ECAI 2023
K. Gal et al. (Eds.)

doi:10.3233/FAIA230398

Automatic Parallel Portfolio Selection

Haniye Kashgarani [a;*] **and Lars Kotthoff** [a]

[a]University of Wyoming
ORCiD ID: Haniye Kashgarani https://orcid.org/0000-0001-6059-1920, Lars Kotthoff https://orcid.org/0000-0003-4635-6873

Abstract. Algorithms to solve hard combinatorial problems often exhibit complementary performance, i.e. where one algorithm fails, another shines. Algorithm portfolios and algorithm selection take advantage of this by running all algorithms in parallel or choosing the best one to run on a problem instance. In this paper, we show that neither of these approaches gives the best possible performance and propose the happy medium of running a subset of all algorithms in parallel. We propose a method to choose this subset automatically for each problem instance, and demonstrate empirical improvements of up to 19% in terms of runtime, 81% in terms of misclassification penalty, and 26% in terms of penalized averaged runtime on scenarios from the ASlib benchmark library. Unlike all other algorithm selection and scheduling approaches in the literature, our performance measures are based on the actual performance for algorithms running in parallel rather than assuming overhead-free parallelization based on sequential performance. Our approach is easy to apply in practice and does not require to solve hard problems to obtain a schedule, unlike other techniques in the literature, while still delivering superior performance.

1 Introduction

For many types of hard combinatorial problems, different algorithms that exhibit complementary performance are available. In these cases, a portfolio of algorithms often achieves better performance than a single one [10, 7]. The algorithms can be run in parallel, or a single one selected for each problem instance to solve. The so-called Algorithm Selection Problem [28] is often solved using machine learning models which, given characteristics of the problem instance to solve, decide which algorithm should be chosen [17, 14]. The machine learning models built for per-instance algorithm selection are not perfect, like most models. In some cases, they lead to choosing an algorithm that does not provide the best overall performance, resulting in wasted resources.

Running all algorithms in parallel avoids this issue, but again wastes resources. Even if the user is only interested in optimizing the elapsed time, i.e. it does not matter how many things are run in parallel, results are sub-optimal as parallel executions compete for shared resources such as caches. With more solvers running in parallel, more runs time out, which results in a large overhead. Even for a relatively small number of parallel runs, this overhead becomes prohibitive, resulting in overall performance worse than using imperfect machine learning models to choose a single algorithm [13].

* Corresponding Author. Email: hkashgar@uwyo.edu

In this paper, we propose a middle path – select the most promising subset of algorithms to be run in parallel on a single non-distributed computing machine. This mitigates the impact of both imperfect machine learning models and overhead from parallel runs. We formalize the problem of choosing a subset of algorithms from a portfolio, unifying approaches from the literature. We propose a solution to this problem based on the predictions of algorithm performance models and their uncertainties and compare empirically to other approaches from the literature. We demonstrate improvements of up to 81% in terms of misclassification penalty, establishing a new state of the art in per-instance algorithm selection with multiple algorithms. We assume that the algorithms to run are not parallelized themselves, i.e. each algorithm consumes the same computational resources, and we run on a single machine. We do not consider the case of running algorithms in a distributed setting on multiple machines.

2 Related Work

2.1 Algorithm Selection

The performance of algorithms designed to solve NP-complete problems, such as Boolean Satisfiability and the Traveling Salesman Problem, can vary significantly depending on the specific problem being addressed. There is no one algorithm that performs optimally in all circumstances. However, we can take advantage of these performance disparities by creating algorithm portfolios that incorporate the complementing strengths of several algorithms [7, 10].

The algorithm portfolios proposed in [7, 10] run multiple algorithms in parallel, however, they do not measure the actual execution time when running in parallel but simulate parallel execution based on sequential performance. [13] found that the performance of the portfolio can deteriorate substantially when algorithms are executed in parallel, in particular for more than 10 algorithms. [20] has also determined that running various configurations of an algorithm in parallel can introduce overhead, and this factor should be considered when designing portfolios. Alternatively, we can choose a subset of the best algorithms from the portfolio for a given problem instance to avoid the overhead of running a large number of solvers in parallel. In the case where we choose only a single algorithm, this is known as the algorithm selection problem [28]. Typically, this is accomplished through the use of machine learning techniques and features derived from the instances [17, 14] and algorithms [27]. However, choosing only a single algorithm to run often achieves suboptimal performance because of incorrect choices. This can be addressed through better algorithm selection models; in this paper, we explore the alternative of choosing more than one algorithm to run in parallel on a single node.

Algorithm selection has been applied successfully in many problem domains. Some of the most prominent systems are SATzilla, Hydra, and Autofolio [34, 21, 32]. While these systems focus on SAT, algorithm selection also has been used in the constraint programming and mixed integer programming domains, where it has been shown to achieve good performance [26, 35]. AutoFolio has been applied in additional areas, e.g. ASP, MAXSAT, and QBF. [15] apply algorithm selection for the TSP, and ME-ASP [24] apply algorithm selection for answer set programming to create a multi-engine solver. Algorithm selection has also been used to choose between evolutionary algorithms [9, 37, 25, 36]. The ASlib benchmarking library [4] collects benchmarks from many different problem domains and is the de facto standard library for evaluating algorithm selection approaches.

Notation. We follow [22] in the notation we use in this paper. Given a portfolio of algorithms (solvers) S, a set of instances I, and a performance metric $m : S \times I \to \mathbb{R}^+$, we aim to find a mapping $s : I \to S$ from instances I to algorithms S such that the performance across all instances is optimized. This performance metric can be for example the time needed to solve the instance and we assume w.l.o.g. that the performance metric should be minimized. In practice, we estimate the value of the performance metric based on the predictions of machine learning models for new problem instances; we denote this estimate $\hat{m}$. We want to select the solver with the best-predicted performance for each instance:

$$min \frac{1}{|I|} \sum_{i \in I} \hat{m}(s(i), i) \tag{1}$$

2.2 Portfolio Scheduling

Different approaches have been proposed for sub-portfolio selection. Some approaches choose a number of suitable solvers for sequential execution and assign time slices that sum to the total available time to each algorithm. Others have implemented parallel execution of the selected solvers, while a few have combined these two methods, utilizing parallelization across computing processors and splitting the available time of each processor across different algorithms.

