From Sequential Algorithm Selection to Parallel Portfolio Selection

M. Lindauer[1] and H. Hoos[2] and F. Hutter[1]

[1]University of Freiburg and [2]University of British Columbia

Abstract. In view of the increasing importance of hardware parallelism, a natural extension of per-instance algorithm selection is to select a set of algorithms to be run in parallel on a given problem instance, based on features of that instance. Here, we explore how existing algorithm selection techniques can be effectively parallelized. To this end, we leverage the machine learning models used by existing sequential algorithm selectors, such as *3S*, *ISAC*, *SATzilla* and *ME-ASP*, and modify their selection procedures to produce a ranking of the given candidate algorithms; we then select the top n algorithms under this ranking to be run in parallel on n processing units. Furthermore, we adapt the pre-solving schedules obtained by *aspeed* to be effective in a parallel setting with different time budgets for each processing unit. Our empirical results demonstrate that, using 4 processing units, the best of our methods achieves a 12-fold average speedup over the best single solver on a broad set of challenging scenarios from the algorithm selection library.

Keywords: Algorithm Selection, Parallel Portfolios, Constraint Solving, Answer Set Programming

1 Introduction

For many challenging computational problems, such as SAT, ASP or QBF, there is no single dominant solver. Instead, the state of the art for these problems consists of a set of non-dominated solvers, each of which performs best on certain types of problem instances. In this situation, per-instance automated algorithm selection techniques can be used to leverage the strength of such complementary sets, or portfolios, of solvers (see, e.g., [27, 16]). Fundamentally, for a new problem instance, these techniques map a set of cheaply computable instance features to a solver to be run. This mapping is typically learned, using machine learning techniques, from a representative set of training data. Unfortunately, even the best per-instance algorithm selection techniques do not always succeed in identifying the best solver for all problem instances, and their performance can suffer as a result of such incorrect selections.

Considering the fact that increases in computational power are nowadays primarily achieved through increased hardware parallelism, one approach for improving instance-based algorithm selection techniques is to select not one, but multiple solvers from a given portfolio, and to run these in parallel. The key idea

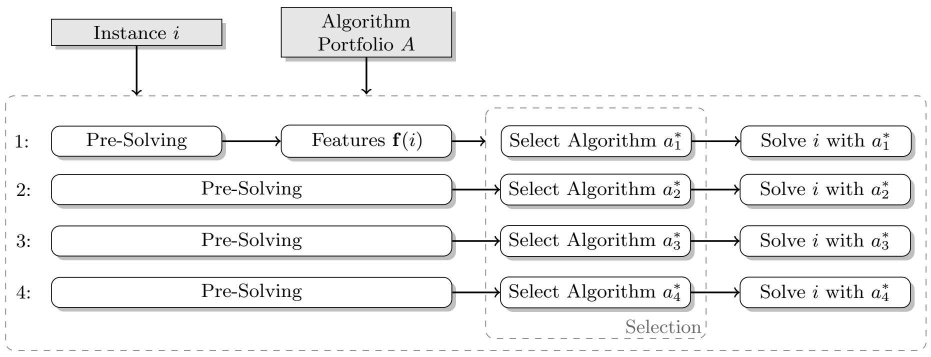


Fig. 1: Parallel portfolio selection with parallel pre-solving for four processing units.

behind this approach is to hedge against incorrect single-solver selections while exploiting readily available parallelism. There is some evidence in the literature that manually crafted per-instance parallel portfolio selectors can achieve impressive performance. For example, the portfolio SAT solver *CSHCpar* [21, 22] won the open parallel track in the 2013 SAT Competition. The idea of *CSHCpar* is simple yet effective: It always runs, independently and in parallel, the parallel SAT solver *Plingeling* with 4 threads, the sequential SAT solver *CCASat*, and three per-instance selected solvers. These per-instance solvers are selected by three models that are trained on application, hard-combinatorial and random SAT instances, respectively. While *CSHCpar* is particularly designed for the SAT Competition with its 8 available cores and its three types of instances, in the following, we investigate a general, fully automated approach for selecting parallel portfolios without any of the special assumptions underlying *CSHCpar*.

Given the large variety of existing sequential algorithm selectors, we study the question how such existing selectors can be effectively parallelized. To this end, we use the learned models of sequential algorithm selectors and modify the selection procedure such that we rank algorithms for a given instance and select the top n algorithms for n processing units (e.g., processors or processor cores.)

State-of-the-art algorithm selectors make extensive use of pre-solving schedules, i.e., they run a sequence of solvers prior to per-instance algorithm selection [31, 14]. This makes it possible to solve easy instances quickly, without inducing the overhead of feature computation. To effectively use parallel resources and minimize sequential bottlenecks, our approach uses parallel rather than sequential pre-solving schedules, which can be obtained using parallel algorithm schedule systems, such as, *aspeed* [8] or *3S* [14].

Figure 1 shows the extension of sequential algorithm selection to parallel portfolio selection with pre-solving schedules. On the first processing unit, we execute the standard workflow of sequential algorithm selectors: to solve a given instance i, we run a pre-solving schedule for a limited amount of time (e.g., 10% of the overall time budget [14]); if the pre-solving schedule fails to solve i, we compute instance features $\mathbf{f}(i)$ (i.e., numerical properties of the instance), and then, based on $\mathbf{f}(i)$, select the putatively best algorithm for the given instance. In the parallel workflow, we can spend the time used by feature computation for longer pre-solving schedules on all threads but the first.

To ensure the scalability of parallel portfolios selection to many algorithms and processing units, we aim to develop methods that satisfy the following requirement:

(i) the online selection of parallel portfolios has to be efficient, i.e., polynomial in the size of the parallel portfolio.

Our general methods for parallel portfolio selection are applicable in any scenario for which the following assumptions hold:

(ii) the algorithm portfolio consists of deterministic algorithms; and on each processing unit, we select a different algorithm;
(iii) algorithms running in parallel do not communicate (e.g., no clause sharing of SAT solvers); and
(iv) we do not have special structural knowledge about the problem domain (e.g., we do not know that SAT instances can be divided into three types).

Assumption (ii) simplifies the selection of a parallel portfolio because there is no noise in the training data, and repeated runs of algorithms are not increasing the chance of solving an instance such that we should select each algorithm at most once. Since communication between algorithms often results in stochastic behavior (e.g., nearly all parallel SAT solvers with clause sharing are stochastic), Assumption (iii) helps to satisfy Assumption (ii). Furthermore, if algorithms would communicate, the performance of an algorithm could not be estimated independently from the other algorithms in the portfolio. We note that Assumption (ii) does not allow the selection of parallel algorithms, e.g., the parallel SAT solver *Plingeling*. Last but not least, Assumption (iv) states that, in contrast to the *CSHCpar* solver, we have no structural knowledge about the problem domain, since such knowledge is only available for specific problems.

