Chapter 8
Selection and Configuration of Parallel Portfolios

Marius Lindauer¹, Holger Hoos², Frank Hutter¹, and Kevin Leyton-Brown³

Abstract In recent years the availability of parallel computation resources has grown rapidly. Nevertheless, even for the most widely studied constraint programming problems such as SAT, solver development and applications remain largely focussed on sequential rather than parallel approaches. To ease the burden usually associated with designing, implementing and testing parallel solvers, in this chapter, we demonstrate how methods from automatic algorithm design can be used to construct effective parallel portfolio solvers from sequential components. Specifically, we discuss two prominent approaches for this problem. (I) Parallel portfolio selection involves selecting a parallel portfolio consisting of complementary sequential solvers for a specific instance to be solved (as characterised by cheaply computable instance features). Applied to a broad set of sequential SAT solvers from SAT competitions, we show that our generic approach achieves nearly linear speed-up on application instances, and super-linear speed-ups on combinatorial and random instances. (II) Automatic construction of parallel portfolios via algorithm configuration involves a parallel portfolio of algorithm parameter configurations that is optimized for a given set of instances. Applied to gold medal-winning parameterized SAT solvers, we show that our approach can produce significantly better-performing SAT solvers than state-of-the-art parallel solvers constructed by human experts, reducing time-outs by 17% and running time (PAR10 score) by 13% under competition conditions.

Marius Lindauer
University of Freiburg, Germany, e-mail: lindauer@cs.uni-freiburg.de
Holger Hoos
University of British Columbia, Canada & Leiden University, Netherlands, e-mail: hoos@cs.ubc.ca
Frank Hutter
University of Freiburg, Germany, e-mail: fh@cs.uni-freiburg.de
Kevin Leyton-Brown
University of British Columbia, Canada, e-mail: kevinlb@cs.ubc.ca
8.1 Introduction

Given the prevalence of multicore processors and the ready availability of large compute clusters (e.g., in the cloud), parallel computation continues to grow in importance. This is particularly true in the vibrant area of propositional satisfiability (SAT), where over the last decade, parallel solvers have received increasing attention and shown impressive performance in the influential SAT competitions. Nevertheless, development and research efforts remain largely focused on sequential rather than parallel designs; for example, 29 sequential solvers participated in the main track of the 2016 SAT Competition, compared to 14 parallel solvers.

One key reason for this focus on sequential solvers lies in the complexity of designing, implementing and testing effective parallel solvers. This involves a host of challenges, including coordination between threads or processes, efficient communication strategies for information sharing, and non-determinism due to asynchronous computation. As a result, it is typically difficult to effectively parallelise a sequential solver; in most cases, fundamental redesign is required to harness the power of parallel computation. Methods that can produce effective parallel solvers from one or more sequential solvers automatically (or with minimal human effort) are therefore very attractive, even if they cannot generally be expected to reach the performance levels of a carefully hand-crafted parallel solver design. In this chapter, we give an overview of several such automatic approaches. We illustrate these for SAT solvers, in part because SAT is one of the most widely studied NP-hard problems, but also because these approaches, although not limited to SAT solving, were first developed in this context.

One of the simplest automatic methods for constructing a parallel solver is to run multiple sequential solvers independently in parallel on the same input; this is called a parallel algorithm portfolio. For SAT, this approach has been applied with considerable success. A well-known example is pfolio [70], which, despite the simplicity of the approach, won several categories of the 2011 SAT Competition; pfolio runs several sequential SAT solvers (including CryptoMiniSat [74], Lingeling [13], clasp [24], TNM [54], and march_hi [33]) as well as one parallel solver (Plingeling [13]) in parallel, without any communication between the solvers, except that all portfolio components are terminated as soon as the first solves the given SAT instance. This works well when the component solvers have complementary strengths. For example, CryptoMiniSat and Lingeling perform well on application instances, clasp excels on crafted instances, and TNM and march_hi are particularly effective on randomly generated SAT instances. The pfolio portfolio was constructed manually by experts with deep insights into the performance characteristics of SAT solvers, drawing from a large set of sequential SAT solvers and using limited computational experiments to assemble hand-picked components into an effective parallel portfolio.
In the following, we focus on generic methods that automate the construction of
effective parallel solvers from given sequential components. Such methods can be
seen as instances of \textit{programming by optimization} [35] and \textit{search-based software
engineering} [30]. There are several advantages to using automatic methods for
parallel solver construction: substantially reduced need for rare and costly human
expertise; easier exploitation of new component solvers; and easier adaptation to
different sets or distributions of problem instances. Broadly speaking, there are two
automatic methods for parallel solver construction:

\begin{itemize}
\item \textbf{Parallel Portfolio Selection.} Parallel portfolio selection focuses on combining
a set of algorithms by means of per-instance algorithm selection or algorithm
schedules. In per-instance algorithm selection [69, 39, 51], we select one solver
from a given set based on features of that instance, with the goal of optimizing
performance on the given instance. Per-instance selection can be generalised to
produce a parallel portfolio of solvers rather than a single solver [56]; this portfolio
consists of sequential solvers that run concurrently on a given problem instance.
Algorithm schedules exploit solver complementarity through a sequence of runs
with associated time budgets. This strategy can be parallelised by concurrently
running multiple sequential schedules, each on a separate processing unit [36].
\item \textbf{Automatic Construction of Parallel Portfolios (ACPP).} In algorithm configura-
tion [45], the goal is to set the parameters of a given algorithm (e.g., a SAT solver)
to optimise performance for a given set or distribution of problem instances. Auto-
matic configuration can also be used to determine a set of configurations [79, 50]
that jointly perform well when combined into a parallel portfolio [58].
\end{itemize}

These two approaches address orthogonal problems: the former allows us to
effectively use an existing set of solvers for each \textit{instance}, while the latter builds an
effective portfolio for a given \textit{instance set} from complementary components drawn
from a large (often infinite) configuration space of solvers.

Both of these approaches are based on the assumption that different solvers or
solver configurations exhibit sufficient \textit{performance complementarity}: i.e., they differ
substantially in efficacy relative to each other depending on the problem instance to
be solved. Solver complementarity is known to exist for many NP-hard problems—
notably SAT, where it has been studied by Xu et al. [82]—and is also reflected in the
excellent performance of many portfolio-based solvers [28, 70, 25, 15, 6]. While in
the following we focus on SAT, solver complementarity has also been demonstrated
and exploited for a broad range of other problems, including MAXSAT [4], quantified
Boolean formulas [68, 53], answer set programming [64], constraint satisfaction [67],
AI planning [31, 73], and mixed integer programming [41, 81]; we thus expect that
the techniques we describe could successfully be applied to these problems.

This chapter is organized as follows. We discuss parallel portfolio selection in
Section 8.2 and automatic construction of parallel portfolios from parameterized
solvers in Section 8.3. We conclude the chapter by discussing limitations as well

\footnote{We note that parallel resources can also be used for parallel algorithm configuration [44]; while this
is an important area of study, in this chapter, we focus on methods that produce parallel portfolios as an output.}
as possible extensions and combinations of the two approaches (Section 8.4). The material in this chapter builds on and extends previously published work on parallel portfolio selection [56] and automatic construction of parallel portfolios [58].

8.2 Per-Instance Selection of Parallel Portfolios

Well-known per-instance algorithm selection systems for SAT include SATzilla [65, 80, 83], 3S [49], CSHC [62], and AutoFolio [57]. The algorithm portfolios such systems construct have been very successful in past SAT competitions, regularly outperforming the best non-portfolio solvers. Algorithm selection systems perform particularly well on heterogeneous instance sets, for which no single solver (or parameter configuration of a solver) performs well overall [71]. For example, the instance sets used in SAT competitions include problems from packing, argumentation, cryptography, hardware verification, planning, scheduling, and software verification [12].

