mathcal{C}_1$ required to reach (at least) $\mathcal{C}_2$'s final solution quality. For consistency with Table 3, we give the speedups of $4 \times \mathcal{C}$ over $\mathcal{C}$, $16 \times \mathcal{C}$ over $4 \times \mathcal{C}$, and $64 \times \mathcal{C}$ over $16 \times \mathcal{C}$. We also report the speedups of $16 \times \mathcal{C}$ over $\mathcal{C}$, and of $64 \times \mathcal{C}$ over $\mathcal{C}$.

Scenario	Unit	Bootstrap median of average test set performance			
		$64 \times$ d-SMAC(1)	$16 \times$ d-SMAC(4)	$4 \times$ d-SMAC(16)	d-SMAC(64)
CLS	$[0.1\%]$	2.37	1.96	**1.76**	**1.81**
CORLAT	$[10^8\%]$	10.9	3.41	**1.96**	**2.26**
MIK	$[0.01\%]$	8.68	**1.2**	**2.03**	**2.46**
Regions200	$[1\%]$	1.91	1.77	1.58	**1.52**
MASS	$[10^9\%]$	3.88	4.00	3.39	**3.2**

Table 5. Performance comparison of various possibilities of allocating 64 cores for a wall clock time of 560 seconds in D-SMAC. For each combination of independent runs and number of workers in D-SMAC, we show median test performance; we bold-faced entries for configurators that were not significantly worse than the best configurator for the respective configuration space, based on a Mann-Whitney U test.

than those for multiple independent runs, with near-perfect speedups up to 16 workers and speedup factors between 1.2 and 2.6 for increasing the number of workers by another factor of 4 to 64. Overall, D-SMAC(64)'s speedups in the time required to find configurations of the same quality as D-SMAC(1) were between 21 and 52. Figure 3 visualizes the results for three configuration scenarios. The left side of this figure demonstrates that the substantial speedups D-SMAC achieved with additional workers were consistent across scenarios and across D-SMAC's trajectory.[6] In particular, speedups for early phases of the search were much more robust than for parallelization by multiple independent runs. The right side of Figure 3 demonstrates that D-SMAC(p) used its combined CPU time almost as well as D-SMAC(1) would, but required a factor p less wall clock time.

4.3 Multiple Independent Runs of Distributed SMAC

Next, we studied various allocations of a fixed number of $N = 64$ CPUs to independent runs of D-SMAC (that is, different variants of $k\times$ D-SMAC(p) with constant $k \times p = 64$). Figure 4 shows that $1\times$ D-SMAC(64) tended to perform better than the other

[6] The only exception is that D-SMAC(4) performed better than D-SMAC(16) early in the search for scenario CORLAT. Here, several of the D-SMAC(4) runs started out an order magnitude faster than D-SMAC(1); however, after about 60 seconds of search time D-SMAC(16) dominated D-SMAC(4) as expected.