2.2.1 Time slice allocation on single processor

Typically, sub-portfolio selection strategies are built for sequential solver runs, e.g. Sunny [2] creates a sub-portfolio of solvers using k-nearest neighbor (kNN) models and builds a sequential schedule for the selected solvers by allocating time slices based on the predicted performance. CPHydra [26] also employs case-based reasoning and allocates time slices for the selected CSP solvers to run sequentially. 3S [12] dynamically selects and sequentially schedules solvers for a given SAT instance using integer programming. ASPEED [8] creates static sequential schedules through answer set programming that optimizes a static sequential time budget allocation for solvers. Depending on the number of algorithms, solving the scheduling problem can take substantial time. Building on the methodologies of 3S [12] and ASPEED [8], ISA (Instance-Specific ASPEED) [18] uses kNN to identify the closest training problem instances to a given problem instance to solve. It then employs ASPEED to determine a schedule that minimizes the number of timeouts across these instances. [18] also introduced TSunny which is a modified version of Sunny that limits the number of solvers to run, thus increasing the chance of success by allocating larger time slices to each algorithm.

2.2.2 Time slice allocation on multiple processors

Other portfolio techniques have focused on scheduling solvers to run in parallel and allocating time slots on different processors. P3S [23] is a parallel version of 3S and uses the kNN algorithm for selecting solvers and scheduling them using integer programming with a specific runtime allocation strategy, where it runs a static set of solvers for the first 10% of the available runtime and solvers selected for the instance for the remaining 90%. ASPEED [8] can also define a fixed schedule for running solvers on multiple processors, which is chosen based on the average solver performance across a set of instances. Flexfolio [18] incorporates a reimplementation of the P3S approach utilizing the same 10-90 strategy. However, rather than employing integer programming to address the scheduling problem, Flexfolio makes use of ASPEED and solves it through answer set programming. Sunny-cp [3] can simultaneously execute multiple CSP and COP solvers. First, a portion of the total available time is allocated to a pre-solving phase that follows a fixed schedule. The remaining time is then distributed amongst the other selected solvers dynamically, based on the predictions of kNN performance models. As there are often more solvers than processors, all processors except one are assigned to the corresponding number of top-ranked solvers, while the time on the final processors is split among the remaining solvers.

2.2.3 Running algorithms in parallel

One of the first parallel SAT solvers is ppfolio [29]. It selects solver portfolios to solve sets of problem instances optimally, but does this only for entire sets of instances, not on a per-instance basis as we do here. The success of ppfolio has inspired many other researchers to create sub-portfolios of solvers to run in parallel. For example, [19] extended existing algorithm selectors like 3S, SATzilla, and ME-ASP to greedily choose the top n solvers to run in parallel by producing a ranking of candidate algorithms; however, the number of solvers has to be specified by the user and the actual runtime of parallel runs is not considered – the same runtime as for sequential execution is assumed. Running algorithms in parallel on the same machine is slower than running sequentially in practice due to the overhead incurred because of shared caches and shared memory. This has been shown in experiments [13], simulations [38, 30], and analyses [1] for the parallel executions of solvers – in practice, ignoring the overhead that parallel execution introduces reduces overall performance.

In this paper, we consider the problem of selecting the optimal subset of algorithms to run in parallel on a single machine. This is computationally much easier to solve on a per-instance basis than more complex scheduling approaches, e.g. the ones used by ASPEED and 3S, which means that our method is easier to deploy and introduces less overhead. As long as the best solver is part of the selected portfolio, we will achieve optimal performance or close to it, whereas approaches that allocate time slices may choose the best solver, but fail to achieve optimal performance if too little time is allocated to it. We leverage more information from algorithm selection models than most approaches in the literature, in particular the uncertainty of performance prediction. This allows us to trade off the number of algorithms to choose with the chance of success in a principled way. Our approach is designed to optimize the usage of parallel computational resources when solving combinatorial problems while taking into account the overhead that arises from parallel runs. To the best of our knowledge, there are no other approaches that solve parallel algorithm selection this way.

3 Parallel Portfolio Selection

We aim to choose a sub-portfolio of solvers $P_i \subseteq S$ for a given instance $i \in I$ that includes the algorithms with the best performance on i ($A \in S$ and $A \in P_i$) to run in parallel, based on the predicted performance of each solver. Given the predicted performance metric $\hat{m}$, we can define a total order of the algorithms in the portfolio S for a given instance i. This total order is induced by the ranking of the algorithms based on their predicted performance for instance i. Formally, the total order can be defined as:

$$A < B \quad \text{if} \quad \hat{m}(A,i) < \hat{m}(B,i); A,B \in S \tag{2}$$

Given the total order, the rank of each algorithm A on each instance i can be defined as the number of algorithms that are predicted to be strictly better than A for the instance and denoted $r_{A,i}$. Ties are broken arbitrarily. A portfolio of a specified size n is then defined as the top n algorithms according to rank for that particular instance. The portfolio of size n for instance i can be expressed mathematically as:

$$P_i = \{A \in S \mid r_{A,i} \leq n\} \tag{3}$$

In a slight abuse of notation, we will denote the rank of an algorithm as a subscript, i.e. $r_{A,i}$ is the rank of algorithm A on instance i and $A_{1,i}$ is the algorithm of rank 1 (the best performing algorithm) on instance i.

This allows to choose a subset of algorithms with the best predicted performance for a given instance, which can then be executed in parallel. However, determining the portfolio size n for a given problem instance is the key challenge for parallel portfolios. As discussed above, choosing only a single algorithm or all algorithms is unlikely to give optimal performance in practice. The larger the number of algorithms we include, the larger the chance that the best algorithm is in the chosen portfolio, but also the larger the overhead from running many algorithms in parallel.