In parallel portfolio selection, we select a set of algorithms to run together in a parallel portfolio. This offers robustness against errors in solver selection, can reduce dependence on instance features, and provides a simple yet effective way of exploiting parallel computational resources.

8.2.1 Problem Statement

Formally, the algorithm selection problem is defined as follows:

**Definition 1 (Sequential Algorithm Selection).** An instance of the *per-instance algorithm selection problem* is a 4-tuple \( (I, \mathcal{D}, \mathcal{A}, m) \), where

- \( I \) is a set of instances of a problem,
- \( \mathcal{D} \) is a probability distribution over \( I \),
- \( \mathcal{A} \) is a set of algorithms for solving instances in \( I \), and
- \( m : \mathcal{A} \times I \rightarrow \mathbb{R} \) quantifies the performance of algorithm \( A \in \mathcal{A} \) on instance \( \pi \in I \).

The objective is to construct an *algorithm selector*, i.e., a mapping \( \phi : I \rightarrow \mathcal{A} \), such that the expected performance measure \( \mathbb{E}_{\pi \sim \mathcal{D}} [m(\phi(\pi), \pi)] \) across all instances is optimised. In this chapter, we will consider a performance measure based on minimizing running time.

The mapping \( \phi \) is computed by extracting features \( f(\pi) \in F \) from a given instance \( \pi \) that are subsequently used to determine the algorithm to be selected [66, 80, 48];

\[ \text{2 New SAT competition rules limit portfolio systems to two SAT solving engines. Nevertheless, algorithm selection systems have remained quite successful; e.g., Riss BlackBox [3] won 3 medals in 2014.} \]
a mapping from this feature space $F$ to algorithms is typically constructed using machine learning techniques. Instance features for algorithm selection must be cheap to compute (normally costing at most a few seconds) to avoid taking too much time away from actually solving the instance. Some important examples include:

- **Size features**, such as the number of variables and clauses, or their ratio [20];
- **CNF graph features** based on the variable-clause graph, variable graph [32], or clause graph;
- **Balance features**, such as the fraction of unary, binary or ternary clauses [66, 80];
- **Proximity to Horn formula features**, such as statistics on horn clauses [66];
- **Survey propagation features**, which estimate variable bias with probabilistic inference [38];
- **Probing features**, which are computed by running, e.g., DPLL solvers, stochastic local search solvers, LP solvers or CDCL solvers for a short amount of time to obtain insights in their solving behavior [66], such as number of unit propagations at a given search tree depth;
- **Timing features**, the time required to compute other features [48].

For a full list of currently used SAT features, we refer the interested reader to Hutter et al. [48] and to Alfonso et al. [2].

Some performance metrics based on running time penalize solvers for spending seconds to solve instances that can be solved in milliseconds. (A complex performance metric of this type has been used in some past SAT competitions.) In such cases, evaluating features for every instance can lead to unacceptable penalties. Such penalties can be mitigated via static presolving schedules [80, 49, 37]. Based on the observation that many solvers either solve a given instance quickly or not at all, a presolving schedule runs a sequence of complementary solvers, each for a small fraction of the overall running time cutoff. If the given instance is solved in any of these runs, the remainder of the presolving and algorithm selection workflow is skipped. Furthermore, the presolving schedule is static, meaning that it does not vary between instances. Beyond saving the time to compute features, static presolving schedules also have another benefit: by running more than the finally-selected algorithm, to some degree we hedge against suboptimal selection outcomes based on instance features.

Parallel portfolio selection takes this idea further, selecting a whole set of solvers to run in parallel. Thus, instead of learning a single mapping $\phi : I \to A$ to select a
solver, we learn a mapping \( \phi_k : I \rightarrow A^k \) to select a portfolio with \( k \) components for a given number of processing units \( k \).

Formally, the parallel portfolio selection problem [56] is defined as follows.

**Definition 2 (Parallel Portfolio Selection).** An instance of the per-instance parallel portfolio selection problem is a 5-tuple \( \langle I, \mathcal{D}, \mathcal{A}, m, k \rangle \), where

- \( I \) is a set of instances of a problem,
- \( \mathcal{D} \) is a probability distribution over \( I \),
- \( \mathcal{A} \) is a set of algorithms for instances in \( I \),
- \( k \) is the number of available processing units, and
- \( m : \mathcal{A} \times I \times I \rightarrow \mathbb{R} \) quantifies the performance of a portfolio \( A_1 \) on an instance \( \pi \in I \) for any given portfolio size \( l \).

The objective is to construct a parallel portfolio selector, i.e., a mapping \( \phi_k : I \rightarrow A^k \), such that the expected performance measure \( \mathbb{E}_{\pi \sim \mathcal{D}}[m(\phi_k(\pi), \pi)] \) across all instances is optimised. If the concurrently-running algorithms in the selected portfolio neither interact nor communicate, the objective can be written as \( \mathbb{E}_{\pi \sim \mathcal{D}}[\min_{A^*} m(A^*, \pi)] \).

As in the case of selecting a single solver, we can extend parallel portfolio selection to include a static presolving schedule. Figure 8.2 shows the workflow of a parallel portfolio selection procedure. First, we run a parallel presolving schedule on all processing units. Since feature computation is currently still a sequential process, we run a short presolving schedule on the first unit and then start feature computation if necessary. On all other units, we presolve until feature computation finishes. We then select an algorithm for each processing unit.

### 8.2.2 Parallelization of Sequential Algorithm Selectors

We now discuss a general strategy for parallelizing sequential algorithm selection methods. This approach is motivated by the availability of a broad range of effective sequential selection approaches that use an underlying scoring function \( s : \mathcal{A} \times I \rightarrow \mathbb{R} \).
8.2.2.1 Performance-based Nearest Neighbor (PNN)

The algorithm selection approach in 3S [62] in its simplest form uses a $k$-nearest neighbour approach. For a new instance $\pi$ with features $f(\pi)$, it finds the $k$ nearest training instances $I_k(\pi)$ in the feature space $F$ and selects the algorithm that has the best training performance on them. Formally, given a performance metric $m : \mathcal{A} \times I \rightarrow \mathbb{R}$, we define $m_k(A, \pi) = \sum_{\pi' \in I_k(\pi)} m(A, \pi')$ and select algorithm $\arg\min_{A \in \mathcal{A}} m_k(A, \pi)$.

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

$$s_{PNN}(A, \pi) = m_k(A, \pi). \quad (8.1)$$

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

8.2.2.2 Distance-based Nearest Neighbor (DNN)

ME-ASP [64] 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 in experiments by the authors of ME-ASP [64]. At training time, this approach memorizes the best algorithm $A^*(\pi')$ on each training instance $\pi' \in I$. For a new instance $\pi$, it finds the nearest training instance $\pi'$ in the feature space and selects the algorithm $A^*(\pi')$ associated with that instance.

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

$$s_{DNN}(A, \pi) = \min\{d(f(\pi), f(\pi')) \mid \pi' \in I \land A^*(\pi') = A\}. \quad (8.2)$$
If \( \pi' \in I \) such that \( A'(\pi') = A \) is empty (because algorithm \( A \) was never the best algorithm on an instance) then \( s_{DNN}(A, \pi) = \infty \) for all instances \( \pi \). 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, \( O(|I| \cdot \#f) \), where \( \#f \) is the number of features.

### 8.2.2.3 Clustering

The selection part of ISAC [50]\(^3\) 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 \( \pi \), we score each algorithm \( A \) by the minimum of the distances between \( \pi \) and any cluster associated with \( A \). Formally, using \( d(f(\pi), z) \) to denote the distance in feature space between instance \( \pi \) and cluster center \( z \), we have the following scoring function:

\[
s_{Clu}(A, \pi) = \min \{d(f(\pi), z) \mid z \in Z \land \hat{A}(z) = A \}.
\]

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 \( s_{Clu}(A, \pi) = \infty \) for all instances \( \pi \). The time complexity is as for DNN, replacing the number of training instances \( |I| \) with the number of clusters \( |Z| \).