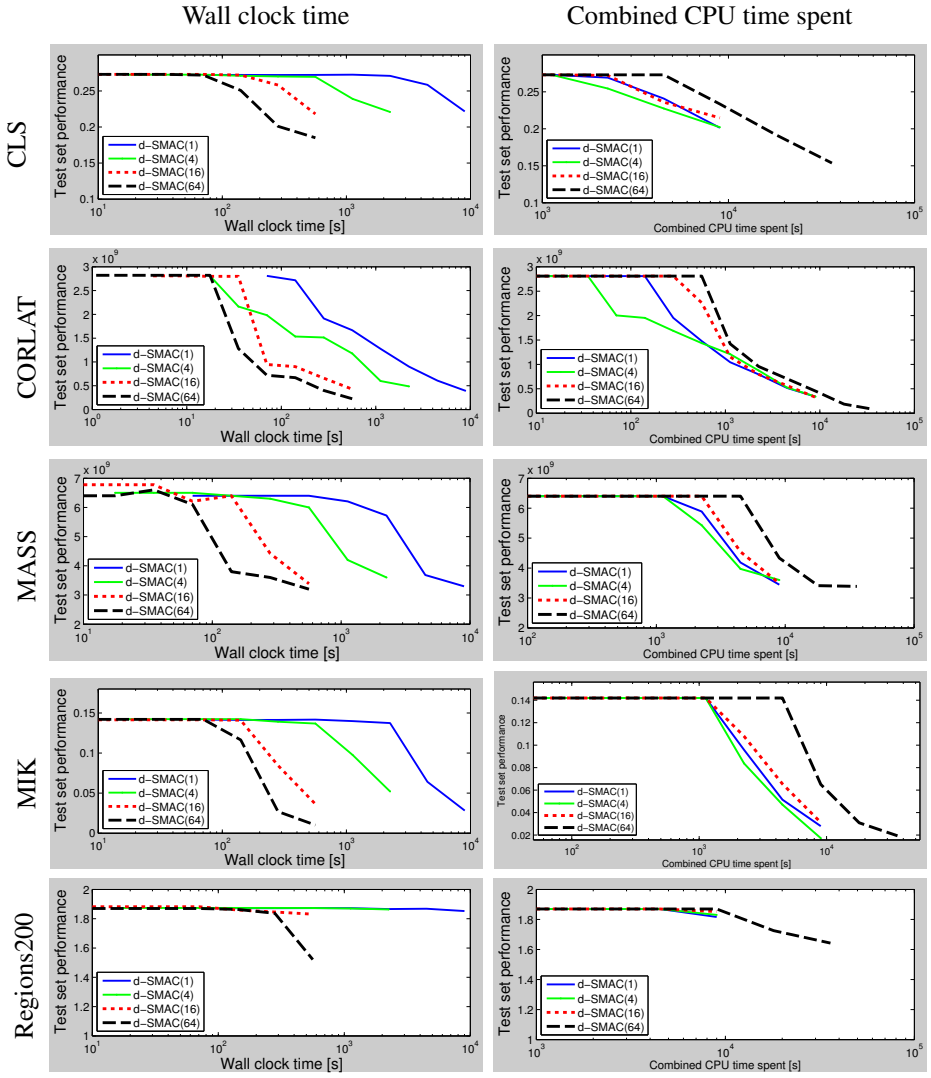


Fig. 3. Parallelization benefits for D-SMAC with different numbers of workers. (Plots in the left and right columns are based on different bootstrap samples.)

combinations across all domains and time budgets. Table 5 shows that, given the same time budget of 10 CPU hours (or 562.5 wall seconds on 64 processors), $1\times$ D-SMAC(64) statistically significantly outperformed all other combinations we tested in 2 of 5 cases, and tied for best on the remaining 3 cases. These results demonstrate that performing a small number of parallel independent runs of D-SMAC(p) can be useful, but that using all available processors in D-SMAC(p) tends to yield the best performance. While these results do not preclude the possibility that for even higher degrees of parallelization multiple independent runs of D-SMAC might be more beneficial, they do provide evidence that D-SMAC's fine-grained parallelization strategy is effective.

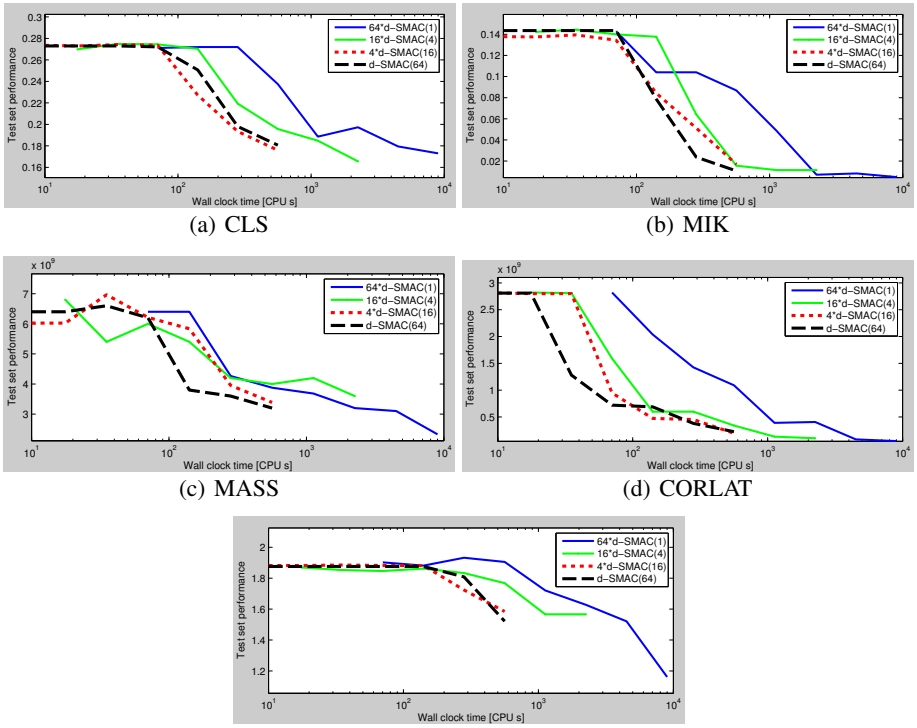


Fig. 4. Comparison of different ways of using 64 cores in d-SMAC.