Given Assumption (ii), our approach cannot utilize more processing units than there are algorithms in our portfolio (4 to 31 algorithms in our experiments). Therefore, the approaches we consider focus on parallelization with a relatively modest number of processing units, as found in current off-the-shelf computing hardware. Other approaches exist for scaling to higher degrees of parallelism (see, e.g., [3]).

In the following, we will first discuss related work (Section 2). Next, in Section 3, we extend well-known algorithm selection approaches from *SATzilla* [31], *ME-ASP* [23], *ISAC* [15], and *3S* [14] to parallel portfolio selection respecting our requirements. Then, in Section 4, we adapt the algorithm (pre-solving) schedules of *aspeed* [8] to the setting of Figure 1, with different time budgets for each processing unit. Finally, we present an evaluation of our per-instance parallel portfolios on scenarios of the algorithm selection library[1], which allows a fair

[1] `aslib.net`

and thorough evaluation on a set of 12 different constraint solving domains from SAT, MAXSAT, CSP, QBF, and ASP, and pre-marshalling.

2 Related Work

Our work draws on two lines of research reaching back to John Rice's seminal work on algorithm selection [27] and the work by Huberman et al. [10] on parallel algorithm portfolios. It addresses the *dynamic resource allocation* challenge, which has been identified as one of the seven challenges in parallel SAT solving [7].

Recently, the algorithm selection approach of *CSHC* [21], which is based on cost-sensitive hierarchical clustering, was extended to selection of parallel portfolios in *CSHCpar* [22][2]. This approach differs from ours in that it relies on explicitly identified, distinct classes of problem instances (as is the case with the different tracks of the SAT Competition) and provides no obvious way of adjusting the number of processing units.

The extension of *3S* [14] to parallel portfolio selection, dubbed *3Spar* [20][2], selects a parallel portfolio using k-NN to find the k most similar instances in instance feature space. Using integer linear programming (ILP), *3Spar* constructs a static pre-solving schedule offline and a per-instance parallel algorithm schedule online, based on training data of the k most similar instances. The ILP problem that needs to be solved for every instance is *NP*-hard and its time complexity grows with the number of parallel processing units and number of available solvers. Unlike our approach, during the feature computation phase, *3Spar* runs in a purely sequential manner. Since feature computation can require a considerable amount of time (e.g., more than 100 seconds on industrial SAT instances), this can leave important performance potential untapped.

ISAC [15] combines algorithm configuration and algorithm selection by (i) clustering the training instances in the feature space and (ii) using an algorithm configuration procedure [12, 2] to optimize a parametric solver on each cluster. For a new problem instance i to be solved, *ISAC* selects the configuration which was determined for the cluster closest to i. The most recent *ISAC* version, *ISAC* 2.0[3], performs only algorithm selection and uses the best of a fixed set of algorithms in Step (ii); it also provides a method for selecting parallel portfolios for each cluster of instances by searching over all $\binom{|A|}{n}$ combinations of $|A|$ algorithms and n processing units. As this approach quickly becomes infeasible for growing $|A|$ and n, Yuri Malitsky, author of *ISAC* 2.0, recommends to limit its use to at most 4 processing units (README file).

The *aspeed* system [8] solves a similar scheduling problem as *3Spar*, but generates a static algorithm schedule during an off-line training phase, thus avoiding overhead in the solving phase. Unlike *3Spar*, *aspeed* does not support including parallel solvers in the algorithm schedule, and the algorithm schedule is static and not per-instance selected. For this reason, *aspeed* is not directly

[2] Unfortunately, no implementation of *CSHCpar* and *3Spar* is publicly available.
[3] `https://sites.google.com/site/yurimalitsky/downloads`

applicable to per-instance selection of parallel portfolios; however, our approach uses it to effectively compute parallel pre-solving schedules.

RSR-WG [34] combines a case-based-reasoning approach from *CP-Hydra* [24] with greedy construction of parallel portfolio schedules via *GASS* [28] for CSPs. Since the schedules are constructed on a per-instance basis, *RSR-WG* relies on instance features. In the first step, a schedule is greedily constructed to maximize the number of instances solved within a given cutoff time, and in the second step, the components of the schedule are distributed over the available processing units. In contrast to our approach, *RSR-WG* optimizes the number of timeouts and is not directly applicable to arbitrary performance metrics. Since the schedules are optimized online on a per-instance base, *RSR-WG* has to solve an *NP*-hard problem for each instance, which is done heuristically.

Finally, there is some work on parallel portfolios with dynamically adjusted timeshares (see e.g., [5]). Such approaches could eventually be used to dynamically adjust a portfolio determined by any of the methods we study in the following.

3 Selection of Parallel Portfolios

In this section, we show how to extend existing sequential algorithm selection approaches to handle parallel portfolio selection. Formally, the selection of parallel portfolios is an extension of the per-instance algorithm selection problem, in which not only one algorithm is selected, but rather a set of algorithms to be run in parallel.

Definition 1. *A* per-instance parallel portfolio selection problem *can be defined by a 5-tuple* $\langle I, \mathcal{D}, A, U, m \rangle$*, where*

- I *is a set of instances of a problem,*
- $\mathcal{D}$ *is a probability distribution over* I*,*
- A *is a set of algorithms for instances in* I
- U *is a set of parallel processing units available, and*
- $m : I \times A \to \mathbb{R}$ *is a performance metric measuring the performance of algorithm* $a \in A$ *on instance* $i \in I$*.*

A solution of this problem is a mapping $\phi_u : I \to A$ *for each processing unit* $u \in U$*; we refer to such a mapping as a* parallel selection portfolio*. The performance metric we aim to minimize across the possible parallel selection portfolios is* $\mathbb{E}_{i \sim \mathcal{D}} \min_{u \in U} m(i, \phi_u(i))$*.*

Since we assume that the algorithms do not communicate with each other (Assumption (iii)), the performance of a parallel selection portfolio is the performance of the best algorithm in the selected portfolio. Therefore, a perfect parallel selection portfolio would, for each instance, select a set of algorithms containing the best algorithm for that instance. Ultimately, we would therefore like to model the per-instance correlations between solvers to select complementary sets of solvers for each instance.