Here, we want to include the algorithms that, according to their predicted performance on a new problem instance, have the highest chances of achieving optimal performance, while also taking into account the computational overhead of running multiple solvers in parallel. We leverage the uncertainty of the predictions of the performance models to gauge the likelihood that a given algorithm would be competitive. To the best of our knowledge, there are no algorithm selection approaches that do this.

Instead of considering only a point prediction, we consider the predicted distribution of performance metric values, characterized by its mean and standard deviation. Formally, we denote the standard deviation of the prediction $\hat{m}(A,i)$ as $\sigma_{A,i}$ for each solver A and instance i. We assume that the predictions of our performance models follow a normal distribution, i.e. the predicted value is the mean of that distribution and allows to characterize it completely together with the standard deviation. We assess the likelihood of two algorithms performing equally well by computing the overlap between their distributions. If two algorithms are predicted to perform very similarly, then the overlap between the distributions will be very large.

We are in particular interested in the predicted performance distribution of the best-predicted algorithm $A_{1,i}$ (no algorithms are predicted to perform better than it), and how the predictions for the other algorithms compare to it. Formally, for the best predicted solver $A_{1,i}$ on instance i the distribution of predictions is $\hat{m}(A_{1,i},i) \sim \hat{M}(\mu_{A_{1,i},i}, \sigma^2_{A_{1,i},i})$ with probability density function $f_{A_{1,i},i}$ and cumulative distribution function $F_{A_{1,i},i}$. The performance distributions for other algorithms are defined similarly.

For the distributions of the predicted performance of two algorithms A_x and A_y on instance i, the point of intersection c can be computed as $f_{A_x,i}(c) = f_{A_y,i}(c)$. That is, the predicted probability of achieving this particular performance is equal for both distributions (illustrated in Figure 1). For $\mu_{A_x,i} < \mu_{A_y,i}$, c is defined as (we omit the index i for the sake of brevity here):

$$c = \frac{\mu_{A_y}\sigma^2_{A_x} - \sigma_{A_y}\left(\mu_{A_x}\sigma_{A_y} + \sigma_{A_x}\sqrt{(\mu_{A_x}-\mu_{A_y})^2 + 2\left(\sigma^2_{A_x}-\sigma^2_{A_y}\right)\log\left(\frac{\sigma_{A_x}}{\sigma_{A_y}}\right)}\right)}{\sigma^2_{A_x}-\sigma^2_{A_y}} \tag{4}$$

Given c, the overlap between the distributions is defined as the joint probability of A_x performing worse than c and A_y performing better than c:

$$p(\hat{m}(A_{x,i},i) \geq c) \cdot p(\hat{m}(A_{y,i},i) \leq c) = 1 - F_{A_{x,i},i}(c) + F_{A_{y,i},i}(c) \tag{5}$$

We define $p_\cap \in [0,1]$ as a threshold for the computed joint probability to include a given algorithm:

$$P_i = \{A \mid (p(\hat{m}(A_{1,i},i) \geq c) \cdot p(\hat{m}(A_{x,i},i) \leq c)) \geq p_\cap\} \tag{6}$$

$p_\cap$ is 1 for the best predicted algorithm, and 0 for algorithms whose distribution does not have any overlap with that of the best predicted algorithm, i.e. the probability of performing at least as good as the best predicted algorithm is 0.

We can control the size of the parallel portfolio by adjusting the value of $p_\cap$. If $p_\cap$ is set to 1, only the best predicted algorithm and ones that are predicted to perform exactly like it are included. On the other hand, if $p_\cap$ is set to 0, all algorithms will be included. This allows us to tune our approach to a given algorithm selection scenario and choose the algorithms to run in parallel very flexibly, also accommodating potentially inaccurate performance predictions.

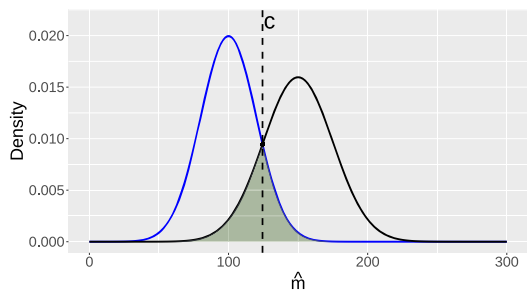

Figure 1. Overlapping area of two normal distributions. The point c is the performance both distributions are equally likely to achieve. The shaded area denotes the probability of overlap between the two distributions; in our case, the probability that the candidate solver will perform as least as well as the best predicted solver.

4 Experimental Setup

4.1 Data Collection

We used three scenarios from the ASlib benchmark repository [4]: MAXSAT19-UCMS, SAT11-INDU, and SAT18-EXP. Additionally, we created two new scenarios: SAT16-MAIN, which utilizes solvers and instances from the SAT Competition 2016, and IPC2018, which incorporates solvers and instances from the International Planning

Competition 2018. As ASlib only offers algorithm performance data for single runs, we conducted our own measurements for parallel runs on individual machines. We also measured the performance for single runs again and repeated the instance feature extraction steps to ensure that all experiments were performed on the same hardware. For MAXSAT19-UCMS, SAT11-INDU[1], SAT16-MAIN, and SAT18-EXP, we used SATZilla's feature computation code [33], and extracted 54 different features. For IPC2018 we used the feature extraction code by [6] which extracts 305 features for planning problems in PDDL format. We excluded 26 instances of SAT Competition 2016 from the SAT16-MAIN scenario because we were unable to extract features within two hours of computational time. We also omitted two solvers, glocusePLE and Scavel_SAT, from SAT16-MAIN because of frequent out-of-memory errors on multiple instances. From IPC2018, we omitted three solvers, MSP, maplan-1, and maplan-2, because they require an unavailable version of CPLEX. Table 4.1 gives an overview of the scenarios, algorithms, instances, and features we use in our evaluation.

Table 1. Number of algorithms, instances, and instance features for all scenarios.