### 8.2.2.4 Regression

The first version of SATzilla [65] used a regression approach, which, for each \( A \in \mathcal{A} \), learns a regression model \( r_A : F \rightarrow \mathbb{R} \) to predict performance on new instances. For a new instance \( \pi \) with features \( f(\pi) \), it selected the algorithm with the best predicted performance, i.e., \( \arg\min_{A \in \mathcal{A}} r_A(f(\pi)) \).

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

\[
s_{Reg}(A, \pi) = r_A(f(\pi))
\]

(8.4) to select the \( A \) algorithms predicted to perform best. The complexity of model evaluations differs across models, but it is 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 \( O(P_{reg} \cdot |\mathcal{A}|) \).

\(^3\) In its original version, ISAC is a combination of algorithm configuration and selection, but only the selection approach was used in later publications.
8 Selection and Configuration of Parallel Portfolios

8.2.2.5 Pairwise Voting

The most recent SATzilla version \[82\] uses cost-sensitive random forest classification to learn for each pair of algorithms \( A_1 \neq A_2 \in \mathcal{A} \) 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 \( v(A,\pi) = \sum_{A' \in \mathcal{A} \setminus \{A\}} c_{A,A'}(f(\pi')) \) denote the sum of votes algorithm \( A \) receives for instance \( \pi \); then, SATzilla selects \( \arg\max_{A \in \mathcal{A}} v(A,\pi) \).

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:

\[
s_{\text{Vote}}(A,\pi) = -v(A,\pi). \tag{8.5}
\]

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, in particular random forests; we denote this polynomial function by \( P_{\text{class}} \). Since we need to evaluate pairwise classifiers for all algorithm pairs, the time complexity to select a parallel portfolio is then \( O(P_{\text{class}} \cdot |\mathcal{A}|^2) \).

8.2.3 Parallel presolving Schedules

As mentioned previously, our approach for parallel portfolios does not only consider parallel portfolios selected on a per-instance basis, but also uses parallel presolving schedules (see Figure 8.2). Fortunately, Hoos et al. \[36\] already proposed an effective system to compute a static parallel schedule for a given set of instances, called \text{aspeed}. This system is based on an answer set programming (ASP) encoding of the NP-hard problem of algorithm scheduling, and we only need to add one further constraint in this encoding to shorten the schedule in the first processing unit to allow for feature computation. We approximate the required time for feature computation by the allowed upper bound.

Computationally, it is not a problem that finding the optimal algorithm schedule is NP-hard, since this step is performed offline during training and not online in the solving process. Furthermore, the empirical results of Hoos et al. \[36\] indicated that the problem of optimizing parallel schedules gets easier with more processing units such that it also scales well with an increasing number of processing units.

8.2.4 Empirical Study on SAT Benchmarks

To study the performance of our selected parallel portfolios, we show results on the SAT scenarios of the algorithm selection library (ASlib \[17\]). ASlib scenarios define a cross validation split scheme, i.e., the instances are split into 10 equally sized
In particular, we study the performance of parallel portfolio selection systems on two different SAT scenarios. Similar to the SAT competitions, each scenario is divided into application, crafted (a.k.a. handmade or hard combinatorial) and random tracks:

1. **SAT11*. The SAT11 scenarios consider the SAT solvers, instances and measured runtimes from the SAT Competition 2011. As features, we used the features from the SATzilla [83] feature generator.

2. **SAT12*. The SAT12 scenarios include all instances used in competitions prior to the SAT Competition 2012; the solvers are from all tracks of the previous SAT Competition 2011. The instance features are the same as in the SAT11 scenarios. The data was used to train SATzilla [83] for the SAT Competition 2012.

Table 8.1 shows the details of the used scenarios. The main differences are that the SAT11 scenarios have fewer instances and fewer algorithms with a larger running time budget in comparison to the SAT12 scenarios. Comparing the different tracks, the time to compute the instance features is largest for industrial instances, followed by crafted instances and random instances. However, in our experiments we use only the 54 “base” features that do not include any probing features and are much cheaper to compute.

Table 8.2 shows the speedup of our parallel portfolio selection approaches based on PAR10 scores\(^4\) depending on the number of processing units \(k\). Since all approaches have different sequential performance, we use the performance of the sequential single best solver (SB, i.e., the solver with the best performance across all training instances) as the baseline for the speedup computation; for example, a speedup of 1.0 corresponds to the same performance as the SB. We applied a paired statistical test (i.e., a permutation test) with significance level \(\alpha = 0.05\) to mark statistically

\(^4\) PAR10 [45] is the penalized average running time, counting each timeout as 10 times the running time cutoff.
VBS depends on the complementarity of the component solvers. The set of all SAT random and crafted instances) and there was no clear winner amongst the different cores. Of 65 algorithms, the maximal speedup that can be achieved differs between the scenarios. Complementarity is not always the same across different instance sets and available solvers in a SAT competition tends to be quite complementary [82], but since this each instance, or running all available solvers in parallel). The performance of the

indistinguishable performance from the best performing system for each number of processing units. We note that algorithm selection \((k = 1)\) already performs better than the SB in all settings except DNN on SAT12-RAND.

Since we do not consider clause sharing in our experiments, the maximal possible speedup is limited by the virtual best solver (VBS, i.e., running the best solver for each instance, or running all available solvers in parallel). The performance of the VBS depends on the complementarity of the component solvers. The set of all SAT solvers in a SAT competition tends to be quite complementary [82], but since this complementarity is not always the same across different instance sets and available algorithms, the maximal speedup that can be achieved differs between the scenarios. The extremes in our experiments were SAT11-RAND with a maximal speedup factor of 65.7 using 8 cores and SAT12-RAND with “only” a speedup factor of 12.1 using 8 cores.

Overall, the speedups were quite large (sometimes superlinear, particularly for the random and crafted instances) and there was no clear winner amongst the different

<table>
<thead>
<tr>
<th>(k)</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAT11-INDU (VBS: 21.4)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PNN</td>
<td>1.1</td>
<td>1.5</td>
<td>2.6</td>
<td>5.2</td>
</tr>
<tr>
<td>DNN</td>
<td>1.4</td>
<td>1.9</td>
<td>2.6</td>
<td>7.8</td>
</tr>
<tr>
<td>clustering</td>
<td>1.3</td>
<td>1.9</td>
<td>3.3</td>
<td>5.3</td>
</tr>
<tr>
<td>pairwise-voting</td>
<td>2.0</td>
<td>2.4</td>
<td>3.6</td>
<td>4.7</td>
</tr>
<tr>
<td>regression</td>
<td>1.3</td>
<td>2.0</td>
<td>3.6</td>
<td>7.8</td>
</tr>
<tr>
<td>SB</td>
<td>1.0</td>
<td>1.7</td>
<td>2.9</td>
<td>7.2</td>
</tr>
</tbody>
</table>

| SAT11-RAND (VBS: 65.7) | | | | |
| PNN | 6.5 | 9.3 | 10.7 | 60.2 |
| DNN | 3.8 | 11.0 | 42.2 | 60.5 |
| clustering | 6.1 | 9.5 | 32.3 | 42.7 |
| 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 |