In this work, however, we pursue a different approach, namely that of generically extending the various existing sequential selection strategies to the parallel selection setting, with the goal of assessing the merit of this overall approach and of empirically studying which sequential selection strategies lend themselves well to this setting. Since these existing sequential selection strategies do not model per-instance correlation between the algorithms, we restrict ourselves to constructing the portfolio in a greedy fashion, choosing the n solvers individually predicted to be best for a parallel portfolio on n processing units. Such a ranking of algorithms is admitted by most algorithm selection approaches [17].

Our approach requires, for each sequential algorithm selection mechanism under consideration, a scoring function

$$s : I \times A \rightarrow \mathbb{R} \tag{1}$$

that ranks the candidate algorithms for a given instance to be solved, such that the putatively best algorithm receives the lowest score value, the second best the second lowest score, etc. Then we simply sort the algorithms in A based on their scorses (breaking ties arbitrarily), using time $\mathcal{O}(|A| \log |A|)$. Thus, if we can compute the scores efficiently, we obtain a computationally efficient approach to parallel algorithm selection, satisfying Requirement (i). In the following, we show that we can indeed efficiently compute such scores for five prominent algorithm selection approaches.

Performance-based Nearest Neighbor (PNN) The algorithm selection approach in *3S* [21] in its simplest form uses a k-nearest neighbour approach. For a new instance i with features $\mathbf{f}(i)$, it finds the k nearest training instances $I_k(i)$ in the feature space F and selects the algorithm that has the best training performance on them. Formally, given a performance metric $m : I \times A \rightarrow \mathbb{R}$, we define $m_k(i, a) = \sum_{i' \in I_k(i)} m(i', a)$ and select algorithm $\arg\min_{a \in A} m_k(i, a)$.

To extend this approach to parallel portfolios, we determine the same k nearest training instances $I_k(i)$ and simply select the n algorithms with the best performance for I_k. Formally, our scoring function in this case is simply:

$$s_{PNN}(i, a) = m_k(i, a). \tag{2}$$

In terms of complexity, identifying the k nearest instances costs time $\mathcal{O}(\#f \cdot |I| \cdot \log |I|)$, with $\#f$ denoting the number of used instance features; and averaging the performance values over the k instances costs time $\mathcal{O}(k \cdot |A|)$.

Distance-based Nearest Neighbor (DNN) *ME-ASP* [23] implements an interface for different machine learning approaches used in its selection framework, but its released version uses a simple nearest neighbour approach with neighbourhood size 1, which also worked best empirically [23]. At training time, this approach memorizes the best algorithm $a^*(i')$ on each training instance $i' \in I$. For a new instance i, it finds the nearest training instance i' in the feature space and selects the algorithm $a^*(i')$ associated with that instance.

To extend this approach to parallel portfolios, for a new test instance i, we score each algorithm a by the minimum of the distances between i and any training instance associated with a. Formally, letting $d(\mathbf{f}(i), \mathbf{f}(i'))$ denote the distance in feature space between instance i and i', we have the following scoring function:

$$s_{DNN}(i, a) = \min\{d(\mathbf{f}(i), \mathbf{f}(i')) \mid i' \in I \wedge a^*(i') = a\}. \tag{3}$$

If $\{i' \in I \mid a^*(i') = a\}$ is empty (because algorithm a was never the best algorithm on an instance) then $s_{DNN}(i, a) = \infty$ for all instances i. Since we memorize the best algorithm for each instance in the training phase, the time complexity of this method is dominated by the cost of computing the distance of each training instance to the test instance, $\mathcal{O}(|I| \cdot \#f)$, where $\#f$ is the number of features.

Clustering The selection part of *ISAC* [15][4] uses a technique similar to *ME-ASP*'s distance-based NN approach, with the difference that it operates on clusters of training instances instead of on single instances. Specifically, *ISAC* clusters the training instances, memorizing the cluster centers Z (in the feature space) and the best algorithms $\hat{a}(z)$ for each cluster $z \in Z$. For a new instance, similar to *ME-ASP*, it finds the nearest cluster z in the feature space and selects the algorithm associated with z.

To extend this approach to parallel portfolios, for a new test instance i, we score each algorithm a by the minimum of the distances between i and any cluster associated with a. Formally, using $d(\mathbf{f}(i), z)$ to denote the distance in feature space between instance i and cluster center z, we have the following scoring function:

$$s_{Clu}(i, a) = \min\{d(\mathbf{f}(i), z) \mid z \in Z \wedge \hat{a}(z) = a\}. \tag{4}$$

As for DNN, if $\{z \in Z \mid \hat{a}(z) = a\}$ is empty (because algorithm a was not the best algorithm on any cluster) then $s_{Clu}(i, a) = \infty$ for all instances i. The time complexity is as for DNN, replacing the number of training instances $|I|$ with the number of clusters $|Z|$.

Regression The first version of *SATzilla* [31] used a regression approach, which, for each $a \in A$, learns a regression model $r_a : F \to \mathbb{R}$ to predict performance on new instances. For a new instance i with features $\mathbf{f}(i)$, it selected the algorithm with the best predicted performance, i.e., $\arg\min_{a \in A} r_a(\mathbf{f}(i))$.

This approach trivially extends to parallel portfolios; we simply use scoring function

$$s_{Reg}(i, a) = r_a(\mathbf{f}(i)) \tag{5}$$

to select the n algorithms predicted to perform best. The complexity of model evaluations differs across models, but it is a polynomial for all models in common use; we denote this polynomial by P_{reg}. Since we need to evaluate one model per algorithm, the time complexity to select a parallel portfolio is then $\mathcal{O}(P_{reg} \cdot |A|)$.

[4] In its original version, *ISAC* is a combination of algorithm configuration and selection, but only the selection approach was used in later publications.

Pairwise Voting The most recent *SATzilla* version [32] uses cost-sensitive random forest classification to learn for each pair of algorithms $a_1 \neq a_2 \in A$ which of them performs better for a given instance; each such classifier $c_{a_1,a_2} : F \rightarrow \{0,1\}$ votes for a_1 or a_2 to perform better, and *SATzilla* then selects the algorithms with the most votes from all pairwise comparisons. Formally, let $v(i,a) = \sum_{a' \in A \setminus \{a\}} c_{a,a'}(\mathbf{f}(i'))$ denote the sum of votes algorithm a receives for instance i; then, *SATzilla* selects $\arg\max_{a \in A} v(i,a)$.