Scenario	Algorithms	Instances	Instance Features
IPC2018	15	240	305
MAXSAT19-UCMS	7	572	54
SAT11-INDU	14	300	54
SAT16-MAIN	25	274	54
SAT18-EXP	37	353	54

We ran all solvers on all instances on compute nodes with 32 processors and 40 MB cache size (Intel(R) Xeon(R) CPU E5-2683 v4 @ 2.10GHz), 128 GB memory, and Red Hat Linux version 8.6. We use the same time limits as in the ASlib scenarios; 5000 CPU seconds for SAT18-EXP and SAT11-INDU, and 3600 CPU seconds for MAXSAT19-UCMS. For the new scenarios, we use the same time limits as the respective competitions; 5000 CPU seconds for SAT16-MAIN and 1800 CPU seconds for IPC2018. We ran each algorithm individually with 2-10 parallel runs. For all experiments, we ensured that only the given number of parallel runs were executed on a single machine. As our previous work showed that performance becomes worse than algorithm selection of a single solver for more than 10 parallel runs [13], we did not evaluate more than 10 parallel runs.

4.2 Training and Tuning

We built random forest regression models to predict the performance of an algorithm on an instance using LLAMA [16]. Random forests usually result in the best algorithm selection performance and performance predictions [4, 11]. Our setup mirrors that of [4]: we removed constant-valued instance features and imputed missing feature values with the mean of all non-missing values for that feature. The hyperparameters of the random forest models were tuned using random search with 250 iterations, with $ntree$ ranging from 10 to 200 and $mtry$ from 1 to 30 in a nested cross-validation with three inner folds and 10 outer folds [4].

Our regression random forest models predict the runtime for each solver as the mean of the underlying distribution, and estimate the standard deviation using the Jackknife method [31, 5], which calculates the standard deviation of the mean predictions over all observations used to train the random forest. The random forest is trained on $n-1$ observations and makes a prediction for the remaining observation. This process is repeated for all observations. The mean prediction for each tree is determined by averaging its predictions for the left-out observations. The Jackknife method assumes that the distribution of the predictions is normal, and their standard deviation is the uncertainty of the overall prediction.

To determine the optimal value of $p_\cap$ in Equation 5 for each scenario, we perform a grid search in the $[0, 1)$ interval with a resolution of 0.01 for a total of 100 values. Additionally, we determine the overall optimal value of $p_\cap$ across all five scenarios.

We evaluate the proposed approach using penalized average runtime with a factor of 10 (PAR10), misclassification penalty (MCP), and runtime. The PAR10 score is equal to the actual runtime when the algorithm succeeds in solving the instance within the timeout, otherwise, it is the timeout times 10. The misclassification penalty is the difference between the performance of the selected algorithm and the performance of the optimal algorithm.

4.3 Baselines

We compare the performance of our approach to several baseline methods, in particular the sequential virtual best solver (VBS), which is the optimal algorithm from the portfolio per problem instance (with a cumulative misclassification penalty of zero) and the sequential single best solver (SBS), which is the algorithm from the portfolio with the best average performance across all problem instances. The VBS for parallel runs is the best solver for each instance, but including the overhead for n parallel runs. The parallel SBS is computed similarly, with the best solvers on average instead of the best on each instance. We run multiple solvers in parallel to measure the actual runtime of the best solver in this case, rather than assuming the sequential runtime.

We further compare to per-instance algorithm selection that simply runs the top n predicted algorithms in parallel without considering the overlap of the distributions of the performance predictions, with the same performance models we use for our approach. In the notation we introduced above, we set $p_\cap = 0$ and cap the number of runs at the number of available processors. We use a simple scheduling method as a further baseline, where algorithms are scheduled according to their predicted rank and allocated a time slice equal to the predicted performance plus the standard deviation. This allows to run more than one algorithm per processor. This approach prioritizes the best-predicted algorithms but also potentially allows other algorithms to run.

ASPEED [8] provides a general schedule for all instances in a given scenario, rather than a schedule for each instance individually. Therefore, we do not include ASPEED in our experimental evaluation – static schedules across large sets of problem instances do not achieve competitive performance, as shown in [18]. The Flexfolio paper [18] shows experiments for Instance-Specific ASPEED and TSunny, but the available source code does not contain these algorithm selection methods and we are unable to compare to them.

Finally, we compare our approach to 3S as implemented in Flexfolio [18], as the original 3S implementation is unavailable. In this implementation, the number of neighbors for the kNN models was set to 32, and ASPEED [8] is used to schedule the chosen solvers instead of the original integer programming scheduler.

We normalize all performances across scenarios by the perfor-

[1] For SAT11-INDU, the ASlib benchmark repository contains 115 extracted features, including those from SATZilla. However, we were unable to find the feature extraction for this scenario and used the same 54 instance features extracted by SATZilla.

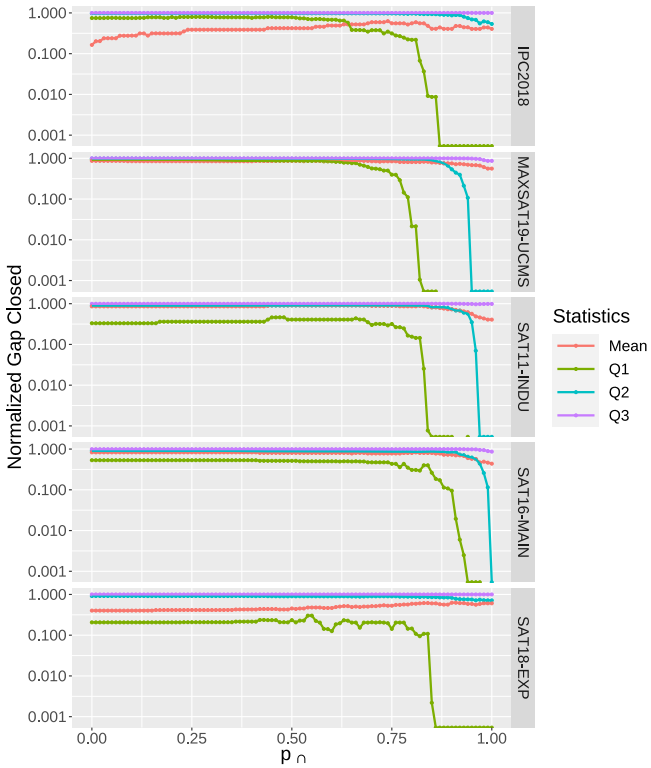

Figure 2. Sensitivity of portfolio performance to $p_\cap$. The plot illustrates the mean, Q1 (25th percentile), Q2 (50th percentile), and Q3 (75th percentile) runtime performance of each scenario for various values of $p_\cap$ as defined in Equation 6. Note the log scale for the normalized gap closed.

mances of the VBS and SBS and report the fraction of the gap between them that was closed by a particular approach. On this normalized scale, 0 corresponds to the performance of the SBS and 1 to the performance of the VBS. All code and data are available at https://github.com/uwyo-mallet/auto-parallel-portfolio-selection.