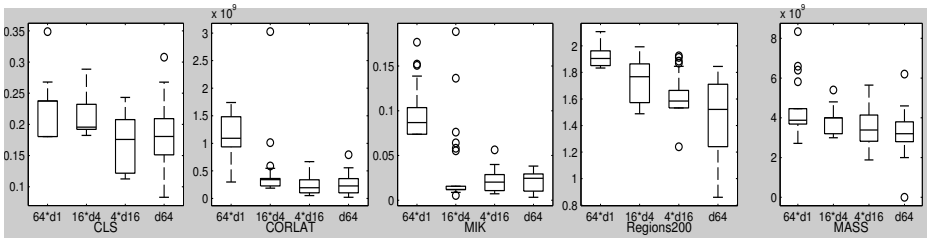
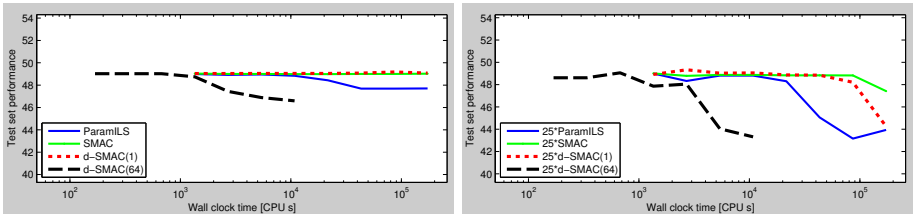


Fig. 5. Performance comparison of various possibilities of allocating 64 cores in d-SMAC; k*dp denotes k independent runs of D-SMAC(p), with $k * p = 64$. For each combination of k and p, we show boxplots of test set performance, using the same data as underlying Figure 4.

4.4 Evaluation On a Hard Instance Distribution

Finally, we investigated whether similar speedups could be obtained for configuration on more challenging benchmark sets, comparing the performance of ParamILS, SMAC, D-SMAC(1), and D-SMAC(64) for configuration scenario RCW. We performed 200 runs of ParamILS and SMAC with a configuration budget of 2 days each, as well as 25 runs of D-SMAC(64) with a budget of three wall clock hours (for a combined budget of 8 CPU days). Figure 6 shows median test performance for each of these procedures.



(a) Base performance (b) Best of 25 independent runs

Fig. 6. Comparison of PARAMILS, SMAC, and D-SMAC(64) for configuration on the challenging instance set RCW. We plot median performance across 25 configuration runs on the left, and performance of the run with best training performance on the right.

While the SMAC variants did not yield noticeable improvements over the CPLEX default configuration in its time budget, PARAMILS found somewhat better configurations. D-SMAC(64) already improved over the default configuration after roughly 20 wall clock minutes, required less than one wall clock hour to find a configuration as good as the one PARAMILS found after 2 days, and consistently improved afterwards. We also studied the performance of $25\times$PARAMILS, $25\times$SMAC, $25\times$D-SMAC(1), and $25\times$D-SMAC(64) for this benchmark set. Multiple independent runs improved the performance of all configurators. $25\times$D-SMAC(64) performed best, requiring roughly two hours to achieve the performance $25\times$PARAMILS and $25\times$D-SMAC(1) achieved in two days. It also matched D-SMAC(64)'s final performance in roughly a quarter of the time and found substantially better configurations afterwards. While a single run of D-SMAC (1600) might have yielded even better performance (we did not try, for lack of computing resources), this result shows that even AC procedures that implement large-scale fine-grained parallelism can benefit from performing multiple independent runs.

5 Conclusion

Parallel computing is key to reducing the substantial amount of time required by automatic algorithm configuration methods. Here, we presented the first comparative study of the two fundamental approaches for parallelizing automated configuration procedures—multiple independent runs and fine-grained parallelization—investigating how effective each of them is in isolation and to which extent they complement each other. We showed that the generic multiple independent runs parallelization approach is suprisingly effective when applied to the state-of-the-art configuration procedures PARAMILS and SMAC. We also introduced D-SMAC, a fine-grained parallelization of the state-of-the-art model-based algorithm configuration procedure SMAC, and showed that it achieves even better parallelization efficiencies, with speedups up to around 50 when using 64 parallel worker processes on a cluster of standard quad-core machines. Overall, we showed that using 64 parallel workers can reduce the wall clock time necessary for a range of challenging algorithm configuration tasks from 2 days to 1-2 hours. We believe that reductions of this magnitude substantially expand the practical applicability of existing algorithm configuration procedures and further facilitate their integration into the algorithm design process.

References

[1] F. Hutter, D. Babić, H. H. Hoos, and A. J. Hu. Boosting verification by automatic tuning of decision procedures. In *Proc. of FMCAD'07*, pages 27–34. IEEE Computer Society, 2007.