<table>
<thead>
<tr>
<th>(k)</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAT11-INDU (VBS: 15.4)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PNN</td>
<td>1.6</td>
<td>2.3</td>
<td>3.9</td>
<td>5.7</td>
</tr>
<tr>
<td>DNN</td>
<td>2.0</td>
<td>2.4</td>
<td>3.4</td>
<td>5.0</td>
</tr>
<tr>
<td>clustering</td>
<td>1.3</td>
<td>2.1</td>
<td>2.8</td>
<td>4.6</td>
</tr>
<tr>
<td>pairwise-voting</td>
<td>2.4</td>
<td>3.0</td>
<td>3.8</td>
<td>5.4</td>
</tr>
<tr>
<td>regression</td>
<td>1.9</td>
<td>2.5</td>
<td>3.5</td>
<td>6.3</td>
</tr>
<tr>
<td>SB</td>
<td>1.0</td>
<td>1.5</td>
<td>2.5</td>
<td>4.8</td>
</tr>
</tbody>
</table>

| SAT12-RAND (VBS: 34.7) | | | | |
| PNN | 2.0 | 2.8 | 4.9 | 7.5 |
| DNN | 3.7 | 6.2 | 11.4 | 14.3 |
| clustering | 1.8 | 2.3 | 3.3 | 4.6 |
| 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-RAND (VBS: 12.1) | | | | |
| PNN | 1.2 | 2.1 | 4.8 | 7.3 |
| DNN | 0.8 | 1.5 | 4.7 | 8.6 |
| clustering | 1.3 | 1.7 | 2.7 | 4.9 |
| 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 8.2: Speedup on PAR10 (wallclock) in comparison to SB with one processing unit \((k = 1)\). 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 \(\alpha = 0.05\)).
approaches. On the industrial and crafted scenarios, the pairwise-voting approaches from SATzilla [83] and DNN performed consistently well. Surprisingly, in contrast, on the random instances pairwise-voting had amongst the worst performance, but simply selecting statically the $n$-best performing solvers ($SB$) from the training instances performed well.\footnote{We note that the solvers in SAT*-RAND are randomized but the scenarios in ASlib do not reflect this; thus, these performance estimates are probably optimistic [19].}

\section{Other Parallel Portfolio Selection Approaches}

A relevant medal-winning system in the SAT Competition 2013 was the parallel portfolio selector CSHCpar [63], which is based on the algorithm selection of cost-sensitive hierarchical clustering (CSHC [62]). It always runs, independently and in parallel, the parallel SAT solver Plingeling with 4 threads, the sequential SAT solver CCASat, and three solvers that are selected on a per-instance basis. These per-instance solvers are selected by three models that are trained on application, crafted and random SAT instances, respectively. While CSHCpar is particularly designed for the SAT Competition with its 8 available cores, it does not provide an obvious way of adjusting the number of processing units and does not support use cases without explicitly identified, distinct instance classes (such as industrial, crafted and random).

The extension of 3S [49] to parallel portfolio selection, dubbed 3Spar [61], 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 presolving 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 exponentially 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.

EISAC [60] clusters the training instances in the feature space and provides a method for selecting parallel portfolios for each cluster of instances by searching over all $\binom{|\mathcal{A}|}{k}$ combinations of $|\mathcal{A}|$ algorithms and $k$ processing units. As this approach quickly becomes infeasible for growing $|\mathcal{A}|$ and $k$, Yuri Malitsky, author of EISAC, recommends to limit its use to at most 4 processing units (README file\footnote{https://sites.google.com/site/yurimalitsky/downloads}).

The aspeed system [36] 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 selected on a per-instance basis. For this reason, aspeed is not directly applicable to per-instance selection of parallel portfolios; however, our approach uses it to effectively compute parallel presolving schedules.

RSR-WG [84] combines a case-based-reasoning approach from CP-Hydra [67] with greedy construction of parallel portfolio schedules via GASS [75] 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 are also different possible extensions of algorithm schedules to per-instance schedules [49, 55], which aim to select an algorithm schedule on an instance-by-instance basis.

8.3 Automatic Construction of Parallel Portfolios from Parameterized Solvers

So far, we have assumed that we are given a set of solvers for a given problem, such as SAT, and that for a problem instance to be solved, we select a subset of these solvers to be run as a parallel portfolio. Now, we focus on a different approach for constructing parallel portfolios, starting from the observation that solvers for computationally challenging problems typically expose parameters, whose settings can have a very substantial impact on performance. For SAT solvers, these parameters control key aspects of the underlying search process (e.g., the variable selection mechanism, clause deletion policy and restart frequency); by choosing their values specifically for a given instance set, performance can often be increased by orders of magnitude over that obtained using default parameter settings [40, 45, 43, 23, 47]. The task of automatically determining parameter settings such that performance on a given instance set is optimised is known as algorithm configuration [45].

Based on the success of algorithm selection and configuration systems, we conjecture that there is neither a single best algorithm nor a single best parameter configuration for all possible instances. Therefore, by combining complementary parameter configurations into a parallel portfolio solver more robust behaviour can be achieved on a large variety of instances. In fact, many parallel SAT solvers already exploit this idea by using different parameter settings in different threads, e.g., ManySAT [28], clasp [25] or Plingeling [16]. However, these portfolios are hand-designed, which requires a tedious, error-prone and time-consuming manual parameter optimization process.

Combining the ideas of parallel portfolios of different parameter settings and automatic algorithm configuration leads to our approach of automatic construction
Algorithm Configuration Scenario

**Configuration Space** \( \Theta \)

**Algorithm Configurator**

Call \( A(\theta) \) on \( \pi \in I \)

**Target Algorithm** \( A \)

**Instances** \( I \)

Solves

Return performance \( m(\theta, \pi) \)

Fig. 8.3: Algorithm configuration workflow.

**of parallel portfolios** (ACPP). In its simplest form, the only required input is a single parameterized sequential SAT solver. Using an automatic algorithm configuration procedure, we determine a set of complementary parameter configurations and run them in parallel to obtain a robust and efficient parallel portfolio solver.

### 8.3.1 Problem Statement

The traditional algorithm configuration task consists of determining a parameter configuration with good performance on a set of instances from the configuration space of a given algorithm. Formally, this gives rise to the following problem.

**Definition 3 (Algorithm Configuration; AC).** An instance of the *algorithm configuration problem* is a 6-tuple \((A, \Theta, I, D, \kappa, m)\) where:

- \( A \) is a parameterized target algorithm,
- \( \Theta \) is the parameter configuration space of \( A \),
- \( I \) is a set of instances of a problem,
- \( D \) is a probability distribution over \( I \),
- \( \kappa \in \mathbb{R}^+ \) is a cutoff time, after which each run of \( A \) will be terminated if still running, and
- \( m : \Theta \times I \to \mathbb{R} \) quantifies the performance of configuration \( \theta \in \Theta \) on instance \( \pi \in I \) w.r.t. a given cutoff time \( \kappa \).

The objective is to determine a configuration \( \theta^* \in \Theta \) that achieves near-optimal performance across instances \( \pi \in I \) drawn from \( D \). As in the previous section, we consider a performance measure based on running time, which we aim to minimise; therefore, we aim to determine \( \theta^* \in \arg\min_{\theta \in \Theta} E_{\pi \sim D}[m(\theta, \pi)] \).

The workflow of algorithm configuration is visualized in Figure 8.3. An AC procedure iteratively determines algorithm runs to be performed by selecting appropriate pairs of configurations \((\theta \text{ and instances } \pi)\), executing the corresponding algorithm runs, and observing their performance measurements. Finally, after a given configuration budget—usually a given amount of computing time—has been exhausted, the
AC procedure returns its *incumbent* parameter configuration $\hat{\theta}$ at that time, i.e., its best known configuration.

For several reasons, AC is a challenging problem. First, the only mode of interaction between the AC procedure and the target algorithm $A$ is to run $A$ on some instances and observe its performance. Thus, $A$ is treated as a black box, and no specific knowledge about its inner workings can be directly exploited. As a result, automatic algorithm configuration procedures are broadly applicable, but have to work effectively with very limited information.