5 Results

5.1 *Tuning of $p_\cap$*

The tuning of $p_\cap$ shows that the optimal value depends on the scenario. For the IPC2018 scenario, the ideal $p_\cap$ value is 0.59, for the MAXSAT19-UCMS scenario 0.55, for SAT11-INDU 0.63, for SAT16-MAIN 0.33, and for SAT18-EXP 0.81. Figure 2 shows the normalized gap closed for the mean, 25th percentile, 50th percentile, and 75th percentiles for each scenario depending on $p_\cap$. While the optimal values are very different across different scenario, the differences in terms of gap closed are relatively small as long as $p_\cap$ is not too large. The best average value for $p_\cap$ across all scenarios is 0.82, which yields performance improvements over the baselines in most cases (see Table 2). For the overall best performance, we recommend to tune $p_\cap$ for the particular scenario, but using 0.82 is a reasonable starting point that gives good performance across the range of scenarios we consider here.

The optimal value of $p_\cap$ allows us to draw conclusions with respect to the predictive accuracy of the performance models we are using. A small value would suggest that the predictions of the performance models are not very accurate, as we have to include even solvers whose predicted runtime distribution has a small overlap with the runtime distribution of the best predicted solver to include solvers that are actually good. If the optimal value of $p_\cap$ was 0, we would have to include all solvers, even the ones whose predicted distribution has no overlap with the best predicted solver – in other words, the predicted runtime distribution of the actual best solver has no overlap with the predicted runtime distribution of the best predicted solver. Here, the optimal values for $p_\cap$ are relatively large in most cases, and even the smallest values are far greater than 0. This indicates that the predictions of the performance models are quite good – while the best predicted solver is not always the actual best solver for a given problem instance, the predicted runtime distribution of the actual best solver has a large overlap with the predicted runtime distribution of the predicted best solver.

5.2 *Algorithm Selection Results*

To evaluate the effectiveness of our approach, we carried out a series of experiments using the optimum and the average best value for $p_\cap$ for each scenario where we varied the number of processors used for parallel execution from one to ten for the SAT18-EXP, SAT16-MAIN, SAT11-INDU, and IPC2018 scenarios. For the MAXSAT19-UCMS scenario, we used a maximum of seven processors as there are only seven algorithms. Figure 3 shows the PAR10 score results in terms of the normalized performance gap between the sequential single best solver and sequential virtual best solver for all scenarios and numbers of processors.

The figure demonstrates the promise of the approach we propose here. In three out of five scenarios, we achieve the overall top performance with the maximum number of processors (even better than the parallel virtual best solver!) and for the remaining two scenarios only the parallel virtual best solver is better. We are able to achieve better performance than the parallel virtual best solver when running in parallel because our approach does not necessarily use all available processors, unlike the baseline approaches that we compare to. While the performance of the virtual best solver suffers for a large number of parallel runs, our approach keeps the overhead of running many things in parallel low and is thus better overall. We emphasize that the results we show here are actual measured values for running in parallel, rather than assuming overhead-free parallelization based on sequential runtimes, as is commonly done in the literature. Our results demonstrate that this common assumption is unrealistic except for a small number of parallel runs.

Even for a small number of processors, our approach yields better performance than others. Initially, the performance is similar to AS_0 (running the top n solvers), but our approach quickly becomes better as the number of available processors increases. This is expected, as for a single processor the two methods run exactly the same solver, but for a larger number of processors our method may not run as many as AS_0, thus decreasing overhead and overall solving performance.

For the IPC2018 scenario, we achieve the best overall results, improving performance substantially over all other approaches for 10 processors. The 3S approach is never close to the performance of our method and consistently yields worse results. The greedy time-splitting method also underperformed, often allocating time slices smaller than required to solve the instance and thus wasting resources. For more than seven parallel runs, the parallel virtual best solver, i.e. choosing the actual best solvers for each instance to run in parallel, starts to perform worse than our method, which does not use as many processors and incurs lower overhead.

The results for the other scenarios are qualitatively similar. While

Figure 3. Summary of results. The plot shows the degree to which the gap between the SBS and VBS PAR10 scores is closed by each method. For the VBS and SBS, we choose the top n solvers, where n is the number of processors, for a given problem instance and across all instances, respectively. AS_0 chooses the top n solvers predicted by algorithm selection, without regard for any overlap in their predicted runtime distributions. $AS_{p_\cap}$ represents the proposed formulation, with the number of processors restricted to at most the specific value indicated on the x axis – depending on the overlap of the predicted runtime distributions, fewer solvers than the maximum may be chosen. The $p_\cap$ values for IPC2018, MAXSAT19-UCMS, SAT11-INDU, SAT18-EXP, and SAT16-MAIN are 0.59, 0.55, 0.63, 0.81, and 0.33 and respectively. Time Splitting is the baseline approach that allocates time proportional to the predicted runtime and standard deviation for each solver, scheduling more than one solver to be run per processor.