[2] F. Hutter, H. H. Hoos, and K. Leyton-Brown. Automated configuration of mixed integer programming solvers. In *Proc. of CPAIOR-10*, pages 186–202, 2010.

[3] C. Fawcett, M. Helmert, H. H. Hoos, E. Karpas, G. Röger, and J. Seipp. FD-Autotune: Domain-specific configuration using fast-downward. In *Proc. of ICAPS-PAL 2011*, 2011. (8 pages).

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

[5] C. Ansotegui, M. Sellmann, and K. Tierney. A gender-based genetic algorithm for the automatic configuration of solvers. In *Proc. of CP-09*, pages 142–157, 2009.

[6] F. Hutter, H. H. Hoos, and K. Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *Proc. of LION-5*, volume 6683 of *LNCS*, pages 507–523, 2011.

[7] H. H. Hoos and T. Stützle. Local search algorithms for SAT: An empirical evaluation. *Journal of Automated Reasoning*, 24(4):421–481, 2000.

[8] C. P. Gomes, B. Selman, N. Crato, and H. Kautz. Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. *Journal of Algorithms*, 24(1), 2000.

[9] C.C. Ribeiro, I. Rosseti, and R. Vallejos. On the use of run time distributions to evaluate and compare stochastic local search algorithms. In *Proc. of SLS-09*, pages 16–30, 2009.

[10] H. H. Hoos and T. Stützle. Towards a characterisation of the behaviour of stochastic local search algorithms for SAT. *Artificial Intelligence*, 112(1–2):213–232, 1999.

[11] F. Hutter. *Automated Configuration of Algorithms for Solving Hard Computational Problems*. PhD thesis, University Of British Columbia, Department of Computer Science, Vancouver, Canada, October 2009.

[12] M. López-Ibáñez, J. Dubois-Lacoste, T. Stützle, and M. Birattari. The `irace` package, iterated race for automatic algorithm configuration. Technical Report TR/IRIDIA/2011-004, IRIDIA, Université Libre de Bruxelles, Belgium, 2011.

[13] D. R. Jones. A taxonomy of global optimization methods based on response surfaces. *Journal of Global Optimization*, 21(4):345–383, 2001.

[14] M. Schonlau, W. J. Welch, and D. R. Jones. Global versus local search in constrained optimization of computer models. In N. Flournoy, W.F. Rosenberger, and W.K. Wong, editors, *New Developments and Applications in Experimental Design*, volume 34, pages 11–25. Institute of Mathematical Statistics, Hayward, California, 1998.

[15] N. Srinivas, A. Krause, S. Kakade, and M. Seeger. Gaussian process optimization in the bandit setting: No regret and experimental design. In *Proc. of ICML-10*, 2010.

[16] D. Ginsbourger, R. Le Riche, and L. Carraro. *Computational Intelligence in Expensive Optimization Problems*, chapter Kriging Is Well-Suited to Parallelize Optimization, pages 131–162. Springer, Berlin, Germany, 2010.

[17] C. Nell, C. Fawcett, H. Hoos, and K. Leyton-Brown. HAL: A framework for the automated analysis and design of high-performance algorithms. In *Proc. of LION-5*, volume 6683 of *LNCS*, pages 600–615, 2011.

[18] A. Balint, D. Gall, G. Kapler, R. Retz, D. Diepold, and S. Gerber. EDACC - an advanced platform for the experiment design, administration and analysis of empirical algorithms. In *Proc. of LION-5*, volume 6683 of *LNCS*, pages 586–599, 2011.

[19] A. Atamtürk. On the facets of the mixed–integer knapsack polyhedron. *Mathematical Programming*, 98:145–175, 2003.

[20] A. Atamtürk and J. C. Muñoz. A study of the lot-sizing polytope. *Mathematical Programming*, 99:443–465, 2004.

[21] K. Leyton-Brown, M. Pearson, and Y. Shoham. Towards a universal test suite for combinatorial auction algorithms. In *Proc. of EC'00*, pages 66–76, 2000.

[22] C.P. Gomes, Willem-Jan van Hoeve, and Ashish Sabharwal. Connections in networks: A hybrid approach. In *Proc. of CPAIOR-08*, pages 303–307, 2008.

[23] M. Cote, B. Gendron, and L. Rousseau. Grammar-based integer programing models for multi-activity shift scheduling. Technical Report CIRRELT-2010-01, Centre interuniversitaire de recherche sur les réseaux d'entreprise, la logistique et le transport, 2010.

[24] K. Ahmadizadeh, C.P. Dilkina, B.and Gomes, and A. Sabharwal. An empirical study of optimization for maximizing diffusion in networks. In *Proc. of CP-10*, 2010.