To extend this approach to parallel portfolios, we simply select the n algorithms with the most votes by defining our scoring function to be minimized as:

$$s_{Vote}(i,a) = -v(i,a). \tag{6}$$

As for regression models, the time complexity for evaluating a learned classifier differs across classifier types, but it is polynomial for all commonly-used types; we denote this polynomial function by P_{class}. Since we need to evaluate pairwise classifiers for all algorithm pairs, the time complexity to select a parallel portfolio is in $\mathcal{O}(P_{class} \cdot |A|^2)$.

4 Parallel Pre-Solving Schedules

State-of-the-art algorithm selectors commonly make use of algorithm schedules for pre-solving, i.e., they run a sequence of solvers prior to per-instance algorithm selection [31, 14]. If one of the pre-solvers already solves a given instance, we do not need to compute instance features for the algorithm selection phase and save the time induced by the feature computation.

Malitsky et al. [21] and Hoos et al. [8] have already presented how to find timeout-optimal parallel algorithm schedules. In their settings, the schedules on all processing units get the same amount of runtime. However, as shown in Figure 1, the computation of instance features is limited to one processing unit, and we can run longer pre-solving schedules on all other units. The feature computation time differs from instance to instance, but since we compute our presolving schedule offline, we require a constant estimate of the feature computation runtime, $FeatT$. To err on the pessismistic side, in each algorithm selection scenario we estimate $FeatT$ as the maximal feature computation time observed across the scenario's training instances.

We added a constraint to the flexible and declarative Answer Set Programming (ASP [4]) encoding of *aspeed* [8][5] to ensure that the pre-solving schedule on the first core is limited by a maximal pre-solving time budget, $PreT$. All pre-solving schedules on the other processing units are given an additional budget of $FeatT$ to ensure we use them while the first core computes features. Please refer to Listing 1.1 for our ASP encoding.

The problem of optimizing an algorithm schedule is NP-hard. However, the empirical results of Hoos et al. [8] indicated that the problem of optimizing parallel schedules gets easier with more processing units. In contrast, $ISAC$ has

[5] Since *3S* [21] is not publicly available, using it was not an option.

```
:- not [ slice(1,A,T) = T ] PreT.
:- not [ slice(U,A,T) = T : U != 1 ] PreT+FeatT, unit(U).
```

Listing 1.1: ASP constraints in the language of the ASP grounder gringo [6]. `slice(U,A,T)` denotes that algorithm `A` has a runtime slice `T` on processing unit `U`. The first integrity contraint limits the sum of runtimes `T` of all algorithms `A` on processing unit 1 by the maximal pre-solving runtime `PreT` (an external constant). The second integrity constraint does the same for all other units, but extends the maximal pre-solving runtime by the feature computation time `FeatT` (external constant).

to solve a problem offline that gets more complex with more processing units and is not applicable with more than 4 units.

5 Empirical Evaluation

We now turn to an empirical assessment of our parallel portfolio selection approaches on twelve algorithm selection scenarios that make up the Algorithm Selection Library (*ASlib*).[6] These scenarios involve a wide range of hard combinatorial problems; each of them includes the performance data of a range of solvers for a set of instances, instance features[7] organized in feature groups (we use the default feature groups), and associated costs for these features (see Table 1). We refer to the *ASlib* website (`aslib.net`) for the details on all scenarios; we chose *ASlib* as the basis for our evaluation since this allows us to compare our approach in a fair and uniform way against other algorithm selection methods. Since all experiments are based on the data in the scenarios, we did not need to run any of the algorithms in the portfolio. This ensures repeatability of our experiments, but it also means that resource contention between algorithms running in parallel are not reflected in our results. Depending on the hardware used (e.g., multi-core vs. multi-processor systems), performance may be reduced when running too many algorithms in parallel.

Setup We implemented our parallel selection approach in the open-source and flexible algorithm selection framework of *claspfolio 2* (2.1.0; using *scikit-learn* 0.14.1 [25]).[8] For the choice of machine learning models, *claspfolio 2* follows the implementations of well-known algorithm selectors; we used random forests for pairwise voting (*SATzilla*11 [32]), ridge regression for regression (*SATzilla'09* [31]) and k-means for clustering (*ISAC* [15]).[9]

[6] Since the *CSP-2010* scenario consists of only 2 algorithms, it already admits a perfect portfolio using two processing units. Therefore, we excluded it from our experiments.

[7] Instance features typically consist of cheap syntactic features, such as number of variables and number of clauses, and probing features, i.e., extracting runtime statistics by running an algorithm for a short time on a given instance.

[8] `www.cs.uni-potsdam.de/claspfolio/`

[9] The original *ISAC* [15] uses g-means, which automatically determines the number of clusters. In preliminary experiments, we observed that the square root of the number

Scenario	$\|I\|$	$\|U\|$	$\|A\|$	$\#f$	$\#f_g$	$\varnothing t_f$	t_c	Ref.
ASP-POTASSCO	1294	82	11	138	4	1.3	600	[9]
MAXSAT12-PMS	876	129	6	37	1	0.1	2100	[1, 13]
PREMARSHALLING	527	0	4	16	1	0	3600	[29]
PROTEUS-2014	4021	428	22	198	4	6.4	3600	[11]
QBF-2011	1368	314	5	46	1	0	3600	[18, 26]
SAT11-HAND	296	77	15	115	10	41.2	5000	[32, 13]
SAT11-INDU	300	47	18	115	10	135.3	5000	[32, 13]
SAT11-RAND	600	108	9	115	10	22.0	5000	[32, 13]
SAT12-ALL	1614	20	31	115	10	40.5	1200	[33, 13]
SAT12-HAND	767	229	31	115	10	39.0	1200	[33, 13]
SAT12-INDU	1167	209	31	115	10	80.9	1200	[33, 13]
SAT12-RAND	1362	322	31	115	10	9.0	1200	[33, 13]

Table 1: The *ASlib* algorithm selection scenarios – information on the number of instances $|I|$, number of unsolvable instances $|U|$ ($U \subset I$), number of algorithms $|A|$, number of features $\#f$, number of feature groups $\#f_g$, the average feature computation cost of the used default features $\varnothing t_f$, and runtime cutoff t_c.

Within *claspfolio 2*, we use *aspeed* [8] with the ASP tools *gringo* (3.0.5) and *clasp* (2.2) [6] with a time budget of at most 300 CPU seconds to effectively compute pre-solving schedules. Our pre-solving schedules are limited to at most 256 seconds on the first processing unit and an additional 10% of the runtime cutoff on the other processing units (10% of the runtime cutoff is the maximal feature computation runtime - parameter of *claspfolio 2*; the runtime cutoff differs across the *ASlib* scenarios).