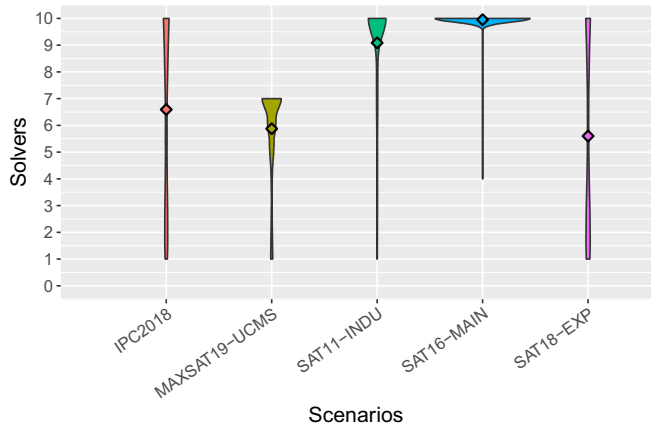

Figure 4. Violin plot of the distribution of the number of selected solvers to run in parallel across all problem instances for each scenario for the respective optimal $p_\cap$ and the maximum level of parallelism (seven processors for MAXSAT19-UCMS and 10 for all other scenarios). The diamond denotes the mean value.

for a small number of processors, other methods perform similar to ours, the gap between them widens as the number of parallel runs increases. 3S consistently shows worse performance, whereas running the top n solvers based on algorithm selection (without considering the predicted performance distributions) is usually competitive and in some cases gives the same performance as our method. The baseline of allocating a time to run proportional to the predicted runtime and standard deviation for each solver is not competitive, consistently showing bad performance – this baseline is worse than simply running the top n single best solvers in parallel on three scenarios for large numbers of parallel runs. For the IPC2018 and MAXSAT19-UCMS scenarios, the performance of some methods becomes worse than the single best solver for large numbers of processors, showing the limitations of these approaches. For the SAT2016-MAIN scenario, our approach is performing so close to the naïve parallel algorithm selection (top n solvers based on algorithm selection) because the standard error of the predictions was large and this resulted in large parallel portfolios for the majority of instances.

Table 2 shows more detailed results. We see that our method results in substantial savings in terms of all three measures across all scenarios – the proposed approach is always the best overall, regardless of the performance measure. Note that we never beat the sequential VBS, which represents the upper bound on the performance of any algorithm selection system – we cannot do better than only running the actual best solver. In many cases, the actual performance we achieve is close to the sequential VBS though. The results also show that using the "generic" best value for $p_\cap$ of 0.82 still gives substantial performance improvements over other approaches – usually it gives the second best performance. The only exception to this are the MAXSAT19-UCMS and SAT2016-MAIN scenarios, where running the top n solvers predicted by algorithm selection does better. The gap is relatively small though, and we still beat most of the other baselines.

5.3 *Number of Selected Solvers*

As mentioned above, allowing our approach to use up to a certain number of processors does not mean that this exact number of parallel runs will be done. In practice, it is often much lower than that,

Table 2. Detailed results. Mean and standard deviation of values for runtime, MCP, and PAR10 across all problem instances in a scenario for the sequential virtual best solver, sequential single best solver, and single top predicted algorithm in the initial three rows. The second set of rows for each scenario shows the results for the maximum number of processors (10 for SAT18-EXP, SAT16-MAIN, SAT11-INDU, and IPC2018, and 7 for MAXSAT19-UCMS) for our approach and the baselines we compare to. All numbers were rounded to integers. The best value for each scenario and measure is shown in **bold** (excepting the sequential VBS, which is by definition always the best), the second best in *italics*.

Scenario	Approach	Runtime [s]	MCP	PAR10
IPC2018	**1 Processor**			
	VBS	508±697	0	3478±6903
	AS	607±751	99±301	4657±7725
	SBS	734±770	226±414	5459±8072
	10 Processors			
	3S	645±770	137±471	5235±8047
	Time Splitting	637±797	129±348	5565±8241
	AS_0	612±779	104±307	5134±8027
	$AS_{p_\cap=0.59}$	**569±745**	**61±223**	*4484±7651*
	$AS_{p_\cap=0.82}$	*579±742*	*70±233*	**4359±7548**
MAXSAT19-UCMS	**1 Processor**			
	VBS	858±1476	0	7768±14717
	AS	1037±1555	179±641	9363±15684
	SBS	1190±1657	332±940	11386±16696
	7 Processors			
	3S	953±1480	95±437	8317±15031
	Time Splitting	908±1523	51±308	8668±15353
	AS_0	*894±1506*	*37±247*	*8258±15062*
	$AS_{p_\cap=0.55}$	**891±1496**	**33±215**	**8141±14975**
	$AS_{p_\cap=0.82}$	928±1513	70±364	8461±15175
SAT11-INDU	**1 Processor**			
	VBS	1140±1836	0	8040±17905
	AS	1535±2058	395±1037	11735±20768
	SBS	1818±2168	678±1340	14268±22154
	10 Processors			
	3S	1298±1898	158±546	9098±18780
	Time Splitting	1335±2009	225±708	10635±20138
	AS_0	1272±1927	161±548	8922±18645
	$AS_{p_\cap=0.63}$	**1241±1901**	*131±451*	**8591±18349**
	$AS_{p_\cap=0.82}$	*1247±1900*	**123±431**	*8747±18501*
SAT16-MAIN	**1 Processor**			
	VBS	1867±2193	0	15005±22530
	AS	2315±2273	448±1109	19066±23883
	SBS	2560±2294	693±1415	21940±24464
	10 Processors			
	3S	2093±2228	226±547	16874±23228
	Time Splitting	2101±2247	234±732	16717±23149
	AS_0	*2065±2221*	*198±652*	*16189±22931*
	$AS_{p_\cap=0.33}$	**2065±2221**	**198±652**	**16189±22931**
	$AS_{p_\cap=0.82}$	2094±2222	228±730	16383±22993
SAT18-EXP	**1 Processor**			
	VBS	1146±1945	0	9687±19547
	AS	1615±2138	468±1192	13470±21889
	SBS	2400±2249	1254±1832	20629±24280
	10 Processors			
	3S	1625±2228	479±1265	15010±22802
	Time Splitting	1714±2292	571±1384	15992±23222
	AS_0	1702±2301	559±1389	16235±23355
	$AS_{p_\cap=0.81}$	**1518±2172**	**372±1124**	**13884±22265**
	$AS_{p_\cap=0.82}$	*1532±2178*	*386±1146*	*14025±22336*