Second, the configuration space of many solvers is large and complex. These spaces typically involve parameters of different types, such as categorical and continuous parameters, and often exhibit structure in the form of conditional dependencies between parameters or forbidden parameter configurations. For example, the configuration space of *Lingeling* [15] in the configurable SAT solver challenge [47] had 241 parameters giving rise to $10^{974}$ possible parameter configurations.

Third, particularly when solving *NP*-hard problems (such as SAT), even a single evaluation of a target algorithm configuration on one problem instance can be costly in terms of running time. Therefore, AC procedures typically can only evaluate a small number of pairs $(\theta, \pi)$ in a high-dimensional search space—often, only hundreds (and sometimes, thousands) of evaluations are possible even within typical configuration budgets of 12–48 hours of computing time.

Nevertheless, in recent years, algorithm configuration systems have been able to substantially improve the performance of SAT solvers on many types of SAT instances [47]. Well-known algorithm configuration systems include (i) *ParamILS* [46, 45], which performs iterated local search in the configuration space; (ii) *GGA* [5, 4], which is based on a genetic algorithm; (iii) *irace* [59], which uses F-race [9] for racing parameter configurations against each other; and (iv) *SMAC* [43, 42], which makes use of an extension of Bayesian optimization [18] to handle potentially heterogeneous sets of problem instances. For some more details on the mechanisms used in these configuration procedures, we refer the interested reader to the report on the Configurable SAT Solver Challenge [47].

Our extension of algorithm configuration to parallel problem solving is called parallel portfolio construction. The task consists of finding a parallel portfolio $\theta_{1:k}$ of $k$ parameter configurations whose performance (e.g., wallclock time) is evaluated by the first component of $\theta_{1:k}$ that solves a given instance $\pi$. We formally define the problem as follows:

**Definition 4 (Parallel Portfolio Construction).** An instance of the *parallel portfolio construction problem* is a 7-tuple $(A, \Theta, I, \mathcal{D}, \kappa, m, k)$ where:

- $A$ is a parameterized target algorithm,
- $\Theta$ is the parameter configuration space of $A$,
- $I$ is a set of problem instances,
- $\mathcal{D}$ is a probability distribution over $I$,
- $\kappa \in \mathbb{R}^+$ is a cutoff time, after which each run of $A$ will be terminated if still running,
- $k$ is the number of available processing units, and
• \( m : \Theta^l \times I \rightarrow \mathbb{R} \) quantifies the performance of a portfolio \( \theta_{1,l} \in \Theta^l \) on instance \( \pi \in I \) w.r.t. a given cutoff time \( \kappa \) for any given portfolio size \( l \).

The objective is to construct a parallel portfolio \( \theta^*_1 \in \Theta^k \) from the \( k \)-fold configuration space \( \Theta^k \) that optimizes the expected performance across instances \( \pi \in I \) drawn from \( \mathcal{P} \); in the case of minimising a performance metric based on running time, as considered here, we aim to find

\[
\theta^*_1 \in \arg\min_{\theta_{1,k} \in \Theta^k} \mathbb{E}_{\pi \sim \mathcal{P}} [m(\theta_{1,k}, \pi)].
\]

If the configurations in the portfolio \( \theta^*_1 \) are run independently, without any interaction (e.g., in the form of clause sharing), and the overhead from running configurations in parallel is negligible, this is identical to identifying

\[
\theta^*_1 \in \arg\min_{\theta_{1,k} \in \Theta^k} \mathbb{E}_{\pi \sim \mathcal{P}} \left[ \min_{i \in \{1 \ldots k\}} m(\theta_i, \pi) \right].
\]

Compared to algorithm configuration, parallel portfolio construction involves even larger configuration spaces. For a portfolio with \( k \) parameter configurations, an algorithm configuration procedure has to search in a space induced by \( k \) times the number of parameters of \( A \), and therefore of total size \( |\Theta|^k \).

### 8.3.2 Automatic Construction of Parallel Portfolios (ACPP)

In the following, we explain two methods to address automatic construction of parallel portfolio (ACPP). Since this problem is an extension of the algorithm configuration problem, we consequently build upon an existing algorithm configuration procedure and extend it for ACPP.

#### 8.3.2.1 Multiplying Configuration Space: **GLOBAL**.

Algorithm 1 shows the most straightforward method for using algorithm configuration for ACPP. The **GLOBAL** approach consists of using algorithm configuration procedure \( AC \) on \( \Theta^k \), the \( k \)-fold Cartesian product of the configuration space \( \Theta \).\(^7\)

The remaining parts of the procedure follow the standard approach for algorithm configuration: instead of running \( AC \) only once with configuration budget \( t \), we perform \( n \) runs of \( AC \) with a budget of \( t/n \) each (Lines 1 and 2). Each of these \( AC \) runs ultimately produces one portfolio of size \( k \). Performing these \( n \) runs in parallel reduces the wallclock time required for the overall configuration process by leveraging parallel computation. Of the \( n \) portfolios obtained from these independent

\(^7\) The product of two configuration spaces \( X \) and \( Y \) is defined as \( \{x||y \mid x \in X, y \in Y\} \), with \( x||y \) denoting the concatenation (rather than nesting) of tuples.
Algorithm 1: Portfolio Configuration Procedure GLOBAL

Input: parametric algorithm with configuration space $\Theta$; desired number $k$ of component solvers; instance set $I$; performance metric $m$; configuration procedure $AC$; number $n$ of independent configurator runs; total configuration time $t$

Output: parallel portfolio solver with portfolio $\hat{\theta}_{1:k}$

1. for $j := 1 \ldots n$ do
2. obtain portfolio $\theta_{1:k}^{(j)}$ by running $AC$ for time $t/n$ on configuration space $\Theta^k$ on $I$ using $m$
3. choose $\hat{\theta}_{1:k} \in \arg\min_{\theta_{1:k} \in \Theta^k} \sum_{\pi \in I} m(\theta_{1:k}, \pi)$ that achieved best performance on $I$ according to $m$
4. return $\hat{\theta}_{1:k}$

Algorithm 2: Portfolio Configuration Procedure PARHYDRA

Input: parametric algorithm with configuration space $\Theta$; desired number $k$ of component solvers; instance set $I$; performance metric $m$; configurator $AC$; number $n$ of independent configurator runs; total configuration time $t$

Output: parallel portfolio solver with portfolio $\hat{\theta}_{1:k}$

1. let $\theta_{\text{init}}$ be the default configuration in $\Theta$
2. for $i := 1 \ldots k$ do
3. for $j := 1 \ldots n$ do
4. obtain portfolio $\theta_{1:k}^{(j)} := \hat{\theta}_{1:i-1} || \theta^{(j)}$ by running $AC$ on configuration space $\{ \hat{\theta}_{1:i-1} \} \times \{ (\theta) \mid \theta \in \Theta \}$ and initial incumbent $\hat{\theta}_{1:i-1} || \theta_{\text{init}}$ on $I$ using $m$ for time $t/(k \cdot n)$
5. let $\hat{\theta}_{1:i} \in \arg\min_{\theta_{1:k} \in \Theta^k} \sum_{\pi \in I} m(\theta_{1:k}, \pi)$ be the portfolio which achieved best performance on $I$ according to $m$
6. let $\theta_{\text{init}} \in \arg\min_{(\theta) \in \Theta} \sum_{\pi \in I} m(\hat{\theta}_{1:i} || (\theta), \pi)$ be the configuration that has the largest marginal contribution to $\hat{\theta}_{1:i}$
7. return $\hat{\theta}_{1:k}$

In principle, this method can find the best parallel portfolio, but the configuration space grows exponentially with portfolio size $k$ to a size of $|\Theta|^k$, which can become problematic even for small $k$.