Since speedup is a commonly used performance metric in parallelization and PAR10 (penalized average runtime, counting each timeout as 10 times the runtime cutoff) is a commonly used performance metric in algorithm selection, we assessed our approaches based on PAR10-speedups over the (sequential) single best algorithm (*SB*) in the given algorithm portfolios. We note that the possible speedup is bounded from above by the performance of a perfect algorithm selector (the virtual best solver *VBS*) that always selects the best algorithm for a given instance without inducing feature computation costs. We used 10-fold cross validation (i.e., 10 different training and test splits) to obtain an unbiased performance estimate for *claspfolio 2*, as given in *ASlib*. To avoid artificially inflating PAR10 scores, we removed from the test sets all instances that could be solved neither by any of the candidate algorithms nor during feature computation. Furthermore, to verify which approaches performed statistically indistinguishable from the best approach, we used permutation tests with 100 000 permutations at significance level $\alpha = 0.05$.

Comparison of approaches within *claspfolio 2* In Table 2, we report performance results for the approaches presented in Section 3 as implemented in

of instances gives a good upper bound for the number of clusters; therefore, we did not used g-means but k-means with this cluster bound.

$\lvert U \rvert$	1	2	4	8
ASP-POTASSCO (*VBS*: 25.1)				
DNN	2.0	3.8	6.3	**18.7**
clustering	2.7	3.7	5.8	15.2
PNN	**4.1**	**5.0**	**8.9**	8.6
pairwise-voting	**3.2**	**4.7**	**7.3**	10.8
regression	2.3	3.9	**8.6**	**18.2**
SB	1.0	3.0	**7.4**	8.8
MAXSAT12-PMS (*VBS*: 51.8)				
DNN	**8.4**	**21.4**	**51.5**	(51.8)
clustering	4.8	10.4	21.2	(51.8)
PNN	4.7	7.2	11.6	(51.8)
pairwise-voting	**7.7**	**15.8**	**31.1**	(51.8)
regression	4.7	6.4	21.6	(51.8)
SB	1.0	1.3	1.8	(51.8)
PREMARSHALLING (*VBS*: 30.8)				
DNN	2.7	5.8	(30.8)	(30.8)
clustering	2.5	**7.0**	(30.8)	(30.8)
PNN	2.4	4.4	(30.8)	(30.8)
pairwise-voting	**2.8**	**7.6**	(30.8)	(30.8)
regression	2.6	4.7	(30.8)	(30.8)
SB	1.0	1.4	(30.8)	(30.8)
PROTEUS-2014 (*VBS*: 408.9)				
DNN	5.2	9.2	**19.6**	**46.0**
clustering	**7.4**	7.7	14.2	30.2
PNN	5.9	**9.5**	**23.3**	**50.9**
pairwise-voting	5.8	**11.0**	**20.8**	**42.6**
regression	6.4	**9.8**	**20.4**	**53.8**
SB	1.0	2.7	5.9	10.4

$\lvert U \rvert$	1	2	4	8
QBF-2011 (*VBS*: 95.6)				
DNN	6.7	14.7	33.6	(95.6)
clustering	4.5	10.5	22.4	(95.6)
PNN	6.3	**23.0**	40.6	(95.6)
pairwise-voting	**8.9**	**23.1**	**93.7**	(95.6)
regression	4.7	11.9	60.2	(95.6)
SB	1.0	2.7	13.6	(95.6)
SAT11-HAND (*VBS*: 37.2)				
DNN	**3.2**	**5.2**	**9.6**	**23.9**
clustering	1.6	2.9	4.2	7.0
PNN	2.3	2.8	**8.4**	10.8
pairwise-voting	**3.4**	**4.8**	**8.6**	**10.9**
regression	**2.9**	**4.5**	**8.4**	**12.5**
SB	1.0	1.2	1.9	6.2
SAT11-INDU (*VBS*: 21.4)				
DNN	1.4	**1.9**	**2.6**	**7.8**
clustering	1.3	**1.9**	**3.3**	**5.3**
PNN	1.1	1.5	2.6	**5.2**
pairwise-voting	**2.0**	**2.4**	**3.6**	**4.7**
regression	1.3	**2.0**	**3.6**	**7.8**
SB	1.0	1.7	**2.9**	**7.2**
SAT11-RAND (*VBS*: 65.7)				
DNN	3.8	**11.0**	**42.2**	60.5
clustering	**6.1**	**9.5**	32.3	42.7
PNN	**6.5**	**9.3**	10.7	60.2
pairwise-voting	4.4	8.3	11.4	60.4
regression	5.9	7.8	8.3	60.3
SB	1.0	5.9	6.8	**64.8**

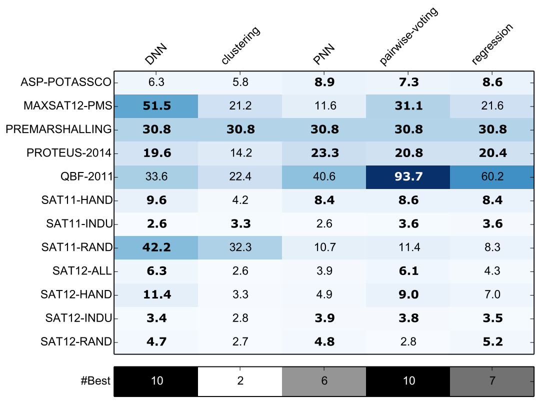
$\lvert U \rvert$	1	2	4	8
SAT12-ALL (*VBS*: 31.6)				
DNN	2.4	**4.2**	**6.3**	**10.7**
clustering	1.5	1.9	2.6	3.5
PNN	2.2	2.9	3.9	7.7
pairwise-voting	**2.8**	**4.1**	**6.1**	9.0
regression	2.1	2.9	4.3	7.1
SB	1.0	1.0	1.4	1.7
SAT12-HAND (*VBS*: 34.7)				
DNN	**3.7**	**6.2**	**11.4**	**14.3**
clustering	1.8	2.3	3.3	4.6
PNN	2.0	2.8	4.9	7.5
pairwise-voting	**4.2**	**5.4**	**9.0**	**12.4**
regression	2.9	4.2	7.0	**9.8**
SB	1.0	1.0	1.4	1.9
SAT12-INDU (*VBS*: 15.4)				
DNN	2.0	2.4	**3.4**	**5.0**
clustering	1.3	2.1	2.8	4.6
kNN	1.6	2.3	**3.9**	**5.7**
pairwise-voting	**2.4**	**3.0**	**3.8**	**5.4**
regression	1.9	2.5	**3.5**	**6.3**
SB	1.0	1.5	2.5	4.8
SAT12-RAND (*VBS*: 12.1)				
DNN	0.8	1.5	**4.7**	**8.6**
clustering	**1.3**	1.7	2.7	4.9
PNN	**1.2**	**2.1**	**4.8**	**7.3**
pairwise-voting	1.1	1.7	2.8	6.4
regression	**1.3**	**1.8**	**5.2**	8.3
SB	1.0	1.5	**4.0**	**6.8**