as we see when comparing the performance of our approach to AS_0, which runs the top n predicted solvers in parallel. Figure 4 shows the distribution of the number of selected solvers for each scenario. The mean number of solvers chosen for IPC2018 is around 6.5, for MAXSAT19-UCMS around 6 (out of 7), for SAT11-INDU around 9, for SAT16-MAIN around 10, and for SAT18-EXP around 5.5. We see that the largest difference to the maximum number of parallel runs occurs for the two scenarios where we observe the largest performance improvements of our approach, IPC2018 and SAT18-EXP. Similarly, the scenario with the highest number of solvers chosen on average (SAT16-MAIN) is where we see the smallest performance improvement. This clearly shows again that the advantage of our approach is that it does not simply use as many parallel processors as are available, which increases overhead, but intelligently chooses how many of the available processors to use for best performance. In at least some cases, more is less, and we show how to leverage this.

Figure 4 also shows that our approach uses the full range of available parallel runs in most cases, from running only a single solver to as many parallel runs as there are processors. Our approach is not simply a one-size-fits all that usually uses a similar number of runs, but varies the size of the selected parallel portfolio dynamically, based on the instance to be solved.

6 Conclusions and Future Work

In this study, we proposed a general method for selecting solvers from a portfolio of solvers and scheduling them in parallel, taking into account the predicted runtime distribution to intelligently choose not only which solvers to run, but also how many. This is in contrast to most other approaches in the literature, which either choose a constant number or use all available processors. Further, we measured the actual runtime when running more than one algorithm in parallel, rather than assuming the sequential runtime. We demonstrated substantial performance improvements across a wide range of scenarios, handily beating baseline methods and other approaches from the literature. The proposed method establishes a new state of the art in parallel algorithm selection and is simple to apply in practice – we are only using information that is readily available in common algorithm selection methods, and while for the best performance the parameter $p_\cap$ of our method should be tuned, a reasonable default already shows good performance. This parameter allows our method to be tailored to specific application domains and scenarios.

While we do show substantial performance improvements, there is room for further advances. We have focused our investigation on state-of-the-art random forest performance models and the jackknife method for estimating uncertainties, but other methods exist. It is possible that other types of models may perform better in this context if the uncertainty estimates of their predictions are better, for example for Gaussian Processes. It is also possible to combine different types of performance models for different algorithms, allowing much more flexibility and potentially greater performance improvements. While our baseline method that allocates resources to each algorithm did not perform well, investigating more sophisticated approaches for this would also be interesting.

Acknowledgements

This work was supported in part by NSF grant 1813537 and the School of Computing at the University of Wyoming. We thank Damir Pulatov for helping collect the solvers.

References

[1] Martin Aigner, Armin Biere, Christoph M Kirsch, Aina Niemetz, and Mathias Preiner, 'Analysis of portfolio-style parallel sat solving on current multi-core architectures', *POS@ SAT*, **29**, 28–40, (2013).

[2] Roberto Amadini, Maurizio Gabbrielli, and Jacopo Mauro, 'SUNNY: a lazy portfolio approach for constraint solving', *Theory Pract. Log. Program.*, **14**(4-5), 509–524, (2014).

[3] Roberto Amadini, Maurizio Gabbrielli, and Jacopo Mauro, 'A multicore tool for constraint solving', in *Proceedings of the 24th International Conference on Artificial Intelligence*, p. 232–238, (2015).

[4] Bernd Bischl, Pascal Kerschke, Lars Kotthoff, Marius Lindauer, Yuri Malitsky, Alexandre Fréchette, Holger Hoos, Frank Hutter, Kevin Leyton-Brown, Kevin Tierney, and Joaquin Vanschoren, 'Aslib: A benchmark library for algorithm selection', *Artificial Intelligence*, **237**, 41 – 58, (2016).

[5] Bernd Bischl, Michel Lang, Lars Kotthoff, Julia Schiffner, Jakob Richter, Erich Studerus, Giuseppe Casalicchio, and Zachary M. Jones, 'mlr: Machine learning in r', *Journal of Machine Learning Research*, **17**(170), 1–5, (2016).

[6] Chris Fawcett, Mauro Vallati, Frank Hutter, Jörg Hoffmann, Holger Hoos, and Kevin Leyton-Brown, 'Improved features for runtime prediction of domain-independent planners', *Proceedings of the International Conference on Automated Planning and Scheduling*, **24**(1), 355–359, (2014).

[7] Carla Gomes and Bart Selman, 'Algorithm portfolios', *Artificial Intelligence*, **126**, 43–62, (2001).

[8] Holger H. Hoos, Roland Kaminski, Marius Thomas Lindauer, and Torsten Schaub, 'aspeed: Solver scheduling via answer set programming', *TPLP*, **15**(1), 117–142, (2015).

[9] Mengqi Hu, Teresa Wu, and Jeffery D. Weir, 'An intelligent augmentation of particle swarm optimization with multiple adaptive methods', *Information Sciences*, **213**, 68–83, (2012).

[10] Bernardo A. Huberman, Rajan M. Lukose, and Tad Hogg, 'An economics approach to hard computational problems', *Science*, **275**(5296), 51–54, (1997).

[11] Frank Hutter, Lin Xu, Holger H. Hoos, and Kevin Leyton-Brown, 'Algorithm runtime prediction: Methods & evaluation', *Artificial Intelligence*, **206**, 79–111, (2014).

[12] Serdar Kadioglu, Yuri Malitsky, Ashish Sabharwal, Horst Samulowitz, and Meinolf Sellmann, 'Algorithm selection and scheduling', in *17th International Conference on Principles and Practice of Constraint Programming*, pp. 454–469. Springer, (2011).