8.3.2.2 Iterative Approach: PARHYDRA.

To avoid the complexity of GLOBAL, the iterative, greedy ACPP procedure outlined in Algorithm 2 can be used. Inspired by Hydra [79, 81], PARHYDRA determines one parameter configuration in each iteration and adds it to the final portfolio. The configuration to be added is determined such that it best complements the configurations that have previously been added to the portfolio.
In detail, our PARHYDRA approach runs for $k$ iterations (Line 2) to construct a portfolio with $k$ components. In each iteration, we fix one further parameter configuration of our final portfolio. As before, we perform $n$ AC runs in each PARHYDRA-iteration $i$ (Lines 3–5). The configuration space consists of the Cartesian product of the (fixed) portfolio $\hat{\theta}_{i-1}$ constructed in the previous $i-1$ iterations with the full configuration space $\Theta$. Each AC run effectively determines a configuration to be added to the portfolio such that the overall portfolio performance is optimised. As configuration budget, each AC run is allocated $t/(k \cdot n)$, where $t$ is the overall budget.

An extension in comparison to Hydra [79, 81] is that the initial parameter configuration $\theta_{initial}$ for the search is adapted in each iteration. For the first iteration, we simply use the default parameter configuration (Line 1)—if no default parameter configuration is known, we could simply use the mean parameter value from the parameter domain ranges or randomly sample an initial configuration. At the end of each iteration, we determine which returned parameter configuration $\theta^{(j)}$ from the last $n$ AC runs ($j \in \{1, \ldots, n\}$) would improve the current portfolio $\hat{\theta}_{i-1}$ the most (Line 6). This configuration is used to initialize the search in the next iteration. This avoids discarding all of each iteration’s unselected configurations, keeping at least one to guide the search in future iterations.\footnote{Note that this strategy assumes multiple configuration runs per iteration (e.g., $n$ independent runs of a sequential algorithm configuration procedure) and would not directly be applicable if we used a parallel algorithm configuration procedure [44] that only returns a single configuration. Whether one can gain more from using parallel algorithm configuration or from having a good initialization strategy is an open question.}

### 8.3.2.3 Comparing Global and PARHYDRA

On the one hand, in comparison to Global, PARHYDRA has the advantage that it only needs to search in the original space $\Theta$ in each iteration (in contrast to the exponentially larger $|\Theta|^k$). On the other hand, PARHYDRA has $k$ times less time per iteration, and the configuration tasks may get harder in each iteration because less configurations will be complementary for growing portfolio size. It is also not guaranteed that PARHYDRA will find the optimal portfolio because of its greedy nature; for example, if our instance set $I$ would consist of two homogeneous subsets $I_1 \cup I_2 = I$, Global can in principle directly find a well-performing configuration for each of the two subsets. In contrast, PARHYDRA will optimize the configuration on the entire instance set $I$ in the first iteration and can only focus on one of the two subsets in the second iteration. Therefore, PARHYDRA could return suboptimal solutions.

We note, however, that this suboptimality is bounded, since PARHYDRA’s portfolio performance is a submodular set function (the effect of adding a further parameter configuration to a smaller portfolio of an early iteration will be larger than adding it to a larger portfolio of a later iteration). This property can be exploited to derive bounds for the performance of Hydra-like approaches [73], such as PARHYDRA.
Selection and Configuration of Parallel Portfolios

Table 8.3: Running time statistics on the test set from application and hard combinatorial SAT instances achieved by single-processor (SP) and 8-processor (MP8) versions. DEFAULT-MP(8) was Lingeling in case of Lingeling and clasp \texttt{\small -t 8} for clasp where we show results with (+CS) and without clause sharing (-CS). The performance of a solver is shown in boldface if it was not significantly different from the best performance, and is marked with an asterisk (*) if it was not significantly worse than DEFAULT-MP(8)+CS (according to a permutation test with 100 000 permutations and significance level $\alpha = 0.05$). The best ACPP portfolio on the training set is marked with a dagger (†).

<table>
<thead>
<tr>
<th>Solver Set</th>
<th>#TOs</th>
<th>PAR10</th>
<th>PAR1</th>
<th>#TOs</th>
<th>PAR10</th>
<th>PAR1</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEFAULT-SP</td>
<td>72</td>
<td>2317</td>
<td>373</td>
<td>137</td>
<td>4180</td>
<td>481</td>
</tr>
<tr>
<td>CONFIGURED-SP</td>
<td>68</td>
<td>2204</td>
<td>368</td>
<td>140</td>
<td>4253</td>
<td>473</td>
</tr>
<tr>
<td>DEFAULT-MP(8)-CS</td>
<td>64</td>
<td>2073</td>
<td>345</td>
<td>96</td>
<td>2950</td>
<td>385</td>
</tr>
<tr>
<td>DEFAULT-MP(8)+CS</td>
<td>53*</td>
<td>1730*</td>
<td>399*</td>
<td>90*</td>
<td>2763*</td>
<td>333*</td>
</tr>
<tr>
<td>GLOBAL-MP(8)</td>
<td>52*</td>
<td>1702*</td>
<td>398*</td>
<td>98</td>
<td>3011</td>
<td>365</td>
</tr>
<tr>
<td>PARHYDRA-MP(8)</td>
<td>55†</td>
<td>1788†</td>
<td>303†</td>
<td>96†</td>
<td>2945†</td>
<td>353†</td>
</tr>
</tbody>
</table>

8.3.2.4 Empirical Study on SAT 2012 Challenge

We studied the effectiveness of our two proposed ACPP methods, i.e., GLOBAL and PARHYDRA, on two award-winning solvers from the 2012 SAT Challenge: Lingeling [14] and clasp [25]. To this end, we compared the default sequential solver settings (DEFAULT-SP), the configured sequential solvers (CONFIGURED-SP), the default parallel counterparts of both solvers (i.e., Lingeling for Lingeling) without (DEFAULT-MP(8)-CS) and with clause sharing (DEFAULT-MP(8)+CS) and finally, with GLOBAL and PARHYDRA. As instance sets, we used the instances from the application track and hard combinatorial track of the 2012 SAT Challenge for Lingeling and clasp, respectively. Both instance sets were split into a training set for configuration and test set to obtain an unbiased performance estimate. The parallel solvers used eight processing units ($k = 8$). We used SMAC [43, 42], a state-of-the-art algorithm configuration procedure, to minimise penalized average running time, and every configuration approach (i.e., CONFIGURED-SP, GLOBAL and PARHYDRA) was given the same configuration budget $t$.

Table 8.3 summarizes our results. First of all, we note that algorithm configuration on heterogeneous instance sets, such as the instance sets from SAT competitions and challenges, is challenging, because various instances are solved best by potentially very different configurations, which can pull the search process into different directions. Therefore, the configured sequential version (CONFIGURED-SP) of Lingeling performed only slightly better than the default, and the performance of clasp even slightly deteriorated due to overtuning [45], i.e., it showed a performance improvement on the training instances which did not generalize to test instances. The default
parallel versions of *Lingeling* and *clasp* performed consistently better than their sequential counterparts. Enabling clause sharing (CS) for both solvers improved their performance even further.

Our automatically constructed parallel portfolio solvers performed well in comparison to the manually hand-crafted parallel solvers. To verify whether the observed performance differences were statistically significant, we used a permutation test to compare the best-performing approach against all others. This analysis revealed that the portfolios manually built by human experts did not perform significantly better than those automatically constructed using PARHYDRA. We emphasize that our ACPP portfolios do not use any clause sharing strategies, and the configuration process was initialized with the parameter setting of DEFAULT-SP. Therefore, our methods had no hint how to construct a parallel solver. Nevertheless, our automatic approach produced parallel solvers performing as well as those manually designed by experts within a few days of computing time on a small cluster.