Table 2: Speedup on PAR10 (wallclock) in comparison to *SB* with one processing unit (U). Entries for which the number of processing units exceed the number of candidate algorithms are marked 'NA'. Entries shown in bold-face are statistically indistinguishable from the best speedups obtained for the respective scenario and number of processing units (according to a permutation test with 100 000 permutations and $\alpha = 0.05$). If more processing are available than algorithms, we run all algorithms and achieve a perfect VBS score (number in parentheses).

claspfolio 2, for 1 to 8 processing units (U). For 4 processing units, the speedup over the single best algorithm is between 2.6 (*SAT11-INDU* with PNN) and 93.7 (*QBF-2011* with pairwise voting). The best parallelization approach differed between the scenarios, which is not too surprising since it is known that there is no single dominant approach for sequential per-instance algorithm selection. We note that the recent work on *autofolio* [19] proposes using algorithm configuration to determine a well-performing algorithm selection approach and its parameters for a given scenario. On average, DNN and pairwise voting performed best across our scenarios; for 4 processing units, both approaches achieved performance levels that were was not significantly worse than the best approach on 10 out of 12 scenarios. The geometric average speedup was 11.89 for DNN and 10.90 for pairwise voting, respectively.[10] In contrast, on one processing unit, DNN performed best on only 3 scenarios and pairwise voting on 9 scenarios. We note that performance differences between the approaches decreased as the number of processing units increased: all approaches got closer to the optimal speedup achieved when running all candidate algorithms in parallel.

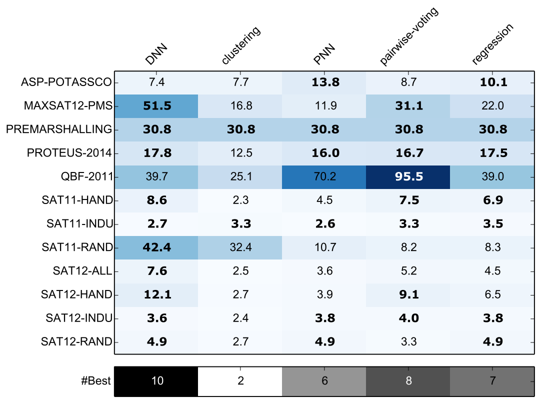
Overall, our approaches and also the *VBS* do not scale as well on some of the SAT scenarios as they do on the other scenarios (e.g., the maximal *VBS* speedup is 95.6 on *QBF-2011* but only 12.1 on *SAT12-RAND*). We speculate that this is

[10] We note that we have to use a geometric average instead of an arithmetic average, because we aggregate over speedup *factors*. This can be seen when considering a case with speedups of 2 and 0.5, where the arithmetic average gives a misleading 1.25.



	DNN	clustering	PNN	pairwise-voting	regression
ASP-POTASSCO	6.3	5.8	**8.9**	**7.3**	**8.6**
MAXSAT12-PMS	**51.5**	21.2	11.6	**31.1**	21.6
PREMARSHALLING	**30.8**	**30.8**	**30.8**	**30.8**	**30.8**
PROTEUS-2014	**19.6**	14.2	**23.3**	**20.8**	**20.4**
QBF-2011	33.6	22.4	40.6	**93.7**	60.2
SAT11-HAND	**9.6**	4.2	**8.4**	**8.6**	**8.4**
SAT11-INDU	**2.6**	**3.3**	2.6	**3.6**	**3.6**
SAT11-RAND	**42.2**	32.3	10.7	11.4	8.3
SAT12-ALL	**6.3**	2.6	3.9	**6.1**	4.3
SAT12-HAND	**11.4**	3.3	4.9	**9.0**	7.0
SAT12-INDU	**3.4**	2.8	**3.9**	**3.8**	**3.5**
SAT12-RAND	**4.7**	2.7	**4.8**	2.8	**5.2**
#Best	10	2	6	10	7

(a) With pre-solving



	DNN	clustering	PNN	pairwise-voting	regression
ASP-POTASSCO	7.4	7.7	**13.8**	8.7	**10.1**
MAXSAT12-PMS	**51.5**	16.8	11.9	**31.1**	22.0
PREMARSHALLING	**30.8**	**30.8**	**30.8**	**30.8**	**30.8**
PROTEUS-2014	**17.8**	12.5	**16.0**	**16.7**	**17.5**
QBF-2011	39.7	25.1	70.2	**95.5**	39.0
SAT11-HAND	**8.6**	2.3	4.5	**7.5**	**6.9**
SAT11-INDU	**2.7**	**3.3**	**2.6**	**3.3**	**3.5**
SAT11-RAND	**42.4**	32.4	10.7	8.2	8.3
SAT12-ALL	**7.6**	2.5	3.6	5.2	4.5
SAT12-HAND	**12.1**	2.7	3.9	**9.1**	6.5
SAT12-INDU	**3.6**	2.4	**3.8**	**4.0**	**3.8**
SAT12-RAND	**4.9**	2.7	**4.9**	3.3	**4.9**
#Best	10	2	6	8	7