[13] Haniye Kashgarani and Lars Kotthoff, 'Is algorithm selection worth it? comparing selecting single algorithms and parallel execution', in *AAAI Workshop on Meta-Learning and MetaDL Challenge*, volume 140 of *Proceedings of Machine Learning Research*, pp. 58–64. PMLR, (2021).

[14] Pascal Kerschke, Holger H. Hoos, Frank Neumann, and Heike Trautmann, 'Automated Algorithm Selection: Survey and Perspectives', *Evolutionary Computation*, **27**(1), 3–45, (2019).

[15] Pascal Kerschke, Lars Kotthoff, Jakob Bossek, Holger H. Hoos, and Heike Trautmann, 'Leveraging TSP Solver Complementarity through Machine Learning', *Evolutionary Computation*, **26**(4), 597–620, (2018).

[16] Lars Kotthoff, 'LLAMA: leveraging learning to automatically manage algorithms', *CoRR*, **abs/1306.1031**, (2013).

[17] Lars Kotthoff, 'Algorithm selection for combinatorial search problems: A survey', *AI Magazine*, **35**(3), 48–69, (2014).

[18] Marius Lindauer, Rolf-David Bergdoll, and Frank Hutter, 'An empirical study of per-instance algorithm scheduling', in *Proceedings of the Tenth International Conference on Learning and Intelligent Optimization, LION'16, in: Lecture Notes in Computer Science*, pp. 253–259. Springer, Springer, (2016).

[19] Marius Lindauer, Holger Hoos, and Frank Hutter, 'From sequential algorithm selection to parallel portfolio selection', in *International Conference on Learning and Intelligent Optimization*, pp. 1–16. Springer, (2015).

[20] Marius Lindauer, Holger Hoos, Kevin Leyton-Brown, and Torsten Schaub, 'Automatic construction of parallel portfolios via algorithm configuration', *Artificial Intelligence*, **244**, 272–290, (2017).

[21] Marius Lindauer, Holger H Hoos, Frank Hutter, and Torsten Schaub, 'Autofolio: An automatically configured algorithm selector', *Journal of Artificial Intelligence Research*, **53**, 745–778, (2015).

[22] Marius Lindauer, Jan N. van Rijn, and Lars Kotthoff, 'Open algorithm selection challenge 2017: Setup and scenarios', in *Proceedings of the Open Algorithm Selection Challenge*, volume 79, pp. 1–7. PMLR, (2017).

[23] Yuri Malitsky, Ashish Sabharwal, Horst Samulowitz, and Meinolf Sellmann, 'Parallel sat solver selection and scheduling', in *Proceedings of the 18th International Conference on Principles and Practice of Constraint Programming - Volume 7514*, p. 512–526. Springer-Verlag, (2012).

[24] Marco Maratea, Luca Pulina, and Francesco Ricca, 'A multi-engine approach to answer-set programming', *Theory and Practice of Logic Programming*, **14**(6), 841–868, (2014).

[25] Jorge Maturana, Álvaro Fialho, Frédéric Saubion, Marc Schoenauer, Frédéric Lardeux, and Michèle Sebag, *Adaptive Operator Selection and Management in Evolutionary Algorithms*, 161–189, Springer, 2012.

[26] Eoin O'Mahony, Emmanuel Hebrard, Alan Holland, Conor Nugent, and Barry O'Sullivan, 'Using case-based reasoning in an algorithm portfolio for constraint solving', in *Irish conference on artificial intelligence and cognitive science*, pp. 210–216. Proceedings of the 19th Irish Conference on Artificial Intelligence and Cognitive Science, (2008).

[27] Damir Pulatov, Marie Anastacio, Lars Kotthoff, and Holger Hoos, 'Opening the black box: Automated software analysis for algorithm selection', in *Proceedings of the First International Conference on Automated Machine Learning*, volume 188, pp. 6/1–18. PMLR, (2022).

[28] John R Rice, 'The algorithm selection problem', in *Advances in computers*, volume 15, 65–118, Elsevier, (1976).

[29] Olivier Roussel, 'Description of ppfolio (2011)', *Proc. SAT Challenge*, 46, (2012).

[30] Oğuz Torağay and Shaheen Pouya, 'A monte carlo simulation approach to the gap-time relationship in solving scheduling problem', *Journal of Turkish Operations Management*, **7**(1), 1579 – 1590, (2023).

[31] Stefan Wager, Trevor Hastie, and Bradley Efron, 'Confidence intervals for random forests: The jackknife and the infinitesimal jackknife', *The Journal of Machine Learning Research*, **15**(1), 1625–1651, (2014).

[32] Lin Xu, Holger H. Hoos, and Kevin Leyton-Brown, 'Hydra: Automatically configuring algorithms for portfolio-based selection', in *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence*, number 1 in AAAI'10, p. 210–216. AAAI Press, (2010).

[33] Lin Xu, Frank Hutter, Holger Hoos, and Kevin Leyton-Brown, 'Satzilla: Portfolio-based algorithm selection for sat', *Journal of Artificial Intelligence Research*, **32**, 565–606, (2008).

[34] Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown, 'Satzilla: Portfolio-based algorithm selection for sat', *J. Artif. Int. Res.*, **32**(1), 565–606, (2008).

[35] Lin Xu, Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown, 'Hydra–mip: Automated algorithm configuration and selection for mixed integer programming', in *Proceedings of the 18th RCRA workshop*, pp. 16–30, (2011).

[36] Shiu Yin Yuen, Chi Kin Chow, and Xin Zhang, 'Which algorithm should i choose at any point of the search: An evolutionary portfolio approach', in *GECCO 2013 - Proceedings of the 2013 Genetic and Evolutionary Computation Conference*, pp. 567–574, (2013).

[37] Shiu Yin Yuen, Yang Lou, and Xin Zhang, 'Selecting evolutionary algorithms for black box design optimization problems', *Soft Computing*, **23**(15), 6511–6531, (2019).

[38] Xi Yun and Susan L. Epstein, 'Learning algorithm portfolios for parallel execution', in *Revised Selected Papers of the 6th International Conference on Learning and Intelligent Optimization - Volume 7219*, LION 6, p. 323–338. Springer-Verlag, (2012).