### 8.3.2.5 ACPP with Multiple Solvers

Even though it is appealing to automatically generate a parallel solver from a sequential solver, our ACPP methods are not limited to a single solver as an input. Using more than one solver often increases the opportunity for leveraging performance complementarities, since SAT solvers often implement complementary strategies [82]. To construct a parallel portfolio solver from a set of parameterized solvers as an input using our ACPP methods, we only need to adapt our configuration space \( \Theta \). Following the idea of Thornton et al [76], we introduce a top-level parameter that indicates which solver to use. The parameters of the individual solvers are then conditionally dependent on this new selector parameter, leading to structured configuration spaces as illustrated in Figure 8.4.
### 8.3.3 Automatic Construction of Parallel Portfolios from Parallel Parameterized Solvers

The ACPP methods presented thus far always assumed that one or more sequential SAT solvers are given. However, over the course of the last decade, many parallel SAT solvers have been developed (e.g., [28, 72, 25, 8, 16]). On the one hand, these solvers often expose performance-critical parameters (e.g., controlling clause sharing); on the other hand, these solvers can also be used in our ACPP methods as components to include in a parallel portfolio. In the following, we discuss both approaches to further improve the performance of parallel SAT solvers by using algorithm configuration.

#### 8.3.3.1 Configuration of Clause Sharing

Clause sharing is an important strategy to reduce redundant work in parallel SAT solving and hence, to improve the performance of parallel SAT solvers. However, clause sharing also has many open implementation options, e.g., communication topology, how often to share learned clauses, which learned clauses to share, which clauses to integrate in the clause database, etc. The best configuration of these parameters can have crucial impact on performance and depends on the nature of the instances to be solved. Therefore, we can use algorithm configuration to optimise the settings of the parameters that control clause sharing.

The results shown in Table 8.4 have been obtained following the same experimental setup as already described in Section 8.3.2.4 for clasp on hard-combinatorial instances. As a starting point, we used the parallel clasp portfolio found by PARHYDRA—without using any clause sharing. Adding the default clause sharing policy on top of the PARHYDRA portfolio lead to solving 6 more instances, which is equivalent to the performance of DEFAULT-MP(8)+CS (see Table 8.3). However, clasp allows adjustment of the clause sharing distribution and integration policies. Using automatic algorithm configuration to optimize these policies, the clasp portfolio was able to solve two additional instances. We note that it is suboptimal to first configure a parallel portfolio without any communication between component solvers, and to then add clause sharing to the portfolio thus obtained. In principle, configuring the portfolio and the clause sharing mechanism jointly should result in

<table>
<thead>
<tr>
<th>clasp variant</th>
<th>#TOs</th>
<th>PAR10</th>
<th>PAR1</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Clause Sharing</td>
<td>96</td>
<td>2945</td>
<td>353</td>
</tr>
<tr>
<td>Default Clause Sharing</td>
<td>90</td>
<td>2777</td>
<td>347</td>
</tr>
<tr>
<td>Configured Clause Sharing</td>
<td>88</td>
<td>2722</td>
<td>346</td>
</tr>
</tbody>
</table>

Table 8.4: Comparison of different clause sharing strategies on top of our constructed PARHYDRA-MP(8) portfolio with clasp on the test instances of the hard combinatorial set.
Algorithm 3: Portfolio Configuration Procedure PARHYDRA

Input: set of parametric solvers $A \in \mathcal{A}$ with configuration spaces $\Theta_A$; desired number $k$ of component solvers; number $b$ of component solvers simultaneously configured per iteration; instance set $I$; performance metric $m$; configurator $AC$; number $n$ of independent configurator runs; total configuration time $t$

Output: parallel portfolio solver with portfolio $\hat{\theta}_{1:k}$

1. $i := 1$
2. let $\theta_{init}$ be a portfolio with $b$ times the default configuration in $\Theta$ of a default solver $A \in \mathcal{A}$.
3. while $i < k$ do
   4. $i' := i + b - 1$
   5. for $j := 1$ to $n$
       6. obtain portfolio $\theta_{1:i'}^{(j)} := \hat{\theta}_{1:i-1} || \theta_{i'}^{(j)}$ by running $AC$ for time $t \cdot b/(k \cdot n)$ on configuration space $\{\hat{\theta}_{1:i-1}\} \times (\prod_{A \in \mathcal{A}} (\Theta_A | \theta \in \Theta_A))$ and initial incumbent $\hat{\theta}_{1:i-1} || \theta_{init}$ on $I$ using $m$
   7. let $\hat{\theta}_{1:i'} \in \arg\min_{\theta_{1:i'}^{(j)} | j \in \{1...n\}} \sum_{\pi \in I} m(\theta_{1:i'}^{(j)}, \pi)$ be the portfolio that achieved best performance on $I$ according to $m$
   8. let $\theta_{init} \in \arg\min_{\theta_{1:i'}^{(j)} | j \in \{1...n\}} \sum_{\pi \in I} m(\hat{\theta}_{1:i'} || \theta_{init}, \pi)$ be the portfolio that has the largest marginal contribution to $\hat{\theta}_{1:i'}$
   9. $i := i + b$
10. return $\hat{\theta}_{1:k}$

better performance; therefore, the results presented here only give a lower bound on what can be achieved.

8.3.3.2 Portfolio Construction using Parallel Solvers

Another way of using existing parallel solvers is to allow them to be part of an automated parallel portfolio solver. Similar to $ppfolio$, we could run a parallel solver, such as $Plingeling$, in some execution threads and some sequential solvers on others. To do this, we can use the trick of adding a top-level parameter to decide between different sequential and parallel solvers (see Section 8.3.2.5). If a parallel solver gets selected $l$ times by top-level parameters of each portfolio component, we merge these components into one call of the parallel solver with $l$ threads. By using this approach, we can directly apply the $GLOBAL$ methods to determine a well-performing automatically constructed parallel portfolio including other parallel SAT solvers.

Unfortunately, PARHYDRA cannot be directly applied to this setting because its reliance on a greedy algorithm makes it suboptimal. For example, if the portfolio $\theta_{1:i}$ already includes a configuration of sequential solver $A_s$ in iteration $i$, PARHYDRA will never add the parallel counterpart $A_p$ of $A_s$, because in each iteration, PARHYDRA
can only pick $A_p$ for one thread, which is outperformed by $A_s$. As a concrete example, let us consider the highly parameterized sequential solver Lingeling and its non-parameterized parallel counterpart, Plingeling. After a configuration of Lingeling was added to the portfolio, PARHYDRA never added Plingeling with a single thread in later iterations, because the optimized Lingeling outperformed Plingeling.

To permit a trade-off between the problems of GLOBAL (exponential increase of the search space) and PARHYDRA (suboptimality in portfolio construction), we propose an extension of PARHYDRA, called PARHYDRA$_b$, which adds not just one, but $b$ configurations to the portfolio in each iteration. Algorithm 3 shows an outline of the PARHYDRA$_b$ approach. The main idea is the same as of PARHYDRA, but we use an additional variable $i'$ to keep track of the parameter configurations added in each iteration. For example, if we have already fixed a portfolio $\theta_{1:i}$ with 4 components and want to add two configurations ($b = 2$) per iteration, we are in iteration $i = 5$, in which we will determine the fifth and sixth configuration $\theta_{i=5:6}$ (Lines 4 and 6) to be added to $\theta_{1:i-1}=4$. Furthermore, the starting point for the configuration process in the following iteration is now obtained by adding a portfolio of size $i' - i + 1 = b$ (Line 8) to $\theta_{1:i}$ from the previous iteration. Other than that, PARHYDRA$_b$ is the same as PARHYDRA.