(b) Without pre-solving

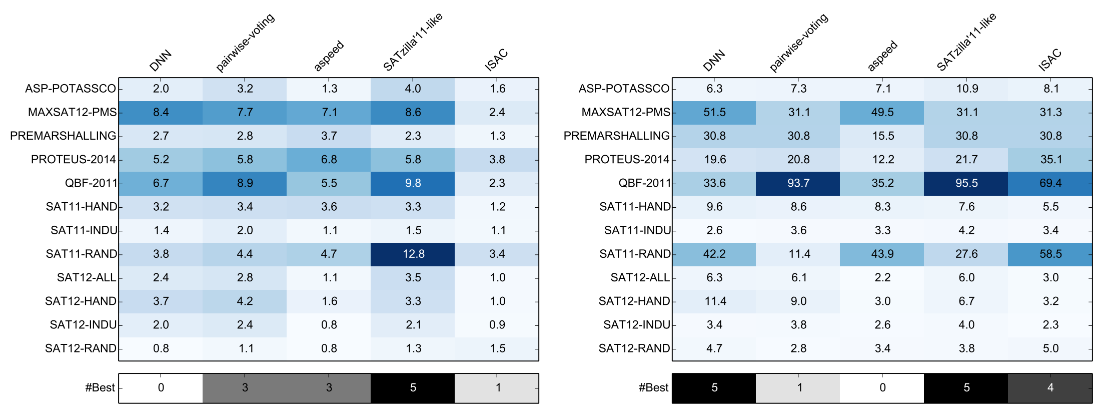
Fig. 2: Heatmap for PAR10 speedups (wallclock) against sequential *SB* on 4 processing units. A value is printed in bold-face if a statistical test (i.e., permutation test with 100 000 and $\alpha = 0.05$) cannot find evidence that it is significantly lower than that of the best approach for a given number of threads. The last row shows how often an approach was en par with the best.

due to (i) the relatively large number of SAT solvers (which makes it harder to perform as well as the *VBS*) and (ii) the relatively low performance correlation between some of those solvers.

As can be seen in Figure 2, using pre-solving schedules improved the performance on 4 out of our 5 approaches on 4 processing units. Surprisingly, the distance-based nearest neighbor approach (DNN) performed slightly better without pre-solving schedules, which we believe may be caused by over-fitting to the training data.

Comparison with other Systems While the previous experiment fixed all design decisions except the selection strategy, we now compare the results for our two best results (DNN and pairwise-voting) with three other strategies: `SATzilla'11-like`, a variant of our pairwise-voting approach in which we restrict the number of presolvers and their time limit to resemble more closely the strategy used in *SATzilla* 2011 [32]; the *aspeed* system, which does not perform per-instance selection, but produces static parallel schedules and is used for per-solver scheduling in *claspfolio 2*; and the *ISAC* system, for which we have written a converter from the *ASlib* format into its native input format. Since *ISAC* determines its cross-validation folds internally and only outputs a single performance number, we cannot perform statistical tests for this experiment and only report the number of times each method performed best, as well as the methods' (geometric) mean speedups.

Figure 3 shows the performance of these systems on 1 and 4 processing units. In the sequential case, `SATzilla'11-like` performed best overall (best on 5 of



	DNN	pairwise-voting	aspeed	SATzilla'11-like	ISAC
ASP-POTASSCO	2.0	3.2	1.3	4.0	1.6
MAXSAT12-PMS	8.4	7.7	7.1	8.6	2.4
PREMARSHALLING	2.7	2.8	3.7	2.3	1.3
PROTEUS-2014	5.2	5.8	6.8	5.8	3.8
QBF-2011	6.7	8.9	5.5	9.8	2.3
SAT11-HAND	3.2	3.4	3.6	3.3	1.2
SAT11-INDU	1.4	2.0	1.1	1.5	1.1
SAT11-RAND	3.8	4.4	4.7	12.8	3.4
SAT12-ALL	2.4	2.8	1.1	3.5	1.0
SAT12-HAND	3.7	4.2	1.6	3.3	1.0
SAT12-INDU	2.0	2.4	0.8	2.1	0.9
SAT12-RAND	0.8	1.1	0.8	1.3	1.5
#Best	0	3	3	5	1

(a) $|U| = 1$

	DNN	pairwise-voting	aspeed	SATzilla'11-like	ISAC
ASP-POTASSCO	6.3	7.3	7.1	10.9	8.1
MAXSAT12-PMS	51.5	31.1	49.5	31.1	31.3
PREMARSHALLING	30.8	30.8	15.5	30.8	30.8
PROTEUS-2014	19.6	20.8	12.2	21.7	35.1
QBF-2011	33.6	93.7	35.2	95.5	69.4
SAT11-HAND	9.6	8.6	8.3	7.6	5.5
SAT11-INDU	2.6	3.6	3.3	4.2	3.4
SAT11-RAND	42.2	11.4	43.9	27.6	58.5
SAT12-ALL	6.3	6.1	2.2	6.0	3.0
SAT12-HAND	11.4	9.0	3.0	6.7	3.2
SAT12-INDU	3.4	3.8	2.6	4.0	2.3
SAT12-RAND	4.7	2.8	3.4	3.8	5.0
#Best	5	1	0	5	4

(b) $|U| = 4$

Fig. 3: Comparison of parallelization approaches of different algorithm selection mechanisms on 1 and 4 processing units; we can't assess statistical difference since *ISAC* only outputs a single performance value.

our 12 benchmarks; average speedup 3.80), followed by pairwise voting (best on 3 benchmarks; average speedup of 3.49), and *aspeed* (best on 4 benchmarks; average speedup 2.34). Using 4 processing units, while `SATzilla'11-like` still performed best (best on 5 benchmarks; average speedup 12.27), it was now closely followed by the other two approaches: DNN (also best on 5 benchmarks, average speedup 11.89) and ISAC (best on 4 benchmarks, average speedup 10.72).

We conclude that, while `SATzilla'11-like` yields stable performance, the performance of different methods scales differently as the number of processing units grows. We also note that, going up to 4 processing units, the best average speedups obtained were roughly linear in the number of units. While *ISAC* should not be used with more than 4 processing units (due to its exponential time requirements in the number of units), Table 2 shows that our methods (especially DNN) even improved further based on 8 processing units, without increasing the effort to train or use them.

6 Conclusions

Motivated by the increasing importance of hardware parallelism, in this work, we considered the problem of selecting a parallel portfolio of solvers based on features of a problem instance to be solved. In particular, we investigated generic ways of extending well-known sequential per-instance algorithm selection methods to produce parallel portfolios. Since current algorithm selectors do not learn or assess per-instance correlation in the performance of candidate solvers, we simply use the scoring (or ranking) function underlying all algorithm selectors to select the n algorithms scored best for a parallel portfolio. A future research goal would be to develop a method to consider the per-instance performance correlation between candidate solvers, which should permit the construction of even better per-instance parallel portfolios.

Our extensive empirical study demonstrated that all methods we considered performed quite well on the large range of scenarios from the algorithm selection library, with speedups from 2.6 to 93.7 on 4 processing units in comparison to running only the single best available algorithm sequentially. Overall, we found our distance-based nearest neighbor (DNN) and pairwise-voting approaches to perform better than other approaches.

However, as for any algorithm selection approach, the performance of our parallel portfolio selectors is bounded by that of an oracle selector, i.e., a perfect algorithm selector that always selects the single best algorithm for a given instance. We see two ways to overcome this obstacle, (i) use of per-instance algorithm configuration [15, 30] to improve the performance of the candidate set of algorithms and hence of the oracle; and (ii) to permit communication between the algorithms in the parallel portfolio (e.g., clause sharing between SAT solvers). Both avenues can potentially be pursued by extending the techniques investigated here.

References

1. Ansótegui, C., Malitsky, Y., Sellmann, M.: Maxsat by improved instance-specific algorithm configuration. In: Proc. of AAAI'14. pp. 2594–2600 (2014)
2. Ansótegui, C., Sellmann, M., Tierney, K.: A gender-based genetic algorithm for the automatic configuration of algorithms. In: Proc. of CP'09. pp. 142–157 (2009)
3. Arbelaez, A., Codognet, P.: From sequential to parallel local search for SAT. In: Procs. of EvoCOP'13. pp. 157–168 (2013)
4. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press (2003)
5. Gagliolo, M., Schmidhuber, J.: Towards distributed algorithm portfolios. In: Proc. of DCAI'08. pp. 634–643 (2008)
6. Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., Schneider, M.: Potassco: The Potsdam answer set solving collection. AICom 24(2), 107–124 (2011)
7. Hamadi, Y., Wintersteiger, C.: Seven challenges in parallel SAT solving. AI Magazine 34(2), 99–106 (2013)
8. Hoos, H., Kaminski, R., Lindauer, M., Schaub, T.: aspeed: Solver scheduling via answer set programming. TPLP First View, 1–26 (2014)
9. Hoos, H., Lindauer, M., Schaub, T.: claspfolio 2: Advances in algorithm selection for answer set programming. TPLP 14, 569–585 (2014)
10. Huberman, B., Lukose, R., Hogg, T.: An economic approach to hard computational problems. Science 275, 51–54 (1997)
11. Hurley, B., Kotthoff, L., Malitsky, Y., O'Sullivan, B.: Proteus: A hierarchical portfolio of solvers and transformations. In: Proc. of CPAIOR'14. pp. 301–317 (2014)
12. Hutter, F., Hoos, H., Leyton-Brown, K., Stützle, T.: ParamILS: An automatic algorithm configuration framework. JAIR 36, 267–306 (2009)

13. Hutter, F., Xu, L., Hoos, H., Leyton-Brown, K.: Algorithm runtime prediction: Methods and evaluation. AIJ 206, 79–111 (2014)
14. Kadioglu, S., Malitsky, Y., Sabharwal, A., Samulowitz, H., Sellmann, M.: Algorithm selection and scheduling. In: Proc. of CP'11. pp. 454–469 (2011)
15. Kadioglu, S., Malitsky, Y., Sellmann, M., Tierney, K.: ISAC - instance-specific algorithm configuration. In: Proc. of ECAI'10. pp. 751–756 (2010)
16. Kotthoff, L.: Algorithm selection for combinatorial search problems: A survey. AI Magazine pp. 48–60 (2014)
17. Kotthoff, L.: Ranking algorithms by performance. In: Proc. of LION'14 (2014)
18. Kotthoff, L., Gent, I., Miguel, I.: An evaluation of machine learning in algorithm selection for search problems. AICom 25(3), 257–270 (2012)
19. Lindauer, M., Hoos, H., Hutter, F., Schaub, T.: Autofolio: Algorithm configuration for algorithm selection. In: Proc. of Workshops at AAAI'15 (2015)
20. Malitsky, Y., Sabharwal, A., Samulowitz, H., Sellmann, M.: Parallel SAT solver selection and scheduling. In: Proc. of CP'12. pp. 512–526 (2012)
21. Malitsky, Y., Sabharwal, A., Samulowitz, H., Sellmann, M.: Algorithm portfolios based on cost-sensitive hierarchical clustering. In: Rossi, F. (ed.) Proc. of IJCAI'13. pp. 608–614 (2013)
22. Malitsky, Y., Sabharwal, A., Samulowitz, H., Sellmann, M.: Parallel lingeling, ccasat, and csch-based portfolio. In: Proc. of SAT Competition 2013. pp. 26–27 (2013)
23. Maratea, M., Pulina, L., Ricca, F.: A multi-engine approach to answer-set programming. Theory and Practice of Logic Programming First View, 1–28 (2013)
24. O'Mahony, E., Hebrard, E., Holland, A., Nugent, C., O'Sullivan, B.: Using case-based reasoning in an algorithm portfolio for constraint solving. In: Proc. of AICS'08 (2008)
25. Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., Duchesnay, E.: Scikit-learn: Machine learning in Python. JMLR 12, 2825–2830 (2011)
26. Pulina, L., Tacchella, A.: A self-adaptive multi-engine solver for quantified boolean formulas. Constraints 14(1), 80–116 (2009)
27. Rice, J.: The algorithm selection problem. Advances in Computers 15, 65–118 (1976)
28. Streeter, M., Golovin, D., Smith, S.: Combining multiple heuristics online. In: Proc. of AAAI'07. pp. 1197–1203 (2007)
29. Tierney, K.: An algorithm selection benchmark of the container pre-marshalling problem. Tech. Rep. DS&OR Working Paper 1402, DS&OR Lab, University of Paderborn (2014)
30. Xu, L., Hoos, H., Leyton-Brown, K.: Hydra: Automatically configuring algorithms for portfolio-based selection. In: Proc. of AAAI'10. pp. 210–216 (2010)
31. Xu, L., Hutter, F., Hoos, H., Leyton-Brown, K.: SATzilla: Portfolio-based algorithm selection for SAT. JAIR 32, 565–606 (2008)
32. Xu, L., Hutter, F., Hoos, H., Leyton-Brown, K.: Evaluating component solver contributions to portfolio-based algorithm selectors. In: Proc. of SAT'12. pp. 228–241 (2012)
33. Xu, L., Hutter, F., Shen, J., Hoos, H., Leyton-Brown, K.: SATzilla2012: improved algorithm selection based on cost-sensitive classification models. In: Proc. of SAT Challenge 2012. pp. 57–58 (2012)
34. Yun, X., Epstein, S.: Learning algorithm portfolios for parallel execution. In: Proc. of LION'12. pp. 323–338 (2012)