### 8.3.3.3 Empirical Study on 2012 SAT Challenge

Again, we demonstrate the effect of our ACPP methods using parallel SAT solvers on the industrial instance set of the 2012 SAT Challenge. The winning parallel solver in this challenge was pfolioUZK [78], a hand-designed portfolio consisting of sequential and parallel portfolios. In particular, pfolioUZK uses satUZK, glucose, contrasat and Plingeling with 4 threads, leaving one of the 8 available CPU cores unused; however, the set of solvers considered during the design of pfolioUZK involved additional solvers that do not appear in the final design. To fairly compare with this manually constructed portfolio, we used the same underlying set of solvers as the starting point for our ACPP methods:

- contrasat [26]: 15 parameters;
- glucose 2.0 [7]: 10 parameters for satelite preprocessing and 6 for glucose;
- Lingeling 587 [14]: 117 parameters;
- Plingeling 587 [14]: 0 parameters;
- march 2009 [33]: 0 parameters;
- MPhaseSAT _M_ [21]: 0 parameters;
- satUZK [27]: 1 parameter;
- sparrow2011 [77]: 0 parameters$^{10}$; and

---

$^9$ In principle, one could imagine grouping $A_s$ and $A_p$ to effectively see them as the same solver, allowing PARHYDRA to add $A_p$ and join this with $A_s$ into a 2-thread version of $A_p$. However, this kind of grouping is not supported by PARHYDRA.

$^{10}$ Although sparrow2011 should be parameterized [77], the source code and binary provided with pfolioUZK does not expose any parameters.
Table 8.5: Comparison of parallel solvers with 8 processors on the test set of application. The performance of a solver is shown in boldface if its performance was at least as good as that of any other solver, up to statistically insignificant differences (according to a permutation test with 100 000 permutations at significance level $\alpha = 0.05$).

- TNM [54]: 0 parameters.

We note that of these, Plingeling is the only parallel SAT solver and the only one to make use of clause sharing.

Table 8.5 shows the performance of glucose-2.1 (which won the main application SAT+UNSAT track of the 2012 SAT Challenge), Plingeling (ala) with clause sharing, pfolioUZK (which won the parallel application SAT+UNSAT track) and our ACPP methods. Surprisingly, on 8 cores, Plingeling performed only slightly better than glucose. However, pfolioUZK solved 18 instances more than Plingeling within the cutoff time used in the competition. By applying GLOBAL (i.e., PARHYDRA$_b$ with $b = k = 8$), we obtained a parallel portfolio performing as well as pfolioUZK. PARHYDRA$_b$ with $b = 4$ performed statistically better than pfolioUZK by solving 6 instances more.

Looking at the performance achieved by PARHYDRA$_b$ for different values of $b$ reveals that $b$ is an important parameter of our method. One might be concerned that PARHYDRA$_4$-MP(8) performed as well as it did as a result of over-tuning on $b$. We note, however, that PARHYDRA$_4$-MP(8) also performed best on the training instances used for configuration, which are different from the test instance results shown in Table 8.5.

8.4 Conclusions and Future Work

In this chapter, we presented two generic approaches for automatically generating parallel portfolio solvers for computationally challenging problems from one or more sequential solvers. While our focus was on SAT, the techniques we discussed are in
no way specific to this particularly well-studied constraint programming problem, and can be expected to give rise to similarly strong performance when applied to a broad range of CP problems, and, indeed, to many other NP-hard problems. We note that there are three fundamental assumptions that need to be satisfied in order for these generic parallelisation methods to scale well with the number of processing units $k$.

1. **Performance complementarity**: within a given set $\mathcal{A}$ of solvers that are available (as in algorithm selection) or within the parameter space of a single solver (as in algorithm configuration), there is sufficient performance complementarity. In algorithm selection with deterministic algorithms, algorithm selectors cannot perform better than the virtual best solver (VBS) of the given algorithm portfolio. Therefore, a parallel portfolio selector can scale at most to a number of processing units that equals the number of candidate solvers in $\mathcal{A}$. Unfortunately, this upper bound will usually not be attained, because in most sets $\mathcal{A}$, some solvers will have little or no contribution to the virtual best solver [82].

In parallel portfolio configuration, the given parameter space $\Theta$ is often infinite; still, in our experiments, little or no performance improvement was obtained beyond a modest number of portfolio components (e.g., using PARHYDRA, the performance of our automatically constructed parallel portfolio based on Lingeling improved only for the first 4 portfolio components – for details, see [58]). This could indicate that our current approaches are too weak to find better and larger portfolios (since the complexity of the search problems increases with the size of the portfolio), or that such portfolios simply do not exist for the instance sets we considered, and that the smaller portfolios we found basically exhaust the complementarity of the parameter space. Which of these two explanations holds is an interesting subject for future research.

2. **Heterogeneity of instances**: the given instance set $I$ is sufficiently heterogeneous given a set of solvers or parameter configurations. If the instance set is perfectly homogeneous, a single solver or configuration is dominant on all instances, and a parallel portfolio (without communication between component solvers) cannot perform better. In contrast, if each instance in $I$ requires a different solver or configuration to be solved most effectively, our generic parallel portfolio construction methods could in principle scale to a number of processing units equal to the size of the instance set. Therefore, in practice, the performance potential of these approaches depends on characteristics of the set or distribution of problem instances of interest in a given application context—the more diverse that set, the larger the potential for large speed-ups due to parallelisation. How to assess the heterogeneity of an instance set in an effective yet computationally cheap way is an open problem.

3. **Minimal interference between runs**: when sequential solvers are run concurrently, there is only minimal impact on performance due to detrimental interference. If each solver runs on a separate system, this assumption can easily be guaranteed, and because neither of our approaches requires much communication between portfolio component solvers, this scenario is quite feasible.
However, since modern machines are equipped with multi-core CPUs, it is generally desirable to run more than one solver on a single machine, such that the component solvers share resources, such as RAM and CPU cache. Since solver performance can substantially depend on the available CPU cache [1], running several solvers on multiple CPU cores with shared cache can lead to significant slow-down due to cache contention.

The extent to which this happens depends on the characteristics of the execution environment and on the solvers in question. For example, in the experiments reported in Section 8.3, we observed that Lingeling suffered more from this effect on the larger industrial instances than clasp did on the smaller crafted instances. Furthermore, we have observed that Lingeling’s performance is less affected on newer CPUs with larger amounts of cache. Therefore, we believe that in the future, with the advent of CPUs with even more cache memory, this issue might become less critical.

There are many prominent avenues for future work on generic parallelisation techniques, and we see much promise in the combination of the two approaches discussed in this chapter. For example, one could run PARHYDRA to generate many complementary configurations of one or more parameterised solvers and then use parallel portfolio selection on those configurations to create a per-instance parallel portfolio selector for a given number of processing units. Another interesting extension is the automatic configuration of parallel portfolio selectors, analogously to AutoFolio [57]. Similarly, we see promise in the configuration of parallel algorithm schedules, similarly to Cedalion [73]. It could also be interesting to use an approach such as aspeed [36] to post-optimize an automatically-generated parallel portfolio into a parallel algorithm schedule.

We see substantial promise in exploring instance features specifically designed for parallel portfolio selection, e.g., probing features of parallel solvers, possibly related to the communication between solver components. Finally, it would be interesting to improve the construction of portfolios that include randomized parallel component solvers with clause sharing by estimating the potential risks and gains of adding such component solvers based on their running time distributions.

References

44. Hutter, F., Hoos, H., Leyton-Brown, K.: Parallel algorithm configuration. In: Hamadi and Schoenauer [29], pp. 55–70


71. Schneider, M., Hoos, H.: Quantifying homogeneity of instance sets for algorithm configuration. In: Hamadi and Schoenauer [29], pp. 190–204
75. Streeter, M., Golovin, D., Smith, S.: Combining multiple heuristics online. In: Holte and Howe [34], pp. 1197–1203